Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Scenarios meets Lego Mindstorms - Von szenariobasierten Verhaltensspezifikationen zu ausführbarer Software für Robotersysteme

Bachelorarbeit

im Studiengang Informatik

von

Victor Hartmann

Prüfer: Prof. Joel Greenyer
Zweitprüfer: Matthias Becker
Betreuer: Prof. Joel Greenyer

Hannover, 21. April 2015

Erklärung der Selbständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 21. April 2015

Victor Hartmann

Zusammenfassung

In dieser Arbeit untersuche ich die Möglichkeit, ein System informell durch Szenarien zu beschreiben, diese mit Hilfe von Diagrammen zu formalisieren, genannte Diagramme automatisch zu verifizieren und schließlich in Programmcode umzuwandeln, um sie am echten System zu testen. Dieses echte System wird von einem Lego EV3 Roboter repräsentiert. Das Ziel der Arbeit ist es, diesen Ansatz am Beispiel durchzuführen und zu bewerten. Zur Verwirklichung kombiniere ich mehrere Werkzeuge: Für die Erstellung der Diagramme verwende ich das Eclipse Modeling Framework, speziell *Papyrus*. Die Diagramme werden durch das Eclipse Plugin *ScenarioTools* um Modalitäten erweitert und des Weiteren werden sie automatisch validiert sowie zu Zustandsautomaten synthetisiert. Für die Lego Mindstorms Roboterprogrammierung verwende ich das Java-basierte *leJOS*, ein Eclipse Plugin sowie Linux-basiertes Betriebssystem für den Lego EV3 Roboter.

Abstract

In this thesis I examine the possibility to describe a system informally with the use of scenarios, to formalize these with the help of diagrams, automatically verify named diagrams and finally transform them into program code to test them on an existing system. This existing system is represented by a Lego EV3 robot. The objective of this work is to execute and rate that approach on an example. To realize all of this I combine several tools: for the construction of the diagrams I am using the Eclipse Modeling Framework, especially *Papyrus*. Using the Eclipse Plugin *ScenarioTools* the diagrams are extended with modalities and furthermore they get automatically validated and synthesized to state-machines. For the programming of the Lego Mindstorms robot I am using the Java-based *leJOS*, an Eclipse plugin and Linux-based operating system for the Lego EV3 robot.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In der Industrie wie auch als Privatanwender stößt man auf eine immer größer werdende Anzahl von softwaregesteuerten Systemen, welche in sicherheitskritischen Bereichen eingesetzt werden. In dieser Art von Systemen arbeiten einzelne Komponenten einander zu um verschiedenste Aufgaben zuverlässig zu erfüllen. Das effiziente und beständige Zusammenspiel der Software, Hardware und Umwelt sind wichtig für die Sicherheit und die Zuverlässigkeit dieser Systeme.

Um die Symbiose dieser Einzelteile zu gewährleisten, setzt man beim Software Engineering in einem frühen Stadium der Entwicklung auf eine szenariobasierte Spezifikation, mit der der Ablauf von Aktionen und die Zusammenarbeit einzelner Komponenten in einem System geplant und modelliert werden kann. Dabei werden Probleme, Aktivitäten und Informationen zusammengetragen, die für das zu entwickelnde System eine Rolle spielen. Auch mögliche Risiken und Fehlerquellen aus Umwelteinflüssen werden hier einbezogen. Dieser Entwurf basiert auf der Erstellung von Szenarien, die das Verhalten des Systems in bestimmten Situationen beschreiben.

Hierbei kann es sich zum Beispiel um einen Produktionsroboter in einer Fabrik handeln. Die Aufgabe dieses Roboters sei es, zu erkennen, wann ein neues Werkstück eintrifft, dieses zu greifen und auf einer anderen Plattform abzulegen (ProductionCell Beispiel aus [4]). Diese Situation gilt es zu formalisieren und Eventualitäten abzudecken. Was passiert, wenn ein Werkstück eintrifft, während der Roboter gerade das vorherige Werkstück aufgehoben hat? Arbeiten die verschiedenen Komponenten an einem seriellen oder parallelen Ablauf? Diese Fragen können mit Szenarien abgedeckt und so eine Überraschung während des Prototypentests vermieden werden.

In der Bachelorarbeit soll eine Methode untersucht und entwickelt werden, die die Bewertung einer szenariobasierten Spezifikation auf der letzten Ebene, also der Ausführung im System, ermöglicht. Programmieren ist lästig. Vor allem für Menschen mit Ideen, die damit gar keine oder nur wenig Erfahrung haben. Deshalb gibt es Mittel und Wege, um Ideen schneller

und leichter zu verwirklichen, sodass sie nicht verworfen werden, nur weil ein großer Aufwand mit ihrer Realisierung verbunden ist. Ein Mittel um dies zu bewerkstelligen ist, die Programmierarbeit möglichst gering zu halten. Die Vision ist natürlich, die Programmierung automatisch geschehen zu lassen. Dies soll mithilfe einer Codegenerierung geschehen, die eine Spezifikation in für LEGO EV3 Roboter ausführbaren Code transformiert. Durch den damit möglichen Test der Spezifikation auf dem echten System fallen Widersprüche und Unvollständigkeiten in einem sehr frühen Stadium der Entwicklung auf, was die folgenden Phasen deutlich beschleunigt.

1.1 Problemstellung

Bei der Implementierung von Software für sicherheitskritische Systeme ist es wichtig, nicht nur das Verhalten des Systems zu spezifizieren, sondern auch das der Umwelt, da immer Ausnahmefälle eintreten können, die im einfachen Entwurf nicht modelliert werden können. Um den Aspekt der Umwelt in den Entwurf miteinzubeziehen bietet sich der szenariobasierte Ansatz an. Das Verhalten aller beteiligten Komponenten wird in Szenarien modelliert und Ausnahmefälle damit möglichst vollständig abgedeckt. Dabei kann es aber zu Widersprüchen kommen, wenn zum Beispiel zwei Szenarien gegensätzliches Verhalten fordern. Für die weitgehend automatisierte Evaluation eines szenariobasierten Entwurfs, also das Kenntlichmachen solcher Widersprüche und Unvollständigkeiten, wird seit einigen Jahren das Eclipse Plugin ScenarioTools [5] entwickelt. Dieses Werkzeug bedient sich der formalen Synthese, um eine Strategie zu finden, die es ermöglicht das System so auszuführen, so dass keine Eingabefolge eine Verletzung eines Szenarios nach sich zieht. Die Spezifikationen werden dafür in modalen Sequenzdiagrammen, kurz MSDs, modelliert und anschließend von dem Tool synthetisiert. ScenarioTools erstellt aus den MSDs einen Zustandsautomaten und erlaubt es dem Entwickler die Zustandsübergänge innerhalb der Entwicklungsumgebung zu simulieren und das Systemverhalten in bestimmten Situationen zu beobachten. Es gibt jedoch keine Möglichkeit diesen Automaten außerhalb der Simulation an dem echten System zu testen. Um das zu tun, muss der szenariobasierte Entwurf manuell in Code übersetzt werden. Es ist gewährleistet, dass der Entwurf fehlerfrei ist und annähernd vollständig was eventuell eintretende Ereignisse betrifft, jedoch kann nicht garantiert werden, dass die Implementierung genauso fehlerfrei sein wird. Denn diese wird manuell durchgeführt, was immer eine Fehlerquelle darstellt.

1.2 Lösungsansatz

Um dieses Problem zu lösen, wäre es angenehm, wenn aus dem von ScenarioTools synthetisierten Zustandsautomaten automatisch der Pro-

grammcode für das System generiert werden könnte. Somit würden sich keine neuen Fehler bei der Implementierung einschleichen und die komplette Entwicklung würde beschleunigt werden, da die Implementierungsphase wegfällt. In dieser Arbeit werde ich mich damit beschäftigen, wie man eine solche Codegenerierung umsetzen kann und ob der Lösungsansatz einen Vorteil gegenüber schon bestehenden Methoden, einen Lego Roboter zu programmieren, bietet.

1.3 Zielsetzung der Arbeit

Das Ziel dieser Arbeit ist es, eine Spezifikation am durchgängigen Beispiel zu erproben. Das bedeutet, jeden einzelnen Schritt, angefangen bei dem Entwurf über die Entwicklung eines Szenarios, die Analyse durch ScenarioTools, die Codegenerierung und Evaluation am Beispiel durchzugehen und das Ergebnis zu bewerten. Des Weiteren werde ich die Entwurfsmethode bewerten und offene Herausforderungen feststellen. Um die Transformation der Szenarien in Code zu testen, werde ich ein simples Beispiel erläutern. Um die Machbarkeit und offene Herausforderungen zu ergründen, erarbeite ich ein größeres Beispiel. Eine Idee ist ein Kartengeber-Roboter, der die Anzahl von Spielern abfragt und dann entsprechend Karten an verschiedenen Stellen vergibt. Zusätzlich soll mit einem Sensor geprüft werden, wann die eingelegten Karten vollständig ausgegeben sind. In diesem Fall soll der Roboter entsprechend reagieren und das Programm abbrechen. Der Qualitätsaspekt Zuverlässigkeit wird von mir besonders wichtig eingeschätzt und wird dementsprechend überprüft. Dazu erstelle ich MSDs, in denen zum Beispiel Sensoren angesprochen werden, die im Szenario nicht vorgesehen sind oder das doppelte Ansteuern von Motoren (evtl. gleichzeitig in verschiedene Richtungen) modelliert wurde. Diese Fehlerquellen sollen vor Ausführung von ScenarioTools festgestellt werden, damit keine fehlerhafte Implementierung an den Roboter gelangt und ausgeführt wird.

1.4 Struktur der Arbeit

Diese Arbeit ist wie folgt strukturiert: Ich beginne mit einer Einleitung zur Erläuterung der Motivation und des Ziels der Arbeit in Kapitel 1. In Kapitel 2 stelle ich die Grundlagen zum Verständnis der folgenden Kapitel vor. Dazu gehören das Konzept des szenariobasierten Entwurfs und die dazugehörigen Diagrammarten und der Zusammenhang mit ScenarioTools. Anschließend stelle ich kurz das Lego Mindstorms Projekt vor und schließlich folgt eine Beschreibung von leJOS, einer Java Umsetzung für Lego Mindstorms Roboter. In Kapitel 3 erfolgt eine Skizze der Fallstudie. Danach beschäftige ich mich mit der Ausführung von Szenarien auf einem Lego Roboter und letztlich folgen Fragen an die Entwurfs- und Entwicklungs-Methodik, die

während der Problemanalyse aufgetreten sind. Die Codegenerierung wird danach in Kapitel 4 erläutert. Das Kapitel 5 beinhaltet den beispielhaften Vorgangsablauf von der Erstellung eines Entwurfs, über die Spezifikation, die Entwicklung und Validierung bis hin zur Codegenerierung und Ausführung auf dem Roboter. Dieser Ablauf wird anschließend anhand der Fragen bewertet, die sich während der Problemanalyse gestellt haben. In Kapitel 6 stelle ich eine verwandte Arbeit vor, bei der Lego Roboter mithilfe von Diagrammen programmiert wurden. Im abschließenden Kapitel 7 folgt eine Zusammenfassung meiner Arbeit sowie ein Ausblick auf zukünftige Möglichkeiten.

Kapitel 2

Grundlagen

2.1 Szenariobasierter Entwurf

Bei der Entwicklung von Systemen wird ein bestimmter Ablauf eingehalten. Ein Projekt beginnt mit der Planung, Feststellung des Problems und Ideen zur Lösung des Problems, bzw. der Realisierung einer Idee. Danach folgt eine Spezifikation in Form eines Entwurfs. Dieser Entwurf soll das Ganze möglichst stark abstrahieren um eine Formalisierung zu ermöglichen. Dies wird so gestaltet, dass sich aus dem Entwurf möglichst leicht ein Konzept oder ein Prototyp ableiten lässt. Das Konzept des *szenariobasierten Entwurfs* baut darauf auf. Hier wird speziell der Aspekt der Ausnahmebehandlung hervorgehoben. Dazu versucht man möglichst viele Szenarien zu formalisieren, die verschiedene Situationen abbilden, in die ein System gelangen kann und beschreiben, was das System in dieser Situation tun kann, tun muss oder welche Aktion sogar verboten ist. Szenariobasierte Entwürfe werden mithilfe von Diagrammen modelliert. Ein besonders wichtiges ist dabei das *Modale Sequenzdiagramm*.

2.2 UML Sequenzdiagramme

Mit Sequenzdiagrammen lassen sich Programmabläufe modellieren und so anschaulich machen. Einzelne Objekte werden durch sogenannte Lebenslinien dargestellt. Über Aufrufe können die Objekte, wie in Abb. 2.1, innerhalb ihrer Lebenslinien kommunizieren und Nachrichten austauschen.

2.3 Modale Sequenzdiagramme

Modale Sequenzdiagramme (im Folgenden MSD) sind UML Sequenzdiagramme die um eine Modalität *temperature* erweitert wurden. Der Vorschlag dazu kam 2008 von Harel und Maoz [7]. Die Erweiterung durch Modalitäten bringt einen großen Vorteil mit sich.

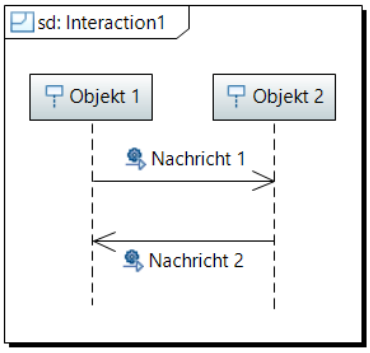

Abbildung 2.1: Ein UML Sequenzdiagramm

2.3.1 Modalitäten

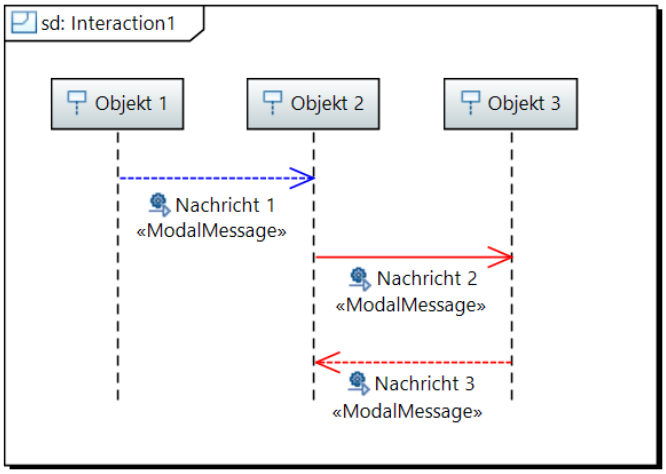

Abbildung 2.2: Ein modales Sequenzdiagramm (MSD)

Nachrichten können entweder *hot* oder *cold* sein. Man unterscheidet die Temperatur durch die Farben rot für *hot* und blau für *cold*.

Brenner et al. verwenden zusätzlich die Modalität *execution kind* [3]. Nachrichten können dann zusätzlich *executed* (muss gesendet werden) oder *monitored* (muss nicht gesendet werden) sein. Der Status wird mit durchgezogenen Linien für *executed* und gestrichelten Linien für *monitored* visualisiert. In Abb. 2.2 habe ich ein solches MSD beispielhaft modelliert. Die Objekte, die von den Lebenslinien dargestellt werden, können entweder der Umwelt oder dem System angehören. In diesem Beispiel gilt die Annahme, dass *Objekt 1* der Umwelt (Environment) angehört und die *Objekte 1 und 2* dem System. Die *Nachricht 1* ist cold und monitored, *Nachricht 2* ist hot und executed und *Nachricht 3* ist hot und monitored. Die Interaktion bedeutet,

dass es Systemdurchläufe geben kann, in denen *Nachricht 1* nicht ausgelöst wird. Wenn sie jedoch ausgelöst wird, darf keine weitere Nachricht auslösen, bis *Nachricht 2 und 3* gesendet wurden.

2.3.2 Requirements und Assumptions

Eine Voraussetzung der szenariobasierten Spezifikation mit ScenarioTools ist die Unterscheidung zwischen Requirement MSDs (Anforderungen an das System) und Assumption MSDs (Annahmen über die Umwelt). In den Requirements wird das gewünschte Kernverhalten des Systems beschrieben. Die Assumptions beinhalten dagegen Annahmen über die Umwelt.

2.4 ScenarioTools

In Eclipse lassen sich durch das Eclipse Modeling Framework (EMF) die Diagramme eines Entwurfs modellieren. Um Entwickler bei der Validierung szenariobasierter Entwürfe zu unterstützen, haben Greenyer et al. [5] ein Plugin für Eclipse entwickelt. In seiner Dissertation beschreibt Greenyer den Einsatz von szenariobasierten Entwürfen in Bezug auf mechatronische Systeme mit Hilfe von ScenarioTools [4].

Das Systemverhalten in den Szenarien wird durch Sequenzdiagramme, die durch ScenarioTools um Modalitäten erweitert werden, modelliert. Der Entwickler gibt seinen Entwurf als Sammlung von Requirement und Assumption MSDs ein, ScenarioTools analysiert die modellierten Abläufe und entwickelt durch formale Synthese, falls möglich, eine Strategie, in der bei jeder Eingabefolge keins der spezifizierten Szenarien verletzt wird. Diese Strategie lässt sich mittels des Play-out Algorithmus simulieren.

2.5 Play-out

Auf der ScenarioTools Homepage wird Play-out folgendermaßen beschrieben (sinngemäß übersetzt) [5, Play-out]:

> *Das grundlegende Prinzip des Algorithmus ist, dass wenn ein environment event auftritt und dies zur Folge hat, dass es ein oder mehrere aktive MSDs mit aktiven Systemnachrichten gibt, soll der Algorithmus nichtdeterministisch eine zugehörige Nachricht senden, solange dies keine safety violation in einem anderen aktiven MSD nach sich zieht. Der Algorithmus sendet so lange wiederholt Systemnachrichten, bis kein aktives MSD mit aktiven Nachrichten übrig ist. Danach wartet der Algorithmus auf das nächste environment event und der Prozess wiederholt sich. Wenn der Algorithmus einen Zustand erreicht, in dem es aktive*

Nachrichten gibt, die jedoch alle zu einer safety violation führen würden, nennt sich dies eine violation.

2.6 Lego Mindstorms

Lego Mindstorms ist eine Erweiterung der Lego Technic Familie um eine programmierbare Steuereinheit und Sensoren. Mit ihr lassen sich programmierbare Systeme mit Lego Bausteinen konstruieren. Kern des Systems ist der programmierbare *Brick*. In der neuesten Lego Mindstorms Version, genannt EV3, kann der Brick mehr als je zuvor. Er ist inzwischen ein kleiner Linux Computer und bietet auch PC typische Schnittstellen: einen USB Host Anschluss, einen microSD Karten Einschub, ein Bluetooth Modul sowie Eingabeknöpfe und ein Display. Darüber hinaus natürlich noch jeweils vier Anschlüsse für Sensoren und Motoren. Über die Bluetooth Schnittstelle ist es möglich, mehrere EV3 Bricks miteinander kommunizieren zu lassen. So können große Projekte realisiert werden, bei denen viele Motoren und Sensoren verwendet werden müssen.

Die Zubehörteile reichen von einfachen Motoren über den Berührungssensor bis hin zu einem Ultraschallsensor. Einige Unternehmen haben sich sogar darauf spezialisiert feinere und sogar komplett neue Sensoren zu entwickeln, wie zum Beispiel einen Temperatursensor oder ein sehr genaues Gyroskop. Eine mitgelieferte PC Software erlaubt die visuelle Programmierung des *Bricks* im Baustein-Verfahren. Der *Brick* wird per USB oder kabellos (über einen Wifi USB Stick am Brick) mit dem PC verbunden. Die von Lego mitgelieferte Software-Umgebung lässt sich erweitern, sodass Drittanbieter-Sensoren darin über vom Hersteller zur Verfügung gestellte spezielle Blöcke verwendet werden können.

2.7 LeJOS

LeJOS ist eine Java Umsetzung für Lego Mindstorms Robotersysteme [2]. Es wurde 1999 von José Solórzano als Hobby-Projekt entwickelt und wurde inzwischen von vielen Entwicklern erweitert. Brian Bagnall, Jürgen Stuber und Paul Andrews wurden später die drei Hauptentwickler [1, Geschichte]. Es besteht aus drei Teilen: ein Eclipse Plugin für die Kommunikation mit dem Brick (Konfiguration und Upload der Programme), eine Library um auf die Funktionen des EV3 zugreifen zu können und ein Linux basierendes Betriebssystem für den EV3 Brick. Bei frühere Versionen von Lego Mindstorms musste die Firmware des Bricks mit der leJOS Firmware überschrieben werden. Da der EV3 Brick jedoch selbst auf Linux setzt, ist es möglich leJOS von der microSD Karte zu starten, um so die Original-Software trotzdem weiterhin verwenden zu können. Dieser Punkt war für meine Arbeit sehr von Vorteil, da ich die beiden Systeme miteinander

vergleichen musste. So hatte ich die Möglichkeit, ohne großen Aufwand zwischen den Betriebssystem zu wechseln.

LeJOS bietet die Möglichkeit Programme für den Brick in Java zu programmieren und somit komplexere Abläufe zu implementieren, als mit der Lego Software möglich wäre. Der Mehrwert gegenüber der Lego Software laut dem Wikipedia Eintrag zu leJOS [1] ist in folgende Punkte zusammengefasst:

- *ausgefeilte Parallelverarbeitung durch Multithreading inklusive aller Java-Bordmittel (synchronize, das Paket java.util.concurrent)*
- *einfache Möglichkeit, den Source-Code mit Versionskontroll-Systemen zu versionieren*
- *Robotik-API: behavior-based robotic, Steuerungsmodelle für Lenk- und Kettenantriebs-Roboter, abstrakte Navigations-Klassen*

Kapitel 3

Problemanalyse

3.1 Skizze der Fallstudie

Bei der Entwicklung der Software für einen Lego Roboter gibt es mehrere Möglichkeiten. Lego selbst bietet eine grafische Programmierumgebung an, die auf das modulare Baustein-Prinzip aufsetzt. Die Funktion wird dabei abstrahiert dargestellt, sodass auch Laien ein leichter Einstieg geboten wird und die Frustrationsgrenze erst bei komplexen Programmabläufen erreicht wird. Bei der Entwicklung mit dieser Software steht die Usability im Vordergrund. Bausteine werden intuitiv verbunden, wenn zwei Komponenten nicht kompatibel sind, lassen sie sich nicht verbinden. Unterschiede bei Parametern werden symbolisch sichtbar gemacht. Bei der Implementierung mit dieser Programmierumgebung findet kein Software-Entwurfsprozess statt. Es ähnelt eher dem trial-and-error Prinzip. Dies birgt viele Nachteile und ist bisweilen zeitaufwendig, jedoch bei kleineren Systemen oft der schnellste Weg, um an einen funktionierenden Prototypen zu kommen. Es stellt sich die Frage, bis zu welchem Grad man mit der Lego Software imstande ist, Funktionsabläufe zu implementieren. Ein großer Nachteil dieser einfachen Art der Implementierung ist zum Beispiel die fehlende Behandlung von Ausnahmefällen und Umwelteinflüssen.

Dies und noch viel höhere Komplexität kann mit alternativen Programmiersprachen, wie zum Beispiel leJOS, auf dem Lego EV3 Roboter realisiert werden. Kompliziertere Programme erfordern jedoch vor der Implementierung einen Entwurf, bei dem sich der szenariobasierte Ansatz anbietet.

3.2 Ausführung der Szenarien auf dem Lego Roboter

Am Beispiel SimpleBrick wird der Unterschied zwischen der Implementierung in der Lego Software gegenüber leJOS deutlich. In diesem Beispiel gibt es das Herzstück des Roboters, Brick genannt, einen Berührungssensor sowie

einen Motor. Sobald der Berührungssensor berührt wird, soll der Motor eine Drehung um 90 Grad vollziehen. Wie in Abb. 3.1 zu sehen, erfordert dieses Beispiel nur wenige Blöcke in der Lego Software. Eine Schleife überprüft kontinuierlich, ob der Berührungssensor gedrückt wird. Wenn das der Fall ist, wird die Funktion im Inneren ausgeführt. Diese lässt den Motor die 90 Grad Drehung durchführen. Die Implementierung eines solchen einfachen Vorgangs nimmt mit der Lego Software nur wenige Minuten in Anspruch. Außerdem ist das Ergebnis durch die grafische Darstellung schön anzusehen und auch Dritten wird die Funktion des Programms schnell bewusst. Im Gegensatz dazu steht der Programmieraufwand der leJOS Implementierung (siehe Anhang A „Code“ C). Viele Zeilen Code entstehen alleine durch die Initialisierung der verschiedenen Komponenten. Der eigentlich auszuführende Code stellt nur einen kleinen Teil vom Ganzen dar, siehe Codeabschnitt 3.1. Hier sei anzumerken, dass die Methode *TouchSensorPressed()* nicht Teil von leJOS ist, sondern eine von mir eingeführte Abstraktion zur Erleichterung der Implementierung. Der Inhalt der Methode kann im Anhang C eingesehen werden.

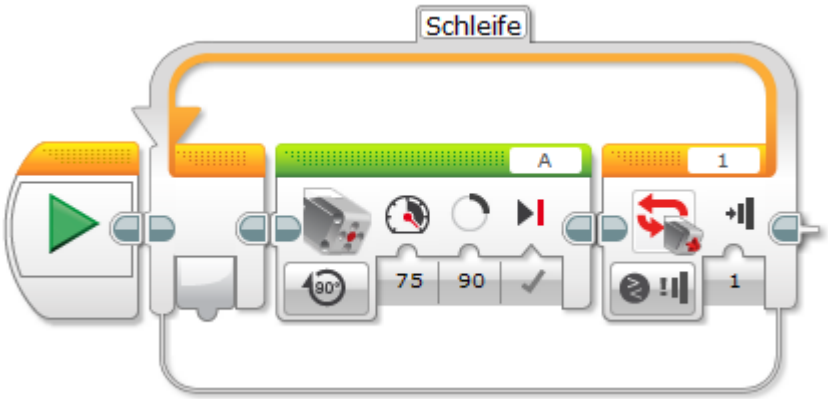

Abbildung 3.1: SimpleBrick in der Lego Programmierumgebung

Listing 3.1: Die Main Methode der SimpleBrick leJOS Implementierung

```
public static void main(String[] args) {
     // end program when escape-button is pressed on the EV3 brick
     if (Button.ESCAPE.isDown())
        System.exit(0);

     // rotate the motor 90 degrees if touch sensor is pressed
     while (Button.ESCAPE.isUp()) {
        if (TouchSensorPressed())
           motor.rotate(90);
     }
```

```
}
```

Werden die Abläufe jedoch komplexer, so wird auch die grafische Programmierung zunehmend unübersichtlich. Zwar gibt es die Möglichkeit Programmabschnitte in eigene Blöcke auszulagern, aber die Ausführung muss immer noch manuell getestet werden. Das macht die Fehlersuche schwierig und bei großen Projekten fast unmöglich. Dies wird am Beispiel des Kartengeber-Roboters deutlich. Der Roboter wartet auf die Eingabe einer Spielerzahl, danach wird die Anzahl der Karten abgefragt. Nach der Eingabe der Daten, gibt der Roboter die entsprechende Anzahl Karten für jeden Spieler aus. Dafür dreht er sich ein Stück nachdem er die Karten für einen Spieler ausgegeben hat. Wenn während des Ausgebens der Karten der Kartenstapel geleert wird, wird das Programm abgebrochen. In Abb. 3.2 sieht man den Aufbau des Programms in der Lego Software. Die türkisen Blöcke sind selbst erstellt und beinhalten selbst auch wieder Blöcke. Wie aufwendig das komplette Programm ist, wird deutlich, wenn man den Inhalt der selbst erstellten Blöcke sieht. Abbildungen aller Programme von CardBot sind im Anhang B zu finden.

Die Implementierung via leJOS scheint für so ein kleines Programm wie SimpleBrick eher aufgeblasen und kompliziert, was jedoch nur der erstmaligen Initialisierung zu verdanken ist, die viele Aufrufe benötigt. Bei größeren Projekten bleibt der leJOS Code übersichtlich und kann Java-typisch aufgeteilt werden.

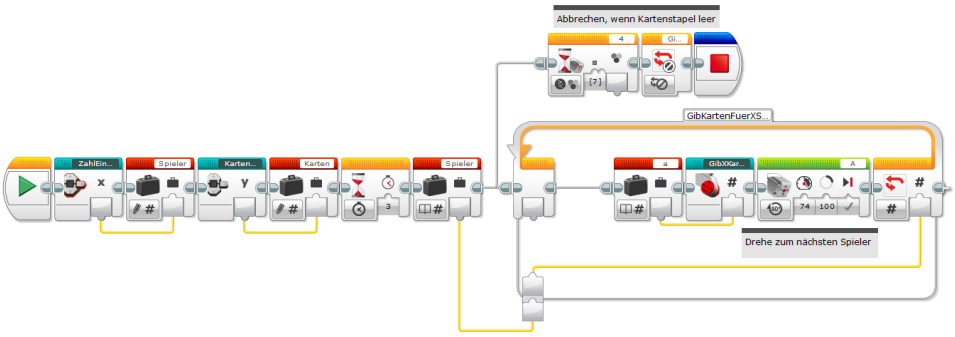

Abbildung 3.2: Das Hauptprogramm von CardBot in der Lego Software

3.3 Fragen an die Entwurfs- und Entwicklungs-Methodik

3.3.1 Usability

Die Usability hat bei der hauseigenen Lego Software ganz klar Vorrang. Man fühlt sich schnell mit der Umgebung vertraut und erste Erfolge lassen sich in wenigen Minuten erzielen. Es kann durchaus von Vorteil sein, wenn der Anwender Programmiererfahrung mitbringt, etwa bei der Verwendung von Schleifen oder das Auslagern von Aktionen in eigene Blöcke. Grundsätzlich ist es nicht unmöglich auch komplexe Programmabläufe damit zu implementieren, aber die Übersicht geht schnell verloren und die Fehlersuche wird zu einer großen Herausforderung. Änderungen in der Funktion benötigen schnell mehrere Stunden, wenn die Implementierung sehr komplex ist und Abläufe nicht vollständig durch selbst erstellte Blöcke abstrahiert wurden. Es gibt außerdem keinen Debug Modus oder eine Möglichkeit Laufzeitfehler zu beobachten, Refactoring zu betreiben sowie automatisierte Tests auszuführen. Dass das Programm nicht wie gewünscht funktioniert, wird erst bei der Ausführung auf dem Roboter deutlich. Und in diesem Fall kann nur mithilfe von manueller Analyse der Fehler gefunden werden.

Lego Programme in Java zu schreiben ist lange nicht so komfortabel wie das Zusammenstecken von fertigen Blöcken, bietet aber gegenüber dessen viele Vorteile. Die Usability zeigt sich hier nicht durch den visuellen Kontext, sondern wird im späteren Verlauf der Implementierung deutlich. Nämlich zu dem Zeitpunkt, zu dem es um die Fehlersuche geht oder zum Beispiel einzelne Funktionen umgeschrieben werden müssen. Was in der Lego Umgebung durch das Auslagern in Blöcke realisiert wird, ist in Java die Auslagerung von Methoden. Des Weiteren bietet die Java Implementierung per leJOS die Möglichkeit, in einem Team, zum Beispiel per Git, gemeinsam an einem Projekt zu arbeiten. Da die Programmierung nur per Code geschieht, ist ein direkter Einstieg ohne Visualisierung oder vorherige Spezifikation des Systemverhaltens nicht zu empfehlen. Die on-the-fly Erstellung von Programmcode ist fehleranfällig und oft werden Variablen aus der Umwelt oder Ausnahmefälle nicht behandelt.

Der szenariobasierte Ansatz bietet zwar einen schwierigen Einstieg, verspricht aber komplexes Verhalten beherrschbar zu machen, bevor es als Implementierung existiert. Die Erstellung von Szenarien in MSDs soll das Verhalten des Systems in bestimmten Fällen darstellen und Ausnahmen behandeln.

3.3.2 Versprechen des szenariobasierten Ansatzes

Der szenariobasierte Ansatz verspricht die Verbesserung des Entwicklungsprozesses durch folgende Punkte:

Komplexität/Nebenläufiges Verhalten

Auch komplexe Abläufe, speziell parallel ablaufende, können modelliert und überprüft werden. Dies hilft vor allem bei der Implementierung von Verhalten, welches durch Umwelteinflüsse ausgelöst wird. Zum Beispiel soll die eigentliche Funktion eines Roboters kontinuierlich ausgeführt werden, aber bei bestimmten Ereignissen soll er entsprechend reagieren.

Inkrementalität

Die Idee ist, dass das Projekt einfach erweitert werden kann, indem neue Szenarien neben den schon bestehenden hinzugefügt werden. Wenn zum Beispiel ein Roboter mit einem zusätzlichen Sensor ausgerüstet wird, kann in einem weiteren Szenario Verhalten spezifiziert werden, wie der Roboter auf bestimmte Umwelteinflüsse reagieren soll, die den Sensor ansprechen.

Behandlung von Ausnahmen

Ausnahmefälle können ebenso modelliert werden und bestärken so die Integrität des Entwurfs und der Implementierung. Ein Beispiel wäre, dass ein Kartengeber-Roboter eine Karte geben soll, es sich jedoch keine Karten mehr im Stapel befinden. Solch eine Ausnahme wird erfasst und bei Eintritt bestimmtes Verhalten ausgeführt.

Intuitivität

Durch die Verwendung von Modellen werden Funktionsabläufe abstrahiert dargestellt, dadurch wird Intuitivität gewährleistet.

Widersprüche früh aufdecken

Widersprüche im der Spezifikation, die die Szenarien betreffen, werden früh aufgedeckt und erreichen nicht die Testphase. Vor allem dieser Punkt ist vielversprechend, da die Fehlersuche nach der Implementierung sehr zeitaufwendig und bisweilen kompliziert ist.

Kapitel 4

Codegenerierung

Die Codegenerierung habe ich als Plugin für Eclipse realisiert. ScenarioTools erstellt bei der Synthese eine `*.msdruntime` Datei, welche den von ScenarioTools aus den MSDs synthetisierten Zustandsautomaten enthält.

4.1 Analyse der Eingabedaten

Um Code aus dem Zustandsautomaten zu generieren, muss dieser eingelesen und Zustand für Zustand ausgewertet werden. Aus der `*.msdruntime` Datei lassen sich die Zustände, Transitionen und Nachrichten des Automaten auslesen. Jeder Zustand enthält Informationen über alle bei ihm ein- und ausgehenden Transitionen. Die Transitionen können kontrollierbar sein, mit anderen Worten *executed*, also ein auszuführender Befehl. Nicht kontrollierbare Transitionen sind *monitored*, also Warten zum Beispiel auf Sensorinput. Der Zustandsautomat beinhaltet auch die Namen der spezifizierten Nachrichten/Events und ihre übergebenen Parameter.

4.2 Verarbeitung

Um den Java Code zu erzeugen, verwende ich mehrere StringBuffer, in die die verwendeten Methoden und Anweisungen geschrieben werden. Ein StringBuffer enthält das äußere Gerüst des Codes, welches den Package Namen und die Imports enthält. Ein weiterer StringBuffer enthält die Main Methode, die erst am Ende mit den, aus den Zuständen erstellten, StringBuffern gefüllt wird. Mein Plugin läuft alle Zustände und ihre Transitionen durch. Dabei wird geprüft ob eine Transition kontrollierbar ist oder nicht. Durch die Information über Kontrollierbarkeit werden if-Anweisungen erstellt und in einen StringBuffer geschrieben. Ist eine Nachricht nicht kontrollierbar, wird sie in den StringBuffer der if-Anweisung geschrieben. Folgen zwei nicht kontrollierbare Transitionen aufeinander, wird die eine in den Rumpf der ersten if-Anweisung gesetzt. Die kontrollierbaren Elemente werden in

den Rumpf der innersten if-Anweisung geschrieben und werden so nach der Kompilierung ausgeführt, wenn alle Bedingungen wahr werden. Des Weiteren werden alle Nachrichten nach Stichwörtern wie *Touch, LargeMotor, MediumMotor oder Color* durchsucht um automatisch die für den leJOS Code benötigten Imports zu erstellen und eventuelle Sensoren und Motoren zu initialisieren. Die Nachricht stellt dabei einen abstrakten Aufruf dar, die eigentliche Funktion ist durch den Codegenerator vordefiniert und wird als Methode eingebunden. Außerdem können den Nachrichten Parameter zugewiesen sein. Diese werden ebenfalls verarbeitet und in die dazugehörigen Aufrufe eingefügt. Alle erforderlichen Methoden werden automatisch erstellt, insofern sie unterstützt werden.

4.3 Ausgabe

Nachdem ScenarioTools eine erfolgreiche Synthese durchführen konnte, wird mit einem Aufruf des Punktes `ScenarioTools > Generate leJOS Code` im Kontextmenü der `*.msdruntime` Datei die Codegenerierung gestartet. Die Ausgabe erfolgt in einer neuen `*.java` Datei, das Namensschema des Projektes wird dabei übernommen. Da die Imports bereits automatisch generiert wurden, muss dem Projekt nur noch die leJOS EV3 Runtime Library zugewiesen werden. Es empfiehlt sich außerdem, die Java Datei in ein eigenes Package zu verschieben. Dazu wird im Stammverzeichnis des Projekts ein Ordner `src` angelegt und darin ein neues Package mit dem Namen des Projekts. Dies ist eine Maßnahme um sicherzustellen, dass das leJOS Plugin seinen Dienst fehlerfrei verrichten kann.

Die spezifizierten Szenarien benötigen abstrakte Methoden um die Modellierung zu erleichtern. Falls Methoden verwendet wurden, die vom Code Exporter nicht unterstützt werden, bereitet er diese vor. Sie müssen in dem Fall vom Entwickler mit Funktionen bestückt werden. Die eigentliche Ausführung des Szenarios geschieht in der Main Methode. Die Aufrufe darin stellen den synthetisierten Zustandsautomaten dar. Ausgehend davon, dass das leJOS Plugin bereits eingerichtet und ein EV3 Brick spezifiziert wurde, kann die Implementierung nun von leJOS zu einer `*.jar` Datei verarbeitet werden. Diese wird danach automatisch an den EV3 Brick gesendet und ausgeführt.

Kapitel 5

Durchführung und Bewertung der Fallstudie

5.1 Vorgehen bei der Entwicklung

Ich gehe die Entwicklung einer Implementierung von Roboterverhalten am Beispiel vollständig durch. Dafür teste ich drei Entwicklungsansätze:

- Implementierung per Lego Entwicklungssoftware
- Implementierung in leJOS
- Szenariobasierter Ansatz

5.1.1 Spezifikation

Bei der Spezifikation macht es keinen Unterschied, wie man am Ende zu der Implementierung gelangt. Hier muss formal beschrieben werden was der Roboter machen soll, wo er eingesetzt wird und welche Module (Sensoren, Motoren) er benutzt.

Im Falle von SimpleBrick gibt es einen EV3 Brick, einen Berührungssensor sowie einen großen Motor. Diese Komponenten müssen nun abgegrenzt werden. Der EV3 Brick stellt das kontrollierende System dar. Der Berührungssensor sowie der Motor sind Teile der Umwelt (Environment). Die Funktion von SimpleBrick ist folgende: Wenn der Touch Sensor gedrückt wird, soll der Motor eine Drehung um 90 Grad machen.

Lego Entwicklungssoftware und leJOS

Bei der Realisierung per Lego Software gibt es keinen Zwang zur formalen Spezifikation. Es ist einem selbst überlassen, ob Diagramme erstellt werden oder nur informell die Funktion beschrieben wird. In vielen Fällen folgt aus einer Idee direkt die Implementierung, ohne vorherige Spezifikation.

Es wird also komplett auf einen Entwurf verzichtet, direkt prototypisch programmiert und das Ergebnis per trial-and-error ständig bis zur finalen Version überarbeitet.

Szenariobasierter Ansatz

Die Komponenten des Systems werden in einem Klassendiagramm (Abb. 5.2) aufgeführt, das Zusammenspiel der Komponenten kann in einem Komponentendiagramm (Abb. 5.3) visualisiert werden. Außerdem sollte es verschiedene Container für die *Assumptions* und *Requirements* geben. Die einzelnen Container werden durch einen weiteren Container *Integrated* zusammengeführt (Abb. 5.1).

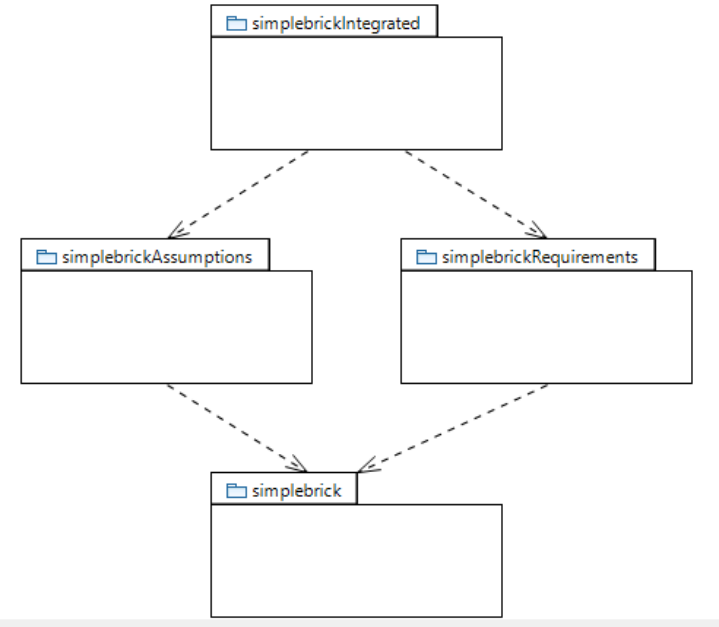

Abbildung 5.1: Überblick der Teile des Projekts

5.1.2 Szenario/Entwicklung

Lego Entwicklungssoftware und leJOS

In dieser Methode ist keine Erstellung von Szenarien vorgesehen. Das Programm wird direkt visuell entwickelt. Die Funktion einzelner Elemente wird durch Blöcke abstrakt dargestellt.

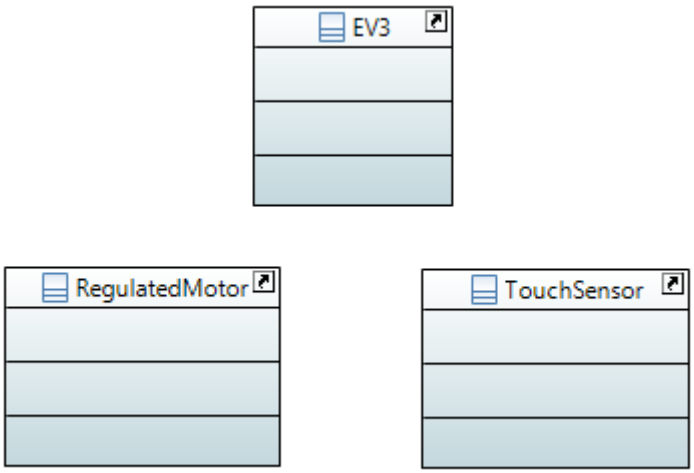

Abbildung 5.2: Klassendiagramm SimpleBrick

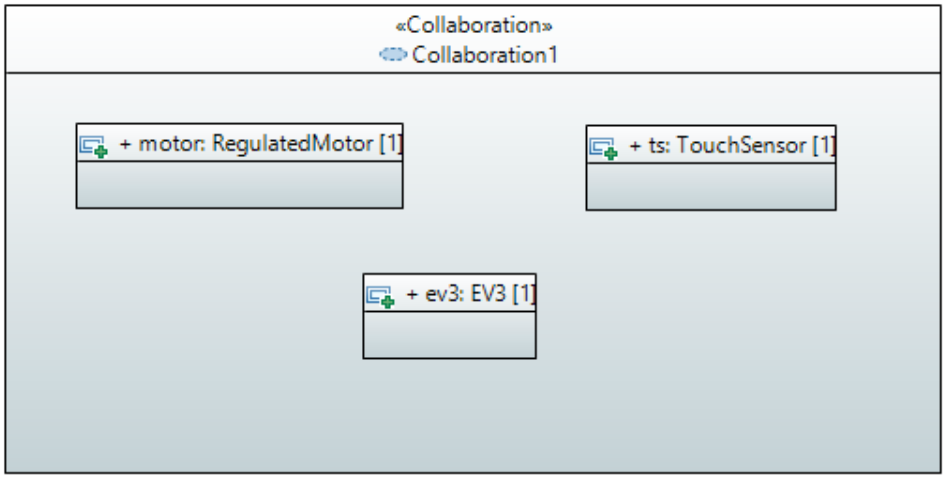

Abbildung 5.3: Kollaboration Simplebrick

LeJOS

Der Code kann nach bestimmten Pattern geschrieben werden, jedoch hilft das nicht bei der Validierung des Verhaltens.

Szenariobasierter Ansatz

Für jedes Szenario wird ein MSD benötigt. In den Requirements wird die Grundfunktion modelliert, kompliziertere Aufgaben können in kleinere Szenarien aufgeteilt werden (Divide and Conquer). In den Assumptions

werden Aktionen modelliert, die nicht erlaubt sind.

Die Abb. 5.4 zeigt das MSD, welches das Verhalten von SimpleBrick darstellt. Wenn der Berührungssensor meldet, gedrückt worden zu sein, löst der EV3 Brick die Motorbewegung aus. Nachdem sich der Motor wieder im Stillstand befindet, sendet er eine Nachricht, die dies quittiert. Jetzt muss man ein paar Randfälle abdecken, wie zum Beispiel, dass es nicht erlaubt ist, den Berührungssensor zu drücken, solange der Motor noch arbeitet (siehe Abb. 5.5).

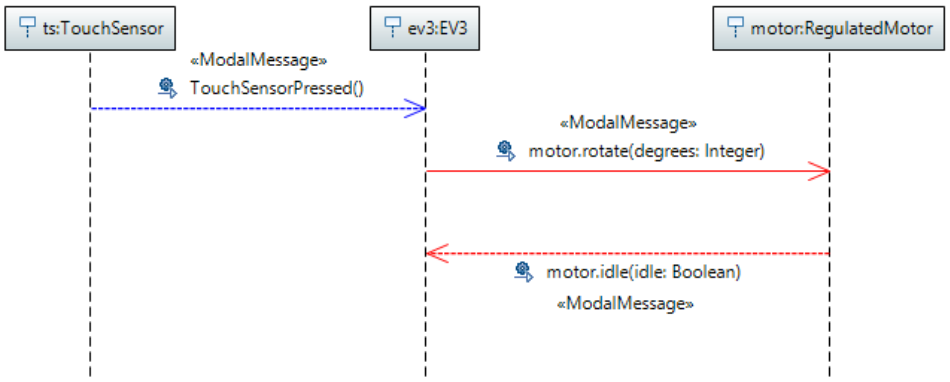

Abbildung 5.4: MSD des simplen Beispiels SimpleBrick

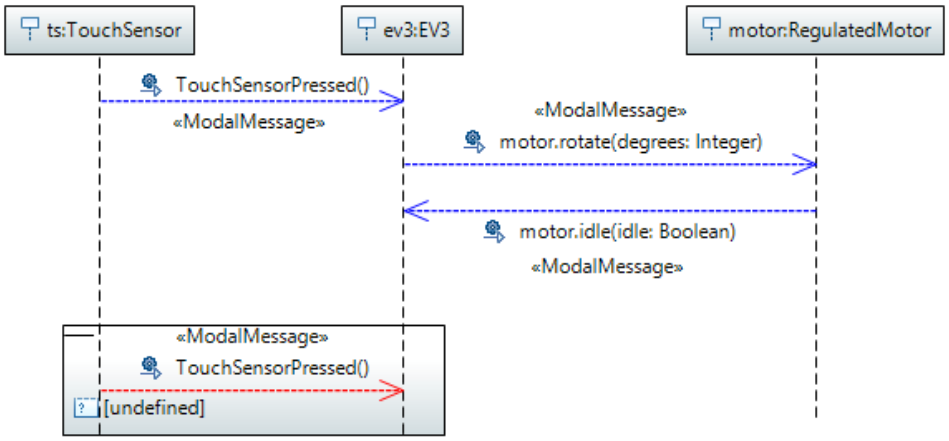

Abbildung 5.5: MSD einer sog. Assumption von SimpleBrick

5.1.3 Validierung

Lego Entwicklungssoftware und leJOS

Wenn die Spezifikation nicht formal erfolgt, kann sie nicht automatisch validiert werden. Der Entwickler muss seine Idee in diesem Fall manuell auf Fehler prüfen und die Fehlersuche während der Implementierung durch trial-and-error realisieren.

Szenariobasierter Ansatz

Die in Eclipse erstellten Modelle lassen sich durch eingebaute Funktionen automatisch validieren. Wenn die Diagramme validiert wurden, können sie durch ScenarioTools zu einer Strategie synthetisiert werden. Diese Strategie wird von einem Zustandsautomaten dargestellt, der von ScenarioTools simuliert sowie als grafische Abbildung ausgegeben werden kann. Wie in der Abbildung 5.6 zu sehen, geht der Zustandsautomat in einen Deadlock über, falls der Berührungssensor gedrückt wird, bevor der Motor zum Stillstand kommt.

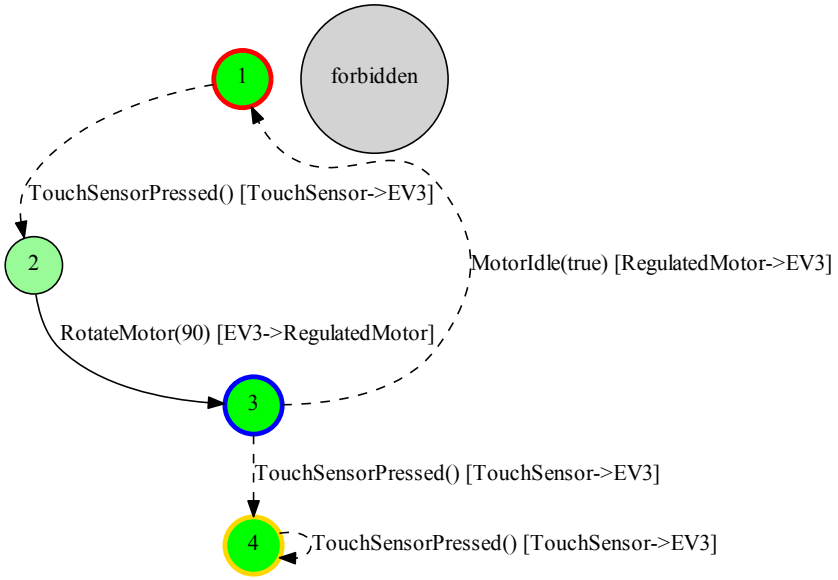

Abbildung 5.6: Zustandsautomat von SimpleBrick

5.1.4 Codegenerierung

Lego Entwicklungssoftware

Die Lego Software generiert aus der erstellten Blockstruktur automatisch ausführbaren Code und sendet diesen an den EV3 Brick.

LeJOS

Das leJOS Plugin kompiliert den geschriebenen Code und sendet ihn an den EV3 Brick.

Szenariobasierter Ansatz

Aus dem Zustandsautomaten wird eine neue Java Datei generiert, die den auf leJOS basierenden Programmcode enthält. Da die Nachrichten in den MSDs nur abstrakt formulierte Aufrufe darstellen, wird die Funktion der Aufrufe ausgelagert. In Zukunft könnte dies über eine Bibliothek geschehen, die bestimmte Roboterfunktionen beinhaltet und mit einfachen Aufrufen ausführen kann. Die Codegenerierung enthält bereits Methoden für alle Aktionen des Beispiels SimpleBrick. Der generierte Code für SimpleBrick sieht folgendermaßen aus:

Listing 5.1: Generierter leJOS Code für SimpleBrick

```
package de.vhartmann.bscthesis.simplebrick;

import lejos.hardware.motor.EV3MediumRegulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.robotics.RegulatedMotor;
import lejos.hardware.port.SensorPort;
import lejos.hardware.sensor.EV3TouchSensor;
import lejos.hardware.sensor.SensorMode;
import lejos.hardware.Button;

public class model_simplebrickIntegratedController {

   // Something wants to know if the motor is idle
   public boolean MotorIdle(){
      if (!motor.isMoving())
         return true;
      else
         return false;
   }

   // Something in this project wants a motor to rotate some degrees
   static RegulatedMotor motor = new
       EV3MediumRegulatedMotor(MotorPort.A);
   public static void RotateMotor(int degrees){
```

```
      motor.rotate(degrees);
   }

   // Seems like this project uses the TouchSensor, better
       initialize it
   static boolean TouchSensorPressed() {
      // get touch sensor
      EV3TouchSensor touchSensor = new EV3TouchSensor(SensorPort.S1);
      SensorMode touch = touchSensor.getTouchMode();
      // prepare sample for the measurement
      float[] sample = new float[touch.sampleSize()];
      // grab a sample
      touch.fetchSample(sample, 0);
      // close the touch sensor
      touchSensor.close();
      if (sample[0] == 1.0) return true;
      else return false;
   }

   public static void main(String[] args) {
      // end program when ESCAPE is pressed on the EV3 brick
      if (Button.ESCAPE.isDown())
         System.exit(0);
      while (Button.ESCAPE.isUp()) {
         if (TouchSensorPressed()){//outer if
            if (MotorIdle()){
               //Controllable
               RotateMotor(90);
               //inner if
               //Code goes here
            }
         }
      }
   }
}
```

5.1.5 Test auf dem Roboter

Lego Entwicklungssoftware

Das in der Lego Software erstellte Programm wird auf dem Brick ausgeführt und es wird einem visuell angezeigt, in welchem Teil des Programms der Brick sich gerade befindet. Das hilft dabei, Fehlerquellen einzugrenzen.

LeJOS

Wenn der Code fehlerfrei kompiliert, ist er bereit um auf dem Brick ausgeführt zu werden. Der Zustand der Ausführung lässt sich verfolgen,

in dem man an kritischen Stellen im Programmcode eine Konsolenausgabe implementiert.

Szenariobasierter Ansatz

Der generierte Code wird vom leJOS Plugin auf den Roboter übertragen. Auch hier kann der aktuelle Stand nur abgelesen werden, wenn Konsolenausgabe implementiert wird.

5.2 Bewertung

Bei der Bewertung beziehe ich mich auf die von mir in 3.3.2 angesprochenen Versprechen des szenariobasierten Ansatzes. Die Frage ist, inwiefern die Verwendung dieses Ansatzes die Entwicklung beeinflusst hat.

Der szenariobasierte Ansatz hält sein Versprechen bezüglich nebenläufiger Aktivitäten. Sie lassen sich gut modellieren und auch komplexe Abläufe bleiben während des Entwurfs übersichtlich. SimpleBrick musste während der Programmausführung ständig den Status des Berührungssensors überprüfen und parallel andere Aktionen ausführen. Dieses Verhalten konnte ich in MSDs modellieren und es ließ sich auch funktionierender leJOS Code darauf generieren. Auch die Ansätze der Lego Software und leJOS machen es möglich parallele Abläufe zu programmieren. Die Implementierung muss jedoch von vornherein manuell geschehen. Die Funktionen des Roboters lassen sich durch das Entwickeln neuer Szenarien erweitern. SimpleBrick könnte mit weiteren Sensoren bestückt werden, die zusätzliche Aktionen auslösen. Ein Punkt, der den szenariobasierten Entwurf besonders herausstellt, ist die Behandlung von Ausnahmen. Durch den Einsatz von Assumption MSDs können viele Ausnahmen modelliert werden. Diese werden in der Synthese berücksichtigt und somit in die Programmierung des Roboters mit aufgenommen. Eine Ausnahme, die in SimpleBrick modelliert wurde, war, dass der Berührungssensor keine Aktion auslösen darf, während der Motor noch in Bewegung ist. Diese Ausnahme habe ich in einem MSD modelliert und sie wurde bei der Synthese von ScenarioTools berücksichtigt. Ein Kritikpunkt ist die Intuitivität, welcher leider eine steile Lernkurve gegenüber steht. Der Entwurf der Szenarien geschieht durch den Einsatz von Freitext noch weitgehend intuitiv. Die Übertragung der Szenarien in Modelle erfordert eine Einarbeitungsphase, da ScenarioTools eine gewisse Modellstruktur voraussetzt. Der Einstieg in die Programmierung mit der Lego Software ist im Vergleich ein Kinderspiel. Auch die Arbeit mit leJOS ist für erfahrene Programmierer sehr leicht. Hier ergibt sich eine offene Herausforderung, nämlich die Eingabe der Modelle intuitiver zu gestalten. Anders als bei der Lego Programmierumgebung, wo Widersprüche erst während des Tests aufgedeckt werden (trial-and-error), hat der szenariobasierte Ansatz schon früh im Entwicklungsprozess seine Stärken gezeigt. Nach der

Modellierung konnte bereits festgestellt werden, ob und wo Fehler existieren. Die fehlerhafte Implementierung ist also nicht an den Lego Roboter gelangt. Abschließend bewerte ich den szenariobasierten Ansatz als gutes Werkzeug, komplexe Vorgänge zu implementieren. Bei der Implementierung kleiner Projekte würde ich die Verwendung der Lego Software empfehlen. Sobald erweiterte Funktionen verwendet werden, kann auf leJOS zurückgegriffen werden. Für sehr große Projekte oder den Einsatz in sicherheitskritischen Bereichen ist der szenariobasierte Ansatz zu empfehlen.

Kapitel 6

Verwandte Arbeiten

In einem Projekt von Habad und Leibovich [6] wird aus Komponenten-Diagrammen leJOS Code generiert. Habad und Leibovich bauen dabei auf die vorherige Lego Mindstorms Version *NXT* auf. Bei diesem Projekt wird jedoch im Gegensatz zur Verwendung von Szenarien keine Ausnahmebehandlung durchgeführt. Außerdem kann die Behandlung von Umwelteinflüssen nachträglich nicht so einfach hinzugefügt werden, als es bei der Verwendung des szenariobasierten Ansatzes durch das Hinzufügen eines Szenarios möglich ist.

Für das Projekt wurde eigens eine Modellsprache, die *Component Model Language*, entwickelt. Der Nutzer baut die Roboter Funktionen in Diagrammen modular auf. Funktionen können abstrahiert werden, indem in Diagrammen andere Komponenten-Diagramme aufgerufen werden. Roboterverhalten kann dabei auch als Zustandsautomat formuliert werden. Ein Projekt besteht aus einer Hauptkomponente und ihren Subkomponenten. Abstrakt benannte Zugriffe auf Sensoren und Motoren werden durch eine Bibliothek in leJOS NXT Code übersetzt. Auf ihrer Homepage zeigen Habad und Leibovich die Funktion an einem Roboter, der vorwärts fährt, bis sein Ultraschallsensor ein Objekt erkennt. Dann fährt der Roboter rückwärts, bis ein Berührungssensor auf der Rückseite eine Kollision meldet. Daraufhin fährt er wieder vorwärts und der Vorgang wiederholt sich so lange bis ein Knopf auf dem Roboter gedrückt wird und das Programm abbricht. Die Funktion des Roboters wurde in der *Component Model Language* modelliert. Das Haupt-Komponenten-Diagramm von Simple Robot(siehe Abb. 6.1) enthält den Berührungssensor, den genannten Knopf und die Aktion SystemExit als Aufrufe der Komponenten Bibliothek. Die Kollisionserkennung sowie das Fahren wurden in eigene Komponenten-Diagramme ausgelagert.

In das Komponenten-Diagramm, welches das Fahren des Roboters beschreibt, wurde eine weitere Subkomponente eingefügt die die Grundfunktion des Fahrens in einem Zustandsautomaten beschreibt (Abb. 6.2).

Habad und Leibovich haben eine Komponenten Bibliothek aufgebaut,

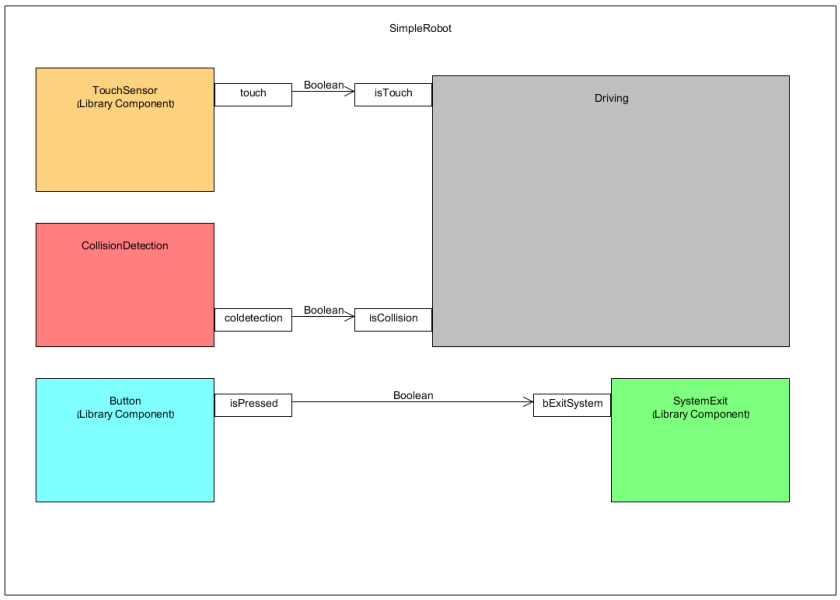

Abbildung 6.1: Haupt-Komponenten-Diagramm von SimpleRobot

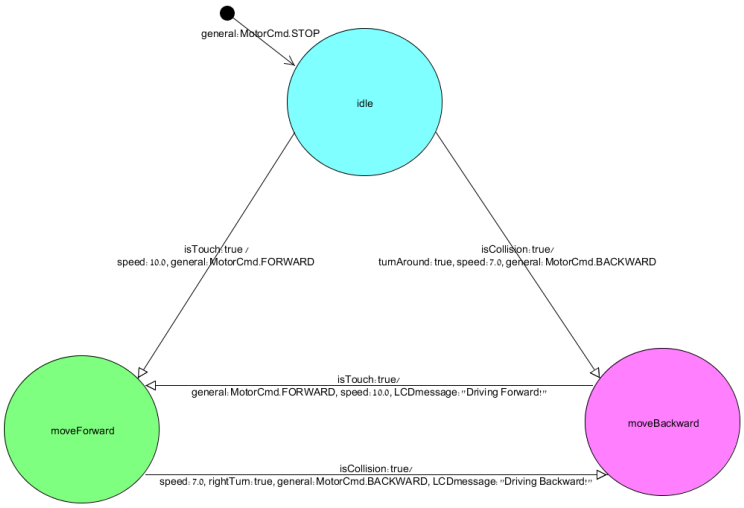

Abbildung 6.2: Zustandsautomat für die Komponente „Driving“

wie ich sie mir auch für mein Projekt vorstellen könnte. Sie ermöglicht es, komplexe Abläufe durch einfache Aufrufe zu abstrahieren. Das steigert die Usability und hebt den Ansatz auf eine höhere Abstraktionsebene.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

In dieser Arbeit stellte ich mich dem Problem, Verhalten, das in einem szenariobasierten Entwurf spezifiziert wurde, auf ein Lego Robotersystem zu übertragen. Speziell durch Umwelteinflüsse ausgelöste Ausnahmen sollten in den Entwurf aufgenommen und in der Implementierung berücksichtigt werden.

Zur Lösung dieses Problems untersuchte ich, ob der szenariobasierte Ansatz diesen Anforderungen gerecht wird. Ich entwickelte ein Eclipse Plugin, welches ScenarioTools um die Funktion erweitert, aus synthetisierten Zustandsautomaten für Lego Roboter den zugehörigen leJOS Code zu generieren. Mit Hilfe dieses Plugins ist es nun möglich, modellierte Szenarien für Lego Roboter automatisch zu implementieren. Das spart Zeit und gewährleistet einen fehlerfreien Programmablauf, da ScenarioTools die Modelle prüft und immer einen korrekten Zustandsautomaten ausgibt, insofern es eine erfüllende Strategie findet. Da die Nachrichten in den MSDs abstrakte Aufrufe darstellen, müssen diese Aufrufe schon vorher implementiert worden sein. Die Codegenerierung ruft sie dann gemäß des Zustandsautomaten auf.

Die Bewertung des Ansatzes fand anhand zweier Beispiele statt. Das Beispiel `SimpleBrick` zeigt, dass der szenariobasierte Ansatz auch auf einfaches Verhalten angewendet werden kann. `Cardbot` beschreibt komplexeres Verhalten, dadurch das Eingaben durch den Nutzer erfolgen und Ausnahmen, wie z.B. dass der Kartenstapel leer ist, behandelt werden müssen. Auch hier zeigte sich, dass ein szenariobasierter Entwurf es direkt ermöglicht, solche Ausnahmen zu berücksichtigen und zu behandeln.

Als abschließendes Fazit kann gesagt werden, dass der szenariobasierte Ansatz vielversprechende Möglichkeiten öffnet und die Implementierung von Verhalten unter Berücksichtigung von Umwelteinflüssen sehr gut unterstützt.

Die Übertragung dieses Konzepts auf die Entwicklung des Verhaltens eines Lego Roboters war erfolgreich und belegt die Vielseitigkeit des szenariobasierten Ansatzes.

7.2 Ausblick

Meine Erkenntnisse zeigen, dass der szenariobasierte Ansatz sehr gut geeignet ist, um Systemverhalten für Lego Roboter zu spezifizieren. Am Beispiel des Lego Roboters habe ich gezeigt, dass eine Codegenerierung für Robotersysteme prinzipiell möglich ist. In Zukunft könnte es Bibliotheken geben, die bestimmtes Roboterverhalten in abstrakten Methoden beinhalten. Die jetzige Codegenerierung beruht darauf, dass die Events in den MSDs vom Code Exporter unterstützt werden. Mit einer Bibliothek, die alle möglichen Roboterfunktionen beinhaltet, ließe sich eine Codegenerierung sauberer durchlaufen und es müsste weniger Interaktion durch den Entwickler geben.

Die Codegenerierung könnte in Zukunft auch für industrielle Robotersysteme realisiert werden, um es Entwicklern besser zu ermöglichen, Ausnahmen und Umwelteinflüsse, die das Robotersystem betreffen, zu behandeln. Dabei ist der Lösungsansatz nicht nur auf Robotersysteme begrenzt, denn szenariobasierte Entwürfe können überall zur Entwicklung von Systemen eingesetzt werden. Das Ergebnis liegt in Form eines Zustandsautomaten vor, dieser ermöglicht es durch eine Codegenerierung jegliches Systemverhalten direkt zu implementieren. Eventuell kann in Zukunft auch eine Möglichkeit entwickelt werden, die Arbeit von Habad und Leibovich [6], welche ich in 6 als verwandte Arbeit beschrieben habe, mit dem szenariobasierten Ansatz zu verbinden und zum Beispiel einzelne Komponenten eines Modells von ScenarioTools synthetisieren zu lassen und so die Konzepte zu vereinen.

Anhang A

Gegenstand dieser Arbeit

Gegenstand dieser Arbeit sind diese Ausarbeitung, die Dateien des Plugins, welches ich zur Codegenerierung geschrieben habe, in der Version die zum Zeitpunkt der Abgabe der Arbeit aktuell waren und die Dateien der Beispiele SimpleBrick und CardBot. Das Plugin besteht aus folgenden Paketen:

- org.scenariotools.codegen.lejos
- org.scenariotools.codegen.lejos.ui

Außerdem befinden sich im Repository das Beispiel SimpleBrick als native leJOS Implementierung (de.vhartmann.bscthesis.nativeSimplebrick) sowie als MSD Spezifikation (de.vhartmann.bscthesis.simplebrick/01-turn90degrees) und als Lego EV3 Projekt-Datei (SimpleBrick.ev3). Die EV3 Projekt-Datei des Kartengeberoboters befindet sich ebenfalls in den Downloads des Repository (D3ALERBOT.ev3).

Git Repository: `https://bitbucket.org/jgreenyer/students-vhartmann/` URL zum Klonen des Git Repository: `https://vhartmann@bitbucket.org/jgreenyer/students-vhartmann.git`

Zum Zeitpunkt der Abgabe ist die ID des aktuellsten Commits: **5a576d4**.

In der Entwicklungsumgebung wird vorausgesetzt, dass die Plugins leJOS und ScenarioTools installiert sind.

Alle diese Dateien liegen auch auf der beigelegten CD vor.

Anhang B

CardBot

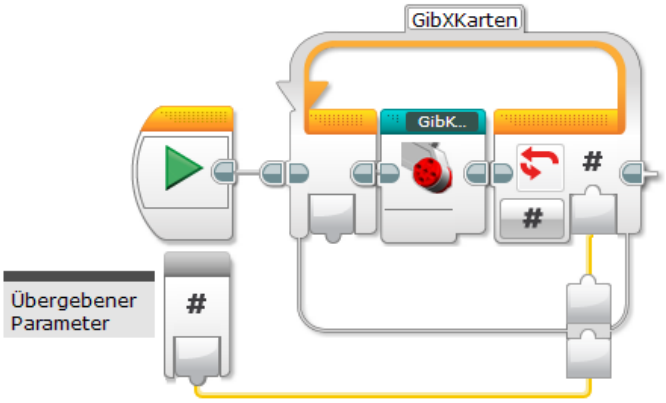

Abbildung B.1: Inhalt des GibXKarten-Blocks (dieser beinhaltet wiederum einen selbst erstellten Block)

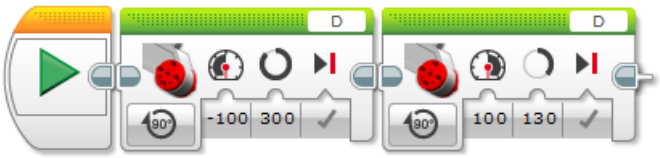

Abbildung B.2: Inhalt des GibKarte-Blocks

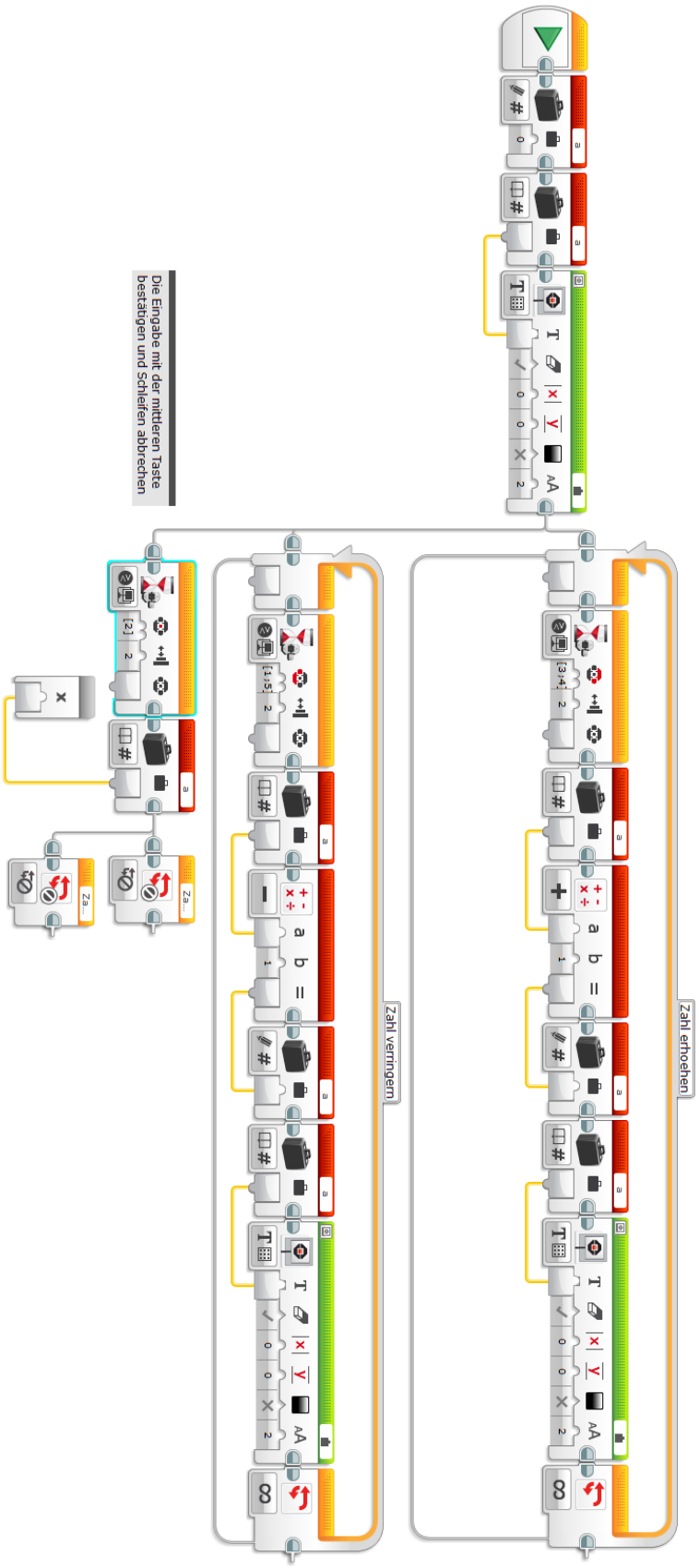

Abbildung B.3: Inhalt des ZahlEingeben-Blocks

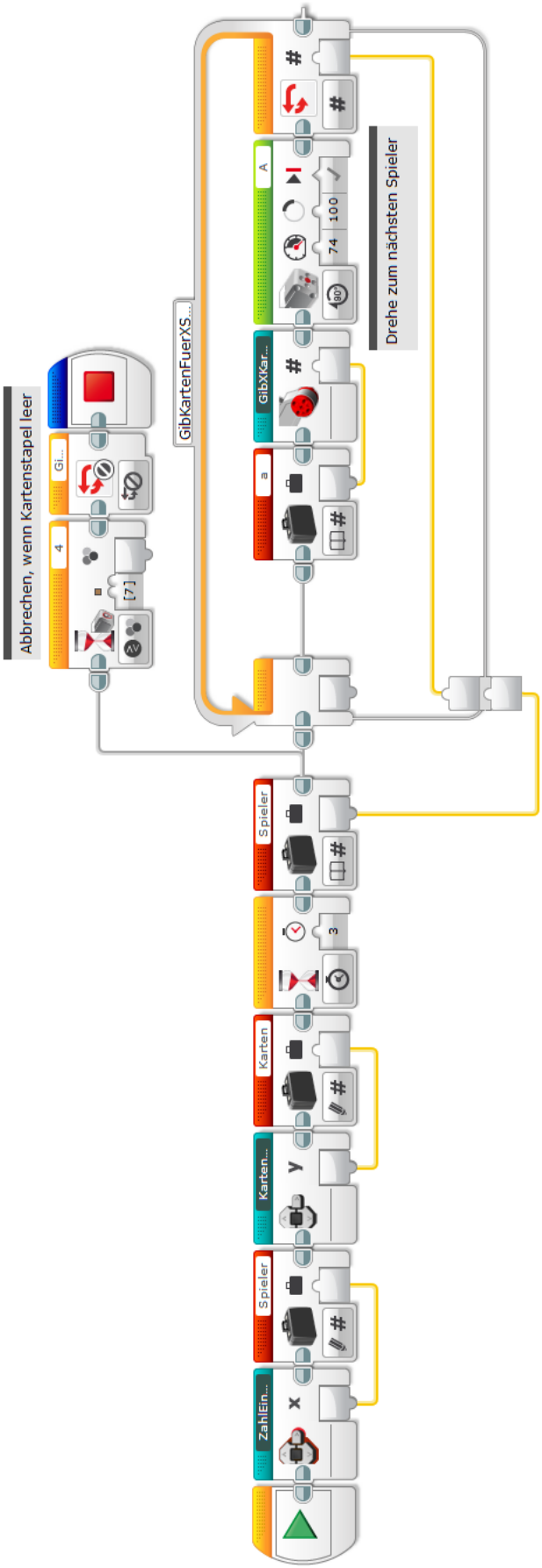

Abbildung B.4: CardBot Hauptprogramm

Anhang C

Code

```
package nativeSimplebrick;

import lejos.hardware.Button;
import lejos.hardware.motor.EV3MediumRegulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.hardware.port.SensorPort;
import lejos.hardware.sensor.EV3TouchSensor;
import lejos.hardware.sensor.SensorMode;
import lejos.robotics.RegulatedMotor;

public class NativeSimplebrick {

   static RegulatedMotor motor = new
       EV3MediumRegulatedMotor(MotorPort.A);

   static boolean TouchSensorPressed() {
      // get touch sensor
      EV3TouchSensor touchSensor = new EV3TouchSensor(SensorPort.S1);
      SensorMode touch = touchSensor.getTouchMode();

      // prepare sample for the measurement
      float[] sample = new float[touch.sampleSize()];

      // grab a sample
      touch.fetchSample(sample, 0);

      // close the touch sensor
      touchSensor.close();

      // if touch sensor is pressed, return true; else return false
      if (sample[0] == 1.0)
         return true;
      else
         return false;
```

```
   }

   public static void main(String[] args) {
      // end program when escape-button is pressed on the EV3 brick
      if (Button.ESCAPE.isDown())
         System.exit(0);

      // rotate the motor 90 degrees if touch sensor is pressed
      while (Button.ESCAPE.isUp()) {
         if (TouchSensorPressed())
            motor.rotate(90);
      }
   }
}
```

Literaturverzeichnis

[1] leJOS Wikipedia-Eintrag. `http://de.wikipedia.org/wiki/LeJOS`, April 2015.

[2] B. Bagnall, J. Stuber, and P. Andrews. leJOS Projekt Homepage. `http://www.lejos.org/`, April 2015.

[3] C. Brenner, J. Greenyer, J. Holtmann, G. Liebel, G. Stieglbauer, and M. Tichy. Scenariotools real-time play-out for test sequence validation in an automotive case study. In *Proceedings of the 13th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2014)*, volume 67. EASST, 2014.

[4] J. Greenyer. *Scenario-based Design of Mechatronic Systems*. PhD thesis, University of Paderborn, Paderborn, October 2011.

[5] J. Greenyer, V. P. L. Manna, and C. Brenner. ScenarioTools. `http://scenariotools.org/`, April 2015.

[6] E. Habad and I. Leibovich. Component Model Language for Lego Mindstorms NXT. `http://www.cs.tau.ac.il/~eranhaba/SMLAB/index.htm`, 2013.

[7] D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and System Modeling*, 7(2):237–252, 2008.