technische universität dortmund

fakultät für informatik

Endbericht

Projektgruppe AgES (576)

Lehrstuhl 5 für Programmiersysteme

Aspektgetriebene Entwicklung sicherer Steuerungssysteme

Chris Apfelbeck, Simon Dierl, Ba Phuoc Dinh, Christian Drescher, Florian Eckey, Tobias Görgen, Florian Lechner, Malte Ludewig, Janina Kim Marks, Kim Quermann, Yan Rudall

6. Mai 2014

Betreuer:

Dipl-Inf. Oliver Bauer	im 1. Semester
Dipl-Inf. Michael Lybecait	im 2. Semester
Dipl-Inf. Stefan Naujokat	in beiden Semestern

Inhaltsverzeichnis

KAPITEL 1

Einleitung

Diese Arbeit ist der abschließende Bericht der Projektgruppe „AgES – Aspektgetriebene Entwicklung sicherer Steuerungssysteme“, die sich vom Sommersemester 2013 bis zum Wintersemester 2013/2014 mit der Absicherung von Steuerungssystem durch Technologien und Werkzeuge aus der aspektorientierten Softwareentwicklung beschäftigte.

Dieses Kapitel führt in das Thema der Projektgruppe und die zugrunde liegende Problemstellung ein. Daraufhin wird erläutert, welche minimalen und optionalen Ziele die Projektgruppe sich setzte, um die Problemstellung zu lösen. Schlussendlich wird der Aufbau dieses Berichtes erläutert, um einen Überblick über die Struktur und den Zusammenhang der einzelnen Kapitel zu schaffen.

1.1 Motivation und Hintergrund

Bei der Entwicklung von Software zur Steuerung physikalischer Systeme wie Fabrikanlagen, Flugzeugen und Ähnlichem ist oftmals zu beachten, dass von der korrekten Funktionalität der Software nicht nur das Erreichen eines Geschäftszieles, sondern auch die Sicherheit von Personen oder zumindest teurer Hardware abhängt. Bei der Entwicklung dieser Software ist es daher von höchster Bedeutung, dass das System entweder fehlerfrei ist oder im Fall eines Fehlers einige Sicherheitsmerkmale garantiert werden, also ein „kontrolliertes“ Versagen stattfindet. Das System wird dadurch fehlertolerant.

Ein Beispiel für solche Systeme sind Quadrokopter, die inzwischen auch im Hobby-Umfeld eine zunehmende Verbreitung finden. Zwar ist es eher unwahrscheinlich, dass eine Person durch einen der leichtgewichtigen Quadrokopter ernsthaft verletzt werden kann, aber Pilotenfehler können dennoch zu Beschädigungen am Quadrokopter selbst führen. Im Fall programmatisch gesteuerter Quadrokopter können entsprechend Fehler im Programm zu diesen Schäden führen; somit sind die oben genannten Grundsätze zu beachten.

Da ein steuerndes Programm sehr komplex werden kann, ist es beinahe unmöglich, vertretbare Entwicklungskosten und vollständig korrekte Software miteinander zu vereinbaren. Daher ist es sinnvoll, sich auf die Entwicklung von fehlertoleranten Systemen zu beschränken. Zwar ist es möglich, Systeme fehlertolerant zu gestalten, indem bei der Entwicklung jeder Softwarekomponente diese Toleranz implementiert wird, aber dies ist aufwändig und fehleranfällig, da Mängel in einer Komponente das ganze System beeinflussen können. Oft ist es jedoch möglich, die Sicherheitsanliegen orthogonal zur Logik des Programmes zu betrachten und diese in einen separaten Aspekt auszugliedern, der automatisiert mit dem restlichen Programm verwoben wird. Die Menge an abzusichernden Komponenten kann dadurch drastisch gesenkt werden, ohne die Sicherheitsgarantien abzuschwächen. Dieses Verfahren wird als Aspektorientierung bezeichnet.

Die Aspektorientierung ist für viele Programmiersprachen bereits intensiv erforscht worden und wird in der Praxis mit diesen genutzt. Eine Ausnahme ist jedoch die modellgetriebene Softwareentwicklung, die versucht, komplexe Funktionalität nicht durch Quellcode, sondern durch grafisch darstellbare Modelle zu beschreiben. Diese Techniken sind jedoch außerordentlich nützlich, um beispielsweise die Komplexität einer Steuerungssoftware bewältigen zu können.

Die Projektgruppe AgES wurde mit dem Ziel gebildet, eine Brücke zwischen der Aspektorientierung und der modellgetriebenen Softwareentwicklung am Beispiel von Steuerungssoftware, insbesondere der Steuerungssoftware für einen Quadrokopter, zu schlagen.

1.2 Anforderungen

Im Zuge der Bildung der Projektgruppe wurde ein Satz an Anforderungen definiert, der erfüllt werden sollte, um das Ziel der Projektgruppe zu realisieren. Zusätzlich zu diesen Minimalzielen wurden bald weitere Ziele offenkundig, die zwar nicht essentiell waren, aber dennoch wünschenswert, um die Praktikabilität der gewählten Ansätze weiter zu untersuchen. Dieser Abschnitt stellt die beiden Anforderungskataloge vor. Eine Diskussion, ob diese Anforderungen erfüllt wurden, findet sich in Kapitel 12 am Ende dieses Berichtes.

1.2.1 Minimalziele

Die Minimalziele wurden als absolut notwendig eingestuft, um den oben erwähnten „Brückenschlag“ als geglückt werten zu können und zu demonstrieren, dass die entwickelte Lösung reif sei, um für ein praktisches Problem eingesetzt zu werden. Die einzelnen Minimalziele werden im Folgenden kurz vorgestellt.

Simulationsumgebung Um die Arbeit der Projektgruppe nicht auf einem realen Quadrokopter testen zu müssen, war es nötig, eine Simulationsumgebung zu verwenden, mit der gefahrlos neuer Quellcode in einer Umgebung getestet werden konnte, die die Eigenschaften eines realen Qudarokopters möglichst realitätsgetreu emulierte.

Steuerungssystem Da das entwickelte System nicht nur komplett autonome Steuerungssysteme, sondern auch Programme zur manuellen Steuerung des Quadrokopters absichern können sollte, musste eine Software implementiert werden, die es erlaubte, den Quadrokopter manuell (mit einem geeigneten Eingabegerät) zu fliegen. Diese Software sollte natürlich mit Techniken der modellgetriebenen Softwareentwicklung umgesetzt werden, um sie mit der neu implementierten Lösung sicher gestalten zu können.

Sicherheitsaspekte Um die Verwendbarkeit der entwickelten Lösung zu testen, war es nötig, die notwendigen Sicherheitsgarantien für den Quadrokopter zu untersuchen und in Form von Aspekten zu implementieren. Beispiele für die als essentiell gewerteten Aspekte waren Kollisionsvermeidung und eine Notlandung bei niedrigem Batteriestand.

Aspektweber Das zentrale Element der Brücke zwischen Aspektorientierung und modellgetriebener Softwareentwicklung stellte der zu entwickelnde Aspektweber da. Diese Software hat die Aufgabe, die oben beschriebenen Sicherheitsaspekte und das Steuerungssystem zu einem sicheren Steuerungssystem zu vereinigen. Dieses System würde dann beispielsweise automatisch verhindern, dass ein Pilot den Quadrokopter gegen ein Hindernis fliegen und

diesen dadurch beschädigen kann. Der Aspektweber musste einfach zu benutzen sein und sich in existierende Technologien integrieren.

1.2.2 Optionale Ziele

Einige weitere Ziele wurden zwar als nicht essentiell, aber als wünschenswert eingestuft. Diese waren zwar nicht nötig, um das Kernziel zu erreichen, erlaubten es aber, die Nützlichkeit der entwickelten Software unter Beweis zu stellen. Diese Ziele sind im Folgenden kurz erläutert.

Missionen Um die Verwendbarkeit der Software nicht nur für manuelle, sondern auch für programmatische Steuerung unter Beweis zu stellen, mussten Steuerprogramme entwickelt werden, die den Quadrokopter nutzen, um Missionen durchzuführen, wie beispielsweise das Erreichen eines Ziels in einem nicht erforschten Gebäude. Diese Missionen sollten selbstverständlich auch modellgetrieben entwickelt werden.

Aspekte für Missionen Die Modellierung von Missionen würde es selbstverständlich auch erlauben, diese mit Techniken der Aspektorientierung zu erweitern. Dazu sollte untersucht werden, welche Aspekte zur Aufwertung von Missionen sinnvoll sind und diese mit den geschaffenen Werkzeugen implementiert werden.

Erweiterte Aspektorientierung Klassische Aspektorientierung ist auf Quellcode-zentrische Transformation von Programmen ausgelegt, orientiert sich also oft an Funktionen, Methoden etc. Die Erweiterung auf Modelle könnte es erlauben, über die Grenzen dieser Techniken hinaus Ansätze zur Verschmelzung von Programm und Aspekten zu finden. Das Potenzial solcher Techniken sollte untersucht werden und möglicherweise im Aspektweber umgesetzt werden.

1.3 Aufbau der Arbeit

Dieser Endbericht gliedert sich im Wesentlichen in drei Teile: Die Kapitel 1 und 2 erläutern die Grundlagen, die zum Verständnis des Berichtes benötigt werden, Kapitel 3 erläutert den Aufbau und das Zusammenspiel der entwickelten Softwarekomponenten, ohne auf die Implementierung dieser Komponenten einzugehen. Der zweite Teil betrachtet die einzelnen Komponenten, wobei die Erklärungen einem Top-Down-Ansatz folgen: Kapitel 4 erläutert die Interaktion mit dem Benutzer, während sich Kapitel 5 mit der Weitergabe von Befehlen an den Quadrokopter und Kapitel 6 mit der Befehlsverarbeitung in der Simulationsumgebung befassen. Das folgende Kapitel 7 beschreibt, wie die Eingabeverarbeitung mit den restlichen Komponenten durch ein Modell verzahnt wurde. Das Kapitel 8 beschäftigt sich mit den Modellen, die Missionen realisieren. In den nächsten Kapiteln 9 sowie 10 werden der Aspektweber und die Sicherheitsaspekte beschrieben, die in die vorher beschriebenen Modelle eingewebt wurden. Der dritte Teil umfasst Kapitel 11, welchen ohne Bezug zu den vorherigen Kapiteln einige genutzte Techniken zur Softwareentwicklung skizziert und Kapitel 12, in dem die Ergebnisse zusammen gefasst werden.

KAPITEL 2

Grundlagen

In diesem Kapitel werden alle Grundlagen vermittelt, die zum Verständnis der folgenden Kapitel benötigt werden. Dazu gehören sowohl Konzepte aus dem Bereich der Softwareentwicklung, wie zum Beispiel Aspektorientierung, als auch Hardware nahe Themen wie die Funktionalität eines Quadrokopters. Zusätzlich wird auf das Prinzip der Stabilisierung des Quadrkopers in der Simulation, sowie der notwendigen physikalischen Prinzipien, eingegangen.

2.1 Aspektorientierung

Die aspektorientierte Programmierung [LR09] ist, ähnlich der objektorientierten Programmierung, ein Paradigma zur strukturierten Entwicklung von Software. Aspektorientierung dient dazu, die Möglichkeiten anderer Paradigmen, wie der Objektorientierung, zu erweitern. Bei der Entwicklung von Software wird meist angestrebt, notwendige Funktionen und Anliegen in unabhängige Module zu unterteilen, was auch *Seperation of Concerns* genannt wird. In der Praxis existieren jedoch Anforderungen, die mit gängigen Techniken nicht in unabhängige Module unterteilt werden können, da diese Anliegen sich quer durch die ganze Software ziehen und eine Vielzahl von Modulen betreffen. Ein Beispiel für eine solche Funktionalität ist das Logging und Tracing oder Sicherheit. Dadurch kann das sogenannte *Code-Scattering* entstehen, falls eine Funktionalität über mehrere Module hinweg implementiert ist. Auch entsteht das *Code-Tangling* sobald einzelne Module nicht mehr ausschließlich ihre vorgesehene Funktionalität implementieren, sondern auch Teile, die beispielsweise Logging oder Sicherheit betreffen. Das Problem der miteinander verwobenen Anforderungen wird auch als *Cross-Cutting Concerns* bezeichnet, denn sie „schneiden“ quer durch alle logischen Schichten des Systems. Die Aspektorientierung stellt Möglichkeiten zur Verfügung, mit der diese *Cross-Cutting Concerns*, auf deutsch *querschneidende Belange* in separaten Modulen, sogenannten *Aspekten*, realisiert werden können.

Ein querschneidender Belang wird bei der Implementierung der anderen Komponenten nicht berücksichtigt und als separater Aspekt implementiert. In diesem Aspekt wird sowohl die geforderte Funktionalität realisiert als auch die Stellen definiert, an denen sie in die anderen Komponenten eingefügt werden müssen. Die Verhalten der Module wird somit nachträglich durch Aspekte modifiziert. Dieser Vorgang wird *Einweben eines Aspekts* durch einen *Aspektweber* genannt. Nach dem Einweben werden zusätzliche Schritte in die Ausführung der Methoden, einer Klasse extra Datenelemente hinzugefügt, oder die Vererbungshierarchie angepasst. Im Folgenden werden grundlegende Begriffe der Aspektorientierung erläutert.

Interceptor Während der Ausführung des Programms muss der reguläre Programmablauf unterbrochen werden und der zum Aspekt gehörende Programmabschnitt ausgeführt werden. Ein Interceptor unterbricht an der richtigen Stelle die Ausführung des Programms, führt weiteren Aspekt-Code aus und setzt anschließend an der gleichen Stelle im eigentlichen Quellcode die Ausführung fort.

Joinpoint Ein Joinpoint spezifiziert die Stelle im Quellcode, an welcher der Aspektweber den Aspekt einfügen muss. Joinpoints können Methodenausführungen sein. Sie können aber

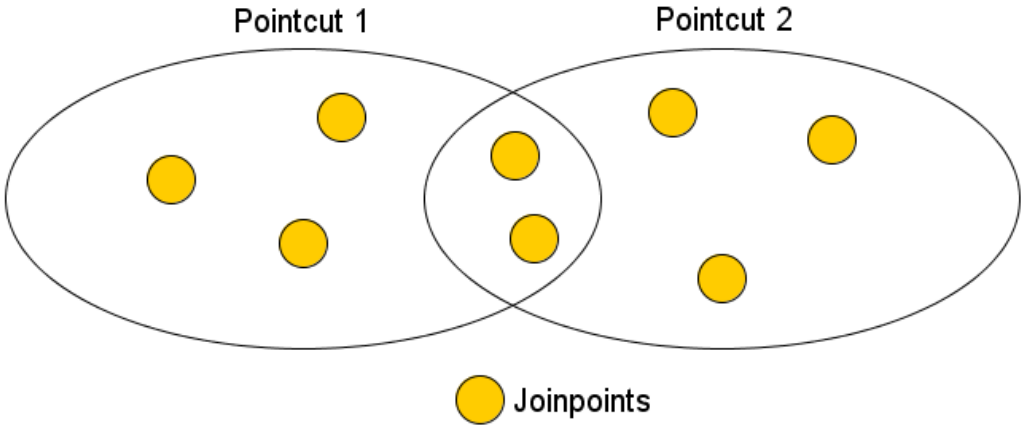


Abbildung 2.1: Beziehung zwischen Joinpoints und Pointcuts

auch andere Joinpoints definieren: die Zugriffe auf die Attribute eines Objekts, die Aufrufe der Operationen einer Klasse, die Erstellung von Objekten oder auch das Auslösen einer Exception.

Pointcut Ein Pointcut ist eine Menge von Joinpoints. Ein Pointcut definiert, an welchen konkreten Joinpoints Interceptoren aufgerufen werden sollen. Während Joinpoints nur Einstiegspunkte für Interceptoren definieren, legen Pointcuts fest, welche Joinpoints in einem konkreten Fall eingesetzt werden sollen. Innerhalb eines Aspekts können mehrere Pointcuts definiert werden. Pointcuts müssen nicht disjunkt sein. Wie in Abbildung 2.1 dargestellt ist, können verschiedene Pointcuts teilweise die gleichen Joinpoints abdecken, so dass ein Joinpoint zu mehreren Pointcuts gehören kann und ein Pointcut eine Menge von Joinpoints ist.

Advice Ein Advice beschreibt, was an den Joinpoints eines Pointcuts passiert. Das heißt, ein Advice legt fest, welche Interceptoren in den Programmfluss eingefügt werden und welcher Code an diesen Stellen ausgeführt wird, der die Funktionalität des querschneidenen Belangs implementiert.

Aspekt Joinpoints, Pointcuts und Advices werden in einem Aspekt zusammengefasst.

Aspektweber Der Aspektweber ist das Werkzeug, mit dem der Aspekt in die bestehende Funktionalität eingewebt wird.

2.2 Modellgetriebene Softwareentwicklung

Bei der Anwendungsentwicklung bedienen sich die Entwickler diverser Entwurfsmuster, deren Implementierungsdetails die Qualitätsmerkmale wie Wiederverwendbarkeit, Wartbarkeit, Performance, etc. beeinflussen. Auf der Ebene des Quellcodes sind diese Entwurfsmuster aufgrund des geringen Abstraktionsniveaus und zahlreichen Implementierungdetails für einen Leser nur schwer oder gar nicht zu erkennen. In vielen Projekten wird versucht diese Entwurfsmuster zu realisieren. Problematisch ist, dass zwischen Modell und Code nur eine gedankliche Verbindung besteht, die von einem Softwareentwickler manuell umgesetzt werden muss. Das Modell stellt nur eine Form der Dokumentation dar. Die Modelle bieten die Möglichkeit, durch das Ausblenden irrelevanter Details die Verständlichkeit von Systemen zu erhöhen. Die Modelle mit klar definierter Syntax und Semantik ermöglichen, verständlichere Sichten auf Systeme zu erlangen und ihre Komplexität zu beherrschen. Dabei kann deren Verwendung in Softwareentwicklung verschiedenen Zwecken dienen, beispielsweise zur Unterstützung bei Softwaresystemherstellung, oder ob Modelle bei der Realisierung von Systemen konstruktiv genutzt werden. Bei der Anwendung zum Unterstützungszweck hat die Erstellung der Modelle keinen direkten Einfluss auf das zu erstellende System. Diese Art, Modelle zu verwenden, wird als modellbasierte Softwareentwicklung bezeichnet [Ste+07].

Bei dem Ansatz der „modellgetriebenen Softwareentwicklung “ (MDSD, Model Driven Software Development) [Ste+07] beeinflusst die Modellerstellung direkt das System. Hier sind die erstellten Modelle Implementierungsartefakte von der gleichen Wichtigkeit wie der Quellcode einer klassischen Programmiersprache. Der Übergang vom Modell zum Code wird mit Hilfe eines Generators und einer Reihe definierter Transformationsvorschriften vorgenommen. Modelle sind — anders als Code — durch eine grafische Darstellung für die meisten Menschen verständlich und erleichtern so die Entwicklung auch für inhaltliche Experten, die keine Codeentwicklungskompetenzen haben. Der Programmcode einer Software wird in drei verschiedene Teile geteilt: Generierter, schematischer und individueller Sourcecode. Der generierte Code ist der Teil einer Software, der für alle Komponenten eines Softwaresystems identisch ist und automatisch aus der Entwicklungsumgebung erzeugt wird. Der schematische Sourcecode ist der Anteil, in dem zwar nicht alle Komponenten eines Softwaresystems identisch, aber zumindest systematisch gleich sind. Zm Beispiel können die Komponenten die gleichen Design-Muster haben. Der individuelle Code ist anwendungsspezifisch unterschiedlich und wird manuell erstellt. Modellgetrieben entwickelt wird lediglich der schematische Teil des Programmcodes. Dieser schematische Teil der Software soll automatisch aus einem domänenspezifischen Anwendungsmodell in die Programmiersprachen der Zielplattform überführt werden. Dies geschieht mit Hilfe eines Generators und einer Reihe definierter Transformationsvorschriften.

Bei MDSD werden Modelle als abstrakt und formal angesehen. Sie sind abstrakt, weil implementierungsspezifische Details weggelassen werden. Graphisch dargestellt, unterstützen die Modelle das Verständnis für die Problemstellung. Die Modelle sind formal, da die genutzten Modellierungssprachen meist so genannte domänenspezifische Sprachen (engl.: Domain Specific Language – DSL) sind. Dabei handelt es sich um die für eine bestimmte Anwendungsdomäne angepasste Abstraktion der eingeschränkten Ausdrucksmächtigkeit, durch die die Distanz zwischen Softwarelösung und tatsächlicher Problemstellung verringert werden soll. Obwohl viele UML-Werkzeuge signifikant verbessert wurden und über intelligente Werkzeuge zur Erstellung von Benutzeroberflächen, Anwendung von Entwurfsmustern und über Codeskelettgenerierung verfügen, sind sie nicht in der Lage, Änderungen am Designmuster automatisch und iterativ auf den Quellcode der gesamten Anwendung zu übertragen. Bei MDSD werden fast alle Teile des Quellcodes und nicht nur Codeskelette aus den Modellen generiert. Dies setzt voraus, dass die Modelle dementsprechend exakt und ausdrucksstark sein müssen, um Applikationen samt ihrer Funktionalität exakt beschreiben zu können. Aus diesem Grund gibt es keine allgemeine und für alle Bereiche passende Modellierungssprache, sie hätte entweder einen sehr großen Umfang oder wäre zu ungenau und abstrakt. Daher setzt MDSD immer eine zu entwickelnde domänenspezifische Sprache voraus, die auf das zu lösende Problem angepasst ist. Nur wenn die Anwendung in all ihren Funktionen mithilfe dieser Sprache genau beschrieben wird, ist es möglich, durch entsprechende Generatoren das Modell der Applikation in ausführbaren Code zu transformieren. Bei MDSD wird nicht nur übersetzbarer Quellcode erzeugt, sondern auch Modelle und Infrastrukturen, mit deren Hilfe der Quellcode erst übersetzt bzw. transformiert werden kann.

Ziel der modellgetriebenen Softwareentwicklung

Der Ansatz der MDSD verfolgt das Ziel der Automatisierung des Software-Entwicklungsprozesses und der Verbesserung der Übersichtlichkeit und Verständlichkeit der entwickelten Systeme. Im Folgenden werden die Ziele näher betrachtet.

- größere Entwicklungseffizienz: Durch die automatisierte Erstellung von sich wiederholenden Codeabschnitten kann der Entwicklungsprozess beschleunigt werden.
- bessere Integration von Fachexperten: Die Modelle verbergen die irrelevanten Details und ermöglichen dadurch eine kompaktere und übersichtlichere Beschreibung von Systemen. Dies fördert die Kommunikation zwischen Fachexperten, Entwicklern und Auftraggeber.

- bessere Änderbarkeit der Software: Dies spielt eine wichtige Rolle, besonders in der Wartungsphase, wobei Fehler behoben werden und das System an geänderte Anforderungen angepasst wird.

Begriffe im MDSD

Im Folgenden werden die drei wichtigsten Begriffe für die modellgetriebene Entwicklung / MDSD [Jör13] kurz vorgestellt. Diese Begriffe haben in verschiedenen technologischen Bereichen unterschiedliche, etwas abgewandelte, Bedeutungen. Von daher wird versucht, Definitionen zu präsentieren, welche sowohl für das Projekt als auch für MDSD im Allgemeinen zutreffen und eindeutig sind.

Modelle Modelle sind eine vereinfachte Darstellung einer Applikation, da sie nicht alle Details des Originals abbilden. Sie repräsentieren Strukturen, Funktionen oder Verhaltensweisen auf abstrakte Weise.

Domäne Es wird immer im Kontext von Domänen entwickelt, d.h. bezogen auf ein bestimmtes abgegrenztes Problem, Wissens- oder Interessengebiet. Modelle gehören zu einem solchen Gebiet und grenzen sich so voneinander ab. Dabei kann eine Domäne in kleinere, weniger komplizierte Subdomänen unterteilt werden, zu denen jeweils Modelle entwickelt werden können. Hier gilt das grundlegende Prinzip: „Teile und herrsche!“. In diesem Szenario, welches auf eine große Klasse von Softwaresystemen zutrifft, besteht der Bedarf, alle beteiligten Modellen miteinander zu integrieren, um die Konsistenz des Gesamtsystems sicherzustellen. Zwei Modelle sind miteinander integriert, wenn es eine explizite Verbindung zwischen ihnen gibt, die für ein automatisiertes Werkzeug erkennbar ist. Diese Informationen sind in entsprechenden Artefakten enthalten, die auf der Basis einer formalen Notation erstellte Dokumente sind.

Metamodelle Ein Metamodell formalisiert die Strukturen einer Domäne. Es bildet die Grundlage fur eine Automatisierung im Rahmen der Entwicklung. Metamodelle beschreiben wesentliche Eigenschaften von Modellen, die wiederum rekursiv Metamodelle für andere Modelle sein können. Alle Modelle, welche von einem Metamodell beschrieben werden, sind Instanzen des Metamodells.

Transformation

Neben der Codegenerierung sind Modelltransformationen eine der wichtigsten Techniken der MDSD. Auf der Grundlage eines Metamodells können verschiedene Modelle definiert werden, die das gleiche Konzept beschreiben. In solchen Fällen ist es möglich, ein Modell in ein anderes zu transformieren. Dies setzt voraus, dass beide Modelle ein gemeinsames Metamodell haben. Eine Modelltransformation ist eine Abbildung eines Modells auf ein anderes. Der Transformationsbegriff steht im direkten Zusammenhang mit den Begriffen Modell und Metamodell. Durch Modelltransformationen werden z.B. plattformunabhängige Modelle in plattformabhängige Modelle transformiert. Solche Transformationen sind oft ein Schritt des Generators zwischen der Validierung des übergebenen Modells und der Codegenerierung. Sie werden ohne ein manuelles Eingreifen ausgeführt. Eine einfache Transformation ist beispielsweise das Hinzufügen von Funktionalität zu einem existierenden Modell. Eine Transformation kann aber auch aus mehreren Modellen ein neues Modell erstellen, das Instanz eines anderen Metamodells ist.

Im Zuge der Softwareentwicklung begegnet man oft aufwändig zu modellierenden Funktionalitäten, die wiederholt eingesetzt und verwendet werden. Durch Modelltransformation kann die Wiederverwendbarkeit solcher Funktionalitäten erreicht werden. Anstatt mehrmals dieselbe Funktionalität zu modellieren, wird sie einmal modelliert und durch Transformation an den benötigten Stellen ergänzt. Transformationen können helfen, Plattformunabhängigkeit zu erreichen, wobei die Anforderungen und Eigenschaften der Plattform, auf der die Software nach der Bereitstellung ausgeführt werden soll, auf der Modellebene unberücksichtigt bleiben und anschließend während einer Transformation des Modells in ein anderes Modell respektiert werden. Drei verschiedene Arten von Transformation sind zu unterscheiden [Jör13]:

- Bei der *Modellmodifikation*, die in Abbildung 2.2 veranschaulicht wird, wird das Ausgangsmodell nur um zusätzliche Bestandteile ergänzt. Das entstandene Modell respektiert dabei das Metamodell des Modells aus der Eingabe.
- Bei der *Modelltransformation*, wie in Abbildung 2.3 zu sehen, wird ein Modell in ein anderes Modell überführt, wobei das neu entstandene Modell in der Regel ein anderes Metamodell als das Eingabemodell hat. Ein Beispiel für eine solche Transformation ist die Überführung eines Klassendiagramms in ein Metamodell für ein relationales Datenbankschema.
- Das in Abbildung 2.4 gezeigte *Model-Weaving* ist ein Spezialfall der Modellmodifikation. Hier werden mehrere Modelle, die unterschiedliche Metamodelle haben können, miteinander verwebt (weaving).

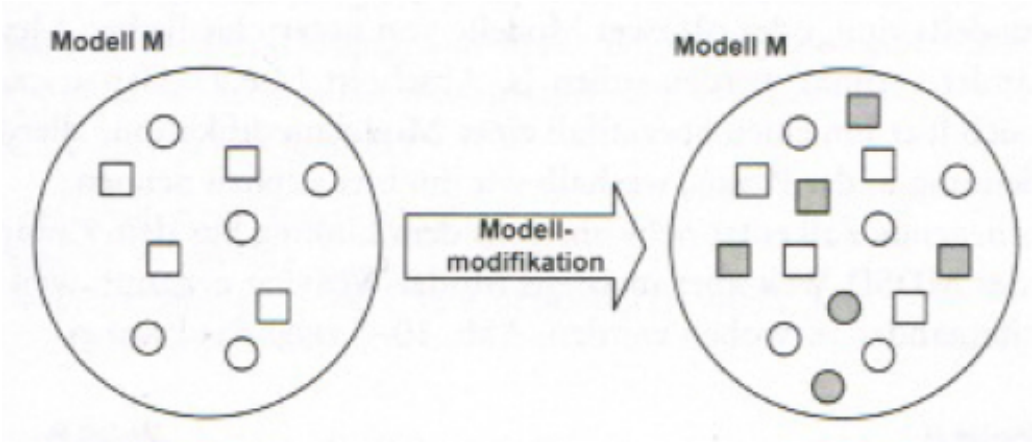


Abbildung 2.2: Modellmodifikation [Jör13]

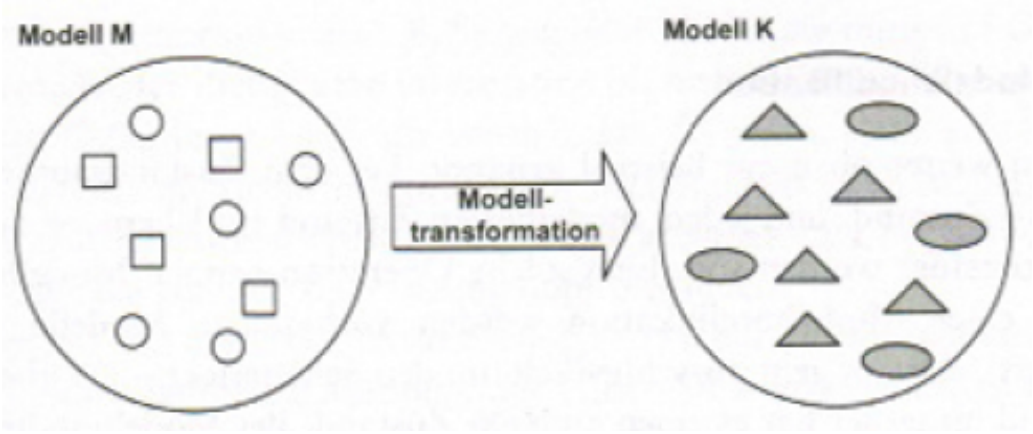


Abbildung 2.3: Modelltransformation [Jör13]

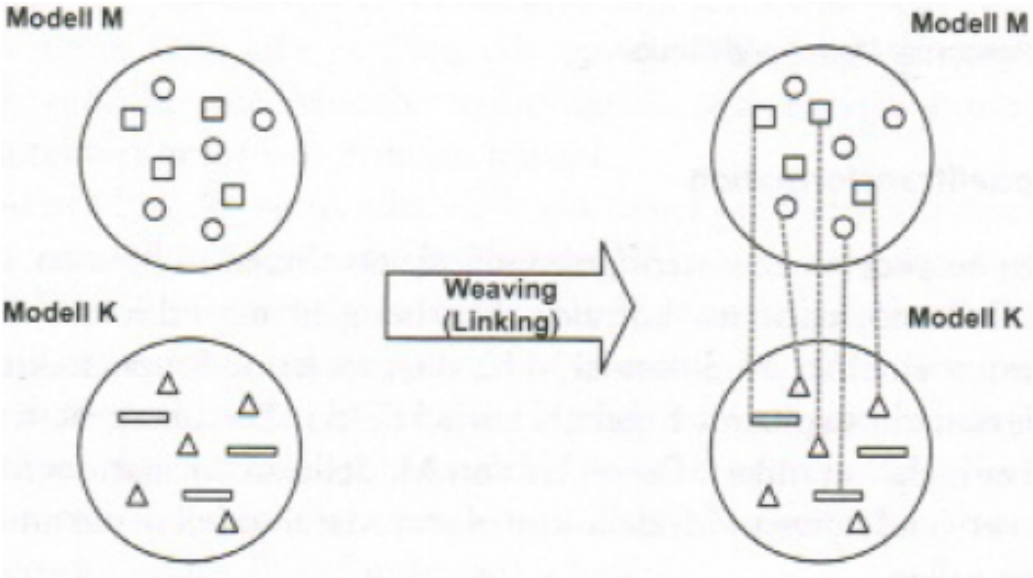


Abbildung 2.4: Model-weaving [Jör13]

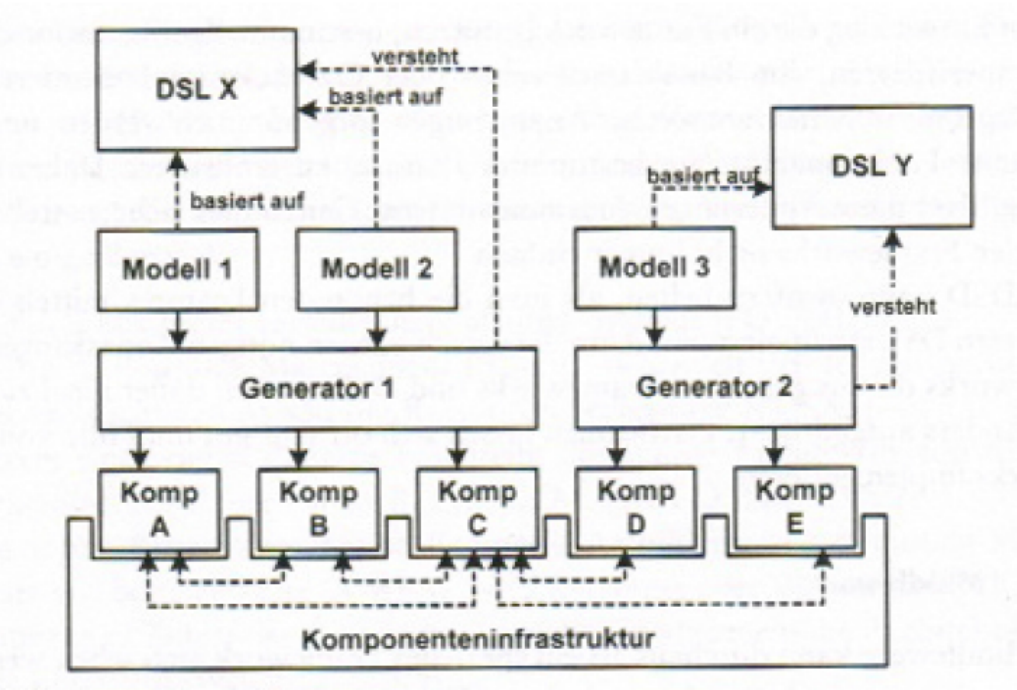


Abbildung 2.5: Generatoren für Komponenteninfrastruktur [Jör13]

Codegenerierung

Der Prozess der Codegenerierung besteht aus mehreren Schritten. Zuerst werden die Modelle eingelesen, zusammengefügt, auf Validität überprüft und gegebenfalls transformiert. Das Ergebnis dieses Prozesses wird anschließend in ausführbaren Code umgewandelt. Codegeneratoren sind Programme, die ein Modell als Eingabe bekommen und Programme in einer spezifischen Programmiersprache ausgeben. Um aus einem Modell Code erzeugen zu können muss ein Generator Vorschriften haben. Diese Vorschriften sind auf das Metamodell zurückzuführen. Das heißt, dass ein Generator für ein bestimmtes Metamodell geschrieben wird. Er kann aus allen Instanzen des Metamodells Code generieren. Dies bedeutet, dass der Generator bei jeder Änderung des Metamodells angepasst werden muss. Diese Generatoren können zu einer Infrastruktur zusammengesetzt werden, wie sie in Abbildung 2.5 illustriert wird.

2.2.1 jABC

Im Folgenden wird ein auf Java basierendes Tool vorgestellt, mit dem man im Sinne von MDSD Modelle in einer grafischen Umgebung erstellen kann. Das **jABC** [Ste+07] ist ein vom Lehrstuhl für Programmiersysteme der TU Dortmund entwickeltes Programm zur modellgetriebenen Ent-

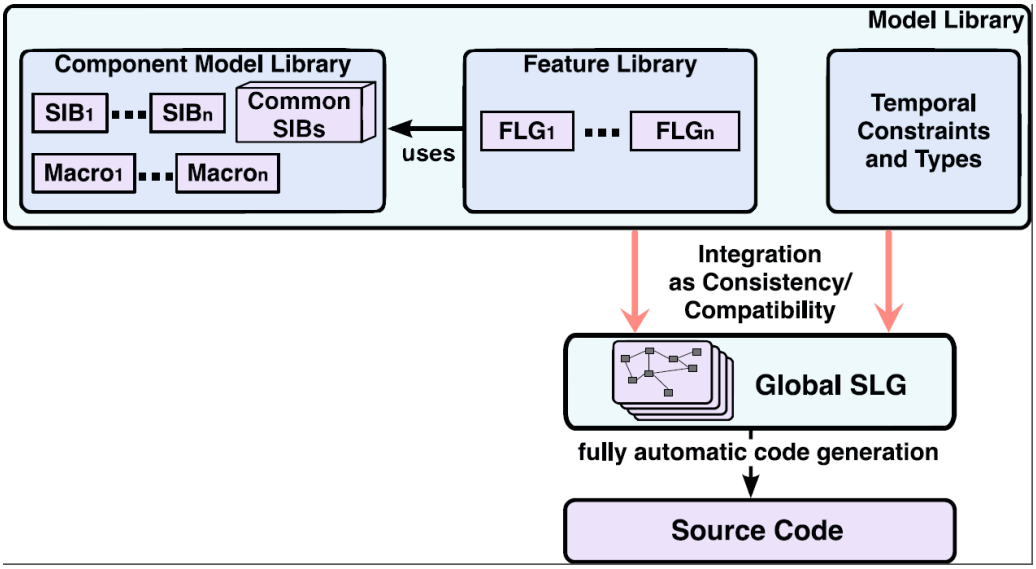


Abbildung 2.6: Das jABC-Framework [JSM11]

wicklung von Software. Es ermöglicht, ohne Kenntnisse einer klassischen Programmiersprache Anwendungen zu modellieren.

In Abbildung 2.6 ist die Komponente **Global SLG** zu sehen. **SLG** steht für *Service Logic Graph* und bezeichnet ein Modell, welches in jABC entworfen wurde. Ein Modell wird dabei als ein gerichteter Graph notiert, welcher das gewünschte Verhalten kodiert. Eine komplette jABC-Anwendung kann aus einem oder mehreren SLGs bestehen.

Ein SLG besteht aus Knoten und Kanten, die im jABC-Konzept SIBs und Branches genannt werden. Die Knoten repräsentieren die aufeinanderfolgenden Aktionen eines Arbeitsablaufs, die Kanten verbinden diese. Wie nun genau eine Aktion definiert ist, lässt sich der **Model Library**, zu finden im oberen Teil der Abbildung, entnehmen. Die Model Library stellt die Menge aller verfügbaren Aktionen dar, die in einem SLG verwendet werden können.

Innerhalb der Model Library befinden sich wiederum drei Gruppen: Die **Component Model Library**, die **Feature Library** und die **Temporal Constraints and Types**. In der **Component Model Library** befinden sich die **SIBs** *(Service Independent Building Blocks)*. Dies sind die Grundbausteine zur Zusammenstellung eines Workflows. Sie repräsentieren einen atomaren Dienst mit einer einzigen Funktionalität, sei es von einem Altsystem, einer COTS *(Commercial off-the-shelf)* Software[1] oder einem Webservice. In der Standardinstallation des jABC ist be-

[1] *Commercial off-the-shelf* ist eine englische Umschreibung für seriengefertigte Produkte. *Commercial off-the-shelf Software* steht dementsprechend für Standardsoftware, welche in großer Stückzahl „von der Stange“ gekauft werden kann.

reits eine umfassende Bibliothek von SIBs enthalten. Diese sind in Abbildung 2.6 in dem Block *Common SIBs* dargestellt.

Neben diesen atomaren Diensten können auch andere SLGs als Bausteine verwendet werden. Diese werden in jABC als **Makros** bezeichnet. Sie verhalten sich in der Nutzung wie die SIBs. Der einzige Unterschied ist, dass ein SIB als atomarer Dienst stets eine konkrete (nicht jABC-basierte) Implementierung hat. Ein Makro hingegen setzt sich aus SIBs und gegebenenfalls weiteren Makros zusammen.

Die **Feature Library** bezeichnet die Menge an vollständigen, wiederverwendbaren SLGs. Dabei kann es sich beispielsweise um Fehlerbehandlung oder Sicherheitsmanagement handeln. Diese Features werden nur einmal als SLG modelliert und stehen allen zukünftigen Applikationen zur Verfügung.

Schließlich gibt es noch die **Temporal Constraints and Types**, die einzuhaltenden Regeln für die Applikation. Hierbei handelt es sich um formale Ausdrücke welche mithilfe von Techniken des *Model Checking* geprüft werden. Mit diesen Regeln und anderen Techniken kann die Konsistenz und Validität eines Modells geprüft werden.

2.2.2 Anwendungsbeispiel jABC

Der Anschaulichkeit halber soll eine Beispielanwendung in jABC betrachtet und erläutert werden. In Abbildung 2.7 ist das Programmfenster von jABC zu sehen. Auf der linken Seite findet man in der oberen Hälfte den Projektexplorer sowie die Taxonomie der SIBs. In der unteren Hälfte sind Inspektoren zu finden.

jABC wird mit einer Reihe von Beispielprojekten ausgeliefert, in Abbildung 2.7 sind einige davon zu erkennen: Im Ordner `mini` befinden sich die Modelle `Countdown`, `NewYear` und `TextFile`. Das Modell `NewYear` gehört jedoch nicht zu den standardmäßig mitgelieferten Beispielen, es wurde für diesen Endbericht angelegt.

Im rechten Teil der Abbildung, dem Canvas, ist eine Modellierung für einen simplen Countdown-Timer zu sehen. Anhand dieses Beispiels sollen nun grundlegende Elemente von jABC-Modellen erläutert werden [Ste+07].

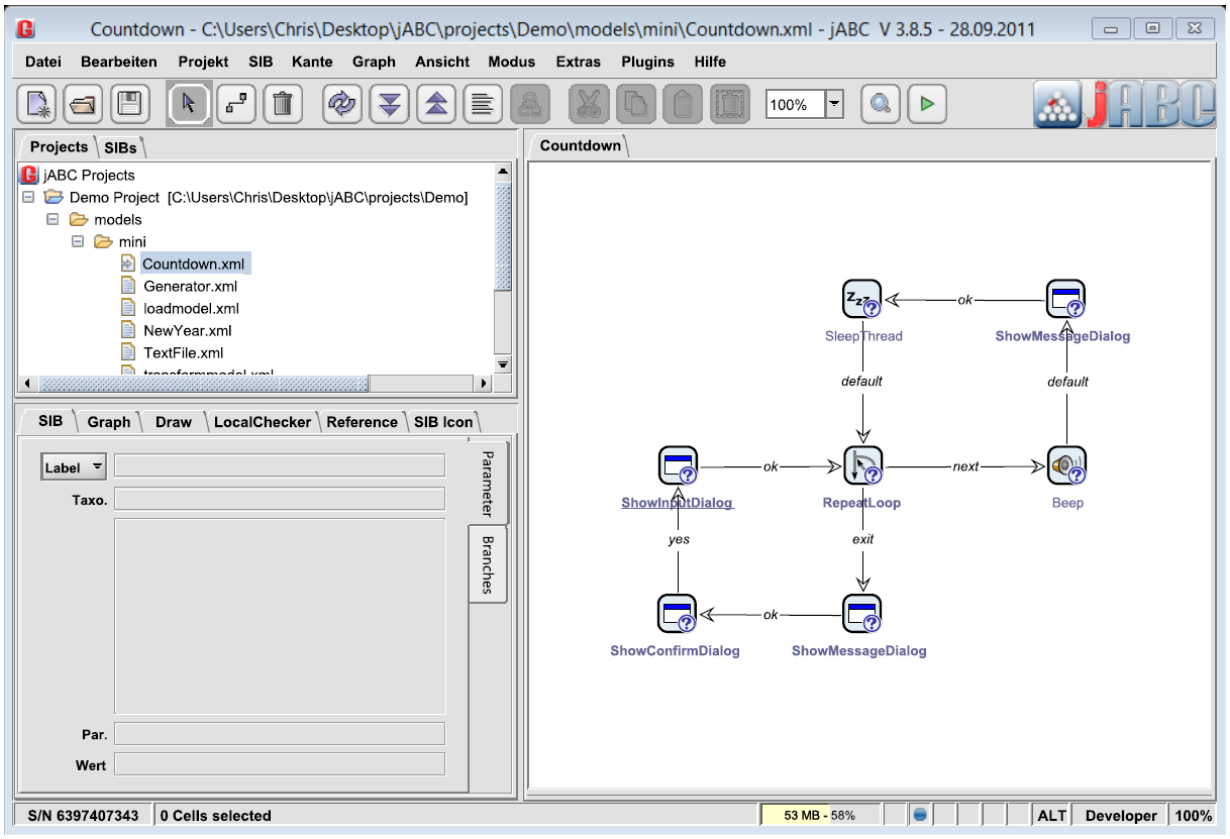


Abbildung 2.7: jABC Programmfenster

Einstiegspunkt

Als erstes benötigt jedes Modell einen **Einstiegspunkt** (`StartSIB`). Dies ist die Anweisung, mit welcher der Arbeitsablauf beginnt. In jABC ist das Start-SIB anhand der unterstrichenen Bezeichnung zu erkennen. In dem `Countdown`-Beispiel ist dies das SIB `ShowInputDialog`.

SIBs

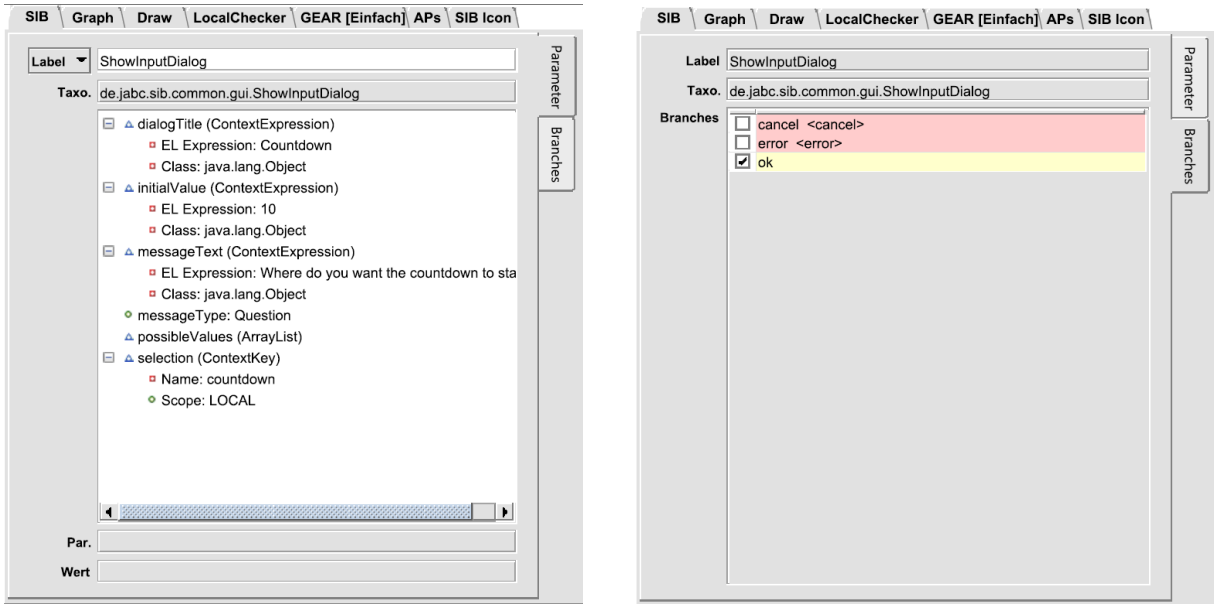
SIBs sind die Grundbausteine für Modelle in jABC. IT-Experten sind für die Implementierung dieser einzelnen Funktionsblöcke zuständig. Anwendungsexperten schalten diese Funktionsblöcke wiederum zu einer Anwendung zusammen. Wie bereits am Anfang dieses Abschnitts erwähnt, liefert die Standard-jABC-Installation eine umfassende SIB-Bibliothek mit. Daher besteht auch das gesamte `Countdown`-Modell aus mitgelieferten SIBs. In der Regel beschreiben die Namen dieser SIBs ihre Funktionalität. Das Einstiegs-SIB `ShowInputDialog` beispielsweise zeigt dem Anwender einen Dialog, in dem er eine Eingabe *(Input)* vornehmen kann. Darüber hinaus werden Informationen über jedes vorgefertigte SIB in einem Tooltip und in der Onlinedokumentation zur Verfügung gestellt.

Parameter und Branches

In Abbildung 2.8(a) finden sich alle **Parameter** des `ShowInputDialog`-SIBs. Dazu gehören Titel und Nachricht des Dialogs, sowie der initiale Wert des Eingabefelds. Besonders wichtig ist der Parameter `selection`. Dieser bezieht sich auf die getroffene Eingabe im Dialogfeld. Der Parameter entspricht einer Variable, der ein Name und ihr Geltungsbereich zugewiesen werden kann. Im weiteren Verlauf dieses Abschnitts wird klar, wozu die Variable `countdown` benötigt wird.

Nachdem ein SIB ausgeführt wurde, kann es mehrere Resultate haben. Diese werden über sogenannte **Branches** modelliert. Bei Betrachtung von Abbildung 2.8(b) ist zu erkennen, dass `ShowInputDialog` drei Ausgaben haben kann: *OK, Cancel*, und *Error*. *OK* und *Cancel* entsprechen den betätigten Schaltflächen im Dialog, *Error* beschreibt das generelle Auftreten eines Fehlers.

Aus Abbildung 2.7 lässt sich entnehmen, dass von dem SIB `ShowInputDialog` jedoch lediglich eine Kante ausgeht. Dieser Kante ist der Branch *OK* zugeordnet. Das heißt, dass nur in dem Fall, dass der Dialog mit OK bestätigt wird, der Kante gefolgt wird.



(a) Parameter-Tab (b) Branches-Tab

Abbildung 2.8: Die Parameter- und Branches-Tabs im Eigenschaftenfenster

Es stellt sich also die Frage was für die Branches *Cancel* und *Error* geschieht. Dazu ist ein erneuter Blick auf Abbildung 2.8(b) nötig. Mit der dort sichtbaren Notation `cancel <cancel>` beziehungsweise `error <error>` wird dargestellt, dass der Branch des SIBs (z.B. *Cancel*) auf den Modellbranch (ebenfalls *Cancel*) weitergeleitet wird. Konkret hieße das im Fall von `ShowInputDialog`, dass, sobald in dem Dialog auf Abbrechen geklickt wird, auch das gesamte Modell abbricht. Analog würde im Falle eines Fehlers im Dialog auch das Modell einen Fehler melden.

Neben den Modellbranches gibt es auch Modellparameter. Diese funktionieren nach dem selben Prinzip wie die Modellbranches: Ein Parameter für ein SIB wird nicht im Kontext des aktuellen SLGs definiert, sondern muss von dem Modell, welches das aktuelle SLG aufruft, angegeben werden.

Variablen

Die in `ShowInputDialog` gesetzten Variable `countdown` soll nun für den Countdown heruntergezählt werden. Dazu sei das `RepeatLoop`-SIB betrachtet (siehe Abb. 2.7) sowie das zugehörige Parameterblatt (Abbildung 2.9).

Der `RepeatLoop` ist ein SIB zur Realisierung von Schleifen mit Zählervariablen. Daher enthält das SIB auch die Parameter `counter` (Zählervariable des SIB), `start` (Anfangswert des

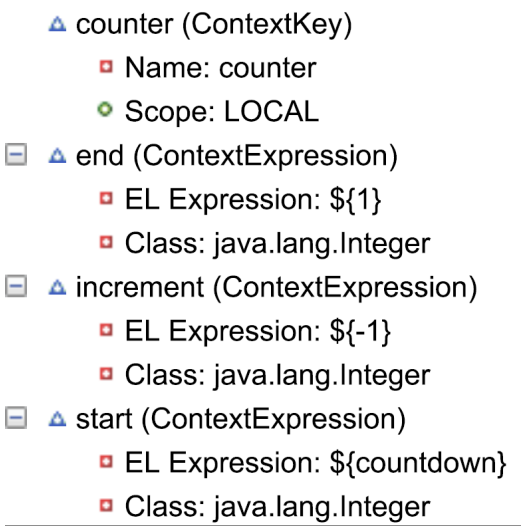


Abbildung 2.9: Variables

Zählers), `increment` (Veränderung des Zählers pro Iteration) und `end` (Zählerstand bei dem die Schleife beendet ist). Bei dem Parameter `start` sieht man nun, dass dieser den Wert `$(countdown)` hat. Das bedeutet, dass die `countdown`-Variable aus dem `ShowInputDialog`-SIB hier wiederverwendet wird, um die Zählervariable zu initiieren. Bei Betrachtung der Belegung von `increment` und `end` lässt sich feststellen, dass `RepeatLoop` von `countdown` bis 1 herunter zählt.

Aus technischer Sicht gibt es bei jABC zwei Arten von Variablen: `ContextKey` und `ContextExpression`. Anhand Abbildung 2.9 lässt sich der Unterschied gut erläutern. `ContextKeys` sind Variablen, auf die andere SIBs zugreifen können. In diesem Fall wäre das die Variable `counter`, also der aktuelle Zählerstand. Bei einem `ContextKey` wird stets definiert, unter welchem Namen auf ihn zugegriffen werden kann und welchen Gültigkeitsbereich er hat:

global: der `ContextKey` kann von allen Modellen adressiert werden

local: der `ContextKey` ist nur für das momentan aktive SLG erreichbar

parent: auf den `ContextKey` kann nur das Elternmodell zugreifen

declared: hierbei wird der Kontext adressiert, der an oberster Stelle auf dem Stack liegt und den `ContextKey` enthält

Beim erwähnten Stack handelt es sich um den *Execution Context Stack*: Sobald ein SLG aufgerufen wird (z.B. über ein Makro) wird auf dem Stack ein Ausführungskontext abgelegt, der alle `ContextKeys` (inklusive deren Belegungen) des entsprechenden SLGs enthält. [Ste+07] empfiehlt jedoch auf die explizite Nutzung der verschiedenen Stackebenen zu verzichten, da dies

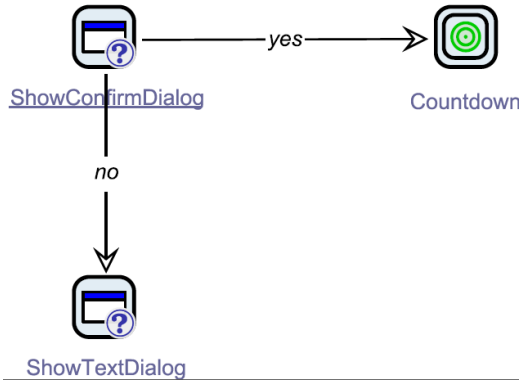


Abbildung 2.10: `NewYear`-Modell

in den meisten Fällen nicht nötig ist und die Anwendung nur unnötig kompliziert macht. Die Benutzung der verschiedenen Stackebenen kommt selten vor, nur wenn es sich unbedingt um eine Modellerekursion handelt oder eine komplexe Modellehierarchie angefordert wird, in der die Existenz von `Scopes` unterschiedlicher Variablen gewährleistet werden müssen.

Mit `ContextExpression` wird ein dynamischer Ausdruck bezeichnet, der auf einen oder mehere `ContextKeys` zurückgreifen kann. Bei der Variable `start` beispielsweise wird auf den Context-Key `countdown` zugegriffen, der vorher im SIB `ShowInputDialog` definiert und gesetzt wurde. Ohne nun weiter im Detail auf die weiteren SIBs einzugehen, erschließt sich recht intuitiv, was das `Countdown`-Modell (Abbildung 2.7) tut:

Nachdem der Nutzer ein Startwert angegeben hat, wird von diesem Wert bis 1 herunter gezählt, und in jedem Schritt ein Signalton ausgegeben, eine Dialogbox mit dem Zählerstand angezeigt und eine kurze Pause eingelegt. Ist der Zähler bei 1 angekommen wird eine Dialogbox angezeigt (*„Happy New Year!“*), sowie ein Bestätigungsdialog, ob die Demo nochmal ausgeführt werden soll. Hierbei ist zu beachten, dass nur der Branch `yes` auf eine konkrete Kante gelegt ist, der Branch `no` wurde mit dem Modellbranch `default` verbunden. Der Name `default` wird für den Branch genutzt, der eine erfolgreiche Ausführung des Modells signalisiert, die keine weitere Behandlung einer Ausnahmesituation durch den Aufrufenden erfordert.

Makros

Alle im `Countdown`-Modell verwendeten Bausteine gehören zu den mitgelieferten SIBs des jABC. Darüber hinaus können jedoch auch fertige Modelle in anderen Modellen als Baustein verwendet werden. Dazu sei das für diese Ausarbeitung zusammengesetzte Modell `NewYear` betrachtet (Abbildung 2.10).

Die Funktionsweise dieses Modells ist recht simpel: Es wird ein Dialog angezeigt, welcher bestätigt oder abgelehnt werden kann. Wird er bestätigt, wird der Countdown gestartet, ansonsten wird eine Meldung ausgegeben. Hier ist zu sehen, dass das Modell `Countdown` als ein **Makro** im Modell `NewYear` eingebettet ist. Des Weiteren lässt sich gut erkennen, wie die Modellbranches von `Countdown` genutzt werden können, um im Falle eines Abbruchs oder Fehlers einen Signalton auszugeben. Zur Erinnerung: In `Countdown` werden die *Cancel*- und *Error*-Branches einzelner SIBs auf die entsprechenden Modellbranches weitergeleitet. Hier lässt sich erkennen wie mit jABC auch Fehlerbehandlung und -weiterreichung realisiert werden können.

Hierarchien

Mithilfe der Makros ist es möglich, beliebig tiefe Modellierungshierarchien zu erstellen. Bei dem oben genannten Beispiel liegt das Modell `NewYear` auf oberster Hierarchieebene, das in `NewYear` aufgerufene SLG `Countdown` liegt auf zweiter Hierarchieebene. Wie die Kapselung bei der Programmierung ist die Nutzung von Makros und das Entwerfen von sinnvollen Hierarchien ein hilfreiches Mittel um die Anwendung übersichtlich zu modellieren.

Dabei spielen Modellbranches und -parameter eine große Rolle. Neben der im vorherigen Abschnitt vorgestellten Weiterleitung der *Cancel*- und *Error*-Branches können beliebig viele eigens definierte Branches als Modellbranches deklariert werden. Modellbranches bieten nicht nur eine Möglichkeit, dem *aufrufenden* Modell Statusrückmeldungen zu geben, sie erlauben es auch im *aufgerufenen* Modell zu definieren, an welchen Stellen das Modell über die Modellbranches verlassen werden kann.

Ebenso wie Branches können auch Parameter auf Modellebene weitergeleitet werden. Dann muss das aufrufende Modell dem Makro die benötigten Parameter übergeben. Diese Modellparameter kann man mit den Eingabeparametern eines Methodenaufrufs vergleichen.

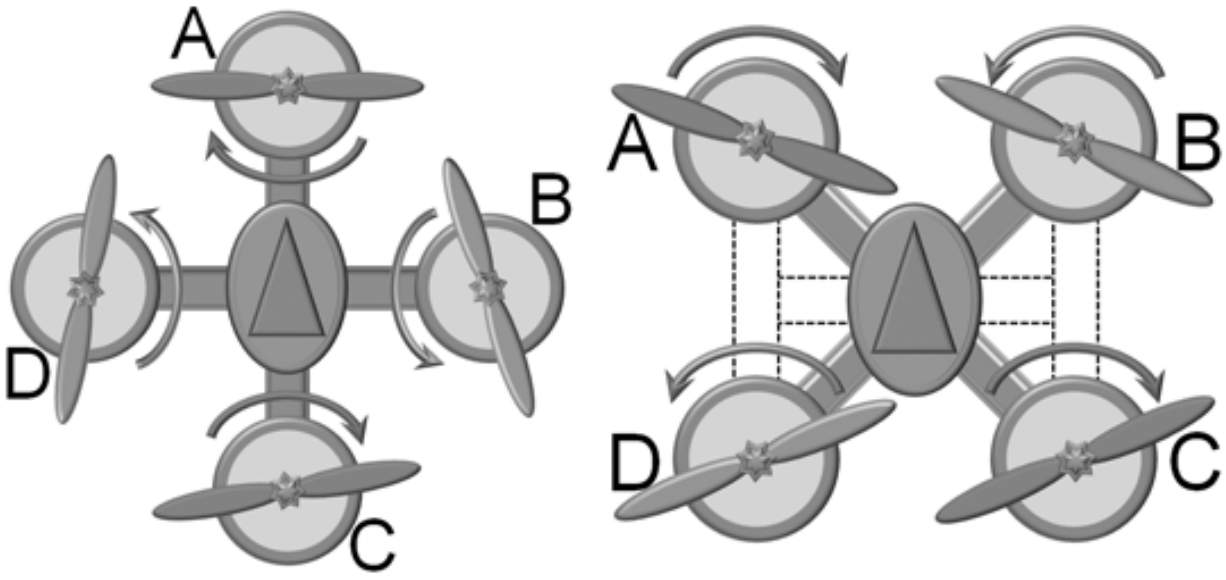


Abbildung 2.11: Plus-Konfiguration und X-Konfiguration eines Quadrokopters [Hei13a; Hei13b]

2.3 Quadrokopter

Dieser Abschnitt erläutert die Konzepte, die dem Bau und dem Flug mit Quadrokoptern zugrunde liegen. Ein Quadrokopter ist ein Hubschrauber mit vier vertikal und starr nach unten ausgerichteten Rotoren. Quadrokopter sind hauptsächlich im Modellflug vertreten. Sie sind unbemannt und haben in der Regel eine Größe bis ca. 1m. Ein Quadrokopter kann sich in alle drei Dimensionen bewegen und ebenso wie ein Hubschrauber auf der Stelle schweben. Die Steuerung erfolgt dabei allein durch Variation der Rotordrehzahl. Elektronische und Software-technische Regelung sorgt für ein stabiles Flugverhalten.

2.3.1 Technischer Aufbau

Für die Anordnung der Rotoren eines Quadrokopters existieren im Wesentlichen zwei Konfigurationen, die Plus-Konfiguration sowie die X- bzw. H-Konfiguration, diese veranschaulicht Abbildung 2.11.

Die in Abbildung 2.11 gezeigte Plus-Konfiguation erlaubt eine einfache Umsetzung der Steuerung, da für Änderungen entlang der Längs- und Querachse nur ein Motorenpaar angesteuert werden muss. Bei der X- bzw. H-Konfiguration müssen stets alle vier Motoren angesteuert werden. Dies erlaubt höhere Drehbeschleunigungen und die Anbringung einer Kamera in Flugrichtung, da diese nicht durch einen Propeller verdeckt wird. Trickreich ist nicht nur die Steuerung

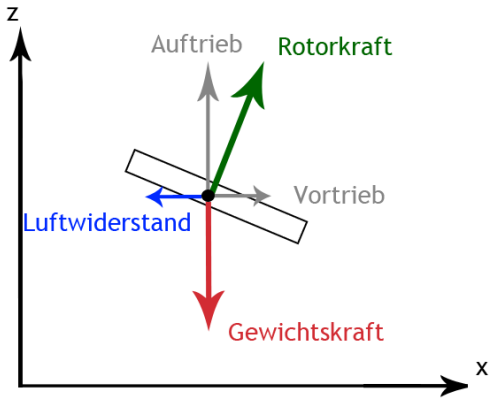


Abbildung 2.12: Geradeausflug

der Motoren bei einem Quadrokopter, sondern auch die Berechnung der auf den Quadrokopter wirkenden Kräfte. Die physikalischen Effekte müssen bei der Steuerung berücksichtigt werden.

2.3.2 Auf einen Quadrokopter wirkende Kräfte

Die Kraft, die jedes Flugobjekt überwinden muss, ist die Gewichtskraft. Sie ist abhängig von dem Gewicht des Flugobjekts. Die Rotoren des Quadrokopters erzeugen eine Rotorkraft, die abhängig von der Drehzahl der Rotoren ist. Die Effizienz eines Rotors nimmt mit der Flughöhe ab. Auf den Quadrokopter wirkt außerdem der Luftwiderstand ein, der ihn abbremst. Als zusätzliche Kraft kann Wind auf den Quadrokopter einwirken, der seine Richtung stets ändern und daher aus allen Richtungen auf den Quadrokopter einwirken kann. Abb. 2.12 veranschaulicht die auf einen Quadrokopter wirkenden Kräfte.

2.3.3 Steuerung eines Quadrokopters

Dieser Abschnitt erklärt, wie ein Quadrokopter gesteuert wird. Im folgenden Abschnitt wird illustriert, wie ein Quadrokopter um alle drei Achsen gesteuert wird und wie ein Quadrokopter beschleunigt und abgebremst wird.

Drehung um die Achsen

Generell erfolgt die Steuerung des Quadrokopters nur durch Erhöhung oder Verringerung der Rotordrehzahl, was eine Änderung des Auftriebs bewirkt. Jeweils zwei Propeller drehen sich im bzw. gegen den Uhrzeigersinn. Solange sich alle vier Propeller gleich schnell drehen, heben sich

die Drehmomente[2] gegenseitig auf und der Quadrokopter dreht sich nicht um die Hochachse, wie es bei einem normalen Hubschrauber passiert. Die Drehung des Quadrokopters erfolgt um die Hochachse (Gierachse). Dies wird dadurch bewirkt, dass die links und rechts drehenden Rotoren mit unterschiedlicher Drehzahl angesteuert werden. Dadurch hebt sich das Drehmoment der Rotoren nicht mehr gegenseitig auf und der Quadrokopter dreht sich um die Hochachse.

Die Drehung um die Längs- bzw. Querachse erreicht man durch die unterschiedliche Ansteuerung der auf den jeweiligen Achsen liegenden Rotoren. Damit der Quadrokopter dabei nicht um die Hochachse giert, müssen die Drehzahlen der links beziehungsweise rechts drehenden Rotoren umgekehrt proportional zueinander verändert werden.

Durch Kombination der Steuerbefehle kann jede der genannten Bewegungen gleichzeitig ausgeführt werden, was den Quadrokopter in alle drei Dimensionen steuerbar macht. Je nach Umsetzung können Quadrokopter auch Loopings, Rollen und enge Turns[3] fliegen.

Horizontaler Geradeausflug & Bremsmanöver

Um in der horizontalen Ebene geradeaus zu fliegen, muss sich der Quadrokopter zunächst um die Querachse nach vorne neigen. Dazu werden die Drehzahlen der Propeller entsprechend angepasst. Hat der Quadrokopter sich um den gewünschten Winkel α geneigt, kann nun die Rotordrehzahl aller Rotoren erhöht werden. Durch die Neigung besitzt die Auftriebskraft der Propeller jetzt sowohl eine Komponente in x- als auch in z-Richtung.

In Abb. 2.12 sind alle auf den Quadrokopter wirkenden Kräfte im Gleichgewicht wodurch er seine Fluggeschwindigkeit und Höhe beibehält. In dieser Fluglage behält der Quadrokopter auch seine Höhe und Geschwindigkeit bei. Würde die Rotordrehzahl verringert werden, so würde sich auch die Auftriebskraft in z-Richtung verringern und der Quadrokopter würde sinken. Um den maximalen Vortrieb zu erzeugen und gleichzeitig die Höhe beizubehalten, muss der Quadrokopter in die größtmögliche Neigung von 30°nach vorne ausgerichtet werden.

Um zu bremsen muss sich der Quadrokopter entgegengesetzt der Flugrichtung nach hinten neigen. Die Kraft der Propeller erhält dadurch eine Kraft-Komponente entgegen der Flugbahn, die den Quadrokopter abbremst. Sobald die Geschwindigkeit 0 beträgt, kann der Quadrokopter die Neigung aufheben um seine Position zu halten.

[2]Ein Drehmoment ist eine Rotationskraft die dafür sorgt, dass ein Körper sich dreht.

[3]Ein Turn bezeichnet eine eng geflogene Kurve die der Quadrokopter mit einer starken seitlichen Neigung durchführt.

Seitwärtsflug / Kurvenflug

Analog zum Geradeausflug kann der Quadrokopter auch seitwärts fliegen, indem er sich um die Längsachse neigt. Dadurch erzeugt die Rotorkraft eine Kraft-Komponente in y-Richtung, die den Quadrokopter auf die linke bzw. rechte Seite bewegt. Der Kurvenflug besteht aus einer Kombination aus Vorwärts- und Seitwärtsflug. Der Quadrokopter neigt sich dazu nach vorne und zu der Seite, zu der die Kurve ausgeführt werden soll. Je steiler dabei der seitliche Anstellwinkel, desto schneller und enger kann die Kurve geflogen werden.

Flugstabilität in Boden- und Deckennähe

Bei Quadrokoptern, die in Bodennähe fliegen, kommt es zu einem physikalischen Bodeneffekt. Durch die Nähe des Bodens kann die durch den Rotor nach unten strömende Luft nicht so schnell entweichen. Dadurch erhöht sich der Auftrieb eines Quadrokopters in Bodennähe. Je näher der Quadrokopter dem Boden ist, desto größer ist die Auftriebsverstärkung. Der umgekehrte Effekt tritt bei Quadrokoptern auf, die nahe an der Decke fliegen. Je näher ein Quadrokopter an der Decke fliegt, desto stärker wird er von ihr „angezogen“.

Drift von der Sollposition und -Ausrichtung

In Quadrokoptern ist ein Mikroprozessor verbaut, der darauf programmiert ist, die Höhe, Position und Ausrichtung des Quadrokopters beizubehalten. Dazu bezieht er seine Lagedaten aus einem elektronischen Gyroskop und einem Ultraschall-Höhenmesser. Allerdings können gewisse Schwankungen beim Halten der Position nicht verhindert werden. Dies liegt nicht allein am Wind, sondern die Steuerung der Rotordrehzahlen kann nicht vollständig exakt ausgeführt werden. Bereits durch diese kleinen Differenzen ändert sich die Lage des Quadrokopters fortlaufend. Diesen ungewollten Schwankungen wirkt die eingebaute Elektronik entgegen. Doch sie kann die Schwankungen nicht voraussehen, sondern reagiert mit nachträglichen Korrekturen auf sie. Dadurch entstehen kleine Positions- und Ausrichtungsschwankungen in allen drei Dimensionen.

Bei dem im Rahmen dieser Projektgruppe eingesetzten Fluggerät handelt es sich um einen Quadrokopter von Parrot mit der AR.Drone, welcher im Folgenden vorgestellt wird. Dabei wird insbesondere auf die von der Drone verwendeten Komponenten und Sensoren eingegangen, die für jeden Quadrokopter notwendig sind.

2.3.4 Parrot AR Drone

Im Jahr 2010 wurde von dem französischen Unternehmen Parrot die Parrot AR.Drone 1.0 vorgestellt. Zwei Jahre später folgte mit der Parrot AR.Drone 2.0 eine nächste, verbesserte Version des ferngesteuerten Quadrokopters. Abbildung 2.13 zeigt die Drone. Tabelle 2.1 liefert zudem eine Übersicht der technischen Daten des Fluggeräts, welche im Anschluss weiter erläutert werden.
Eingangs ist zu erwähnen, dass die AR.Drone 2.0 mittels ihrer Frontkamera HD-Videoaufnahmen ermöglicht, die per WLAN auf ein entferntes Gerät übertragen und dort gespeichert werden können. Eine zweite Kamera, die am unteren Teil des Quadrokopters befestigt ist, dient zur Messung der Grundgeschwindigkeit. Eine Übertragung des Videosignals ist hier ebenfalls möglich. In beiden AR.Drone-Modellen wurde eine 1000mAh-Lithium-Polymer-Batterie mit 11,1

Abbildung 2.13: Parrot AR.Drone 2.0 mit Indoor-Hülle [Red12]

Volt verbaut, sodass eine maximale Flugzeit von etwa 12 Minuten erreicht wird.
Im Bereich der Sensortechnik ist das erste zu nennende Instrument das Gyroskop. Gyroskope sind Kreiselinstrumente, die Rotationen um die Raumachsen erfasst. Durch die Verwendung eines Gyroskops kann die Lage der Drone in der Luft bestimmt werden. Im Falle der AR.Drone 2.0 erfolgte der Einbau eines dreiachsigen Gyroskops, welches die Rotationserfassung um alle drei Raumachsen ermöglicht und somit eine genaue Lageerfassung für die Drone bietet.
Weiterhin wird ein Magnetometer benutzt, um eine Art Kompassfunktion zu realisieren. Durch diese Sensorkombination kann die Lage der Drone noch genauer gemessen werden. Wie schon in dem Modell AR.Drone 1.0 wurde auch in der AR.Drone 2.0 ein dreiachsiger Beschleungigungssensor, auch Accelerometer genannt, verbaut. Dieser Sensor misst die Beschleunigung in Richtung aller drei Raumachsen.

Modell	**AR.Drone 2.0**
Prozessor	ARM-Cortex-A8-Prozessor 1 GHz, digitaler 800 MHz Videoprozessor
RAM	1GB DDR2-RAM (200 MHz)
Kamera (vorne)	92°mit 720p (30 fps)
Kamera (hinten)	QVGA (60 fps)
Batterie	1000-mAh-Lithium-Polymer-Batterie mit 11,1 Volt (3 Zellen)
Flugdauer	$\sim$ 12 Minuten
Gyroskop	dreiachsig
Magnetometer	dreiachsig
Beschl.sensor	dreiachsig
Abstandssensor	Ultraschall 6m, Luftdrucksensor
Anschlüsse	USB
Betriebssystem	Linux
Konnektivität	Wi-Fi b/g/n
Maße (ind.)	52,5 cm $\times$ 51,5 cm
Maße (outd.)	45 cm $\times$ 29 cm
Gewicht (ind.	420 Gramm
Gewicht (outd.)	380 Gramm
Motoren	Brushless-Innenläufer (14,5 Watt / 28.500 1/min)

Tabelle 2.1: Daten der Parrot AR.Drone 2.0 [Par12a]

Zur Abstandsmessung verwendet die Parrot AR.Drone 2.0 einen Ultraschallsensor. Hierbei sendet der Sensor eine Ultraschallwelle aus. Sobald die Welle auf ein Hindernis trifft, wird sie zum Sensor zurückgeworfen. Der Sensor berechnet den Abstand zum Hindernis mit Hilfe der Zeit, die vergangen ist, seitdem die Ultraschallwelle vom Sensor ausgesendet wurde. Der in der AR.Drone verbaute Ultraschallsensor kann durch diese Technik Hindernisse in einem Abstand von bis zu sechs Metern erkennen. Der Ultraschallsensor ist am Unterteil des Quadrokopters verbaut. Die Position des Sensors wurde so gewählt, damit ein sicheres Landen des Quadrokopters möglich ist. Das neue Modell AR.Drone 2.0 bietet zusätzlich zu dem Ultraschallsensor einen Luftdrucksensor, der es ermöglicht, die Flughöhe des Quadrokopters über eine Höhe von sechs Metern hinaus zu bestimmen.
Die Konnektivität der AR.Drone wurde in der zweiten Version des Modells um einen USB-Anschluss erweitert. Über diese Schnittstelle können Videos und Fotos des Quadrokopters di-

rekt auf einem USB-Stick gespeichert werden. Eine drahtlose Verbindung zur Fernsteuerung der AR.Drone erfolgt über WLAN, wobei die Standards 802.11 b, g und n zum Einsatz kommen. Zur Steuerung eröffnet die AR.Drone ein WLAN-Netzwerk, mit welchem sich das Steuerungsendgerät zunächst verbinden muss. Sobald diese Verbindung besteht, kann der Quadrokopter ferngesteuert werden. Die Art des Steuerungsendgerätes variiert. So ist es beispielsweise möglich, die AR.Drone mit der Free-Flight-App [Par12b] auf einem Smartphone zu steuern.
Weiterhin besteht die Möglichkeit die AR.Drone 2.0 über Java-APIs zu steuern. Die Option einer programmatischen Kontrolle mittels Java machte den Quadrokopter zu einer optimalen Basis für Experimente. Wie genau die Steuerung im Zuge der Projektgruppe erfolgte, wird in Sektion 3.2 erläutert.

2.4 Physiksimulation

Bei der Physiksimulation im Computer wird versucht, die Bewegung von Körpern möglichst realitätsgetreu nachzubilden. Im Falle der Simulation eines Quadrokopters kann auf die newtonsche Mechanik[Pal05, p. 21ff] zurückgegriffen werden. In der newtonschen Mechanik wird die Bewegung von Körpern mittels Gleichungen ausgedrückt. Diese Gleichungen werden Bewegungsgleichungen genannt und stellen die Grundlage der Simulation dar. Bei den Bewegungsgleichungen handelt es sich um gewöhnliche Differentialgleichungen, in denen die auf den Quadrokopter wirkenden Kräfte enthalten sind. Für eine möglichst realistische Modellierung des Systems müssen alle auf den Körper wirkenden Kräfte berücksichtigt werden.

Die Simulation mit dem Computer besteht aus den folgenden Schritten:

1. Aufstellen der Differentialgleichungen
2. Anfangs- und Randbedingungen definieren
3. Lösen der Gleichungen

Der 3. Schritt ist der Kern der Simulation. In einem festen Zeitintervall dt müssen die wirkenden Kräfte neu berechnet werden. Die Lösung der Differentialgleichung liefert die Änderung der Position und Geschwindigkeiten des Objekts. Für eine realistische Simulation muss dieser Rechenschritt in sehr kleinen Abständen wiederholt werden. Dieser Schritt kann sehr rechenaufwendig sein. In der Regel ist jedoch die Performance wichtiger als die Exaktheit, sodass man versucht einen Kompromiss zwischen Rechenzeit und Rechengenauigkeit zu finden. Für eine Ingenieurs-Simulation ist Exaktheit wichtiger als für eine Spiele-Simulation. Die hier geplante Simulation des Quadrokopters soll möglichst realistisch wirken, zählt allerdings eher zu der Gruppe der Spiel-Simulationen.

Bei der Bewegungssimulation im Computer müssen zunächst Parameter definiert werden, welche die Position und die Geschwindigkeit des Objekts zu einem Zeitpunkt t definieren. Zur Simulation im dreidimensionalen Raum verwendet man dazu x, y und z-Koordinaten. Die aufgestellte Bewegungsgleichung kann in eine Bewegung in x, y und z-Richtung und eine Rotationsbewegung um die drei Achsen zerlegt werden. Die entsprechenden Parameter für die relative Bewegung werden mit vx, vy und vz bezeichnet. Beide Bewegungen beziehen sich auf den Schwerpunkt des Objekts. Für die Lage des Objekts im Raum gibt man die Winkel zu den drei Achsen an mit α, β , γ.

Zur Lösung der Differentialgleichungen muss die Zeit in diskrete Zeitschritte dt zerlegt werden. Diese Schrittweite hat direkten Einfluss auf die Genauigkeit und damit die Realitätsnähe der

Simulation. Je kleiner der Zeitabstand gewählt ist, desto genauer ist die Simulation. Für eine Spiel-Simulation, wie die geplante Quadrokopter-Simulation, ist keine exakte Genauigkeit erforderlich.

Es existieren mehrere Methoden zur Näherungslösung von Differentialgleichungen: [Pal05, p. 56ff]

- Newton-Methode
- Runge-Kutta
- Leapfrog
- Verlet Integration
- Spezielle Predictor-Corrector-Methoden (wie z.B. Bulirsch-Stoer)

Das Runge-Kutta Verfahren[Pal05, p. 56ff] ist ein einfaches und robustes Verfahren zur Lösung der Gleichungen (solange keine Exaktheit erwartet wird) und wird häufig verwendet.

2.5 Regelungstechnik

Dieses Kapitel bietet eine kurze Einführung in die Regelungstechnik[4]. Die Regelungstechnik befasst sich mit der gezielten Manipulation dynamischer Systeme, also Systeme deren Kenngrößen sich mit der Zeit ändern. Der Unterschied zu einer Steuerung ist der, dass die Ausgangsgröße zurückgeführt wird und mit der Führungsgröße verglichen wird (siehe Abbildung 2.14). Dabei wird das Ziel verfolgt, eine messbare Größe auf einen vorgegebenen Wert zu bringen. Ein Beispiel dazu ist ein Tempomat in einem Auto. Der Fahrer des Autos gibt dem Bordcomputer eine bestimmte Führungsgröße, beispielsweise 80 km/h, vor. Das Auto versucht dann diese Größe zu erreichen und darüber hinaus zu halten. Zusätzlich ist es wichtig mit Störungen umgehen und diese unterdrücken zu können. Bezogen auf das Autobeispiel wäre eine solche Störung ein Hügel. Beim Befahren des Hügels wird mehr Kraft benötigt als auf gerader Strecke und es ist somit eine Anpassung der Stellgröße notwendig.

Die Abbildung 2.14 verdeutlicht den prinzipiellen Aufbau eines Regelkreises. Dort sind zwei Komponenten zu sehen. Die erste Komponente stellt die Regeleinrichtung, Regler genannt, dar. Dieser Regler ist dafür zuständig in Abhängigkeit von der aktuellen Differenz zwischen Führungsgröße und Regelgröße eine Stellgröße zu berechnen. Wie bei einer Steuerung ist die Führungsgröße $w(t)$ eine messbare Größe, die als eine Funktion über die Zeit t beschrieben wird. Die Regelgröße $y(t)$ spiegelt dabei die aktuelle Ausgangsgröße des dynamischen Systems wieder, die auch messbar und eine Funktion über die Zeit ist t. Die Differenz zwischen der Führungsgröße und der Regelgröße wird Regeldifferenz genannt und mit $e(t)$ bezeichnet. Das Ziel ist es nun dass die Regeldifferenz möglichst zu 0 wird und über die Zeit 0 bleibt. Die zweite Komponente ist die Regelstrecke und stellt das zu regelnde dynamische System dar. Auf die Regelstrecke kann noch ein weiterer Einfluss, nämlich eine Störgröße $d(t)$, wirken. Diese Störgröße ist nicht vorhersagbar und der Funktionsverlauf wird in der Regel nicht bekannt sein [Lun10].

Das nächste Beispiel veranschaulicht das Verfahren eines Regelkreises anhand eines Quadrokopters: Ein Quadrokopter befindet sich in einer beliebigen Höhe und soll in einen Hovermodus (ein Modus in dem sich der Quadrokopter stabil in der Luft hält) übergehen. Dieser Hovermodus gibt vor, dass der Quadrokopter eine Höhe von 1m halten soll. Die Führungsgröße entspricht nun also 1m. Die Stellgröße stellt hierbei das Ergebnis der Regelung dar und wirkt direkt auf das System ein. Das Ziel ist es immer, dass die Stellgröße so gewählt wird, dass das System mit Störgrößen umgehen kann. Die Regelgröße wird häufig durch einen Sensor gemessen. Im Falle des Quadrokopter könnte dies durch einen Ultraschallsensor geschehen. Dieser gibt in einer bestimmten Frequenz die aktuelle Höhe des Quadrokopter an. Die Regelgröße

[4]http://www.rn- wissen.de/index.php/Regelungstechnik

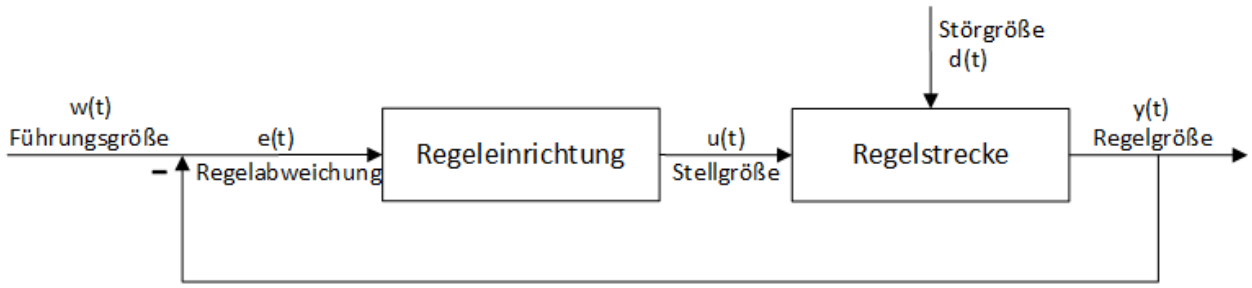


Abbildung 2.14: Aufbau einer Regelstrecke

wird nun zurückgeführt und mit der vorgegebenen Führungsgröße verglichen. Ist der Abstand zur Führungsgröße relativ groß, beispielsweise wenn sich der Quadrokopter in einer Höhe von 3m befindet und die Führungsgröße ein Meter beträgt, so wird die Stellgröße relativ groß ausfallen. Ist die Regelgröße relativ nah an der Führungsgröße, so wird der Effekt, der auf das System wirkt, relativ klein sein. Zusammenfassend soll hier hervorgehoben werden, dass eine Regeleinrichtung drei Hauptbereiche eingeteilt werden kann: [Lun10]

1. Messen

   In der Regel werden Systeme behandelt, die eine zu regelnde messbare Größe haben. Diese Größe kann durch verschiedene Sensoren gemessen werden. Ziel ist dabei, dass das Sensorrauschen so klein wie möglich bleibt, da sonst Fehlinformationen auftreten können.

2. Vergleichen

   In einer bestimmten Frequenz wird die Führungsgröße mit der aktuellen Regelgröße verglichen. Das Ergebnis wird Regelabweichung genannt.

3. Stellen

   Die Regeleinrichtung berechnet in Abhängigkeit zur Regelabweichung die Stellgröße die auf das System gegeben wird. Dadurch wird versucht eine Änderung des dynamischen Systems zu erreichen.

2.5.1 Winkelregler

Die Flugrichtung des Quadrokopters, wird durch die Größen der Winkel β und γ bestimmt. Um den Quadrokopter zu steuern, ist es nötig, einen Algorithmus zu implementieren, mit dem es möglich ist die Winkel β und γ im Flug vorzugeben. Das Ziel des Algorithmus ist es, dass der Quadrokopter diese Flugwinkel erreicht und darüber hinaus auch beibehält. Zusätzlich gibt es

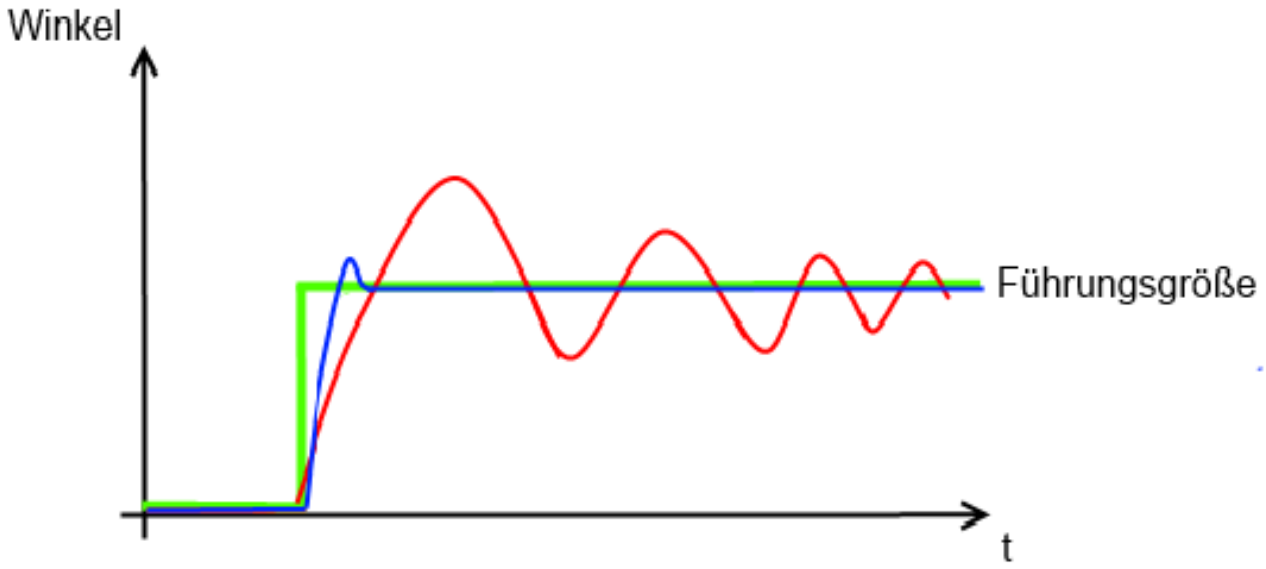


Abbildung 2.15: Beispielhaftes Führungsverhalten eines Regelkreises. Die rote Linie deutet dabei ein schlechtes Führungsverhalten an. Die blaue Linie zeigt ein gewünschtes Führungsverhalten auf

einige Bedingungen die vom Algorithmus erfüllt werden müssen. Dazu zählt, dass ein zu weites Überschwingen unerwünscht ist. Ein Überschwingen ist ein Begriff aus der Regelungstechnik und bedeutet, dass ein Zielwert nicht um einen bestimmten Grad verfehlt werden darf. In Abbildung 2.15 ist ein solches Verhalten beispielhaft dargestellt. Die grüne Linie deutet an, dass nach einer bestimmten Zeit eine Winkelgröße vorgegeben wird. Dabei zeigt die rot dargestellte Linie einen unerwünschten Verlauf dar. Bei diesem Verlauf ist ein deutliches Überschwingen zu erkennen. Zusätzlich braucht der Quadrokopter eine gewisse Zeit, bis er sich der Führungsgröße nähert. In der Praxis würde der Quadrokopter mit einem solchen Verhalten stark anfangen in der Luft zu wackeln. Ein gewünschtes Verhalten hingegen, stellt die blaue Linie dar. Bei diesem Verlauf ist zwar ein leichtes Überschwingen zu erkennen, dieses stört aber in der Praxis nicht. Wichtig ist nur, dass das System nicht anfängt zu schwingen, denn dann wäre ein sicheres Fliegen nicht möglich (siehe rote Linie). Zu beobachten ist ferner, dass dieser Verlauf die geforderte Führungsgröße zeitnah erreicht, und darüber hinaus hält.

Um einen Winkelregler zu realisieren, wird ein PID - Regler genutzt und implementiert [Lun10]. PID - Regler steht für proportional, integral und differential - Regler. Wie im Unterkapitel *Einführung in die Regelungstechnik* beschrieben, werden auch hier die drei Hauptbereiche, nämlich das Messen, Vergleichen und das Stellen genutzt. Es wird zunächst der aktuelle Winkel α gemessen. Dieser Winkel wird in jedem Zyklus mit dem Sollwert α_{soll} verglichen und nachfolgend

als Regelfehler e mit $e = \alpha - \alpha_{soll}$ bezeichnet. Die Stellgröße y stellt die Kraft dar und wird mit der nachfolgenden PID - Gleichung bestimmt:

$$y = K_p * e + K_i * e_{sum} + K_d * \frac{e - e_{old}}{T}$$

Der Term $K_p * e$ stellt dabei den proportionalen Anteil des Reglers dar. Dieser Teil berechnet, in Abhängigkeit von der Regeldifferenz e, die Stellgröße y. Je größer also der Abstand zwischen dem gemessenen Wert und dem Sollwert ist, desto größer wird die Stellgröße ausfallen. Der Wert K_p gibt an, um welchen Faktor die Regeldifferenz multipliziert wird. Dieser Wert wird durch empirisches Testen ermittelt. Ganz allgemein bedeutet das empirischen Testen, dass ein zufälliger Wert für K_p gesucht und das Verhalten des Reglers überprüft wird. Ist der Regler zu aggressiv und zeigt ein deutliches Überschwingen, so wird der Wert verringert. Sollte der Regler zu passiv arbeiten, d.h die Stellgröße hat keine Auswirkung auf den Quadrokopter, so muss K_p erhöht werden. Zusätzlich stellt der Term $K_i * e_{sum}$ den Integralen-Anteil der Gleichung dar. Bei diesem Term wird e_{sum} mit $e_{sum} = e_{sum} + e$ stetig wachsen. Denn der Regelfehler e wird in einem weiteren Schritt stetig auf eine Variable e_{sum} hinzu addiert. Dies bezweckt, dass e_{sum} irgendwann so groß wird, dass die Stellgröße immer weiter wächst und die Führungsgröße irgendwann erreicht wird. Zu beachten ist, dass $e_{sum} = 0$ gesetzt wird, sobald die Regelgröße gleich dem Sollwert ist. Wäre dies nicht so, so würde im nächsten Zyklus die Regelgröße nicht korrekt berechnet werden. Der Wert K_i wurde auch in diesem Term durch empirisches Testen bestimmt. Der verbleibende Term $K_d * \frac{e - e_{old}}{T}$ stellt den Differentialen-Anteil dar. Bei diesem Term wird der Regelfehler aus dem vorherigen Zyklus mit eingerechnet. Die Variable T stellt die Abtastrate dar und berechnet sich aus der zeitlichen Differenz zwischen zwei Zyklen. Anschaulich bedeutet der D-Anteil, dass wenn der Unterschied vom aktuellen Regelfehler zum vorherigen Regelfehler sehr groß ist, so wird der Term $\frac{e - e_{old}}{T}$ auch sehr groß. Praktisch ausgedrückt bedeutet dies, dass wenn der Winkel zwischen zwei Zyklen sich sehr stark ändert und die Abtastrate dabei sehr gering ist, die Stellgröße dementsprechend groß ausfallen wird. Der D-Anteil bezweckt, dass die Führungsgröße schnell erreicht wird.

Durch dieses Verfahren ist es möglich geworden die Kräfte, die auf die vier Propeller wirken, so zu berechnen, dass der Quadrokopter einen vorgegeben Flugwinkel erreicht und hält. In vielen Experimenten, die hier nicht näher erläutert werden sollen, wurde dies geprüft und für korrekt befunden. Da aber der Quadrokopter bei gleichbleibendem Flugwinkel jedoch immer weiter beschleunigt und somit die Geschwindigkeit immer weiter zunimmt, ist ein Geschwindigkeitsregler vonnöten.

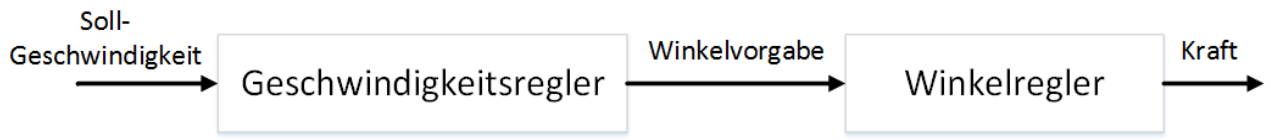


Abbildung 2.16: Wirkung des Geschwindigkeitsreglers auf den Winkelregler. Der Geschwindigkeitsregler gibt eine Geschwindigkeit vor, die der Winkelregler erfüllen muss

2.5.2 Geschwindigkeitsregler

Die Führungsgrößen bei diesem Regler sind die Geschwindigkeiten v_x entlang der x- und v_y entlang der y-Achse und werden vom Benutzer vorgegeben. Ziel bei diesem Regler ist es, die Stellgrößen β und γ zu berechnen, so dass der Quadrokopter sich den Eingaben des Benutzers entsprechend verhält. Auch hier wird zur Umsetzung des Geschwindigkeitsreglers ein PID-Regler (siehe Winkelregler) verwendet. Nach der Implementierung des Regel-Algorithmus, ist es wie schon zuvor die Aufgabe, die Konstanten $K_p, K_i und K_d$ zu bestimmen. Dabei wurde das empirische Testen angewendet und die Werte dieser Konstanten experimentell bestimmt.

In Abbildung 2.16 wird die Wirkung des Geschwindigkeitsreglers auf den Winkelregler deutlich. Der Benutzer gibt dem Geschwindigkeitsregler eine Vorgabe. Der Regler berechnet dann als Stellgröße einen Winkel, der als Führungsgröße für den Winkelregler dient. Dieser berechnet wiederum eine Kraft als Stellgröße, die auf jeweils einen der Propeller wirkt.

Zusätzlich ist festzuhalten, dass der Geschwindigkeitsregler dazu dient, den Quadrokopter auf der Stelle schweben zu lassen. Dieses ist der Fall, wenn der Benutzer keine Eingaben tätigt und die Führungsgrößen somit gleich $v_x = 0$ und $v_y = 0$ sind. Der Regler regelt dann automatisch hinsichtlich dieser Geschwindigkeit, was zum Effekt hat, dass der Quadrokopter auf der Stelle stehen bleibt.

2.5.3 Höhenregler

Ein weiterer Regler ist der Höhenregler. Dieser dient dazu dem Benutzer die Möglichkeit zu bieten, die aktuelle Höhe des Quadrokotpers zu beeinflussen. Auch hier wird als Regelalgorithmus ein PID-Regler genutzt, dessen Regelkonstanten K_p, K_i und K_d durch empirisches Testen ermittelt wurden. Als Führungsgröße dient die Geschwindigkeit v_z entlang der z-Achse (siehe Eulerwinkel). Der Höhenregler berechnet als Stellgröße eine Kraft. Diese Kraft wird in Abhängigkeit der Regeldifferenz erhöht bzw. erniedrigt und wirkt auf die Propeller. Dadurch kann der Benutzer die aktuelle Höhe des Quadrokopters ändern.

2.5.4 Rotationsregler

Damit der Benutzer den Quadrokopter hinsichtlich des Winkels α steuern kann, wurde ein Rotationsregler implementiert. Als Führungsgröße dient hier der Winkel α, der von Benutzer vorgegeben wird. Die Stellgröße ist auch hier wieder eine Kraft. Zur Umsetzung wird ein PID-Regler genutzt, dessen Regelkonstanten K_p, K_i und K_d durch empirisches Testen ermittelt wurden. Der Benutzer ist nunmehr dazu in der Lage, durch eine Steuereinheit Vorgaben zu tätigen, so dass der Quadrokopter sich um die eigene Achse α dreht.

2.5.5 Fazit

Nach der Implementierung der vier Regler, ist der Benutzer dazu in der Lage, die Geschwindigkeiten v_x, v_y, v_z, sowie den Winkel α vorzugeben. Dies ermöglicht es ihm, den Quadrokopter entlang allen drei Achsen x, y und z zu steuern. Zusammenfassend bedeutet die Implementierung der Regler, dass die Kräfte die auf die Rotoren wirken, in jeder Iteration des Programms neu berechnet und in einer Gleichung eingefügt werden. Diese berechneten Gleichungen werden von V-REP interpretiert und ausgeführt und der Quadrokopter damit in Bewegung versetzt. In diversen Experimenten wurde festgestellt, dass das Flugverhalten zufriedenstellend stabil und die eine gute Steuerbarkeit durch einen Benutzer damit gegeben ist.

KAPITEL 3

Architektur

In diesem Kapitel wird ein Überblick über das Zusammenspiel der einzelnen Komponenten gegeben. Dazu sei zunächst das Architekturdiagramm betrachtet (Abbildung 3.1).

Im Mittelpunkt der Software stehen die Steuergraphen. Die Ausführung des Steuergraphen wird über jABC realisiert. Über die Benutzerschnittstelle, welche mit der *Lightweight Java Gaming Library* (LWJGL)[LWJ14] realisiert wurde, wird der Steuergraph mit Eingaben gefüttert. Die Eingaben werden dann an die Quadrokopter-Schnittstelle weitergeleitet, welche wiederum an eine konkrete Instanz weiter delegiert (Simulation oder tatsächlicher Quadrokopter).

Neben diesen für den wesentlichen Workflow notwendigen Komponenten gibt es noch zwei weitere Elemente im Architekturdiagram: Zum einen der Aspektweber, der das Einweben von Sicherheitsaspekten auf Modellebene realisiert (Model-to-Model-Transformation). Zum anderen wurden Missionsgraphen entwickelt, um dem Quadrokopter gewisse autonome Flugmanöver zu ermöglichen.

In den folgenden Abschnitten werden nun die einzelnen Komponenten näher erläutert.

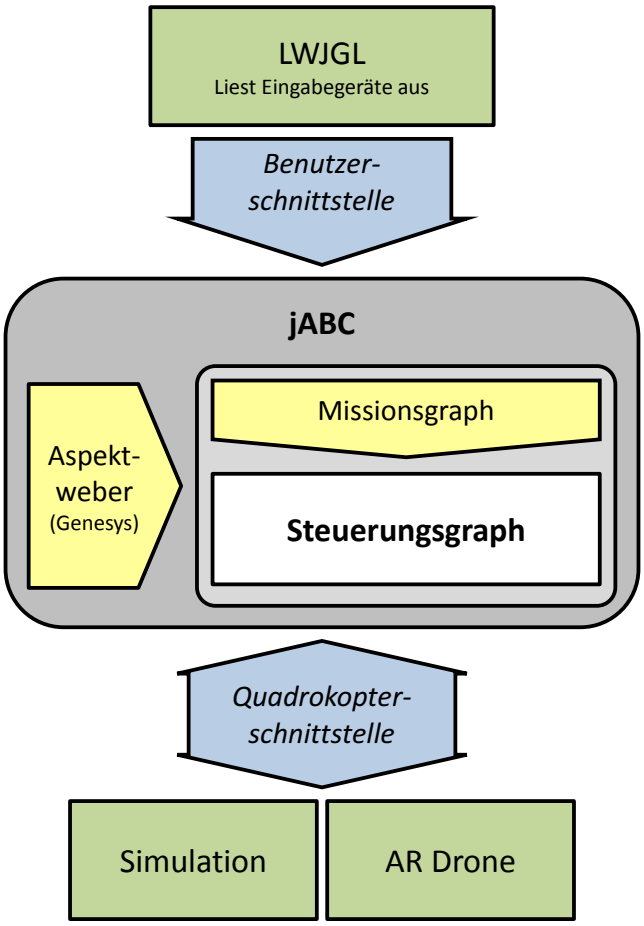


Abbildung 3.1: Übersicht der Architektur der AgES-Software

3.1 Eingabeschnittstelle

Um mehrere verschiedene Eingabegeräte zu ermöglichen, wurde ein Interface entworfen, welches alle notwendigen Methoden eines Eingabgeräts definiert:

Methode	Beschreibung	Wertebereich
`int getFrontBackTilt()`	Neigung um die Querachse	$[-100, 100]$, wobei 100 der maximalen Neigung nach vorne entspricht.
`int getLeftRightTilt()`	Neigung um die Längsachse	$[-100, 100]$, wobei 100 der maximalen Neigung nach rechts entspricht.
`int getVerticalSpeed()`	Rotorengeschwindigkeit zur Höhenveränderung	$[-100, 100]$, wobei 100 der maximalen Geschwindigkeit aufwärts entspricht.
`int getAngularSpeed()`	Drehgeschwindigkeit um die Gierachse	$[-100, 100]$, wobei 100 der maximalen Drehgeschwindigkeit nach rechts entspricht.
`boolean isCancelled()`	Abbruchsignal gesendet?	`true` falls gesendet, sonst `false`
`boolean isTakenOff()`	Abhebesignal gesendet?	`true` falls gesendet, sonst `false`
`boolean isLanded()`	Landesignal gesendet?	`true` falls gesendet, sonst `false`
`boolean isReseted()`	Rücksetzsignal?	`true` falls gesendet, sonst `false`
`void cleanUp()`	Aufräumarbeiten	—
`String getName()`	Namen des Eingabegeräts	—

Tabelle 3.1: Controller Interface Methoden

Dabei werden über die Methoden `getFrontBackTilt()`, `getLeftRightTilt()`, `getVerticalSpeed()` und `getAngularSpeed()` die aktuell vom Benutzer eingegebenen Bewegungsrichtungen ausgelesen. Die Methoden `isCancelled()`, `isTakenOff()`, `isLanded()` und `isReseted()` geben einen boolschen Wert zurück, ob ein Signal empfangen wurde, um die Drohne in den entsprechenden Zustand zu überführen (z.b. Abheben, Landen, etc.). Die Rückgabewerte wer-

den dann an die entsprechenden Methoden der Quadrokopter-Schnittstelle weitergeleitet (siehe Abschnitt 3.2).

Im Rahmen der Projektgruppe wurde das Interface für drei verschiedene Arten von Eingabegeräten implementiert: Eine Computertastatur, ein XBox 360-Controller und ein Playstation 3 Controller. Auf die konkreten Implementierungen wird in Abschnitt 4.1 näher eingegangen.

3.2 Quadrokopter-Schnittstelle

Die Quadrokopter-Schnittstelle dient zur Abstraktion der Steuerung des Quadrokopters. Dies ist notwendig, um das Steuerungsmodell sowohl auf die Parrot ARDrone als auch auf den simulierten Quadrokopter anwenden zu können. Die Schnittstelle ist als Java-Interface realisiert. Sie stellt alle benötigten Methoden zur Steuerung des Quadrokopters bereit. Außerdem bietet die Schnittstelle Zugriff auf Sensorwerte, wie Abstände oder den Batteriestatus. Die zentrale Methode zur Steuerung des Quadrokopers ist die `move`-Methode. Der Methode werden insgesamt vier Parameter übergeben. Der Wertebereich der Parameter ist $[-100, 100]$. Die beiden Parameter `frontBackTilt` und `leftRightTilt` bestimmen die Lage des Quadrokopters um die Längs- und Querachse. Mit einem Wert `frontBackTilt=100` soll der Quadrokopter die höchstmögliche Neigung nach vorn einnehmen, was einem horizontalen Vorwärtsflug entspricht. Die jeweilige Implementierung der Quadrokopter-Schnittstelle muss dafür sorgen, dass die Neigung auf einen bestimmten Winkel begrenzt wird, der 100% entspricht. Ein negativer Wert `frontBackTilt=-100` bewirkt eine Neigung des Quadrokopters nach hinten und somit einen horizontalen Rückwärtsflug. Als weitere Parameter werden der `move`-Methode die angestrebte Rotation um die Hochachse `angularSpeed`, sowie die angestrebte vertikale Geschwindigkeit `verticalSpeed` übergeben. In der Tabelle 3.2 werden alle über die Schnittstelle bereitgestellten Methoden beschrieben.

Methode	Beschreibung
`move(leftRightTilt, frontBackTilt, angularSpeed, verticalSpeed)`	Zentrale Methode zur Steuerung des Quadrokopters
`getDistanceFront()`	Gibt die Distanz vorn zurück
`getDistanceLeft()`	Gibt die Distanz links zurück
`getDistanceRight()`	Gibt die Distanz rechts zurück
`getDistanceBack()`	Gibt die Distanz hinten zurück
`getDistanceTop()`	Gibt die Distanz nach oben zurück
`getBattery()`	Gibt den Batteriestatus in Prozent zurück
`getAltitude()`	Gibt die Distanz nach unten zurück
`connect()`	Baut die Verbindung zum Quadrokopter auf
`disconnect()`	Beendet die Verbindung zum Quadrokopter
`takeoff()`	Lässt den Quadrokopter senkrecht abheben und anschließend in einer Höhe von 1m schweben
`hover()`	Schweben in der aktuellen Position
`land()`	Löst einen senkrechten Sinkflug bis zu einer Höhe von 30cm aus und schaltet dann die Rotoren aus, um zu landen
`getOrientation()`	Gibt die Orientierung um die Hochachse zurück
`getPosX()`	Gibt die x-Position im Raum zurück
`getPosY()`	Gibt die y-Position im Raum zurück
`getPosZ()`	Gibt die z-Position im Raum zurück
`getVelocityX()`	Gibt die Geschwindigkeit in x-Richtung zurück
`getVelocityY()`	Gibt die Geschwindigkeit in y-Richtung zurück
`getVelocityZ()`	Gibt die Geschwindigkeit in z-Richtung zurück

Tabelle 3.2: Methoden des Quadrokopter-Interfaces

3.3 Steuergraphen

Um eine Drohne mithilfe eines Eingabegeräts zu steuern, muss in kurzen Intervallen die Eingabe abgefragt und an die Drohne gesendet werden. Das Hauptelement des Steuergraphen besteht also aus einer sich wiederholenden Schleife aus Eingabe abfragen und Eingabe verarbeiten. Vorher müssen gegebenfalls noch Initialisierungen getroffen werden. Ein abstraktes Steuerungsmodell könnte also wie folgt aussehen:

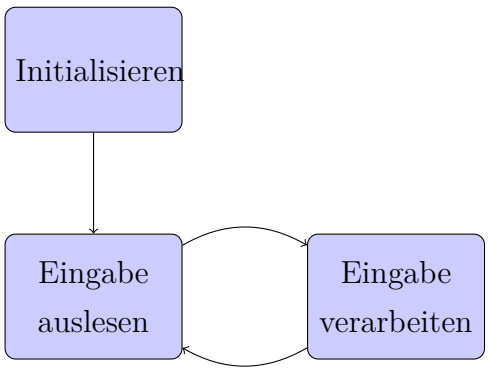


Abbildung 3.2: Abstrakter Steuerungsgraph

Eingabe auslesen und Eingabe verarbeiten sind dabei nur abstrakte Platzhalter für komplexe Implementierungen. Wichtig ist, dass beide Aktivitäten eine sehr kurze Laufzeit haben, da für eine (gefühlt) latenzfreie Steuerung der Drohne eine Zyklusdauer von $\leq$ 20ms wünschenswert ist.

Dieser Steuerungsgraph dient als Basis für die Modellierung eines konkreten Steuerungsmodells in jABC. In der konkreten Modellierung werden dann Initialisieren, Eingabe auslesen und Eingabe verarbeiten durch geeigenete SIBs oder Untermodelle ersetzt. Dabei bieten die in 3.1 und 3.2 vorgestellten Schnittstellen essentielle Methoden für die Realisierung der Aktivitäten.

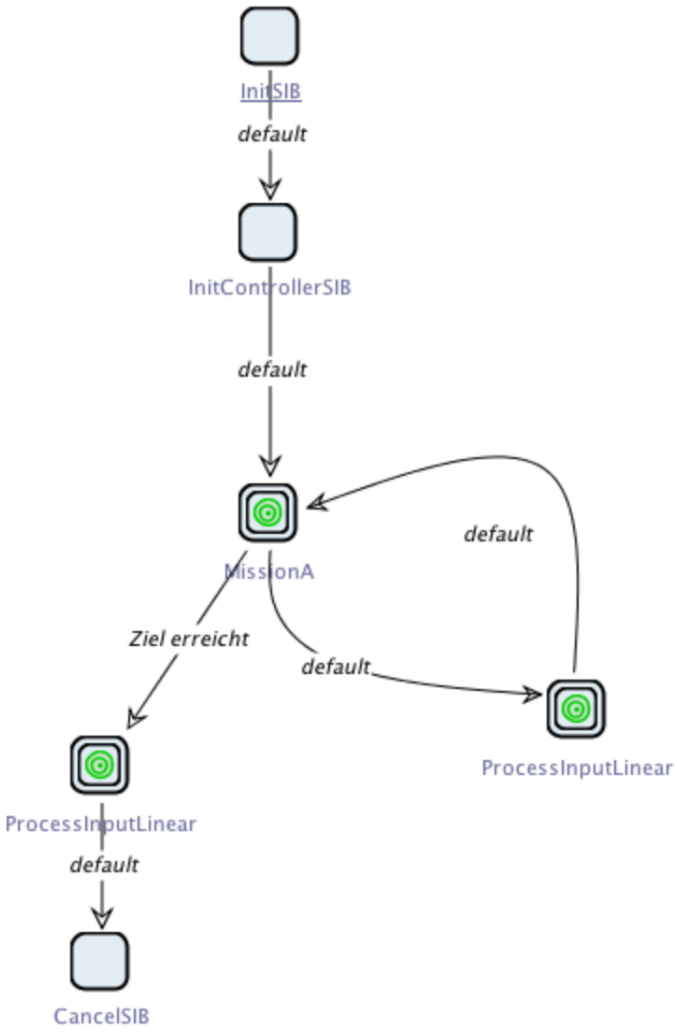


Abbildung 3.3: Einbinden der Missionsgraphen in das Steuerungsmodell

3.4 Missionsgraphen

Missionsgraphen werden eine Alternative zur manuellen Steuerung darstellen. Ein solcher Graph beschreibt eine Mission des Quadrokopters, beispielsweise die Erkundung eines Labyrinths. Der Graph nutzt die Steuergraphen, um das Fluggerät zu lenken.

Das Missionsmodell wird, wie in Abb. 3.3 zu sehen ist, als automatisierter Controller realisiert. Es ersetzt demnach im Modell das *ReadControllerInputSIB*. Der Missionsgraph schreibt wie der Controller Steuersignale in den Kontext, die im SIB *ProcessInputLinear* verarbeitet werden.

3.5 Aspektweber

Der Aspektweber ist ein Werkzeug, welches im Kontext des Ziels der Projektgruppe das Steuerungsmodell des Quadrokopters mit Steuerungsoptionen anreichert, so dass ein erweitertes Steuerungsmodell entsteht. Dieses erweiterte Steuerungsmodell enthält nach der Verwebung durch den Aspektweber zusätzliche Funktionen, wie *Notlandung* und *Sicheres Fliegen* (der Quadrokopter kann nicht gegen ein Hindernis geflogen werden).

Diese Funktionen sind möglichst modular spezifiziert, wie es in der apektorientierten Programmierung (AOP) üblich ist und werden in Aspekten definiert. Das Einweben eines Aspekts in das Steuerungsmodell durch den Aspektweber generiert ein neues Steuerungsmodell, welches die gewünschten Aspekte enthält. Realisiert wird dies durch Modell-Transformationen von Steuerungsmodellen, wie in Abbildung 3.4 dargestellt.

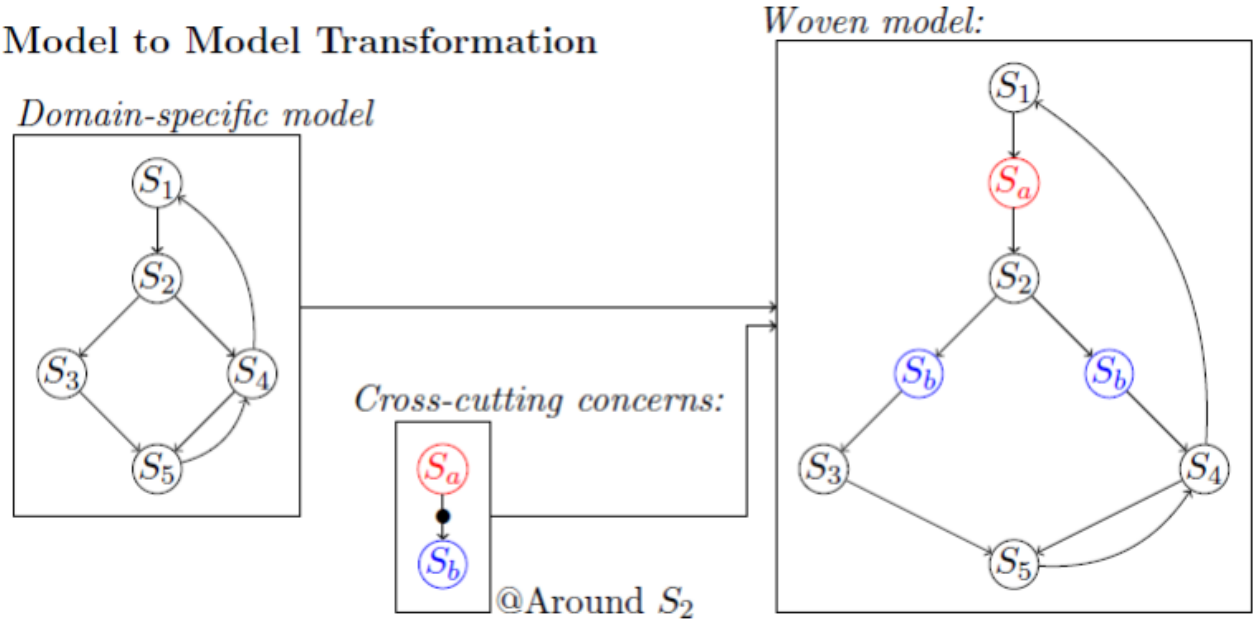


Abbildung 3.4: Einweben eines Aspekts/querschneidenen Belangs (crosscutting concern) in ein Steuerungsmodell.

KAPITEL 4

Benutzerschnittstelle

In diesem Kapitel wird beschrieben, wie die Interaktion von Nutzer und Software realisiert wurde. Es wird dabei in Abschnitt 4.1 speziell auf die dafür benutzte Bibliothek eingegangen, sowie die bei der Entwicklung aufgetretenen Probleme und Hindernisse.

4.1 LWJGL

Das Auslesen der Informationen von unterschiedlichen Eingabegeräten war eine der frühen Herausforderungen im Rahmen der Projektgruppe. Von Beginn an stand fest, dass neben einer Tastatur auch Gamepads als Eingabegeräte unterstützt werden sollten. Dabei wurde sich auf den Microsoft XBox 360 und den Playstation 3 Controller geeignet, im wesentlichen wegen ihrer hohen Verfügbarkeit. Einstiegspunkt war dabei der Xbox 360 Controller, da dieser nativ kompatibel mit modernen Mircosoft Windows Betriebssystemen ist. Dies ersparte die Installation von Drittanbieter-Software bei den Windows-Nutzern.

Bevor die Wahl auf die *Lightweight Java Gaming Library* (LWJGL)[LWJ14] fiel, wurden einige andere Bibliotheken untersucht. Da die LWJGL jedoch sehr gut dokumentiert und in aktiver Nutzung bei Spieleentwicklungen ist, wurde sich für diese entschieden. Weitere Vorteile sind ein vorhandenes Maven-Repository für diese Bibliothek sowie die leichtere Realisierung von Plattformunabhängigkeit im Vergleich mit anderen Bibliotheken.

Einmal instanziiert, erkennt das LWJGL Framework Tastatur und angeschlossene Controller. Voraussetzung dafür ist jedoch, dass die korrekten Treiber für den entsprechenden Controller installiert sind. Dies gestaltete sich besonders bei den Teilnehmern mit Linux-basierten oder Macintosh Betriebssystemen als Herausforderung. Sobald die Treiber jedoch korrekt installiert sind und der Controller von der LWJGL erkannt wird (die Identifizierung der Controller geschieht dabei über den Namen) stellt die Bibliothek eine Reihe nützlicher Funktionen zum Auslesen der Eingabe zur Verfügung.

An dieser Stelle ein kurzer Ausschnitt wie mithilfe der LWJGL das in Abschnitt 3.1 vorgestellte Controller-Interface implementiert werden kann:

```
1  public class XboxController implements edu.udo.pg.ages.controller.IController {
2      org.lwjgl.input.Controller controller;
3      [...]
4      @Override
5      public int getFrontBackTilt() {
6          update();
7          return applyThreshold((int) (-controller.getYAxisValue() * 100));
8      }
9      [...]
10 }
```

In dem Codeausschnitt findet sich in Zeile 7 der Aufruf `controller.getYAxisValue()`. Diese beispielhafte Methode wird von der LWJGL zur Verfügung gestellt und liefert die Position des linken Joysticks. Der Rückgabewert ist eine Gleitkommazahl im Bereich von $[-1, 1]$, wobei

-1 für vollen Ausschlag nach „oben“ und 1 für vollen Ausschlag nach „unten“ steht. Da das Interface als Rückgabewert einen Integer zwischen $[-100, 100]$ erwartet, muss vor der Rückgabe noch entsprechend umgewandelt werden.

Auf ähnliche Weise wurden so die Methoden des Controller-Interfaces für Keyboard, Xbox 360 Controller und Playstation 3 Controller implementiert. Die Architektur der AgES Software erlaubt es, nachträglich weitere Eingabegeräte hinzuzufügen. Dazu muss lediglich eine neue Controller-Implementierung geschrieben werden. Darüber hinaus muss das Eingabegerät in dem SIB zur Initialisierung des Eingabegeräts registriert werden, dies ist näher unter Abschnitt 7.1 beschrieben.

KAPITEL 5

Kommunikation mit dem Quadrokopter

Die Kommunikation mit dem Quadrokopter soll innerhalb dieser Projektgruppe sehr variabel bleiben. Das bedeutet, dass keine konkrete Hardware direkt angesteuert wird, sondern ein Interface zur Abstraktion genutzt wird. Durch die Nutzung eines Interfaces wird die Implementierung der Kommunikationsadapter austauschbar, sodass verschiedene Hard- oder Software angesprochen werden kann. Das bedeutet in diesem Fall ganz konkret, dass neben einer Implementierung für die Parrot AR.Drone auch eine Implementierung für eine Simulationsumgebung existiert.

Das Diagramm aus Abbildung 5.1 zeigt das Interface und die verschiedene Implementierungen. Der ARDroneAdapter dient zur Kommunikation mit der Parrot AR.Drone. In der Klasse Simulation wird die in diesem Projekt eingesetzte Simulationsumgebung VREP angesprochen. Da zu Beginn der Projektgruppe der Eigenbau eines Quadrokopters nicht ausgeschlossen wurde, soll die Klasse Eigenbau eine Kommunikation mit eben diesem ermöglichen. Zu Debuggingzwecken wurde weiterhin die Klasse PrintDrone entwickelt, die lediglich Textausgaben generiert, die das aktuelle Steuerungskommando darstellen.

Ein Schwerpunkt der Projektgruppe stellt zunächst der ARDroneAdapter dar. Grundlagen zur Entwicklung dieses Adapters werden in Kapitel 5.1 vorgestellt. Im weiteren Verlauf bildete sich mit der Simulation, s. Kapitel 5.2, der zweite Schwerpunkt.

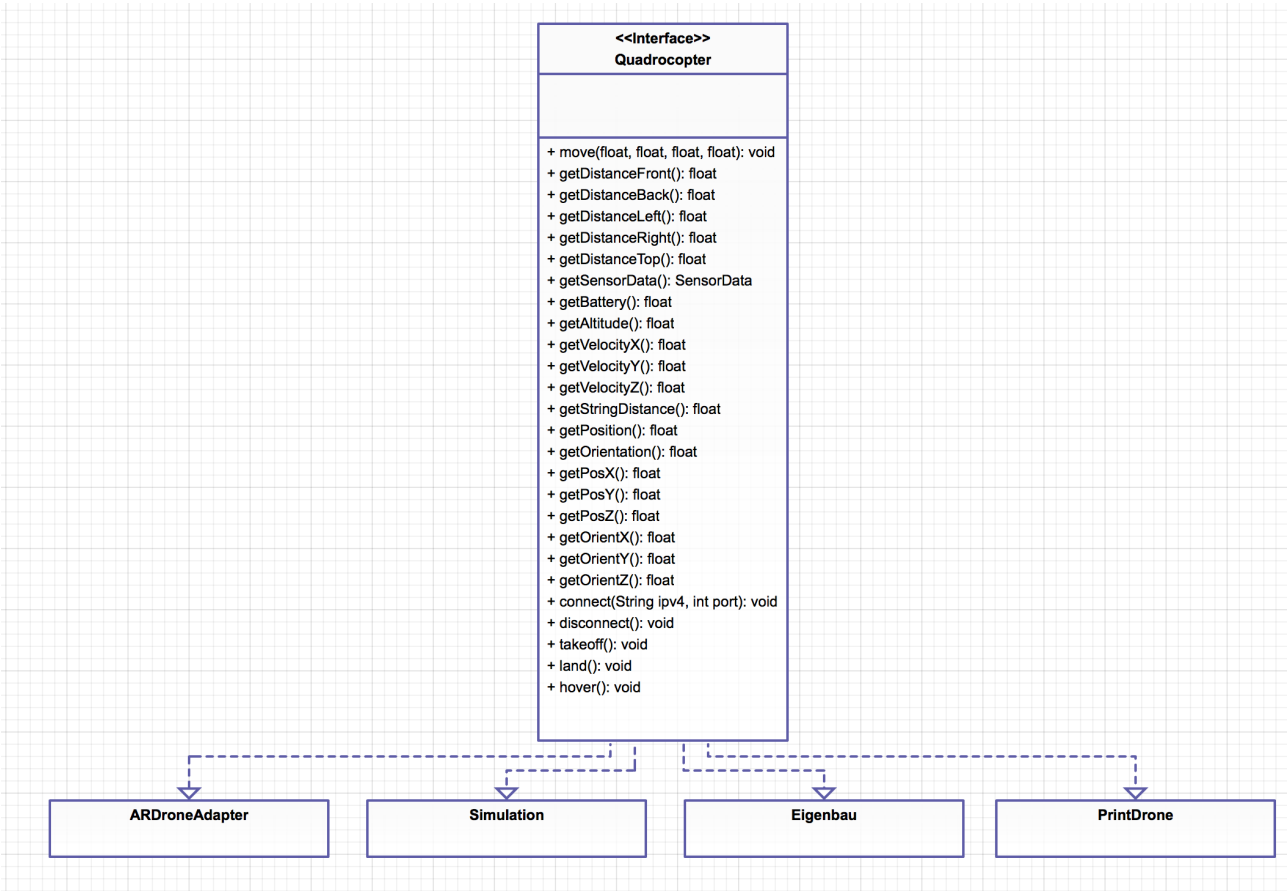


Abbildung 5.1: Abstraktion durch *Quadrocopter* Interface

5.1 Parrot AR.Drone

Zur Steuerung des Quadrokopters wurden bereits verschiedene Java-APIs entwickelt. Eine Auswahl dieser Entwicklungen wird im Folgenden kurz vorgestellt. Die ARDroneForP5-API[Yos11] sowie die javadrone-API[Shm13] stellen hierbei die Hauptentwicklungen dar. Weiterhin existiert die easydrone-API[Man11], welche allerdings lediglich eine Vereinfachung auf Basis des javadrone-Projekts liefert und daher im weiteren Verlauf nicht weiter beachtet wird. Letztlich lässt sich mit der YADrone-API[Bad14] arbeiten, die versucht auf Basis der ARDroneForP5-API die Vorzüge dieser und der javadrone-API zu vereinen. Keine der APIs stellte sich für das Projekt als optimal heraus, sodass ähnlich der YADrone-API eine Eigenentwicklung auf Basis der anderen APIs stattfand.
Die Art der Steuerung ist für jede der Implementierungen ähnlich. Die Steuerungskommandos, die an den Quadrokopter gesendet werden sollen, werden in eine Commandqueue gelegt. Aus dieser Commandqueue wird in einem festgelegten Intervall ein Kommando entfernt und an die AR.Drone gesendet. Um eine kontinuierliche Bewegung zu gewährleistet wird dieses Intervall auf etwa 200ms gesetzt. Das bedeutet, dass ein Steuerungskommando, das an den Quadrokopter geschickt wird, lediglich eine kurze Auswirkung auf den Quadrokopter hat. Um den Quadrokopter beispielsweise zehn Sekunden lang kontinuierlich vorwärts zu bewegen, muss das Vorwärts-Kommando für zehn Sekunden alle 200ms an den Quadrokopter geschickt werden. Der Quadrokopter empfängt das Vorwärts-Kommando in diesen zehn Sekunden demnach 50 Mal. Um den Quadrokopter erfolgreich per Java-API zu steuern muss das Steuerungsendgerät mit dem Hotspot des Quadrokopters verbunden sein. Das Steuerungsendgerät sei in diesem Fall ein Laptop.

ARDroneForP5

Die ARDroneForP5-API bietet komplexe Möglichkeiten für den Zugriff auf die Parrot AR.Drone 2.0. Es können neben Navigationsdaten auch Kameradaten abgefragt werden. Auch zusätzliche Systeminformationen wie den Batteriestand lassen sich über diese API erfragen. Der für diese PG relevanteste Teil der API ist die Steuerung des Quadrokopters. Bei der ARDroneForP5-API hat man sich für folgendes Konzept zur Bewegungssteuerung entschieden:

Es existiert eine *move3d*-Methode, die Geschwindigkeiten für alle Achsen im Raum übergeben bekommt. Diese werden dann gesammelt in einem Kommando an die Parrot AR.Drone geschickt. Dieses Konzept zur Steuerung ist speziell angepasst an das zeitkritische System des Quadrokopters. Es müssen keine zwei Kommandos an den Quadrokopter geschickt werden um

Geschwindigkeiten für verschiedene Achsen zu realisieren. Somit ist eine Bewegung wie *diagonal fliegen* leichter umsetzbar.

Diese API ist sehr gut an das neue Modell der Parrot AR.Drone angepasst worden. Allerdings konzentrierten sich die Entwickler auf eine funktionsfähige Bewegungssteuerung, sowie die mediale Unterstützung für das Kamerabild. Im Gegensatz dazu lässt die im nächsten Kapitel vorgestellte API Zugriff auf komplexe Navigationsdaten zu.

javadrone

Das Konzept der ARDroneForP5 API ist sehr gut angepasst an das Modell Parrot AR.Drone 2.0 und kann daher sämtliche Navigationsdaten auslesen. Das ist ein großer Vorteil gegenüber vielen anderen APIs, die weniger oder gar keinen Zugriff auf die Navigationsdaten zur Verfügung stellen. Nicht nur Informationen zur Geschwindigkeit auf den verschiedenen Achsen oder zur Flughöhe werden geliefert. Es stehen auch so genannte *Vision-Tags* zur Verfügung. Diese Vision-Tags (s. Abb. 5.2) dienen der Umgebungsanalyse. Sie werden beliebig im Raum befestigt und werden dank des Kamerabildes und der internen AR.Drone Software erkannt und analysiert. Mit Hilfe der javadrone-API kann ein Entwickler Informationen wie den Abstand zu einem VisionTag erhalten und für Zwecke der Umgebungsanalyse nutzen.

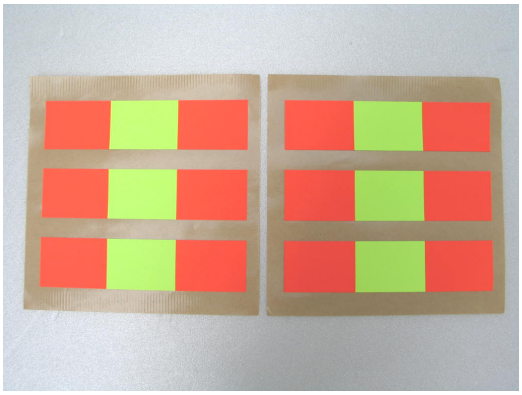


Abbildung 5.2: Beispiel eines Vision-Tags [AUS14]

Eigenentwicklung angelehnt an YADrone

Erste Tests mit der AR.Drone wurden innerhalb der Projektgruppe mit den oberhalb erläuterten APIs durchgeführt. Dabei stellten sich diverse Probleme heraus.
Bei der Nutzung der javadrone-API, welche ursprünglich für die erste Version der AR.Drone entwickelt wurde, kam es zu Kompatibilitäts-Problemen mit der innerhalb der Projektgruppe

```
1  public class ARDroneConfiguration {
2
3      private int detectType = 10,
4                  detectionsSelectH = 7,
5                  detectionsSelectV = 7,
6                  enemyWithoutShell = 1,
7                  enemyColors = 1;
8
9      private boolean outdoor = true;
10
11     ...
12 }
```

Listing 5.1: ARDroneConfiguration

verwendeten Version zwei. Diese zeigten sich in generellen Verbindungs-Problemen und Problemen beim Auslesen der Video-Streams.
Die ARDroneForP5-API hingegen zeigte in diesen Bereichen überzeugende Eigenschaften. Dafür existierten bei der Nutzung dieser Schnittstelle Probleme beim Auslesen der bereits in 5.1 erwähnten Vision-Tags, welche im Zuge des Projekts für die Abstandserkennung eine wichtige Rolle spielen. Nähere Informationen zum genauen Einsatz finden sich in Abschnitt **??**.
Diese Feststellungen führten dazu, dass es nötig erschien, zur ausreichenden Ansteuerung des Quadrokopters eine Eigenentwicklung zu kreieren, welche die Vorzüge beider vorgestellten APIs vereint. Recherchen sorgten an dieser Stelle für die Erkenntnis, dass diese Entwicklung mit der YADrone-API[Bad14] in genau dieser Form bereits an der Universität Hamburg erfolgt war. Daraufhin durchgeführte Tests zeigten, dass die YADrone-API alle Anforderungen erfüllt und somit gut in der Projektgruppe eingesetzt werden könnte. Mit Blick auf den Quelltext der API erschienen die Änderungen allerdings deutlich weitreichender und umfangreicher als für das Projekt nötig. Somit fiel letztlich die Entscheidung, selbst eine schlanke Vereinigung beider Schnittstellen zu implementieren. Als Basis wurde dabei wie in der YADrone-API die ARDroneForP5-API verwendet. Diese wurde in einem ersten Schritt um die in Abschnitt 5.1 angesprochene Navigationsdaten-Komponente der javadrone-API erweitert, welche neben den normalen Navigationsdaten auch in der Lage ist, die erkannten Vision-Tags zuverlässig auszulesen.
Anschließend wurde die vereinte API noch um eine Konfigurations-Klasse (siehe Listing 5.1) erweitert, welche dem Konstruktor der `ARDrone`-Klasse übergeben werden muss. Dies ermöglichte das einfache Testen verschiedener Konfigurationen. Die Attribute beziehen sich abgesehen von *outdoor* auf die Erkennung der Vision-Tags. In Abhängigkeit der enthaltenen Werte werden nach dem Verbinden zum Quadrokopter zugehörige Konfigurationskommandos an die

AR.Drone gesendet. Das Attribut *detectType* bestimmt, welche Art von Vision-Tags allgemein erkannt werden sollen. *detectTypeH* und *detectTypeV* bestimmen die Erkennungsarten für die horizontale und vertikale Kamera. *enemyColors* legt fest, Vision-Tags welcher Farbkombination erkannt werden sollen (siehe Abbildung 5.2), und *enemyWithoutShell* kennzeichnet, ob nur AR.Dronen dieser Farben oder auch die in der Abbildung gezeigten Aufkleber gefunden werden sollen. Nähere Informationen zu den Kommandos und Attributwerten finden sich in Kapitel 8.10 des AR.Drone Developer Guides [Pis+12]. Das Attribut *outdoor* legt letztlich fest, ob der Quadrokopter mit Wind zu rechnen hat oder sich geschützt innerhalb eines Gebäudes befindet. Nach der Fertigstellung wurde die Eigenentwicklung in ihrer Funktionalität verifiziert und mit Hilfe von Maven als Komponente in der Projektumgebung bereitgestellt. Nähere Informationen hierzu finden sich in Abschnitt 3.

5.2 Simulierte Quadrokopter

Die Kommunikation mit der Simulation ist ebenso einfach möglich wie die Kommunikation mit der Parrot AR.Drone. Aufgrund des gewählten Abstraktionsgrads kann in der Simulationsklasse eine Verbindung zur Vrep-Simulationsumgebung hergestellt werden. In jeder implementierten Methode wird ein Kommando an Vrep geschickt. Vrep wiederum verarbeitet das Kommando und antwortet. Die Antwort dient als Rückgabewert der in Java implementierten Methode. Auf welche Art und Weise die einzelnen Methoden des *Quadrocopter*-Interfaces für die Simulation implementiert sind, wird in Kapitel 6 detailliert erläutert.

KAPITEL 6

Simulationsumgebung

Bevor die in der PG implementierten Sicherheitsaspekte auf dem echten Quadrokopter verwendet werden, muss die Richtigkeit der Implementierung der Sicherheitsaspekte mit Hilfe einer virtuellen Simulationsumgebung überprüft werden. Deshalb spielt die Simulationsumgebung als Vorbereitungs- und Garantiefaktor des Erfolgs für den Einsatz der Sicherheitsaspekte auf dem realen Quadrokopter eine erhebliche Rolle.

6.1 Anforderungen

Die Simulationsumgebung muss einige Voraussetzungen erfüllen, damit die Ergebnisse, die in der PG erarbeitet wurden, umfassend und korrekt getestet werden konnten. Ziel der Simulationsumgebung ist es, eine dreidimensionale Umgebung zu erschaffen, in der sich ein Modell eines Quadrokopters bewegen kann. Dabei soll die Eingabe der Steuerungssignale des Quadrokopters sowohl direkt über Eingabegeräte wie Tastatur, Joystick bzw. Gamepad und Maus erfolgen können, als auch über entfernte Eingabegeräte mittels Netzwerk (z.B. WLAN). Außerdem muss es möglich sein, die Messwerte der Sensoren des Quadrokopters über ein Netzwerk an andere Geräte zu übermitteln. Es muss eine Methode zur Kollisionserkennung des Quadrokopters mit der Umgebung vorhanden sein und grundlegende physikalische Eigenschaften die für das Flugverhalten des Quadrokopters wichtig sind, wie Gravitationskraft und Trägheit, simuliert werden können. Außerdem müssen die einzelnen Sensoren des Quadrokopters (wie beispielsweise der nach unten gerichtete Ultraschallsensor) simuliert werden können. Die Simulationsumgebung sollte das Verhalten des Quadrokopters möglichst realitätsgetreu simulieren, damit es später bei den Tests mit dem echten Quadrokopter keine oder nur geringe Abweichungen zwischen dem Verhalten in der Simulation und dem Verhalten des echten Quadrokopters gibt.

Die Simulationsumgebung soll eine dreidimensionale Umgebung darstellen können, in der sich der Quadrokopter bewegt. Diese Umgebung soll mit Hindernissen und Objekten ausgestattet werden können, so dass sich hier die Funktionen des Quadrokopters testen lassen. Darunter fallen u.a. Funktionen wie die Berechnung des Abstandes zu Hindernissen, die sich vor dem Quadrokopter befinden, wie z.B. Wände oder die Berechnung der Flughöhe mit Hilfe einer Abstandsmessung zum Boden. Auch lassen sich in der Simulationsumgebung Probleme erkennen und Lösungen testen, ohne dass dabei der echte Quadrokopter beschädigt wird.

Einer der wichtigsten Punkte in der Simulationsumgebung ist die Physik, d.h. die physikalisch korrekte Simulation des Quadrokopters. Dies beinhaltet Bewegungsgrundlagen, wie beschleunigen und abbremsen oder auch die Gewichtskraft, aber auch Eigenschaften wie Massenträgheit des Quadrokopters. Auch die Kollisionserkennung fällt in diesen Bereich. Die Simulationsumgebung muss in der Lage sein, zu erkennen, ob das Model des Quadrokopters mit einem Hindernis kollidiert ist (z.B. ob er gegen eine Wand geflogen ist).

Es gab mehrere Möglichkeiten eine eigene Simulationsumgebung zu erstellen, die diese Anforderungen erfüllt:

1. Eine eigene Simulationsumgebung in Java komplett neu erstellen.

2. Aus mehreren vorhandenen Drittbibliotheken eine Simulationsumgebung zusammenstellen.

3. Die Verwendung einer kompletten Engine, die unsere Anforderungen erfüllt.

Letztendlich haben wir uns für die Software V-REP entschieden. Dies ist eine fertige Simulationsumgebung für die physikalische Simulation verschiedenster Roboter, welche im Januar 2013 kurz vor Beginn der PG als open-source-Software veröffentlicht wurde [Cop14]. Sie stellt den Kern unserer Simulationsumgebung dar und bietet sämtliche Funktionen, die für eine Simulation des Quadrokopters wichtig sind: Eine 3D-Grafik, um den Quadrokopter ähnlich wie in einem Flugsimulator testen zu können, zwei(austauschbare) Physikbibliotheken, eine Java-API für die Kommunikation und die Möglichkeit, das Verhalten des Quadrokopters mittels eines Skripts festlegen zu können.

6.2 V-REP

Die Simulationssoftware V-REP von Coppelia Robotics [Cop14] stellt eine Umgebung für die physikalische Simulation von Robotern bereit. Für Studenten und Universitäten kann V-REP dank der „Educational License“ kostenlos genutzt werden, solange die Software nicht für kommerzielle Projekte verwendet wird. Es handelt sich um eine integrierte Entwicklungsplattform (IDE) mit graphischer Benutzeroberfläche. In der Simulationsumgebung sind bereits vorkonfigurierte Roboter und eine einfache 3D-Umgebung vorhanden. Ebenfalls ist bereits ein Quadrokopter-Modell in der V-REP-Bibliothek vorhanden. Das Verhalten der Roboter wird durch die Skriptsprache LUA [PUC14] definiert und kann vom Entwickler verändert werden. Die Roboter können vom Entwickler mit vielfältigen Sensoren ausgestattet werden. Eine Client-API für diverse Plattformen ermöglicht den Netzwerkzugriff auf die V-REP Umgebung. Mit der Schnittstelle kann der Simulator gesteuert und die Sensordaten ausgelesen werden. Es existieren auch fertige 3D-Elemente, wie Wände und Säulen, die den leichten Aufbau einer komplexen 3D-Umgebung ermöglichen. Aus diesen Elementen wurde direkt in V-REP ein „Level“ als Testumgebung erstellt, in welchem einige Wände, Säulen und Hindernisse platziert wurden. Die Simulation selbst läuft auf allen gängigen Plattformen wie Windows, Mac und Linux. Die Client-Bibliothek ist u. a. als Java und C++ Variante erhältlich. Allerdings verwendet die Java-API eine Windows-spezifische DLL-Datei, die erst auf andere Plattformen portiert werden muss. Die V-REP Umgebung bietet alle Voraussetzungen für die Simulation des Quadrokopters aus einer Hand. Es stellt die 3D-Umgebung bereit und enthält bereits einen vorkonfigurierten Quadrokopter einschließlich physikalischer Simulation. Mit der Java-API kann der Quadrokopter über WLAN angesteuert und die Sensordaten ausgelesen werden.

6.3 Die V-REP-API

Um die Simulationsumgebung über entfernte Eingabegeräte ansteuern zu können, benutzen wir die Remote API von V-REP. Damit ist es nicht nur möglich eine laufende Simulation zu steuern, sondern auch eine laufende Simulation zu pausieren oder ein neues Level (d.h. eine neue Simulation) zu starten. Die Remote API stellt über 100 Funktionen zur Steuerung der Simulation zur Verfügung. Die Remote API besteht aus zwei Teilen, ein Teil für die Client-Seite und ein Teil für die Server-Seite.

Serverseitig wird die Remote API als Plugin für die V-REP-Simulationsumgebung realisiert. Clientseitig wird für die Verwendung der Remote API unter Java das Paket „coppelia“ benötigt, welches 12 Klassen enthält, sowie die dem verwendeten Betriebssystem entsprechende Bibliothek (remoteApiJava.dll, remoteApiJava.dylib or remoteApiJava.so).

Um aus dem Java-Programm heraus die Verbindung zu V-REP aufzubauen muss die „simxStart()“-Methode aufgerufen werden. Hier muss per Parameter die IP-Adresse des Rechners, auf dem das V-REP Programm läuft, sowie der entsprechende Port angegeben werden. Über die restlichen Parameter kann das Verhalten bei Verbindungsabbruch, der Timeout-Wert sowie die Rate, in der die Datenpakete hin und zurück gesendet werden.

Nachdem die Verbindung zu V-REP hergestellt wurde, muss die Simulation mit „simxStartSimulation()“gestartet werden. Hier kann der Operationsmodus für die Kommunikation festgelegt werden. Wir verwenden den Modus „simx_opmode_oneshot_wait“, welcher festlegt, dass nach dem Senden eines Befehls auf die Antwort gewartet wird, bevor mit dem Programm fortgefahren wird.

Der Teil der Kommunikation, der für die Steuerung des Quadrokopters zuständig ist, funktioniert wie folgt:

1. Auf der Client-Seite werden im Java-Programm die Eingaben des Nutzers abgefragt (welche Taste gedrückt wurde oder auf welcher Position der Joystick steht).
2. Diese Werte werden als Parameter der move-Methode des „Quadrokopter“-Interface übergeben.
3. Diese Methode bereitet die Werte nun auf (beschränkt sie auf ein festgelegtes Intervall und skaliert sie).

4. Falls die Werte einen bestimmten Schwellwert nicht überschreiten, wird davon ausgegangen, dass die Eingabe nicht gewollt war und der entsprechende Wert wird nicht übermittelt. Statt dessen wird eine „clearSignal“-Nachricht gesendet, die das Signal dessen Wert nicht groß genug war, entfernt.

5. Ist ein Wert größer als die Untergrenze, dann wird mit „simxSetFloatSignal“ein Signal an die Simulationsumgebung gesendet, welches die entsprechenden Werte enthält.

6. In der V-REP-Simulation wird dieses Signal nun empfangen und im LUA-Script, das dem Quadrokopter-Objekt im V-REP Editor zugeordnet ist, die Methode, die dem Signalnamen entspricht, ausgeführt. Der Signalwert, der zwischen -100 bis +100 begrenzt ist, wird der Methode als Parameter übergeben.

Folgende Signale sind in unserer Simulationsumgebung realisiert:

FORWARDBACK: Vor- und rückwärts fliegen (negativer Wert für rückwärts)
LEFTRIGHT: Nach links bzw. rechts fliegen
DOWNUP: Nach oben bzw. unten fliegen
ROTATE: Um die eigene Achse drehen
LAND: Landen
START: Starten

Mit den beiden Methoden „simxSetFloatSignal“ und „simxClearFloatSignal“ lassen sich also alle Signale setzen, die für die Steuerung des Quadrokopters nötig sind.

Außer dem Senden von Signalen lassen sich mit der Remote-API auch Daten aus dem Quadrokopter bzw. der Simulation auslesen, wie z.B. die aktuelle Position des Quadrokopters sowie die Sensordaten.

Um die Position des Quadrokopters auszulesen wird zuerst ein Zeiger auf das Objekt benötigt, das in V-REP den Quadrokopter darstellt. Von diesem Objekt kann dann die Position mit Hilfe der get-Methoden ausgelesen werden.

Sensordaten werden auf ähnliche Weise ausgelesen. Diese Sensordaten werden automatisch durch das main-Script des V-REP Editors erstellt. Es ist also nicht nötig, den Sensor über die Remote-API zu aktivieren. Lediglich die aktuellen Sensordaten müssen abgefragt werden.

Zuerst muss ein Handle auf das Sensorobjekt angefordert werden. Nun können die Sensordaten mit Hilfe der Methode „simxReadProximitySensor“ abgefragt werden und die Entfernung zum getroffenen Objekt kann berechnet werden.

6.4 Umsetzung

In der Simulationsumgebung V-REP sind einige Komponenten gegeben. Unter anderem sind dies eine funktionierende Physik, sowie ein Quadrokoptermodell. Die Aufgabe die sich aus der Projektbeschreibung ergibt, ist das Quadrokoptermodell durch Benutzereingaben in der Simulation zu steuern. Um eine Steuerung in V-REP zu entwickeln, muss es allgemein ermöglicht werden, ein beliebiges Objekt durch Benutzereingaben bewegen zu können. Eine Möglichkeit dies zu realisieren, ist das Erzeugen einer Kraft, deren Größe, sich durch den Benutzer beeinflussen lässt. Diese Kraft wirkt auf ein ausgewähltes Objekt und versetzt es damit in Bewegung. Eine Möglichkeit das Quadrokoptermodell in Bewegung zu versetzten ist es vier Kräfte zu generieren, wovon jeweils eine auf einen Propeller wirkt. Die Problematik die sich hier dabei ergibt, ist die folgende:

Um den Quadrokopter zu steuern werden vier gleichartige Kräfte erzeugt, die jeweils auf einen Propeller wirken. Das Ziel ist es, den Quadrokopter mit dieser Methode in eine gerade und stabile Fluglage zu versetzen. Da aber die Berechnung der Kräfte zeitdiskret stattfindet, werden die Berechnungen zum Erzeugen der Kraft auf einen Propeller nicht gleichzeitig ausgeführt. Aus diesem Grund werden die Kräfte nicht genau gleichzeitig auf die Propeller wirken und der Quadrokopter wird in der Luft gekippt und stürzt ab. Sobald der Quadrokopter in der Luft gekippt ist, kann er sich nicht mehr stabilisieren. Was nun gebraucht wird, ist ein Mechanismus, der den Quadrokopter stabil in der Luft hält, indem die Kräfte untereinander ausgeglichen werden. Das Ausgleichen der Kräfte geschieht durch einen Software-Regler. Ein Regelalgorithmus ist dazu in der Lage, die Kräfte des Quadrokopters so zu berechnen, dass dieser sich in der Luft stabilisieren und vorgegebene Winkelgrößen erreichen kann. Die nötigen Grundlagen aus der Regelungstechnik werden im Kapitel 2.5 vorgestellt.

Das Quadrokoptermodell bewegt sich in einem dreidimensionalem Koordinatensystem (siehe Abbildung 6.1). V-REP bietet die Möglichkeit, durch eine Abfrage im Programmcode, die Position und die Orientierung aller Objekte zu erlangen. Wie schon beschrieben, ist es das Ziel, den Quadrokopter in einer stabilen Fluglagen zu bringen und darüber hinaus zu halten. Diese Aussage ist äquivalent dazu, dass die Flugwinkel $\gamma = 0$ und $\beta = 0$ sind. Zusätzlich soll es möglich sein, den Quadropkopter im dreidimensionalen Raum zu bewegen. Zunächst beschränken wir uns nur auf das Fliegen auf den Achsen x und y. Das Ziel ist es, einen Regler zu entwerfen, der bestimmte Winkelgrößen (auch Winkel = 0) erreicht und darüber hinaus auch hält. Ein Winkel ungleich null bedeutet, dass der Quadrokopter in der Luft gekippt wird und sich entlang der x- oder y- Achse bewegt. Dieser Regler nennt sich im folgenden Winkelregler und wird weiter unten im Abschnitt *Winkelregler* beschrieben. Dieser Winkelregler wird alleine jedoch nicht ausreichen. In einem Experiment wurde beobachtet, dass der Quadrokopter bei einem Flug mit

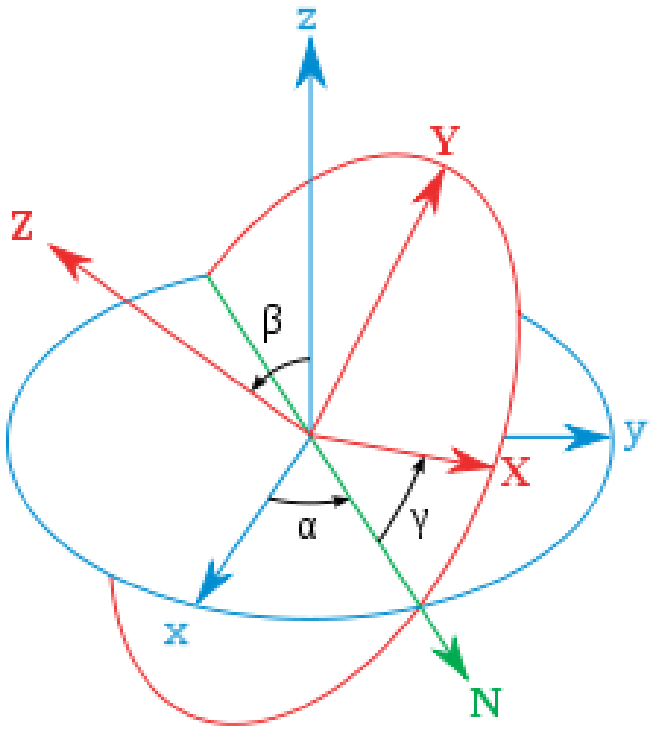


Abbildung 6.1: Darstellung der drei Flugrichtungen und der drei Drehwinkel[1]

den Winkeln $\gamma = 0.5$ und $\beta = 0$ zwar vorwärts fliegt, aber immer weiter beschleunigt und damit schneller wird. Dieses Verhalten macht es notwendig, die Geschwindigkeit zu begrenzen. Realisiert wurde dies durch einen Geschwindigkeitsregler (siehe Unterkapitel *Geschwindigkeitsregler*. Dieser Regler bewirkt, dass der Quadrokopter vorgegebene Geschwindigkeiten annimmt und darüber hinaus auch hält. Mit diesen beiden Reglern ist es möglich, den Quadrokopter in zwei Flugrichtungen, in einer vorgegebenen Geschwindigkeit fliegen zu lassen. Zusätzlich bewirkt das setzen keiner Stellgröße (Geschwindigkeit = 0), dass der Quadrokopter seine Fluglage stabilisiert und gerade in der Luft bleibt.

Die noch fehlende Flugrichtung, ist der Flug entlang der z-Achse. Der Flug entlang der z-Achse wird durch einen Höhenregler realisiert. Dieser Höhenregler wird im Unterkapitel *Höhenregler* vorgestellt. Um die Höhe des Quadrokopters zu ändern, wird eine vom Benutzer generierte Führungsgröße(siehe Unterkapitel Einführung in die Regelungstechnik) vorgegeben. Der Höhenregler nutzt diese Führungsgröße dazu, um die wirkende Kraft auf den Propellern zu erhöhen oder zu verringern. Dies führt dazu, dass der Quadrokopter auf- oder absteigt.

Die letzte noch fehlende Größe, die es zu beeinflussen gilt, ist die Änderung des Winkels α. Um eine Rotation um diesen Winkel zu erzeugen, wurde ein Rotations-Regler implementiert (siehe Unterkapitel Rotationsregler). Dieser nimmt eine Führungsgröße vom Benutzer entgegen und regelt hinsichtlich dieser.

Durch diese vier Regler(Winkel-, Geschwindigkeits-, Höhen- und Rotationsregler) ist es möglich, den Quadrokopter entlang aller drei Achsen zu bewegen und um alle drei Achsen rotieren zu lassen.

KAPITEL 7

Steuerungsmodell

Dieses Kapitel beschreibt das Steuerungsmodell, das mit jABC entworfen wurde. Das Steuerungsmodell ist dafür verantwortlich, in kurzen Abständen die Eingabe vom Nutzer abzufragen und an den Quadrokopter weiterzuleiten. Dieses Wechselspiel von Abrufen und Verarbeiten von Nutzereingaben ist der Lebenszyklus der Anwendung.

In Abbildung 7.1 ist das jABC-Steuerungsmodell zu sehen. Dort sind, dem Hauptzyklus (`ReadControllerInput` und `ProcessInputLinear`) vorangestellt, noch zwei weitere SIBs zur Initialisierung zu sehen.

In den folgenden Abschnitten werden nun die Initialisierung sowie das Abrufen und Verarbeiten von Nutzereingaben behandelt. Schließlich folgt ein Abschnitt speziell zur Sensorik. Dort wird beschrieben, mit welchen SIBs auf Sensordaten des Quadrokopters zugegriffen werden kann.

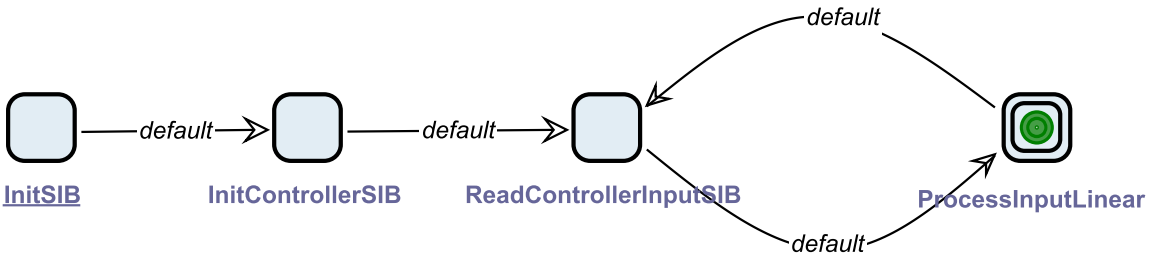


Abbildung 7.1: Das Steuerungsmodell als jABC-Modell

7.1 Initialisierung

Bevor im Steuerungsmodell die Hauptschleife zur Eingabeabfrage und -verarbeitung betreten wird, müssen Initialisierungen getroffen werden. Im wesentlichen umfasst dies zwei Entscheidungen: Welche Implementierung des Quadrocopter-Interfaces und welche des Controller-Interfaces sollen genutzt werden? Hier stehen z.B. die Tastatur und der Xbox Controller als Eingabemittel bzw. die Simulationsumgebung oder die Parrot AR.Drone als Quadrokopter zur Verfügung. Da diese Entscheidung dem Endnutzer überlassen werden soll, bietet sich ein Dialogfenster zu Auswahl an.

Realisiert wurde dies in jABC über zwei SIBs: Das InitSIB und das InitControllerSIB. Abbildung 7.2 hebt die beiden SIBs im Steuerungsmodell hervor.

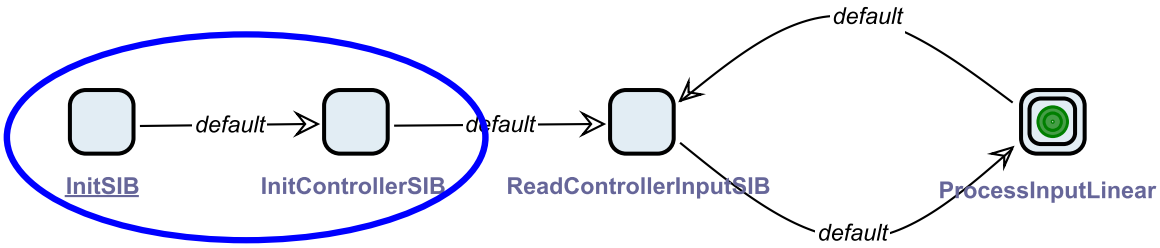
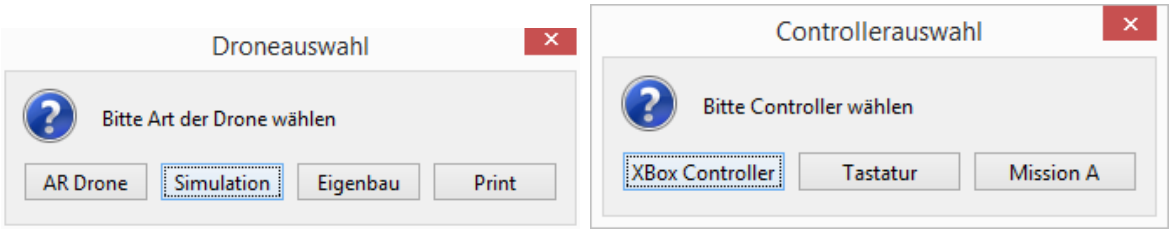


Abbildung 7.2: Initialisierungen im Steuerungsmodell

Beide SIBs zeigen ein Dialogfenster an, in denen ein Button pro Auswahlmöglichkeit angezeigt wird. Der Auswahldialog wird über ein `JOptionPane` aus dem `javax.swing`-Paket realisiert. Abbildung 7.3 zeigt die beiden Auswahldialoge.



(a) Drohnenauswahl (b) Eingabegerätauswahl

Abbildung 7.3: Auswahldialoge

Nach Auswahl einer Option wird eine lokale Variable des entsprechenden Interfaces gemäß der Auswahl instanziiert. Danach wird diese Variable über einen `ContextKey` im Ausfühungskontext abgelegt. Dieser `ContextKey` wird anschließend von anderen SIBs genutzt, um auf die Quadrokopter- bzw. Controller-Instanz zuzugreifen.

7.2 Eingabeverarbeitung

Sobald die Initialisierung durchgeführt wurde kann mit dem Abfragen und Verarbeiten der Eingabe begonnen werden. Ziel war es, einen simplen zweigeteilten Zyklus zu erstellen, in dem abwechselnd die Steuerungsinformationen aus dem Eingabegerät abgefragt werden und daraufhin von dem Quadrokopter verarbeitet werden. Verlassen wird diese Schleife nur, wenn vom Eingabegerät ein Abbruchsignal gesendet wird.

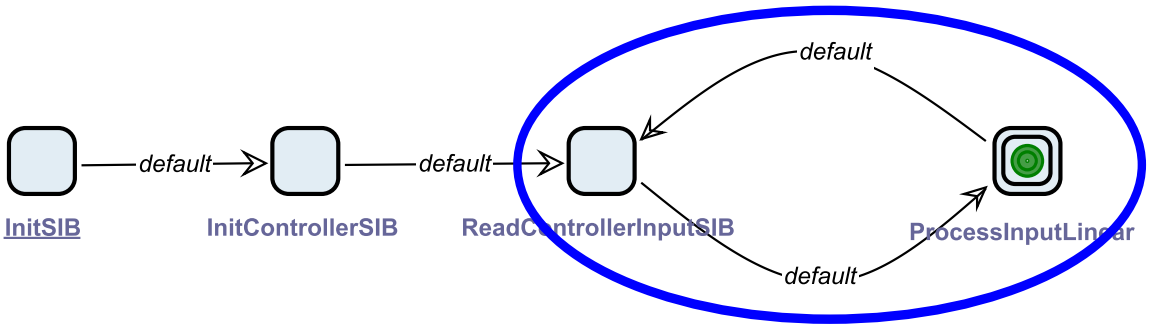


Abbildung 7.4: Eingabeverarbeitung im Steuerungsmodell

Wichtig war hier, das Auslesen und Verarbeiten so zu implementieren, dass ein Zyklus eine sehr kurze Laufzeit hat. Wird die Zyklusdauer zu lang, so wird dem Nutzer die Verzögerung zwischen seiner Eingabe und deren Verarbeitung bewusst. Daher wurde angestrebt, die Dauer eines Zyklus stets unter 20 ms zu halten.

Das Auslesen der Eingabe stellte sich diesbezüglich als unproblematisch heraus. Hierzu wurde das ReadControllerInputSIB implementiert, welches lediglich die Controller-Instanz aus dem Anwendungskontext holt und die Eingabeinformationen in passenden `ContextKeys` im Kontext ablegt. Dies dauert im Regelfall nicht länger als 1 ms.

Die Verarbeitung der Eingabe hingegen bereitete in der Quadrokopter-Simulation zunächst Probleme was die Laufzeit anging. Hier konnte etwas Abhilfe geschafft werden, indem die Anfragen, die an die Simulationsumgebung geschickt wurden, von synchron auf asynchron geändert wurden. Das heißt, das Steuerungsmodell wartet nicht auf die Antwort von der Simulationsumgebung bezüglich der Anfrage, sondern schickt sie einfach raus und fährt mit der Ausführung fort. Des weiteren ist die Simulationsumgebung sehr leistungshungrig. Besonders bei Laptops gab es hier Leistungseinbrüche, was dementsprechend zu höheren Zykluslaufzeiten führte. Bei vielen Rechnern half es, verschiedene Werte bei der Abtastrate und der Berechnungsgenauigkeit der Simulationsumgebung auszuprobieren und die für den jeweiligen Rechner am besten geeignete Konstellation zu verwenden.

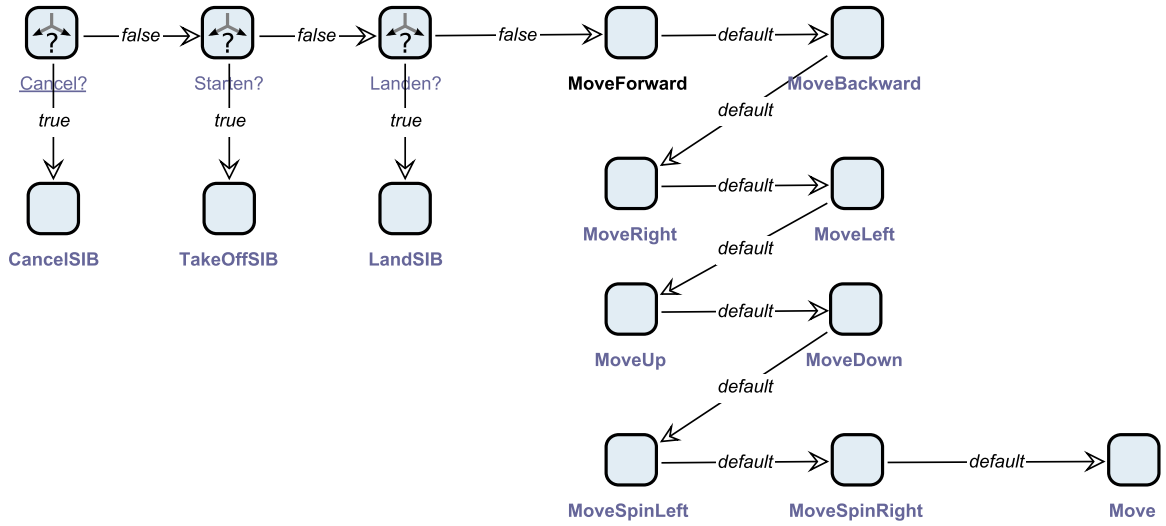


Abbildung 7.5: Submodell zur Eingabeverarbeitung

7.3 Bewegung

In diesem Abschnitt wird nun erläutert, wie genau der Quadrokopter aus dem jABC-Modell heraus angesteuert wird.

Während zunächst die Verarbeitung der Eingabe in einem einzelnen SIB realisiert wurde (Eingabeinformationen aus dem Anwendungskontext holen und an die Quadrokopter-Instanz schicken), wurde bald klar, dass dort eine feingranularere Kapselung notwendig ist. Grund dafür ist, dass bei der später geplanten Einwebung von Aspekten jedes eingehende Signal sowie dessen Verarbeitung individuell „manipuliert“ werden können soll.

Dementsprechend wurde also das Submodell `ProcessInputLinear` entworfen (siehe Abbildung 7.5). In diesem Modell wurden zunächst die Überprüfung der Abhebe-, Lande- und Abbruchsignale in einzelne SIBs extrahiert, sowie das Senden des entsprechenden Signals an den Quadrokopter. Bei den Bewegungssignalen wurde noch feiner aufgeteilt. Statt der in den Interfaces genutzten Winkel und Geschwindigkeiten, die jeweils vom negativen bis zum positiven Maximum gehen, wurde hier für jede Richtung ein einzelnes SIB implementiert. Das heißt, dass beispielsweise `frontBackTilt` in die ContextKeys `forwardAcceleration` und `backwardAcceleration` aufgeteilt wird. Dies geschieht über das `MoveForward`- und `MoveBackward`-SIB, welche jeweils den aktuellen `frontBackTilt` des Controllers auslesen und dann den entsprechenden Wert in den entsprechenden `ContextKey` schreibt.

Letztendlich fügt das `Move`-SIB dann wieder alle einzelnen Werte zusammen und sendet sie mit der `Quadrocopter.move(...)` Methode des Interfaces an die entsprechende Instanz. Dies mag redundant erscheinen, jedoch ist diese Aufteilung für das Einweben von Sicherheitsaspekten

von großem Vorteil. Als Beispiel sei hier die Kollisionsvermeidung betrachtet: Befindet sich ein Hindernis vor dem Quadrokopter, so kann durch Einweben der entsprechenden SIBs um das `MoveForward`-SIBs, die dann den `ContextKey` `frontAcceleration` manipulieren, eine Kollision vermeidet werden. Detaillierte Informationen zu Sicherheitsaspekten und deren Einwebung finden sich in Kapitel 9.

7.3.1 Sensoren

Zur Vermeidung von Kollisionen wird Sensorik benötigt, mit deren Hilfe die Bewegung des Quadrokoptes sowie die Umwelt analysiert werden können. Dazu gehören beispielsweise Ultraschall- und Luftdrucksensoren, sowie eine Kamera. Auch der Ladestand der Batterie ist für eine Notlandung relevant. Die Abfrage der Sensorik zur Umgebungs- und Selbstanalyse wird in drei verschiedene Sensor-SIBs gekapselt.

Das **GetAltitudeSIB** gibt die aktuelle Flughöhe das Quadrokopters zurück. Dazu wird der Luftdrucksensor der Parrot AR.Drone genutzt. Das **GetBatterySIB** liefert den Ladezustand der Batterie wieder, sodass frühzeitig eine Notlandung ausgelöst werden kann. Mit Hilfe des **GetDistanceSIB** wird die Umgebung analysiert. Bei der Implementierung dieser Funktionalität wurde im Falle der AR.Drone auf das Kamerabild zurückgegriffen, da zur Abstandserkennung lediglich am Boden des Quadrokopters ein Ultraschallsensor verbaut wurde. Zur Bestimmung des Abstands seitens der AR.Drone werden Markierungsstreifen bestimmter Farbe eingesetzt. Diese werden automatisch von der AR.Drone erkannt und in der Programmierung als VisionTag-Objekt (siehe Kapitel **??**) zurückgegeben. Mit Hilfe dieses Objekts kann nun mittels *obj.getDistance()* auf die Entfernung zur Markierung zurückgegriffen werden. Da Bildanalyse den Rahmen der Projektgruppe überschreiten würde, wurde der Einsatz auf ein Minimum reduziert und auf bereitstehende Funktionen der AR.Drone zurückgegriffen.

Die Markierung ist ungefähr 30 cm von den Hindernissen entfernt auf den Boden geklebt. Diese Minimaldistanz dient zur Erleichterung der Berechnung in der Kollisionserkennung des Quadrokopters. Abbildung 7.6 illustriert die Abstandserkennung.

Die reale Distanz d_r ist die Entfernung des Quadrokopters zum Hindernis, die mittels *obj.getDistance()* ermittelt werden kann. Die Distanz d zu einem Hindernis wird berechnet durch $d = \sqrt{d_r^2 - h_f^2} + 0.3$, wobei h_f die Flughöhe vom Quadrokopter ist, die mit Hilfe des Ultraschallsensor ermittelt werden kann.

Die Orientierung der Markierung wird mit Hilfe des Kamerabilds berechnet. Das Bild der Kamera nimmt einen Quadrat um den Quadrokopter herum auf und wirkt wie ein Koordinatensystem, welches die obere linke Ecke (0,0) und die untere rechte Ecke (1000,1000) als Koordinaten hat. Im Bild befindet sich der Quadrokopter in der Mitte und seine Koordinaten lauten (500,500). Mit dem Koordinatensystem des Kamerabildes simulieren wir vier Blickrichtungen des Quadrokopters mit vier entsprechenden Dreiecken:

1. vordere Richtung entspricht Dreieck (0,0),(1000,0),(500,500)
2. hintere Richtung entspricht Dreieck (500,500),(0,1000),(1000,1000)

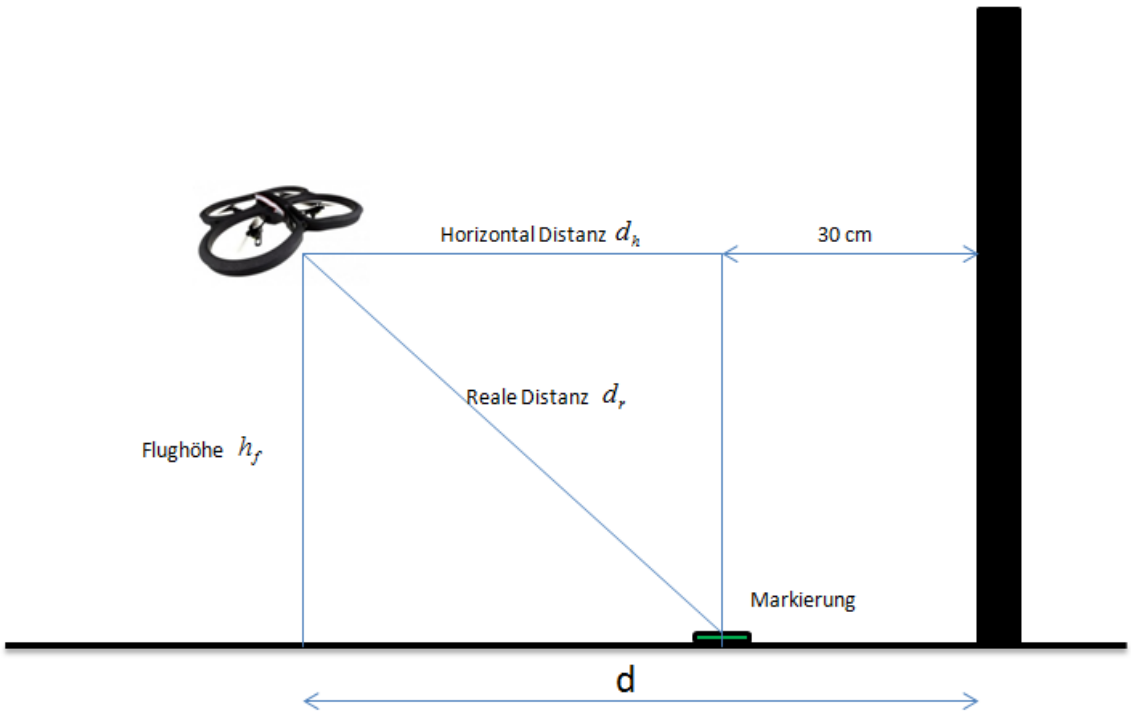


Abbildung 7.6: Quadrokopter, Markierung und Hindernis

3. linke Richtung entspricht Dreieck (0,0),(0,1000),(500,500)
4. rechte Richtung entspricht Dreieck (500,500),(1000,0),(1000,1000)

Abbildung 7.7 illustriert die Sehwinkel und die entsprechenden Dreiecke.

Indem wir die Anwendung der Baryzentrische Koordinaten in der Dreiecksgeometrie einsetzen, können wir herausfinden, in welcher Richtung sich die Markierung befindet. Die Position (v_x, v_y) der Markierung im Bild wird mittels der APIs des Quadrokopters ermittelt. Um zu überprüfen, ob die Markierung innerhalb eines Dreiecks positioniert ist, werden folgende Bedingungen geprüft:

1. Die Koordinate (v_x, v_y) von der Markierung ist die lineare Kombination der Koordinaten der Eckpunkten des Dreiecken: $(v_x, v_y) = \alpha \left(v_{a_x}, v_{a_y}\right) + \beta \left(v_{b_x}, v_{b_y}\right) + \gamma \left(v_{c_x}, v_{c_y}\right)$
2. Die Koeffizienten α, β, und γ müssen gleichzeitig größer als 0 und kleiner als 1 sein.
3. $\alpha + \beta + \gamma = 1$

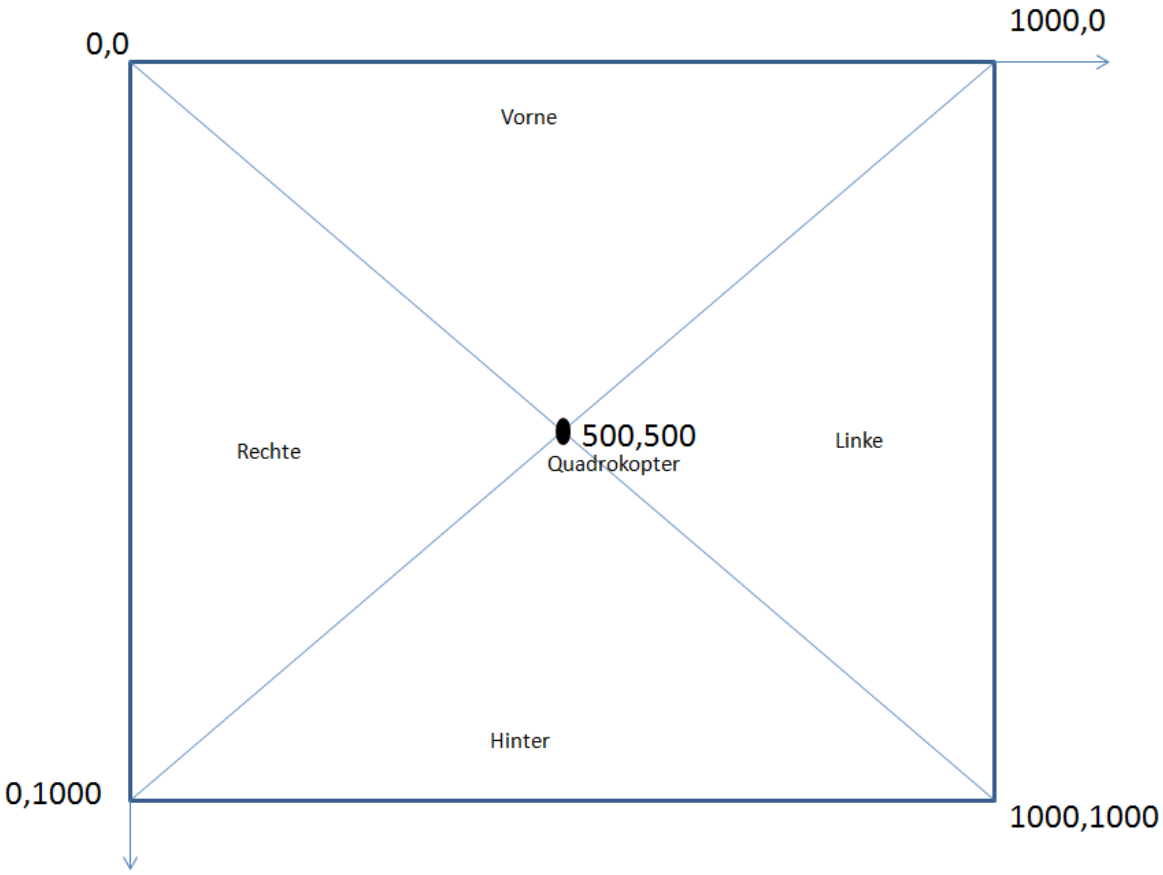


Abbildung 7.7: Quadrokopter, Sehrichtungen und Dreiecke

Ein Gleichungssystem wird für die obigen Bedingungen aufgestellt. Wenn das Gleichungssystem eindeutig lösbar ist, ist die Markierung mit der Position (v_x, v_y) innerhalb des Dreiecks, welches von den 3 Punkten $\left(v_{a_x}, v_{a_y}\right)$, $\left(v_{b_x}, v_{b_y}\right)$, $\left(v_{c_x}, v_{c_y}\right)$ aufgespannt wird.

KAPITEL 8

Missionsmodelle

In diesem Kapitel wird die Thematik der Missionsmodelle behandelt. Nach einer theoretischen Einführung wird geschildert, wie im Verlauf der Projektgruppe ein Missionsmodell entstanden ist, welches in der Lage ist, einen Quadrokopter sicher durch ein unbekanntes Gelände zu einem bestimmten Zielpunkt zu führen.

8.1 Grundlagen

Als Missionsmodell wird ein Modell bezeichnet, welches an ein steuerbares Objekt (im Rahmen der Projektgruppe ein Quadrokopter) automatisiert Steuersignale sendet, um bestimmte Aufgaben zu erfüllen. Die Frage, die es zu beantworten gilt ist, was diese Aufgaben sein können. Hierbei wurde zwischen vier verschiedenen Aufgaben unterschieden.

- *Sicherheitsaspekt:* Suche nach einem Landeplatz
- *Maximale Gebietsabdeckung:* Abfliegen/Erkunden eines (un-)bekannten Terrains
- *Sicheres Manövrieren:* Abfliegen einer vorgegebenen Route durch (un-)bekanntes Terrain
- *(Effiziente) Suche:* Suchen eines Zieles in (un-)bekanntem Terrain

Die erste Aufgabe ist ein besonderes Ziel der Projektgruppe und ein Missionsmodell, welches zur Realisierung einer sicheren Landung benötigt wird. Die sichere Landung kann als Sicherheitsaspekt eingewoben werden und wird daher an anderer Stelle beschrieben. Bei der zweiten Aufgabe hat der Quadrokopter das Ziel, ein Gebiet möglichst gut zu erkunden. Das bedeutet, dass möglichst viele Informationen über das Gebiet gesammelt werden sollen. Ein Ansatz für solche Arten von Missionsmodellen findet sich in [FK10].

Das sichere Manövrieren ist das Abfliegen einer bestimmten Route durch ein Gebiet. Hierbei kann diese Route durch Wegpunkte gegeben sein, aber auch durch zeitlich begrenzte Steuerangaben, die der Quadrokopter umsetzen soll. Ein Vorgehen, bei dem sich das automatisiert fliegende Objekt eine solche Route generiert, ist in [Rao95] dargestellt.

Das Suchen eines Ziels in einem Gebiet kann in zwei unterschiedliche Fälle eingeteilt werden. Zum einen kann der Zielort vorgegeben sein und dem Quadrokopter bekannt sein, so dass sich dieser zu jeder Zeit an dem Zielpunkt orientieren kann. Dieses ist das Missionsmodell, welches im Rahmen der Projektgruppe realisiert wurde. Eine ähnliche Mission wird in [TA09] durch den Einsatz von Fuzzy-neuronalen Netzen gelöst. Der andere Fall ist, dass der Quadrokopter nicht weiß wo der Zielort ist, sondern anhand einer bestimmten Information beim Entdecken des Zielortes erkennen würde, dass er an der richtigen Stelle ist.

8.2 Labyrintherkundung

Wie im vorherigen Abschnitt erwähnt, wurde im Rahmen der Projektgruppe ein Missionsmodell modelliert, welches dem Quadrokopter ermöglicht einen Zielpunkt in einem Gebiet anzufliegen, ohne weitere Eingaben vom Benutzer erhalten zu müssen. Der Quadrokopter ist hierbei in der Lage Hindernissen auszuweichen und somit das Ziel sicher zu erreichen.

Hierbei wurde der Ansatz verfolgt ein zustandsbasiertes Modell zu nutzen, wie es in Abb. 8.1 zu sehen ist. In fünf verschiedenen Modi können die benötigten Bewegungen für das Missionsmodell realisiert werden: Zielausrichtung, Vorwärtsflug, Parallelausrichtung, Parallelflug und Linksflug. Die verschiedenen Modi werden werden in Abschnitt 8.2.1 detailliert erläutert. Zu Beginn einer Mission befindet sich der Quadrokopter im Zielausrichtungsmodus. Das bedeutet, dass der Quadrokopter sich zunächst einmal zum Ziel ausrichtet. Sobald das Ziel in Sicht ist wird der Vorwärtsflug aktiviert. Der Vorwärtsflug wird gestoppt sobald der Abstand zu einem Hindernis zu klein geworden ist oder ein Vorwärtsflug mit einer bestimmten Zykluszahl veranlasst wurde. Das wird beispielsweise bei der Überwindung einer äußeren Ecke benötigt. Im einfachen Fall wird der Vorwärtsflug demnach verlassen, sobald ein Hindernis erkannt wurde. In

diesem Fall wird die Parallelausrichtung aktiviert. Wie die Bezeichnung des Modus schon sagt, richtet sich der Quadrokopter parallel zum Hindernis aus. Sobald die Parallelausrichtung erfolgreich abgeschlossen wurde, wird der Parallelflug aktiviert. Während des Parallelflugs fliegt der Quadrokopter mit einem festen Abstand parallel zum Hindernis. Der Parallelflugmodus wird beendet sobald kein Hindernis mehr existiert zu dem parallel geflogen werden kann oder ein Hindernis den Weg nach vorne versperrt. Im letzteren Fall wird erneut eine Parallelausrichtung mit anschließendem Parallelflug durchgeführt. Im anderen Fall wird ein Vorwärtsflug mit fester Zyklusanzahl durchgeführt. Darauf folgt ein Linksflug ebenfalls mit fester Zyklusanzahl. Diese Kombination dient zur Überwindung von äußeren Ecken. Sobald die Ecke überwunden wurde, der Linksflug also endet, wird erneut die Zielausrichtung durchgeführt und die Abfolge der Modi beginnt von vorn unter den eben geschilderten Bedingungen.

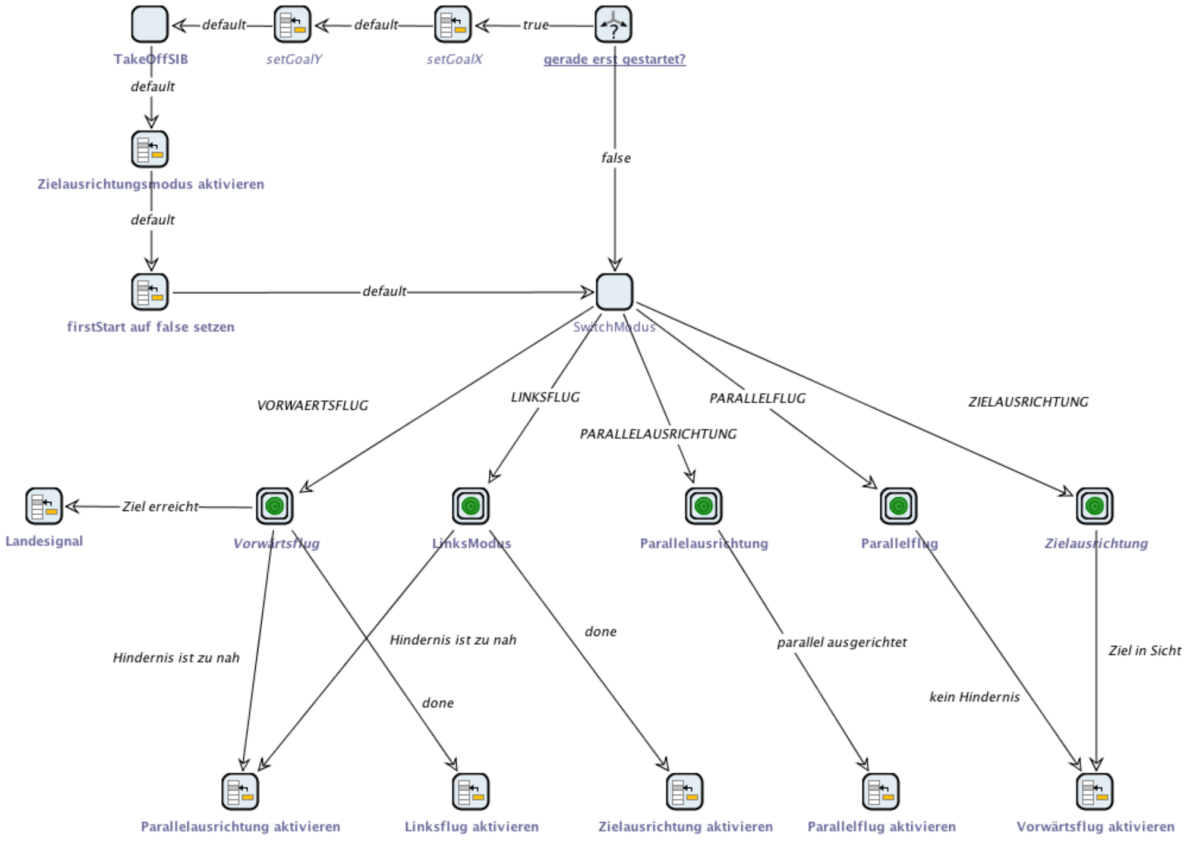


Abbildung 8.1: Missionsmodell: Labyrintherkundung

8.2.1 Modi

Im Folgenden werden die verschiedene Modi, die in der Labyrintherkundung genutzt werden, detailliert erläutert.

Zielausrichtung

Dieser Modus dient dazu das Flugobjekt zum Ziel hin auszurichten. Wie in Abb. 8.2 zu sehen ist wird dazu zunächst innerhalb des CheckTargetInSightSIB die erforderliche Drehrichtung berechnet. Wie genau diese Berechnung realisiert wird ist in Abschnitt 8.2.2 beschrieben. Sobald das Ziel in Sicht ist werden die Controller-Signale auf 0 gesetzt und ein Moduswechsel kann stattfinden.

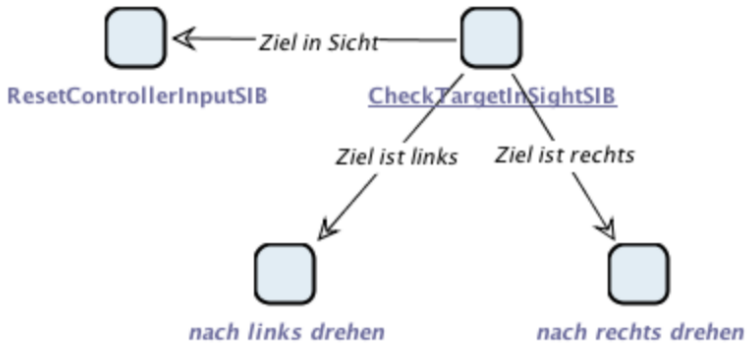


Abbildung 8.2: Modus Zielausrichtung

Vorwärtsflug

Der in Abb. 8.3 dargestellte Vorwärtsflug dient dazu ein Ziel auf direktem Wege anzufliegen. Es wird bei der Ausführung dieses Modus in jedem Schritt geprüft, ob man das Ziel erreicht hat. Sollte dies der Fall sein ist die Mission erfolgreich verlaufen. Während des Flugs wird zusätzlich geprüft, ob das Ziel noch in Sicht ist, also ob der aktuelle Flugwinkel zu stark vom Zielflugwinkel abweicht. Sollte die Differenz zwischen diesen beiden Winkeln einen bestimmten Wert überschreiten, so wird der Flugwinkel angepasst. Auf diese Weise wird verhindert, dass das Ziel nicht korrekt angeflogen wird. Der Vorwärtsflug wird beendet sobald eine Kollision verhindert wurde. Wie genau diese Kollisionsvermeidung erkannt wird ist in Abschnitt 8.2.2 beschrieben.

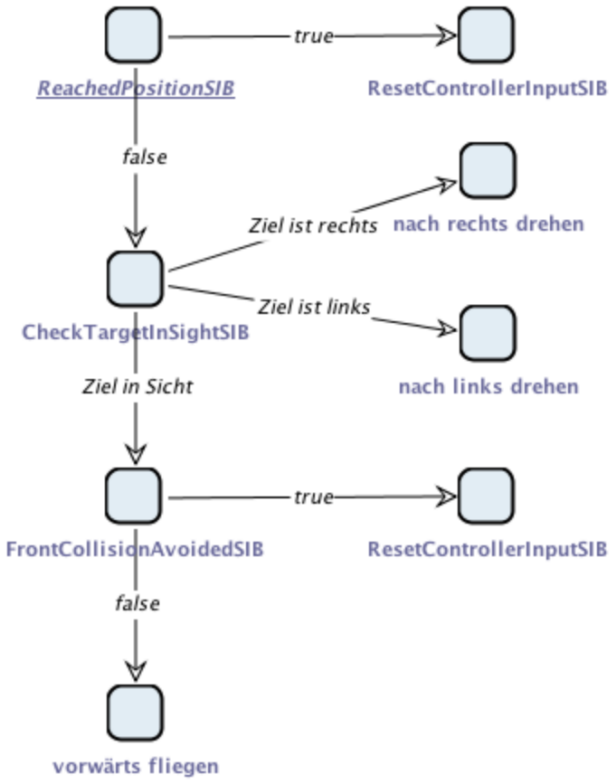


Abbildung 8.3: Modus Vorwärtsflug

Parallelausrichtung

Dieser Modus wird benötigt um Hindernisse zu überwinden. Der Quadrokopter soll dazu parallel zum Hindernis ausgerichtet werden um es anschließend umfliegen zu können. Es gibt zwei unterschiedliche Vorgehensweisen um den Quadrokopter parallel zum Hindernis auszurichten.

Variante A setzt hierbei auf das kontinuierliche Messen des Abstands nach links und ist in Abb. 8.4 abgebildet. Sobald ein minimaler Abstand erreicht wird stoppt die Drehung und der Quadrokopter ist parallel ausgerichtet. Um einen minimalen Abstand zu erkennen muss der Quadrokopter in jedem Fall über den minimalen Abstand hinwegdrehen. Das bedeutet, dass keine genaue Parallelausrichtung auf diese Art und Weise möglich ist.

Variante B soll diese Ungenauigkeit vermeiden und basiert auf verschiedenen Winkelberechnungen. Der Quadrokopter dreht nach rechts bis sowohl der Sensor nach vorne als auch nach links einen Abstand zum Hindernis liefern. Mit Hilfe dieser beiden Abstandsdaten wird die Orientierung des Hindernisses bestimmt. Um eine Parallelausrichtung zum Hindernis zu erreichen, muss der Quadrokopter die gleiche Orientierung im Raum annehmen. Dazu wird er gedreht bis die berechnete Orientierung erreicht ist.

Parallelflug

Der Parallelflugmodus ist in Abb. 8.5 abgebildet und besteht im Prinzip nur aus einer Überprüfung der Distanz zum Hindernis nach links. Dies ist notwendig um Ungenauigkeiten bei der Ausrichtung vernachlässigen zu können. Der Quadrokopter soll beim Parallelflug eine bestimmte Distanz zum Hindernis halten. Die Messung wird in vier unterschiedliche Intervalle eingeteilt:

- OK - dies ist die Distanz, in der der Parallelflug normal fortgesetzt werden kann und der Quadrokopter sich weiter nach vorne bewegt. Zusätzlich wird geprüft ob eine Kollision nach vorne erkannt wurde, um ein sicheres Fliegen zu gewährleisten
- TOOCLOSE - in diesem Fall ist der Quadrokopter zu nah am Hindernis und muss nach rechts fliegen, um wieder einen akzeptablen Abstand zu erhalten
- TOOFAR - hier ist der Quadrokopter zu weit vom Hindernis entfernt. Um den Parallelflug fortsetzen zu können, muss er wieder näher ans Hindernis heranfliegen
- NOOBSTACLE - Der Quadrokopter ist an einer Ecke des Hindernisses angekommen und hat deshalb am linken Sensor keinen Kontakt mehr zum Hindernis. Zur Überwindung der Außenkante wird ein endlicher Vorwärtsflug eingeleitet.

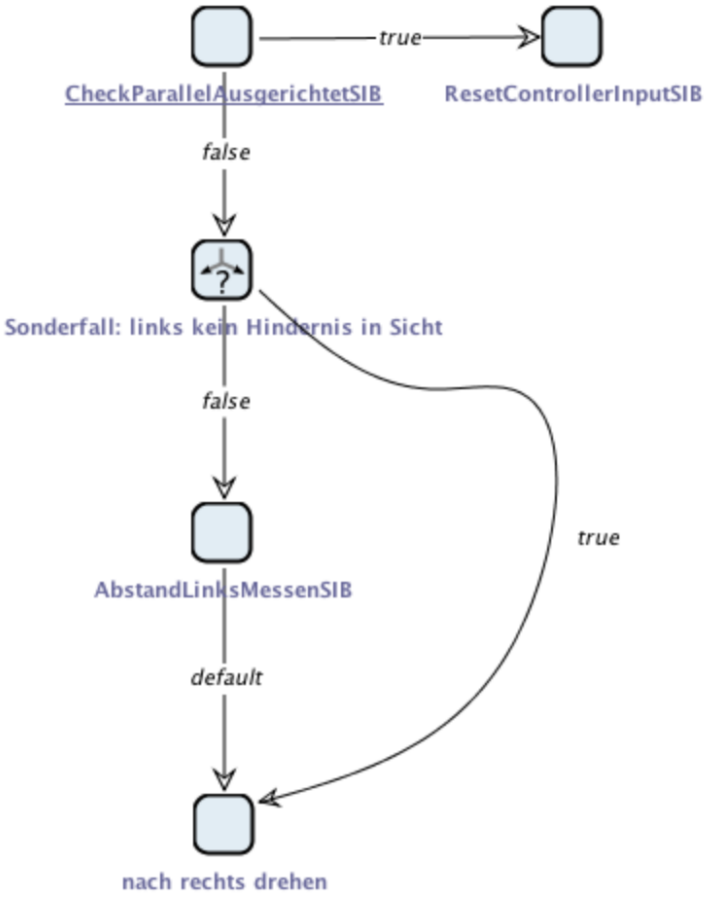


Abbildung 8.4: Modus Parallelausrichtung (Variante A)

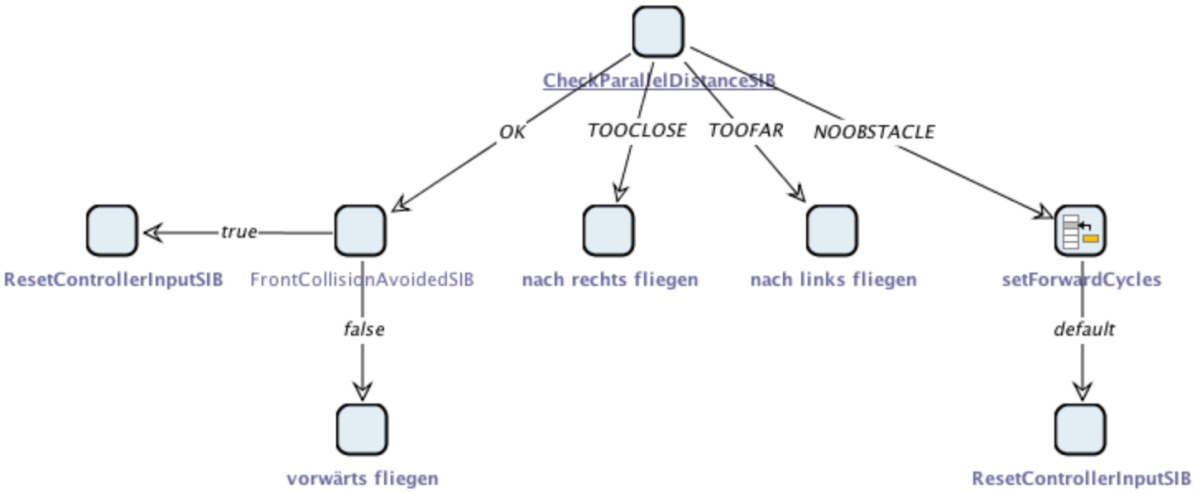


Abbildung 8.5: Modus Parallelflug

Linksflug

Der endliche Linksflug (s. Abb. 8.6) wird in Kombination mit dem endlichen Vorwärtsflug dazu genutzt um den Quadrokopter um eine Außenecke herum zu bewegen. Dieser wird aufgerufen wenn der endliche Vorwärtsflug durchgeführt wurde. Sobald der Linksflug aktiviert ist, wird ein Zyklus x-mal durchlaufen. Dadurch wird ermöglicht, dass der Linksflug nach einer bestimmten Zeit beendet wird und der Quadrokopter die Außenecke umflogen hat.

8.2.2 *SIB*-Library

Um u.a. komplexe mathematische Berechnungen nicht modellieren zu müssen, wurden einige Funktionalitäten, die in dem erstellten Missionsmodell benötigt wurden, als SIBs realisiert. Diese werden im Folgenden detailliert beschrieben.

AbstandLinksMessenSIB

Dieses SIB wird für das Finden des Minimalabstands bei der Parallelausrichtung (Variante A) benutzt. Zunächst wird der Abstand nach Links gemessen. Dieser Abstand wird unter bestimmten Bedingungen in den ContextKey *minDistanceForParallel* geschrieben. Hierbei muss

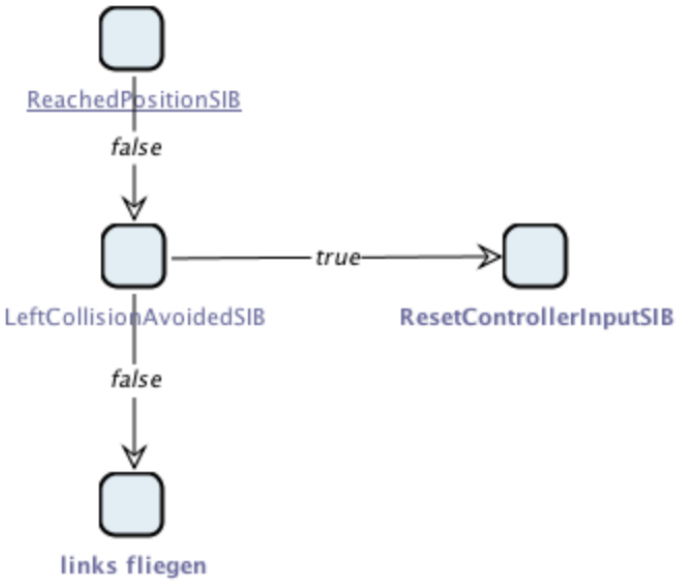
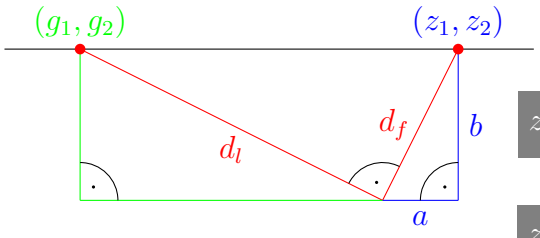


Abbildung 8.6: Modus Linksflug

darauf geachtet werden, dass die Abstandsmessung sehr schwanken kann. Daher sollte der Abstand nicht in jedem Fall in den Kontext geschrieben werden. Die erste Bedingung ist, dass der Abstand kleiner sein muss, als derjenige der bisher im Kontext steht. Um Schwankungen möglichst zu neutralisieren muss der Abstand beispielsweise drei mal in Folge kleiner sein um in den Kontext geschrieben zu werden. Hierfür sind verschiedene Vorgehensweisen denkbar.

CheckParallelAusgerichtetSIB - *Variante A*

In *Variante A* wird die Parallelausrichtung durch das Finden einer minimalen Distanz zum Hindernis erreicht. Dafür dreht sich der Quadrokopter so lange rechts herum bis sich der Abstand nach links nicht mehr verringert. In der Praxis kann dieses Vorgehen lediglich prüfen, ob der Abstand bereits wieder nennenswert größer ist. Bei diesem Vorgehen kommt es jedoch zu Ungenauigkeiten und damit einhergehend zu einer eher schlechten Parallelausrichtung. Entweder wird fälschlicherweise zu früh von einem minimalen Abstand ausgegangen, beispielsweise aufgrund von Messungenauigkeiten. Oder es wird zu spät erkannt, dass der minimale Abstand bereits erreicht wurde und es kommt zu einer Überdrehung. Um besser mit den Messungenauigkeiten umgehen zu können existiert daher auch eine *Variante B* dieses SIBs.



$$z_1 = x + d_f * \sin(90° - \arctan(\tfrac{d_l}{d_f}))$$

$$z_2 = y + d_f * \cos(90° - \arctan(\tfrac{d_l}{d_f}))$$

$$Orientierung = \arctan(\tfrac{g_2 - z_2}{g_1 - z_1})$$

Abbildung 8.7: Mathematische Berechnung zur Parallelausrichtung - *Variante B*

CheckParallelAusgerichtetSIB - *Variante B*

Bei *Variante B* wird die Parallelausrichtung durch Angleichung der Quadrokopterorientierung an die Orientierung des Hindernisses im Raum erreicht. Die Orientierung des Quadrokopters ist bekannt. Die Orientierung des Hindernisses im Raum wird benötigt. Diese kann mit einfachen Mitteln berechnet werden, sobald zwei Punkte, die auf dem Hindernis liegen, bekannt sind. Dazu wird der Quadrokopter gedreht bis sowohl der vordere als auch der linke Sensor einen Abstand zum Hindernis liefern. Abbildung 8.7 zeigt die gegebene Situation. Hierbei stellt d_f den Abstand nach vorn und d_l den Abstand nach links zum Hindernis dar. Mit Hilfe der Koordinaten des Quadrokopters (x, y), sowie den Abständen d_f und d_l können die beiden Punkte (g_1, g_2) und (z_1, z_2) berechnet werden. Für z_1 ergibt sich $z_1 = x + d_f * \sin(90° - \arctan(\frac{d_l}{d_f}))$ und für z_2 ergibt sich $z_2 = y + d_f * \cos(90° - \arctan(\frac{d_l}{d_f}))$. Die Koordinaten g_1 und g_2 des zweiten Punktes werden analog dazu berechnet.

Mit Hilfe der zwei Punkte kann nun die Orientierung des Hindernisses im Raum berechnet werden. Die Orientierung wird mittels $\arctan(\frac{g_2 - z_2}{g_1 - z_1})$ berechnet. Zur Parallelausrichtung muss die Orientierung des Quadrokopters lediglich an die berechnete Orientierung angepasst werden.

CheckParallelDistanceSIB

Das CheckParallelDistanceSIB prüft den Abstand nach links und teilt die Messung in vier Intervalle ein: *OK*, *TOOFAR*, *TOOCLOSE*, *NOOBSTACLE*.

CheckTargetInSightSIB

Zur Zielausrichtung wird mit Hilfe der Zielkoordinaten die Orientierung berechnet, die erreicht werden muss. Diese Berechnung kann in vier verschiedene Fälle unterteilt werden, die in Abb. 8.8 dargestellt sind. Wenn sich das Ziel im linken vorderen Sektor befindet (s. Abb. 8.8(a)), ergibt sich $\alpha = 180° - \arctan(\frac{|GK|}{|AK|})$. Im rechten vorderen Sektor (s. Abb. 8.8(b)) ergibt sich $\alpha = \arctan(\frac{|GK|}{|AK|})$. In Abb. 8.8(c) befindet sich das Ziel im linken hinteren Sektor und für α ergibt sich $\alpha = -\arctan(\frac{|GK|}{|AK|})$. Im letzten Fall (s. Abb. 8.8(d)) befindet sich das Ziel im rechten hinteren Sektor und für α ergibt sich $\alpha = -180° + \arctan(\frac{|GK|}{|AK|})$. Mit Hilfe des Ergebnisses von α kann die aktuelle Orientierung des Quadrokopters durch Rechts- oder Linksdrehung an α angepasst werden.

FrontCollisionAvoidedSIB

Mit Hilfe dieses SIBs kann festgestellt werden, ob das sichere Steuerungsmodell gerade aktiv eine Kollision vermeidet. Dazu wird geprüft, ob die Controller-Signale zur Vorwärtsbewegung durchgereicht werden oder ob diese zunächst verändert werden. Wenn die Vorwärtsbewegung durch die Sicherheitsaspekte verändert wurden, dann wurde eine Kollision verhindert.

LeftCollisionAvoidedSIB

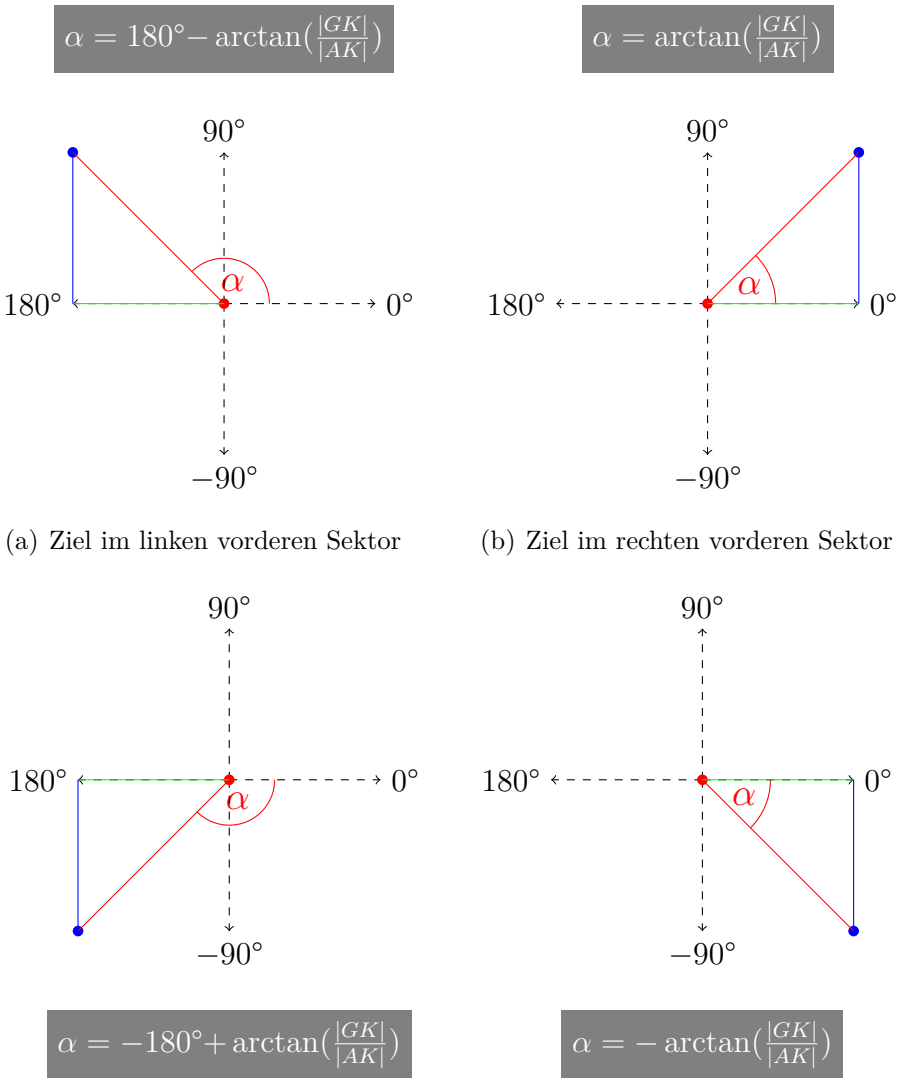
Dieses SIB prüft genau wie das FrontCollisionAvoidedSIB ob eine Kollision verhindert wurde. Allerdings wird hierbei eine Kollisionsvermeidung zur linken Seite erkannt.

ReachedPositionSIB

Das ReachedPositionSIB dient dazu, die Zielerreichung des Quadrokopters festzustellen. Hierfür wird geprüft, ob die x- und y-Koordinate des Quadrokopters mit den Zielkoordinaten übereinstimmen. Bei dieser Übereinstimmung wird eine Toleranzgrenze genutzt.

SetControllerMovementsSIB

Mit Hilfe dieses SIBs werden alle Controller-Signale auf einmal gesetzt. Dadurch wird verhindert, dass übrig gebliebene Controller-Signale aus vorangegangen Modi nicht auf 0 gesetzt werden.



(a) Ziel im linken vorderen Sektor

(b) Ziel im rechten vorderen Sektor

(c) Ziel im linken hinteren Sektor

(d) Ziel im rechten hinteren Sektor

Abbildung 8.8: Mathematische Berechnung zur Zielausrichtung

SwitchModusSIB

Dieses SIB wird im Hauptmodell des Missionsmodells genutzt und dient zum Wechseln zwischen den verschiedenen Modi. Abhängig von einer String-Variablen aus dem Context, wird der jeweils zu dem String passende Modus ausgewählt. Das Umschalten selber erfolgt innerhalb der einzelnen Modi durch das Verändern der String-Variablen.

MoveToPositionSib

Das MoveToPositionSib stellt eine weitere Möglichkeit dar, den Quadrokopter zu einem bestimmten Ziel zu fliegen. Es wird davon ausgegangen, dass sich keine Hindernisse zwischen dem Quadrokopter und dem Ziel befinden. Ausgehend von einem Regelfehler e der die Differenz des Weges zwischen Quadrokopter und dem Ziel bildet, wurde ein PID-Regler (siehe Kapitel Regelungstechnik) implementiert. Dieser ermittelt ausgehend vom Regelfehler e eine Stellgröße für den Quadrokopter. Da aber der Quadrokopter um einen bestimmten Winkel α gedreht ist, muss dieser Winkel in die Berechnung der Stellkräfte mit einfließen. Ermöglicht wird dies durch die Multiplikation einer Rotationsmatrix R auf die errechneten Stellgrößen (x, y):

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\alpha)x & -\sin(\alpha)y \\ \sin(\alpha)x & \cos(\alpha)y \end{pmatrix}$$

Die Berechnung der Stellgrößen (x', y') ermöglicht es, unabhängig von einem Drehwinkel α zu einer bestimmten Koordinate zu fliegen.

8.2.3 Szenarien

In diesem Kapitel werden einige Missionen vorgestellt um die Missionsmodelle zu verdeutlichen. Dabei werden drei grundlegende Missionen betrachtet. Die erste Mission dient dazu, den Vorwärtsflug und die Zielausrichtung zu verdeutlichen. Dabei soll der Quadrokopter auf direktem Wege zum Ziel finden. In den nachfolgenden beiden Missionen werden Szenarien mit Hindernissen vorgestellt.

Szenario 1 - Kein Hindernis

Bei dieser Mission ist es das Ziel, dass der Quadrokopter eigenständig startet und ein vom Benutzer gegebenes Ziel anfliegt. In Abbildung 8.9 ist zu sehen, dass der Quadrokopter sich bevor er das Ziel anfliegt zunächst einmal zum Ziel hin ausrichten muss. Wenn dies erfolgt ist, wird vom Zielausrichtungsmodus in den Vorwärtsflugmodus gewechselt. Dieser ermöglicht, dass die Drohne sich in Richtung Ziel bewegt und schließlich erreicht.

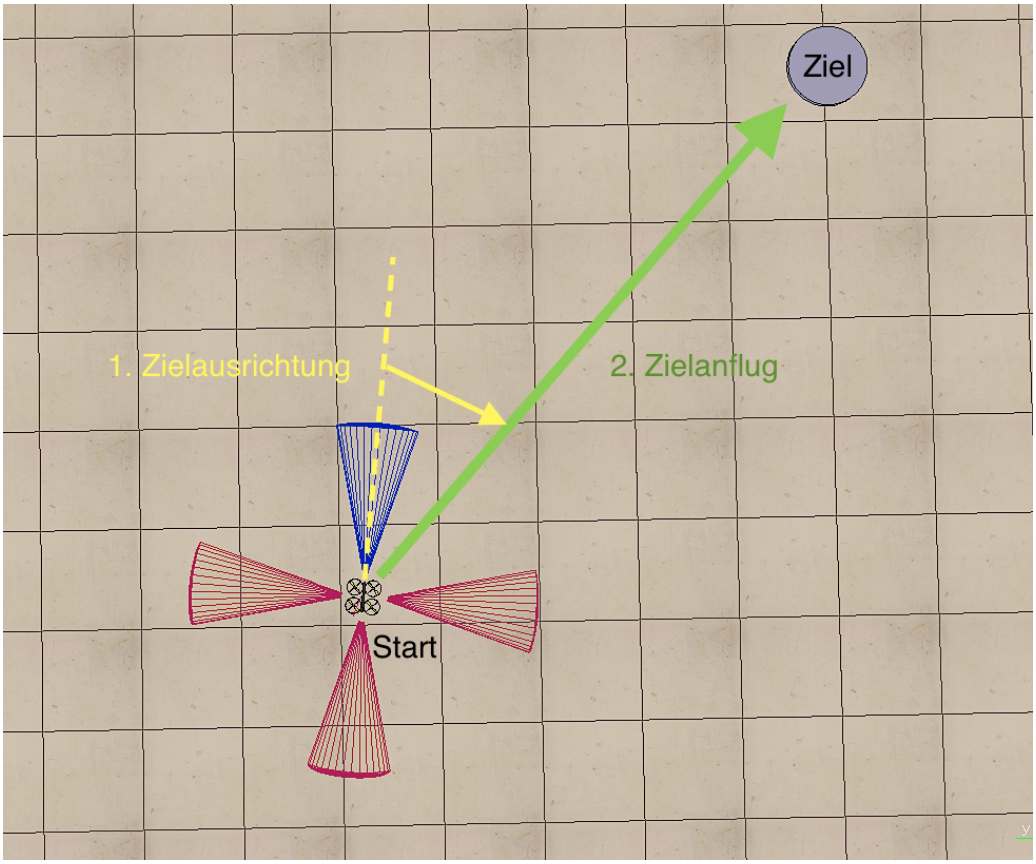


Abbildung 8.9: Abbildung einer Mission ohne Hindernis. Der gelbe Pfeil deutet die Zielausrichtung und der grüne Pfeil deutet den Zielanflug an.

Szenario 2 - Einfaches Hindernis

Wie in Szenario 1 ist das Ziel der Mission, das Erreichen eines vom Benutzer festgelegten Zielpunkts. Bei diesem Szenario muss der Quadrokopter allerdings ein einfaches Hindernis, in Form einer Wand, umfliegen. Abbildung 8.10 zeigt die einzelnen Manöver, die der Quadrokopter dabei ausführt. Zunächst führt der Quadrokopter eine Zielausrichtung aus und beginnt dann den Zielanflug geradewegs auf das Ziel zu. Wenn der Quadrokopter ein Hindernis voraus entdeckt, so umfliegt er dieses Hindernis rechts herum. Dazu führt er eine Parallelausrichtung zu dem

Hindernis aus. Nach der Parallelausrichtung fliegt der Quadrokopter mit einem gewissen Toleranzbereich parallel an der Wand entlang, bis er am Ende der Wand angelangt ist. Daraufhin fliegt er noch ein Stück weiter gerade aus und fliegt anschließend eine Linkskurve, um die Ecke der Wand zu überwinden. Im Anschluss kann sich der Quadrokopter wieder zum Ziel ausrichten und seinen Zielanflug fortsetzen.

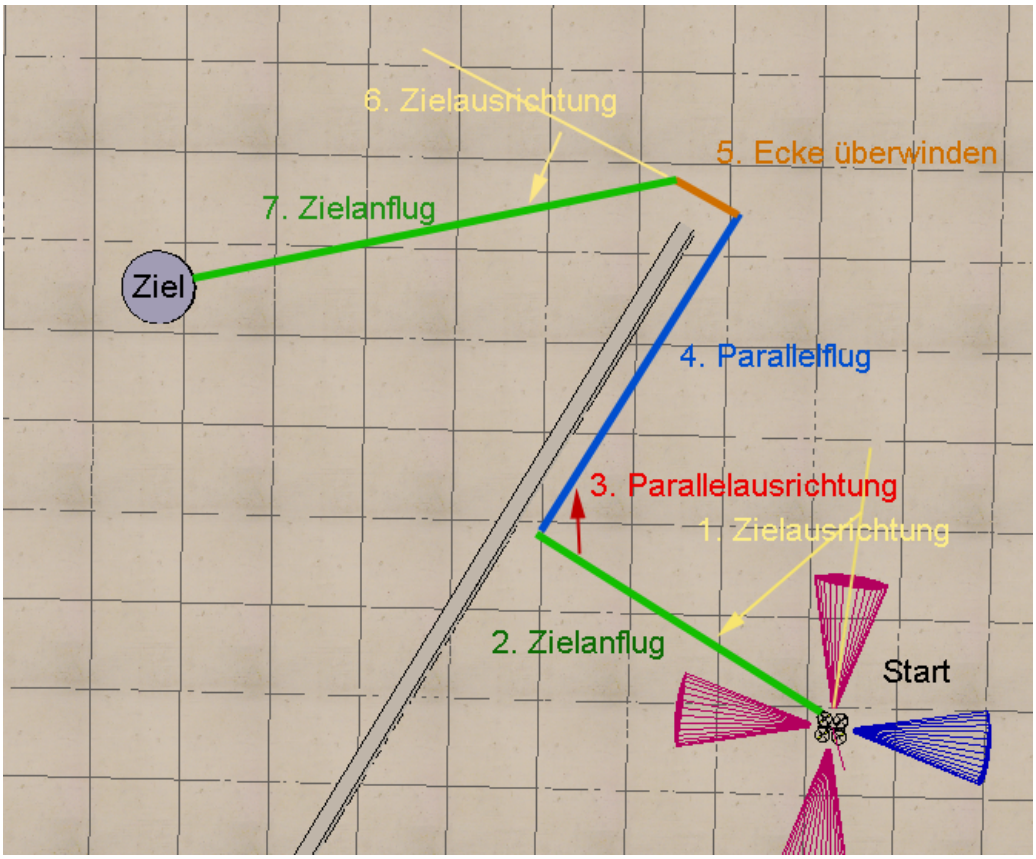


Abbildung 8.10: Abbildung einer Mission mit einem einfachen Hindernis.

Szenario 3 - Komplexes Hindernis

In Szenario 3 muss der Quadrokopter ein komplexes Hindernis überwinden, bestehend aus einer Wand mit einer inneren und einer äußeren Ecke. Abbildung 8.11 zeigt die Flugbewegungen des Quadrokopters um das Ziel zu umfliegen. Der Quadrokopter richtet sich zunächst zum Ziel aus und fliegt gerade darauf zu. Entdeckt er voraus ein Hindernis, so richtet er sich wieder parallel dazu aus und fliegt an dem Hindernis im Parallelflugmodus entlang. Stößt er im Parallelflugmodus auf ein Hindernis in Form einer inneren Ecke, so dreht sich der Quadrokopter solange nach rechts, bis der vordere Sensor kein Hindernis mehr erkennt. Nun kann der Quadrokopter im Parallelflugmodus an der zweiten Wand entlang fliegen. Die weiteren Schritte zum überwinden der äußeren Ecke sind analog zu Szenario 1.

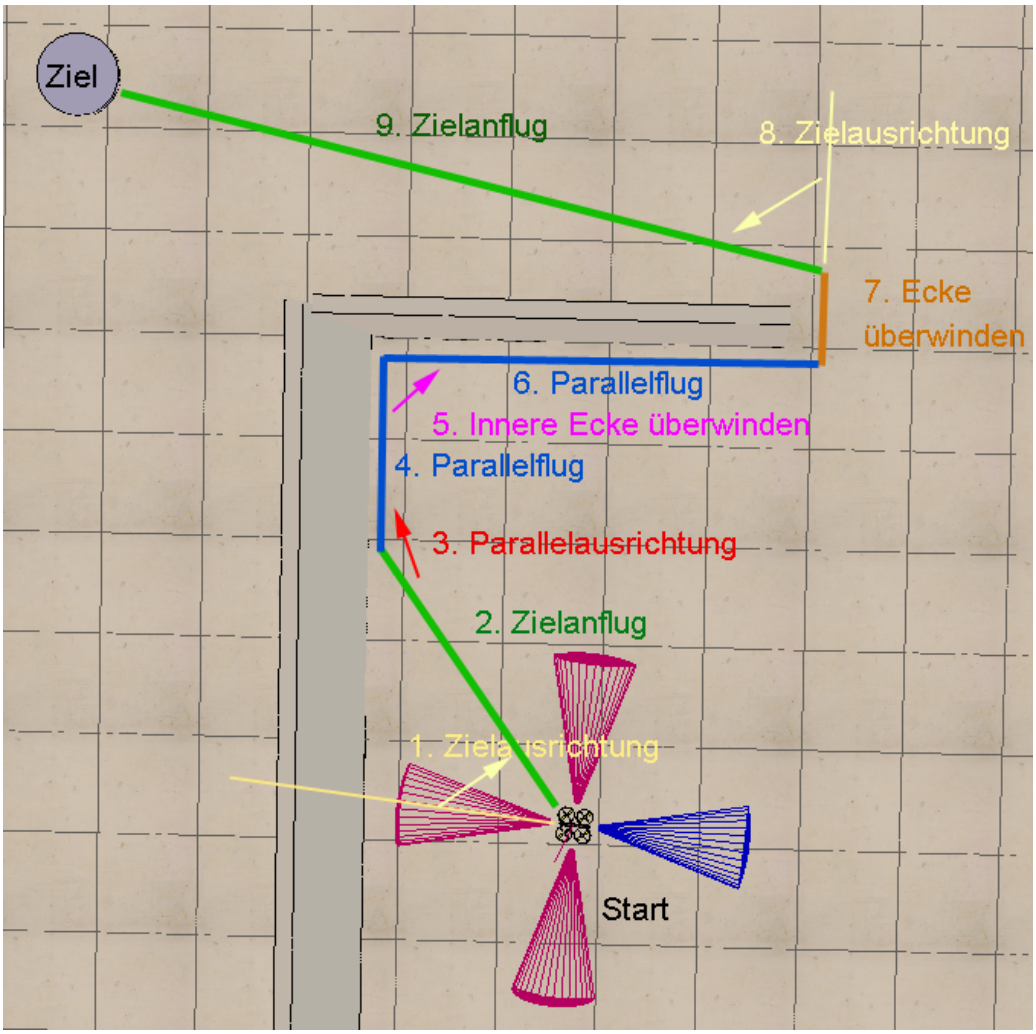


Abbildung 8.11: Abbildung einer Mission mit einem komplexen Hindernis.

KAPITEL 9

Aspektweber

Dieses Kapitel erläutert die theoretischen Grundlagen und den Aufbau des Aspektwebers, der zur Realisierung von Aspektorientierung auf Graphen entwickelt wurde. Anfänglich wird eine allgemeines Modell für die aspektorientierte Programmierung auf Graphen vorgestellt, aus dem dann eine formale Spezifikation hergeleitet wird. Daraufhin wird die tatsächliche Implementierung des Webers und deren Evolution diskutiert; zum Schluss wird das Genesys-Framework vorgestellt, auf das die finale Fassung des Aspektwebers aufsetzt.

9.1 Aspektorientierung auf Modellebene

Die ursprüngliche Konzeption der Aspektorienterung in [Kic+97] diskutiert zwei potenzielle „Spielarten“ der Aspektorientierung: die Transformation des Flussgraphen eines Programmes in Common Lisp und die Modifikation von Methoden in Java. Der zweite Fall modifiziert die Eingaben von Methoden, um unnötige Kopieroperationen auf Objekten zu reduzieren und beschreibt den Ansatz, der auch von Software wie AspectJ [Ecl14] umgesetzt wurde. Dieser Ansatz konzentriert sich darauf, Programme, die in klassischen Programmiersprachen realisiert wurden, an „natürlichen“ Stellen mit Aspekten zu verweben. Dies sind Aufrufe von Funktionen und Methoden; Konzepte, die sich nicht direkt auf die modellgetriebene Softwareentwicklung übertragen lassen. Dieser Abschnitt erläutert, wie die Konzepte angepasst werden können, um sich auf ein Graph-basiertes Modell im jABC (oder einer ähnlichen Umgebung) übertragen zu lassen.

9.1.1 Programmiersprachen

Da sich Modelle normalerweise auf einer sehr hohen Abstraktionsebene befinden und, anstatt grundlegene Funktionalität zu realisieren, die Zusammenarbeit von Komponenten auf niedrigeren Abstraktionsebenen zu orchestrieren, arbeiten querschneidende Anliegen auf Modellen in der Regel auf der gleichen, hohen Ebene. Daher erscheint es nahe liegene, die Anliegen, also die eingewebten Advices, in der gleichen Programmiersprache zu realisieren wie die ursprüngliche Komponente. Im Falle des jABC sollten beide als jABC-Graph-Modell vorliegen.

Die Spezifikation, die beschreibt, wie Advice und Komponente verwebt werden sollen, kann ebenfalls als Graph realisiert werden; das Ergebnis ist eine enorme Flexibilität beim Transformieren von Graphen. Der Preis dafür ist jedoch ein hoher Aufwand beim Gestalten der Spezifikationen. Um diesen zu reduzieren, kann stattdessen ein datengetriebener Ansatz verwendet werden, bei dem eine Spezifikation in einer Sprache wie XML von einem Aspektweber interpretiert wird, um die Webeoperation durchzuführen. Da das jABC SIBs bietet, um Modelle zu modifizieren, kann der Aspektweber wiederum als jABC-Graph realisiert werden.

9.1.2 Schnittpunkte

Im Gegensatz zu Programmen in Sprachen wie Lisp und Java haben Modellgraphen keine Einheiten wie Methoden und Funktionen, die einzigen „natürlichen“ Funktionseinheiten sind Graphen und SIBs. Daher erscheint es natürlich, die aus der klassischen Aspektorientierung

bekannten Joinpoints an diesen Einheiten anzusetzen. Da durch die hohe Abstraktionsebene von Modellen bereits komplexe Logik in nur einem Graph-Modell realisiert werden kann, sind Graphen eine zu weit fassende Abstraktionsschicht, die SIBs erscheinen eher als geeignete Realisierungen für Joinpoints. Dies deckt sich mit den ursprünglichen Ansätzen der AOP: SIBs entsprechen den atomischen Komponenten, auf denen durch Aspekte querschneidende Anliegen modelliert werden.

Da SIBs einzelne Schritte im Kontrollfluss beschreiben, können sogar die klassischen „Orte“ des Aspektwebens übernommen werden:

before Während Advices, die vor einen Aufruf gewebt werden, klassisch die Eingabeparameter modifizieren, können auf der hohen Ebene der Modelle auch weitere Schritte in Form von jABC-SIBs eingewebt werden, um den Aufruf des Joinpoints vorzubereiten.

after Klassisch modifizieren Advices, die hinter einen Aufruf gewebt werden, den Rückgabewert. Auf Modellebene können auch hier zusätzliche SIBs integriert werden, um Joinpoints vorzubereiten.

around Advices, die um einen Aufruf gewebt werden, sind in der klassischen AOP sowohl für die Aufgaben von before- und after-Advices verantwortlich, können aber auch die Ausführung der Komponente verweigern (beispielsweise, um Zugriffskontrolle zu realisieren) oder wiederholen (beispielsweise für Software Transactional Memory). Modell-Advices können ähnliche Aufgaben übernehmen, indem der Joinpoint durch den Aspekt ersetzt wird, der Aspekt aber die Möglichkeit hat, das SIB des Joinpoints ein- oder mehrfach aufzurufen. Natürlich stehen einem solchen Advice auch alle Optionen der beiden anderen Typen offen.

Diese Korrespondenz zwischen klassischer Aspektorientierung und Aspektorientierung auf Graph-Modellen ermöglicht es, die Konzepte der klassischen AOP weitgehend auf die Ebene der Modelle zu übertragen. Dies genügt für eine Vielzahl von Use Cases, aber komplexe Transformationen, wie sie im ersten Use Case von [Kic+97] beschrieben wurden, sind nicht zu realisieren.

9.1.3 Komplexe Transformationen

Da in Modellen der Programmfluss naturgemäß in einer zugänglicheren Form vorliegt, als dies in den meisten Programmiersprachen der Fall ist, ist es denkbar, in diesen aggressiver einzugreifen, als es die oben beschriebenen Techniken erlauben, und gleichzeitig nicht die Verständlichkeit der Struktur zu opfern. Dazu sind zwei Ansätze denkbar:

datengetrieben Durch die Implementierung komplexerer Aspektweber ist es möglich, die Spezifikation als interpretierten Datensatz zu gestalten. Beispielsweise kann ein Tausch von SIBs — in der klassischen AOP undenkbar — durch eine zusätzliche Direktive in der Spezifikationssprache ermöglicht werden. Die Grenzen der möglichen Transformationen würden hier durch die Bibliothek der Aspektweber-Features definiert; je mehr Direktiven unterstützt werden, um so mächtiger wird die AOP. Durch die datengetriebene Natur des Systems wird die für den Benutzer spürbare Komplexität jedoch gering gehalten.

programmatisch Anstatt den Aspektweber Spezifikationen interpretieren zu lassen, können die Transformationen als „Plugins“ des Aspektwebers in einer geeigneten Programmiersprache (bspw. als jABC-Graph) realisiert werden. Der Weber dient als Framework, um die Ausführung der Transformationen zu koordinieren. Dieser Ansatz ermöglicht beliebig komplexe Transformationen, allerdings kann die resultierende Komplexität das Programm leicht unverständlich machen.

Natürlich sind auch hybride Ansätze denkbar, bei denen datengetriebene Transformationen sich mit programmatischen ergänzen. Diese hätten das Potenzial, die Vorteile beider Ansätze zu vereinen, wären allerdings auch aufwändig zu implementieren.

9.2 Formale Spezifikation

Dieses Kapitel definiert die Anforderungen für einen Graphen-Aspektweber, der „around“-Aspektweben auf jABC-Graphen durchführen kann. Dazu werden die Anforderungen an die modellierten Advices und die Spezifikation des Webers diskutiert; danach wird das gewünschte Verhalten des Webers erläutert.

9.2.1 Anforderungen

Um einen Aspekt in einen bestehenden Graphen einweben zu können, sind drei Element erforderlich: Programm, Advice und Spezifikation. Das Programm ist ein existierender jABC-Graph, der um den Advice erweitert werden soll. Der Advice ist ein weiterer jABC-Graph, der in das Programm eingewoben wird. Die Spezifikation definiert den Pointcut für den Aspekt, also die Stellen (Joinpoint), an denen der Advice in das Programm eingewebt wird.

Programm

Das Programm, welches vom Aspektweber modifiziert wird, ist ein regulärer jABC-Graph. Um das Aspektweben universell möglich zu machen, werden keine spezifischen Anforderungen an das Programm gestellt. Es ist jedoch denkbar, jABC-Annotationen zu nutzen, um Metadaten verfügbar zu machen, die zur Spezifikation des Pointcuts genutzt werden können.

Advice

Ein Advice wird durch einen jABC-Graphen dargestellt, der in einen bestehenden Graphen eingewebt werden kann. Der Advice ist in der Lage, diverse Operationen auszuführen, um den Ablauf des Programms zu modifizieren:

- Aufruf beliebiger SIBs im Kontext des modifizierten Graphen
- Aufruf des SIBs, um welches der Advice gewebt wird (Joinpoint); dies kann mehrfach oder nie geschehen
- Verlassen des Advice über eine Modellkante

Es ist nicht möglich, den Advice über eine Kante, die keine Modellkante ist, zu verlassen. Der Aufruf des Ziel-SIBs geschieht über ein Proxy-SIB, welches während des Webevorgangs durch das eigentliche SIB ersetzt wird. Das Proxy-SIB hat keine vordefinierten Kanten (mit Ausnahme von `OTHER`-Kanten), alle Kanten müssen vom Benutzer definiert werden. Dieses Proxy-SIB ist

zu unterscheiden vom jABC eigenen Proxy SIB. Wir haben hier ein völlig eigenes SIB definiert, dessen Name identisch zu einem anderen vorhandenen SIB ist, uns jedoch nur als Platzhalter innerhalb der Advices dient. Dies ermöglicht Flexibilität bei der Definition des Pointcuts. Um Einweben zu ermöglichen, müssen Advices ein Start-SIB definieren.

`OTHER`-Kanten sind ausgehende Kanten des Proxy-SIBs mit dem speziellen Label `OTHER`. Sollte diese Kante des Proxy-SIBs tatsächlich definiert sein, so erzeugt der Weber eine Kopie des restlichen Graphen für jede ausgehende Kante des Joinpoints, die nicht gesondert behandelt wird. Wird der Advice über die Modellkante `OTHER` verlassen, so wird stattdessen die entsprechende ausgehende Kante des Joinpoints als Modellbranch genutzt.

Spezifikation

Die Spezifikation definiert den Pointcut des Advices, also die SIBs, um die der Advice gewebt wird. Die Spezifikation wird als XML-Dokument (Abb. 9.1) bereitgestellt, in welchem via String-Matching die Joinpoints spezifiziert werden. Ein weiteres Dokument (Abb. 9.2) definiert die Tupel aus Advice und Pointcut, d. h. welche Advices an welchen Pointcuts eingewoben werden müssen.

Pointcuts können über die Namen oder die Taxonomie von SIBs definiert werden, als Methoden des String-Matchings werden reguläre Ausdrücke und Glob-Syntax (bekannt aus Shells und Suchfunktionen) unterstützt. Die Spezifikation selber koppelt Advices und Pointcuts. Die Advices werden in Reihenfolge der Deklaration angewendet; ein Advice kann mehreren Pointcuts zugeordnet werden, um einfache Vereinigung von Pointcuts zu erlauben.

Zusätzlich kann die Spezifikation definieren, wie Modellbranches des Aspektes behandelt werden sollen, die keine gültigen Kanten des Joinpoints sind. Mögliche Strategien sind:

1. Die Modellkante entfernen; wenn die Kante gewählt wird, wird ein Fehler erzeugt
2. Die Modellkante als eine neue Modellkante des Programmes verwenden
3. Während des Webens einen Fehler auslösen

9.2.2 Semantik

Der Aspektweber muss eine bestimme Semantik einhalten, um Advices und Programme korrekt zu verweben. Die Semantik definiert, wie ein Advice-Graph in einen Programmgraphen integriert wird und wie die Spezifikationsdokumente verarbeitet werden müssen.

```
<pointcut name="move">
  <includes>
    <label regex="Move.*"/>
    <taxonomy glob="udo.edu.pg.ages.sibs.move.*"/>
  </includes>
  <excludes>
    <label regex=".*Safe"/>
  </excludes>
</pointcut>
```

Abbildung 9.1: Pointcut-Spezifikation

```
<specification>
  <advice name="emergency-landing">
    <pointcut name="read-input" unmatched-branch="ERROR">
  </advice>
  <advice name="safe-movement">
    <pointcut name="move" unmatched-branch="UNDEFINE">
  </advice>
</specification>
```

Abbildung 9.2: Pointcut-Advice-Kopplung

Graphenintegration

Gegeben sei ein Advicegraph $\mathcal{A}$ mit SIBs $a_1, \ldots, a_n$, Start-SIB a'. $a_{p_1}, \ldots, a_{p_m}$ sind alle Proxy-SIBs innerhalb des Graphen. Gegeben sei auch ein Programmgraph $\mathcal{P}$ mit Joinpoints $p_{j_1}, \ldots, a_{j_i}$. Aus Gründen der Einfachheit wird angenommen, dass nur ein Joinpoint p_j existiert; der Weber wiederholt das gegebene Verfahren im Allgemeinen für jeden Joinpoint. Um den Advice in $\mathcal{A}$ um p_j zu weben, geht der Weber wie folgt vor:

1. $a_1, \ldots, a_n$ und die Kanten zwischen ihnen werden von $\mathcal{A}$ nach $\mathcal{P}$ kopiert.
2. Falls $a \in a_{p_1}, \ldots, a_{p_m}$ eine `OTHER`-Kante hat, wird für jeden definierten Branch b von p_j, der nicht eine ausgehende Kante von a ist, eine neuer Subgraph erzeugt, der alle erreichbaren Knoten als Kopie enthält. Kann ein SIB $\in a_{p_1}, \ldots, a_{p_m}$ erreicht werden, wird dieses nicht kopiert, sondern das Original genutzt. Alle `OTHER`-Modellkanten von dabei kopierten SIBs werden in b umbenannt.
3. $a_{p_1}, \ldots, a_{p_m}$ werden durch Kopien von s ersetzt.
4. Alle ausgehenden Kanten von $a_{p_1}, \ldots, a_{p_m}$, die keinen gültigen Branch von p_j definieren, werden gelöscht.
5. Alle nach p_j führenden Kanten werden durch Kanten nach a' ersetzt.
6. Alle Modellkanten von $\mathcal{A}$, die einen in $\mathcal{P}$ existierenden Branch von p_j definieren, werden durch Kanten zu dessen Ziel ersetzt.
7. Alle Modellkanten von $\mathcal{A}$, die gleich zu Modellkanten von $\mathcal{P}$ sind, bleiben erhalten.
8. Alle Modellkanten von $\mathcal{A}$, die keine Modellkanten von $\mathcal{P}$ sind,...
   - ... werden, falls der `unmatched-branch`-Modus `UNDEFINE` ist, gelöscht.
   - ... werden, falls der `unmatched-branch`-Modus `KEEP` ist, beibehalten.
   - ... lösen, falls der `unmatched-branch`-Modus `ERROR` oder nicht gesetzt ist, einen Fehler aus.
9. p_j wird mit allen ausgehenden Kanten gelöscht.
10. Falls p_j kein Start-SIB war, wird das Start-SIB-Attribut von a' entfernt.

Dieses Vorgehen ersetzt p_j durch $\mathcal{A}$ und versucht, die Modellkanten von $\mathcal{A}$ intelligent in $\mathcal{P}$ zu integrieren, ohne den Vertrag von $\mathcal{P}$ zu beeinflussen. Diverse Optimierungen, wie das Löschen nicht erreichbarer SIBs, sind möglich, beeinflussen aber die Semantik des verwebten Graphen nicht. Der Aspektweber muss Makro-SIBs berücksichtigen, um auch in Untermodellen den Advice zu verweben.

Spezifikation

Der Aspektweber verarbeitet genau ein Spezifikationsdokument und genau einen Graphen. Dabei iteriert er in Reihenfolge durch die in der Spezifikation genannten Advicses. Für jeden Advice iteriert der Weber über alle SIBs im Programmgraphen, die nicht im gleichen Schritt hinzugefügt wurden (um Endlosschleifen zu vermeiden). Wenn für ein SIB ein dem Advice zugeordneter Pointcut existiert, so dass

- das SIB mindestens einen `<include />` erfüllt und
- das SIB keinen `<exclude />` erfüllt,

dann wird der Advice um das SIB gewoben.

Das Vorgehen stellt sicher, dass die Reihenfolge der Webeoperationen wohldefiniert ist, keine Endlosschleifen entstehen und die Komposition von Pointcuts möglich ist.

9.2.3 Zusammenfassung

Diese Spezifikation stellt einen Entwurf für einen klassischen Aspektweber für das jABC vor. Da sich „before“- und „after“-Advices durch „around“-Advices modellieren lassen, die vor bzw. nach dem SIB keine weiteren Schritte durchführen, lassen sich alle Konzepte der klassischen AOP durch den Weber modellieren.

9.3 Umsetzung des Webers

In den vorangehenden Abschnitten wurde zunächst ein Überblick der Aspektorientierung auf Modellebene gegeben. Anschließend ließ sich daraus eine formale Spezifikation eines Aspektwebers ableiten. Im Folgenden gilt es nun zu erläutern, auf welche Weise die erhaltene Spezifikation im Zuge der Projektgruppe praktisch umgesetzt, getestet und angewandt wurde. Als abschließendes Ziel ist hierbei vorweg die Umsetzung als jABC-Modell zu nennen, sodass das Weben über dieses in einer Model-To-Model-Transformation erfolgt.

Da eine direkte Umsetzung als Modell umständlich erschien, erfolgte eine zielgerichtete Implementierung in drei aufeinander aufbauenden Schritten. Zunächst wurde aus der theoretische Auslegung des Aspektwebers in der Umgebung von BaseX mit Hilfe der Techniken XQuery und XPath ein Prototyp gewonnen, um die Grundfunktionalität zu testen. Eine knappe Erläuterung hierzu findet sich in Abschnitt 9.3.1. Auf Basis dieses Prototyps wurde anschließend eine Version des Webers in Java implementiert. Diese erweitert den XQuery-Prototypen und bildet alle spezifizierten Funktionalitäten ab. Die Implementierung wurde an dieser Stelle auf Java portiert, da somit eine Verwendung von Teilen das Quelltextes bei der Umsetzung als Modell gewährleistet werden konnte. Details zu dem daraus resultierenden Java-Weber finden sich in Abschnitt 9.3.2. Abschließend wurden die Java-Umsetzung des Aspektwebers als jABC-Modell abgebildet, um das eingangs erwähnt Ziel zu erreichen. Die verwendeten SIBs wurden dabei teilweise der jABC-Bibliothek entnommen und teilweise auf Basis des Java-Webers neu erstellt. Eine genaue Erläuterung der Modellierung findet sich in Abschnitt 9.3.3.

9.3.1 Rapid Prototyping mittels XQuery

Die erste Entwicklung eines Aspektwebers entstand aus dem XML-Format, welches jABC als Ausgabe für Modelle anbietet. Da der Aspektweber eine M2M-Transformation durchführt um aus dem Eingabe Modell ein neues Modell mit eingewobenen Aspekten, war die Idee eine XML-Transformation auf dem Ausgabe-Format von jABC durchzuführen. Dadurch entstand der erste native Aspektweber, wobei für die Transformation XQuery und als Entwicklungsumgebung BaseX 7.7 (http://basex.org/) verwendet wurde. Die Idee war diesen Aspektweber in Java einzubinden und dadurch auch für die jABC Plattform zur verfügung zu stellen. Allerdings stellte sich schnell heraus, dass dies nicht die endgültige Lösung sein wird, da sich das Ausgabeformat in höheren jABC Versionen ändern könnte, bzw. vermutlich auch ändern wird. Dadurch wäre der Aspektweber nicht mehr mit höheren Versionen von jABC kompatibel.
Der native Aspektweber konnte jedoch einen guten Einblick geben, was beim Einweben eines Aspektes auf Programmierebene passiert und bot schon einmal die Möglichkeit in das Ergebnis des verwobenen Modells zu schauen. Der Quellcode der Transformation in XQuery konnte dann anschließend in Java übersetzt werden, die das selbe tat, allerdings die Graph-Bibliothel des jABC nutzt.
Parallel zur komplett-Implementierung des Aspektwebers in Java, wurde anhand der XQuery Implementierung des Aspektwebers ein jABC-Modell erstellt, welches nur NoOp-SIBs beinhaltete, um die Funktionalität des Webers zu visualisieren. Letztendlich wurden die NoOp-SIBs mit Funktionalität hinterlegt und das Modell wurde damit ausführbar gemacht.

9.3.2 Implementierung in Java

Nach einer prototypischen Umsetzung der Weber-Spezifikation in XQuery, wurde diese mit vollem Funktionsumfang hin zu Java portiert. Dieser Abschnitt dient nun der näheren Erläuterung der Implementierung des Java-Webers.

Dabei ist eingangs zu erwähnen, dass auch der Java-Weber ein Prototyp ist, welcher lediglich zu Testzwecken und als Code-Basis für die Umsetzung als Modell dienen sollte. Aus diesem Grund ist der Weber funktional gehalten und besteht im Grunde nur aus einer einzelnen Klasse. Hinzu kommen weiterhin nur Klassen, welche die Dateistruktur einer in Abschnitt 9.2 im Zuge der Spezifikation eingeführten Aspekt-Definitions-Datei abbilden. Tabelle 9.1 liefert eine kurze Übersicht der Klassen. Zusätzlich ist zu sagen, dass der Java-Weber im Gegensatz zum Prototypen in XQuery Gebrauch von der jABC-API macht, um Modelle zu laden, zu manipulieren und wieder zu speichern. Somit wurde die Möglichkeit gewährleistet, Teile des Quelltextes als Basis für SIBs der Modell-Umsetzung wieder zu verwenden.

Klasse	**Bedeutung**
SimpleJavaWeaver	Beinhaltet die komplette Logik des Webers basierend auf dem Vorbild der Spezifikation
Aspects	Bildet die Struktur der Aspektdefinitions-Datei ab
Pointcut	Wird von Aspects verwendet und repräsentiert einen Pointcut innerhalb der Aspekt-Definition

Tabelle 9.1: Klassen des Java-Webers

Der Spezifikation entsprechend benötigt der Weber eine Konfiguration-Datei, die definiert, an welcher Stelle welche Advices einzuweben sind. Im Zuge der Java-Umsetzung wurde diese Konfiguration mithilfe der Klassen Aspects und Pointcut programmiert und mittels XStream, welches bereits von jABC verwendet wird, als XML-Datei serialisiert. Listing 9.1 verdeutlicht die Struktur einer solchen Datei beispielhaft.

Wie in Tabelle 9.1 aufgeführt, stellt die Klasse Aspects die Basis der Aspekt-Definition dar. Diese beinhaltet eine *HashMap<String,String>*, welche Dateipfade zu Advices mit einer eindeutigen String-Id verknüpft. Weiterhin enthält Aspects zur Konfiguration der Pointcuts eine Liste von Pointcut-Objekten. Diese wiederum verknüpfen einen Advice aus der HashMap über das Attribut adviceId mit einem regulären Ausdruck, der angibt, an welchen Stellen der referenzierte Advice einzufügen ist.

```
<edu.udo.pg.ages.weaver.domain.Aspects>
  <advices>
    <entry>
      <string>stringAdviceId</string>
      <string>../path/to/adviceModelFile.xml</string>
    </entry>
    ...
  </advices>
  <pointcuts class="linked-list">
    <edu.udo.pg.ages.weaver.domain.Pointcut>
      <adviceId>stringAdviceId</adviceId>
      <matches>RegularExpression.+</matches>
    </edu.udo.pg.ages.weaver.domain.Pointcut>
    ...
  </pointcuts>
</edu.udo.pg.ages.weaver.domain.Aspects>
```

Listing 9.1: Beispiel einer Aspekt-Definition

```
File baseModel = new File( "../path/to/baseModel.xml" );
File outputFile = new File( "../path/to/outputFile.xml" );
File aspectsFile = new File( "../path/to/aspectsDefinition.xml" );

SimpleJavaWeaver weaver =
   new SimpleJavaWeaver( baseModel , aspectsFile , outputFile );
weaver.weave();
```

Listing 9.2: Beispiel der Verwendung des Java-Webers

Die Verwendung des Java-Webers gestaltet sich einfach und wird in Listing 9.2 angeführt. Der Aufruf benötigt erwartungsgemäß drei Dateien. Zum einen das Modell, welches um Aspekte erweitert werden soll, zum anderen eine Datei, in welche das gewobene Modell gespeichert werden soll und letztlich die Aspekt-Definition in Form einer im voraus exemplarisch erläuterten XML-Datei. Der Start des Webe-Prozesses erfolgt abschließend über die Methode *weave*. Im Folgenden wird nun in Anlehnung an die Spezifikation grob beschrieben, wie sich der Weber in Java verhält.

Nachdem mithilfe der jABC-API das übergebene Basis-Modell und mittels XStream die Aspekt-Definitions-Datei geladen wurde, iteriert der Java-Weber über die definierten Aspekte. Für jeden Aspekt werden dabei im geladenen Modell passende Stellen gesucht und im Falle eines Fundes als Joinpoint markiert. Anschließend werden die Joinpoints durchlaufen und die Advices jeweils über eine Funktion *replaceSibWithAdvice* eingefügt. Letztlich wird das geänderte Basis-

Modell in die Zieldatei geschrieben. Die Methode *replaceSibWithAdvice* verhält sich wie folgt gemäß der Spezifikation. Zunächst werden die Kanten vom zu ersetzenden SIB zu dem Start-SIB des Advices umgelenkt und das Start-SIB gegebenen Falls angepasst. Anschließend werden Proxy-SIBs, falls vorhanden, durch ein SIB vom Typ des zu ersetzenden SIBs ausgetauscht und korrekt verknüpft. Daraufhin erfolgt die Behandlung von Other-Kanten im Vergleich mit dem Basis-Modell. Der Java-Weber geht an dieser Stelle rekursiv vor und klont eine betrachtet Other-Kante samt weiterführender Knoten und Kanten für jede gefunden Abweichung im Basis-Modell. Nachfolgend werden die Modellkanten des Advices wie spezifiziert betrachtet, sodass der Advice korrekt mit den ausgehenden Kanten des zu ersetzenden SIBs verbunden wird. Abschließend erfolgt über die Methode *cleanUpModel* einen Bereinigung des Models um verwaiste Kanten und SIBs, welche vorher nicht beachtet wurden, da ansonsten Probleme beim Laden der Modelle in jABC entstehen.

9.3.3 Modellierung

Nach einer prototypischen Umsetzung, zunächst in XQuery und anschließend in Java, wurde der Aspektweber letztlich in einem jABC-Modell umgesetzt. Dessen Beschaffenheit wird im Folgenden näher erläutert. Vorab ist noch zu erwähnen, dass bei der Konstruktion SIBs aus der jABC Graph-SIB-Bibliothek, der Basis-Bibliothek und zusätzlich auf Basis des Java-Webers erstellte SIBs verwendet wurden. Weiterhin wurde das Modell in Anlehnung an den Java-Weber in drei Modelle aufgeteilt. Das erste Modell (Abbildung 9.3) ist lediglich für die Initialisierung zuständig und bindet das zweite Modell (Abbildung 9.4) als Submodell ein. Dieses durchläuft die definierten Aspekt und inkludiert das dritte Modell (Abbildung 9.5), welches die Funktionalität der Methode *replaceSibWithAdvice* (s. Abschnitt 9.3.2) bereitstellt. Die folgenden Absätze beschreiben den modellierten Weber nun anhand der Abbildungen.

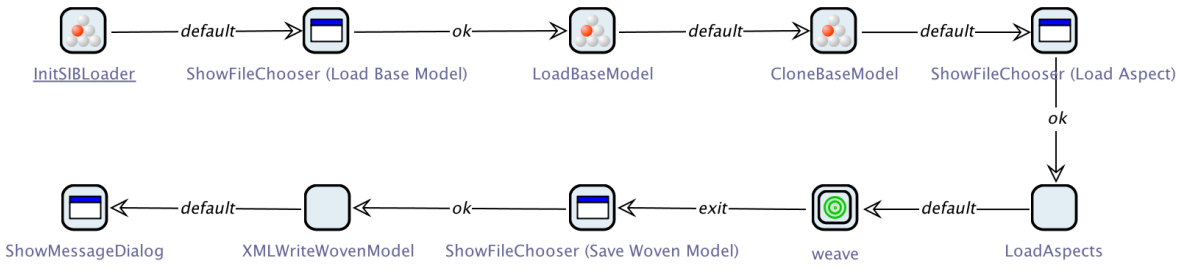


Abbildung 9.3: Das Initialisierungs-Modell des Webers

Wie eingehend erwähnt, übernimmt das erste Modell (s. Abbildung 9.3) des Aspektwebers lediglich initialisierende Aufgaben. Dabei wird zunächst das SIB *InitSIBLoader* verwendet, welches nötig ist, um nachfolgend mit Hilfe der Graph-SIB-Library Modelle zu laden und zu manipulieren. Im Anschluss daran wird über ein *ShowFileChooser*-SIB dazu aufgefordert ein Basis-Modell für den Prozess des Webens auszuwählen. Dieses wird daraufhin mit Hilfe der folgenden zwei SIBs in den Ausführungskontext geladen und zur Bearbeitung geklont. Im nächsten Schritt wird erneut *ShowFileChooser* verwendet, um den Benutzer die Aspekt-Definition auswählen zu lassen. Diese wird anschließend mit dem aus dem Java-Weber extrahierten SIB *LoadAspects* für eine spätere Nutzung in den Kontext geladen. Der nächste Schritt im Prozess ist der Aufruf eines Submodells, in diesem Fall des zweiten Modells (s. Abbildung 9.4). Ist dieser Schrittdurchlaufen, ist das Modell gewoben und muss lediglich noch gespeichert werden. Für das Auswählen der Zieldatei wird ein weiteres *ShowFileChooser*-SIB verwendet. Die Schreib-Operation wird von *XMLWriteWovenModel* durchgeführt. Dieses SIB wurde ebenfalls dem Java-Weber entnommen und nutzt die jABC-API, um die Zieldatei zu füllen. Abschließend zeigt *ShowMessageDialog* als Indikator für den Erfolg eine Meldung an. Es sei an dieser Stelle noch gesagt, dass das erste

Teilmodell in mehreren Varianten existiert. So gibt es beispielsweise ein zweite Version, welche auf Datei-Dialoge verzichtet und die benötigten drei Pfade als Modellparameter erhält.

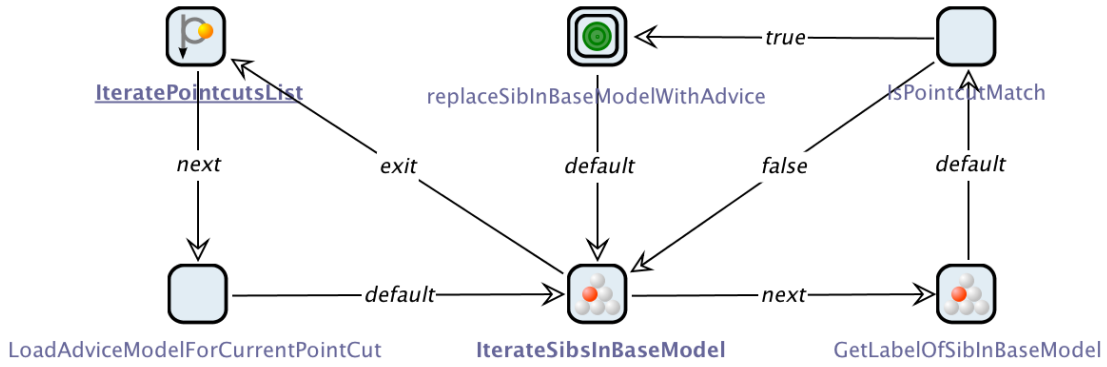


Abbildung 9.4: Die Pointcut-Iteration des Webers

Das zweite Modell (s. Abbildung 9.4) des Aspektwebers besteht im Grunde nur aus zwei geschachtelten Schleifen, in welchen die geladenen Aspekte sowie die SIBs des Basis-Modells durchlaufen werden, um Joinpoints zu finden und ein sofortiges Einsetzen eines Advices anzustoßen. Dafür wird zunächst das SIB *IterateElements*, hier dem Zweck entsprechend İterate-PointcutList"genannt, verwendet. Die passende Liste wurde durch *LoadAspects* in Abbildung 9.3 in den Ausführungskontext gelegt. Für jeden einzelnen Aspekt bzw. Pointcut wird über das SIB *LoadAdviceModel* der definierte Advice geladen und im den Kontext hinterlegt. Anschließend werden in der zweiten Schleife mit Hilfe des SIBs *IterateSibsInModel* alle SIBs des Basis-Models betrachtet. Dabei wird zunächst mit *GetSIBLabel* die Bezeichnung des aktuellen SIBs in den Kontext gespeichert. Passt diese zu dem regulären Ausdruck des aktuell betrachteten Pointcuts, ist ein Joinpoint gefunden. *IsPointcutMatch* prüft dies anhand der zwei Werte aus dem Kontext mittels der String-Funktion *matches* in Java. Ist diese Bedingung erfüllt, wird das dritte Submodell des Webers eingebunden. Ansonsten wird umgehend das nächste SIB des Basis-Modells untersucht.

Das dritte Modell (s. Abbildung 9.5) ist der Kern des Webers. Es beschreibt, was passiert, wenn der Joinpoint im Modell erreicht wurde. An dieser Stelle wird in dem SIB *CopyModelSIB* der passende Advice in das Base-Model kopiert. Das heißt der Advice liegt nun unabhängig vom Base-Model in eben diesem drin. Im nächsten Schritt *IsSibToReplaceAStartSiB?* wird überprüft, ob das gefundene Joinpoint-SIB ein Start-SIB ist. Ist dies der Fall muss das das erste SIB des Advice zu einem Start-SIB gemacht werden, bzw. die Start-Markierung des Start-SIBs des Advice darf hier nicht entfernt werden. Ist das Joinpoint-SIB kein Start-SIB, muss die Start-Markierung des Start-SIBs im Advice entfernt werden. Diese beiden Fälle werden in den

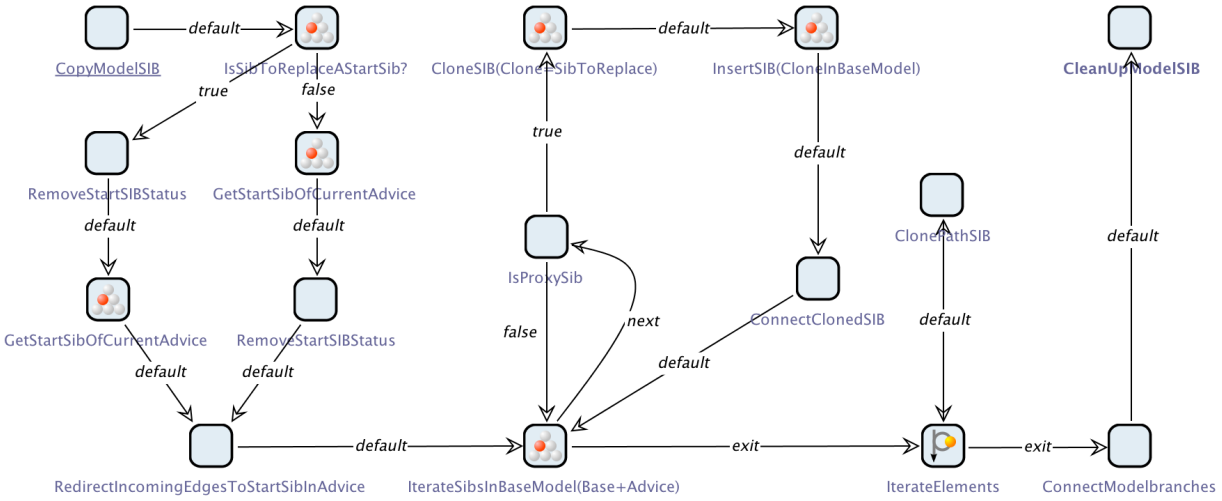


Abbildung 9.5: Das Modell zum Einfügen eines Advices

beiden ausgehenden Branches des SIBs *IsSibToReplaceAStartSiB?* beschrieben.
Nun müssen noch die Branches des Advice mit den SIBs des Base-Models verbunden werden, was in dem folgenden Teil des Modells ab dem SIB *RedirectIncommingEdgesToStartSibInAdvice* geschieht.
Zunächst werden in dem genannten SIB *RedirectIncommingEdgesToStartSibInAdvice* alle eingehenden Branches des Joinpoint-SIBs mit dem Start-SIB des Advice verbunden. Daraufhin folgt die Ersetzung des bzw. der Proxy-SIBs im Advice mit dem Joinpoint-SIB. Es wird dazu über alle SIBs des Advice iteriert und an den Stellen, wo ein Proxy-SIB gefunden wurde, das Jointpoint-SIB kopiert, eingefügt und die Kanten neu verbunden. Das Verbinden der Kanten passiert in dem SIB *ConnectClonedSIB*.
Es fehlt nun nur noch der etwas komplexere Teil des Webers. In dem Teil ab dem SIB *IterateElements*, *ClonePathSIB* und *ConnectModelbranches* steckt die Logik, wie die Other-Kanten behandelt und die ausgehenden Kanten des Advice (ModelBranches des Advice) verbunden werden. Wie diese im Detail funktioniert wurde bereits im Kapitel 9.2 im Teil 9.2.1 beschrieben.
Abschließend müssen noch alle Kanten und SIBs entfernt werden, die nicht oder nur teilweise mit dem Base-Model verbunden sind, damit das Model valide ist. Dies geschieht in *CleanUpModelSIB*. Damit ist das Submodell des Webers durchlaufen und es geht eine Hierarchie-Ebene höher mit dem nächsten Joinpoint weiter.

9.4 Genesys

Das Framework Genesys [LSV10; JSM11] wurde als Add-on für das jABC entwickelt, um jABC-Modelle automatisiert in Java- und anderen Quellcode transformieren zu können. Da Genesys flexibel aufgebaut ist, ist möglich, nicht nur Model-to-Source-Transformationen, also Codegenerierung, sondern auch Model-to-Model-Transformationen mit Genesys zu realisieren. Da Aspektweben ein spezieller Fall von Model-to-Model-Transformation ist, lässt sich also auch der Aspektweber mit Genesys realisieren.

9.4.1 Komponenten

Genesys ist in mehrere Komponenten geteilt, von denen die für das AgES-Projekt relevanten hier vorgestellt werden.

Framework Die Kernkomponente von Genesys übernimmt die automatische Ausführung von Transformatoren. Diese können entweder als jABC-Modell implementiert sein oder in kompilierter Form als Java-Programm vorliegen. Im letzteren Fall wird die kompilierte Form jedoch meistens aus einem Graphen erzeugt. Auch diese Transformation kann von Genesys automatisiert werden.

jABC-Plugin Genesys kann in das jABC integriert werden, um ad hoc aus einem gerade bearbeiteten Modell Quellcode erzeugen zu können. Dazu müssen die beteiligten SIBs Metadaten bereitstellen, um die Generierung ermöglichen zu können. Für den Aspektweber sind (außer der Konfiguration des Webers) keine Metadaten nötig, so dass eine Transformation aus dem jABC-Editor möglich ist.

Maven-Plugin Genesys kann in das Buildsystem Maven (siehe Abschnitt 11.3) integriert werden, um den kombinierten Vorgang aus Aspektweben, Java-Codegenerierung und Kompilierung automatisieren zu können. Die Anforderungen sind die selben wie für das jABC-Plugin, aber hier wird keine grafische Oberfläche oder Nutzerinteraktion benötigt.

9.4.2 Integration in die AgES-Software

Genesys wurde an mehreren Stellen für das Projekt eingesetzt.

Aspektweber Da der Aspektweber als jABC-Graph implementiert wurde, konnte er mit wenigen Änderungen als Genesys-Generator genutzt werden. Dieser Generator wurde unter Benutzung von Genesys in Java-Quellcode transformiert, der dann weiter kompiliert und als eigenständige Komponente weiter benutzt werden konnte.

Modell-Aspekte Der generierte Aspektweber wurde im gleichen Projekt eingesetzt, um in den Graph-Modellen Aspekte einzuweben. Das Resultat waren jABC-Graphen, in die die Aspekte fest integriert waren, die sicheren Modelle.

sichere Modelle Die verwobenen, sicheren Modelle wurden im letzten Schritt mit den Standard-Codegeneratoren des jABC in eigenständigen Java-Code ohne Abhängigkeiten auf die jABC-Laufzeitumgebung transformiert, die die abgesicherte Funktionalität realisiert. Dieser Code kann mit dem Java-Compiler kompiliert werden.

KAPITEL 10

Sichere Modelle

In den voranstehenden Kapiteln wurden das (unsichere) Steuerungsmodell und der Aspektweber erläutert. Ziel dieser Projektgruppe war es, die modellbasierte Softwareentwicklung mit der aspektgetriebenen Entwicklung zu verbinden. Wir haben dazu modellbasiert ein Steuerungsmodell für den Quadrokopterflug entwickelt. Dieses Steuerungsmodell reagiert nur auf die Benutzereingaben mittels Controller, ohne dabei auf relevante Aspekte der Sicherheit zu achten. Die nächsten Schritte sind demnach die Definition und Modellierung von Sicherheitsaspekten sowie das Verweben dieser mit dem Steuerungsmodell. Das Ergebnis ist ein sicheres Steuerungsmodell.

10.1 Motivation und Anforderungen

Eine Vielzahl von kritischen Situationen sind beim Quadrokopterflug denkbar. Mit dem bisher betrachteten Steuerungsmodell könnte der Quadrokopter willentlich gegen ein Hindernis geflogen werden, aber auch mit unbeabsichtigten Gefahren kann das Steuerungsmodell bisher nicht umgehen, hierzu gehört z.B. der Fall, dass die Batterie des Quadrokopters nahezu leer ist. Ein sicheres Steuerungsmodell soll Aspekte der technischen Sicherheit (Safety) und der IT Sicherheit (Security) berücksichtigen. Sollte der Einsatz von (autonomen) Drohnen z.B. im militärischen Bereich in Zukunft ausgebaut werden, wird es umso wichtiger, dass deren Steuerung sicher ist. Fehlende Berücksichtigung von Sicherheitsaspekten bei technischen Entwicklungen kann teuer sein. Dies hat in diesem Jahr der Verteidigungsminister Thomas de Maiziere bewiesen. Das Verteidigungsministerium wollte im Rahmen eines Rüstungspaketes, eine Aufklärungsdrohne, Euro Hawk, entwickeln lassen. Nach gut 12 Jahren Entwicklungszeit für die Drohne wurde dieser jedoch die Zulassung für den deutschen Luftraum verweigert. Zu diesem Zeitpunkt waren mehr als 700 Millionen Euro in das Projekt geflossen. Grund dafür, dass die Luftzulassung nicht erteilt wurde, war die mangelnde Berücksichtigung von Sicherheitsaspekten für solch eine unbemannte Drohne. Insbesondere fehlte ein automatisches Ausweichverfahren. Wir haben im Rahmen dieser Projektgruppe nur einige ausgewählte Aspekte definiert und umgesetzt, da es nicht Anforderung war, das Steuerungsmodell gegen alle möglichen Risiken abzusichern, sondern lediglich Konzepte für die Verknüpfung von modellgetriebener und aspektorientierter Entwicklung zu definieren. Die von uns umgesetzten Aspekte sind:

- Kollisionserkennung
- Verlust des Controller Signals
- Sichere Landung
- Notlandung

Im Abschnitt 10.2 wird erläutert, welcher Gefahr der jeweilige Aspekt begegnen soll und wie die Umsetzung erfolgte.

10.2 Vorgesehene Aspekte und ihre Umsetzung

Dieser Abschnitt stellt die einzelnen Aspekte vor. Dazu wird der Zweck des Aspekts beschrieben, die Struktur des Advices erklärt. Der Weber bekommt für jeden Aspekt ein Modell, das so

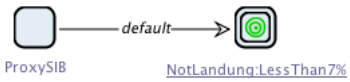


Abbildung 10.1: Aspekt Beispiel

aufgebaut ist, wie in Abbildung 10.1. Die eigentliche Funktionalität des Aspekts wurde von uns in Makro SIBs gekapselt. Demnach gibt unser Modell nur an, ob ein Advice vor oder nach dem vorgesehenen SIB eingewoben werden soll. Für dieses SIB steht im Modell ein Proxy-SIB als Platzhalter. Kapitel 9 erklärt, wie der dazugehörige Pointcut definiert wird. Für das Beispiel der Notlandung ist das Proxy SIB ein Platzhalter für das *GetBattery-SIB* im Steuerungsmodell, d.h. der Advice soll unmittelbar nach dem holen des Wertes für die Akkuladung ausgeführt werden.

10.2.1 Kollisionserkennung

Dieser Aspekt sorgt dafür, dass der Quadrokopter nicht mutwillig gegen ein Hindernis gesteuert werden kann. Dazu werden die Sensordaten, die die Abstände messen und vom SIB *CheckDistanceSIB* ermittelt und in den Context geschrieben werden, überprüft. Mit jeder Iteration durch das Steuerungsmodell wird für jede mögliche Steuerungsrichtung überprüft, ob Hindernisse in einem kritischen Abstand in dieser Richtung liegen. Nicht nur die unmittelbare Flugrichtung wird hierbei überprüft, sondern auch die Abstände im rechten Winkel nach links bzw. rechts zur zu prüfenden Flugrichtung. Dies hat den Sinn, sicher zu gehen, dass der Quadrokopter zwar in Flugrichtung kein Hindernis erkennt, jedoch seitlich ein Hindernis liegt, mit dem er zu kollidieren droht. Abbildung 10.2 zeigt das allgemeine Modell für die Erkennung von Kollisionen in eine Flugrichtung. Hier wird sowohl für mögliche Objekte in Flugrichtung als auch für Objekte links und rechts der Flugrichtung überprüft, ob der Abstand einen kritischen Wert unterschreitet. Der Grenzwert für die Flugrichtung wird mit *upperThreshold* definiert, die kritischen Seitenabstände werden mit *lowerTreshold* festgelegt. Wird einer der Werte unterschritten, so wird der jeweilige Contextkey für die Beschleunigung in die überprüfte Flugrichtung auf 0 gesetzt, was zur Folge hat, dass das *MoveSIB* den Quadrokopter sich nicht mehr in diese Richtung bewegen lässt. Zur Vereinfachung kann die Trägheit bei genügend großer Wahl der zu überprüfenden Kollisionsdistanz ignoriert werden.

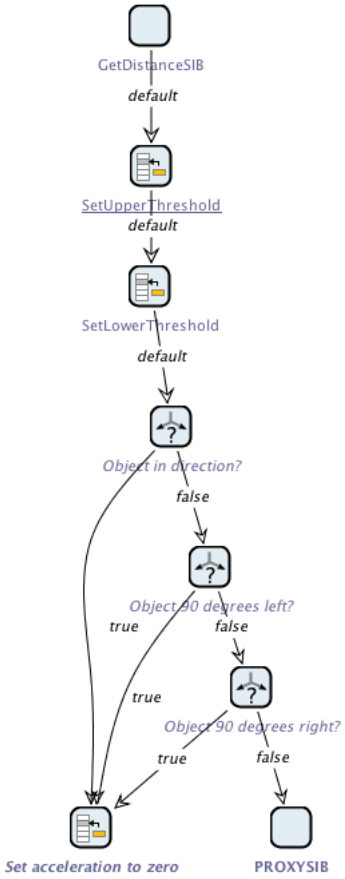


Abbildung 10.2: Modell für den kollisionsfreien Flug in eine Richtung

Eben wurde das allgemeine Modell für jede Flugrichtung (geradeaus, rückwärts, links, rechts, hoch, runter) beschrieben. In unserer Umsetzung gibt es für jede Flugrichtung ein Modell / einen Aspekt. Die Flugrichtungen in der horizontalen Ebene arbeiten genau wie oben beschrieben.

10.2.2 Sichere Landung

Die Quadrokopter soll eine sichere Landung einleiten, sobald absehbar ist, dass die Batterie bald leer ist und die verbleibende Ladung nur noch ausreicht, um selbstständig nach einer geeigneten Landestelle zu suchen. Geeignet ist eine Landestelle, wenn ringsum keine zu großen Gefälle sind, sprich die Fläche möglichst eben ist. Abbildung 10.3 zeigt das Vorgehen der sicheren Landung. Sollte die mittels *getBattery-SIB* ermittelte Batterieladung einen bestimmten Grenzwert unterschreiten, wird dieses Modell aufgerufen. Der Quadrokopter versucht nun ein Rechteck abzufliegen. Dazu fliegt er jede Seite des Rechtecks nacheinander ab. Während eine Seite abgeflogen wird, merkt sich der Quadrokopter die Flughöhe zu Beginn der Strecke und überprüft während des Flugs immer wieder, ob sich die aktuelle Höhe nicht allzu stark von der

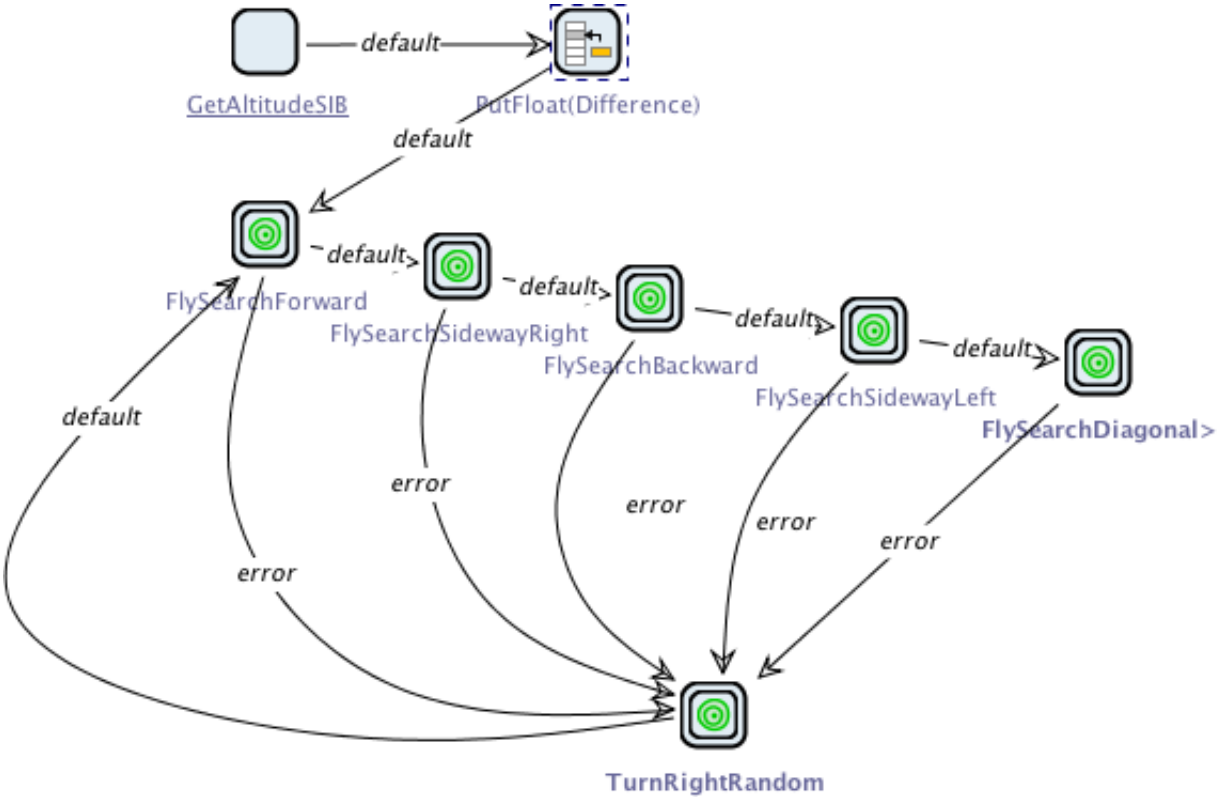


Abbildung 10.3: Modell für das Suchen einer geeigneten Landestelle

Ausgangshöhe unterscheidet. Sollte die Höhendifferenz, die mittels *PutFloat(Difference)* definiert wird, zu groß werden, ist der Untergrund der abgeflogenen Strecke zu uneben, um an dieser Stelle sicher landen zu können. In diesem Fall dreht sich der Quadrokopter mit einer zufälligen Geschwindigkeit nach rechts und die Suche wird mit dieser Ausgangsposition und Flugrichtung erneut gestartet. Konnte eine Strecke des Rechtecks komplett abgeflogen werden, d.h. die Höhenunterschiede waren akzeptabel, fliegt der Quadrokopter nacheinander die nächsten Strecken des Rechtecks ab und überprüft für diese Strecken ebenfalls wieder die Eignung der Strecke für eine Landung. Hat der Quadrokopter alle vier Seiten des Rechtecks erfolgreich abgeflogen, d.h. die Landefläche ist geeignet, so fliegt er diagonal zum Mittelpunkt des Rechtecks und leitet dort die Landung ein. Im Folgenden wird der innere Aufbau der verwendeten Makro-SIBs für das Abfliegen einer Strecke exemplarisch für die erste Strecke nach vorne beschrieben (FlySearchForward, Abbildung 10.4). In der ersten Iteration durch dieses Modell wurde der Contextkey *firstforwardstart* noch nicht gesetzt. Dieser wird daher gesetzt und markiert, dass der Vorwärtsflug nun beginnen soll. Die Startposition wird gespeichert, um in jeder weiteren Iteration durch das Modell festzustellen, ob bereits das Ende der abzufliegenden Strecke erreicht wurde. Anschließend werden die Contextkeys für alle Flugrichtungen außer die für den Vorwärtsflug auf 0 gesetzt und das Submodell wird verlassen. Das nun folgende Modell FlySearchRightward wird aufgerufen und sogleich wieder verlassen, weil es keinen Contextkey findet, der zeigt, dass be-

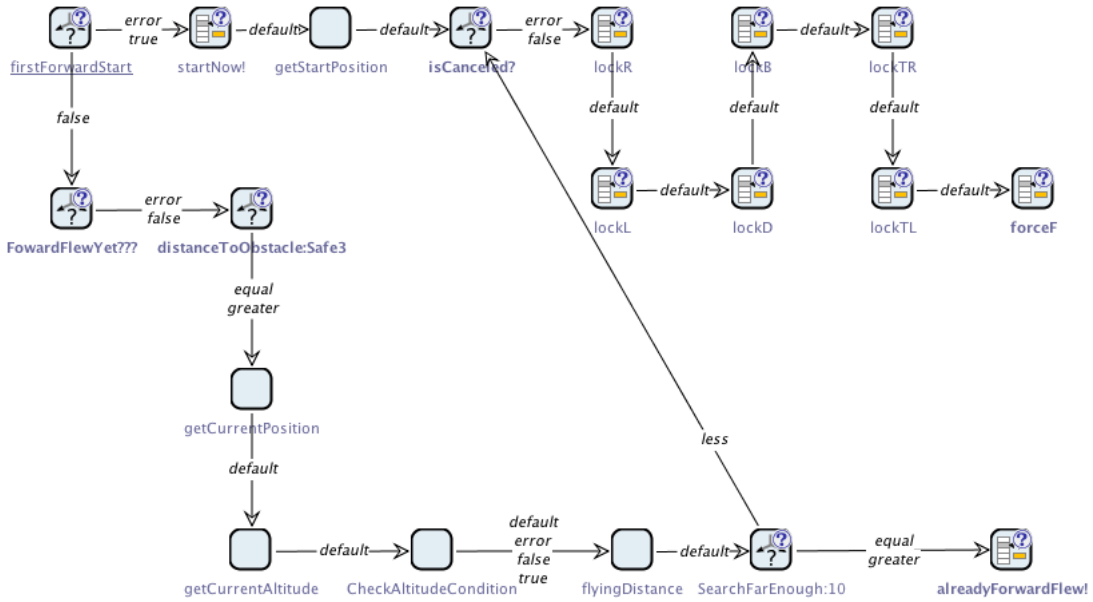


Abbildung 10.4: Modell des Makros flySearchForeward

reits die erste Strecke nach vorne abgeflogen wurde. Die anderen nachfolgenden Modelle für die weiteren Richtungen werden analog verlassen. In der nächsten Iteration durch das Steuerungsmodell wird durch den Wert von *firstforwardstart* festgestellt, dass bereits begonnen wurde nach vorne zu fliegen. Es wird anhand des Contextkeys *alreadyForwardFlew* überprüft, ob bereits die ganze Strecke abgeflogen wurde. Ist dies der Fall wird das Modell verlassen und das Modell für den Flug nach rechts aufgerufen. Ist dies nicht der Fall, wird die oben angesprochene Höhendifferenz zwischen der Startposition und der aktuellen Position mit dem SIB *CheckAltitudeCondition* überprüft. Sollte die Überprüfung ergeben, dass eine zu unebene Landefläche vorliegt, wird das Modell per error-Branch verlassen und die zufällige Drehung nach rechts für einen erneuten Versuch aufgerufen. Ist die Landefläche bis dahin geeignet, wird überprüft, ob nun das Ende der Strecke erreicht wurde. Sollte dies der Fall sein, wird der Kontexkey *alreadyForwardFlew* gesetzt, andernfalls fliegt der Quadrokopter erneut ein Stück nach vorne. Mittels der so verwendeten Contextkeys kann der Quadrokopter sich über mehrere Ausführungszyklen hinweg merken, in welchem Zustand er sich befindet, also in welche Richtung er gerade fliegen muss.

10.2.3 Verlust des Controllersignals

Sollte die Verbindung zum Controller abbrechen, so soll der Quadrokopter kurz hovern und schließlich sicher landen. Abbildung 10.5 zeigt das Modell dieses Safety Aspekts. Der Verlust der Verbindung zum Controller lässt sich nur indirekt feststellen. Dazu wird in jeder Iteration

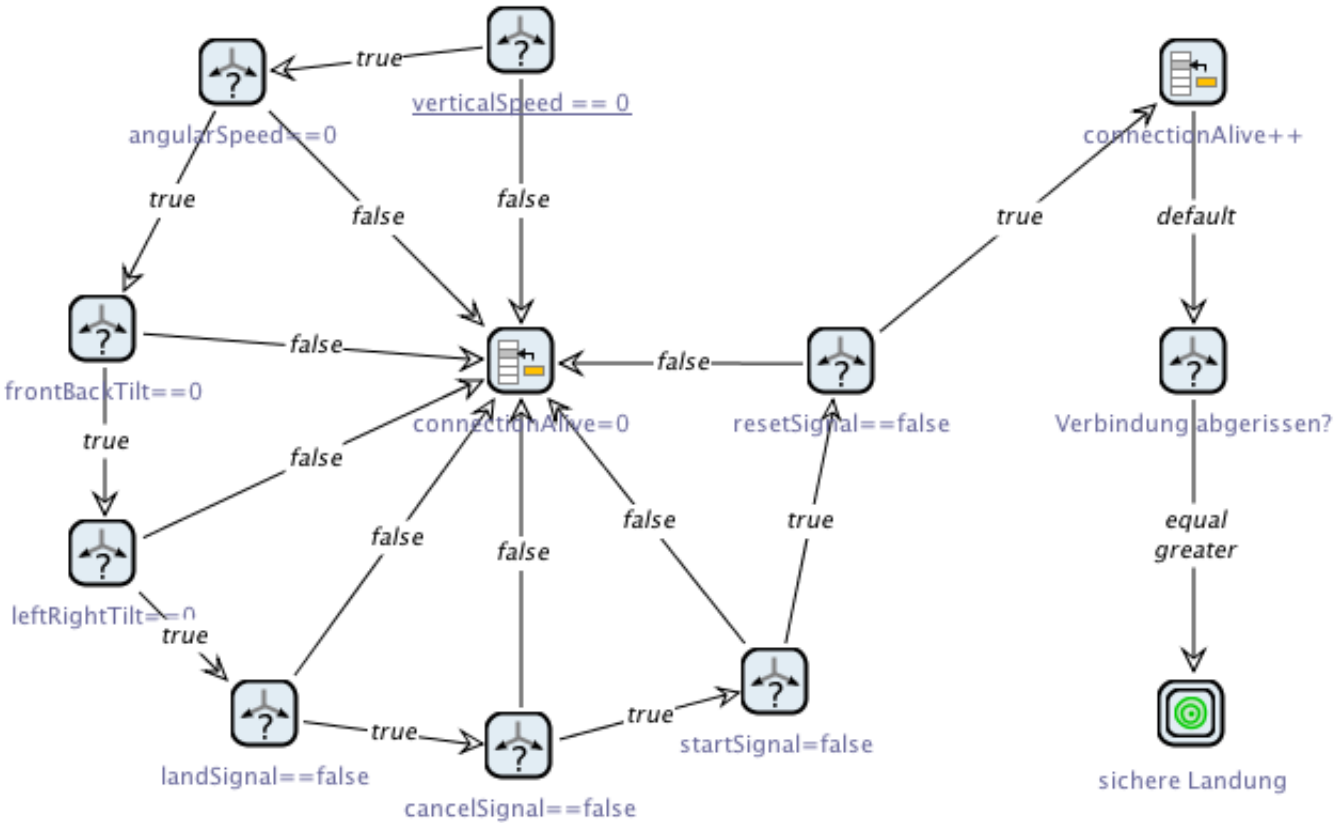


Abbildung 10.5: Modell für den Aspekt bei verlorenem Controllersignal

durch das Steuerungsmodell überprüft, ob irgendeine Taste gedrückt wurde. Das SIB *ReadControllerInput* empfängt die Werte des Controllers und schreibt diese in den Context. Dies macht sich das Modell dieses Aspekts zu eigen und prüft für jeden der Contextkeys, ob ein Wert ungleich 0 gesetzt wurde. Ist dies für mindestens einen der Keys der Fall, so wurde offensichtlich ein Signal vom Controller übermittelt. Wurde in einer Interation keine Taste gedrückt, so sind die Werte all der geprüften Contextkeys gleich 0. In diesem Fall wird der Contextkey *ConnectionAlive* um eins inkrementiert. Erreicht dieser Zähler einen bestimmten Maximalwert - in unserem Test hat sich ein Wert von 100 als angemessen erwiesen -, so wird eine sichere Landung eingeleitet, da entweder lange keine Taste am Controller gedrückt wurde oder die Verbindung abgebrochen ist.

10.2.4 Notlandung

Hat der Quadrokopter nur noch sehr wenig Akku, so dass davon auszugehen ist, dass die Energie nicht mehr ausreicht um einen geeigneten Landeplatz zu suchen und sicher zu landen, muss er stattdessen sehr langsam zu Boden gehen, um so auch bei einer unebenen Fläche möglichst

unbeschadet landen zu können. Abbildung 10.6 zeigt die Modellierung dieser Sicherheitsanforderung. Der Aspekt prüft zunächst die Batterieladung. Ist die Akkuladung kleiner oder gleich

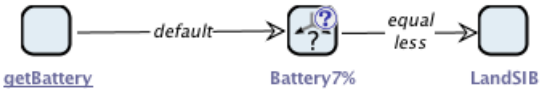


Abbildung 10.6: Modell für den Aspekt der Notlandung

7% der Kapazität, so wird eine Landung durchgeführt. Das dazu aufgerufene *LandSIB*, bring den Quadrokopter langsam senkrecht zu Boden. In jeder Iteration durch das sichere Steuerungsmodell muss demnächst das Batterielevel geprüft werden. Erreicht die Ladung einmal einen kritischen Wert, so wird jedes mal wieder das *LandSIB* aufgerufen und der Quadrokopter weiter abwärts gelenkt.

10.3 Sicheres Steuerungsmodell

Im voranstehenden Abschnitt wurden die von uns vorgesehenen Aspekte und ihre Realisierung vorgestellt. Zusammen mit dem vorgestellten Weber und dem (unsicheren) Steuerungsmodell bilden sie alle notwendigen Grundlagen, um ein sicheres Steuerungsmodell zu erhalten: der Weber webt die Aspekte in das unsichere Steuerungsmodell ein. Das Ergebnis ist ein, im Sinne der berücksichtigten Aspekte, sicheres Steuerungsmodell.

Wichtig ist nun, zu definieren, an welchen Stellen im Steuerungsmodell die Aspekte eingewoben werden sollen.

Tabelle 10.1 gibt die SIBs bzw. Pointcuts an, vor bzw. nach denen jeder Aspekt vorkommen soll. Sicher* meint hier alle Aspekte für die Kollosionserkennung.

Aspekt	**Pointcut**	**Webeart**
Sicher*	MoveSIB	before
Sichere Landung	GetDistanceSIB	after
Verlust des Controllersignals	MoveSIB	
Notlandung	GetDistanceSIB	after

Tabelle 10.1: Übersicht über die Webestellen für Aspekte

Als problematisch erweist sich nun, dass mehrere Aspekte vor dem Move-SIB eingewoben werden sollen. Hat man ein Steuerungsmodell s, eine Webefunktion ω und Aspekte $a_1, ..., a_n$, so ist das Ergebnis von $\omega(\omega(s, a_1), a_2)$ nicht das selbe Ergebnis, wie $\omega(\omega(s, a_2), a_1)$. Die Reihenfolge in der Aspekte eingewebt werden ist also relevant und muss in einer Halbordnung definiert werden. Diese Halbordnung bestimmt, welcher Aspekt zuerst in das Steuerungsmodell gewoben wird und welcher Aspekt erst in das aus einem vorangegangenen Webeprozess entstandene Modell gewoben wird. Die Problematik der Reihenfolge lässt sich am Beispiel der sicheren Landung und der Notlandung verdeutlichen. Beide Aspekte nutzen als Pointcut das *GetDistanceSIB*. Wird in das Steuerungsmodell unmittelbar nach dem *GetDistanceSIB* die Notlandung eingewoben und in das resultierende Modell ebenfalls unmittelbar nach dem *GetDistanceSIB* die sichere Landung eingewoben, so wird der Quadrokopter immer eine sichere Landung anstelle der Notlandung durchführen, auch wenn die Batterieladung bereits eine Notlandung erfordern würde, denn die Bedingung an die Batterie für die sichere Landung wird zuerst geprüft.

Um im Webeprozess sicherzustellen, dass das schlussendlich resultierende Steuerungsmodell auch wie gewünscht funktioniert, muss die Menge der Aspekte $\mathcal{A}$ halbgeordnet sein. Wir haben also eine Halbordnung $(\mathcal{A}, \preceq)$ definiert, die angibt, in welcher Reihenfolge die Aspekte eingewebt

werden sollen. Das Hasse-Diagramm in Abbildung 10.7 stellt die so halbgeordnete Menge $\mathcal{A}$ grafisch dar. Dabei wird ein Aspekt in der Form (Advice-Name, Pointcut-Name) geschrieben.

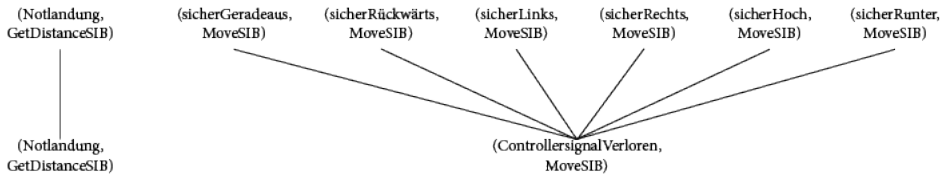


Abbildung 10.7: Hasse Diagramm der Halbordnung über die Menge der Aspekte

Es ist zu sehen, dass ein nicht zusammenhängender Graph mit zwei zusammenhängenden Teilgraphen entsteht. Die Knoten jedes Teilgraphen haben jeweils den Pointcut gemeinsam. Dies ist in so weit sinnvoll, als das bei den von uns vorgesehenen Aspekten die Reihenfolge des Webens nur bei solchen Advices wichtig ist, die den selben Joinpoint haben. Knoten auf der selben Höhe des Diagramms können in beliebiger Reihenfolge gewebt werden. Erst, wenn alle Knoten der höheren Ebene verwoben wurden, können in das resultierende Modell die Aspekte der Ebene darunter eingewoben werden. Demnach ist es nicht relevant, welcher Advice für die Kollisionserkennung zuerst eingewoben wird. Jedoch muss sichergestellt sein, dass erst nach diesen Advices geprüft wird, ob das Controller-Signal verloren wurde, da erst zu diesem Zeitpunkt alle Controller-Signale überprüft werden sollen. Wie im obigen Beispiel gesehen, soll auch die Notlandung erst in ein Modell verwoben werden, in das bereits die sichere Landung verwoben wurde.

KAPITEL 11

Projektmanagement

Dieses Kapitel umreißt die Methoden und Technologien, die für die Entwicklung der Software und die Kommunikation in der Projektgruppe genutzt wurden. Auch wird kurz begründet, wieso diese Technologien initial gewählt wurden und ein Fazit gezogen, ob der Einsatz der Technologie erfolgreich war.

11.1 Redmine

Um die Verwaltung von Dokumentation, Protokollen und anderen Artefakten, die nicht zusammen mit dem regulären Quellcode verwaltet werden sollten, zu ermöglichen, wurde eine Software zur Projektverwaltung eingesetzt. Die Wahl fiel dabei auf Redmine [Lan13], eine Open-Source-Lösung, die bereits als einsatzbereiter Dienst der TU Dortmund bereit stand (in Abbildung 11.1 dargestellt). Durch diese Wahl entstand kein zusätzlicher Arbeitsaufwand für die Projektgruppe, um das System zu warten.

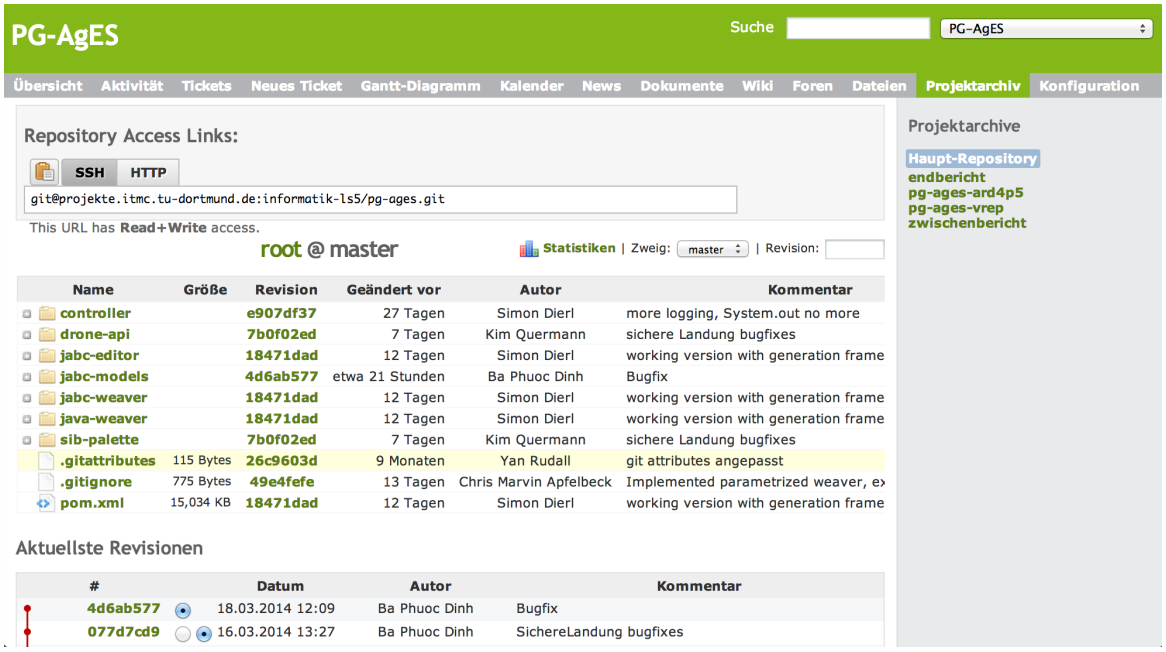


Abbildung 11.1: Die Redmine-Instanz der Projektgruppe

Redmine bietet diverse Features, die sich als nützlich erwiesen:

Wiki Redmine bietet eine ausgereifte Wiki-Lösung, die genutzt werden konnte, um Sitzungsprotokolle, technische Dokumentation und ähnliche Dokumente zu verwalten.

Dateihosting Um große Binärdateien nicht in einem Versionskontrollsystem verwalten zu müssen, konnte die Hosting-Unterstützung von Redmine genutzt werden, um unter anderem die Maps für die Simulationsumgebung bereit zu stellen.

VCS-Betrachter Redmine erlaubt es, über ein Webinterface den aktuellen Stand des Versionskontrollsystems zu überprüfen. Dies vereinfachte die Arbeit mit der Versionsverwaltung beträchtlich.

Die Nutzung von Redmine erwies sich als gute Entscheidung. Die Arbeitsabläufe der Projektgruppe konnten durch Redmine verbessert werden; die Software wies beim Einsatz keine substanziellen Probleme auf.

11.2 Git

Um den im Zuge der Projektgruppe entwickelten Quellcode zu verwalten, sollte ein Versionskontrollsystem genutzt werden, um gemeinsames Bearbeiten und eine historische Sicht auf die Quellen zu erlauben und Datenverlusten vorzubeugen. Da die genutzte Redmine-Instanz nur die Versionskontrollsysteme Subversion und Git [Git14; Cha09] unterstützte, fiel die Wahl auf das technisch fortschrittlichere der beiden Systeme, Git.

Abbildung 11.2: Das Git-Logo

Im praktischen Einsatz stellte sich Git als zweifelhafte Wahl heraus: Die fortschrittlichen Features erwiesen sich eher selten als nützlich, außerdem erwies sich die Unterstützung von Microsoft Windows durch Git als eher schlecht (verglichen mit Unix-artigen Systemen) und das System war auf dieser Plattform daher nur schwer zu bedienen. Dennoch gelang es, die Software erfolgreich zu nutzen.

11.3 Maven

Die Software, die im Rahmen der Projektgruppe entwickelt wurde, benötigt eine Infrastruktur, das das Kompilieren des Java-Quellcodes, Erstellen eines JAR-Archivs und andere Aufgaben automatisiert. Die Wahl fiel aufgrund vorheriger positiver Erfahrungen mit der Software auf Maven [Mav14; OBr+11], eine Anwendung, die diverse Teile des Build-Prozesses automatisieren kann.

Abbildung 11.3: Das Maven-Logo

Diverse Features von Maven, die über das bloße Aufrufen von Compiler, JAR-Archivierer und ähnlichen Standard-Tools herausgingen, erwiesen sich als außergewöhnlich nützlich:

Code-Generierung Durch die Einbindung von Genesys (siehe Abschnitt 9.4) in den Build-Prozess konnten die Generierung des Aspektwebers auf einem jABC-Graphen und die Nutzung des Webers selbst automatisiert werden.

Abhängigkeitsverwaltung Maven ist in der Lage, benötigte Bibliotheken eigenständig herunterzuladen und zu verwalten. Dies erleichterte die Einbindung des jABC und diverser Open-Source-Projekte enorm.

Maven erwies sich als gute Wahl für ein Build-Management-System. Trotz der Komplexität des System ergaben sich nur wenige Probleme, die durch den Nutzen mehr als aufgewogen wurden.

KAPITEL 12

Fazit

An Ende dieses Berichts bleibt nur noch übrig ein Fazit über die erbrachten Leistungen der Projektgruppe zu ziehen. Dieses Fazit kann anhand der in Kapitel 6.1 aufgestellten Ziele erfolgen.

Die Minimalziele wurden komplett erfüllt. Für die **Simulationsumgebung** wurde das Programm VREP herangezogen, welches sich als einfache und bereits existierende Möglichkeit darstellte einen Quadrokopter zu simulieren.

Das von der Projektgruppe entwickelte **Steuerungssystem** ist in der Lage den simulierten Quadrokopter in VREP zu steuern. Hierfür wurden unterschiedliche APIs eingesetzt und ein Interface geschaffen, welches leicht auf einen echten Quadrokopter übertragbar ist. Dieses Steuerungssystem wurde in jABC entwickelt und ermöglicht eine Steuerung per Tastatur, XBox-Controller oder auch das Steuern durch ein Missionsmodell.

Die **Sicherheitsaspekte**, die im Rahmen der Projektgruppe modelliert wurden waren die Kollisionserkennung, die sichere Landung, der Verlust der Controllersignals und die Notlandung. Sie wurden durch unterschiedliche Webarten eingewebt. Hierbei wurde zur Hilfe eine Halbordnung definiert, welche die Reihenfolge bestimmt, wann welcher Aspekt eingewoben wird.

Der **Aspektweber** wurde dreischrittig entwickelt. Der erste Prototyp für den Weber wurde mit Hilfe der Techniken XQuery und XPath entwickelt. Dieser Prototyp wurde dann zu einem

in Java implementierten Weber weiterentwickelt, um diesen dann schlussendlich in die Zielform als jABC-Modell zu übertragen.

Von den optionalen Zielen wurde sich im zweiten Abschnitt der Projektgruppe vor allem mit den **Missionen** beschäftigt. Neben den theoretischen Grundlagen, was Missionsmodelle eigentlich darstellen und was für unterschiedliche Arten von Missionen es gibt, wurde hier eine tatsächliche Mission in jABC modelliert. Diese Mission ist in der Lage den Quadrokopter durch unbekanntes Terrain auf einen Zielpunkt hin zu bewegen. Wünschenswert wäre hier noch gewesen, sich mit **Aspekte für Missionen** auseinander zu setzen. Dies gelang aber nicht mehr ausgiebig im Rahmen der Projektgruppe.

Das gleiche gilt für das Ziel **Erweiterte Aspektorientierung**.

Insgesamt bleibt fest zu halten, dass die Minimalziele der Projektgruppe erfüllt wurden und außerdem weitere Optionalziele bearbeitet wurden.

Literatur

[Cha09] Scott Chacon. *Pro Git*. 1. Aufl. Expert's Voice in Software Development. Apress, 26. Juni 2009. ISBN: 978-1-430-21833-3. URL: `http://git-scm.com/book` (besucht am 25. 02. 2014) (siehe S. 126).

[FK10] Michael A. Folcik und Bijan Karimi. „A Simulator for Robot Navigation Algorithms". In: *International Science Index* 4.4 (2010), S. 113–118. ISSN: 1307-6892. URL: `http://waset.org/publications/12167` (besucht am 28. 03. 2014) (siehe S. 80).

[Jör13] Sven Jörges. „Extreme Model-Driven Development and jABC". In: *Construction and Evolution of Code Generators*. Bd. 7747. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, S. 39–71. ISBN: 978-3-642-36126-5. DOI: `10.1007/978-3-642-36127-2_3` (siehe S. 15–18).

[JSM11] Sven Jörges, Bernhard Steffen und Tiziana Margaria. „Building Code Generators with Genesys: A Tutorial Introduction". In: *Generative and Transformational Techniques in Software Engineering III*. Hrsg. von João M. Fernandes u. a. Bd. 6491. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, S. 364–385. ISBN: 978-3-642-18022-4. DOI: `10.1007/978-3-642-18023-1_10` (siehe S. 19, 112).

[Kic+97] Gregor Kiczales u. a. „Aspect-oriented programming". In: *ECOOP'97 — Object-Oriented Programming*. Hrsg. von Mehmet Akşit und Satoshi Matsuoka. Bd. 1241. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1997, S. 220–242. ISBN: 978-3-540-63089-0. DOI: `10.1007/BFb0053381` (siehe S. 96, 97).

[LR09] Bernhard Lahres und Gregor Rayman. „Aspekte und Objektorientierung“. In: *Objektorientierte Programmierung. Das umfassende Handbuch*. 2. Aufl. Galileo Computing, 2009, S. 527–572. ISBN: 978-3-8362-1401-8 (siehe S. 11).

[Lun10] Jan Lunze. *Regelungstechnik 1. Systemtheoretische Grundlagen, Analyse und Entwurf einschleifiger Regelungen*. 8. Aufl. Springer-Lehrbuch. Springer Berlin Heidelberg, 2010. ISBN: 978-3-642-13807-2. DOI: 10.1007/978-3-642-13808-9 (siehe S. 36–38).

[OBr+11] Tim O'Brien u. a. *Maven. The Complete Reference*. 1.0. Sonatype, Inc., 2011. URL: http://www.sonatype.com/resources/books/maven-the-complete-reference (besucht am 25. 02. 2014) (siehe S. 127).

[Pal05] Grant Palmer. *Physics for Game Programmers*. Expert's Voice in Software Development. Apress, 20. Apr. 2005. ISBN: 978-1-59059-472-8 (siehe S. 34, 35).

[Pis+12] Stephane Piskorski u. a. *AR.Drone Developer Guide*. Version SDK 2.0. Parrot. 21. Mai 2012. URL: http://www.msh-tools.com/ardrone/ARDrone_Developer_Guide.pdf (besucht am 27. 03. 2014) (siehe S. 59).

[Rao95] N.S.V. Rao. „Robot navigation in unknown generalized polygonal terrains using vision sensors“. In: *Systems, Man and Cybernetics, IEEE Transactions on* 25.6 (Juni 1995), S. 947–962. ISSN: 0018-9472. DOI: 10.1109/21.384257 (siehe S. 80).

[Ste+07] Bernhard Steffen u. a. „Model-Driven Development with the jABC“. In: *Hardware and Software, Verification and Testing*. Hrsg. von Eyal Bin, Avi Ziv und Shmuel Ur. Bd. 4383. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, S. 92–108. ISBN: 978-3-540-70888-9. DOI: 10.1007/978-3-540-70889-6_7 (siehe S. 13, 18, 20, 24).

[TA09] Khaldoun K. Tahboub und Munaf S. N. Al-Din. „A Neuro-Fuzzy Reasoning System for Mobile Robot Navigation“. In: *Jordan Journal of Mechanical and Industrial Engineering* 3.1 (März 2009), S. 77–88. ISSN: 1995-6665. URL: http://jjmie.hu.edu.jo/files/v3n1/jjmie-81-08%20Modified.pdf (besucht am 28. 03. 2014) (siehe S. 80).

Weblinks

[AUS14] Australian RC Technologies. *Parrot AR Drone Indoor Hull Stickers (Yellow)*. 2014. URL: http://www.aust-rc-tech.com.au/gallery/Spare+Parts+-+ARDrone/parrot-ar-drone-indoor-hull-stickers-yellow/102804 (besucht am 27. 03. 2014) (siehe S. 57).

[Bad14] Dirk Bade. *YADrone. Yet another AR.Drone framework*. Arbeitsgruppe Verteilte Systeme und Informationssysteme, Fachbereich Informatik, Universität Hamburg. 10. Jan. 2014. URL: http://vsis-www.informatik.uni-hamburg.de/oldServer/teaching/projects/yadrone/ (besucht am 22. 09. 2013) (siehe S. 56, 58).

[Cop14] Coppelia Robotics. *v-rep. virtual robot experimentation platform*. 2014. URL: http://www.coppeliarobotics.com/ (besucht am 10. 03. 2014) (siehe S. 63, 64).

[Ecl14] The Eclipse Foundation. *aspectj. crosscutting objects for better modularity*. 2014. URL: http://www.eclipse.org/aspectj/ (besucht am 15. 03. 2014) (siehe S. 96).

[Git14] Git. *Git*. 2014. URL: http://git-scm.com/ (besucht am 25. 02. 2014) (siehe S. 126).

[Hei13a] Ulrich Heither. *Propellerdrehrichtung eines Quadrokopters in +-Konfiguration*. 2. Juni 2013. URL: http://commons.wikimedia.org/wiki/File:Quadrokopter-+-Konfiguration.gif (besucht am 02. 05. 2014) (siehe S. 27).

[Hei13b] Ulrich Heither. *Propellerdrehrichtung eines Quadrokopters in X- bzw. H-Konfiguration*. 2. Juni 2013. URL: http://commons.wikimedia.org/wiki/File:Quadrokopter-X-H-Konfiguration.gif (besucht am 02. 05. 2014) (siehe S. 27).

[Lan13] Jean-Philippe Lang. *Redmine*. 2013. URL: http://www.redmine.org/ (besucht am 25. 02. 2014) (siehe S. 125).

[LSV10] Chair for Programming Systems. *Genesys*. Version 2.0-beta1. Dortmund University of Technology, Faculty of Computer Science. 10. Nov. 2010. URL: http://jabc.cs.tu-dortmund.de/genesys/ (besucht am 28. 02. 2014) (siehe S. 112).

[LWJ14] LWJGL. *LWJGL. Lightweight Java Gaming Library*. 29. Apr. 2014. URL: http://www.lwjgl.org/ (besucht am 29. 04. 2014) (siehe S. 42, 52).

[Man11] Sven Manske. *easydrone. the javadrone composer*. Sep. 2011. URL: https://kenai.com/projects/easydrone/ (besucht am 04. 03. 2014) (siehe S. 56).

[Mav14] Apache Maven Project. *Welcome to Apache Maven*. The Apache Software Foundation. 19. Feb. 2014. URL: https://maven.apache.org/ (besucht am 21. 02. 2014) (siehe S. 127).

[Par12a] Parrot SA. *AR.Drone 2.0. Parrot new wi-fi quadricopter - Specifications*. 2012. URL: http://ardrone2.parrot.com/ardrone-2/specifications/ (besucht am 04. 03. 2014) (siehe S. 32).

[Par12b] Parrot SA. *Parrot new wi-fi quadricopter - Apps*. 2012. URL: http://ardrone2.parrot.com/apps/ (besucht am 27. 03. 2014) (siehe S. 33).

[PUC14] Pontifícia Universidade Católica do Rio de Janeiro. *The Programming Language Lua*. 2014. URL: http://www.lua.org/ (besucht am 10. 03. 2014) (siehe S. 64).

[Red12] Tobias Redmann. *Parrot AR.Drone 2.0 – Privat-Helicopter mit Android- und iOS-Steuerung*. 14. Juli 2012. URL: http://www.geekovation.de/gadgets/parrot-ar-drone-2-0-privat-helicopter-mit-android-und-ios-steuerung/ (besucht am 04. 03. 2014) (siehe S. 31).

[Shm13] Denis Shmyger. *javadrone. AR.Drone Java API*. März 2013. URL: http://code.google.com/p/javadrone/ (besucht am 04. 03. 2014) (siehe S. 56).

[Yos11] Shigeo Yoshida. *ARDroneForP5. Let's control AR.Drone on Processing!* 29. März 2011. URL: http://kougaku-navi.net/ARDroneForP5/index_en.html (besucht am 04. 03. 2014) (siehe S. 56).