Hochschule RheinMain
Fachbereich Design Informatik Medien
Studiengang Allgemeine Informatik

Bachelor-Thesis
zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

Theoretische Konzepte und Entwicklung einer Software für eine mechanische Turingmaschine

vorgelegt von Jan Gampe

am TODO 2010

Referent: Prof. Dr. Reith
Korreferent: Dipl-Inform. (FH) Marcus Thoss, M.Sc.

Erklärung

Ich versichere, dass ich die Bachelor-Thesis selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Wiesbaden, TODO Datum Jan Gampe

Hiermit erkläre ich mein Einverständnis mit den im Folgenden aufgeführten Verbreitungsformen dieser Bachelor-Thesis:

Verbreitungsform	**ja**	**nein**
Einstellung der Arbeit in die Bibliothek der HS-RM	√	
Veröffentlichung des Titels der Arbeit im Internet	√	
Veröffentlichung der Arbeit im Internet		×

Wiesbaden, TODO Datum Jan Gampe

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Quellcodes

Verzeichnis der Algorithmen

Kapitel 1

Verwendete Marken

LEGO und LEGO Mindstorms sind eingetragene Warenzeichen der Lego Group.

Kapitel 2

Einleitung

Das Modell der Turingmaschine ist die Grundlage für viele Beschreibungen und Beispiele innerhalb der theoretischen Informatik. Im Vergleich zum Lebenszyklus einer Programmiersprache ändert sich an dieser theoretischen Basis nur wenig. In der Lehre der Automatentheorie sind sie ein essentieller Bestandteil. Es gibt viele rein in Software realisierte Simulatoren, um das Thema anschaulicher zu machen.

Im Wintersemester 08/09 haben Studenten der Århus Universität eine mechanische Turingmaschine aus Lego Mindstorms Produkten realisiert [Hav09]. Die Grundidee besteht darin, die TM als Eisenbahn (Schreib/Leseeinheit) auf Schienen (Turing-Band) darzustellen. In einem gemeinsamen Projekt mit meinem Kommilitonen Sebastian Flothow (B.Sc.) haben wir dieses Modell in der Bestrebung übernommen, es konzeptionell und technisch zu verbessern. Am Ziel unseres Projekts steht ein Exponat für die Hochschule RheinMain, mit dem das Modell Alan Turings von einem breiten Publikum nachvollzogen werden kann.

Zu Beginn dieser Thesis ist von ihm die mechanische Turingmaschine als Prototyp fertiggestellt worden. Aufgabe und Thema dieser Abschlussarbeit liegt in der Erstellung einer geeigneten Firmware, sowie einer Applikation auf einem zweiten System, mit der Turingmaschinen modelliert und auf die Lego-Turingmaschine überspielt werden können. Im zweiten Teil werden unter Berücksichtigung der Anwendbarkeit auf der Lego-Turingmaschine Probleme der Berechenbarkeitstheorie betrachtet.

Kapitel 3

Grundlagen

3.1 Die Turingmaschine

1936 hat der britische Mathematiker Alan Turing ein Maschinenmodell entwickelt, um die theoretischen Grenzen von mechanischen Rechenmaschinen zu untersuchen, welches er „Logical Computing Machine“ nannte. (vgl. [Tur48] und [Tur36]).

Bei dieser Maschine handelt es sich um einen Apparat, welche sich auf einem unendlich großem Bandspeicher bewegt und diesen nach den Regeln eines Zustandsautomaten manipuliert. Ein Arbeitsschritt der Maschine hängt von der kleinsten Informationseinheit des Bandes, dem *Bandsymbol*, und dem aktuellen Zustand ab. Das Symbol wird gelesen und abhängig vom aktuellen Zustand ein neues Symbol an die Stelle des alten geschrieben. Die Maschine bewegt sich zu einem der beiden Nachbarn des Bandsymbols und geht in einen neuen Zustand über. Die Sammlung aller Arbeitsvorschriften der Maschine wird *Übergangsfunktion* genannt.

Definition 1. *(vgl. [Tur36]): Eine Turingmaschine ist ein 7-Tupel* $TM = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ *für das gilt:*

- Q *ist die Menge der Zustände*
- Σ *ist das Eingabealphabet*
- $\Gamma \supset \Sigma$ *ist das Bandalphabet*
- δ *ist die Übergangsfunktion*
- $\square \in \Gamma \setminus \Sigma$ *kennzeichnet das Leerzeichen des Bandalphabets*

- $q_0 \in Q$ *ist der Startzustand*
- $F \subseteq Q$ *ist die Menge der Endzustände*

Die Übergangsfunktion δ lautet wie folgt:
$\delta : (Q \setminus F) \times \Gamma \rightarrow Q \times \Gamma \times L, R, N$

Beispiel 1. *Die folgende TM erhöht eine binär repräsentierte Zahl um eins:*
$TM_{binInc} = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$
$Q = \{q_0, q_1, q_2, q_3\}$
$\Sigma = \{0, 1\}$
$\Gamma = \{0, 1, \square\}$
$F = \{q_3\}$
$\delta = \{$
$(q_0, 0) = (q_0, 0, R)$
$(q_0, 1) = (q_0, 1, R)$
$(q_0, \square) = (q_1, \square, L)$
$(q_1, 0) = (q_1, 1, L)$
$(q_1, 1) = (q_2, 0, L)$
$(q_1, \square) = (q_2, 0, L)$
$(q_2, 0) = (q_2, 0, L)$
$(q_2, 1) = (q_2, 1, L)$
$(q_2, \square) = (q_3, \square, R)$
$\}$

Für Bandeingaben gilt, dass die Turingmaschine zu Beginn der Berechnung am linken Ende der Eingabe positioniert ist. Die Maschine arbeitet in ihren Zuständen wie folgt: In q_0 bewegt sich die Maschine ohne die Eingabe zu manipulieren auf die rechte Seite der Eingabe, um an die niedrigstwertige Ziffer zu kommen. q_1 sucht nun von links nach rechts nach dem ersten auftretenden Null oder dem leeren Bandzeichen. Jede Eins wird auf dem Weg dorthin mit einer Null überschrieben. Wie beim klassischen schriftlichen Addieren wird hier die Mitnahme des Übertrags nachgebildet. Wird eine Null oder das Leerzeichen erreicht wird hier der Übertrag mit Null addiert und das Ergebnis Eins geschrieben. Die eigentliche Addieraufgabe ist hiermit getan. q_2 und q_3 bewegen die TM wieder auf die Position der von links ersten Position eines Nicht-Leerzeichens.

Definition 2. *Die Konfiguration einer Turingmaschine ist ein Tripel $K \in (w_a, q, w_b)$ für das gilt:*

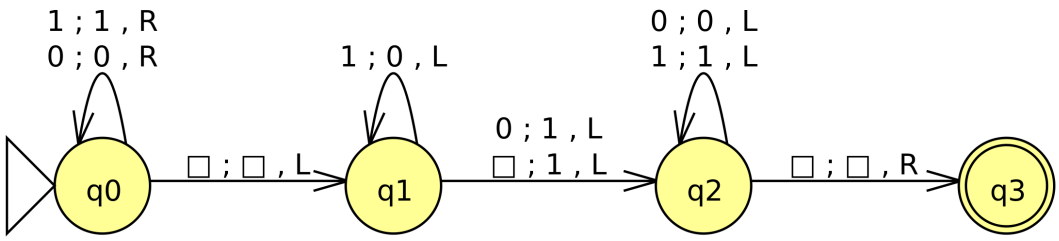


Abbildung 3.1: Automatendiagramm eines Binäraddierers

- $w_a w_b \in \Gamma^*$ *sind die Bandsymbole. Auf dem ersten Symbol aus* w_b *befindet sich der Schreib-/Lesekopf der TM. Die Symbole aus* w_a *befinden sich links von dessen aktueller Position, die übrigen Symbole aus* w_b *rechts davon.*
- $q \in Q$ *ist ein Zustand der Turingmaschine*
- *Die Turingmaschine befindet sich auf dem ersten Symbol von* w_b

Definition 3. *Eine Turingmaschine ist* deterministisch*, wenn es in der Übergangsfunktion in jedem Zustand* $Q \setminus F$ *für jedes Bandsymbol* $x \in \Gamma$ *höchstens einen Übergang gibt.*

Daraus folgt, dass bei einer gegebenen deterministischen TM_X und einer Konfiguration K_n nach einem Arbeitsschritt die Folgekonfiguration K_{n+1} eindeutig bestimmbar ist. Wir schreiben: $K_n \vdash K_{n+1}$.

Beispiel 2. *Wir verwenden* TM_{binInc} *weiter und geben ihr die Zahl* 11011 *als Eingabe. Die Konfiguration, mit der begonnen wird, lautet somit* $(\square, q_0, 11011)$*. Da* TM_{binInc} *deterministisch ist, lassen sich die Arbeitsschritte der TM wie folgt notieren:*
Bewegung bis zum rechten Ende der Eingabe:
$(\square, q_0, 11011) \vdash (1, q_0, 1011) \vdash (11, q_0, 011) \vdash (110, q_0, 11) \vdash$
$(1101, q_0, 1) \vdash (11011, q_0, \square) \vdash (1101, q_1, 1)$
Suche nach dem ersten Auftreten einer Null oder dem Leerzeichen. Überschreibe jede Eins mit Null:
$(1101, q_1, 1) \vdash (110, q_1, 10) \vdash (11, q_1, 000) \vdash (1, q_2, 1100)$
Bewegung bis zum linken Ende der Bandeingabe:
$(1, q_2, 1100) \vdash (\square, q_2, 11100) \vdash (\square, q_2, \square 11100) \vdash (\square, q_2, \square 11100)$
Endposition und Endzustand erreicht:
$(\square, q_2, \square 11100) \vdash (\square, q_3, 11100)$

3.2 Lego NXT

Lego NXT ist ein Roboterbaukasten aus der Mindstorms Produktreihe. Ihre Hauptbestandteile umfassen Motoren und Sensorik, welche sich mit aus Lego Technic bereits bekannten Bauelementen mechanisch verbinden lassen. Das Herzstück bildet der „NXT Brick“, ein frei programmierbarer Baustein, über den Aktoren und Sensoren elektrisch verbunden sind. // TODO: füllen

3.3 Die mechanische Turingmaschine

Die mir zur Verfügung gestellte mechanischen Turingmaschine ist der aus [Hav09] nachempfunden. Ihre drei Hauptkomponenten sind der Bewegungsapparat, der Schreib- und der Lesekopf. Kontrolliert werden sie über einen gemeinsamen Lego NXT Baustein. Die Turingmaschine fährt auf Eisenbahnschienen des Typs 7896. Für eine genauere Beschreibung siehe [Flo].

Lesekopf

Der Lesekopf ist eine Kombination aus einem Motor und zwei Tastsensoren. Diese werden über dem aktuellen Bandsymbol fast vertikal bewegt. Der Motor kann sich konstruktionsbedingt nur innerhalb eines Winkels von ca. 90 Grad bewegen, bevor die Schubstange an das Motorgehäuse gepresst wird und die Bewegung blockiert.

Schreibkopf

Der Schreibkopf ist eine Schiebevorrichtung, welche ein Bandsymbol innerhalb des Freiraums bewegen kann. Basiselement ist ein rechteckiger Rahmen, welcher über Zahnschienen und Zahnräden von einem Motor verschoben werden kann. Die Seiten, die parallel zu den Schienen ausgerichtet sind, reichen tiefer herunter als der Rest, um an die Symbolsteine zu gelangen.
Schreib- und Lesekopf befinden sich stets über dem selben Bandsymbol.

Bewegungsapparat

Der dritte Motor des NXT wird dazu verwendet, die Turingmaschine auf den Schienen zu bewegen. Ein Farbsensor ist dazu montiert, um durch Farbmarkierungen das nächste Bandsymbol zu erkennen. Dieser ist am Ende der Maschine verbaut, sodass die dort

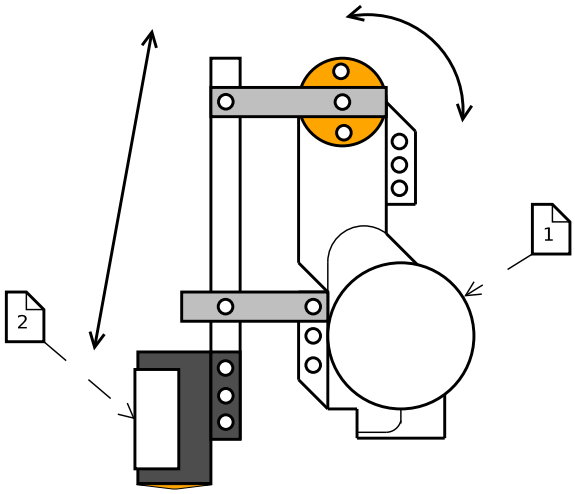


Abbildung 3.2: Schematische Darstellung des Lesekops. Die Rotationsbewegung des Motors (1) führt zu einer fast vertikalen Bewegung der Tastköpfe (2)

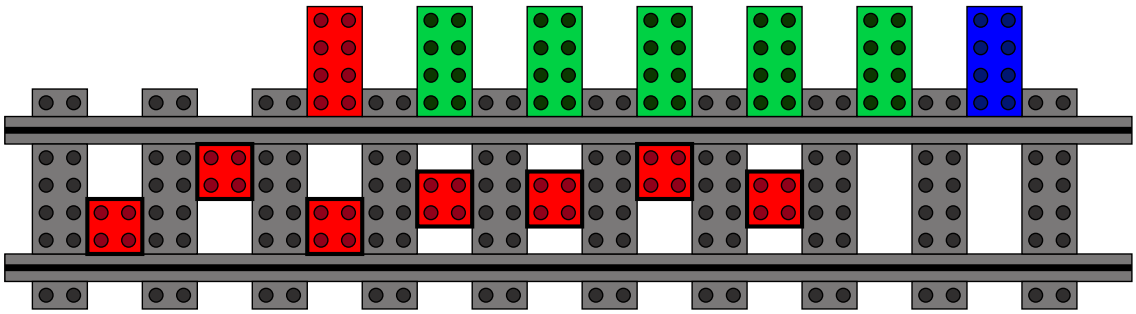


Abbildung 3.3: Schienenabschnitt: “0 1 0 □ □ 1 □ ”

gelesene Farbmarkierung für das Bandsymbol gilt, welches gerade auf der Position des Schreib-/Lesekopfes steht. Das führt dazu, dass die zu einem Bandsymbol gehörende Farbmarkierung um zwei Positionen versetzt angebracht werden muss. Siehe dazu auch die Abbildung 3.3.

Turing-Band

Das Turing-Band besteht aus Schienenelementen. Zwischen jeder Querstrebe der Schienen ist ein Freiraum von 6x2 Legoeinheiten. Ein 2x2 großer Legostein wird als Bandsymbol in diesen Freiraum eingesetzt. Ein Legostein kann drei verschiedene Bandsymbole durch seine Position im Freiraum abbilden. Grüne Farbmarkierungen zeigen dem Bewegungsapparat, dass sich an dieser Stelle ein reguläres Bandsymbol befindet. Eine rote und eine blaue Markierung befindet sich an den Enden des Turingbandes.

Kapitel 4

Theoretische Konzepte

In diesem Kapitel werden diverse Aspekte der Berechenbarkeitstheorie betrachtet, die durch die Implementierung einer mechanischen Turingmaschine ein „greifbares“ Beispiel erhalten.

4.1 Vorschaufunktion und Satz von Rice

Die Clientapplikation ist in der Lage, Turingmaschinen und Konfigurationen zu speichern. Dieser Abschnitt beschäftigt sich mit der Frage, welche Informationen eine Vorschaufunktion, wie es sie in Dateidialogen für z.B.: Medienformate gibt, aus einer serialisierten Turingmaschine entnehmen kann. Die theoretische Grundlage dazu bietet der Satz von Rice:

Satz 1. *(Rice) (vgl. [Ric53]):*
Sei $\mathcal{R}$ die Klasse aller Turing-berechenbaren Funktionen.
Sei $\mathcal{S} \subsetneq \mathcal{R}$, sowie $\mathcal{S} \neq \emptyset$.
Dann ist die Sprache
$C(\mathcal{S}) = \{w |$ *Die von* M_w *berechnete Funktion liegt in* $\mathcal{S}\}$
unentscheidbar.

Informell sagt der Satz, dass es nicht möglich ist, irgendeine semantische Eigenschaft einer Turingmaschine algorithmisch zu bestimmen. Da die Eigenschaft „hält auf dem leeren Band“ bereits unentscheidbar ist, entwickelte Rice folgende Beweisidee:
Für ein beliebiges $\mathcal{S}$ gibt es eine Turingmaschine T, welche, abhängig davon, ob eine intern simulierte TM $T_{halt?}$ auf leerem Band anhält, Element von $C(\mathcal{S})$ ist oder nicht.

Dazu sei die Unentscheidbarkeit des Halteproblems auf leerem Band wiederholt:

Satz 2. *Die Sprache*
$H_0 = \{w | M_w$ *hält bei Eingabe eines leeren Bands*$\}$
ist unentscheidbar.

Beweis. Kontraposition: Es gibt einen Entscheider für H_0.
Für jede Turingmaschine T_i, kombiniert mit jeder Eingabe w gibt es eine Turingmaschine $T_{f(i,w)}$, die auf leerem Band zuerst die Eingabe auf das Band schreibt und dann T_i simuliert. Damit gäbe es durch Abbildung nach $T_{f(i,w)}$ auch einen Entscheider für Turingmaschinen mit Eingabe auf dem Band → Widerspruch, da allgemeines Halteproblem nicht entscheidbar. □

Mit dem Wissen, dass die Sprache H_0 unentscheidbar ist, wird diese nun auf $C(\mathcal{S})$ reduziert.

Beweis. Sei T_S eine Turingmaschine, deren Funktion sich in $\mathcal{S}$ befindet. Da $\mathcal{S} \neq \mathcal{R}$ gibt es eine weitere Turingmaschine $T_{\bar{S}}$, welche eine sich nicht in $\mathcal{S}$ befindliche Funktion berechnet.
Sei $f_\perp$ die überall undefinierte Funktion. Eine Turingmaschine, die diese Funktion berechnet, hält unter keinen Umständen.

Abhängig davon, ob $f_\perp \in \mathcal{S}$, werden leicht unterschiedliche Reduktionen vorgenommen:

Fall 1: $f_\perp \in \mathcal{S}$**, Reduktion von** H_0

Für jede Sprache x und die Frage, ob $x \in H_0$ kann für beliebige $\mathcal{S}$ im Rahmen des Falls 1 eine Turingmaschine $T_{\mathcal{S}?}$ konstruiert werden, welche als Eingabe ein Wort w und eine Turingmaschine $T_{\langle x \rangle}$ erhält. Diese Konstruktionsvorschrift heißt im weiteren Verlauf $f(x)$. Mit ihrer Hilfe wird die Frage $x \in H_0?$ eindeutig auf eine Antwort auf $f(x) \in C(\mathcal{S})?$ abgebildet.

$T_{\mathcal{S}?}$ arbeitet wie folgt:

- Ignoriere w und simuliere $T_{\langle x \rangle}$ auf leerem Band
- Simuliere $T_{\bar{S}}$ mit Eingabe w.

Sollte $T_{\langle x \rangle}$ auf leerem Band anhalten, simuliert $T_{\mathcal{S}?}$ eine Turingmaschine, dessen berechnete Funktion nicht zu $\mathcal{S}$ gehört.

Die Abbildung in diesem Fall lautet wie folgt:

$x \in H_0$ gdw. $T_{\langle x \rangle}$ mit leerer Eingabe hält.
gdw. $f(x) = \langle T_{\mathcal{S}?} \rangle$, $T_{\mathcal{S}?}$ berechnet $s \in \mathcal{S}$
gdw. $f(x) \in C(\mathcal{S})$

Umgekehrt gilt:

$x \notin H_0$ gdw. M_x mit leerer Eingabe hält nicht.
gdw. $f(x) = \langle T_{\mathcal{S}?} \rangle$, $T_{\mathcal{S}?}$ berechnet $s \notin \mathcal{S}$
gdw. $f(x) \notin C(\mathcal{S})$

Fall 2: $f_\perp \notin \mathcal{S}$**, Reduktion von** $\bar{H}_0$

Es wird in diesem Fall eine Abbildung von $x \in \bar{H}_0?$ auf $f(x) \in C(\mathcal{S})?$ vorgenommen. Die Verwendung des Komplements von H_0 macht keinen Unterschied bei der Unentscheidbarkeit. $T_{\mathcal{S}?}$ arbeitet leicht modifiziert zu Fall 1:

- Ignoriere w und simuliere $T_{\langle x \rangle}$ auf leerem Band
- Simuliere $T_{\mathcal{S}}$ mit Eingabe w.

Sollte $T_{\langle x \rangle}$ auf leerem Band anhalten, simuliert $T_{\mathcal{S}?}$ eine Turingmaschine, dessen berechnete Funktion zu $\mathcal{S}$ gehört.

Es gilt nun:

$x \in \bar{H}_0$ gdw. $T_{\langle x \rangle}$ mit leerer Eingabe hält nicht.
gdw. $f(x) = \langle T_{\mathcal{S}?} \rangle$, $T_{\mathcal{S}?}$ berechnet $s \notin \mathcal{S}$
gdw. $f(x) \notin C(\mathcal{S})$

Umgekehrt gilt:

$x \notin \bar{H}_0$ gdw. M_x mit leerer Eingabe hält.
gdw. $f(x) = \langle T_{\mathcal{S}?} \rangle$, $T_{\mathcal{S}?}$ berechnet $s \in \mathcal{S}$
gdw. $f(x) \in C(\mathcal{S})$

In beiden Fällen fand sich eine total und berechenbare Abbildungsfunktion $f(x)$, um H_0, bzw $\bar{H}_0$ auf $C(\mathcal{S})$ zu reduzieren. Da H_0 unentscheidbar ist, ist auch $C(\mathcal{S})$ unentscheidbar. □

Informell abgekürzt: Ein fiktiver ‘"Funktionsbestimmer"´ für Turingmaschinen hätte bei der konstruierten Turingmaschine $T_{S?}$ stets zuerst die Aufgabe, das Halteproblem zu lösen. Dies beweist, dass man *nicht für alle* Turingmaschinen und für beliebige $\mathcal{S}$ sagen kann, ob die berechnete Funktion in ihr liegt. Für die in der Praxis wichtige Verifikation von Software ist genau dieser Aspekt von großer Bedeutung. Es kann nicht für jede Software jede Funktion verifiziert werden. Die Unmöglichkeit im Allgemeinen erlaubt trotzdem eine praktisch brauchbare Möglichkeit im Speziellen. Eine Vorschaufunktion sollte jedoch für möglichst alle Turingmaschinen geeignet sein. Da dies in dem Umfang leider nicht möglich ist, bleibt eine Reduktion der Vorschaufunktion auf triviale Eigenschaften (Anzahl der Zustände, Größe der Alphabete).

4.2 Gödelisierung

TODO: Gödelisierung ist der Urform aller Serialisierung/Deserialisierung.

4.3 Die universelle Turingmaschine

TODO: Nimmt man an, dass der ARM Prozessor turingmächtig ist (dabei das endlose Band weglassen), dann ist die Implementierung bereits eine universelle Turingmaschine. Ansonsten könnte man noch zeigen, dass die Mechanische TM eine UTM ist. Das wäre dann eine UTM, die eine UTM simulieren kann.

4.4 Mehrband-Turingmaschinen

TODO: Die Menge der n-Band TM berechenbaren Funktionen enspricht derer der 1-Band TM.

4.5 Reicheres Eingabealphabet

TODO: Das binäre Alphabet

Kapitel 5

Analyse

5.1 Anforderungen

Die Anforderungen an das Projekt stellen sich primär aus Ansprüchen an ein anschauliches Modell zusammen. Andere entstammen aus dem Grundgedanken, konzeptionelle Verbesserungen gegenüber [Hav09] einzubringen. Desweiteren ist ein Prototyp der mechanischen Turingmaschine zu Beginn dieser Thesis fertiggestellt, dessen Eigenschaften berücksichtigt werden müssen.

5.1.1 Größtmögliche Selbständigkeit der Legoeinheit

Die Turingmaschine aus Århus benötigt einen ständig verbundenen Client, welcher große Teile der Turingmaschinen-Logik verwaltet (vgl. [LDO]). Sie begründen dies unter Anderem mit der einfacheren Realisierbarkeit auf Seiten des PC, sowie dem Fehlen einer Map-Datenstruktur auf dem NXT. Ziel dieses Projektes ist es, eine Lego-Turingmaschine zu entwickeln, welche die gesamte notwendige Logik innerhalb des NXT trägt.

5.1.2 Ein Bandsymbol pro Schienenfeld

Im dänischen Projekt werden Schienenfeldern in der Größe von 5x2 Legoeinheiten und Steine in der Größe 2x2 verwendet. Architekturbedingt lassen sich nicht mehr als zwei unterschiedliche Zustände pro Schienenfeld unterbringen. Zur Vergrößerung des Bandalphabets verwenden sie bis zu 4 Schienenfelder pro Bandsymbol (vgl. [LDO]). Die mechanische TM dieses Projekts erfüllt aus mechanischer Sicht alle Voraussetzungen, um auf

Schienenfeldern der Größe 4x2 mit 2x2 großen Steinen bis zu drei unterschiedliche Bandsymbole zu schreiben und zu erkennen. Die Software soll in der Lage sein, die Mechanik dementsprechend fein ansteuern zu können. Neben einer Verbesserung in der Modelltreue bedeutet dies einen effektiven Speicherplatzgewinn.

5.1.3 Transparenz aller mechanischen Vorgänge

Die LTM ist als Exponat des Fachbereichs konzipiert. Die Vorgänge der Maschine müssen zu jedem Zeitpunkt deutlich sein. Auf mechanischer Seite wurde deshalb beispielsweise Wert darauf gelegt, dass Lesen und Schreiben ohne weitere Bandbewegungen realisiert wird. Von der Software werden daher die folgenden Funktionalitäten erwartet:

- Graphische und textuelle Beschreibung des aktuellen Vorgangs: Das Display des NXT Brick und die PC Software müssen Statusinformationen liefern.

- Stepping: Eine vom Benutzer kontrollierte schrittweise Ausführung des Programms soll zum Verständnis beitragen, wann bspw. ein Zustandsübergang stattgefunden hat.

5.1.4 Unbeaufsichtigter Betrieb

Als Ausstellungsstück soll die LTM in einer geschlossenen Vitrine wartungsarm betrieben werden können. Sie soll sich dazu gegenüber fluktuierenden äußeren Einflüssen entweder resistent zeigen oder zumindest in einen Fehler- oder Tiltzustand übergehen, von dem aus gewartet wird, bis sich die Umstände wieder normalisiert haben. Weitere Funktionalitäten dieser Kategorie:

- Selbsttätige Kalibration: Die LTM kalibriert seine mechanischen Komponenten regelmäßig.

- Automatische Wiederverbindung: Die Kommunikationsschnittstelle prüft den Verbindungsstatus und versucht selbsttätig, eine unterbrochene Verbindung wiederaufzunehmen.

5.1.5 Kompatibilität mit Dateiformaten

5.1.5.1 Import von JFLAP

JFLAP ([JFL] ist eine Software zur graphischen Beschreibung von verschiedenen Automaten. Eine Kompatibilität mit dessem Dateiformat wird angestrebt. Die Software soll mindestens bestehende in JFLAP beschriebene kompatible Konfigurationen einlesen können. Das Dateiformat, in das JFLAP alle Automaten speichert, ist XML 1.0 ohne DTD oder einer anderweitigen Beschreibung der Grammatik. Eine eigene Untersuchung ergibt für die Speicherung von Turingmaschinen folgendes Schema:

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?><!--Created with JFLAP 6.4.-->
2  <structure>
3      <type>turing</type>
4      <automaton>
5          (
6          <block id="0" name="bar"> // "id "wird als lfd. Nummer vergeben
7              <tag>foobar</tag>
8              <x>123.4</x>
9              <y>567.8</y>
10             (<initial/>)?
11             (<final/>)?
12         </block>
13         )* // Beliebig lange Liste von Zustaenden
14         (
15         <transition>
16             <from>0</from> // from/to erwarten obige IDs
17             <to>1</to>
18             <read/>         // Empty Tag bedeutet Leersymbol
19             <write/>        // Sonst: Bandsymbol in Textnode
20             <move>\[LRH\]</move>
21         </transition>
22         )* // Beliebig lange Liste von Uebergaengen
23     </automaton>
24 </structure>
```

Quellcode 5.1: XML Schema einer JFLAP File mittels Regex-Notation verdeutlicht

Es müsste vorab eine Validation geschehen, ob die gegebene Datei eine Turingmaschine enthält und wenn ja, ob die Turingmaschine in den Grenzen der Lego-TM abbildbar ist. Dazu darf die TM nicht mehr als drei Bandsymbole verwenden und muss deterministisch sein. In einem zweiten Schritt kann die Abbildung und der Import vollzogen werden.

5.1.5.2 Export nach LaTeX

Als Exportformat wird eine Beschreibung von Konfigurationen und Turingmaschinen in LaTeX angestrebt. Diese sollen nach den im Grundlagenteil gewählten Definitionen beschrieben werden. Dies hätte den direkten Nutzen in Kombination mit der JFLAP Importfunktion, da JFLAP ein Export nach LaTeX bislang fehlt. Ausserdem ließen sich Beispielprogramme mit dieser Funktion in das Dokument dieser Abschlussarbeit komfortabel einfügen.

5.2 NXT Firmware

Losgelöst von weiteren Fragestellungen kann die Entscheidung zur benutzenden NXT Firmware getroffen werden. Ein Überblick über alle verfügbaren Optionen ist im Gegenzug sogar notwendig, um die Realisierbarkeit architektonischer Konzepte bestimmen zu können. Die Anforderungen an die Firmware lauten wie folgt:

- Unterstützung der Kommunikation mit weiteren Geräten. Der NXT-Brick erlaubt USB und Bluetooth Kommunikation, mindestens eine Option sollte unter einer Linuxumgebung unterstützt werden.
- Unterstützung der Lego-eigenen Motor- und Sensorperipherie. Gerade der mit NXT 2.0 eingeführte Farbsensor wird nicht von allen Umgebungen unterstützt.
- Möglichst geringe Einarbeitungszeit in die Programmiersprache. Im Rahmen des Firmwaresuche wurden C/C++ und Javadialekte bevorzugt, um die Einarbeitungszeit geringzuhalten.

Die Entscheidung fiel zu Gunsten von “LeJos” aus. Neben den Basisanforderungen sprachen die folgenden Punkte für die Java-basierende Firmware:

- Die Programmiersprache erfordert keine weitere Einarbeitung.
- Es existiert ein USB Treiber für Linux.
- Hoher Reifegrad der Firmware und aktive Entwicklercommunity.

5.3 Über LeJos

LeJos NXJ ist der Oberbegriff für eine Programmierumgebung, mit der Programme für das Lego NXT in Java geschrieben werden können. Ihr Hauptbestandteil ist dabei die Firmware, welche eine Java Virtual Machine auf dem NXT bereitstellt.

Geschichte

Das Projekt ist ursprünglich aus einem Fork von TinyVM entstanden (vgl. [TIN], [LEJa]). Dabei handelte es sich um eine minimalistische Javaumgebung für den Vorgänger des Lego NXT, dem RCX. Auf der 32 KB großen Speicher des RCX haben die LeJos RCX Entwickler sich für einen größeren Footprint und mehr Funktionalität der JVM entschieden. Mit der Einführung des Lego NXT 2006 wurde die LeJos RCX Entwicklung eingestellt und auf das neue Modell umgestiegen. Mit höherer Speicher- und Rechenkapaiztät konzentriert sich die LeJos NXT Entwicklung auf eine breite Unterstützung von Sensoren und Aktoren (von Lego wie auch von Drittherstellern) sowie einer vielseitigen API.
Der Lego RCX Roboter *Jitter*, welcher mit Lejos RCX läuft, war im Einsatz auf der Internationalen Raumstation, um frei schwebende Objekte im Raum aufzusammeln (vgl. [JIT]).

Forks und Weiterentwicklungen

Aus Teilen des LeJos Sourcecodes ist das Echtzeitbetriebssystem LeJos-OSEK (heute: nxtOSEK) entstanden (vgl. [NXT]).
Ein LeJos Fork auf der Basis der aktuellen Version 0.85 strebt an, die RTSJ (Real-Time Specification for Java) zu erfüllen (vgl. [LEJb]).

Features

Die LeJos Klassenbibliothek sowie die VM bieten u.A. die folgenden Features:

- Fast vollständiger Sprachumfang von Java 1.6. (neu in Ver. 0.85: Generics, foreach-Schleifen)
- Garbage collection (Seit Januar 2008)
- Persistente Datenspeicherung mittels File I/O auf dem NXT
- Präemptive Threads

- Großer Treiberkatalog

Überall dort, wo es möglich ist, wird auf Java-typische Schnittstellen portiert (Input-/Outputstreams zur Kommunikation, Sockets, Fileoperationen).

Grenzen der Firmware

- Im gesamten LeJos-Konzept (von Bytecode bis Linker) ist die maximale Anzahl der einsetzbaren Klassen pro Programm auf 256 beschränkt [For10b].
- Die Firmware belegt ca. 100 der verfügbaren 256 Kilobyte Flash Speicher. Innerhalb von LeJos Executables verwenden wird mit 16 Bit Offsets gearbeitet, durch das bei ca. 64 Kilobyte Adressierungsprobleme auftreten. Dabei muss beachtet werden, dass in jedem Executable alle zum Betrieb notwendigen Basisklassen statisch gelinkt werden (vgl. [For09a]).
- Eingeschränkte Klassenbibliothek, wenige Datenstrukturen.
- Ein Methode darf nicht mehr als 16 Parameter haben (vgl. [For08]).
- Es darf immer nur eine Datei zur selben Zeit geöffnet sein (vgl. [LEJ09a]).

Kapitel 6

Design

6.1 NXT als Server

Das NXT Brick ist eine eigenständige Einheit. Es hat sämtliche Funktionen, um selbsttätig die Arbeit einer mechanischen Turingmaschine aufzunehmen. Dementsprechend ist er als Server zu sehen, der als Dienst die Arbeit als Turingmaschine anbietet. Es bedient stets nur einen Client, für etwaige Abwechslungen zwischen verschiedenen Clients haben diese selbst zu sorgen. Eine Alternative dazu wäre, den Brick als mechanische Einheit zu sehen, welcher vollständig von einer Aussenstelle betrieben wird. Dies wurde allerdings verworfen, da die erste Option authentischer und näher am theoretischen Modell ist.

6.1.1 Modell der NXT Software

Die NXT Software ist um einen endlichen Automaten zentriert. Die Ein- und Ausgaben des Automaten bilden dabei Nachrichten zwischen den Automat und der zu bedienenden Außenstelle, deren Organisation durch eine Kommunikationsschnittstelle geregelt wird.

Der Automat hat Zustandsübergänge, welche nicht notwendigerweise eine Eingabe brauchen. Zur Veranschaulichung der Arbeitsweise der Turingmaschine kann jedoch die schrittweise Ausführung (Stepping) verlangt werden. Für den Benutzer bedeutet das, dass die Aktion, für die der aktuelle Zustand steht, sowie der darauffolgende Zustandswechsel erst nach einer Step-Anweisung ausgeführt werden. Die Zustände und ihr Verhalten lauten wie folgt:

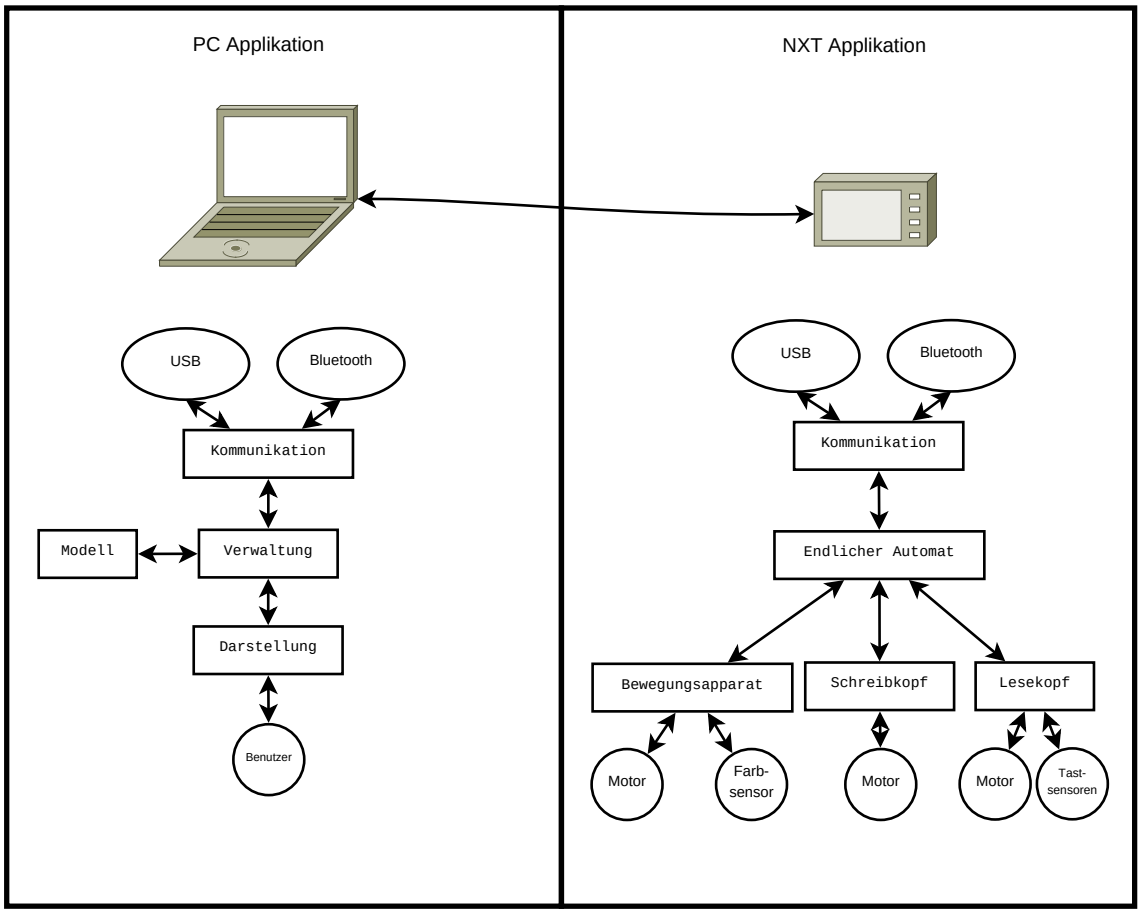


Abbildung 6.1: Komponentenmodell PC und NXT Applikation

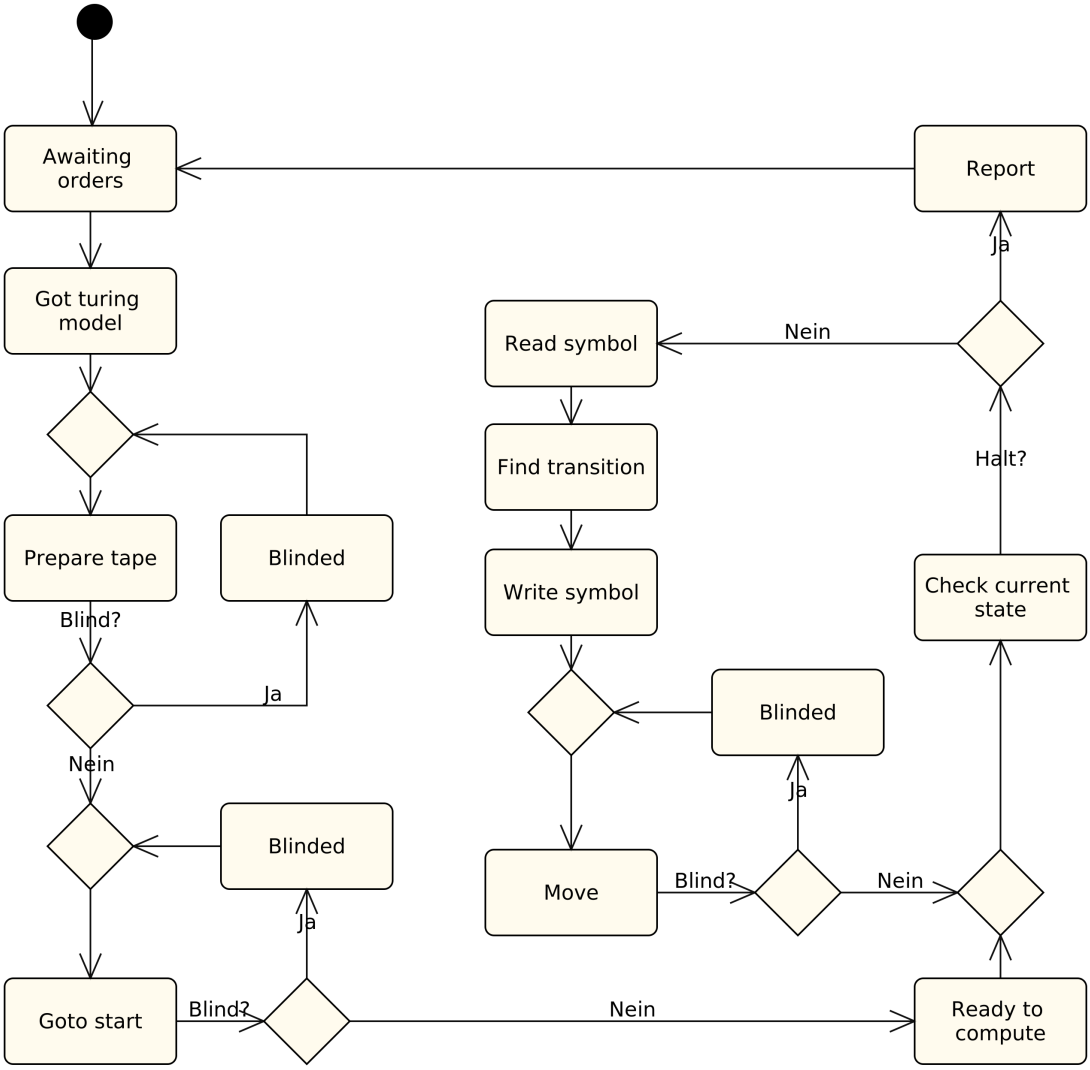


Abbildung 6.2: Endlicher Automat des NXT

Awaiting orders Es wird darauf gewartet, dass eine passende Kombination aus Turingmaschine und Konfiguration übermittelt werden. Sobald dies der Fall ist, geht der Automat in **Got turing model** über.

Got turing model Zustand ohne Arbeitsaufgabe. Alles ist bereit, das gegebene Modell aus Maschine und Konfiguration vorzubereiten. Geht nach **Prepare tape** über.

Prepare tape Sucht den Anfang des Turingbands und schreibt die gegebene Eingabe aus der Konfiguration. Notwendige Kalibrierungen können hier vorab vorgenommen werden. Geht nach **Goto start** über.

Goto start Bewegt die TM zur Startposition, wie sie in der gegebenen Konfiguration definiert ist. Der aktuelle Maschinenzustand wird auf den aus der mitgelieferten Konfiguration gesetzt. Geht nach **Ready to compute** über.

Ready to compute Zustand ohne Arbeitsaufgabe. Alle Vorbereitungen sind getroffen, um die TM in ihrer gewünschten Form auszuführen. Geht nach **Check current state** über.

Check current state Prüft, ob aktueller Zustand ein Haltezustand ist. Falls ja, geht der Automat nach *Report* mit einer Erfolgsmeldung über. Sonst geht der Automat nach **Read symbol** über.

Read symbol Der Lesekopf liest das aktuelle Bandsymbol aus. Wurde kein Bandsymbol erkannt, geht der Automat nach **Report** mit einer entsprechenden Nachricht über. Ansonsten geht er nach **Find transition** über.

Find transition Aus dem gegebenen Maschinenmodell wird für den aktuellen Maschinenzustand der passende Übergang ermittelt. Falls kein Übergang definiert ist, geht der Automat nach **Report** mit Fehlerbericht über. Sonst geht er nach **Write symbol** über.

Write symbol Schreibt mit Hilfe von dem aus **Find transition** gegebenen Übergang das Bandsymbol auf die aktuelle Position. Geht nach **Move** über.

Move Bewegt die Turingmaschine gemäß der gegebenen Übergangsanweisung. Würde die TM dabei die Bandgrenzen überschreiten, geht der Automat nach **Report** mit entsprechender Meldung über. Sonst wird der aktuelle Zustand auf den Folgezustand der Übergangsanweisung gesetzt und der Automat geht nach **Check current state** über.

Report Meldet einen Erfolgs- oder Fehlerfall aus dem vorhergehenden Zustand. Geht nach **Awaiting orders** über.

Blinded Jeder Zustand, in dem der Bewegungsapparat involviert ist, kann zu diesem Zustand wechseln. Die Aussage des Zustands lautet, dass der Farbsensor aufgrund der Lichtverhältnisse nicht in der Lage ist, eine korrekte Positionierung vorzunehmen.

Aus diesem Zustand kehrt der Automat stets wieder für einen erneuten Versuch, die aktuelle Aufgabe durchzuführen, in den aufrufenden Zustand zurück.

Der Automat durchläuft dabei die meisten Stadien ohne externe Anregung. Zu Demonstrations- und/oder Debugzwecken kann jedoch ein Stepping verlangt werden. In diesem Fall muss für jeden Zustandswechsel eine passende Stepping-Eingabe erfolgen.

6.2 Rolle des Client

Ein Client ist ein Kommunikationspartner, welcher sich vom Benutzer eine Turingmaschine samt Konfiguration modellieren lässt. Dabei wird nach dem Model-View-Controller Prinzip die Arbeit aufgeteilt. Die View (Darstellung) repräsentiert die GUI Elemente des Programms, welche vom Controller bedient werden. Der Controller hält sich als Model die aktuelle Arbeitskopie des Turingmodells. Auch die über das Kommunikationsmodul eingehenden Informationen sind als Model zu betrachten. Eine fertig modellierte Turingmaschine wird dann über das gemeinsame Kommunikationsmodul übertragen. Die Verbindung kann zu beliebigen Zeitpunkten ausgesetzt und später wieder aufgenommen werden.

6.2.1 Providermodell

TODO:
Ein Client könnte die Verwaltung übernehmen. Beispielsweise eine Webapplikation, bei der sich Clients des Clients um Turing-Arbeitszeit bewerben.

6.2.2 First come, first serve

TODO:
Bei Bluetooth und Kabelverbindungen mit unterschiedlichen Kollisionsproblemen: Wer sich zuerst mit dem Server verbindet, behält diesen, bis er ihn wieder freigibt.

Kapitel 7

Implementierung

7.1 Turingmaschine

Im weiteren Verlauf wird beschrieben, wie eine Turingmaschine in Java-Klassen modelliert wurde. Dabei sind viele, leicht variierende, Möglichkeiten denkbar gewesen, es galt jedoch die eingeschränkte Javabibliothek für NXT zu berücksichtigen.

Definition 4. *Die Lego-Turingmaschine ist eine TM, für die zusätzlich gilt:* $|\Gamma| \leq 3$.

Das Klassenmodell der LTM muss diese Eigenschaften abbilden können. Dabei wird unter Anderem auf eine separate Abbildung von F verzichtet, da die Kenntnis über die Menge für den Betrieb nicht notwendig ist. Die Klasse `TuringMachine` hält eine Liste ihrer Zustände (`TuringState`). Ein Zustand hält eine Sammlung über die eigenen Übergänge. Ein Übergang ist als Objekt der Klasse `TransitionResult` repräsentiert, welches den Folgezustand (`resultState`, Q), das zu schreibende Symbol (`resultWrite`, Γ) und die Folgebewegung (Enumeration `TuringMovement`, $\{LEFT, RIGHT, HALT\}$) beinhaltet.
Die Abbildung der formalen Definition der Turingmaschine in der entsprechenden Java-Klasse geschieht wie folgt:

- Q ist über eine Liste von `TuringState`s erreichbar
- Σ ist das `byte[] inputAlphabet`, welches alleine zur Prüfung des eingegebenen Worts notwendig ist.
- Γ benötigt keine weitere Repräsentation. Intern wird immer mit dem Maximum von drei möglichen Bandsymbolen gerechnet. Sollten davon welche nicht verwendet werden, so werden diese in `TuringState` als unbenutzt markiert.

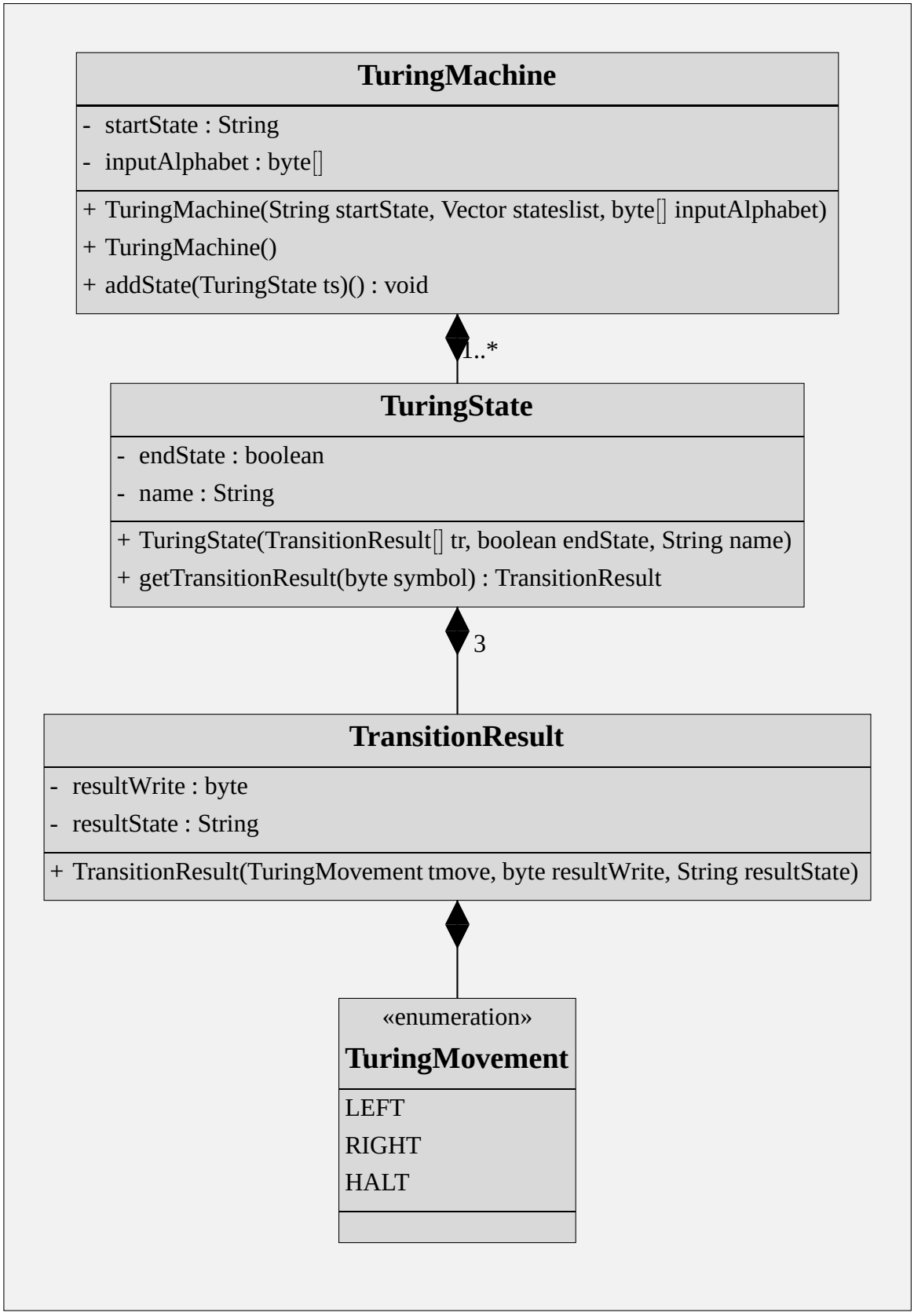
- δ ist über alle `TuringState` Objekte verteilt. Jeder Zustand kennt seine möglichen Übergänge.

- $\square$ ist ein konstantes Bandsymbol. Sein Bytewert lautet $0x2$.

- $q_0 \in Q$ ist im Attribut `startState` hinterlegt

- F kann, wenn nötig, aus Q extrahiert werden.

Klassendiagramm

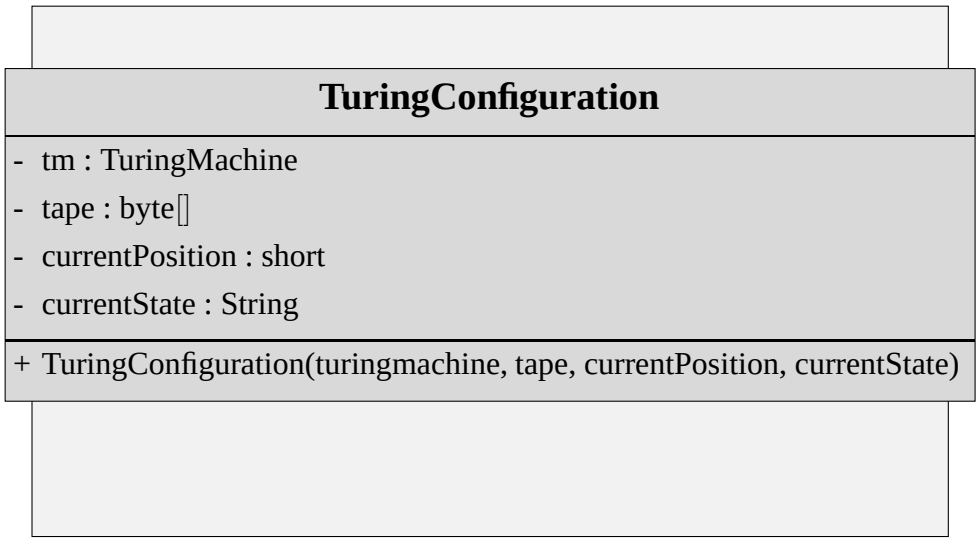


7.1.1 Konfiguration einer Turingmaschine

Eine Konfiguration stellt auf Seiten des NXT Brick die Arbeitsgrundlage dar, die während des Ablaufs der Turingmaschine benötigt wird. Die NXT Software hält über eine Konfiguration den aktuellen Zustand der Turingmaschine fest.

Die Klasse `TuringConfig` hält die aus der Definition beschriebenen Informationen. Konfiguration und Maschine sind auch hier zwei getrennte Modelle, die separat behandelt werden. So können unterschiedliche Konfigurationen auf der mechanischen TM ausprobiert werden, ohne dabei in jedem Schritt das Maschinenmodell mitsenden zu müssen. In der Gegenrichtung kann die NXT Software dem PC die aktuelle Konfiguration schicken, um sie zu speichern und zu einem späteren Zeitpunkt wiederaufzunehmen.

Klassendiagramm



7.2 Ansteuerung der Mechanik

Dieser Abschnitt beschreibt die softwareseitige Implementation der drei mechanischen Komponenten Schreibeinheit, Lesekopf und Bewegungsapparat. Alle Klassen sind nach dem Singletonpattern entworfen, da dies der Realität der mechanischen TM entspricht, die Ansteuerung vereinfacht und eine Menge von Fehlern ausschließt, die durch Mehrfachinstantiierung auftreten können.

7.2.1 Schreibeinheit

Die Schreibeinheit (Klasse `WriteHead`) interagiert mit der Außenwelt über einen Motor, dessen Vor- und Rückwärtsbewegung die Schreibschiene bewegt. Eine feine Kalibrierung ist hier vor allem nötig, um Bandsymbolsteine in die mittlere Position schieben zu können sowie die Schreibschiene zu parken. In Parkposition können Lesekopf und Bewegungsapparat ihre Arbeit aufnehmen, ohne dass die Schiene deren Bewegungen blockieren würde. Während einer Kalibrierung der gesamten Maschine sollte die Schreibschiene vor dem Bewegungsapparat eingestellt werden. Diese erfolgt nach dem Aufruf von `calibrate()` wie folgt ab:

- Bewege die Schiene nach links bis sich die Schiene nicht mehr weiterbewegt (Blockade).
- Streckenmesser auf 0 setzen
- Bewege die Schiene nach rechts bis zur Blockade
- Streckenmesser auslesen, als Referenzstrecke speichern
- Bewege die Schiene um 45 Prozent der Strecke nach links.

Die Schreibschiene befindet sich danach in der Parkposition.
Mittels der Methode `write(byte symbol)` lässt sich nun ein Bandsymbol versetzen. Beim Setzen von Steinen an die linke oder rechte Position wird bis zur Blockade der Stein verschoben und dann zur Parkposition zurückgekehrt.
Für das Setzen in die mittlere Position wird wie folgt verfahren: Zuerst wird der Stein auf die rechte Position gesetzt. Damit ersparen sich Fallunterscheidungen für Schreiboperationen, welche abhängig von der aktuellen Position des Steins sind. Anschließend bewegt sich die Schiene um 53 Prozent der Referenzstrecke nach links. Damit steht der Stein korrekt und die Schreibschiene kehrt in die Parkposition zurück.

7.2.2 Lesekopf

Die Klasse `ReadHead` beinhaltet zwei Exemplare der Klasse `TouchSensor`, welche die beiden Tastköpfe repräsentieren, sowie einen `Motor`, welcher die Tastköpfe bewegt. Die Kalibrierung kann hier statischer verlaufen als bei der Schreibschiene. Eine Aufwärtsbewegung bis zur Blockade stellt sicher, dass die Tastköpfe die Abläufe von Bewegungsapparat und Schreibschiene nicht behindern. Da die exakte Position der Tastköpfe

ausschließlich vom Motor abhängt, kann die Abwärtsbewegung zum Auslesen des Bandsymbols fix auf 85 Grad im Vollwinkel gesetzt werden. Eine Abwärtsbewegung bis zur Blockade wäre auch möglich, erhöht aber das Entgleisungsrisiko und ist langsamer.

7.2.3 Bewegungsapparat

Die Klasse `Movement` des Begungsapparats hält ein `Motor` und ein `ColorLightSensor` Objekt vor. Nach einer gelungenen Kalibrierung sind mittels `move(TuringMovement)` Bewegungen auf dem Band möglich. Die Methode wirft eine `OutOfTapeException`, falls das Ende des Bands mit dieser Bewegung überschritten würde. Während der Bewegung kann es dazu kommen, dass der Farbsensor keine zuverlässigen Daten mehr liefern kann, beispielsweise durch zu starke Beleuchtung. In diesem Fall hält der Bewegungsapparat an, bis dieser Zustand vorüber ist.

7.2.3.1 Anpassungen am Farbsensor

Die Kalibrierung macht es erforderlich, dass der Farbsensor eindeutige und verlässliche Informationen zurückliefert. Es stellte sich heraus, dass die Klasse `ColorLightSensor` in leJOS 0.85 nur Nullwerte für die Licht- und Farbintensität zurückgibt und dies auch seit längerem den Entwicklern bekannt ist (vgl. [For09b]). Ein Fix ist bereits in der aktuellen HEAD Revision des Codes, diese ist aber instabiler als Version 0.85 und hat ungelöste, neue Probleme im Motor und anderen Klassen. Ein Patch auf 0.85, der allein den Farbsensor behebt, wurde angeboten und wird in der weiteren Entwicklung verwendet.

Verbesserung der Markierungserkennung

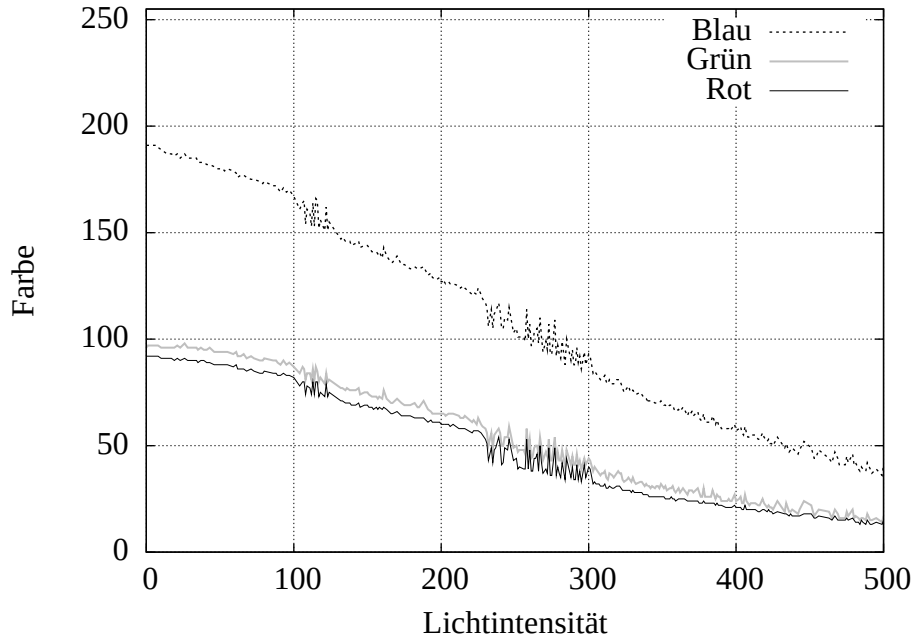
Vom Farbsensor werden fünf mögliche Rückgabewerte erwartet. Rot, grün und blau sollen nur über ebenso farbigen Legosteinen gemessen werden. Jede andere Farbe wird als Zwischenraum interpretiert. Ein Fehlerwert bedeutet, dass eine Messung wegen schlechter Lichtverhältnisse nicht möglich ist. Die aktuelle Implementation der Methode `readColor()` kann diese Unterscheidungen nicht oder nicht in der gewünschten Genauigkeit liefern. Beispielsweise zeigt sie auf Höhe von Schatten der Farbsteine den Wert `Color.BLUE` an. Auch führt starke Beleuchtung zu unzuverlässigen Ergebnissen, sowie Holzoberflächen. Für den Fall der Falsch-Blauerkennung ergeben sich die Werte aus Tabelle 7.1 auf der nächsten Seite.

Tabelle 7.1: Gemessene Rohdaten und Farbinterpretation auf einer Schattenfläche

Rot	137
Grün	138
Blau	138
Helligkeit	30
readColor()	Color.BLUE

Die Lösung besteht darin, die RGB Werte in die Farberkennung miteinzubeziehen. Mittels `readValues(int[] vals)` lassen sich diese abrufen. Diese Daten sind Werte in Bytegrenzen zur Intensität des Rot-, Grün- und Blauanteils, sowie ein Helligkeitswert zwischen 0 und 1023 [1]. Es wurden Messungen vorgenommen, um zu bestimmen, wie deutlich sich die gewünschten Farbtöne von den falsch-positiven Messungen unterscheiden (siehe Abbildungen 7.1, 7.2 auf der nächsten Seite und 7.3 auf der nächsten Seite).

Abbildung 7.1: Gemessene Rohdaten auf blauem Markierungsstein



[1] Dies ist ein Wert aus der API Dokumentation. Helligkeiten jenseits der 700 zu erzeugen ist während dieses Projekts nicht gelungen

Abbildung 7.2: Gemessene Rohdaten auf grünem Markierungsstein

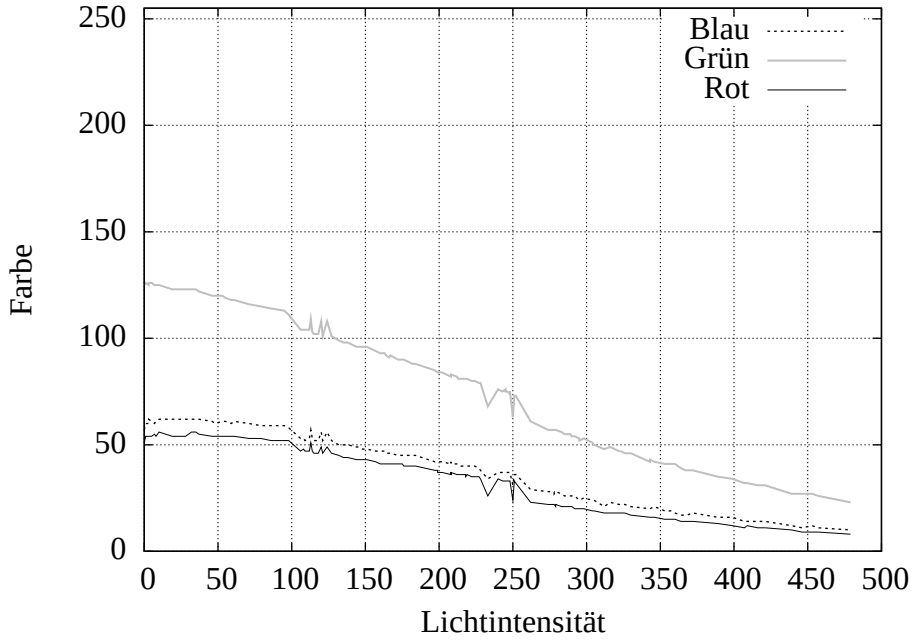
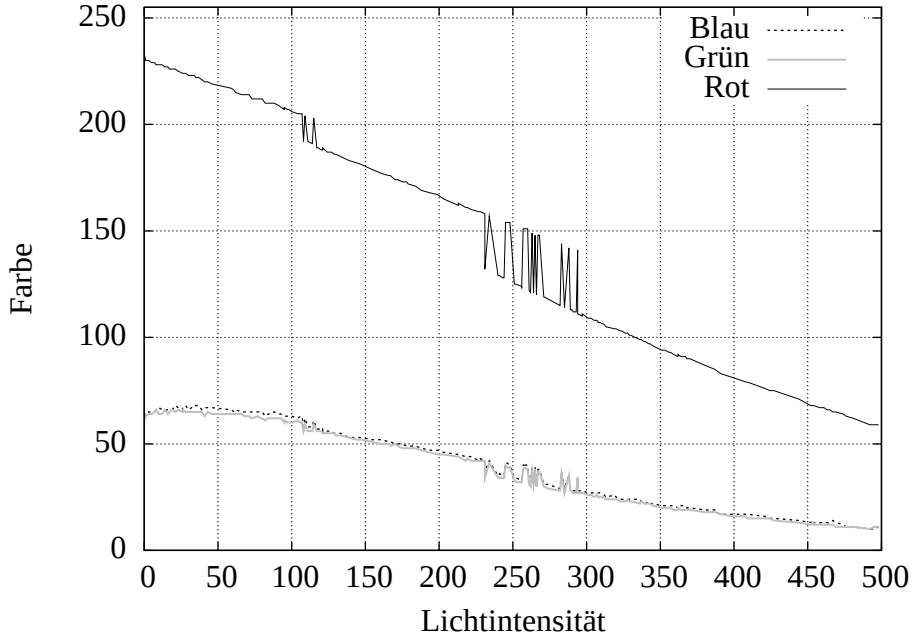


Abbildung 7.3: Gemessene Rohdaten auf rotem Markierungsstein



Die Methode `readColorSensor()` in der Klasse `Movement` übernimmt diesen Zweitabgleich. Neben den obigen Fallprüfungen gibt sie `null` als Fehlerwert zurück, wenn der Helligkeitswert zu hoch liegt.

7.2.3.2 Kalibrierung

Ziel der Kalibrierung ist es, eine definierte Startposition auf dem Band einzunehmen. Dabei fährt die Maschine rechtswärtige Richtung, bis sie den roten Endmarker erreicht. Von dieser Markierung wird vorausgesetzt, dass die Maschine sich nicht rechts davon befinden kann, ohne entgleist zu sein. Eine Prüfung auf Widerstand während der Bewegung wird ebenfalls vorgenommen. Ausserdem wird ein Timeout gesetzt, bis zu dem eine gültige Markierung überfahren worden sein muss, damit im Fall einer Entgleisung oder eines Sturzes weitere Bewegungen vermieden werden.

7.3 Kommunikation

7.3.1 Vorgaben von LeJos

LeJos bietet Kommunikationsmöglichkeiten via USB und Bluetooth. Beide Optionen setzen verschiedene Vorbereitungen voraus. Bluetoothgeräte sollten vor einem Verbindungsaufbau mit dem NXT Brick gepaart werden. Für die USB Verbindung müssen betriebssystemabhängig Treiber installiert und Berechtigungen gesetzt werden. Nach der Wahl des physikalischen Mediums muss man sich für ein Transportprotokoll entscheiden. Bei allen Kommunikationsmöglichkeiten zwischen einem PC und einem NXT Brick handelt es sich um PTP Verbindungen. Zur Auswahl steht die roher Datenübertragung, ein Paketmodus und ein proprietäres Übertragungsformat von Lego. Die letzte Option scheidet sofort aus, da LeJos damit keine beliebige Datenübertragung anbieten kann. Nach näherer Untersuchung der letzten beiden Optionen fiel die Wahl auf den Paketmodus, da er als einziger bislang zuverlässig arbeitet. Ein Paket hat eine fixe Bytelänge für Nutzdaten, welche davon abhängig ist, ob Bluetooth (250 Byte) oder USB (63 Byte) verwendet wird.

7.3.2 Blockierende Verbindungstrennung

Es ist unter Linux nicht möglich gewesen, eine bestehende USB Verbindung zu trennen, wenn das USB Kabel gezogen oder die Gegenstelle abgeschaltet worden ist und noch Daten zur Übertragung vorliegen. Nach Rücksprache mit den LeJos Entwicklern lag das Problem an einer undifferenzierten Behandlung von Fehlercodes der libusb Bibliothek. Die entsprechende JNI Funktion sieht wie folgt aus:

```
1  JNIEXPORT jint JNICALL Java_lejos_pc_comm_NXTCommLibnxt_jlibnxt_1send_1data(JNIEnv *env,
       jobject obj, jlong nxt, jbyteArray data, jint offset, jint len)  {
2    int ret;
3    char *jb = (char *) (*env)->GetByteArrayElements(env, data, 0);
4    if (len > MAX_WRITE) len = MAX_WRITE;
5    ret = nxt_write_buf((long) nxt, jb+offset, len);
6    (*env)->ReleaseByteArrayElements(env, data, (jbyte *) jb, 0);
7     // Assume any error is a  timeout!!! Need to fix this!
8     if (ret < 0) ret = 0;
9    return ret;
10 }
```

Quellcode 7.1: JNI Funktion zum Senden an ein USB Gerät

Die in Zeile drei aufgerufene Funktion übergibt den Rückgabewert der libusb Funktion `usb_write_bulk()`. Diese Art der Fehlerbehandlung hat zur Folge, dass ein `flush()`, welches ein letztes Mal von `close()` aufgerufen wird, in einer Endlosschleife den vermeintlichen Timeout abwartet. Jede Methode, die diese Funktion aufruft, hält den Rückgabewert 0 für einen vorübergehenden Fehler. Trotz der Tatsache, dass hier in der Implementation keine negativen Werte als Fehlercodes vorkommen wird dies von aufrufenden Methoden berücksichtigt. Negative Werte resultieren in einer allgemein formulierten `IOException`. So konnte ohne weitere Veränderungen an anderen Stellen des Codes Zeile acht aus obigem Listing durch

```
1  if((ret == -ETIMEDOUT) || (ret == -EAGAIN)) ret = 0;
```

ersetzt werden. Die Rückgabewerte von `usb_write_bulk()` im negativen Bereich repräsentieren `errno` Fehlercodes mit negativem Vorzeichen. Die modifizierte Version des JNI Codes befindet sich in der Toolchain auf dem beiliegenden Datenträger. Auch sollten Folgeversionen von LeJos diesen Fehler nicht mehr haben.

7.3.3 Vereinheitlichung der Kommunikationsklassen

Die LeJos Klassen zur paketweisen Datenübertragung haben unterschiedliche Eigenschaften auf beiden Seiten der Verbindung, trotz der Tatsache, dass sie beide das Java Interface `InputStream` zur Datenübertragung bieten. Das Lesen von Daten ist auf Seite des PCs nur blockierend möglich. Erschwerend kommt hinzu, dass es keine Möglichkeit gibt, herauszufinden, ob Daten zum Empfang vorliegen. Die dafür vorgesehene `available()` Methode liefert in jedem Fall 0 zurück. Nicht blockierendes Schreiben ist auf beiden Seiten bedingt möglich. Ein FIFO Puffer empfängt und hält die Daten, bis ein `read()` ausgeführt wird. Der Puffer hat eine Größe von zwei Paketen. Ist der Puffer

voll, blockiert ein korrespondierendes `write()`, bis auf Empfangsseite wieder Platz im Puffer ist.

7.3.3.1 LegoStream als Lösung

Zur Kompensation wurden im Rahmen dieser Thesis die Kommunikationsklassen erweitert. Nach außen hin werden nun die Klassen `LegoInputStream` und `LegoOutputStream` mit den folgenden Verbesserungen angeboten:

Nonblocking I/O

Die Klasse `LegoInputStream` kapselt ein Objekt der Klasse `PerpetualThreadedReader`. Diese beheimatet einen Thread, welcher in einer Endlosschleife von einem gegebenen `InputStream` liest. Die gelesenen Bytes kommen in einen Container mit threadsicherem Zugriff. Auf sie kann über den `LegoInputStream` mittels `read()` zugegriffen werden. Eine Methode `available()` gibt Auskunft über den Füllstand dieses Containers. `LegoOutputStream` beinhaltet einen `PerpetualThreadedWriter`, welcher als separater Thread das Schreiben in einen `OutputStream` übernimmt. Hier kommt u.A. das Problem aus 7.3.2 auf Seite 35 zum Tragen. Sowohl beim Input- als auch beim OutputStream führt eine IOException zum Ende des arbeitenden Threads und die Verbindung wird als getrennt angenommen.

Paketgrenzenunabhängige Übertragung

Ein `flush()` in dem von LeJos angebotenen `OutputStream` ist im Paketmodus ohne Wirkung, solange nicht genügend Daten für ein komplettes Paket vorhanden sind. Um diese Begrenzung für einen `LegoOutputStream` zu umgehen, wird zu jedem Paket eine Zusatzinformation über die Menge an Nutzdaten mitgegeben. Die `flush()` Methode schreibt diese Information an den Paketanfang, gefolgt von den Nutzdaten. Der Rest des Pakets wird mit Nullen gefüllt. Auf Seiten des `LegoInputStream` wurde der `PerpetualThreadedReader` erweitert, um das Nutzdatenbyte auszulesen und den relevanten Teil des Pakets in den Lesecontainer zu speichern.

7.3.3.2 Fazit

Die Erweiterungen mittels dieser Klassen bieten eine gute Grundlage zur bequemen Kommunikation zwischen NXT Brick und PC, welche allgemein genug gehalten sind, um

auch für andere Projekte in Frage zu kommen. Die langfristig wünschenswerte Alternative bleibt aber das Schließen der Implementationslücken in der LeJos Bibliothek.

7.3.4 Nachrichten als atomare Informationsträger

Durch die Anforderungsanalyse wurde klar, dass Nachrichten zwischen den Kommunikationspartnern ausgetauscht werden müssen, welche die Paketgrenzen von Bluetooth und USB Verbindungen übersteigen. Das Kommunikationsmodell benötigt daher eine andere Art und Weise, Informationen im LegoStream zu übertragen. Wichtigstes Attribut muss dabei sein herauszufinden, wann eine Information vollständig angekommen ist, um sie dann vollständig auszulesen und interpretieren zu können. Diese Infomationen werden im weiteren Verlauf Nachrichten (Interface `Message`) genannt.
Nachrichten sind Bytecontainer, die bis zu 2^{32} Bytes an Nutzdaten tragen können. Diese Nutzdaten können beispielsweise in einem zweiten Schritt als serialiserte Java-Objekte interpretiert werden.

7.3.4.1 Message API

Eine Nachicht (interface `Message`) kann jede Klasse sein, die sich selbst als Bytearray repräsentieren kann. Es können auch primitive Bytearrays selbst verwendet werden. Ein `MessageWriter` fügt beim Versand der Nachricht den nötigen Header hinzu. Das Bytearray bildet die Nutzlast. Das Format eines Nachrichtencontainers sieht wie folgt aus:

Tabelle 7.2: Aufbau einer Message

Nachricht			
Format:	ID	Länge	Nutzlast
Offset/Länge:	0/4	3/4	6/Länge

7.3.4.2 MessageReader und MessageWriter

Die Klassen MessageReader und -Writer erhalten einen korrespondierenden LegoStream. Die API ist dabei simpel gehalten: Ein korrekt erzeugter Writer schickt mit `writeMessage(Message)`, bzw. `writeMessage(byte[])` eine Nachricht. In

verkehrsüblichen Zeiten kommt die Nachricht bei dem Reader der Gegenseite an.
Auf Seiten des PCs sieht ein Nachrichtenversand wie folgt aus (ohne Fehlerbehandlung):

```
1  void messageExample_PC()
2  {
3      NXTConnector conn = new NXTConnector();
4      // Verbinde mit der Gegenstelle am USB Port. Paketmodus
5      if (conn.connectTo("usb://", NXTComm.PACKET))
6      {
7          LegoOutputStream lout = new LegoOutputStream(conn.getDataOut());
8          MessageWriter writer = new MessageWriter(lout);
9          byte[] payload = {0,2,4,5,7,9,12,42};
10         writer.writeMessage(payload);
11     }
12 }
```

Quellcode 7.2: Beispiel eines Nachrichtenversands auf Seiten des PCs

Auf NXT Seite sieht der initiale Verbindungsaufbau anders aus. Desweiteren sind die Methodenaufrufe der LeJos Klassen unterschiedlich. Sobald aber LegoStreams erzeugt worden sind, sind Klassen, Methoden und der tatsächliche Funktionsumfang einheitlich. Eine wie im oberen Listing versendete Statusnachricht kann wie folgt empfangen werden:

```
1  void messageExample_NXT()
2  {
3      // Warte bis Verbindung im Paketmodus steht
4      USBConnection conn = USB.waitForConnection(0, lejos.nxt.comm.NXTConnection.PACKET);
5      LegoInputStream lin = new LegoInputStream(conn.openDataInputStream());
6      MessageReader reader = new MessageReader(lin);
7      while(!reader.available()){
8          // "Busy waiting" bis Nachricht empfangen
9      }
10     byte[] payload = reader.getMessagePayload();
11 }
```

Quellcode 7.3: Beispiel eines Nachrichtenempfangs auf Seiten des NXT

7.3.4.3 Message deserialisieren

Wie aus dem Bytearray wieder ein `Message` Objekt zu erzeugen ist (falls überhaupt nötig), bleibt dem Entwickler vollständig selbst überlassen. Zur Vereinfachung dieser Arbeit bietet `MessageReader` jedoch an, mithilfe einer ihm übergebenen `MessageFactory` die Deserialisierung vorzunehmen.

Da im Rahmen dieses Projekts eine Vielzahl unterschiedlicher Nachrichtenklassen zusammengekommen ist, wird diese Methode in Form der Klasse `MessageFactoryImpl`

genutzt.

7.3.4.4 Instruktionen und Antworten

Implementationsbedingt gibt es Nachrichten, die nur von PC zu Brick gesendet werden. Diese werden im Folgenden “Instruktionen” (im Klassendiagramm `Instruction`) genannt. Dazu gehört beispielsweise der Befehl, eine Konfiguration einzulesen oder die Frage nach dem Status des NXT. Dieses sendet daraufhin “Antworten” (`Response`), wie z.B.: eine Statusinformation. Diese Unterteilung ist für die Implementation hilfreich, um für beide Seiten Handler zu schreiben.

Tabelle 7.3: Eine Instruktion (hier: Statusabfrage) in einem Nachrichtencontainer

Nachrichtenheader			Instruktion	
Format:	N.ID	Länge	I.ID	Nutzlast
Offset/Länge:	0/4	3/4	0/3	2/3-Länge
“GetStatus”:	“mes:”	0x0000004	“in:”	0x01

Übersicht Instruktionen

Get status:
Fragt nach dem aktuellen Automatenzustand. Erwartet **Status** als Antwort.

Set configuration
Übermittelt eine nun zu verwendende Konfiguration. Falls die neue Konfiguration nicht zur aktuellen Maschine passt, führt dies dazu, dass der Zustandsautomat auf Awaiting orders zurückfällt.

Set machine
Übermittelt eine nun zu verwendende Turingmaschine. Es gelten dieselben Konsequenzen für ein inkompatibles Maschine-Konfigurationspaar wie bei **Set configuration**.

Get machine
Fordert die Übermittlung der momentan verwendeten Turingmaschine. Erwartet **get machine result** als Antwort.

Get configuration
Fordert die Erzeugung einer Konfiguration aus der aktuellen Situation der mechanischen Turingmaschine an. Erwartet **get configuration result** als Antwort.

Step forward

Im Steppingmodus: Der nächste Arbeitsschritt soll vollzogen werden.

Start/Stop

Schaltet den Steppingmodus oder hebt ihn auf.

Do move

Bewegungsapparat soll eine Links- oder Rechtsbewegung ausführen. Erwartet **move result** als Antwort.

Read current cell

Lesekopf soll aktuelles Bandsymbol auslesen und zurückgeben.

Write current cell

Schreibkopf soll ein bestimmtes Bandsymbol schreiben.

Übersicht Antworten

Status

Übermittelt den Automatenzustand

Exception response

Allgemeine Fehlermeldung an den Client.

Move result

Rückgabewert des **do move** Befehls. Meldet ein OK, wenn die Bewegung erfolgreich ausgeführt wurde, ansonsten beschreibt die Nachricht den Fehlergrund.

Read cell result

Gibt zurück, welches Bandsymbol durch **read current cell** gelesen wurde.

7.3.4.5 Kontrollnachrichten

Eine dritte Variante von Nachrichten bilden jene, die sich von der Klasse `Control` ableiten. Diese sind dafür vorgesehen, um den Verbindungszustand zu kontrollieren. Diese Nachrichten können von beiden Seiten verschickt werden.

Übersicht Kontrollnachrichten

Ping

Erreichbarkeitsfrage an die Gegenstelle. Enthält eine zufällige Zahl als Nutzlast, die ein **Pong** zurückschicken muss.

Pong

Erreichbarkeitsantwort auf ein **Ping**.

Close

Ankündigung, die Verbindung zu schließen.

Diese Kontrollnachrichten haben ähnliche Probleme wie Software-Flußkontrollen: Wenn eine Seite mit der Verarbeitung von Nachrichten nicht mehr nachkommt, kommen diese Kontrollnachrichten zu spät an. Hier muss über eine Priorisierung der Bearbeitungsreihenfolge nachgedacht werden.

7.3.4.6 Visitorpattern zur Fallbehandlung

Jede Nachricht erscheint beim Empfänger bislang im besten Fall als `Message` Objekt. Um abhängig von der Nachricht unterschiedliche Funktionen auszuführen, wurde das Visitorpattern eingesetzt. So gibt es für jede Unterklasse von Nachricht ensprechende `Hosts` und `Visitors`. Über diesen Weg lässt sich die Polymorphie der Nachrichtenklassen ausnutzen, um Handlermethoden in Klassen zu schreiben, zu denen sie thematisch gehören.

```
1 public interface InstructionVisitor {
2     public void visit(GetStatus mes);
3     public void visit(UseConfiguration mes);
4     public void visit(StepForward aThis);
5     public void visit(StartStop aThis);
6 }
```

Quellcode 7.4: Visitor zur Behandlung von Instruktionen

7.3.4.7 Fazit und Ausblick

Das hier verwendete Nachrichtenmodell ist die logische Erweiterung der `LegoStream`s, um Informationen beliebiger Art und Größe zu übertragen. Neben diesem Projekt wurde es auch schon für kleine Hilfsprogramme verwendet, um die Farbsensordaten aus 7.2.3.1 auf Seite 32 auf den PC zu übertragen. Das Nachrichtensystem zeigt damit seine Modularität und kann problemlos für andere LeJos Projekte verwendet werden. Die definierten Instruktionen und Antworten sind speziell für die Turingmaschine gedacht, die Kontrollnachrichten sind auch für Folgeprojekte interessant. Mit fortschreitender Entwicklung der LeJos Firmware wird es auch dem von Java bekannten Funktionsumfang weiter angenähert werden. Features wie Objekt(de-)serialisierung und Reflection liegen auch für ein Embedded System wie NXT im Bereich des Möglichen und sollten ihren Weg in die Quellcodes finden. Damit könnte dieses Nachrichtenmodell abgelöst werden.

7.3.5 Verbindungen verwalten

Um die Verbindung zur Gegenstelle zu kontrollieren wurde das Interface `ConnectionManager` entworfen. Implementationen sind für die folgenden Aufgaben verantwortlich:

Automatischer/manueller Verbindungsaufbau

Es soll möglich sein, manuell die Verbindung aufzubauen. Alternativ kann auch ein Thread dafür abgestellt werden, sich um eine eventuelle Neuverbindung zu kümmern. Das blockierende Warten auf einen Verbindungspartner wird so ausgelagert. Dieser Thread schläft über die Zeit, in der eine Verbindung aktiv ist und wird per Interrupt geweckt, wenn eine erneute Verbindung versucht werden soll. Der Verbindungsaufbau läuft auf NXT und PC unterschiedlich ab. Durch dieses Interface wird es vereinheitlicht.

Überwachung durch Kontrollnachrichten

Die in 7.3.4.5 auf Seite 41 vorgestellten Nachrichten werden aus den bereitstehenden Nachrichten eines `MessageReader` gefiltert und beantwortet. Der selbe Thread kümmert sich um den Versand eigener Pingnachrichten zur Verbindungsprüfung. Nach einem definierten Timeout wird die Gegenstelle als unerreichbar angenommen und die Verbindung getrennt.

I/O Wrapping

Der Versand und der Empfang von Nachrichten obliegt allein dieser Klasse. Dies erlaubt die oben genannte `Control`-Priorisierung. Gerade der parallele Versand von `Control` Nachrichten und Instruktionen/Antworten muss synchronisiert werden.

Ausnahme Bluetooth

Es wurde probiert, einen vollautomatischen Verbindungsaufbau mit beliebigen Clients via Bluetooth zu bewerkstelligen. Das Hauptproblem stellt sich dabei im Koppeln (Pairing) der Kommunikationspartner. Es ist bislang nicht möglich, ausserhalb des Hauptmenüs der VM eine Kopplung vorzunehmen. Selbst wenn eine manuelle Kopplung stattgefunden hat dauert es unter Ubuntu 9.10 ca. 20 Sekunden bis zum Verbindungsaufbau. Der Einsatz von Input- und OutputStreams aus einer Bluetoothverbindung in Kombination mit den LegoStreams sowie den MessageReader/Writer ist trotzdem möglich.

7.3.5.1 Synchronisation auf LeJos

In dieser Klasse wird intensiv mit mehreren Threads gearbeitet, die auf gemeinsame Ressourcen zugreifen möchten:

1. Ein Neuverbindungsthread
2. Ein Verbindungskontrollthread
3. Externer Thread zur Nachrichten I/O und Verbindungskontrolle

Zur Synchronisation bietet LeJos eine Implementation des Java-Monitorsystems. Dadurch ist der Einsatz der bekannten Methoden `wait()` und `notify()` zur low-level Threadsynchronisation möglich. Befinden sich mehrere Threads im entry set desselben Monitors, so bevorzugt die LeJos VM 0.85 bei einem `notify()` zuerst höher priorisierte Threads. Bei gleicher Priorität wird ein Thread zufällig ausgewählt [LEJ09b]. Das *synchronized* Schlüsselwort ist ebenfalls implementiert.

Eine unnötige Blockade findet statt, wenn der externe Thread und der Verbindungskontrollthread gleichzeitig die Verbindung trennen wollen. Kritische Bereiche in `disconnect()` sind mittels *synchronized* gekapselt. Es würde für den zweitaufrufenden Thread genügen zu wissen, dass die Verbindung gerade getrennt wird.

Für dieses Projekt wurde ein einfaches Zählsemaphor nach [Dij68] implementiert. Neben `acquire()` und `release()` bietet es noch eine Methode `tryAcquire()`, welche sofort zurückkehrt, wenn die kritische Sektion nicht sofort betreten werden durfte.

7.4 NXT Applikation

7.4.1 View

Das LCD Display des NXT ist ein passiver Informant.

7.4.2 Automatenmodell

Jeder Zustand des Automaten (siehe 6.2 auf Seite 23) wird durch eine Klasse dargestellt, welche das Interface `AutomatonState` implementiert. Dieses definiert die Methoden nach dem Statepattern

- `AutomatonState work(Instruction mes)` und
- `AutomatonState work()`

Jede Zustandsklasse arbeitet unter Rücksichtnahme der Eingabe den von ihm repräsentierten Arbeitsschritt ab und meldet, mit welchem Zustand es weitergeht. Jeder Zustand ist dabei ein `InstructionVisitor` (siehe 7.3.4.6 auf Seite 42), wodurch sich die Beeinflussung des aktuellen Zustands durch eingehende Instruktionen erlaubt.

Zur Repräsentation des aktuellen Zustands wurden Stubs eingeführt. Zu jedem Zustand existiert eine Stubklasse, welche ohne Arbeitsfunktionalität sich selbst auf PC- als auch auf NXT Seite darstellen kann. So hält beispielsweise die Klasse `MoveStub` eine `TransitionResult`, aus dem klar wird, wohin sich die Turingmaschine als Nächstes bewegt, sowie ein Flag, ob Stepping aktiviert ist. Der eigentliche Zustand `Move` implementiert `AutomatonState` und hat Zugriff auf den Bewegungsapparat.

7.5 PC Applikation

Die Anwendung ist nach dem Model-View-Controller Prinzip aufgebaut. Die `TuringView` ist ein Interface, welches die Eigenschaften der GUI definiert, welche von der Klasse `MainWindow` implementiert werden. `TuringModel` ist die interne Repräsentation der Eigenschaften, die von `TuringMachine` und `TuringConfig` benötigt wird. Die Kontrollklasse `TuringControl` verknüpft beide Seiten und hält eine Referenz auf einen `ConnectionManager` zur Kommunikation mit der Turingmaschine.

7.5.1 View

Das Programm auf Seiten des PCs ist aus Benutzersicht in zwei Komponenten aufgeteilt. Die erste Komponente erlaubt das Modellieren einer Turingmaschine sowie einer geeigneten Konfiguration. Die zweite Komponente erlaubt die Verbindung zur Lego Turingmaschine. Die Trennung dieser Aufgabenbereiche wurde durch Tabs realisiert. Die gesamte GUI wurde mit Hilfe des graphischen NetBeans GUI Editors entwickelt.

7.5.1.1 Modellierungs-Tab

Auch die Modellierung ist in weitere Tabs unterteilt. Es wird unterschieden in der Automatenmodellierung und in der Konfigurationsmodellierung. Ein dritter Tab bietet eine

textuelle Zusammenfassung.

Automatenmodellierung

Im Rahmen der Modellierung wurde eine Eingabemaske entwickelt, die jeden Automatenzustand und jeden Übergang einzeln darstellt. Es wurde bewusst nicht in die Entwicklung einer besonders einfachen und intuitiven Modellierungsmaske investiert: Die Software JFLAP liefert dort bereits eine ausgereifte GUI mit graphischer Modellierung. Ihr Dateiformat für Turingmaschinen wird zum Import akzeptiert.

Automatenzustände sind in einer `JList` linksbündig gesammelt. Den Rest des Panels nimmt der Zustandseditor ein. Zur Verbesserung der Usability werden über `TuringControl` Buttons und andere Eingabefelder ausgeblendet, wenn diese gerade nicht genutzt werden können (bspw. weil kein Automatenzustand ausgewählt ist).

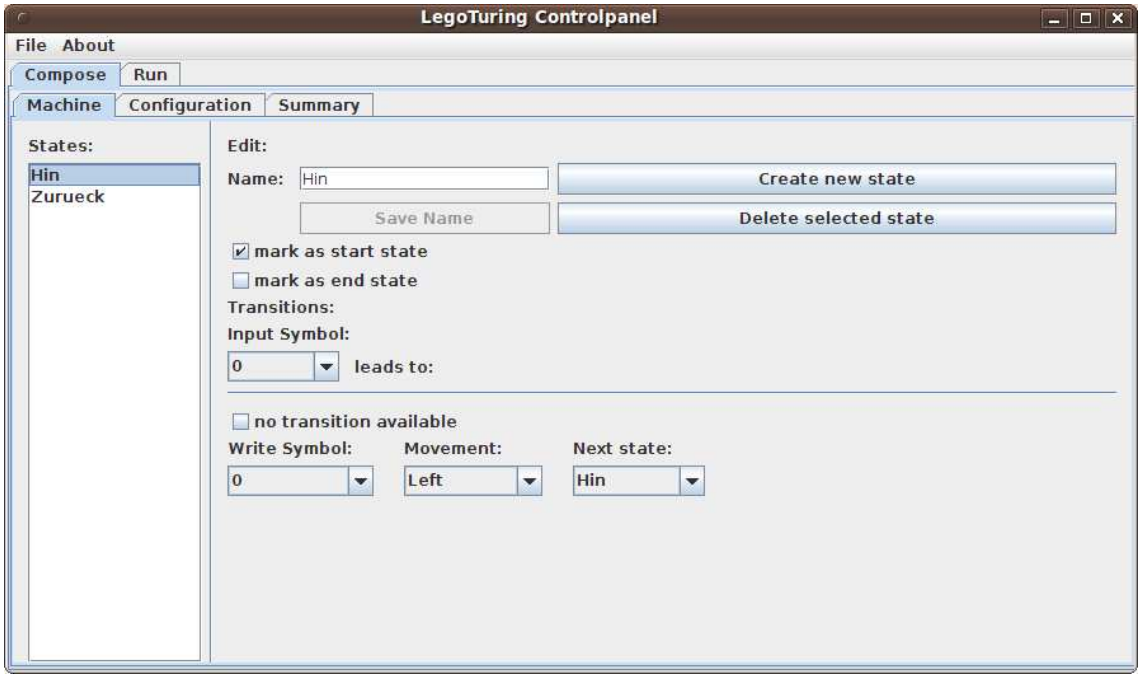


Abbildung 7.4: Editor der Automatenzustände

Konfigurationsmodellierung

Der zweite Tab bietet ein `JTextField`, um das Turingband mit Symbolen zu füllen. Da das □ Symbol verhältnismäßig schwer zu schreiben ist, wird für das leere Symbol ein 'x' verwendet. Ein `JSpinner` bietet die Möglichkeit, die aktuelle Bandposition der TM zu

setzen. Dabei ist gewährleistet, dass stets nur der Zahlenraum einstellbar ist, den das Text-Feld durch seine Eingabe erlaubt. Als besondere Hilfestellung wurde der `JSpinner` mit einem Observer versehen, der das beziehende Bandsymbol im `JTextField` selektiert und so die Positionierung erleichtert.

// TODO Screenshot

Zusammenfassung

In diesem Panel werden alle Eingaben mit Hilfe eines Java LaTeX Renderers für mathematische Formeln zusammengefasst. Die Möglichkeiten von *JLaTeXMath* sind noch begrenzt. So ist es leider nicht möglich, Umbrüche in die Formeln einzubauen. Das `TuringModel` liefert deshalb eine Liste von Strings, die einzeln in Formeln umgewandelt werden. In einem `ScrollPanel` werden die Formeln untereinander organisiert.

Zusätzlich bietet ein Button in diesem Tab die Möglichkeit, sich das aktuelle Modell als LaTeX Formel in die Zwischenablage zu kopieren. Über diesen Weg wurden die in diesem Dokument eingesetzten Beispiele modelliert und übertragen.

// TODO Screenshot

7.5.1.2 Verbindungs-Tab

Die Kontrolle über die Möglichkeiten des `ConnectionManager` befindet sich hier. Über einen Observer im `TuringControl` wird der Verbindungsstatus ständig aktualisiert. Die Funktionen sind in ihrer Tiefenwirkung von oben nach unten sortiert. Oben befinden sich die Basisfunktionen zur Verbindungskontrolle. Darauf folgen die Buttons, um die Turingmaschine und die Konfiguration zu übermitteln. Die Start- und Stopbuttons setzen den Steppingmodus. Neben einem Step können auch Einzelaktionen (Lesen, Schreiben, Bewegen) ausgeführt werden, die vom aktuellen Maschinenmodell losgelöst sind. Wie im Modellierungs-Tab wird durch Deaktivieren von Schaltelementen angezeigt, welche Optionen gerade verfügbar sind.

Ein Animationspanel bildet die aktuelle Handlung der mechanischen Turingmaschine in der Mitte des Panels ab.

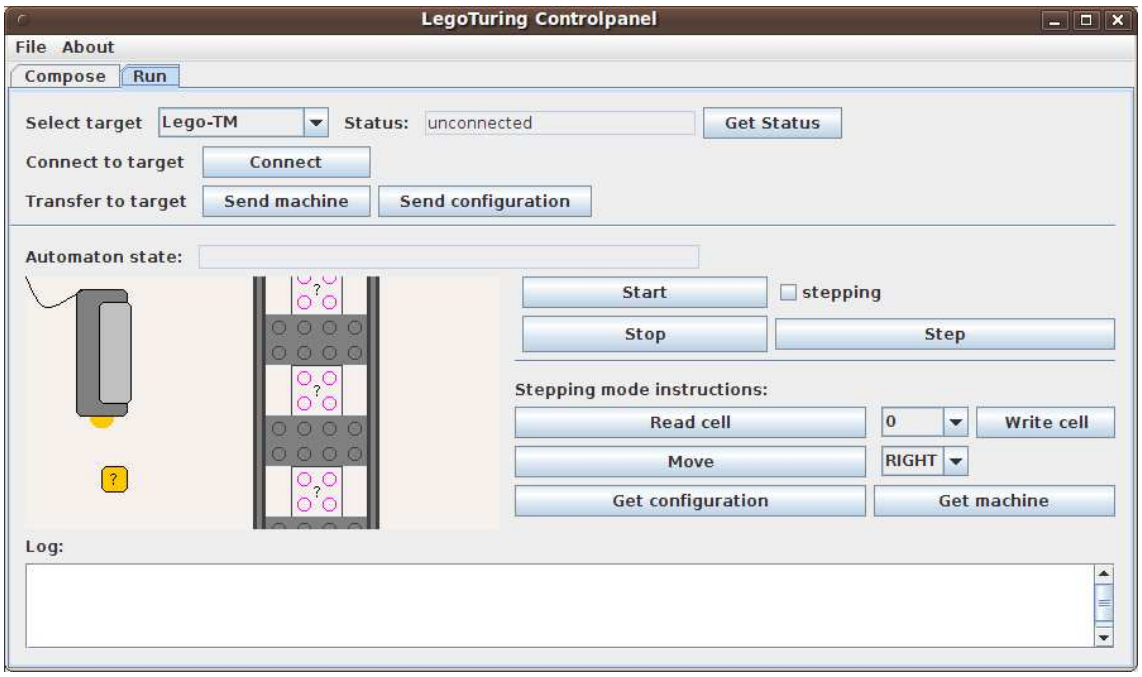


Abbildung 7.5: Verbindungstab

7.6 Web Applikation

Um die Funktionen des Exponats an Interessenten zur Verfügung zu stellen, bot sich zunächst die Möglichkeit an, beliebige Clients mittels Bluetooth mit dem NXT zu verbinden. Während der Arbeit an den Kommunikationsklassen wurde diese Option verworfen (siehe 7.3.5 auf Seite 43). Die nun gezogene Option bildet der Einsatz eines Computers, der sich in ständiger Verbindung mit dem NXT befindet und die Dienste der Lego-Turingmaschine als Netzwerk- oder Webapplikation weitergibt.

7.6.1 Sinatra auf JRuby

Aus älteren Projekten wie Cryptocore ([CRY]) ist das leichtgewichtige Webframework *Sinatra* ([SIN]) bekannt. Dieser wurde hier zum Mittel der Wahl, da die Applikation möglichst nur die bereits geschriebene API weiterleiten soll und keine größere Webseite geplant ist. Mit Sinatra lassen sich in wenigen Zeilen Webserver beschreiben, wie es das „Hello World“ Beispiel von 7.5 auf der nächsten Seite zeigt.
Der Einsatz von Sinatra auf JRuby, des Rubyinterpreters, welcher in einer Java VM läuft, hat mehrere Vorteile. JRuby kann auf das Threadmodell von Java zurückgreifen, um mehrere Verbindungsanfragen gleichzeitig zu bearbeiten. Der zweite, hier entscheidende Vorteil liegt in dem Einsatz von Javaklassen innerhalb von JRuby. So müssen lediglich für

Webseiten geeignete Model, View und Controller geschrieben werden.
Um in JRuby Javabibliotheken einsetzen zu können, muss diese als JAR Datei vorliegen. Die LeJos Bibliothek besteht allerdings auch aus nativen Komponenten. Diese müssen der Java VM noch zusätzlich zur Verfügung gestellt werden (unter Linux bpsw. über die Shellvariable `LD_LIBRARY_PATH`).
Jede im Skript eingesetzte Javaklasse muss einzeln inkludiert werden. Abkürzungen mittels Sternoperator wie in Java zum Import üblich gibt es nicht. Ausserdem muss bei Methodenaufrufen darauf geachtet werden, dass die Parameter den korrekten Datentyp haben, da nur wenige Basistypen aus Ruby auf Javaklassen abgebildet wurden (Beispiel: Javastrings und Rubystrings).

```
1 require 'rubygems'
2 require 'sinatra'
3 get '/hi' do
4   "Hello World!"
5 end
```

Quellcode 7.5: Fünzeiliger Webserver in Sinatra (Beispielcode aus [SIN])

```
# Unterstuetzung von Javaklassen wird eingefordert
include Java
# Laden der Bibliothek der PC Applikation
require "LegoTuring.jar"
require 'rubygems'
require 'sinatra'
require 'haml'

#Jede Javaklasse, die innerhalb des JRuby-Skripts benutzt wird, muss inkludiert werden
include_class("de.witzelbritz.legoturing.comm.manager.PCConnectionManager");
include_class("de.witzelbritz.legoturing.comm.messages.instructions.SetConfiguration");
include_class("de.witzelbritz.legoturing.comm.messages.instructions.SetMachine");
include_class("de.witzelbritz.legoturing.model.TuringConfig");
include_class("de.witzelbritz.legoturing.model.TuringMachine");
include_class("de.witzelbritz.legoturing.comm.messages.instructions.StartStop");

connman = PCConnectionManager.new();
#Verbinde gleich mit dem NXT
connman.connect(true);

#Zeigt ein Uploadformular
get '/' do
  haml :index
end

#Ziel der POST Anfrage des Uploadformulars
post '/upload' do
  #Uebermittelte Dateien auslesen und in ein Java byte[] konvertieren
  config_bytes = params[:configuration][:tempfile].readlines.to_s.to_java_bytes
  machine_bytes = params[:machine][:tempfile].readlines.to_s.to_java_bytes
  #Deserialisierungsaufrufe der Bibliothek einsetzen
  tm_config = TuringConfig::fromByteArray(config_bytes)
  tm_machine = TuringMachine::fromByteArray(machine_bytes)
  #Entsprechende Befehle der Turingmaschine geben:
  connman.sendMessage(SetConfiguration.new(tm_config));
  connman.sendMessage(SetMachine.new(tm_machine));
  connman.sendMessage(StartStop.new(false));
  return "OK";
end
```

Quellcode 7.6: Eine minimale Implementierung des LegoTuring Webservers

7.7 Andere Probleme und Workarounds

7.7.1 ArrayList

Von der Verwendung der generischen Datenstruktur musste abgesehen werden. Bei einem Einsatz in einfachen Applikationen stürzt die virtuelle Maschine ab und ließ sich nur durch Unterbrechung der Stromversorgung neustarten. Dieser Fehler ist seit Juni 2010 bekannt und in der HEAD Revision behoben [For10a]. Einen offiziellen Fix für Version 0.85 gibt es nicht. In diesem Projekt wird daher davon abgesehen, `ArrayList` zu verwenden und stattdessen auf die (spezifische) `Vector` zurückgegriffen.

7.8 Fazit

Die Software der Lego Turingmaschine verwendet (bei 73 eigenen Quellcodedateien) 206 Klassen und 878 Methoden und bleibt damit mit Abstand unter der 255 Klassen Grenze. Die Gesamtgröße des Executables beträgt 49120 Bytes und bleibt auch hier mit etwas Raum nach oben unter der 64 KB Grenze. Die Ausführungsgeschwindigkeit des Programms ist akzeptabel. Serialisierung und Deserialisierung großer Maschinen/Konfigurationen geschieht in Sekundenbruchteilen. Es konnte mit Erfolg um die bekannten und entdeckten Grenzen der LeJos VM gearbeitet werden.

Kapitel 8

Anwendungsbeispiele

8.1 Busy Beaver

Der von Tibor Radó initiierte Wettbewerb (siehe [**?**]) stellt folgende Herausforderung: Welche Turingmaschine mit binärem Alphabet $1, /square$ und n Zuständen schafft es, die meisten Einsen auf das Band zu schreiben und anzuhalten? Die geschaffene mechanische Turingmaschine hat endliches Band mit ca. 30 Zeichen Platz. Es ließe sich zeigen, wie kleine Programme schon einen hohen Speicherbedarf entwickeln können.

8.2 Game of Life

8.3 Tic Tac Toe

Kapitel 9

Literaturverzeichnis

[CRY] *Cryptocore - Grøstl webhash (beta).* http://cryptocore.cs.hs-rm.de

[Dij68] DIJKSTRA, Edsger W.: *Cooperating sequential processes.* http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF. Version: 1968

[Flo] FLOTHOW, Sebastian: *Beschreibung der mechanischen Turingmaschine.* http://www.flothow.de. – TBA

[For08] FORUM, LeJOS: *whats the max num of parameters a constructor can have?* http://lejos.sourceforge.net/forum/viewtopic.php?t=674. Version: 01 2008

[For09a] FORUM, LeJOS: *Maximum program size (extra libraries)?* http://lejos.sourceforge.net/forum/viewtopic.php?t=1742. Version: 10 2009

[For09b] FORUM, LeJOS: *Ultrasonic/Color Sensor don't return values. [sic].* http://lejos.sourceforge.net/forum/viewtopic.php?t=1706. Version: 09 2009

[For10a] FORUM, LeJOS: *Data Abort - No clue.* http://lejos.sourceforge.net/forum/viewtopic.php?t=2183. Version: 06 2010

[For10b] FORUM, LeJOS: *Removing the 256 classes limit in VM.* http://lejos.sourceforge.net/forum/viewtopic.php?t=2081. Version: 03 2010

[Hav09] HAVE, Mikkel Vester; Sean Geggie; Anders Nissen; M.: *Lego of Doom*. `http://legoofdoom.blogspot.com/`. Version: Januar 2009

[JFL] *JFLAP - Duke University*. `http://www.cs.duke.edu/csed/jflap/`

[JIT] *Java goes to space - Roboterprogrammierung in Java am Beispiel eines Lego-Mindstorms-Weltraumroboters*. `http://it-republik.de/jaxenter/artikel/Java-goes-to-space-0232.html`

[LDO] *Lego of Doom - Turing Machine Architecture*. `http://legoofdoom.blogspot.com/2008/12/turing-machine-architecture.html`

[LEJa] *LeJos - Sourceforge*. `http://lejos.sourceforge.net/`

[LEJb] *LeJosRT - Sourceforge*. `http://sourceforge.net/projects/lejosrt/`

[LEJ09a] *LeJos NXT API: Class File*. `http://lejos.sourceforge.net/nxt/nxj/api/java/io/File.html`. Version: 09 2009

[LEJ09b] *LeJos NXT API: Class Object*. `http://lejos.sourceforge.net/nxt/nxj/api/java/lang/Object.html`. Version: 09 2009

[NXT] *nxtOSEK - Sourceforge*. `http://lejos-osek.sourceforge.net/`

[Ric53] RICE, H. G.: Classes of Recursively Enumerable Sets and Their Decision Problems. In: *Trans. Amer. Math. Soc.* 74 (1953), S. 358–366

[SIN] *Sinatra webserver*. `http://www.sinatrarb.com`

[TIN] *TinyVM - Sourceforge*. `http://tinyvm.sourceforge.net/`

[Tur36] TURING, A. M.: On computable numbers, with an application to the Entscheidungsproblem. In: *Proc. London Math. Soc. 2(42)* (1936), S. 230–265

[Tur48] TURING, Alan: *Intelligent Machinery*. 1948. – report for the National Physical Laboratory