Dietmar Watzinger, BSc

YAGI Native für Lego-Mindstorms-Ev3

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

Masterstudium Software Engineering and Management

eingereicht an der

Technischen Universität Graz

Betreuer

Ass. Prof. Dipl. -Ing. Dr. techn. Gerald Steinbauer

Institut für Softwaretechnologie

Graz, August 16

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

_____ _____

Datum Unterschrift

Zusammenfassung

Die aktionsbasierte imperative Programmiersprache YAGI (ein Akronym für **Y**et **A**nother **G**olog **I**nterpreter) basiert auf dem Situationskalkül und IndiGolog. Im Gegensatz zu der verwandten Programmiersprache Golog, besteht der YAGI-Interpreter aus einer dreischichtigen Architektur, um eine klare Trennung von Syntax, Semantik und der Implementierung einzuhalten und um ihn auf verschiedene Zielsysteme einfach adaptieren zu können. YAGI ist besonders gut zur Programmierung von autonomen Systemen und Agenten geeignet. Damit Aktionen der Programmiersprache YAGI auf einem autonomen System ausgeführt werden können, muss jedoch zuvor ein entsprechendes System-Interface integriert werden. In dieser Arbeit wird daher eine System-Interface-Implementation für YAGI für den Lego-Mindstorms-Ev3 vorgestellt. Es werden die technischen Voraussetzungen erörtert, die generell notwendig sind um YAGI zu kompilieren und wie dies am Lego-Mindstorms-Ev3 möglich ist. Ebenso wird beschrieben, wie ein neues System-Interface als neues Zielsystem in die YAGI-Softwarearchitektur hinzufügte werden kann, welche Funktionen für den Lego-Mindstorms-Ev3 bereitgestellt wurden und wie diese auf die Peripherie zugreifen. Das Ergebnis dieser Masterarbeit ist eine Erweiterung des YAGI Systems, damit dieses Aktionen auf dem Lego-Mindstorms-Ev3 ausführen kann. Wünschenswert wäre, dass mit dieser Erweiterung YAGI auch in Schulklassen zum Einsatz kommt um den Schülern verschiedene Algorithmen und fortschrittlicher Programmierparadigmen beizubringen.

Abstract

The action-based imperative programming language YAGI (an acronym for **Y**et **A**nother **G**olog **I**nterpreter) is based on the theoretical foundations of situation calculus and IndiGolog. Contrary to the related programming language Golog, YAGI has an underlying 3-tier architecture for a clear separation of syntax and semantics and to be highly portable to a wide variety of different platforms. The programming language YAGI appears well suited for controlling autonomous robots and agents. In order to be able to perform YAGI actions on autonomous systems, an appropriate system interface must be created. In this thesis we present an implementation of a YAGI system interface for the Lego-Mindstorms-Ev3. Initially we discuss the technical requirements for YAGI in general and especially the setup for the Lego-Mindstorms-Ev3. Furthermore we explain how to implement a new system interface in YAGI, describe the function which are provided for the Lego-Mindstorms-Ev3 in detail and how this functions interact with the periphery. The result of this thesis is an extension of the YAGI interpreter to perform actions on the Lego-Mindstorms-Ev3. With this extension we hope to encourage teachers to use YAGI in school classes.

Danksagungen

An dieser Stelle würde ich gerne einige Menschen danken, welche mich bei dieser Arbeit unterstützt haben.

Zuerst möchte ich mich bei meinem Betreuer Prof. Gerald Steinbauer bedanken, welcher mir die Möglichkeit gegeben hat diese Masterarbeit zu verfassen und mich mit zahlreichen konstruktiven Gesprächen unterstützt hat.

Weiters bedanke ich mich auch bei Clemens Mühlbacher und Martin Kandlhofer für die Fachgespräche in schwierigen Zeiten.

Zum Schluss möchte ich noch meiner Familie und meinen Freunden für die moralische Unterstützung danken und vor allem Corinna Winkler, die mich immer wieder motiviert hat.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Zeichnungsverzeichnis

1 Einführung

1.1 Motivation

Der Einsatz von Robotern im Schulunterricht zur Lehre von Themen aus der Informatik hat im Laufe der letzten Jahre immer mehr an Bedeutung gewonnen. Gerade für Schüler ist es wichtig, das gelernte theoretische Wissen auch praktisch anzuwenden, wie (Kandlhofer und Steinbauer, 2015) und (Papert, 1993) schon festgestellt haben. Dadurch können viele Themen wie Suchalgorithmen, Baumstrukturen, etc. anhand von Beispielprogrammen am Roboter erklärt und durch Schüler nach programmiert werden. (Cliburn, 2006) hat festgestellt, dass sich Lego-Mindstorms-Roboter besonders gut eignen, um Schülern Algorithmen und Kontrollstrukturen zu lehren. Lego-Mindstorms-Roboter sind Lego-Roboterbaukasten, welche umgebaut und programmiert werden können. Ein Bild von dem Lego-Mindstorms-Ev3 ist in Abbildung 1 abgebildet.

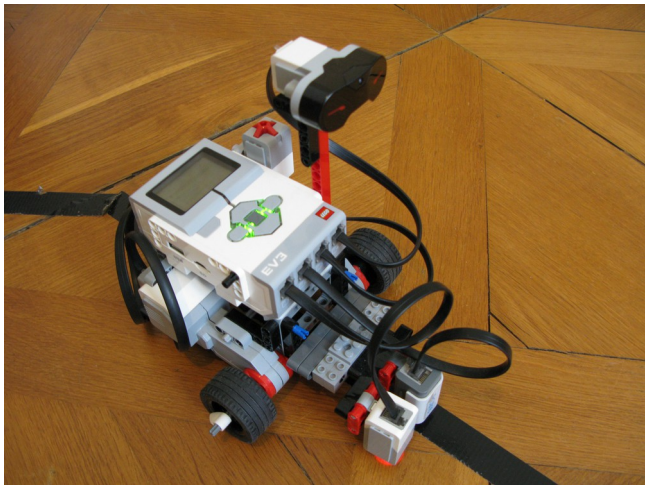

Abbildung 1: Der Lego-Mindstorms-Ev3 als Linienfolger

Lego stellt für alle Lego Mindstorms Produkte eine eigene Entwicklungsumgebung zur Verfügung. Diese Software basiert auf Funktionsblöcken, die aneinander gereiht werden können, um simple Programme zu erstellen. Die Entwicklungsumgebung eignet sich besonders für Kinder, da sie sehr einfach und intuitiv aufgebaut ist. Haben Schüler jedoch mehr Erfahrung im Programmieren, beziehungsweise will man komplexere Programme erstellen, empfiehlt es sich auf eine textuelle Programmiersprache, wie NXC, C, C++, Java, umzusteigen.

Bis zum heutigen Tag ist es nicht leicht, autonomen Robotern eine Aufgabe beizubringen, die für Menschen trivial ist. Ein sehr bekanntes Beispiel ist ein Roboter, der Pakete liefert. Nehmen wir an ein Roboter soll Pakete von einer Person zu einer anderen Person innerhalb eines Gebäudes liefern.

Dazu muss ein Roboter zuerst einmal fähig sein, sich autonom in einem Gebäude zu bewegen, oder Pakete aufzunehmen und abzulegen. Weiters muss der Roboter seine eigenen Aktionen überwachen, wie z.B. die Position im Raum zu ermitteln während sich der Roboter bewegt, oder zu überprüfen, ob das Aufnehmen eines Objektes erfolgreich war. Darüber hinaus muss der Roboter auf Veränderungen in seiner Umgebung reagieren, wie z.B. Anfragen von Personen, die ein Paket liefern wollen.

Um solche Aufgaben zu bewältigen, haben sich jedoch im Laufe der Jahre einige sehr gute Ansätze ergeben. Unter den erfolgreichsten Ansätzen sind Programmiersprachen, die auf dem Situationskalkül aufgebaut sind. Das Situationskalkül, eingeführt von (McCarthy, 1963) und (Reiter, 2001), ist ein Formalismus, der auf second-order Logik mit Gleichheit aufgebaut ist. Die bekannteste dieser Programmiersprachen ist Golog, konstruiert von (Levesque et al., 1994), welche eine formale Domain Spezifikation mit Elementen der imperativen Programmierung kombiniert. Dieses Modell wird auch als „action-based imperative programming“ bezeichnet. In den letzten Jahren gab es viele Erweiterungen von Golog, welche diese zu einer bedeutenden Programmiersprache machte. Da die meisten Golog Interpreter als eine Menge von Prolog-Klauseln implementiert sind, benötigt jede Plattform, die Golog Programme ausführen sollte, einen Prolog Interpreter. Zudem ist die Syntax von Golog sehr herausfordernd für Anfänger beziehungsweise manchmal auch für Experten.

1.2 Ausgangssituation

Die aktionsbasierte, imperative Programmiersprache YAGI (ein Akronym für **Y**et **A**nother **G**olog **I**nterpreter), entwickelt von (Maier et al., 2015), basiert auf dem Situationskalkül und der Programmiersprache IndiGolog (De Giacomo et. al., 2009). Sie ist bewusst entworfen worden, um Probleme von Prolog-basierten Implementationen von Golog zu vermeiden. Mehr Details zu den Eigenschaften von YAGI werden in Abschnitt 2.3 beschrieben. Der YAGI-Interpreter besteht aus einer 3-Schichtenarchitektur, um eine klare Trennung von Syntax, Semantik und der Implementierung einzuhalten und um ihn einfach auf verschiedene Zielsysteme adaptieren zu können. Die oberste Schicht, das Front-End, ist sowohl für das User Interface als auch für die syntaktische Verarbeitung des YAGI Codes verantwortlich. Das Back-End ist die mittlere Schicht, welches die Daten des Front-End auf einem semantischen Level verarbeitet, um die YAGI Domain-Theorie zu modifizieren und YAGI Programme auszuführen. Die letzte Schicht, das System-Interface, ist dafür verantwortlich, die Aktionen des Back-Ends auf dem Zielsystem auszuführen. In Abschnitt 4.1 wird auf die Details der YAGI Architektur eingegangen. Um den YAGI-Interpreter auf ein neues Zielsystem, z.B. einem Lego-Mindstorms-Ev3, zu portieren, müssen alle

Bibliotheken, die YAGI verwendet, auf der Plattform installiert werden. Weiters muss das System-Interface dementsprechend erweitert werden, um mögliche Aktionen auf den neuen Zielsystemen ausführen zu können. Dazu müssen Funktionen auf dem System-Interface bereitgestellt werden, um auf die Peripherie des neuen Systems zugreifen zu können.

1.3 Ziel der Arbeit

Ziel dieser Arbeit ist es die bestehende Programmiersprache YAGI so zu portieren, damit man sie auf dem Lego-Mindstorms-Ev3 verwenden kann. Dazu muss ein Weg gefunden werden, den YAGI-Interpreter auf dem Roboter auszuführen. Weiters muss die YAGI-Architektur um ein System-Interface erweitert werden, welches Befehle für den Lego-Mindstorms-Ev3 als Zielsystem enthält. Diese Befehle sollten möglichst atomar sein, um eine große Anzahl an Funktionen bereit zu stellen. Da YAGI später auch in Schulklassen zu Übungszwecken eingesetzt werden soll, ist ein weiteres Ziel, alle benötigten Funktionen wie das Ansteuern der Motoren, das Auslesen der Sensoren oder einen Linienfolger, zu implementieren. Weiters müssen diese Funktionen abstrahiert und in atomare Aktionen von YAGI abgebildet werden.

1.4 Beiträge dieser Masterarbeit

Die Beiträge dieser Masterarbeit sind wie folgt:

- Es wird ein Ansatz beschrieben, die es ermöglicht, den bestehenden YAGI-Interpreter auf dem Lego-Mindstorms-Ev3 auszuführen. Dazu wird ein Debian-Linux-basiertes Betriebssystem von einer Micro-SD-Karte aus gestartet und entsprechende Bibliotheken installiert.
- In dieser Arbeit wird eine Architektur präsentiert, mit deren Hilfe die Peripherie des Lego-Mindstorms-Ev3 angesteuert werden kann
- Weiters werden die Funktionen erklärt, welche für den Lego-Mindstorms-Ev3 implementiert wurden. Das YAGI System-Interface wird durch ein Ev3-System-Interface erweitert, welches Aktionen für den Lego-Mindstorms-Ev3 zur Verfügung stellt.
- Zu Letzt wurde darauf geachtet, alle Aktionen bereitzustellen, welche für einen Unterricht mit dem Lego-Mindstorms-Ev3 benötigt werden.

1.5 Organisation der Arbeit

Diese Arbeit ist folgendermaßen organisiert. Im nächsten Kapitel werden wichtige Themen vorgestellt, welche Voraussetzung für diese Arbeit waren. Im Kapitel 3 werden Arbeiten, welche mit dieser zusammenhängen diskutiert. Ebenso wird die Bedeutung von Robotern im Unterricht erörtert und welche Vor- und Nachteile das Arbeiten mit Robotern in Hinblick auf das Lernen der Schüler bewirkt. Das grundlegende Konzept von YAGI wird anschließend in Kapitel 4 präsentiert. Dabei wird näher auf die YAGI-Software-Architektur eingegangen und wie darin das System-Interface für den Lego-Mindstorms-Ev3 implementiert wurde. Sämtliche Methoden die für die Peripherie des Lego-Roboters geschrieben wurden, werden in Kapitel 5 genauer erklärt. Zum Schluss werden in Kapitel 7 die erreichten Ergebnisse diskutiert und Ideen für zukünftige Erweiterungen vorgestellt.

2 Voraussetzung

Dieses Kapitel gibt eine Einführung in die Themen, die benötigt wurden, um die Arbeit umsetzen zu können. In Abschnitt 2.1 werden technische Details des Lego-Mindstorms-Ev3 beschrieben und welche Peripherie zur Verfügung steht. Danach wird in Abschnitt 2.2 auf das Situationskalkül eingegangen, welches Grundlage für die hier verwendeten aktionsbasierten Programmiersprachen ist. Zum Abschluss wird noch in Abschnitt 2.3 der existierende YAGI-Interpreter beschrieben, welcher den Ausgangspunkt dieser Arbeit darstellt.

2.1 Lego-Mindstorms-Ev3

Der Lego-Mindstorms-Ev3 ist ein Roboter-Bausatz, den man umbauen und auch selbst programmieren kann. Er ist der Nachfolger des Lego-Mindstorms-NXT. Der Lego-Mindstorms-Ev3 besteht aus folgenden Hauptbestandteilen: dem Ev3-Brick, den großen und mittleren Motoren und den Sensoren. Der wichtigste Teil des Lego-Mindstorms-Ev3-Kit, der Ev3-Brick, ist ein kompakter Computer, auf dem verschiedene Programme laufen können. Sensoren und Motoren können mit Hilfe anderer Lego Teile am Ev3-Brick montiert werden. Diese können über Befehle angesteuert oder Informationen gelesen werden.

Der Ev3-Brick ist laut (LEGO, 2013) ein ARM9 Microcontroller mit 16MB Flashspeicher und 64MB RAM auf dem standardmäßig ein Linux Betriebssystem läuft. Lego entwickelte eine eigene Lego Distribution, welche sich ideal für die LEGO-MINDSTORMS-EV3-Entwicklungsumgebung eignet. Diese hat jedoch den Nachteil, dass man nicht über SSH auf den Ev3-Brick zugreifen und diverse Bibliotheken installieren kann. Dadurch wird die Lego Distribution für andere Anwendungen unbrauchbar. Der Ev3-Brick verfügt zudem über ein Matrixdisplay und einige Knöpfe für die Bedienung. Mit Hilfe der drei LEDs (rot, gelb und grün) wird der Status des Betriebssystems angezeigt. Des weiteren kann über den Mini-USB-Port eine Verbindung zu einem PC aufgebaut werden oder über den USB-Host-Port mit Hilfe eines WLAN-USB-Sticks, der den Zugriff auf den Brick über das Netzwerk ermöglicht. Ebenfalls stellt der Ev3-Brick Bluetooth für die Verbindung zu anderen Geräten zur Verfügung. Der Ev3-Brick besitzt einen Lautsprecher für die Ausgabe von Signalen oder Musik. An den vier Input-Ports können verschiedene Lego-Sensoren angeschlossen werden und mit den vier Output-Ports können diverse Lego-Motoren gesteuert werden. Eine Übersicht des Ev3-Bricks mit dazugehöriger Peripherie wird in Abbildung 2 dargestellt.

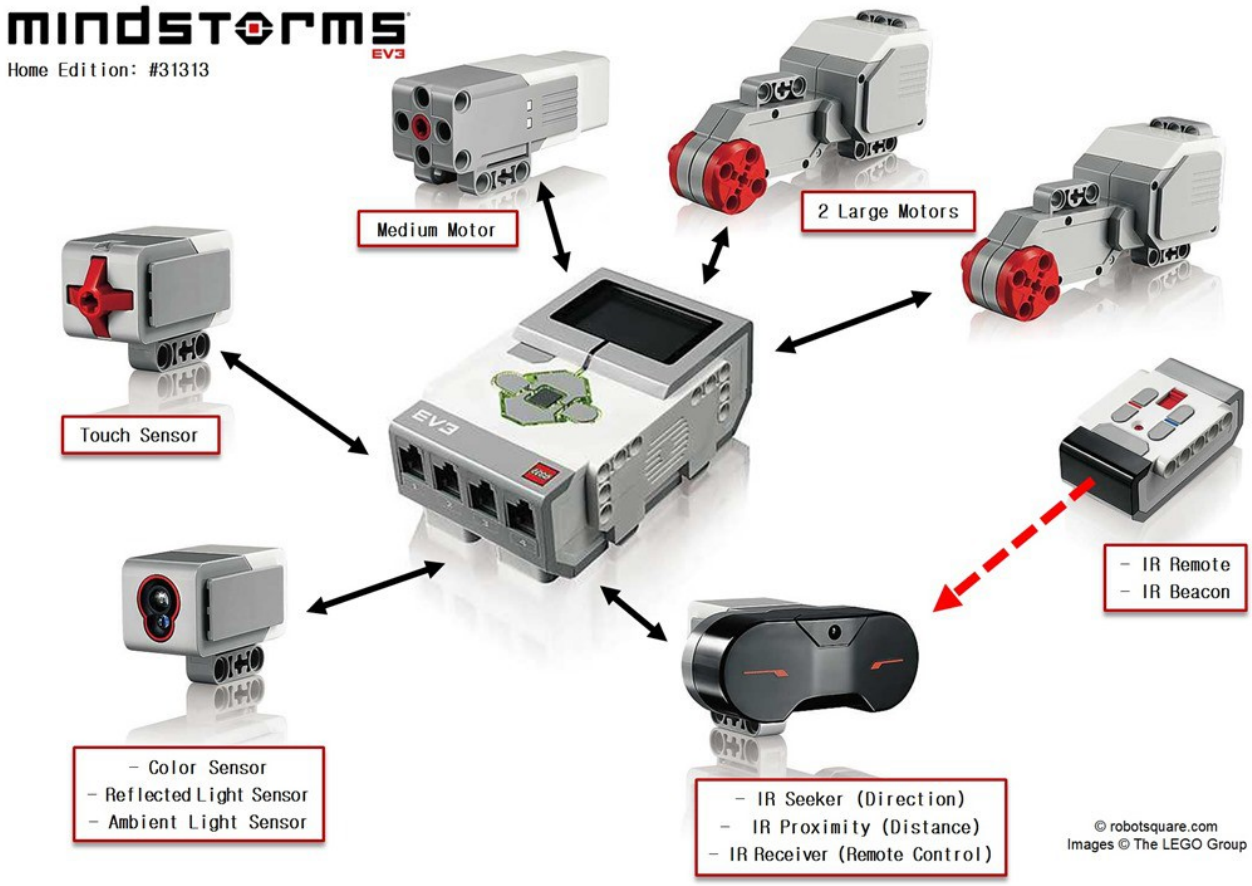

Abbildung 2: Übersicht des Ev3-Bricks mit Peripherie (Quelle: www.robotsquare.com)

2.1.1 Peripherie

Die Motoren des Lego-Mindstorms-Ev3 verfügen über einen eingebauten Encoder mit einer Winkelauflösung von 1°, der eine präzise Steuerung erlaubt. Über entsprechende Software-Befehle können die Motoren gestartet, gestoppt oder die Geschwindigkeit, beziehungsweise die Drehrichtung geändert werden. Weiters kann man Informationen wie Status, Position und Geschwindigkeit der Motoren auslesen. Der Mindstorms-Ev3-Motor verfügt zudem über eine Geschwindigkeitsregulierung, die den Ladezustand des Akkus und das Transportieren verschiedene Lasten berücksichtigt und dementsprechend entgegen wirkt. Eine Grafik des großen Lego-Mindstorms-Ev3-Motors wird in Abbildung 3 gezeigt.

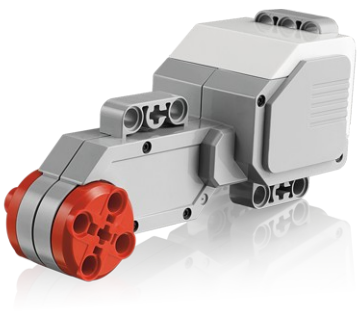

Abbildung 3: Großer Lego-Mindstorms-Ev3-Motor

Am Lego-Mindstorms-Roboter können viele verschiedene Sensoren angehängt werden, jedoch werden in diesem Abschnitt nur die im Lego-Mindstorms-Ev3 Paket mitgelieferten Sensoren beschrieben. Einer der drei mitgelieferten Sensoren ist der Farbsensor, welcher in Abbildung 4 abgebildet wird. Er ist ein digitaler Sensor, der Farben bzw. die Intensität des Lichtes mit einer Abtastrate von 1kHz/sec misst. Der Farbsensor kann in drei verschiedenen Modi verwendet werden. Wählt man den „Color Mode“, so erkennt der Sensor Farben und liefert dementsprechend einen Wert von 0 bis 8 zurück, wobei diese Werte den Farben Schwarz, Blau, Grün, Gelb, Rot, Weiß, Braun oder keiner Farbe entsprechen. Der zweite Modus ist der „reflected Light Intensity Mode“, bei dem der Farbsensor nur die Intensität des Lichtes, dass von der Rotlicht aussendenden Lampe reflektiert wird, misst. Die Rückgabewerte sind im Bereich von 0 (sehr dunkel) bis 100 (sehr hell). Beim „Ambient Light Intensity Mode“ wird die Intensität des Lichtes gemessen, das von der Umgebung zum Sensor gelangt. Dabei kann es sich um ein Sonnenlicht oder eine externe künstliche Lichtquelle handeln. Falls der „reflected Light Intensity Mode“ oder der „Color Mode“ verwendet wird, sollte der Farbsensor im rechten Winkel und nahe an der Oberfläche platziert werden. Der Sensor sollte aber die Oberfläche nicht berühren, da dies die Messung verfälschen würde.

Abbildung 4: Lego-Mindstorms-Ev3-Farbsensor

Ein weiterer Sensor des Lego-Mindstorms-Ev3 ist der Berührungssensor. Dieser digitale Sensor ermittelt ob der rote Knopf auf dem Sensor gedrückt oder losgelassen wurde. Das heißt der Sensor kann auf die Bedienungen gedrückt, losgelassen oder angestoßen (gedrückt und losgelassen) reagieren. Eine Darstellung des Berührungssensors wird in Abbildung 5 gezeigt.

Abbildung 5: Lego-Mindstorms-Ev3-Berührungssensor

Der Infrarotsensor ist ein digitaler Sensor, welcher primär die Infrarotlicht-Reflexionen von Gegenständen misst. Er kann aber auch Infrarotlicht-Signale von entfernten Infrarot-Signalstationen erkennen. Auch der Infrarotsensor kann in drei verschiedenen Modi verwendet werden. Im „Proximity Mode“ verwendet der Sensor reflektierende Infrarotwellen, um die Entfernung zwischen Sensor und Gegenstand zu schätzen. Der Rückgabewert ist zwischen 0 (sehr nahe) und 100 (sehr entfernt). Der Wert 100 entspricht etwa 70cm. Mit Hilfe des so genannten „Beacon Mode“ kann der Sensor eine externe Infrarot-Signalstation erkennen. Dazu wählt man einen der vier Infrarot-Kanäle, die zur Verfügung stehen. Der Sensor ortet das Signal, das auf diesen Kanal gesendet wurde auf eine Distanz von 200 cm in Richtung des Sensors. Der Sensor kann die Hauptrichtung der Signalstation bestimmen und die Entfernung dazu schätzen. Der Rückgabewert der Richtung liegt zwischen -25 und 25, wobei der Wert 0 geradeaus entspricht. Für den letzten Modus, dem „Remote Mode“, benötigt man die im Ev3 Paket enthaltene Remote-Infrarot-Signalstation. Dies ist ein eigenes Gerät, mit dessen Hilfe Befehle an den Mindstorms Ev3 gesendet werden können. Wird ein Knopf oder eine Kombination von Knöpfen der Remote-Infrarot-Signalstation gedrückt, so sendet diese die Informationen mittels Infrarot an den Sensor. Damit kann man dem Mindstorms Ev3 Befehle erteilen. In Abbildung 6 wird der Infrarotsensor mit dazugehöriger Infrarot-Signalstation dargestellt.

Abbildung 6: Lego-Mindstorms-Ev3-Infrarotsensor mit Infrarot-Signalstation

2.1.2 Entwicklungsumgebung

Lego liefert zum Lego-Mindstorms-Ev3 eine eigene Software, Lego-Mindstorms-Ev3 Home Edition, zum Programmieren des Roboters mit. Diese ist eine visuelle Programmiersprache, die laut (Lego, 2013) auf Windows und Apple Macintosh läuft oder auch als Android App erhältlich ist. In dieser Software sind Bauanleitungen zu verschiedenen Robotertypen enthalten. Zu diesen Roboter Typen werden auch diverse Programmiereinführungen und Beispielprogramme zur Verfügung gestellt. Die Lego Software ist als Baukastenprinzip aufgebaut, bei dem verschiedene Blöcke per „drag and drop“ in ein Programm eingebaut werden können. Danach können verschiedene Einstellungen an den jeweiligen Blöcken vorgenommen werden. Die Programmierrichtung ist dabei „straight forward“, das heißt das Programm ist ein Kontrollflussgraph. In Abbildung 7 wird ein Beispielprogramm, welches in der Lego-Mindstorms-Ev3 Home Edition geschrieben wurde, gezeigt. In diesem Beispiel fährt der Roboter Roboter solange gerade aus, bis der Berührungssensor gedrückt wird. Dabei leuchtet die grüne LED. Wird der Sensor gedrückt, zum Beispiel wenn der Roboter gegen ein Hindernis fährt, so ertönt ein Signal und die rote LED leuchtet. Anschließend fährt der Lego-Mindstorms-Ev3 ein Stück zurück und dreht sich nach rechts. Nun fährt der Roboter wieder gerade aus und der Zyklus beginnt von vorne.

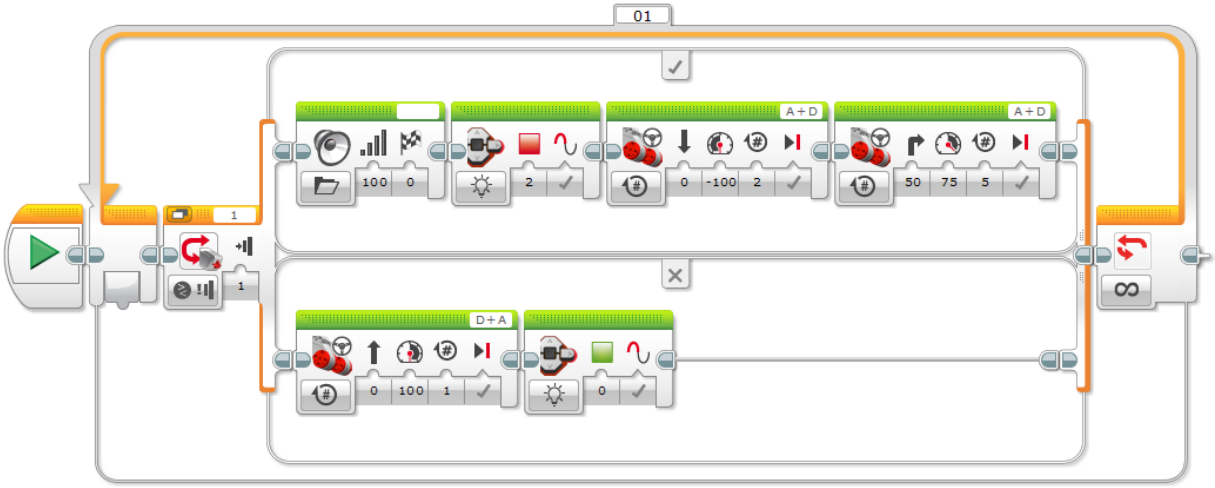

Abbildung 7: Beispielprogramm geschrieben mit Lego-Mindstorms-Ev3 Home Edition

In der Lego-Mindstorms-Ev3-Entwicklungsumgebung gibt es fünf verschiedene Arten von Blöcken: Aktionsblöcke, Flussblöcke, Sensorblöcke, Datenblöcke und erweiterte Blöcke. Mit den Aktionsblöcken werden Hardwaregeräte wie Motoren, Display oder LEDs angesteuert. Will man zum Beispiel den Legoroboter vorwärts bewegen, benötigt man einen Motorblock. Bei diesem Block kann man dann zusätzlich die Fahrtrichtung, Geschwindigkeit und die Dauer einstellen. Die Flussblöcke steuern den Programmfluss. Will man einen Block mehrmals wiederholen, oder ist ein Teil des Programms von den Sensorwerten abhängig, werden Schleifenblöcke oder Schalterblöcke verwendet. Mit Hilfe der Sensorblöcken werden die Werte von den Sensoren gemessen. Dabei kann man auch die verschiedenen Modi der Sensoren auswählen. Die Datenblöcke verwendet man für Variablen, Konstanten und Operationen, während die erweiterten Blöcke für File-Zugriffe oder Bluetooth-Verbindungen zuständig sind.

Ist der Lego-Mindstorms-Ev3 mit dem PC, auf dem die Entwicklungssoftware läuft, verbunden, so erkennt die Software automatisch welche Sensoren und Motoren an welchen Ports angeschlossen sind. Die Ports können jedoch auch manuell in den einzelnen Blöcken eingestellt werden. Da die Software visuell funktioniert, ist sie sehr leicht zu verwenden und daher vor allem für Kinder geeignet. Tippfehler oder Rechtschreibfehler, die in textuellen Programmiersprachen häufig auftreten, kommen daher nicht vor, Ablauf- und logische-Fehler allerdings sehr wohl.

2.2 Situationskalkül

Der Situationskalkül basiert auf second-order Logik und wurde speziell dafür entwickelt, um dynamische Welten zu repräsentieren. Er wurde eingeführt von (McCarthy, 1963) und von (Reiter, 2001) verfeinert. Der Situationskalkül ist über eine Menge von Variablen, Konstanten, Funktionen

und Prädikaten definiert. Er basiert auf der Idee, dass jede Veränderung der Welt das Resultat einer sogenannten *Aktion* ist. Eine mögliche Vergangenheit, welche eine simple Sequenz von Aktionen ist, wird durch den Funktionsterm *Situation* repräsentiert. Die Konstante S_0 wird für die Ausgangssituation verwendet, welche eine leere Sequenz von Aktionen darstellt. Das binäre Funktionssymbol *do(a, s)* repräsentiert, dass die Nachfolgesituation von *s*, aus dem Ausführen der Aktion *a* resultiert. Aktionen sind auch Funktionssymbole, welch parametriert sein können. Zum Beispiel *put(x, y)* könnte dafür stehen, ein Objekt x auf das Objekt y zu legen. Dann bezeichnet *do(put(A, B), s)* die Situation resultierend aus der Aktion, das Objekt *A* auf das Objekt *B* zu legen, in der Situation *s*. Aktionen haben Vorbedingungen, welche erfüllt werden müssen, damit eine Aktion in der derzeitigen Situation ausgeführt werden kann. Dazu wird das Prädikatsymbol *Poss* eingeführt. *Poss(a, s)* bedeutet, dass es möglich ist, die Aktion *a* in der Situation auszuführen, welche durch die Ausführung der Sequenz von Aktionen repräsentiert in *s* entsteht. Wenn es zum Beispiel einem Roboter *r* möglich ist ein Objekt *x* in der Situation *s* aufzunehmen, dann hält er noch kein Objekt, er ist nahe dem Objekt *x* und *x* ist nicht zu schwer:

$$Poss(pickup(r,x),s) \equiv [(\forall z) \neg holding(r,z,s)] \wedge \neg heavy(x) \wedge nextTo(r,x,s)$$

Um die veränderbaren Eigenschaften der Welt zu beschreiben, werden so genannte *fluents* in der Form $F(x_1,\ldots, x_n, s)$ verwendet. Diese werden noch unterschieden zwischen *relational fluents*, welche einen Wahrheitswert einer Relation in einer bestimmten Situation beschreiben, und *functional fluents*, welche einen Wert einer Funktion in einer bestimmten Situation liefern.

Um das Situationskalkül zu formalisieren, hat (Reiter, 2001) ein Set von Axiomen definiert, welche eine sogenannte B*asic Action Theorie* $\mathcal{D}$ wie folgt aufstellt:

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{ss} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}$$

In der *Basic Action Theorie* bezeichnet $\sum$ die vier grundlegenden Bereich-unabhängige Axiome, welche Eigenschaften einer legalen Situation definiert. $\mathcal{D}_{ss}$ beschreibt eine Menge von *successor state axiom* für alle Fluents in der Form von $F(x, do(a,s)) \equiv \Phi_F(x, a, s)$, die für jeden Fluent F festlegen, unter welchen Konditionen sie in der Situation do(a,s) gelten. $\mathcal{D}_{ap}$ enthält eine Menge von Aktion Vorbedingung Axiome in der Form von $Poss(A(y,s)) \equiv \Pi_A(y,s)$, die für jede Aktion A festlegt, unter welchen Bedingungen diese ausgeführt werden kann. $\mathcal{D}_{una}$ hält eine Menge von eindeutigen Namen Axiomen für Axiome in der Form von $A(x) = A(y) \supset x = y$, um zwei Aktionen als gleich zu deuten, wenn sie den gleichen Namen und den gleichen Parameter-Vector haben. Zum Schluss bezeichnet $\mathcal{D}_{S_0}$ ein Set von Axiome, die den Anfangszustand beschreiben. Das können einerseits Fluents sein, welche gelten bevor irgendeine Aktion ausgeführt wird oder andererseits Fakten, welche „zeitunabhängig“ sind.

Es folgen kurze Beispiele zu den Axiomen der *Basic Action Theorie*:

Die vier grundlegende Bereich-unabhängige Axiome Σ:

$$do(a_1, s_1) = do(a_2, s_2) \equiv a_1 = a_2 \wedge s_1 = a_2 \quad ,$$

$$(\forall P).P(S_0) \wedge (\forall a, s)[P(s) \supset P(do(a, s))] \supset (\forall s)P(s) \quad ,$$

$$\neg s \sqsubset S_0 \quad ,$$

$$s \sqsubset do(a, \acute{s}) \equiv s \sqsubset \acute{s} \vee s = \acute{s} \quad ,$$

wobei s $\sqsubset$ s' bedeutet, dass man durch das Hinzufügen von einer oder mehreren Aktionen zu s, die Sequenz von Aktionen s' erhält.

successor state Axiome $\mathcal{D}_{ss}$:

$$holding(r, x, do(a, s)) \equiv a = pickup(r, x) \vee holding(r, x, s) \wedge a \neq drop(r, x)$$

Vorbedingung Axiome $\mathcal{D}_{ap}$:

$$Poss(drop(r, x), s) \equiv robot(r) \wedge holding(r, x, s)$$

eindeutige Namen Axiome $\mathcal{D}_{una}$:

$$pickup(r, x) \neq drop(\acute{r}, y), pickup(r, x) \neq walk(\acute{r}, y)$$

Anfangszustand Axiome $\mathcal{D}_{S_0}$:

$$robot(r), nextTo(r, x, S_0)$$

2.3 YAGI

YAGI (Ein Akronym für **Y**et **A**nother **G**olog **I**nterpreter) ist eine aktionbasierte, imperative Programmiersprache entwickelt von (Maier et al., 2015), die auf dem Situationskalkül und IndiGolog (De Giacomo et. al., 2009) aufgebaut ist. Im Gegensatz zu den meisten Golog-Interpretern, wird eine strikte Trennung von Syntax und Semantik eingehalten, wodurch keine enge Kopplung zur Implementierung entsteht. YAGI wurde bewusst entworfen, um Probleme von Prolog-basierten Implementationen von Golog weitgehend zu vermeiden.

2.3.1 Fundamentale Aktionstheorie

Die Sprache YAGI basiert auf der Idee von (Ferrein et al., 2012). In dieser Arbeit wurden einige funktionale und nicht-funktionale Bedingungen festgelegt, welche bedacht werden müssen, wenn man eine neue aktionbasierte Programmiersprache konstruiert. Weiters wurde ein erstes Beispiel der YAGI Sprachdefinition zur Verfügung gestellt, um die Ideen von YAGI zu veranschaulichen. Zudem definieren (Ferrein et al., 2012) eine Syntax und Semantik für YAGI, die (Maier et al, 2015)

als Grundlage für seine Arbeit verwenden. (Ferrein et. al., 2012) präsentieren ebenfalls eine *Basic Action Theorie* für YAGI, die so genannte YAGI-BAT:

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{pres} \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{unc}.$$

Basierend auf YAGI-BAT, definieren (Maier et al., 2015) eine *Basic Action Theorie* $\mathcal{D}_{YAGI}$ als

$$\mathcal{D}_{YAGI} = \Sigma \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{unc},$$

in welcher $\sum$, $\mathcal{D}_{ssa}$, $\mathcal{D}_{ap}$, $\mathcal{D}_{una}$ und $\mathcal{D}_{S0}$ dieselben Axiome beinhalten wie sie (Reiter, 2001) definiert hat. $\mathcal{D}_{unc}$ beinhaltet die eindeutige-Name Axiome, die verwendet werden, um YAGI-String-Tokens zu repräsentieren. $\mathcal{D}_{YAGI}$ ist eine reduzierte Version von YAGI-BAT, da (Ferrein et. al., 2012) auch das Set von Presburger Arithmetik Axiome (Enderton, 2001) $\mathcal{D}_{pres}$ hinzufügten, um die grundlegenden arithmetischen Operationen zu repräsentieren, welches (Maier, et. al. 2015) zur Vereinfachung weglassen. Die Presburger Arithmetik ist eine mathematische Theorie der natürlichen Zahlen mit Addition. Da in der Presburger Arithmetik die Multiplikation weggelassen wurde, ist sie eine entscheidbare Theorie.

YAGI wurde auch auf den Datentyp String limitiert, da Terme und Formeln in der Prädikatenlogik erster Stufe aus Strings von Symbolen besteht und es sonst zu Schwierigkeiten mit der Theorie kommt.

2.3.2 Die wichtigsten Befehle in YAGI

Um einen Einblick über die Programmiersprache YAGI zu bekommen folgt ein kurzer Überblick der wichtigsten Befehle, die (Maier et al,, 2015) in YAGI implementiert haben. Diese werden anhand des Beispiels eines Roboter Paketdienstes demonstriert:

- **Fluents:** Fluents sind assoziative Arrays, welche die veränderbaren Eigenschaften der Welt beschreiben. Sie werden definiert durch das Schlüsselwort *fluent,* gefolgt von einem Namen und ein oder mehreren Paaren in eckigen Klammern. Innerhalb jeder eckigen Klammer muss eine Domain-Spezifikation in geschwungenen Klammern angegeben werden. Es können Initial-Werte für die einzelnen Fluents zugeteilt werden. Diese haben prinzipiell den Typ „Set von Tupels“, welcher durch spitze Klammern innerhalb von geschwungenen Klammern gekennzeichnet sind. Es gibt drei verschiedene Arten der Zuweisung von Fluents. Mit der ergänzenden Zuweisung (Zuweisungsoperator +=) und der entfernenden Zuweisung (Zuweisungsoperator -=) können Tupels dem Fluent hinzugefügt bzw. entfernt werden. Mit dem überschreibenden Zuweisung (Zuweisungsoperator =) wird der Fluent wahr für alle die angegebenen Tupel. In der nachfolgenden Listing werden die Fluents eines Lego-Mindstorms-Ev3-Paketdienst dargestellt.

```
// location of the robot
fluent location [{"room1", "room2", "room3", "room4"}];
location = {<"room1">};

// holding object
fluent holding [{"false", "true"}];
object = {<"false">};

// requests from sender to receiver
fluent request [{"person1", "person2", "person3"}][{"person1", "person2", "person3"}];
```
Listing 1: Roboter Paketdienst Fluents

- **Facts:** Facts sind ähnlich wie Fluents, mit der Ausnahme, dass diese nach dem Initialisieren nicht änderbar sind. Semantisch gesehen gibt es keinen Unterschied zwischen Facts und Fluents. Ein Beispiel dazu wird in Listing 2 gezeigt.

```
// every person has an office
fact office [{"person1", "person2", "person3"}][{"room1", "room2" , "room3"}];
office = {<"person1", "room1">, <"person2", "room2">, <"person3", "room3">};
```
Listing 2: Roboter Paketdienst Fact

- **Aktionen:** Aktionen beginnen mit dem Schlüsselwort *action* und enden mit dem Schlüsselwort *end action*. Nach dem Namen der Aktion können Parameter in runden Klammern übergeben werden und Rückgabewerte durch das *external* Attribut geholt werden. Durch das *external* Attribut wird die Aktion zu einer Sensing-Aktion, wodurch Eigenschaften der Welt ermittelt werden können. Innerhalb einer Aktion befinden sich der *precondition*-, der *effect*- und der *signal*-Block. Im *precondition*-Block können Vorbedingungen beschrieben werden, unter welchen die Aktion ausgeführt werden kann. Mit Hilfe des *effect*-Blocks werden Anweisungen gegeben, um die Auswirkungen der Aktion auf die Welt zu beschreiben. Der *signal*-Block ist dafür zuständig einen entsprechenden Signal-Befehl an das System-Interface zu senden, damit eine Aktion ausgeführt werden kann. Die Semantik einer normalen YAGI Aktion entspricht der Semantik einer IndiGolog Prozedur. In Listing 3 werden mehrere Aktionen für einen Paketdienst Roboter aufgelistet. Wie in dem Beispiel ersichtlich ist, wird mit dem Präfix $ eine Variable definiert. In den Vorbedingungen und in den Effekten können YAGI Formeln enthalten sein. YAGI Formeln sind laut (Maier et al,, 2015) wie folgt aufgebaut:

⟨formula⟩ ::= ⟨atom⟩
| **not (** ⟨formula⟩ **)**
| **(** ⟨atom⟩ ⟨connective⟩ ⟨formula⟩ **)**
| **exists** ⟨tuple⟩ **in** ⟨setexpr⟩ **(** **such** ⟨formula⟩ **)?**
| **all** ⟨tuple⟩ **in** ⟨setexpr⟩ **(** **such** ⟨formula⟩ **)?**
| ⟨tuple⟩ **in** ⟨setexpr⟩

⟨atom⟩::= ⟨value⟩⟨comp_op⟩⟨value⟩
| ⟨setexpr⟩⟨comp_op⟩⟨setexpr⟩
| (**true** | **false**)

⟨comp_op⟩::= == | != | <= | >= | < | >

⟨connective⟩::= **and** | **or** | **implies**

```
// robot drive to room
action driveToRoom($room)
precondition:
  // robot is not in the room
  not ($room) in location;
effect:
  // now the robot is in the room
  location = {<$room>};
signal:
  "Move to room " + $room;
end action

// robot picks up an object
action pickUpObject()
precondition:
  // robot is not holding an object
  not (<"true">) in holding;
effect:
  // now the robot holds an object
  holding = {<"true">};
signal:
  "Pickup";
end action

// robot puts an object down
action putDownObject()
precondition:
  // robot holds an object
  not (<"false">) in holding;
effect:
  // now the robot do not hold an object
  holding = {<"false">};
signal:
  "Put down";
end action
```

Listing 3: Roboter Paketdienst Aktionen

- **Exogene Events:** Exogene Events sind ähnlich zu Aktionen, werden allerdings vom System-Interface getriggert und verändern somit die interne Repräsentation der Welt, basierend auf ein äußeres Event. Exogene Events beginnen mit dem Schlüsselwort *exogenous-event* und enden mit dem Schlüsselwort *end exogenous-event*, wie in Listing 4 entnommen werden kann.

```
// get request from external
exogenous-event receive_request ($sender, $receiver)
  request += {<$sender, $receiver>};
end exogenous-event
```
Listing 4: Roboter Paketdienst Exogenouse-Event

- **Prozeduren:** Prozeduren beschreiben eine Abfolge von YAGI Befehlen. Sie sind durch das Schlüsselwort *proc* und das Schlüsselwort *end proc* definiert. Prozeduren können durch die Verwendung von simplen Aktionen und anderen YAGI Befehlen eine komplexe Aktion nachbilden. Eine Prozedur zum Liefern von Paketen mit dem Lego-Mindstorms-Ev3 wird in Listing 5 abgebildet.

- **Bedingungen:** Bedingungen erkennt man durch das Schlüsselwort *if* gefolgt von einer Bedingung, in Form einer Formel, in runden Klammern. Danach folgt ein *then*-Block und ein optionaler *else*-Block. Je nach Evaluierung der Formel, wird einer der beiden Blöcke ausgeführt. Conditionals enden mit dem Endtoken *end if*. Ein Beispiel für eine Bedingung kommt in der *main* Prozedur in Listing 5 vor.

- **Schleifen:** In YAGI wurden zwei Arten von Schleifen implementiert, die while-Schleife und die for-Schleife. Die while-Schleife beginnt mit dem Schlüsselwort *while* gefolgt von einer Bedingung in runden Klammern. Dann folgt ein *do*-Block, welcher so lange ausgeführt wird, wie die Bedingung wahr ist. Die while-Schleife endet mit dem Schlüsselwort *end while*. Die for-Schleife ist durch das Schlüsselwort *foreach* und dem Schlüsselwort *end for* definiert. Nach dem Starttoken muss ein Tupel angegeben werden, gefolgt von dem Schlüsselwort *in* und einem Set. Bei einer for-Schleife wird der Block innerhalb der Schleife für jedes Tupel in dem gegebenem Set ausgeführt. In der nachfolgenden Listing wird eine for-Schleife dargestellt. Ein Schlüsselwort, welches in dem nachfolgenden Beispiel ebenfalls verwendet wird ist *pick*. Dabei wird ein Tupel von einem gegebenen Set ausgewählt und der nachfolgende Block mit dem ausgewählten Tupel als Parameter ausgeführt. Ein ähnliches Schlüsselwort in YAGI ist *choose*, bei welchem einer der gegebenen Programmblöcke ausgewählt und ausgeführt wird. *Pick* und *choose* sind nicht deterministisch, daher sollten, für eine sichere Ausführung des Programms, solche Blöcke mit einem *search* Operator verknüpft werden. Mehr Details zum Suchoperator search wird im nächsten Abschnitt beschrieben.

```
proc main()
 // do all orders
 foreach <$sender, $receiver> in request do
  pick <$sender, $roomSender> from office such
       if(not ($roomSender) in location) then
         driveToRoom($roomSender);
       end if
    pickUpObject();
  end pick
  pick <$receiver, $roomReceiver> from office such
       if(not ($roomReceiver) in location) then
     driveToRoom($roomReceiver);
       end if
    putDownObject();
  end pick
 end for
end proc
```

Listing 5: Roboter Paketdienst Prozedur

2.3.3 Der Suchoperator in YAGI

YAGI benutzt die online Ablauf Semantik von Indigolog, mit deren Hilfe Aktionen online ausgeführt werden können. Das heißt Aktionen werden in YAGI schrittweise direkt auf dem Roboter ausgeführt. Dabei können vor jeder Ausführung Informationen über die Umgebung des Roboters, mit Hilfe von Sensoren, gewonnen werden und durch die Vorbedingungen der Aktionen entschieden werden ob die Ausführung möglich ist oder nicht. Dadurch kann es aber passieren, dass der Roboter Aktionen ausführt, welche unter Umständen nicht mehr rückgängig gemacht werden können. Um einen Teil des Programms vor dem Ausführen zu planen, wurde der Operator *search* von (Maier et al., 2015) eingeführt.

Der *search* Operator ist dafür zuständig, einen gültigen Programmpfad im YAGI Programm zu finden, bevor dieser Ausgeführt wird. Wird ein *search* Operator ausgeführt, so wird ein Snapshot des aktuellen Zustand der Welt gemacht. Das heißt, es wird eine Kopie von allen Fluents und Fakten der Datenbank und deren Werten erstellt. Auf diese Kopie werden virtuelle Ausführungen von Aktionen gemacht, d.h. die Ausführungen werden nicht in der realen Welt getätigt, bis ein Programmpfad gefunden wurde, welcher das gewünschte Ergebnis liefert. Ist die virtuelle Ausführung erfolgreich, so können diese auch in der realen Welt ausgeführt werden, anderenfalls wird eine entsprechende Fehlermeldung ausgegeben.

Die Ausführung eines *search* Operators ist vor allem dann interessant, wenn nicht deterministische Entscheidungen Teil des YAGI Programms sind. Kommt es während der virtuellen Ausführung dazu ein *pick* oder *choose* Statement auszuführen, so wird für jede Möglichkeit des *pick* oder *choose* Statements ein neuer Durchführungszweig, d.h. eine weitere Kopie von allen Fakten und Fluents und deren Werten erstellt. Jeder dieser Zweige ist für genau eine Möglichkeit des *pick* oder *choose* Statements verantwortlich. Die Suche über den unterschiedlichen Zweigen erfolgt nach

einer Breitensuche. Das heißt, jeder Zweig führt genau eine YAGI Aktion aus und wartet anschließend bis die anderen Zweige die Aktionen beendet haben. Nachdem alle Aktionen der Zweige ausgeführt wurden, wird überprüft, ob die Ausführungen erfolgreich waren. Konnte eine Aktion wegen einer Verletzung der Vorbedingung nicht ausgeführt werden, so wird dieser Zweig von den möglichen Durchführungszweigen entfernt. Alle Zweige, die bis zur letzten Aktion ausgeführt werden können, werden zur weiteren Programmausführung berücksichtigt.

3 Zusammenhängende Arbeiten

In diesem Kapitel werden einige Themen erörtert, welche im Zusammenhang mit dieser Arbeit stehen. Dazu wird zuerst die Programmiersprache Golog, welche als Grundidee für YAGI hergenommen wurde, in Abschnitt 3.1 vorgestellt. Anschließend wird in Abschnitt 3.2 das Legolog-System beschrieben, welches einen Golog-Controller mit einem Lego-Mindstorms-Roboter kombiniert. Zum Schluss dieses Kapitels, werden in Abschnitt 3.5 die Vor- und Nachteile der Verwendung von Robotern im Schulunterricht diskutiert.

3.1 Golog

Golog, eine Abkürzung für „alGOL in LOGic“, ist eine logische Programmiersprache für dynamische Systeme, entwickelt von (Levesque et al., 1994). Sie ist auf die formale Aktionstheorie aufgebaut, welche in der erweiterten Version des Situationskalkül spezifiziert ist. Golog vermischt den ALGOL Programmierstil, welcher Sequenzen, Konditionen, rekursive Prozeduren und Schleifen beinhaltet, mit Logik. Im Gegensatz zu üblichen Programmiersprachen, bei denen Programmteile lediglich den Maschinenzustand ändern, werden Golog Programme in Primitiven zerlegt, welche sich meist auf Aktionen in der externen Welt beziehen. Dadurch stellt Golog automatisch ein explizites Modell der Welt bereit, welches zur Lauf-Zeit abgefragt und logisch begründet werden kann. Ein Vorteil von Golog ist, dass man Programme auf einem viel höheren Level der Abstraktion schreiben kann. Ebenso hat sich herausgestellt, dass die Sprache sehr gut für Anwendungen im Bereich „hight level control“ von Robotern und industriellen Prozessen, intelligente Software Agenten, ereignisorientierte Simulation, usw. geeignet ist. Ein erster Prototyp wurde bereits von (Levesque et. al., 1994) in Prolog implementiert. (Reiter, 2001) beschriebt in seinen Buch wie man einen Golog Interpreter in Prolog nachbilden kann.

Golog verwendet eine Offline-Ausführungssemantik. Diese hat zufolge, dass beim Ausführen der gesamte Programmcode durchsucht werden muss, um eine legale Sequenz von Aktionen zu finden, bevor diese ausgeführt werden kann. Die Offline-Ausführungssemantik ist ungeeignet für interaktive Systeme. Daher wurde Indigolog (incremental deterministic Golog) von (De Giacomo et al., 2009) eingeführt, welches eine online Programmausführung ermöglicht. Indigolog wählt schrittweise eine Aktion aus und führt diese anschließend gleich aus. Deshalb kann Indigolog auch mit Abtastaktionen und exogenen Events umgehen. Aber auch die Online-Ausführungssemantik hat einen Nachteil, da manche Aktionen in der echten Welt nicht mehr rückgängig gemacht werden können. Um die Offline- und Online-Semantik zu vereinen, haben (De Giacomo et al., 2009) den Suchoperator Σ eingeführt. Der Suchoperator simuliert die Offline-Ausführung nur für einen Teil

des Programms, bevor dieser ausgeführt wird. In YAGI wird dieser Suchoperator durch den *search* Operator eingeführt.

3.2 Legolog

Legolog, entwickelt von (Levesque und Pagnucco, 2000), ist ein System, welches einen Controller der Golog Familie des Planens verwendet, um einen Lego-Mindstorms-Roboter zu kontrollieren. Legolog ist im Stande mit primitiven Aktionen, exogenen Aktionen und Abtastung umzugehen. Das System besteht aus zwei Hauptkomponenten, dem RIS (Robotics Invention System), welches ein LEGO-Mindstorms System ist und dem Golog-Controller. Das Herz des RIS ist der RCX (Robotic Command Explorer), welcher einen Hiatchi H8/3297 Microcontroller enthält. Der RCX-Brick ist der die erste Generation der Lego-Mindstorms Familie und kann mit Hilfe der gratis Software NQC (Kurzform für Not Quite C) in einer Variante von C programmiert werden. Das RIS ist dafür zuständig mit den Geräten, die an den Input- und Output-Ports angehängt sind, zu kommunizieren. Der Golog-Robotic-Controller stellt die High-Level Steuerung des Roboters zur Verfügung.

Die Struktur der Legolog Anwendung kann in Abbildung 8 betrachtet werden. Auf dem Golog Interpreter läuft ein Golog Programm (in diesem Fall IndiGolog). Immer wenn der IndiGolog Interpreter eine Aktion ausführen will, wird Prolog Kommunikation verwendet, um es dem RCX mitzuteilen. Das NQC Programm, welches am RCX läuft, bestätigt die Nachricht und führt die Aktion aus. Der RCX kann ebenfalls das Auftreten einer exogenen Aktion an den Golog Interpreter melden. Da die gesamte Kommunikation von Prolog ausgeht, muss abgefragt werden, ob irgendeine exogene Aktion bemerkt wurde. Prolog läuft hier im Gegensatz zu YAGI auf einem externen Rechner.

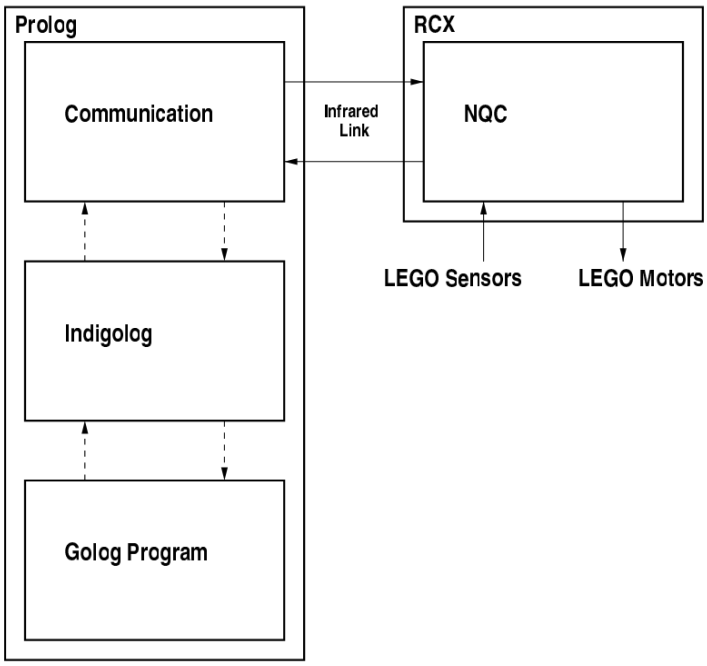

Abbildung 8: Idealisierte Ansicht der Legolog-Struktur von (Levesque et. al., 2000)

Legolog ist zurzeit in Prolog geschrieben und läuft unter den Linux-Betriebssystemen. Das System ist so aufgebaut, dass der Golog-Controller einfach gegen einen anderen alternativen Planer ausgetauscht werden kann. Legolog wurde entwickelt um Experimente und Demonstrationen mit einem Golog-Planer auf dem Lego-Mindstorms zu machen. Ein Nachteil bei Legolog ist, dass das System eine Kommunikation zwischen dem Golog Controller und dem RIS benötigt wird. Es gibt allerdings auch schon Versuche, einen Golog Interpreter direkt auf einem Lego-Mindstorms Brick laufen zu lassen. In einem Artikel von (Ferrein und Steinbauer, 2010) wird beschrieben, wie man einen Golog Interpreter der in der Scriptsprache Lua geschrieben ist, direkt auf einem Lego-Mindstorms laufen lassen kann. Ein Prototyp des Golog Interpreters in Lua wurde bereits von (Ferrein, 2010) entwickelt. YAGI läuft im Gegensatz zu Legolog auch direkt auf dem Lego-Mindstorms Brick.

3.3 OpenPRS

OpenPRS heißt die Open Source Version von PRS (Procedural Reasoning Systems). PRS ist eine High-Level Kontroll- und Supervisionsprache angepasst für autonome Roboter, um Prozeduren, Skripten und Pläne in dynamischen Umgebungen zu repräsentieren und auszuführen. In einem Paper von (Ingrand et. al., 1996) wird beschrieben, dass PRS verwendet wird, um die Kontrolle und Supervision zu implementieren. Andere Komponenten, wie der Hight-Level Planer oder Low-Level Module, welche für den Zugriff und die Steuerung des Roboters zuständig sind, werden in dieser

Architektur verwendet und sind verbunden mit PRS. PRS implementiert verschiedenen Tools zum Repräsentieren von Aktionen bzw. Prozeduren und Mechanismen um Pläne, Skripten und Prozeduren auszuführen. OpenPRS folgt der BDI Architektur (Wooldridge, 2009), welche Vorstellungen über die Umgebung, Ziele des Benutzers und Vorhaben trennt. Durch Sensoren werden Informationen über die Umgebung eingeholt. Danach wird ermittelt, wie die Ziele mit Hilfe einer Planungsbibliothek und der aktuellen Informationen über die Umgebung erreicht werden können. Diese Pläne werden zum Schluss als Aktionen ausgeführt.

PRS besteht aus drei Hauptkomponenten. Die Datenbank beinhaltet Fakten, welche die Systemübersicht der Welt repräsentiert und welche konstant und automatisch aktualisiert wird, wenn ein neues Ereignis passiert. Die zweite Hauptkomponente ist eine Bibliothek von Plänen. Jeder Plan beschreibt eine Sequenz von Aktionen, welche ausgeführt werden können um gegebene Ziele zu erreichen oder um auf bestimmte Situationen zu reagieren. PRS plant nicht durch kombinierten von Aktionen, sondern durch Auswählen von alternativen Plänen. Das heißt die Bibliothek muss alle Pläne bereitstellen, welche benötigt werden, um Aufgaben auszuführen, für die der Roboter bestimmt ist. Die dritte Komponente, der Aufgaben-Graph ist ein dynamisches Set von Aufgaben, welche zurzeit ausgeführt werden. Ein Interpreter manipuliert diese drei Komponenten. Er empfängt neue Ereignisse und interne Ziele und wählt entsprechende Pläne basierend auf diese neuen Ereignissen, Zielen und Systeminformationen. Anschließend platziert er die Prozeduren dieser Pläne auf den Aufgaben-Graph und führt schließlich einen Schritt der aktiven Aufgabe aus.

3.4 PRODIGY

PRODIGY ist wie in einem Paper von (Veloso et. al., 1995) beschrieben wird, eine Architektur, welche Planen, sowie verschiedene Lernalgorithmen integriert hat. Die PRODIGY Architektur wurde anfangs als Künstliche Intelligenz konzipiert, um Ideen zu testen und entwickeln. Dabei übernimmt maschinelles Lernen die Funktion von Planen und Problemlösung. Um beim Lernen im Problemlösen messbare Performance-Steigerungen zu erreichen, wurde PRODIGY als Testumgebung für die ständige Kontrolle der Schleife zwischen Lernen und Performance in Planungssystemen entwickelt. Daher besteht PRODIGY im Kern aus einem All-Zweck-Planer und einigen Lernmodulen, welche sowohl das Wissen im Planungsbereich, als auch das Wissen um den Suchprozess zu leiten und zu kontrollieren, verbessert.

Der Planungsbereich im PRODIGY Planer ist hauptsächlich durch ein Set von Operatoren spezifiziert. Jeder Operator entspricht einer atomaren Planer-Aktion, welche hinsichtlich seiner Effekte und den notwendigen Vorbedingungen beschrieben werden. Die Vorbedingungen eines Operators wird repräsentiert durch einen logischen Ausdruck, welcher Konjunktionen,

Disjunktionen, Negationen und Quantoren enthält. Die Effekte eines Operators bestehen aus einer Liste von Prädikaten, welche hinzugefügt oder gelöscht werden, wenn der Operator verwendet wird und einem Set von konditionalen Effekten, welche abhängig von den einzelnen Bedingungen ausgeführt werden müssen. Ein Planungsproblem wird an PRODIGY als Anfangszustand der Welt und einer Ziel Behauptung, welche erreicht werden soll, übergeben. Der Planungsalgorithmus in PRODIGY ist vollständig.

3.5 Robotik im Unterricht

Die Kosten von Ausbildungsrobotern sind in den letzten Jahren immer mehr gesunken. Dank Firmen wie Lego sind programmierbare Maschinen auch als Spielzeug und im privaten Gebrauch leistbar geworden. Aus diesem Grund verwenden einige Schulen Roboter im Unterricht um Themen aus der Physik, der Mathematik oder der Informatik anhand dieser zu erklären. (Benitti, 2011) vergleicht zehn relevante wissenschaftliche Artikel, die sich mit Robotik im Unterricht beschäftigen. Die meisten dieser Arbeiten beziehen sich auf die Schulfächer Physik und Mathematik. Resultate zeigen, dass die Verwendung von Robotern das Verständnis von Konzepten in den Bereichen Wissenschaft, Technologie, Maschinenbau und Mathematik fördert. Obwohl in den meisten Fällen die Resultate der Artikel positiv waren, gibt es manche, die keinen Unterschied zwischen Schulklassen mit Robotereinsatz und Schulklassen ohne Robotereinsatz in Bezug auf Wissen festgestellt haben. Allerdings berichten alle Arbeiten eine Verbesserung der Qualifikationen Problemlösung und Logik sowie die Fähigkeit wissenschaftliche Nachforschungen anzustellen. Zudem werden durch die Verwendung von Robotern im Schulunterricht die Fähigkeiten Denkvermögen (Beobachtung, Schätzung, Manipulation) und Arbeiten im Teamwork gefördert.

In einer anderen Studie von (Lindh und Holgersson, 2005) wurde untersucht, ob das Arbeiten mit Legorobotern im Schulunterricht einen positiven oder einen negativen Einfluss auf die Fähigkeit Schüler in Bezug auf das Lösen mathematischer und logischer Probleme hat. Dazu wurden Kontrollgruppen und Testgruppen aus Schülern im Alter zwischen 11 und 16 Jahren gebildet. In den Testgruppen wurden Probleme der Mathematik anhand von Legorobotern erörtert, wogegen die Kontrollgruppen lediglich Frontalunterricht erhielten. Das quantitative Ergebnis dieser Studie war nicht signifikant genug, um Leistungsunterschiede zwischen den Testgruppen und den Kontrollgruppen feststellen zu können. Allerdings beobachteten (Lindh und Holgersson, 2005) auch einige qualitative Unterschiede. Schüler in der Testgruppe entwickelten verschiedenste Lernmethoden. Ein Weg für Schüler um zu Lernen ist die „trial-and-error“ Methode, bei der Schüler so lange verschiedene Dinge ausprobieren, bis sie Erfolg haben. Andere Schüler bevorzugen einen kooperativen Weg und diskutieren mit ihren Schulkollegen oder ihren Lehrer. Eher selten kommt es

vor, dass Kinder in Büchern und ähnliche Quellen nach Informationen suchen. Weiters entdeckten (Lindh und Holgersson, 2005), dass das gemeinsame Arbeiten mit Robotern die Teamfähigkeit fördert, was positive Rückmeldungen der Schüler brachte. Ebenso verbesserte sich das Verständnis der Schüler, wie komplexe Systeme mit Hilfe von Programmen gesteuert werden können.

4 Konzept

In diesem Kapitel wird die YAGI-Software-Architektur, die (Maier et al., 2015) erstellt haben veranschaulicht. Es werden die drei Schichten der Architektur beschrieben und wie die Kommunikation zwischen ihnen abläuft. In Abschnitt 4.2 wird definiert wie das Ev3 System-Interface aufgebaut ist. Dazu wird zuerst abgegrenzt was (Maier et al., 2015) in ihrer Arbeit erreicht haben und wo diese Arbeit in YAGI ansetzt. Danach wird diskutiert welche Methode sich eignet, um den YAGI-Interpreter auf dem Lego-Mindstorms-Ev3 auszuführen. Zum Schluss wird noch beschrieben, wie die Funktionen für den Roboter bereitgestellt werden und wie man diese erweitern kann.

4.1 YAGI-Software-Architektur

Wie (Maier et al., 2015) schon in der ursprünglichen Version von YAGI beschrieben haben, war das Ziel der YAGI-Software-Architektur eine klare Trennung von Syntax, Semantik und der Implementierung. Dabei ist es wichtig einfache Erweiterungen und Wartbarkeit zu gewährleisten. Um diese Ziele zu erreichen, besteht YAGI aus einer 3-tier Architektur, die in Abbildung 9 veranschaulicht wird. Diese Architektur erlaubt es YAGI einfach für verschiedene Zielsysteme adaptieren zu können.

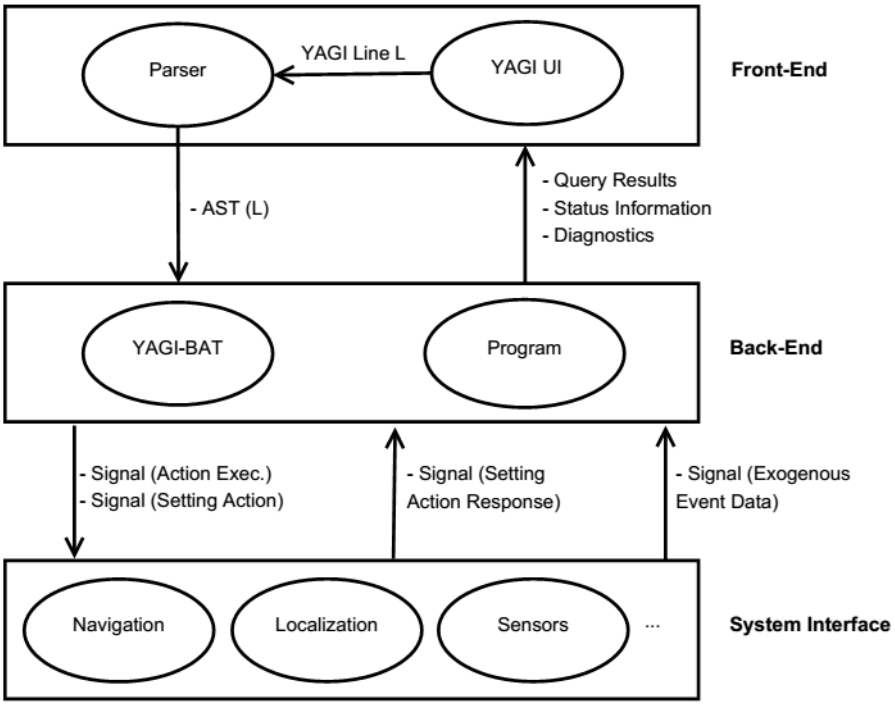

Abbildung 9: YAGI 3-Tier Architektur (Maier, 2015)

4.1.1 Front-End

Das Front-End enthält sowie das YAGI-User-Interface, als auch den Parser für den YAGI-Quellcode. Es handhabt den YAGI-Quellcode auf einem rein syntaktischen Level und ist dafür verantwortlich, die syntaktische Richtigkeit des YAGI-Quellcodes, der vom User eingegeben wurde, zu überprüfen. Weiters wandelt das Front-End den YAGI-Quellcode in eine geeignete Zwischenrepräsentation, den sogenannten „abstract Syntax Tree“, um, die es dem Back-End erlaubt, das YAGI Programm effizient zu verarbeiten (Aho et al., 2007). Das User-Interface kann von reinem konsolenbasierten User-Interface bis hin zum graphischen User-Interface variieren.

4.1.2 Back-End

Das Back-End konsumiert Eingaben vom Front-End, d.h. das Front-End liefert eine abstrakte Repräsentation des YAGI-Programms, den sogenannten „abstract Syntax Tree“, an das Back-End. Dieses speichert den aktuellen Status der Welt, d.h. die YAGI *Basic Action Theorie* $\mathrm{D}_{\mathrm{YAGI}}$, als auch ausführbare Programmelemente wie Prozeduren oder YAGI-Aktionen.

Dazu wird eine SQLite-Datenbank als relationales Datenbankmanagement verwendet. Durch dieses relationale Datenbankmanagement werden Fluents/Facts als Datenbanktabellen, ein Tupel als eine Zeile in der Tabelle und ein Set von Tupels als ein Set von allen Zeilen in der Tabelle dargestellt. Zu Beginn ist die Datenbank ohne Tabellen. Beim Ausführen einer Fluent/Fact Deklaration wird eine neue Tabelle in der Datenbank erstellt, welche das Fluent/Fact F repräsentiert. Der Zustand nach der Fluent/Fact Deklaration entspricht der Initialisierungssituation S_0. Um die Planungsaufgaben zu lösen, verwendet YAGI Progression, wie (De Giacomo und Palatta, 2000) bereits beschrieben haben. Roboter und Agenten können viele Aktionen ausführen und dadurch kann eine große Historik entstehen. Durch Progression wird in der Datenbank einfach die Initialsituation verändert.

Das Backend verarbeitet das YAGI-Programm auf einem semantischen Level. Es modifiziert die YAGI *Basic Action Theorie* und die YAGI-Programm-Elemente entsprechend der Semantik des gegebenen YAGI-Programms. Werden YAGI-Programm-Elemente ausgeführt, so geschieht dies online. Das heißt, das Back-End überprüft vor dem Ausführen einer Aktion die Vorbedingungen und sendet anschließend einen Signal-Befehl an das System-Interface um die Aktion schließlich auf dem Zielsystem auszuführen. Mit Hilfe des *search* Operators wird allerdings ein Teil des Programms vor dem Ausführen offline vom Back-End durchgeplant. Sobald ein gültiger Pfad im Programmcode gefunden wurde wird mit der Ausführung begonnen. Zuletzt antwortet das Back-End dem Front-End abhängig vom Type des YAGI-Statements.

4.1.3 System-Interface

Das System-Interface arbeitet auf der untersten Ebene der YAGI 3-Tier Architektur. Es ist dafür verantwortlich auf einen YAGI-Signal-Befehl, den es vom Back-End erhält, zu reagieren und eine entsprechende Aktion auszuführen. Das System-Interface verarbeitet YAGI-Signal-Befehle nicht parallel, d.h. es wartet bis eine Aktion ausgeführt wurde. Dadurch können Aktionen nicht simultan laufen. Es gibt zwei verschiedene Arten von Aktionen, Einstellungsaktionen und Abtastaktionen. Bei den Einstellaktionen werden Befehle an das autonome System gesendet, welche eine Handlung des autonomen Systems zufolge haben. Die Abtastaktionen hingegen bewirken, dass das autonome System Informationen, wie z.B. Sensordaten, an das Back-End zurücksendet. Weiters liefert das System-Interface externe Daten, (z.B. Daten, die durch eines exogenen Events erzeugt wurden). Das System-Interface variiert je nach Art der Verwendung. Da YAGI in dieser Arbeit auf einem Lego-Mindstorms-Ev3 zur Anwendung kommt, müssen entsprechende Methoden dafür bereitgestellt werden.

Zurzeit sind in YAGI drei verschiedene System-Interfaces implementiert. Eine rein konsolenbasiertes, welches die Signal-Befehle in der Konsole ausgibt und nur zu Testzwecke fungiert. Ein weiteres System-Interface wurde für ROS (Robot Operating System) geschrieben (Quigley et. al., 2009). ROS ist eine Sammlung von Tools, Bibliotheken und Konventionen, welches zum Ziel hat, ein komplexes und robustes Roboterverhalten auf verschiedenste Roboter-Plattformen zu schaffen. Ebenso existiert bereits ein System-Interface für ASRAEL (Abstract Simulator of Robotics AI for Education and Learning).

4.1.4 Kommunikation zwischen Front-End und Back-End

Das Front-End liefert YAGI-Code in Form eines abstrakten Syntax-Baumes zur Weiterverarbeitung an das Back-End. Diese Nachrichten variieren je nach Typ:

- **Fluent Anfrage:** Das Front-End kann Informationen über den aktuellen Status der Welt anfordern, d.h.: Status des Fluents. Das Back-End liefert dem Front-End ein Set von Tupels, für die der Fluent wahr ist, oder *false*, wenn das Fluent nicht definiert ist, zurück.
- **Programm Spezifikation:** Weiters kann das Front-End YAGI-Programme an das Back-End senden. Diese haben abgesehen von den Fluent/Fact Deklaration vorerst keinen Effekt, solange sie nicht ausgeführt werden. Das Back-End antwortet darauf mit *true* wenn das Programm gespeichert wurde, andernfalls mit *false*.
- **Programm Ausführung:** Die dritte Art von Nachrichten, die das Front-End dem Back-End übertragen kann, sind Befehle um eine Prozedur oder eine Aktion auszuführen. Das Back-

End retourniert darauf Informationen über die ausgeführte Aktion oder Prozedur wie Status Informationen, Daten vom Signal-Befehl einer YAGI Aktion oder eine Diagnose von Laufzeitfehlern.

4.1.5 Kommunikation zwischen Back-End und System-Interface

Die Kommunikation zwischen Back-End und System-Interface ist stringbasiert. Signal-Befehle des Back-Ends an das System-Interface können je nach Art des System-Interface als reiner Text, natürliche Sprache oder als ausführbarer Quellcode übertragen werden. Dies variiert je nach Implementation des System-Interface d.h. wie dieses den Inhalt der Nachrichten verarbeitet. Die gesendeten Signal-Befehle können Aktionen der realen Welt auslösen (z.B.: eine Bewegung des Roboters) oder Informationen der realen Welt abfragen (z.B.: Auslesen eines Sensorwertes). Je nach Art des Befehls antwortet das System-Interface mit Status-Informationen über den ausgeführten Befehl oder der realen Welt. Außerdem liefert das System-Interface dem Back-End Daten über exogene Ereignisse mit Hilfe von Call-Backs. Exogene Ereignisse sind ähnlich zu Aktionen mit einem external Attribut, mit Ausnahme dass sie nicht aktiv getriggert werden können, sondern durch ein externes Event ausgelöst werden. Dies geschieht asynchron, d.h. das Back-End speichert exogene Ereignisse, falls das System-Interface zurzeit eine Aktion ausführt. Die Werte des Ereignisses werden nach der Aktion vom Back-End konsumiert und die nächste Aktion ausgeführt.

4.2 Ev3 System-Interface

Die aktuelle Version von YAGI wurden nach der Softwarearchitektur, die oben beschrieben wurde, bereits von (Maier et al., 2015) implementiert. Allerdings wurde das System-Interface nur als Eingaben und Ausgaben von Text auf die Konsole realisiert. D.h. Aktionen die in YAGI aufgerufen wurden, sind nur als Text ausgegeben worden. Um die Aktionen von YAGI auf einem autonomen System, wie dem Lego-Mindstorms-Ev3, auszuführen, muss daher für dieses System ein eigenes System-Interface entworfen werden. Daher wurde in dieser Arbeit ein System-Interface für den Lego-Mindstorms-Ev3 für YAGI entworfen und implementiert. Das Ev3 System-Interface reagiert auf Signal-Befehl, die es vom Back-End bekommt. Ist der Befehl in der Befehlsliste des Ev3 System-Interface vorhanden, wird dieser durch eine entsprechende Aktion am Roboter ausgeführt. Nach erfolgreichem Ausführen des Befehls werden bei Sensing-Aktionen Daten an das Back-End zurückgesendet. Bei unbekannten Befehlen oder falschen Parametern antwortet das Ev3-System-Interface dem Back-End mit eine Fehlermeldung.

4.2.1 Methoden um C++ Programme auf dem Lego-Mindstorms-Ev3 auszuführen

Die aktuelle Version von YAGI wurde in C++ geschrieben, um möglichst plattformunabhängig zu sein. Die Standardsoftware auf dem Lego-Mindstorms-Ev3 ist ein Linux basiertes Betriebssystem mit einer Lego Distribution, welche von LEGO eigens für den Roboter entwickelt wurde. Da auf diesem vorinstallierten Betriebssystem keine C++ Compiler installiert ist, können auf dem Roboter keine C++ Programme kompiliert werden. Weiters gestattet die Lego Distribution keinen Remote-Zugriff per ssh, womit es nicht möglich ist Bibliotheken zu installieren. YAGI verwendet außerdem andere Bibliotheken, wie SQLite (Kreibich, 2010) oder ANTLR (Parr, 2013), welche ebenfalls eingerichtet werden müssen. Deshalb muss eine andere Methode gefunden werden um YAGI auf dem Lego-Mindstorms-Ev3 ausführen zu können. Die Vor- und Nachteile der möglichen Methoden werden anschließend kurz diskutiert.

4.2.1.1 BricxCC

Das BricxCC („Bricx Command Center“, 2016) ist ein 32-bit Windows-Programm, üblicherweise bekannt als „integrated development environment“ (IDE), um Lego-Mindstorms-Roboter aller Mindstorms Familien zu programmieren. Die möglichen Programmiersprachen variieren je nach Art des Lego-Mindstorms-Roboters. BricxCC ermöglicht das C-Programmieren des älteren Lego Mindstorms NXT mit Not eXactly C (NXC) oder Next Byte Codes (NBC). Der neuere Lego-Mindstorms-Ev3 hingegen kann mit Hilfe von cygwin und einer alternativen Firmware wie BrickOS mit den Standard-Programmiersprachen C, C++ oder Java programmiert werden. Da YAGI zu den standardmäßigen Bibliotheken auch speziellere benötigt, welche vorher installiert werden müssen, wurde diese Methode zum Programmieren ausgeschlossen.

4.2.1.2 Robotc

Robotc („ROBOTC.net“, 2016) ist eine Entwicklungsumgebung für C-Programme, die speziell Roboter unterstützt. Sie ist sowohl für den Lego-Mindstorms-NXT, als auch für den Lego-Mindstorms-Ev3 verwendbar. Robotc enthält eine graphische Programmierumgebung und einen Real-Time-Debugger. Mit Hilfe diesem Debuggers können die Zustände von Sensoren und Motoren zur Laufzeit bestimmt werden. Wenn die Software zum ersten mal gestartet wird, muss zuerst ein Firmware Update des Lego-Mindstorms gemacht werden, damit die Entwicklungsumgebung mit dem Roboter kommunizieren kann. Dadurch wird die original Firmware des Lego-Mindstorms überschrieben. Danach können C-Programme über Robotc auf den Lego Mindstorms gespielt und

ausgeführt werden.

4.2.1.3 ev3dev

Ev3dev ist ein vollwertiges Debian-Linux basiertes Betriebssystem, erstellt von (Lechner und Hempel, 2014), das auf Mindstorms EV3 Systemen läuft. Das Betriebssystem wird dabei auf eine SD-Karte kopiert, damit der Mindstorms EV3 von der SD-Karte aus booten kann. Der Vorteil dieser Methode ist, dass die Original-Firmware, die auf dem Lego-Mindstorms-EV3 installiert ist, im Gegensatz zur Robotc Entwicklungsumgebung nicht geändert werden muss. Da das Betriebssystem auf Debian-Linux aufgebaut ist, sind über 43.000 Softwarepakete, unter anderem diverse Entwicklungstools und Programmiersprachen, verfügbar. Des weiteren werden USB- und Bluetooth-Geräte erkannt und können verwendet werden. Dadurch kann sich der Lego-Mindstorms-Ev3 über USB-Kabel oder über einen USB-WIFI-Stick mit dem Internet verbinden. Ev3dev verfügt über Treiber, mit denen auf Sensoren, Motoren, Soundsystem, LCD, Knöpfe und LEDs des Mindstorms EV3 zugegriffen werden kann. Diese Zugriffe basieren auf dem Lesen und Schreiben von Dateisystemen. Ein weiterer Vorteil von ev3dev ist, dass man fast alle Programmiersprachen verwenden kann, die auch auf anderen Linux Distributionen laufen. Für die populärsten Programmiersprachen, wie C, C++, Java oder Python, gibt es bereits eine Einbindung der Mindstorms EV3 Hardware (Lechner und Hempel, 2014). Da diese Methode alle Anforderungen entspricht, wurde beschlossen ein ev3dev Betriebssystem für den Lego-Mindstorms-Ev3 in Verbindung mit YAGI zu verwenden.

4.2.2 Auswahl der Ev3-Aktionen

Um zu entscheiden welche Aktionen im Ev3-System-Interface implementiert werden sollen, muss zuerst die Frage beantwortet werden, wie der Roboter aufgebaut ist. Es besteht die Möglichkeit den Lego-Roboter nach eigenen Wünschen zu gestalten und mit verschiedener Peripherie zu bestücken. Diese Arbeit bezieht sich jedoch auf einen Standard-Aufbau des Roboters, welcher einen differential Antrieb zugrunde liegt. Der Aufbau des Roboters wird in Abbildung 10 dargestellt. Beliebige Sensoren können dann je nach Bedarf an einer beliebigen Stelle des Roboters platziert werden. Durch den differential Antrieb kann der Roboter bei gleichen Geschwindigkeiten der beiden Räder vorwärts bzw. rückwärts fahren. Bei ungleichen Geschwindigkeiten der Räder fährt der Lego-Mindstorms-Ev3 eine Kurve. Wichtig dabei ist, dass die beiden Räder gleichzeitig angesteuert werden können und bei Bedarf mit unterschiedlichen Parametern. Da YAGI im Gegensatz zu Indigolog noch keine Interrupts bzw. parallelen Aktionen implementiert hat, muss das Ev3-System-Interface eine Aktion bereitstellen, welche beide Motoren gleichzeitig angesteuert.

Weiters werden Aktionen benötigt, um Sensoren zu verwalten. Die meisten Sensoren haben verschiedene Modi, welche unterschiedliche Werte der Sensoren liefern. Daher muss eine Aktion zum Wechseln der Modi und eine weitere Aktion zum Auslesen der Sensorwerte implementiert werden.

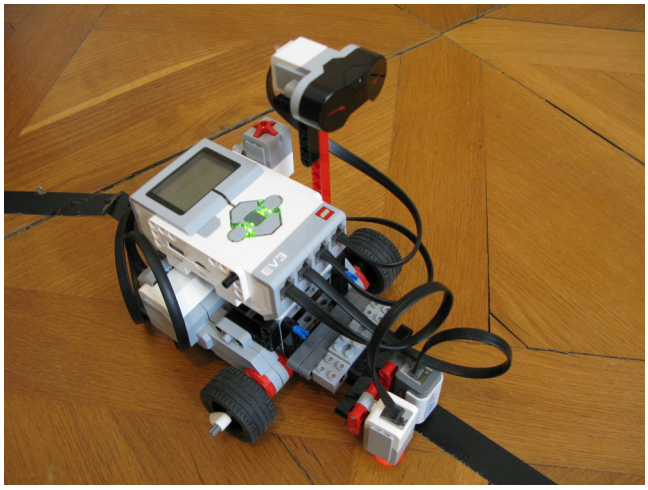

Abbildung 10: Standard Roboter mit differential Antrieb und einigen Sensoren

Da es ein Ziel dieser Arbeit ist, den Lego-Mindstorms-Ev3 auch für Schulklassen als Übungszwecke bereitzustellen, kommen auch aus diesem Bereich weitere Anforderungen. Eine Basisfunktionalität, welche zum Beispiel auch im RoboCupJunior (Eguchi et. al., 2011) verwendet wird, ist der Linienfolger. Beim Linienfolger wird ein Farb- oder Lichtsensor verwendet um einer am Boden markierten Linie zu folgen. Diese Funktion kann verwendet werden, um den Roboter zu einer bestimmten Stelle im Raum bzw. in einen anderen Raum zu bewegen oder um Schüler die Suche in einem Graph näher zu bringen. Eine weitere Funktion, die für Übungszwecke verwendet werden kann ist die Ausgabe eines Signals. Durch das Ertönen eines Signaltons kann eine erfolgreiche Graph-Suche signalisiert oder andere wichtige Ereignisse angekündigt werden. Die LEDs auf dem Roboter können dazu ebenfalls verwendet werden.

4.2.3 Laufzeitprobleme

In den Zielen dieser Arbeit wurde beschrieben, dass die Befehle für den Lego-Mindstorms-Ev3 möglichst atomar sein sollten. Die Idee hinter diesem Ansatz ist, dass komplexere Funktionen in der Programmiersprache YAGI mit Hilfe der atomaren Befehlen einfach erstellt werden können. Das Ausführen der ersten Testprogramme von YAGI auf dem Lego-Mindstorms-Ev3 haben jedoch gezeigt, dass eine Zeitverzögerung zwischen der Ausführung einer Aktion in YAGI und der tatsächlichen Realisierung am Roboter stattfindet. Diese tritt zum einen dadurch auf, dass in YAGI das Programm als abstrakten Syntax Baum in der Datenbank gespeichert wird. Beim Ausführen

einer Aktion werden benötigte Zustandsinformationen für Vorbedingungen aus der Datenbank geladen. Zum anderen werden die Veränderungen in der realen Welt, die durch die Aktion geschehen sind, wieder in die Datenbank gespeichert. Da der Ev3-Brick nur 64MB RAM hat, benötigen diese Datenbankzugriffe eine gewissen Zeit.

Aus diesem Grund wurde beschlossen, dass Funktionen, die Sensorik und Aktorik des Roboters benötigen, direkt am Lego-Mindstorms-Ev3 laufen und als atomare Aktionen in YAGI bereitgestellt werden. Dies stellt sicher, dass die Funktionen nicht durch eine Verzögerung in dem Schleifendurchgang fehlerhaft werden. Diese Entscheidung wird dadurch unterstützt, dass auch in den klassischen Kontroll-Architekturen reaktive Schichten verwendet werden. (Murphy, 2000) erklärt in seinem Buch die Vorteile von Deliberation-Reaktion-Paradigmen. In der Deliberation-Schicht werden die nachfolgenden Aktionen zuerst durchgeplant bevor diese anschließend ausgeführt werden. Während die Reaktion-Schicht direkt auf die Sensordaten reagiert und danach handelt. Dadurch können wichtige Reaktionen schnell abgewickelt werden.

4.2.4 Erweiterbarkeit

Wie im vorherigem Abschnitt bereits beschrieben wurde, sollten komplexere Funktionen direkt auf dem Lego-Mindstorms-Ev3 laufen. Dazu müssen unter Umständen neue Befehle für das Ev3-System-Interface implementiert werden. Um für diese Implementation nicht zu viele Änderungen im YAGI-Code durchführen zu müssen, wurde im Ev3-System-Interface auf eine einfache Erweiterbarkeit geachtet. Deshalb wurde die Methode *registerFunction* im *Ev3SignalHandler* bereitgestellt, durch welche Funktionen als Ev3-Befehle registriert werden. Diese Methode erwartet als Parameter einen Funktionspointer auf die entsprechende Funktion. Für jede neue Funktion muss daher nur einmal die *registerFunction* Methode aufgerufen werden. Details zum Ev3SignalHandler siehe Abschnitt 5.2.

5 Implementierung

In diesem Kapitel wird erklärt, wie das Ev3 System-Interface für YAGI implementiert wurde. Zuerst wird in Abschnitt 5.1 beschrieben welche Voraussetzungen erfüllt werden müssen, um den YAGI-Interpreter auf einer Plattform starten zu können. Abschnitt 5.2 gibt einen Überblick der Klassen, die für das System-Interface verwendet wurden. In Abschnitt 5.3 wird darauf eingegangen, wie das ev3dev-Betriebssystem Geräte verwaltet und wie YAGI auf die Peripherie zugreifen kann. In den darauf folgenden Abschnitten wird die Implementierung der Motoren, Sensoren, LEDs und des Soundsystems erläutert. Zum Schluss dieses Kapitels wird im Abschnitt 5.8 erklärt, wie ein Linienfolger auf dem Lego-Mindstorms-Ev3 und der dazugehörige PID-Regler realisiert wurde.

5.1 Technische Voraussetzung zur Implementation von YAGI

Eine wichtige Eigenschaft der Programmiersprache YAGI ist, das sie plattformunabhängig ist. Um den YAGI-Interpreter auszuführen, müssen trotzdem einige Voraussetzungen erfüllt werden. Da YAGI in C++ implementiert wurde, muss eine aktuelle C++ Bibliothek vorhanden sein. Des weiteren verwendet YAGI das Antlr Framework, welches von (Parr, 2013) entwickelt wurde, um den YAGI Quellcode lexikalisch zu analysieren und einen abstrakten Syntax Baum zu generieren. Auch eine SQLite-Datenbank muss auf der Plattform zur Verfügung stehen, da YAGI ein relationales Datenbank Management verwendet, um den YAGI Status der Welt zu repräsentieren. Es werden noch weitere Bibliotheken in YAGI verwendet, auf welche, in dieser Arbeit, nicht näher eingegangen wird. Die benötigten Bibliotheken, welche zusätzlich auf dem ev3dev Betriebssystem installiert wurden, werden in Tabelle 1 aufgelistet.

Linux Bibliotheken
C++
Antlr3c-3.4
Libsqlite3-dev
Pkg-config
Libreadline-dev
Libncurses-dev

Tabelle 1: benötigte Bibliotheken, um YAGI auf dem ev3dev Betriebssystem zu starten

5.2 Implementierung Ev3-System-Interface

Das Ev3-System-Interface ist die Schnittstelle zwischen dem YAGI Back-End und der Hardware des Lego-Mindstorms-Ev3. Es stellt alle Aktionen zur Verfügung, die der Roboter ausführen kann. Wird ein Signal-Befehl von dem Back-End an das Ev3-System-Interface gesendet, so wird zuerst überprüft, ob dieser einer registrierten Aktion zugeordnet ist. Ist dies der Fall, so wird die entsprechende Aktion ausgeführt, anderenfalls wird eine Fehlermeldung zurückgesendet. Nach erfolgreicher Ausführung einer Aktion werden externen Daten, falls diese durch die Aktion entstanden sind, an das Back-End zurück geliefert. Die dazu verwendeten Klassen und ihre Aufgaben werden im Klassendiagramm in Abbildung 11 abgebildet und im Folgenden kurz erklärt:

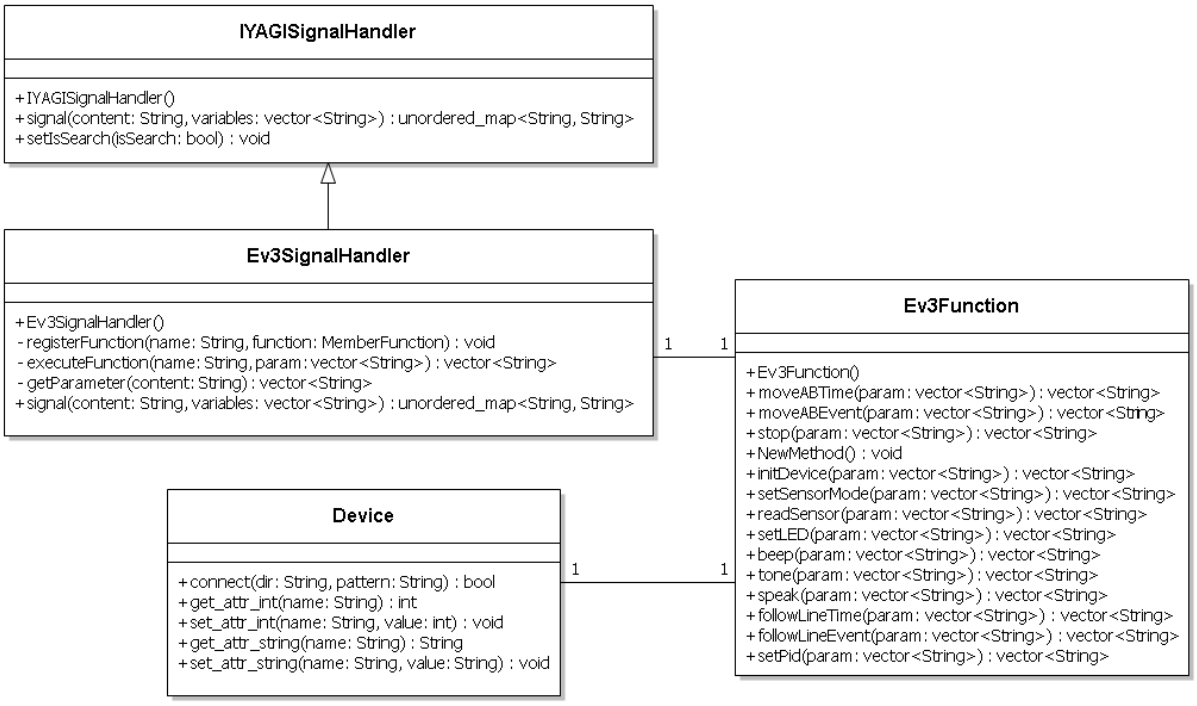

Abbildung 11: Vereinfachtes Klassendiagramm des Ev3-System-Interfaces

- **Ev3SignalHandler:** Die *Ev3SignalHandler* Klasse ist vom IYAGISignalHandler abgeleitet und ist dafür verantwortlich die Signal-Befehle, die es vom Back-End erhält, zu verarbeiten und die entsprechende Aktion des Roboters auszuführen. Zuerst werden im Konstruktor dieser Klasse alle Aktionen registriert, welche durch den Lego-Mindstorms-Ev3 ausgeführt werden können. Dazu wird die Funktion *registerFunction* verwendet, welcher der Name der Aktion und ein Funktionspointer auf die entsprechende Methode übergeben wird. Dies garantiert eine einfache Erweiterbarkeit der Aktionen. Sobald ein Signal-Befehl vom Back-End gesendet wird, muss zuerst der Aktionsname und die dazugehörigen Parameter herausgelesen werden, da die Befehle als reine String-Kommunikation übertragen werden.

Danach wird überprüft, ob der Aktionsname einer registrierten Aktion zugeordnet werden kann und ob die übergebenen Parameter gültig sind. Nach erfolgreicher Validierung, ruft der *Ev3SignalHandler* die entsprechende Aktion mit Hilfe des Funktionspointers auf und wartet bis diese ausgeführt wird. Gegebenenfalls werden externe Daten, wie z.B. Sensorwerte, an das Back-End zurückgesendet. Diese werden zuvor wieder in eine entsprechende String-Repräsentation umgewandelt. Der genaue Ablauf, welcher im Ev3-System-Interface beim Ausführen einer Aktion stattfindet, wird in einem Kontrollflussdiagramm in Abbildung 12 dargestellt.

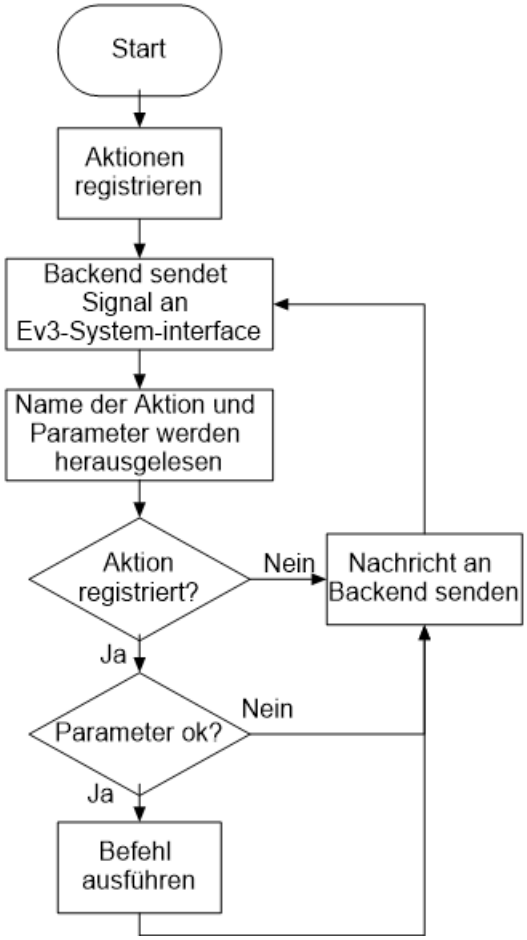

Abbildung 12: Kontrollflussdiagramm des Ev3-System-Interfaces

- **Ev3Function:** Die *Ev3Function* Klasse stellt alle Funktionen zur Verfügung, welche im YAGI Programm für den Lego-Mindstorms-Ev3 ausgeführt werden können. Nach Bedarf können neue Funktionen für den Lego-Mindstorms-Ev3 in dieser Klasse hinzugefügt und im *Ev3SignalHandler* registriert werden. Die Ev3Function Klasse hält alle Instanzen der Peripherie Klassen. Um auf die Peripherie des Roboters zugreifen zu können, wird die *Device* Klasse verwendet.
- **Device:** Die *Device* Klasse ist dafür verantwortlich mit der Peripherie des Lego-

Mindstorms-Ev3 zu kommunizieren. Werden neue Geräte an den Roboter angehängt, so werden diese, falls ein entsprechender Treiber installiert ist, automatisch erkannt und als ein Filesystem repräsentiert. Die *Device* Klasse kann über dieses Filesystem Befehle an die Peripherie schicken, beziehungsweise Daten lesen. Die Geräte können dabei entweder durch den Namen oder durch den Port, an den sie angehängt sind, dem entsprechenden Filesystem zugeordnet werden. Alle Sensoren, Motoren und die LEDs sind von der Device Klasse abgeleitet, wodurch diese angesteuert werden können. Details zur Ansteuerung der Peripherie siehe Abschnitt 5.3.

5.3 Ansteuerung der Peripherie

Im ev3dev Betriebssystem sind schon diverse Treiber installiert, mit deren Hilfe die meisten externen Geräte wie Motoren und Sensoren automatisch erkannt werden. Über die dazugehörigen Treiber wird jedes Gerät durch ein entsprechendes Filesystem repräsentiert. Tacho-Motoren registrieren sich beispielsweise in /sys/class/tach-motor/, Sensoren in /sys/class/lego-sensor/ und LEDs in /sys/class/leds/. Daher wurde die Klasse *Device* erstellt, welche über dieses Filesystem mit der Peripherie kommuniziert. Weiters werden für diese Kommunikation noch die Klassen *Motor, Sensor* und *Led* benötigt, welche von der Klasse *Device* abgeleitet sind. Aus dem Filesystemen können mit Hilfe der Klasse *Device* Attribute zu den Geräten, wie Name, Port oder Arten der Modi ausgelesen werden. Ebenso können Befehle an Motoren, Sensoren und LEDs geschickt werden, wodurch Sensoren in verschiedene Modi versetzt oder Motoren gesteuert werden. Auch das Auslesen der Sensordaten und des Motorstatus erfolgt mit Hilfe des Filesystems. Um verschiedene Sensoren des gleichen Typs zu erkennen, werden diese immer über den Port-Name angesteuert. Damit YAGI die entsprechenden Sensoren erkennt, muss beim Ausführen eines YAGI-Programms für jeden Sensor die Aktion *setup(sensor_name, port)* aufgerufen werden. Da wir in dieser Arbeit einen Differentialantrieb beim Lego-Mindstorms-Ev3 voraussetzen, werden bei den Motoren zurzeit die Ports *outA* und *outB* gemeinsam angesprochen, um dem Roboter das Fahren zu ermöglichen. Auf dem Lego-Mindstorms-Ev3 ist auch ein Soundsystem integriert, mit dessen Hilfe entsprechende Signale auf dem Lautsprecher ausgegeben werden können. Eine Übersicht zur Peripherie wird im vereinfachten Klassendiagramm in Abbildung 13 dargestellt. Wie in dem Klassendiagramm ersichtlich ist, sind die Klassen Motor, Sensor und Led von der Device Klasse abgeleitet. Weiters sind noch etliche abgeleitete Sensor Klassen implementiert, welche bei Bedarf erweitert werden können. Die Sound Klasse ist eigenständig, da die Lautsprecher keine Lego Peripherie sind, sondern direkt auf dem Mikrocontroller implementiert sind und deshalb mit Hilfe eines Linux Treibers angesteuert werden können.

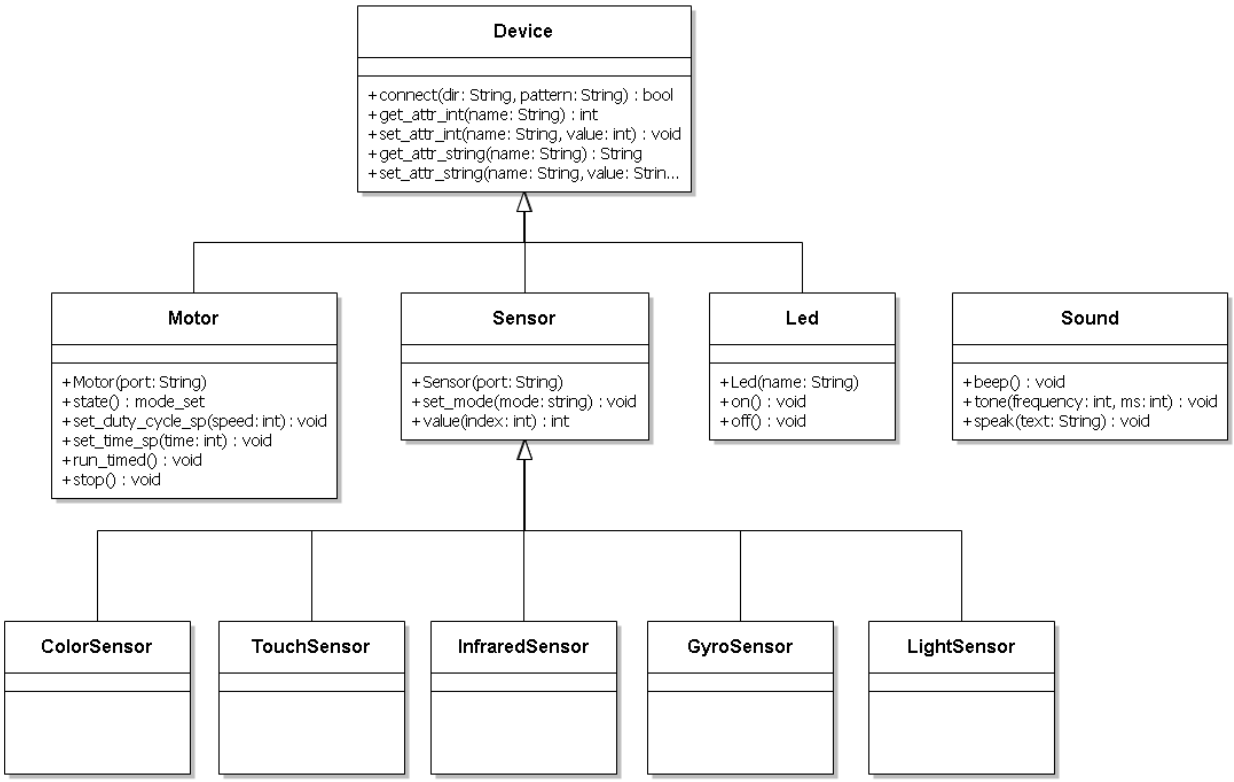

Abbildung 13: Vereinfachtes Klassendiagramm der Peripherie

5.4 Implementierung der Motoren

Der Lego-Mindstorms-Ev3 besitzt vier Ausgangsports an denen Motoren angehängt werden können, *outA*, *outB*, *outC* und *outD*. YAGI unterstützt zurzeit nur die Ports *outA* und *outB*, da in der Grundausstattung des Roboters nur zwei Motoren mitgeliefert werden. Diese zwei Motoren können gleichzeitig mit nur einer Aktion angesteuert werden. Dies ist notwendig, da YAGI im Gegensatz zu Indigolog noch keine Interrupts implementiert hat. Das heißt YAGI wartet solange, bis eine entsprechende Aktion abgeschlossen ist. Dadurch können zwei Aktionen nicht gleichzeitig stattfinden.

5.4.1 Aktionen

Die Aktionen für die Motoren sind in der Ev3Function Klasse implementiert, welche auch die Instanzen der Motor Klasse hält. Es folgt ein kurzer Überblick der Aktionen, die in YAGI für die Motoren implementiert wurden und die YAGI Beispiele dazu werden in Listing 6 abgebildet. Die in der Listing abgebildeten Aktionen sind YAGI Aktionen, welche mit Hilfe des Signal-Befehls dem System-Interface mitteilen, dass eine Low-Level Aktion aufgerufen werden soll. Die Vorbedingungen und die Effekte der YAGI Aktion werden allerdings schon im Back-End behandelt.

- **move_ab_time:** Durch diese Aktion werden die Motoren, die an den Ports *outA* und *outB* angeschlossen sind, angesteuert. Die *move_ab_time(v1, v2, time)* Funktion benötigt drei Parameter. Die ersten beiden Parameter (*v1* und *v2)* geben an, mit welcher Geschwindigkeit sich die Motoren an den entsprechenden Ports drehen sollen. Der Wertebereich dieser Parameter liegt zwischen 0 (der Motor steht still) und 100 (maximale Geschwindigkeit des Motors). Sind beide positiv, bewegt sich der Lego-Mindstorms-Ev3 vorwärts, bei negativen Geschwindigkeiten rückwärts. Wenn die Geschwindigkeiten gleich groß sind (*v1 = v2*), dann fährt der Roboter gerade, sonst einen entsprechenden Kreisbogen. Der dritte und letzte Parameter dieser Funktion ist die Zeit, wie lang der Roboter sich bewegt, in Millisekunden. Wird bei der Zeit der Wert 0 übergeben, so fährt der Lego-Mindstorms-Ev3 solange weiter, bis ein *stop()* – Befehl geschickt wird. An dieser Stelle sollte erwähnt werden, dass das System-Interface immer wartet bis eine Aktion abgeschlossen ist. In diesem spezielle Fall, werden jedoch nur die Motoren gestartet und damit ist die Aktion beendet. Damit wird ein Zusammenarbeiten von Motoren und Sensoren in YAGI Programmen ermöglicht, welches wie in Abschnitt 4.2.3 beschrieben, nicht sinnvoll ist.
- **move_ab_event:** Diese Aktion ist ebenso für die Ansteuerung der Motoren an Port *outA* und Port *outB* zuständig und erwartet drei Parameter. Die ersten beiden Parameter beschreiben die entsprechende Geschwindigkeit, mit der die beiden Motoren laufen. Mit dem dritten Parameter wird ein Eingangsport eines Sensors übergeben. Wird diese Funktion aufgerufen, bewegt sich der Roboter so lange, bis sich der Wert dieses Sensors verändert. Hängt zum Beispiel am Eingangsport ein Farbsensor, so fährt der Lego-Mindstorms-Ev3 bis der Sensor eine andere Farbe detektiert. Diese Aktion ist, wie schon im Abschnitt 4.2.3 erklärt wurde, eine klassenübergreifende Aktion, welche sowohl die Motor Klasse, als auch die Sensor Klasse verwendet. Daher wird auch die Sensoraktion *readSensor(port, vlaue)* verwendet.
- **stop:** Mit Hilfe dieser Aktion werden alle Motoren, die am Roboter angeschlossen sind, gestoppt. Man beachte, dass in YAGI keine zwei Aktionen gleichzeitig ausgeführt werden können. Daher ist dieser Befehl nur sinnvoll, wenn zuvor die Aktion *move_ab_time(v1, v2, time)* mit dem Parameter Zeit = 0 ausgeführt wurde.

```
// make a 180° rotation (North or South)
action turn($dir)
precondition:
  // ev3 do not look to this direction
  not(<$dir> in direction)
effect:
  // now the ev3 look in this direction
  direction = {<dir>};
signal:
  "move_ab_time(30, -30, 3000)";
end action

// stop the ev3
action stop()
effect:
  // now the ev3 is not driving
  driving = {<"FALSE">};
signal:
  "stop()";
end action
```

Listing 6: YAGI Beispiele der Aktionen für die Motoren

5.5 Implementierung der Sensoren

Das Ev3-System-Interface unterstützt alle Sensoren, die für den Lego-Mindstorms-Ev3 bereitgestellt werden. Auch die meisten Sensoren des älteren Lego Mindstorms NXT können im Ev3-System-Interface verwendet werden. Allerdings wurden nicht alle Sensoren getestet, da einige in diesem Projekt nicht zur Verfügung stehen. Um spezielle Sensoren verwenden zu können, muss ein entsprechender Treiber auf dem ev3dev Betriebssystem installiert sein, welcher den Sensor automatisch erkennt. Anschließend muss der Sensor noch in der *Ev3Dev* Klasse ergänzt werden. YAGI Beispiele zu den Sensor-Aktionen werden in Listing 7 gezeigt. Die *setup* Aktion in dem Beispiel ist für die Registrierung des Sensors zuständig. Mit Hilfe der *set_mode* Aktion wird der Sensor in einen anderen Modus versetzt. In der *holdObject* Aktion wird überprüft ob der Berührungssensor gedrückt wurde. Eine gedrückte Taste signalisiert das Halten eines Objektes. Wird die Taste losgelassen, so wird dies als Ablegen des Objektes interpretiert.

5.5.1 Setup der Sensoren

Wie schon zuvor erwähnt, werden die angeschlossenen Sensoren automatisch erkannt und als Filesystem repräsentiert. Um diese jedoch in YAGI verwenden zu können, müssen die Sensoren zuerst registriert werden. Durch die Registrierung wird eine Instanz der entsprechenden Sensorklasse erstellt, welche in der Ev3Function Klasse gehalten wird. Dazu wird die YAGI Aktion *setup(sensor_name, port)* verwendet. Danach kann der entsprechende Sensor mit Hilfe des Ports angesprochen werden. Einige Sensoren verfügen über verschiedene Modi, welche unterschiedliche

Rückgabewerte liefern. Um einen Sensor in einen bestimmten Modus zu versetzen, wird die Aktion *set_mode(port, mode)* bereitgestellt. Ein Überblick zu den verschiedenen Modi der Sensoren, die getestet wurden, wird in Tabelle 2 dargestellt.

Name	**Modus**	**Werte**
Ev3 Color Sensor	COL-REFLECT	Wert0: Intensität des reflektierten Lichtes (0 bis 100)
	COL-AMBIENT	Wert0: Intensität des Umgebungslichtes (0 bis 100)
	COL-COLOR	Wert0: Erkannte Farbe (0 bis 7)
Ev3 Touch Sensor	TOUCH	Wert0: Status (0 oder 1)
Ev3 Infrarot Sensor	IR-PROX	Wert0: Distanz (0 bis 100)
	IR-SEEK	Wert0: Kanal 1 Richtung (-25 bis 25)
		Wert1: Kanal 1 Distanz (0 bis 100)
	IR-REMOTE	Wert0: Kanal 1 (0 bis 11)
Ev3 Gyro Sensor	GYRO-ANG	Wert0: Winkel (-32768 bis 32767)
	GYRO-RATE	Wert0: Rotationsgeschwindigkeit (-440 bis 440)
NXT Light Sensor	REFLECT	Wert0: Intensität des reflektierten Lichtes (0 bis 1000)
	AMBIENT	Wert0: Intensität des Umgebungslichtes (0 bis 1000)

Tabelle 2: Modi der Sensoren

5.5.2 Format der Rückgabewerte

Um Sensorwerte auszulesen wird die Aktion *read_sensor(port, value)* bereitgestellt. Mit dem ersten Parameter wird angegeben an welchem Port der entsprechende Sensor hängt. Der zweite Parameter entscheidet, welcher Wert des Sensors gelesen wird. Die meisten Sensoren liefern nur einen Wert zurück, daher ist dieser Parameter meistens 0. Es gibt jedoch Sensoren, wie zum Beispiel den Infrarot Sensor, bei dem mehrere Werte ausgelesen werden können. Da die Kommunikation in YAGI zwischen System-Interface und Back-End stringbasiert ist, werden die Rückgabewerte auf fünf Stellen mit führenden 0 aufgefüllt. Der Rückgabewert 25 wird zum Beispiel durch den String 00025 dargestellt. Dies erleichtert das Ausführen von Vergleichsoperationen mit den Rückgabewerten.

```
// initialize the touch sensor on port in3
action setup()
signal:
  "setup(EV3 touch, in3)";
end action

// set the mode of the color sensor on port in1 to COLOR MODE
action set_mode()
signal:
  "set_mode(in1, COL-COLOR)";
end action

// check if ev3 is holding an object represented by pressing a button
action holdObjekt() external ($value)
effect:
  if(<"000">in value) then
    object -={<"o">};
  else
    object +={<"o">};
  end if
signal:
  "read_sensor(in3)";
end action
```

Listing 7: YAGI Beispiele der Aktionen für die Sensoren

5.6 Implementierung der LEDs

Auf dem Lego-Mindstorms-Ev3 befinden sich vier LEDs, die angesteuert werden können, zwei rote und zwei grüne LEDs. Dafür wurde die Aktion *set_led(name, value)* implementiert. Mit Hilfe des ersten Parameters wird angegeben welche der vier LEDs, *red_right*, *red_left*, *green_right* oder *green_left*, angesprochen werden soll. Der zweite Parameter bestimmt die Intensität des Lichtes, wobei der Wert 0 bedeutet, dass die LED ausgeschaltet wird. Der maximale Wert der übergeben werden kann ist 255, dabei leuchtet die LED in voller Stärke. Die Verwendung dieser Aktion wird in Listing 8 gezeigt.

```
  action red_led_on()
  signal:
    "set_led(red_right, 255)";
  end action

  action red_led_off()
  signal:
    "set_led(red_right, 0)";
  end action
```

Listing 8: YAGI Beispiele der Aktionen für die LEDs

5.7 Implementierung des Soundsystems

Der Lego-Mindstorms-Ev3 hat einen simplen Lautsprecher auf dem Ev3-Brick implementiert,

welcher für akustische Signale verwendet werden kann. Um das Soundsystem auch für YAGI nutzen zu können, wurden folgende Aktionen bereitgestellt und Beispiele dazu in Listing 9 angegeben:

- **beep:** Diese Aktion gibt einen Ton mit 1000Hz 100ms lang aus. Dazu wird keine Parameter benötigt. Sie kann zum Signalisieren bestimmter Ereignisse verwendet werden.
- **tone:** Mit Hilfe dieser Aktion wird ein bestimmter Ton ausgegeben. Die *tone(frequency, time)* Funktion erwartet zwei Parameter. Der erste Parameter gibt an, welche Frequenz in Hertz der ausgegebene Ton hat. Der zweite Parameter bestimmt die Zeit wie lange der Ton erklingt, in Millisekunden.
- **speak:** Das Soundsystem des Lego-Mindstorms-Ev3 ermöglicht es, dass der Roboter ganze Sätze sprechen kann. Dazu wurde die Aktion *speak(text)* implementiert. Der Roboter spricht den Text, der dieser Funktion übergeben wurde. Da die Sprache des Lego-Mindstorms-Ev3 auf Englisch eingerichtet wurde, sollten nur englische Texte angegeben werden.

```
action beep()
signal:
  "beep()";
end action

action tone()
signal:
  "tone(500, 1000)";
end action

action speak()
signal:
  "speak(Hello!)";
end action
```

Listing 9: YAGI Beispiele der Aktionen des Soundsystems

5.8 Sonstige Aktionen

Ein Ziel dieser Arbeit ist es, die Befehle für den Lego-Mindstorms-Ev3 möglichst atomar zu implementieren, da komplexere Befehle mit Hilfe von simplen Befehlen nachgebildet werden können. Daher wären die grundlegenden Aktionen der Peripherie, wie sie bisher beschrieben wurden, ausreichend. Da es aber eine gewisse Zeit dauert, um die einzelnen YAGI Aktionen auf dem Lego-Mindstorms-Ev3 auszuführen, ist es sinnvoller Funktionen, in der Sensorik und Aktorik in einander greifen, in eigene Aktionen zu packen. Das heißt kontinuierliche Funktionen für den Lego-Mindstorms-Ev3 werden abstrahiert und in eigene Aktionen gepackt. Diese können somit schneller laufen. Da aber gewartet wird bis die Aktion ausgeführt wurde, kann es passieren, dass eine Aktion wegen fehlerhaften Abbruchbedingungen ewig läuft und dadurch das Programm

blockiert.

5.8.1 Linienfolger

Eine weitere Aktion, die in YAGI für den Lego-Mindstorms-Ev3 implementiert wurde, ist der Linienfolger. Bei dieser Aktion wird ein Farb- oder Lichtsensor des Roboters verwendet, um einer Linie auf dem Boden zu folgen. Wichtig dabei ist, dass der Farbunterschied zwischen Linie und Boden möglichst groß ist. Im Ausgangszustand sollte der Roboter so stehen, dass sich der Farb- oder Lichtsensor direkt über der Linie befindet und der Roboter in die gewünschte Fahrtrichtung ausgerichtet ist. Zu Beginn dieser Aktion schwenkt der Roboter hin und her, um zu erkennen welcher Farb- oder Lichtwert dem Boden (r_{floor}) und welcher der Linie (r_{line}) zugeordnet ist. Dadurch lässt sich berechnen bei welchen Wert der Rand der Linie beginnt:

$$r_{edge} = \frac{r_{floor} + r_{line}}{2}$$

Als nächstes sucht der Lego-Mindstorms-Ev3 den rechten Rand der Linie. Dazu schwenkt dieser zuerst nach rechts, bis der Sensorwert r_{floor} entspricht. Dann dreht sich der Roboter langsam nach links, bis der Sensorwert r_{edge} erreicht wird. Nun fährt der Roboter geradeaus, während laufend die Sensorwerte ermittelt werden. Bei Werten zwischen r_{edge} und r_{floor} lenkt der Lego-Mindstorms-Ev3 nach links um wieder an die Linie zurück zu kommen. Sind die Sensorwerte jedoch zwischen r_{edge} und r_{line} , so macht der Roboter eine entsprechende Rechtskurve. Dieser Regelkreis wird durch einen PID Regler erreicht, welcher in Abschnitt 5.8.2 genauer beschrieben wird. Durch diese Vorgehensweise tastet sich der Roboter am rechten Rand der Linie entlang, bis das Abbruchkriterium erreicht wird. Dazu werden in YAGI zwei Methoden bereitgestellt. In der Aktion *follow_line_time(port, time)* folgt der Roboter einer Linie, bis die übergebene Zeit verstrichen ist. Mit Hilfe des Parameters *port* wird angegeben, an welchem Port der Sensor hängt, der für die Aktion verwendet werden soll. Die zweite Aktion, die einen Linienfolger integriert hat, ist *follow_line_event(line port, termination port)*. Der erste Parameter dieser Funktion gibt wieder an, an welchen Port der Sensor hängt, der zum Folgen der Linie verwendet wird. Der zweite Parameter beschreibt den Port des Sensors, welcher als Abbruchkriterium verwendet wird, d.h. der Roboter folgt so lange einer Linie, bis sich der Sensorwert dieses Sensors verändert. Eine mögliche Verwendung dieser Aktion in YAGI wird in Listing 10 gezeigt.

```
// ev3 drive one of the two rooms, using the line follower
action drive_to_room($room)
precondition:
  // ev3 is not in this room
  not(<$room> in location);
effect:
  // now the ev3 is in the room
  location = {<$room>};
  // turn around the Ev3
  turn();
signal:
  "follow_line_event(in1,in2)";
end action
```

Listing 10: YAGI Beispiel des Linienfolgers

In der nachfolgenden Abbildung wird ein Sequenzdiagramm präsentiert, welches das Ausführen einer *FollowLineEvent* Aktion darstellt. In dieser Abbildung ist die 3-Tier Architektur mit Front-End, Back-End und dem System-Interface sehr gut ersichtlich. Man kann auch erkennen, dass es gerade bei Aktionen wie *followLineEvent*, die Sensorik und Aktorik benötigen, wichtig ist, dass diese als atomare Aktion implementiert sind. Ansonsten würde ein Großteil der Kommunikation, die jetzt nur zwischen der *Ev3Function* Klasse und der Hardware ablaufen, über das Backend gehen und somit jedes Mal Daten auf Datenbank speichern.

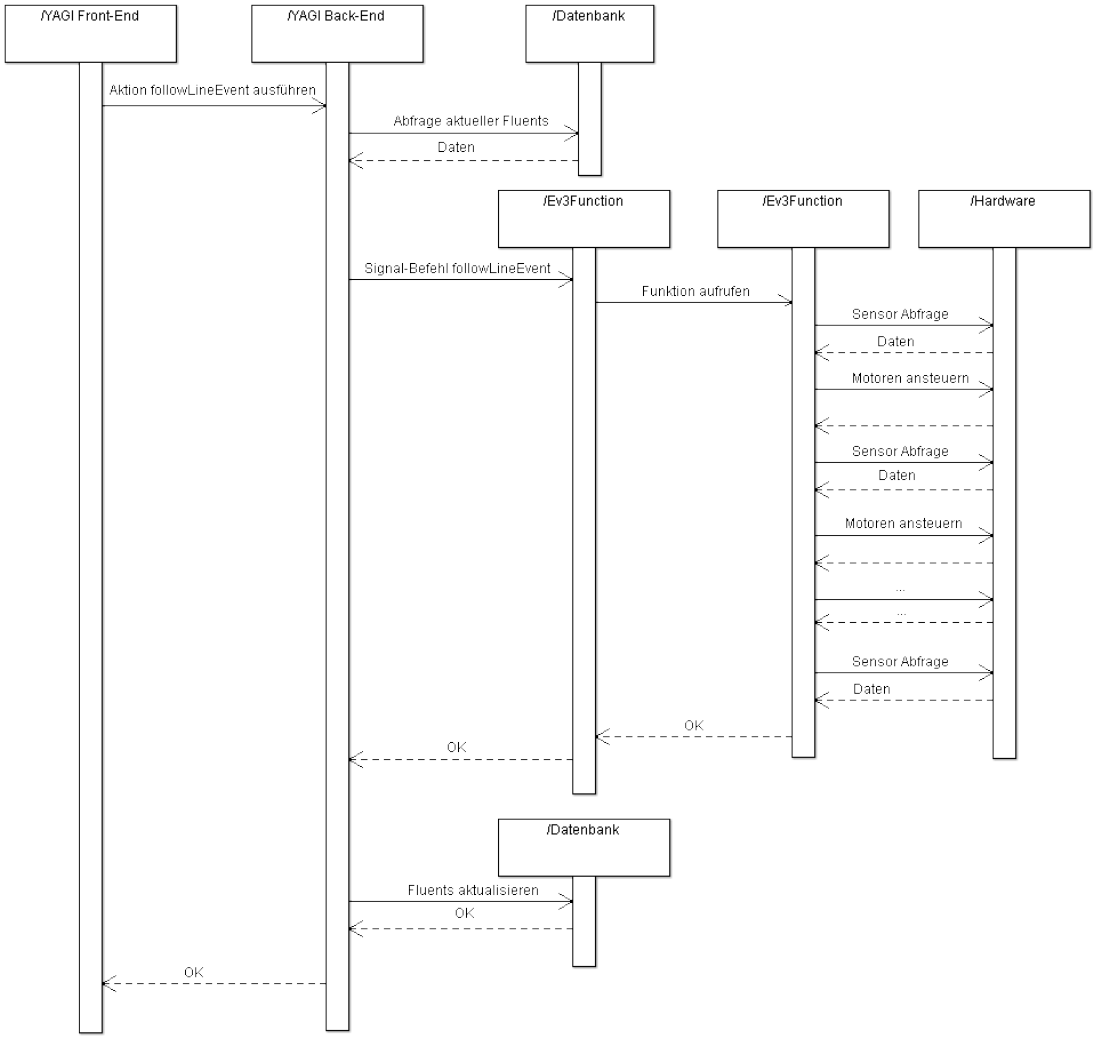

Abbildung 14: Sequenzdiagramm eines followLineEvent Aufrufs

5.8.2 PID-Regler des Linienfolgers

Im Bereich automatisierte Systeme werden sehr häufig Regelkreise verwendet. Einer der bekanntesten ist der PID-Regler, mit proportionalem, integralem und differentialem Verhalten. (Ziegler et. al., 1993) beschriebt in einem Artikel wie man zu den optimalen Einstellungen für PID-Regler gelangt. Der proportionale Anteil ist der bekannteste Effekt und wird praktisch in jedem Regler verwendet. Wie der Name schon sagt, ist das Ausgangssignal eines reinen P-Reglers proportional dem Eingangssignal. Der proportionale Anteil eines Reglers ermittelt den aktuellen Fehler eines Regelkreises, multipliziert diesen mit dem Verstärkungsfaktor K_P und wirkt augenblicklich entgegen. Der Nachteil reiner P-Regler ist die bleibende Regelabweichung. Um diese Regelabweichung zu eliminieren, erweitert man Regler mit einem integralem Verhalten. Der Integral-Anteil summiert die Regelabweichung über die Zeit auf und multipliziert dieses Integral

mit dem Faktor K_I. Reine I-Regler sind langsamer als normale Regler, regeln aber sehr genau. Um die Geschwindigkeit eines Reglers zu erhöhen, wird der Differential-Anteil hinzugefügt. Dieser reagiert nicht auf die Höhe der Regelabweichung, sondern ermittelt die Änderungsgeschwindigkeit und multipliziert diese mit dem Faktor K_D.

Um die optimalen Einstellungen, die (Ziegler et. al., 1993) beschrieben hat, auf dem Lego-Mindstorms-Ev3 zu adaptieren, hat (Sluka, 2014) eine entsprechende Anleitung dazu geschrieben. Wie schon im vorherigen Abschnitt beschrieben, startet der Linienfolger am rechten Rand der Linie und bewegt sich geradeaus, während der Licht- oder Farbsensor die entsprechenden Werte übermittelt. Um am Rand der Linie zu bleiben muss der Roboter seine Richtung der Linie entsprechend korrigieren. Dazu verwendet (Sluka, 2014) diese Formel:

$$turn = K_P * error + K_I * integral + K_D * derivative$$

Dabei entspricht *error* der Abweichung zwischen dem aktuellen Licht- oder Farbwert und dem Wert, der den Rand der Linie repräsentiert. Der Wert *integral* ist die Summe aller vorherigen Fehler und *derivative* beschreibt die Differenz zwischen aktuellem Fehler und vorherigem Fehler. K_P, K_I und K_D sind Konstante, welche durch Tabelle 3 ermittelt werden können. K_C beschreibt den Wert, bei dem der Lego-Mindstorms-Ev3 mit einem reinen P-Regler einer geraden Linie folgen kann, aber sehr stark oszilliert. Man nennt diesen Wert auch „ultimate gain“ oder „critical gain“. Die Schwingungsperiode P_C definiert die Zeit einer Schwingung beim „ultimate gain“. Die Zeit pro Schleifendurchgang *dT* ergibt sich dadurch, wie schnell der Sensor neue Licht- oder Farbwerte einliest und diese verarbeiten kann. Da sich die Zeit pro Schleifendurchgang bei jedem Durchgang ändern kann, wird ein Durchschnittswert ausgerechnet und dieser als konstant angesehen. Die Apostrophen in K_I' und K_D' geben an, dass die Zeit pro Schleifendurchgang *dT* als konstant angenommen wird und *dT* in die *K*-Werte einfließt.

Ziegler–Nichols method giving *K'* values (loop times considered to be constant and equal to dT)			
Control Type	**K_P**	**K_I'**	**K_D'**
P	0.50 K_C	0	0
PI	0.45 K_C	1.2 $K_p dt / P_c$	0
PD	0.80 K_C	0	$K_P P_C$ / (8dT)
PID	0.60 K_C	2 $K_p dt / P_c$	$K_P P_C$ / (8dT)

Tabelle 3: Ziegler-Nichols Methode zur Bestimmung der Werte eines Reglers

Die Werte, die durch die Ziegler-Nichols Methode ermittelt wurden, ergeben zu Beginn einen sehr

akzeptablen PID-Regler. Jedoch ist das Ermitteln der Zeit pro Schleifendurchgang *dT* und der Schwingungsperiode P_C eine schwierige Angelegenheit. Daher werden die Parameter meist noch ein wenig verändert, um an die optimalen Werte zu gelangen. In der Tabelle 4 wird beschrieben, wie sich Änderungen der Parameter auf das Verhalten der Roboters auswirkt. Durch das Erhöhen von K_P findet der Roboter bei Kurven schneller zur Kante der Linie zurück. Allerdings kann bei einem zu hohen Wert passieren, dass der Roboter übersteuert. Dies bewirkt, dass der Roboter auf einer geraden Linie zu oszillieren beginnt. Mit dem Erhöhen von K_I können sowohl steile Kurven leichter bewältigt, als auch der Regelfehler vollständig eliminiert werden. Ein zu hoher Wert übersteuert wie schon zuvor bei K_P den Roboter. K_D wirkt dem Übersteuern des Roboters entgegen und verhindert ein zu starkes Oszillieren.

Effect of <u>increasing</u> parameters				
Parameter	**Rise time**	**Overshoot**	**Settling Time**	**Error at equilibrium**
K_P	Decrease	Increase	Small change	Decrease
K_I	Decrease	Increase	Increase	Eliminate
K_D	Indefinite (small decrease or increase)	Decrease	Decrease	None

Tabelle 4: Effekt der Erhöhung der Parameter des PID Reglers

An dieser Stelle sollte erwähnt werden, dass der Lego-Mindstorms-Ev3 sehr einfach für verschiedene Anwendungen umgebaut werden kann. Daher kann sich der Sensor, der für den Linienfolger verwendet wird, auf verschiedenen Stellen des Roboters befinden. Je nach Ort des Sensors ändern sich auch die entsprechend optimalen Parameter des PID-Reglers. Um diese Werte je nach Anwendung zu ändern, wird in YAGI die Aktion *set_pid(p, i, d)* zur Verfügung gestellt.

6 Evaluierung

Um das Ev3-System-Interface zu evaluieren, wurde ein Testprogramm geschrieben, in dem der Lego-Mindstorms-Ev3 als Paketlieferant fungiert. Der Roboter sollte im Stande sein, einen Auftrag entgegenzunehmen und diesen auszuführen. Dazu muss der Roboter in der Lage sein den Raum zu wechseln und Objekte aufzunehmen bzw. abzugeben. Um dies zu ermöglichen, wurde eine eigene Testumgebung aufgebaut, welche in Abschnitt 6.1 erklärt wird. Anschließend wird in Abschnitt 6.2 das Prinzip des Testprogramm genauer beschrieben und erläutert, welche Einschränkungen dazu notwendig sind. Zum Schluss wird in Abschnitt 6.3 das Ergebnis der Evaluierung präsentiert.

6.1 Aufbau der Testumgebung

Es wurde eine eigene Testumgebung aufgebaut, in der es vier verschiedene Räume gibt. Jeder Raum ist einer Person zugeordnet, d.h jede Person hat ein eigenes Büro. Da der Lego-Roboter sich nicht ohne Hilfe orientieren und zwischen den Räumen bewegen kann, wurde der vorgegebene Weg mit einer durchgehenden Linie am Boden markiert. Somit kann der Lego-Mindstorms-Ev3 von einem Büro zum nächsten gelangen. Damit der Roboter ein Büro als solches erkennt, wurde jedes Büro neben der Linie mit einer unterschiedlich gefärbten Markierung gekennzeichnet. Eine Skizze zur Testumgebung wird in Abbildung 15 dargestellt. Zusätzlich wurde vorgegeben, dass der Roboter zu Beginn in Raum A startet und Richtung Osten gedreht ist. Erreicht der Lego-Mindstorms-Ev3 den Raum D, muss er sich um 180° Richtung Westen wenden.

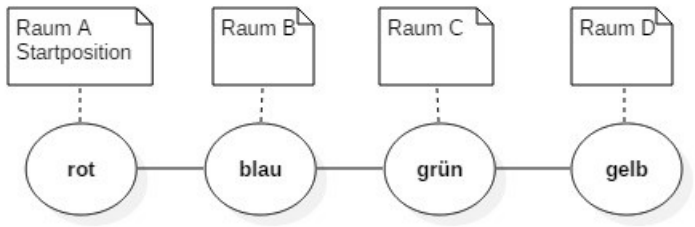

Abbildung 15: Skizze der Teststrecke eines Lego-Mindstorms-Ev3 als Paketlieferroboters

6.2 Prinzip des Testprogramms

Um einen Paketlieferdienst zu realisieren, muss es dem Benutzer möglich sein, dem Roboter einen Auftrag zu erteilen, Pakete abzugeben und Pakete zu empfangen. Der Roboter hingegen muss im

Stande sein, mehrere Aufträge anzunehmen und diese der Reihe nach abzuarbeiten. Dazu muss der Roboter sich in verschiedene Räume bewegen und orientieren können. Zusätzlich benötigt der Roboter einen Mechanismus, mit welcher er erkennt ob ein Objekt aufgenommen bzw. abgelegt wurde. Diese Aufgaben werden in einem Usecase-Diagramm in Abbildung 16 dargestellt.

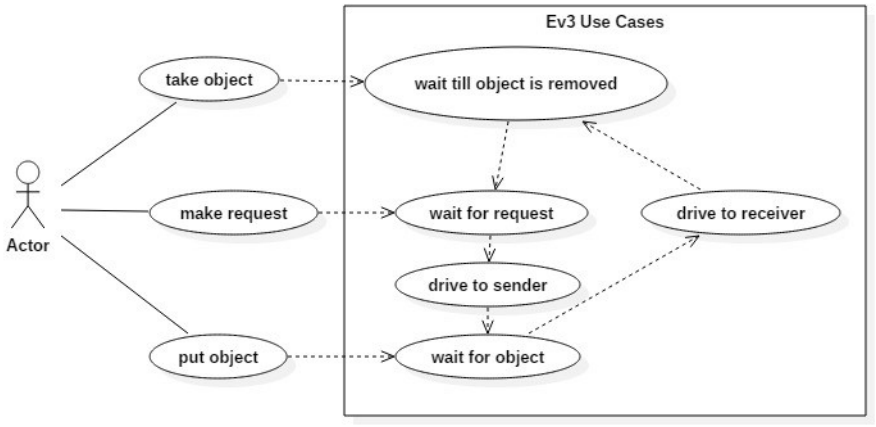

Abbildung 16: Usecase-Diagramm eines Lego-Mindstorms-Ev3 Paketlieferroboters

Zu Beginn des Programms wartet der Roboter bis er einen Auftrag erhält. Das Erteilen eines Auftrages kann in YAGI einfach als exogenen Event implementiert werden. Nun muss sich der Roboter in das Büro des Senders bewegen. Dazu folgt er der am Boden markierten Linie, bis er mit dem zweiten Sensor eine farbliche Markierung erreicht, welche einem Raum zugeordnet ist. Durch diese Markierung wird dem Lego-Mindstorms-Ev3 mitgeteilt in welchem Büro er sich zurzeit befindet. Erreicht der Roboter Raum A oder Raum D, muss er um 180° wenden, da er zum Ende der Linie gelangt ist. Befindet sich der Lego-Roboter im Büro des Senders, so wartet dieser bis er ein Objekt erhalten hat. Der Lego-Mindstorms-Ev3 ist nicht im Stande selbstständig Objekte aufzunehmen oder abzulegen, jedoch kann das Aufnehmen eines Paketes mit Hilfe des Berührungssensors erkannt werden. Ist die Taste am Sensor gedrückt, so liegt ein Paket auf dem Lego-Mindstorms-Ev3. Danach begibt sich der Roboter in das Büro des Empfängers und wartet bis das Objekt abgenommen wird. Somit ist der Auftrag erledigt und es kann der nächste ausgeführt werden.

6.3 Ergebnis der Evaluierung

Das Testprogramm wurde, wie oben beschrieben, erstellt und kann im Anhang begutachtet werden. Im Testprogramm wurde mit Hilfe von Fakten jedem Büro eine Farbe zugeteilt und jeder Person ein

Büro zugeteilt. Die Farben wurden den Büros so zugeordnet, dass die vom Farbsensor gelesene Werte in aufsteigender Reihenfolge sind. Dadurch weiß der Roboter in welche Richtung er fahren muss. Des weiteren wird durch die Initial-Werte der Fluents festgelegt an welcher Position der Roboter startet und in welche Richtung er gedreht ist. Durch ein exogenes Event können dem Roboter neue Aufträge erteilt werden. Im Hauptprogramm wird einen Auftrag aus einer Liste ausgewählt. Danach fährt der Roboter zu dem Büro des Senders und wartet auf ein Objekt. Das Halten eines Objektes wird durch das Drücken des Berührungssensors erkannt. Anschließend fährt der Roboter in das Büro des Empfängers und wartet, bis das Objekt vom Roboter entfernt wurde. Danach beginnt die Schleife von vorne, bis alle Aufträge abgearbeitet sind.

Ebenfalls wurde eine Teststrecke für den Paketlieferroboter konstruiert, welche in Abbildung 17 abgebildet ist. Durch das Testprogramm ist ersichtlich, dass das Ev3-System-Interface erfolgreich in YAGI implementiert wurde. Die Motoren und Sensoren des Lego-Mindstorms-Ev3 können nun in YAGI einfach über die entsprechenden Aktionen angesteuert werden. Weiters zeigt sich, dass komplexere Aufgaben für den Lego-Mindstorms-Ev3, wie das Zustellen von Paketen, in YAGI einfach implementiert werden können.

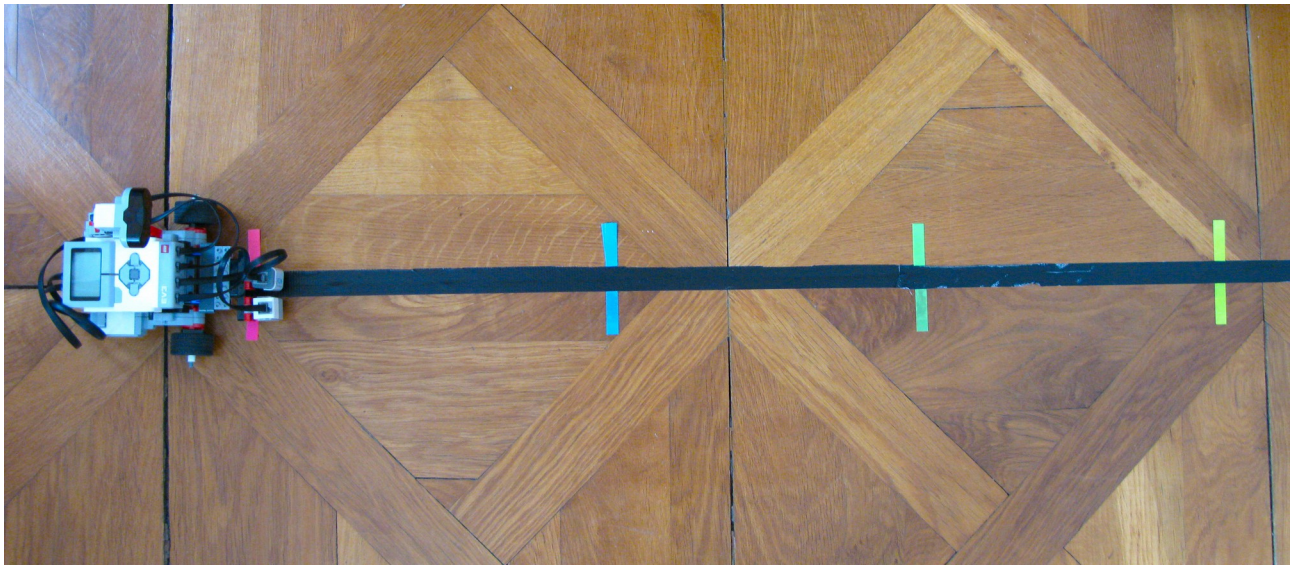

Abbildung 17: Realisierung der Teststrecke eines Lego-Mindstorms-Ev3 als Paketlieferroboters

7 Fazit

7.1 Zusammenfassung

Das Ziel dieser Arbeit war es, den bestehenden YAGI-Interpreter um ein System-Interface für den Lego-Mindstorms-Ev3 zu erweitern, damit YAGI native auf dem Lego-Mindstorms-Ev3 laufen kann. Außerdem sollten die Signal-Befehle der Aktionen in YAGI direkt als entsprechende Aktionen am Lego-Mindstorms-Ev3 ausgeführt werden.

Zu Beginn wurde der bestehende YAGI-Interpreter vorgestellt und wie dieser durch eine 3-Tier Software-Architektur aufgebaut ist. Es wurde besprochen, welche Aufgaben das Front-End, das Back-End und das System-Interface haben und wie ein neues System-Interface für den Lego-Mindstorms-Ev3 implementiert werden kann.

Anschließend wurden verschiedene Möglichkeiten diskutiert, wie der YAGI-Interpreter auf dem Lego-Mindstorms-Ev3 ausgeführt werden kann. Dazu wurde generell beschrieben, welche technischen Voraussetzungen gegeben sein müssen, damit man den YAGI-Interpreter auf einem autonomen System kompilieren kann. Ebenso wurde eine Linux Distribution für den Lego-Mindstorms-Ev3 vorgestellt, welche es ermöglicht nicht nur auf die Peripherie des Roboters zuzugreifen und eine Vielzahl an Bibliotheken zu nutzen, sondern auch den YAGI-Interpreter auszuführen.

Weiters wurde gezeigt, welche Peripherie für den Roboter zur Verfügung steht, und wie diese angesprochen werden kann. Die Implementationen der einzelnen Methoden für die Motoren, Sensoren, LEDs und dem Soundsystem wurden aufgelistet und genauer beschrieben. Ebenso wurde auf die Problematik eingegangen, dass komplexere Funktionen direkt am Lego-Mindstorms-Ev3 laufen und nicht in YAGI mit primitiven Aktionen implementiert werden sollen. Daher wurde auch ein Linienfolger als Roboterfunktion vorgestellt und dessen Regelung erklärt.

Es wurde ebenfalls erwähnt, dass das Ev3-System-Interface nach Belieben erweitert werden kann und daher in der Implementation auf eine einfache Erweiterbarkeit geachtet wurde.

Zum Schluss wurde das Ev3-System-Interface evaluiert, indem der Roboter als Paketlieferant verwendet wurde. Es wurde erklärt, wie die Testumgebung aufgebaut wurde und wie das Testprogramm funktioniert. Dadurch wurde gezeigt, dass das Testprogramm, welches im Anhang begutachtet werden kann, direkt auf dem Lego-Mindstorms-Ev3 ausführbar ist.

Zusammengefasst wurde der YAGI-Interpreter mit einem Ev3-System-Interface erweitert, welches die Benutzung von YAGI native auf dem Lego-Mindstorms-Ev3 ermöglicht.

7.2 Zukünftige Arbeiten

Mit Hilfe der Ev3-System-Interface-Implementation können YAGI-Programme auf dem Lego-Mindstorms-Ev3 ausgeführt und die Peripherie des Roboter verwendet werden. Nachfolgend werden noch einige sinnvolle Erweiterungen dieser Arbeit angeführt.

Die grundlegenden Funktionen für die Ansteuerung der Peripherie des Lego-Mindstorms-Ev3 wurden in dieser Arbeit bereits in das Ev3-System-Interface eingefügt. Wie jedoch in Abschnitt 4.2.3 besprochen, sollten auch komplexere Methoden direkt auf dem Roboter laufen. Daher wäre es sinnvoll das Ev3-System-Interface mit einigen solchen Methoden zu erweitern, um interessante YAGI-Programme schreiben zu können. Da der Ev3-Brick vier Ports zum Ansteuern von Motoren zur Verfügung stellt, könnten zum Beispiel zwei davon genutzt werden, um einen Arm zu steuern, welcher Objekte aufheben kann.

Weiters wurden bisher nur die Sensoren für den Lego-Mindstorms-Ev3 und einige des älteren Lego-Mindstorms-NXT getestet. Somit wäre es interessant verschiedenste Sensoren in YAGI zu testen und diese bei Bedarf dem System-Interface hinzuzufügen. An dieser Stelle sollte erwähnt werden, dass der RFID-Sensor von Codatex, welcher vom älteren Lego-Mindstorms-NXT unterstützt wurde, in YAGI nicht erkannt wird. Die Ursache dafür ist, dass kein entsprechender Treiber im ev3dev-Betriebssystem implementiert ist.

Ein generelles Problem beim Ausführen von YAGI-Code am Lego-Mindstorms-Ev3 ist die Laufzeit, da dem Ev3-Brick nur 64 MB RAM zur Verfügung stehen. Dadurch können Features wie das Planen, die YAGI zur Verfügung stellt, nicht auf dem Roboter genutzt werden. Im Interesse des Anwender ist es daher, eine mögliche Verbesserung der Performance des YAGI-Interpreters zu finden.

Da YAGI plattformunabhängig ist, ist es generell von großer Bedeutung verschiedene System-Interface für andere Roboter zu implementieren. Denn je mehr System-Interface in YAGI vorhanden sind, desto öfter wird es verwendet werden.

Literaturverzeichnis

Aho, A., Lam, M., Sethi, R., und Ullman, J. (2007). Compilers: Principles, Techniques and Tools. *2nd Edition. Pearson Education.*

Benitti, F. B. V. (2012). Exploring the educational potential of robotics in schools: A systematic review. *In Computers & Education. 58 (3), S. 978-988.*

„Brick Command Center“ (2016). *http://bricxcc.sourceforge.net/ [Stand 2016-02-20]*

Cliburn, D. (2006). Experiences with the LEGO Mindstorms™ throughout the Undergraduate Computer Science Curiculum. *In Proceedings. Frontiers in Education. 36th Annual Conference, San Diego, CA, 2006, S. 1-6.*

De Giacomo, G., Lespérance, Y., Levesque, H. J., and Sardina, S. (2009). Indigolog: A high-level programming language for embedded reasoning agents. In *Multi-Agent Programming. Springer.* S. 31–72.

Eguchi, A., Hughes N., Stocker, M., Shen, J., and Chikuma, N. (2011). RoboCupJunior – A Decarde Later. *In RoboCup 2011: Robot Soccer World Cup XV. Springer. S. 63–77.*

Enderton, H. B. (2001). A mathematical introduction to logic. *2nd Edition. Academic press.*

Ferrein, A. (2010). Golog.lua: Towards a Non-Prolog Implementation of Golog for Embedded Systems. *In AAAI Spring Symposium: Embedded Reasoning.*

Ferrein, A., und Steinbauer, G. (2010). On the Way to High-Level Programming for Resource-Limited Embedded Systems with Golog. *In Simulation, Modeling, and Programming for Autonomous Robots. Springer. S. 229 – 240.*

Ferrein, A., Steinbauer, G., und Vassos, S. (2012). Action-based imperative programming with yagi. In *Proceedings of the 8th International Cognitive Robotics Workshop at AAAI-12*.

De Giacomo, G. and Palatta, F. (2000). Exploiting a Relational DBMS for Reasoning about Actions. *In Proc. of CogRob 2000.*

Ingrand, F. F., Chatila R., Alami R., und Robert F. (1996). PRS: a high level supervision and control language for autonomous mobile robots. *Robotics and Automation, 1996. Proceedings. S. 43-49.*

Kandlhofer, M., und Steinbauer G. (2015). Evaluating the impact of educational robotics on pupils' technical- and social-skills and science related attitudes. *In Robotics and Autonomous Systems S. 679-685.*

Kreibich J. A. (2010). Using SQLite. *1st Edition. O'Reilly Media.*

Lechner, D., und Hempel, R. (2014). http://www.ev3dev.org/ *[Stand 2016-02-22]*

Lego (2013). Lego-Mindstorms-Ev3 User Guide. *http://www.lego.com/en-us/mindstorms/learn-to-program [Stand 2016-03-02]*

Levesque, H. J., und Pagnucco, M. (2000). Legolog: Inexpensive Experiments in Cognitive Robotics. *In Proc. of CogRob2000.*

Levesque, H. J., Reiter, R., Lespérance, Y., Lin, F., und Scherl, R. B. (1994). Golog: A logic programming language for dynamic domains. *In The Journal of Logic Programming. 31 (1-3). S. 59-83.*

Lindh, J., und Holgersson, T. (2005). Does lego training stimulate pupils' ability to solve logical problems? *In Computers & Education 49 (4) S. 1097–1111*.

McCarthy, J. (1963). Situations, actions, and causal laws. *Technical report, DTIC Document.*

Maier, C., Wotawa F., Steinbauer G. (2015). YAGI – An Easy and Light-Weighted Action-Programming Language for Education and Research in Artificial Intellegence and Robotics. *Faculty for Computer Science and Bio-medical Engineering. Graz University of Technology. 2015*

Murphy, R. R. (2000). Introduction to AI Robotics. *MIT Press*

Papert, S. (1993). Mindstorms: Children, Computers, and Powerful Ideas. *New York: Basic Books.*

Parr, T. (2013). The Definitive ANTLR 4 Reference. *Pragmatic Bookshelf.*

Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Wheeler, R., und Ng, A. (2009) ROS: an open-source Robot Operating System. *ICRA Workshop on Open Source Software*

Reiter, R. (2001). Knowledge in action: logical foundations for specifying and implementing dynamical systems. *MIT press.*

„Robotc" (2016). *http://www.robotc.net/ [Stand 2016-02-14]*

Sluka, J. (2014). A PID Controller for Lego Mindstorms Robots. *http://www.inpharmix.com/jps/PID_Controller_For_Lego_Mindstorms_Robots.html [Stand 2015-11-03]*

Veloso M., Carbonell J., Pérez A., Borrajo D., Fink E., Blythe J. (1995). Integrating Planning and Learning: The PRODIGY Architecture. *Journal of Experimental and Theoretical Artificial Intelligence. S. 81-120*

Wooldridge M. (2009). An Introduction to MultiAgent Systems. *2nd Edition. Wiley*

Ziegler, J. G., Nichols N. B. (1993). Optimum Settings for Automatic Controllers. *In J. Dyn. Sys., Meas., Control 115(2B), S. 220-222*

Anhang

YAGI Quellcode eines Paketlieferanten für den Lego-Mindstorms-Ev3

```
// location of the ev3
// 00002...blue office
// 00003...green office
// 00004...yellow office
// 00005...red office
fluent location [{"00002", "00003", "00004", "00005"}];
location ={<"00002">};

// direction of the ev3
fluent direction [{"east", "west"}];
direction ={<"east">};

// holding object
// 00000...holding no object
// 00001...holding an object
fluent object [{"00000", "00001"}];
object ={<"00000">};

// every person has an office
fact office [{"person1", "person2", "person3", "person4"}][{"00002", "00003" , "00004" , "00005"}];
office ={<"person1", "00002">,<"person2", "00003">,<"person3", "00004">,<"person4", "00005">};

// requests from sender to receiver
fluent request [{"person1", "person2", "person3", "person4"}][{"person1", "person2", "person3", "person4"}];

// initialize one sensor
action setup($x, $y)
signal:
  "setup(" +$x +"," +$y +")";
end action

// set the mode of a sensor
action set_mode($x, $y)
signal:
  "set_mode(" +$x +"," +$y +")";
end action

// initialize all sensors
proc init()
  setup("EV3 color", "in1");
  setup("NXT light", "in2");
  setup("EV3 touch", "in3");
  set_mode("in1", "COL-COLOR");
end proc

// ev3 drives to the next room, using the line follower
action follow_line_event($x, $y)
signal:
  "follow_line_event(" +$x +"," +$y +")";
end action

// get the room number from the coloured stripe
action get_location() external ($room~)
effect:
  location -={<_>};
  location +={<$room~>};
```

```
signal:
  "read_sensor(in1)";
end action

// move the ev3 exactly to the middle of the room
action move_ab_time()
signal:
  "move_ab_time(30,30,1000)";
end action

// turn around the ev3
action turn($dir)
precondition:
  // ev3 do not look to this direction
  not(<$dir> in direction);
effect:
  // now the ev3 look in this direction
  direction -= {<_>};
  direction += {<$dir>};
signal:
  "move_ab_time(30, -30, 3000)";
end action

// move the ev3 to the requested room
proc drive_to_room($room)
  if (exists <$x> in location such $x <$room) then
    if(exists <$y> in direction such $y !="east") then
      turn("east");
    end if
  else
    if(exists <$y> in direction such $y !="west") then
      turn("west");
    end if
  end if

  while not (exists <$x> in location such $x ==$room) do
        follow_line_event("in2", "in1");
        get_location();
    move_ab_time();
  end while
end proc

// check if ev3 is holding an object represented by pressing a button
action hold_object() external ($value)
effect:
  object -= {<_>};
  object += {<$value>};
signal:
  "read_sensor(in3)";
end action

// get request from external
exogenous-event receive_request ($sender, $receiver)
  request += {<$sender, $receiver>};
end exogenous-event

proc main()
  // initialization for the ev3
  init();

  // do all orders
  foreach <$sender, $receiver> in request do
    // drive to the senders office
    pick <$sender, $roomSender> from office such
      drive_to_room($roomSender);
```

```
    end pick
    // wait for getting the object
    while not (exists <$x> in object such $x == "00001") do
      hold_object();
    end while
    // drive to the receivers office
    pick <$receiver, $roomReceiver> from office such
      drive_to_room($roomReceiver);
    end pick
    // wait for dropping the object
    while not (exists <$x> in object such $x == "00000") do
      hold_object();
    end while
  end for
end proc
```