Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Testen modellgetrieben entwickelter Software auf Basis des Rhapsody-OX-Framework am Beispiel der ARTIS-Plattform

Diplomarbeit

Leipzig, Juli, 2007

vorgelegt von

Schwarzer, Robert
Studiengang Informatik

Fakultät für Mathematik und Informatik

Vorwort

Diese Diplomarbeit stellt den Abschluss meines Studiums zum Diplom-Informatiker an der Universität Leipzig dar, dem ein duales Studium zum Diplom-Ingenieur (BA) für Informationstechnik an der Berufsakademie Mannheim vorausgegangen ist.

An dieser Stelle sei ein besonderer Dank an Herrn Florian Adolf ausgesprochen, der mir während der Betreuung dieser Arbeit jederzeit seine vollste Unterstützung zukommen ließ. Ebenso sei ein herzlicher Dank an Herrn Prof. Frank Thielecke gerichtet, der es mir ermöglichte, diese Arbeit am DLR Standort Braunschweig zu verfassen. Des Weiteren gilt ein ganz besonderer Dank dem Betreuer der Universität Leipzig, Herrn Prof. Hans-Gert Gräbe, ohne dessen Kritiken und Anregungen die Arbeit in dieser Form nicht möglich gewesen wäre.

Der Text dieser Arbeit wurde mit dem Textsatzprogramm LaTeX verfasst. Die Auswertung der Testergebnisse und die Darstellung in Diagrammen geschah mit Hilfe von Microsoft Excel. Die Abbildungen wurden mit CorelDRAW und Microsoft Visio erstellt.

Abstract

Die Diplomarbeit ist im Rahmen des ARTIS-Projektes (Autonomous Rotorcraft Testbed for Intelligent Systems) am Institut für Flugsystemtechnik des Deutschen Zentrums für Luft- und Raumfahrt entstanden. Mit auf Modellhubschraubern basierenden Testplattformen werden Konzepte für Hard- und Software-Lösungen zum autonomen Fliegen erforscht und bis hin zum Flugversuch erprobt.

In dieser Arbeit werden Modellierung und Verifikation eines ereignisbasierten Software-Systems durchgeführt. Der Fokus liegt dabei auf einer Software-Komponente des Entscheidungssystems an Bord des unbemannten Forschungshubschraubers, welche verschiedene Betriebsmodi implementiert und hierzu selbstständig Entscheidungen über auszuführende Systemverhalten trifft. Diese Komponente wird beispielhaft in einen modellbasierten Software-Entwicklungsprozess mit dem Werkzeug Rhapsody und dem dazugehörigen OX-Framework integriert. Der entsprechende Testansatz wird durch Testautomaten ebenfalls modellbasiert realisiert und basiert außerdem auf einem Unit-Testframework.

Als Ergebnis steht die Software-Komponente in Form eines wiederverwendbaren Modells zur Verfügung, welches nach der Code-Generierung und vor der eigentlichen Systemintegration verifiziert werden kann. Zur Modellierung kommen UML State Charts zum Einsatz, die für das Testen der Ereignisdetektierung und der Entscheidungslogik mit dem CppUnit-Testframework verbunden werden.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **3**
2.1 Das ARTIS-Projekt . . . 3
2.1.1 Die ARTIS-Modellhubschrauber . . . 3
2.1.2 Das ARTIS-Systemkonzept . . . 4
2.1.3 Die Methodik der ARTIS-Systemsimulation . . . 7
2.2 Ereignisbasierte Systeme . . . 9
2.2.1 Ereignis und Benachrichtigung . . . 9
2.2.2 Ereignisanforderung . . . 10
2.2.3 Anforderungsklassifikation . . . 11
2.2.4 Benachrichtigungsübertragung . . . 14
2.2.5 Modellierung ereignisbasierter Systeme . . . 15
2.3 UML State Charts . . . 17
2.3.1 Die Syntax . . . 17
2.3.2 Zusammengesetzte Zustände . . . 18
2.3.3 Code-Generierung . . . 22
2.3.4 Modellierungskonflikte . . . 23

3 Re-Engineering des State Charts **25**
3.1 Konzept . . . 26
3.2 Die Event Middleware . . . 29
3.2.1 Ereignisdetektierung und Transformation . . . 29
3.2.2 Algorithmen . . . 31
3.2.3 Restriktionen . . . 33
3.3 Die Ablaufsteuerung . . . 34
3.3.1 Modellierung . . . 34
3.3.2 Zustands- und Transitionsüberdeckung . . . 39
3.3.3 Komplexität . . . 41
3.4 Rhapsody OX-Framework . . . 44
3.4.1 State Chart Management . . . 44
3.4.2 Event Management . . . 45
3.4.3 Timeout Management . . . 46
3.4.4 Thread Management . . . 46
3.5 Die Integration . . . 47
3.5.1 Strukturierung der ARTIS-Systemintegration . . . 47

3.5.2 Modellintegration 50

4 Testverfahren **53**
4.1 Vorbetrachtungen 53
4.1.1 Die Teststrategie 55
4.1.2 Der Testautomat 56
4.1.3 Der Fehlerreport 57
4.2 Test Suite für die Event Middleware 58
4.2.1 Analyse 58
4.2.2 Test Key Generation 60
4.2.3 Build Test Case 61
4.2.4 Run Test Case 63
4.2.5 Report 65
4.2.6 Ergebnisse 66
4.3 Test Suite für die logische Struktur 68
4.3.1 Analyse 68
4.3.2 Test Key Generation 70
4.3.3 Build Test Case 71
4.3.4 Run Test Case 72
4.3.5 Report 74
4.3.6 Ergebnisse 75
4.4 Die Mustermission 78

5 Diskussion der Ergebnisse **83**

6 Zusammenfassung und Ausblick **85**

Literaturverzeichnis **87**

Abbildungsverzeichnis **90**

Abkürzungsverzeichnis **94**

Listings **95**

Symbolverzeichnis **96**

Tabellenverzeichnis **97**

Anhang

A UML State Chart Template **98**
A.1 Introduction 98
A.2 Background 98
A.3 Using the code 101
A.4 Internals 104

B Event Middleware Filter **105**

C Transitionsüberdeckung **111**

Erklärung **113**

Kapitel 1

Einleitung

Im Institut für Flugsystemtechnik des DLR Braunschweig werden in dem ARTIS-Projekt (Autonomous Rotorcraft Testbed for Intelligent Systems) autonome Testplattformen entwickelt. Ein Schwerpunkt des Projekts liegt in der Erforschung von Methoden zur Nachweisführung, um in Zukunft autonome unbemannte Luftfahrzeuge (UAV) auch in zivilen Lufträumen einsetzen zu können. Hierbei muss sich insbesondere auf die Software des unbemannten Systems konzentriert werden. Forschungsprojekte in diesem Bereich weisen hierbei bestimmte Merkmale auf, die bei der Nachweisführung der spezifizierten Funktionsweise zu berücksichtigen sind:

- hohe Entwicklungsdynamik im Bezug auf den Quellcode,
- umfangreiche strukturelle Änderungen und
- Fluktuation der beteiligten Entwickler.

Diese Eigenschaften führen zu verschiedenen Problemen in der Systementwicklung: Zum Einen kann das Verlassen einzelner Entwickler in dem Projekt zu dem Verlust von spezifischen Anforderungen und exaktem Wissen über die Funktionsweise der Algorithmen führen, da das implementierungsspezifische Fachwissen über die einzelnen Entwickler verteilt ist und häufig undetailliert vorliegt.

Zum Anderen können die hohe Entwicklungsdynamik und die strukturellen Änderungen des Software-Systems dazu führen, dass Spezifikation und Quellcode auseinander „driften“, da keine automatischen Synchronisationsverfahren verwendet werden und die manuelle Wartung der Spezifikation aus zeitlichen Gründen oft unberücksichtigt bleiben.

Bereits mit der Diplomarbeit [20] wurde für den Bereich Flugregelung des ARTIS-Projektes gezeigt, dass eine modellbasierte Software-Entwicklung mit MATLAB und Simulink die beiden Nachteile beseitigt und die Software-Qualität nachhaltig erhöht. Das Ziel dieser Diplomarbeit ist es, den allgemeinen Ansatz der

modellbasierten Software-Entwicklung auf einen weiteren UAV-Forschungsbereich zu übertragen. Hierfür wird als spezielles Anwendungsbeispiel die Ablaufsteuerung des Entscheidungssystems, deren Aufgabe es ist, die autonomen Abläufe an Bord der Forschungsplattform zu steuern, und das Werkzeug Rhapsody der Firma I-Logix ausgewählt.

Die Vorteile der modellbasierten Software-Entwicklung werden durch einen möglichst hohen Automatisierungsgrad gewonnen: Dies umfasst die Entwicklung eines plattformunabhängigen Modells (PIM) als Quellartefakt, welches zu einem (oder mehreren) plattformspezifischen Modell(en) (PSM) als Zielartefakt transformiert werden kann (Model-to-Model-Transformation in Bild 1.1 aus MDA für Embedded Systems [22] bzw. modellbasierte Modulprüfung [15]). Wird das Modell direkt in Quellcode (Platform Specific Implementation) transformiert, spricht man speziell von einer Model-to-Code-Transformation.

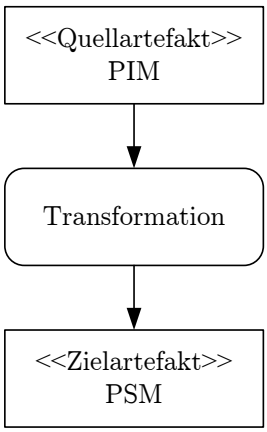

Bild 1.1: Transformation in der modellbasierten Entwicklung

Durch den modellbasierten Ansatz ist es möglich, die Modelle als Entwürfe wieder zu verwenden, die Integration zu automatisieren, Änderungen und Erweiterungen im Modell anstatt im Code durchzuführen sowie die Modelle bereits vor der Implementierung zu testen. Die Model-to-Code-Transformation führt dazu, dass der generierte Code weder lesbar noch wiederverwendbar sein muss, da eine klare Trennung zwischen den beiden Abstraktionsebenen vorliegt. Im Gegensatz dazu muss manuell implementierter Code stets ein Kompromiss zwischen guter Lesbarkeit, hoher Effizienz und guter Wiederverwendbarkeit sein.

Die Aufgaben dieser Diplomarbeit bestehen zunächst darin, den bisher manuell implementierten Quellcode der Ablaufsteuerung in einen modellbasierten Ansatz zu integrieren, sodass es möglich ist, Code-Generierung durchzuführen. Dazu muss ein Konzept entwickelt und realisiert werden, das den bisherigen UML State Chart aus dem Entscheidungssystem in ein transformierbares Modell überführt. Weiterhin soll ein Ansatz entwickelt werden, die ebenfalls manuell entwickelten Tests auf Basis des CppUnit-Testframeworks in den modellbasierten Entwicklungsprozess einzubinden.

Kapitel 2

Grundlagen

2.1 Das ARTIS-Projekt

Am Deutschen Zentrum für Luft- und Raumfahrt (DLR) wird in dem Institut für Flugsystemtechnik in Braunschweig der Flugversuchsträger ARTIS (Autonomous Rotorcraft Testbed for Intelligent Systems) auf der Basis von Modellhubschraubern entwickelt. Dieser Flugversuchsträger dient als Testplattform für den Entwurf und die Erprobung von Algorithmen und Methodiken für autonome intelligente Funktionen in Verbindung mit Hard- und Software. Im Rahmen des ARTIS-Projektes werden u.a. die folgenden Themen betrachtet:

- Modellierung, Simulation und Systemidentifizierung,
- Flugreglerkonzepte für Zuverlässigkeit und hohe Flugleistung,
- Sensorfusion und Navigation,
- Maschinelle Entscheidungsfähigkeit und Autonomiekonzepte,
- Echtzeitfähige, dynamische Flugbahnenplanung,
- Mensch-Maschine-Schnittstelle,
- Selbstorganisation für Multi-UAV (Swarming/Flocking) und
- Bildverarbeitung für Navigationsstützung und Kollisionsvermeidung.

2.1.1 Die ARTIS-Modellhubschrauber

Zur Demonstration der Forschungsergebnisse werden drei Modellhubschrauber unterschiedlicher Größe als Plattformen eingesetzt: miniARTIS, ARTIS und maxiARTIS (Bild 2.1).

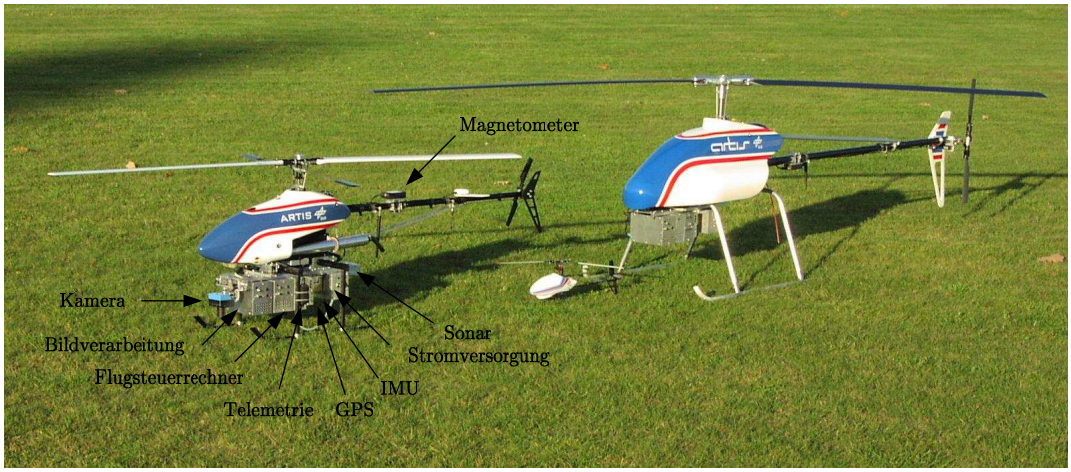

Bild 2.1: Die ARTIS Familie

Die wichtigsten Größen- und Leistungsunterschiede dieser Plattformen sind dem Vortrag über Konzepte und Nutzung von ARTIS [1] entnommen und in der Tabelle 2.1 zusammengefasst. Die dort angesprochene Nutzlast setzt sich aus abbildenden Sensoren, wie einer Stereobildkamera, einem Bildverarbeitungsrechner, einem Flugsteuerrechner, einem Telemetriemodel, der Satellitennavigation über GPS (Global Positioning System), einem Sonar, einem Magnetometer, der Inertialplattform IMU (Inertial Measurement Unit) und einem Akku-System zusammen. Forschungsziel ist es, alle drei Modellhubschrauber mit den gleichen Fähigkeiten zu versehen. Ist dies technisch umgesetzt, können die Modellhubschrauber unterschiedlich eingesetzt werden. Während sich miniARTIS für einen geräuscharmen Flug auch in Gebäuden eignet, kann maxiARTIS für eine schnelle Erkundung weiter Strecken verwendet werden. Als Einsatzmöglichkeit ist z.B. die Branderkennung in Wäldern denkbar.

	miniARTIS	ARTIS	maxiARTIS
Leistung	-	1,5 kW	4,5 kW
Rotordurchmesser	0,40 m	1,9 m	3,0 m
Eigengewicht	0,45 kg	6,0 kg	25 kg
Nutzlast	0,30 kg	6,0 kg	6,0 kg

Tabelle 2.1: Technische Daten der ARTIS-Plattform

2.1.2 Das ARTIS-Systemkonzept

Das ARTIS-System besteht aus zwei wesentlichen Bestandteilen, die über zwei redundante Datenlinks (Funkmodem, WLAN) miteinander verbunden sind. Einerseits der Modellhubschrauber mit eigenständiger Informationsverarbeitung, andererseits die separate mobile Bodenstation mit menschlichen Operatoren (Bild 2.2). Falls bei einem Flugversuch ein Fehler auftreten sollte, kann der Sicherheitspilot jederzeit eingreifen und den Autopiloten durch die Umschaltlogik überbrücken.

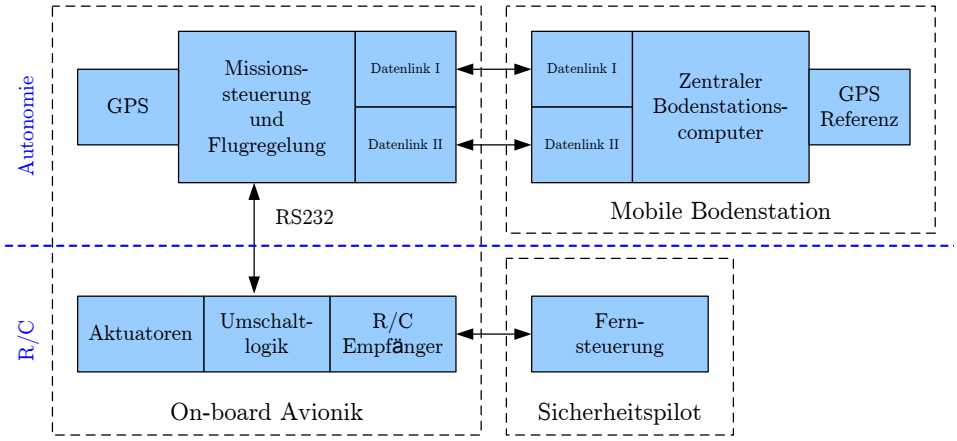

Bild 2.2: Das ARTIS-Systemkonzept

In der mobilen Bodenstation wird die eigenentwickelte Software Maestro eingesetzt. Sie dient als Statusmonitor, zur Instrumentendarstellungen, zur Anzeige des Onboard-Videos, zum Logging und zur Wegpunktplanung auf Kartenbasis. Des Weiteren ist auch die Einbindung von SRTM-Daten[1] möglich. Die Wegpunktplanung findet in Maestro (Bild 2.3) zunächst offline vor dem Start statt. Während des Fluges erfasst das Kamerasystem allerdings bewegliche Hindernisse oder unbekannte Umgebungen, so dass eine Online-Neuplanung der Flugbahn möglich ist. Diese Funktion muss jedoch erst durch Flugversuche in der Praxis erprobt werden.

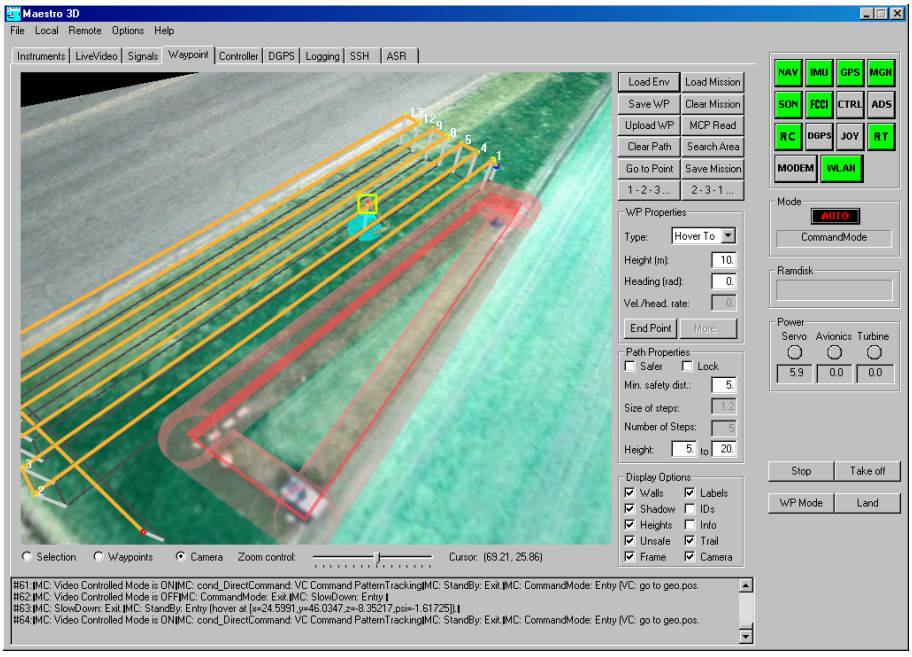

Bild 2.3: Wegpunktplanung mit Maestro

[1]Die Shuttle Radar Topography Mission (SRTM) ermittelte einheitliche, hochauflösende Fernerkundungsdaten von der gesamten Erdoberfläche.

Die Missionssteuerung (Bild 2.4) ist in dem ARTIS-System von zentraler Bedeutung, da dort die wesentlichen Eingabewerte zusammengefasst und für den Flugregler aufbereitet werden. Dazu gehören Statusinformationen des Hubschraubers, erkannte Hindernisse des Bildverarbeitungsrechners sowie Eingaben des Bodenstationsoperators. Es wird dabei eine aus der Robotik bekannte dreischichtige Architektur verwendet, deren genaue Funktionsweise in [23] beschrieben ist.
Die Ablaufsteuerung („Sequence Control System“) reagiert auf eintreffende Ereignisse und bietet durch ein umfangreiches Repertoire an Verhaltensmöglichkeiten Befehlseingaben für den Flugregler („Flight Controller“) an. Diese Vorgehen wird durch das „Supervisory Control System“ überwacht.

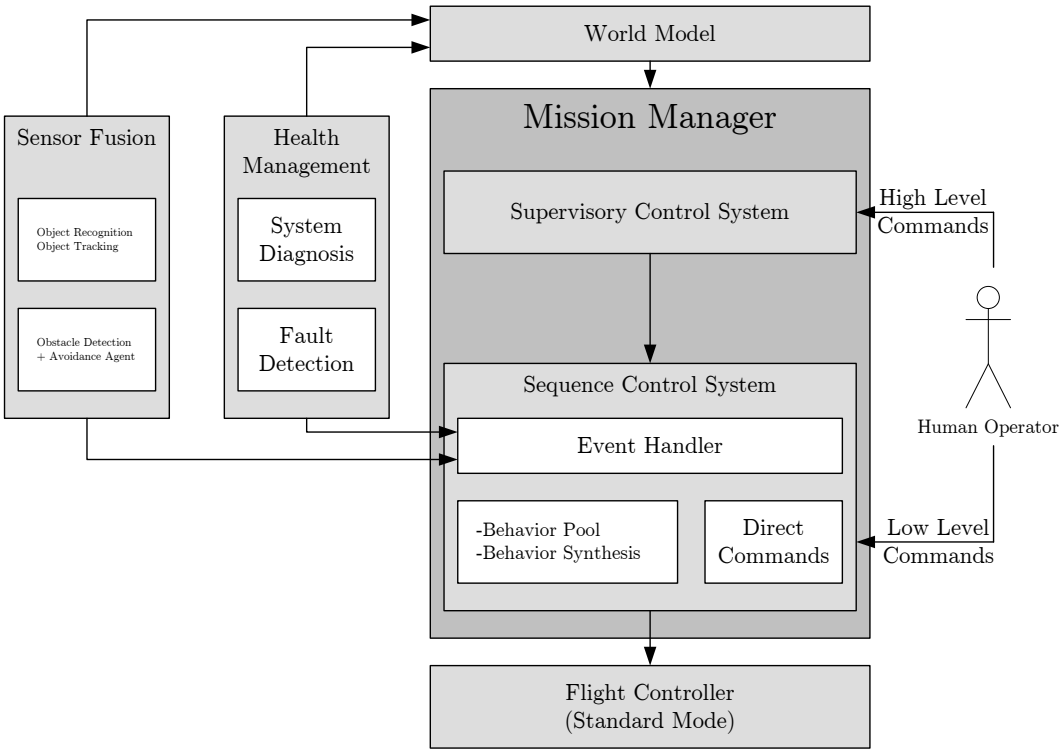

Bild 2.4: Konzept der Missionssteuerung in ARTIS

Eine Sequenz aus Verhaltensmöglichkeiten ist beispielhaft für eine Mosaiking-Mission in Bild 2.5 dargestellt. Diese Mission wird dabei aus Basisverhalten (unter anderem Take Off, Fly To, ...) zusammengesetzt und dem Flugregler nacheinander übermittelt. Kombiniert wird dies mit der Aufnahme von Fotos.

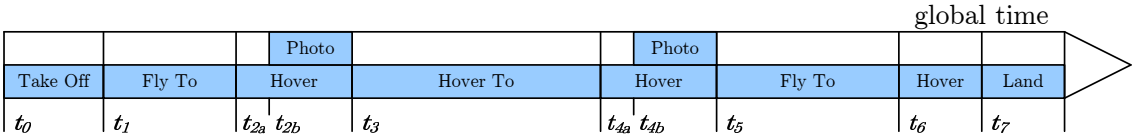

Bild 2.5: Geomosaiking-Verhaltenssequenz

2.1.3 Die Methodik der ARTIS-Systemsimulation

Vor jedem Flugversuch werden die integrierten Komponenten ausführlich in einer Systemsimulation unter Echtzeitbedingungen getestet. Es wird dabei ein Matlab/Simulink-Modell zur Simulation von Flugverhalten, Sensorik und den Aktuatoren eingesetzt. Die entwickelten Algorithmen werden als C/C++ Code in dieses Modell eingebunden oder in Matlab/Simulink direkt entwickelt.

Bei der Systemsimulation werden zwei Varianten unterschieden: Software-in-the-Loop-Simulation (SITL) und Hardware-in-the-Loop-Simulation (HITL). Während die SITL-Simulation (Bild 2.6) auf einem Computer als reine Software-Simulation stattfindet, wird die HITL-Simulation auf drei Computer (Simulations-, Flug- und Bodenstationscomputer) verteilt, um möglichst realistisch das Flugverhalten und die Echtzeitfähigkeit mit der tatsächlichen Hardware zu testen.

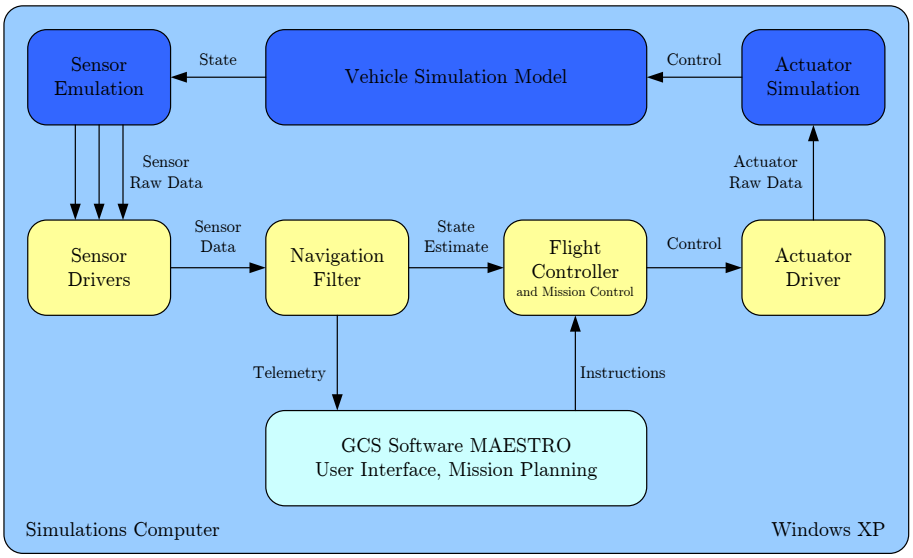

Bild 2.6: Software-in-the-Loop-Simulation

Die Hardware-in-the-Loop-Simulation (HITL) teilt sich, wie eben angesprochen, in drei miteinander kommunizierende Komponenten auf (Bild 2.7). In dem echtzeitfähigen Simulations-Computer der Firma dSPACE werden die Aktuatoren sowie das damit verbundene Flugverhalten simuliert und somit die Sensordaten in Echtzeit berechnet. Die spätere Flugversuchshardware des ARTIS ist mit dem dSpace-System verbunden und verhält sich wie in einem tatsächlichen Flugversuch. Die entwickelte Software wird damit erstmals auf dem Bordrechner mit dem Echtzeitbetriebssystem QNX getestet. Als dritte Komponente ist die Bodenstation involviert, an der die Planung und Überwachung des Versuchs erfolgt. Gleichzeitig können auch an dieser Stelle neue Funktionen von Maestro getestet werden. Vor dem abschließenden Flugversuch wird das Gesamtsystem in einem Bodentest auf Fehler im Bereich Sensorik, Telemetrie und Navigation überprüft.

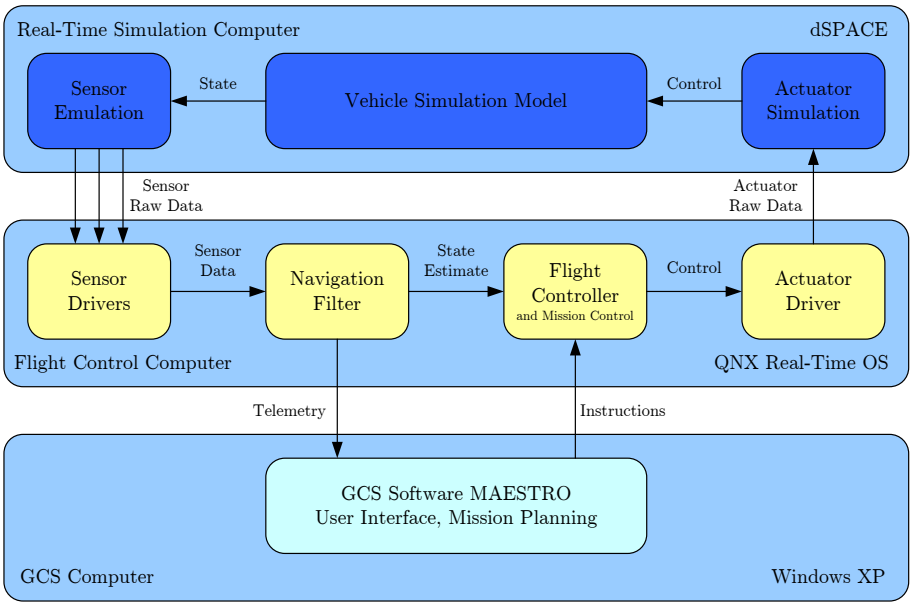

Bild 2.7: Hardware-in-the-Loop-Simulation

Neben den offensichtlichen Vorteilen der ARTIS-Systemsimulation weist diese in der bisherigen Form allerdings auch zwei Nachteile auf. Wenn alle Komponenten vollständig geprüft werden sollen, ist die Systemsimulation sehr zeitaufwändig bis hin zu praktisch nicht durchführbar. Weiterhin mangelt es an einer automatischen Protokollierung der Versuchsdurchführung sowie einer generellen systematischen Methodik.

Die einzelnen Komponenten müssen somit schon vor der Integration ausführlich getestet werden. Im Rahmen dieser Diplomarbeit soll daher untersucht werden, wie sich die Komponenten des Entscheidungssystems vorab auf ihre Korrektheit prüfen lassen. Das zu entwickelnde Testkonzept muss dabei die genannten Nachteile der Systemsimulation kompensieren. Dies umfasst:

- einen vollständiger Testdurchlauf aller möglichen Situationen,
- einen minimalen zeitlichen Testaufwand sowie
- eine automatische Protokollierung von Ursache und Wirkung im Fehlerfall.

Die Umsetzung des Testkonzepts soll, wie die eigentliche Systemerstellung selbst, modellbasiert erfolgen.

2.2 Ereignisbasierte Systeme

2.2.1 Ereignis und Benachrichtigung

In der Informatik wird der Begriff Ereignis je nach dessen Kontext differenziert betrachtet. Zum Beispiel werden Ereignisse bei der Programmierung von grafischen Benutzeroberflächen (GUI's) eingesetzt. Eine Applikation wartet dort solange, bis sie von dem Betriebssystem über eine Aktivität des Benutzers (z.B. Mausbewegung) benachrichtigt wird. Der Applikation steht es danach offen, ob bzw. wie sie diese Benachrichtigung verarbeitet.

Weiterhin wird der Begriff Ereignis auch bei der Prozesssynchronisation (Semaphore, Mutexe) verwendet. In diesem Zusammenhang sind Ereignisse Variablen, die ein Programm setzen bzw. ein anderes auslesen kann. Die Studie [26] zeigt in diesem Zusammenhang Vor- und Nachteile auf.

Eine dritte Möglichkeit ist der Einsatz von so genannten Subject-Observer-Pattern im objektorientierten Design. Der Observer sendet dem Subject eine Beschreibung über Ereignisse, die ihn selber interessieren. Wenn das Subject ein solches Ereignis detektiert, wird der Observer benachrichtigt. Dies erfolgt oftmals durch das Aufrufen einer entsprechenden Methode des Observers.

Ferner wird der Begriff Ereignis in verschiedenen Programmiersprachen eingesetzt. Beispielsweise wird in Java ein Ereignis als Objekt genutzt, wenn die Ereignis-Quelle (Source) den -Empfänger (Listener) benachrichtigt.

Trotz der Vielzahl von Anwendungsmöglichkeiten wurde in dem Grundlagenkapitel für ereignisbasierter Programmierung [3] versucht, die beiden Begriffe Ereignis und Benachrichtigung mit den Definitionen 2.1 und 2.2 einheitlich zu erklären.

Definition 2.1 *Ein Ereignis ist eine detektierbare Bedingung, welche eine Benachrichtigung durch den Sender auslöst.*

Definition 2.2 *Eine Benachrichtigung, welche durch ein Ereignis ausgelöst ist, wird zu einem zur Laufzeit festgelegten Empfänger übertragen.*

Eine Trennung der beiden Definitionen ist an dieser Stelle nicht möglich, da es sich um ein zweistufiges Verfahren handelt. Im Software-Kontext ist das Ereignis somit die Ursache und die Benachrichtigung die Folge (wie in Bild 2.8 dargestellt).

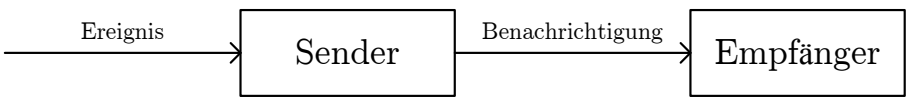

Bild 2.8: Beziehung zwischen Sender und Empfänger

Die Abfrage einer solche Ereignis-Bedingung durch den Sender kann nur boolesche Werte (wahr oder falsch) als Ergebnis liefern. Wird anschließend eine Benachrichtigung für den Empfänger ausgelöst, wird die detektierte Bedingung als Ereignis assoziiert andernfalls nicht. Dieser Prozess muss jedoch deutlich im Modell (Code) erkennbar sein. Das Listing 2.1 zeigt beispielhaft, wie bei einem Schleifendurchlauf der Wert drei für die Schleifenvariable detektiert wird. Anschließend findet der Benachrichtigungsmechanismus über einen Event Handler statt.

```
void Control::Do()
{
    for (int x = 0; x < 10; x++)
    {
        if (x == 3) this->eventHandler("Third iteration");
    }
}
```

Listing 2.1: Detektierung mit Benachrichtigungsmechanismus

Der Empfänger muss nicht zwangsläufig auf die Benachrichtigung reagieren. Falls doch, ist es wichtig, mögliche Kettenreaktionen zu betrachten. Wenn zum Beispiel der Erhalt der Benachrichtigung eine Detektierung einer Bedingung für den Empfänger bedeutet, kann dieser wiederum als Sender einer Benachrichtigung für einen nachfolgenden Empfänger agieren. Somit wird mit der ersten Detektierung eine Folge von Benachrichtigungen über hintereinander geschaltete Stationen ausgelöst.

2.2.2 Ereignisanforderung

Es ist folglich zu untersuchen, wie die Verbindung zwischen Sender und Empfänger aufgebaut wird. Daher wird der Begriff Ereignisanforderung wie folgt definiert.

Definition 2.3 *Unter der Ereignisanforderung wird jener Prozess verstanden, der einen Sender mit einem Empfänger zur Ereignisbenachrichtigung verbindet und dabei die Konfiguration der Benachrichtigungsform festlegt.*

In diesem Prozess veröffentlicht zunächst der Sender eine Liste mit Ereignissen, die dieser detektieren kann. Der Empfänger identifiziert anschließend die Ereignisse, an denen er interessiert ist und überträgt diese als Antwort. Der Sender speichert die Antwort des Empfängers und überträgt die entsprechende Benachrichtigung, wenn ein angefordertes Ereignis detektiert wird. Der Prozess der Ereignisanforderung findet zur Laufzeit vor der Ereignisbenachrichtigung statt, jedoch nicht zur Übersetzungszeit.

Die Ereignisanforderung kann auf vier verschiedenen Varianten erfolgen, die in Bild 2.9 zusammengefasst sind. Es ist möglich, eine Middleware (M) zwischen Sender (S) und Empfänger (E) zu schalten.

Weiterhin kann der Anforderungsprozess über einen separaten Binder (B) erfolgen. Dieser wird so bezeichnet, weil er Sender und Empfänger unabhängig von der Middleware „verbindet“. Der Binder wird an dieser Stelle vorgestellt, um die Integration der Bodenstation für eine dynamische Lösung im Flugversuch später zu erläutern.

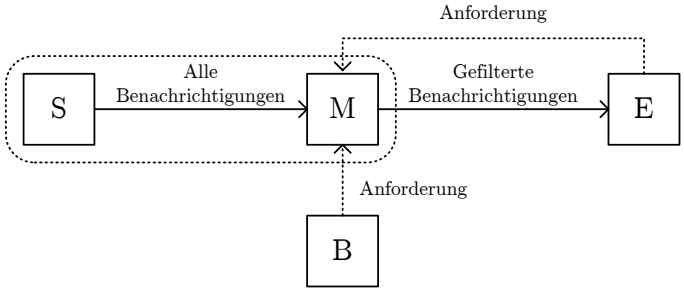

Bild 2.9: Methodik der Ereignisanforderung

Die elementarste Ereignisanforderung besteht nur aus Sender und Empfänger und stellt ein direktes Empfangsmodell dar. Schaltet man zwischen Sender und Empfänger eine Middleware, wird dies als indirektes Empfangsmodell bezeichnet. Der Einsatz eines Binders kann sowohl in dem direkten als auch in dem indirekten Empfangsmodell erfolgen.

2.2.3 Anforderungsklassifikation

Die Ereignisanforderung kann, wie in Bild 2.10 dargestellt, klassifiziert werden. Diese in [3] vorgeschlagene Möglichkeit, welche nur eine von Vielen ist, gliedert sich in die folgenden Kategorien: Kanal, Typ, Filter und Gruppe.

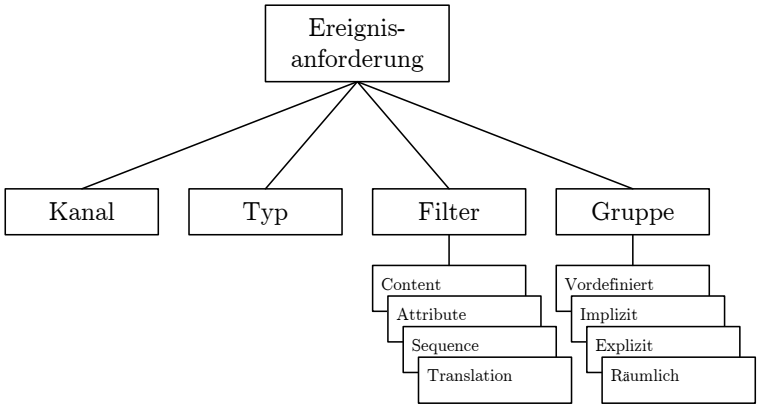

Bild 2.10: Anforderungsklassifikation

Die Kategorie Kanal wird in [3] mit der Definition 2.4 erklärt:

Definition 2.4 *Der Kanal beschreibt den physikalischen oder abstrakten Weg, wie die Benachrichtigung den Empfänger erreichen soll.*

Der Kanal bildet somit die Grundlage für die Benachrichtigungsübertragung und ist weiterhin von zentraler Bedeutung, um die angesprochene Middleware transparent für Sender und Empfänger zu gestalten. Es ist möglich, alle Benachrichtigungen über einen Kanal (Single Channel) zu übertragen oder die Benachrichtigungen über mehrere Kanäle (Multiple Channels) zu versenden. Des Weiteren kann unterschieden werden, ob die Benachrichtigung an garantiert nur einen Empfänger (Unicast) versendet wird, oder an mehrere Empfänger (Broadcast). In diesem Zusammenhang wird später die Kategorie Gruppe erläutert.

Das Verteilung der Benachrichtigungen auf die Kanäle (Channelization) wird entweder von dem Sender (direktes Empfangsmodell) oder von der Middleware (indirektes Empfangsmodell) realisiert. Es können dabei mehrere Typen von Benachrichtigungen auf einem Kanal, oder aber je Kanal nur ein Typ übertragen werden.

Die Kategorien Typ und Filter werden mit den Definitionen 2.5 und 2.6 abgegrenzt.

Definition 2.5 *Der Begriff Benachrichtigungstyp bezeichnet eine Menge von Objekten mit gemeinsamen Eigenschaften, welche durch eine zugehörige Klasse definiert sind. Differenzierte Typen werden durch unterschiedliche Eigenschaften symbolisiert.*

Definition 2.6 *Ein Filter ermöglicht eine Auswahl von Benachrichtigungen nach verschiedenen Kriterien (Filtering Expressions).*

Die Übertragung dieser Benachrichtigungen kann sowohl über einen Kanal als auch über mehrere Kanäle erfolgen. Der Filter wird bei komplexen Systemen oft als separate Komponente zu dem Sender erstellt (Bild 2.11), da das Filtern der Benachrichtigungen recht aufwändig sein kann. Dieser Benachrichtigungsvorgang entspricht dem indirekten Empfangsmodell aus Bild 2.9.

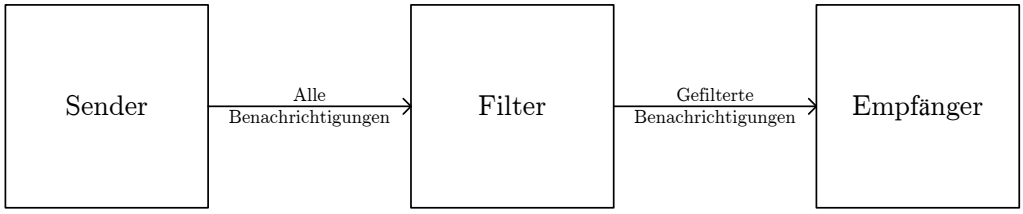

Bild 2.11: Teilung von Sender und Filter

Die Kategorie Filter wird dabei in die Unterkategorien Content, Attribute, Sequence und Translation Filtering unterteilt. Das Content Filtering setzt voraus, dass der Inhalt der Benachrichtigungen über eine Menge von Wörtern oder Ausdrücken beschrieben werden kann. Ein solches Filtern benötigt jedoch eine hohe Rechenleistung und wird deshalb nur bei Systemen mit wenigen Empfängern eingesetzt.

Das Attribute Filtering setzt ein Klassifikation der verschiedenen Benachrichtigungen durch eine Menge von Attributen voraus. Die Attribute können einen mehrdimensionalen Charakter aufweisen.

Es werden zwei verschiedene Techniken unterschieden: Die Klassifikation findet schon zur Übersetzungszeit statt oder die Klassifikation wird „on-the-fly“ durchgeführt. Attribute Filtering besitzt bei eindeutiger Klassifikation eine sehr gute Performance und wird deshalb oft bei eingebetteten Systemen eingesetzt.

Das Sequence Filtering kann auf zwei verschiedene Weisen erfolgen. Zum Einen kann Interesse bestehen eine Sequenz von Benachrichtigungen abzufragen, die in einer zeitlichen Abhängigkeit stehen. Zum Anderen kann aber auch die Frequenz von auftretenden Benachrichtigungen reduziert werden. Tritt zum Beispiel eine Ereignis 60-mal pro Minute auf, kann durch Festlegung einer maximalen Frequenz von einer Benachrichtigung pro Minute der Filter konfiguriert werden.

Das Translation Filtering wird auch als Benachrichtigungstransformation bezeichnet. Dieses Filtern betrifft den Inhalt (Umwandlung in andere Sprachen), den Typ (Umwandlung ähnlicher Typen zu einem gemeinsamen Typ) und die Sequenz, bei welcher elementare Benachrichtigungen zu einer zusammengesetzten Benachrichtigung transformiert werden.

Abschließend wird in der Anforderungsklassifikation die Kategorie Gruppe betrachtet. Diese wird mit der Definition 2.7 erklärt:

Definition 2.7 *Eine Gruppe repräsentiert Empfänger mit den gleichen Ereignisanforderungen an einen Sender.*

Eine Gruppe kann somit auch als ein virtueller Empfänger betrachtet werden, wobei sich verschiedene Gruppen auch überlappen können. Das Bild 2.12 zeigt dazu beispielhaft die Gruppe 1 und 2. Die Gruppe 1 fordert die Benachrichtigungen A, B, C und D an, die Gruppe 2 dagegen C, D und E. Somit überlappen sich die Gruppen 1 und 2 durch die gemeinsame Anforderung der Benachrichtigungen C und D.

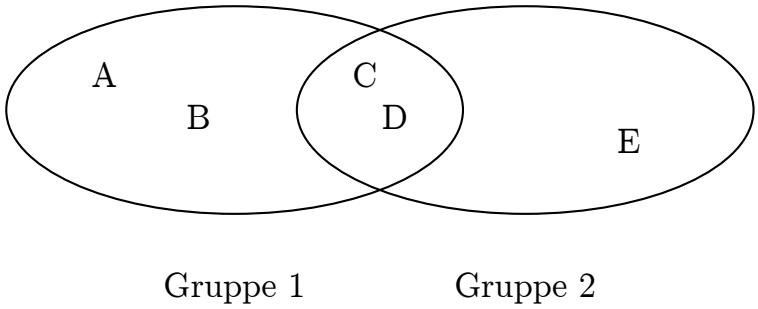

Bild 2.12: Überlappende Gruppen

In der Kategorie Gruppe findet eine Unterteilung in vordefiniert, implizit, explizit oder räumlich statt. Eine vordefinierte Gruppe ist dem Sender bereits zur Übersetzungszeit bekannt. Implizite Gruppen werden von dem Sender automatisch durch eine Analyse der Ereignisanforderung erkannt. Die einzelnen Empfänger werden über die Gruppierung jedoch nicht informiert.

Die Empfänger können mit einer Systemfunktion zur Laufzeit explizite Gruppen bilden und ihnen als Mitglieder beitreten, wobei eine solche Gruppe im System als virtuelle Middleware zu betrachten ist. Räumliche Gruppen sind durch mobile Empfänger charakterisiert, welche durch ihre geografische Position die gleichen Ereignisanforderungen stellen.

2.2.4 Benachrichtigungsübertragung

Die Benachrichtigungsübertragung kann auf unterschiedliche Weise erfolgen. Eine Klassifikation dieser Möglichkeiten ist nach [3] in Bild 2.13 dargestellt. Es werden dabei zwei grundsätzliche Tendenzen skizziert: Zum Einen findet eine Benachrichtigung über gemeinsam genutzte Ressourcen (Dateien, Speicher, Objekte usw.) statt, zum Anderen werden lokale bzw. entfernte Funktionsaufrufe genutzt.

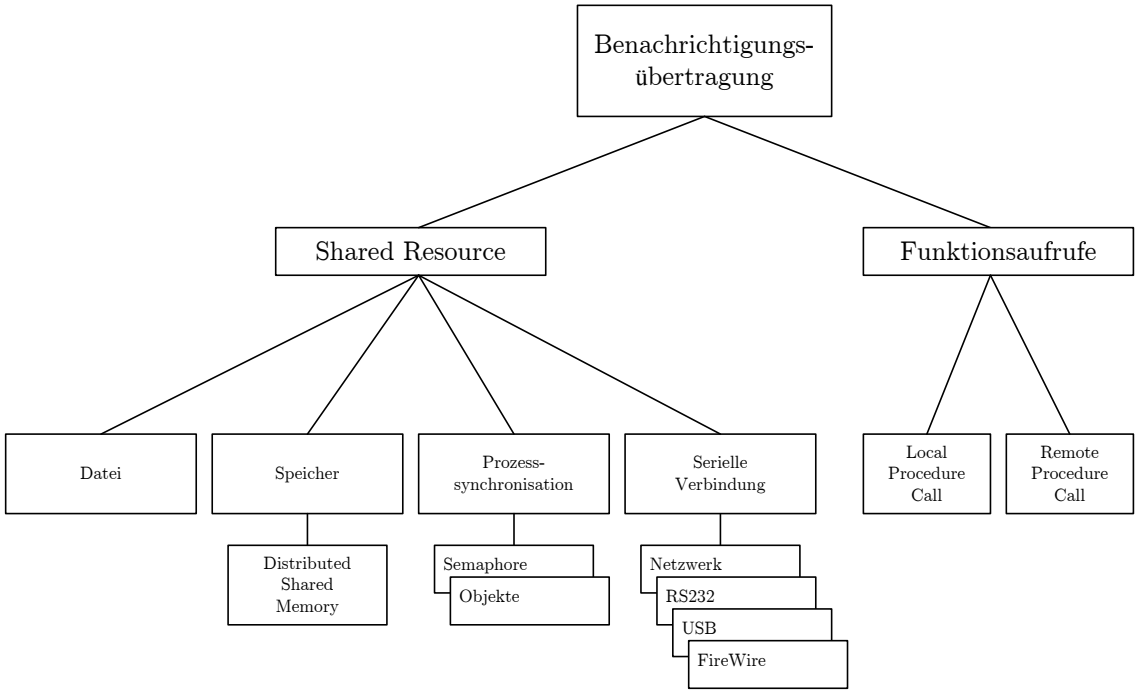

Bild 2.13: Klassifikation der Benachrichtigungsübertragung

In den Fällen der gemeinsam genutzten Ressourcen muss darauf geachtet werden, dass nicht gleichzeitig vom Sender und Empfänger die angesprochene Ressource beschrieben bzw. ausgelesen wird. Es werden in diesem Zusammenhang Prozesssynchronisationstechniken (unter anderem Semaphore oder Mutexe) eingesetzt.

Wird eine Datei als Benachrichtigungsform genutzt, ergeben sich zwei unterschiedliche Möglichkeiten der Nutzung. Einerseits kann der Sender die Daten in eine neue Datei schreiben und im Anschluss schließen, so dass der Empfänger nach dem Öffnen der Datei die Daten auslesen kann. Die Datei wird nach der fertiggestellten Verar-

beitung durch den Empfänger gelöscht. Andererseits kann der Sender neue Daten einer Datei anhängen, wobei das Löschen der Datei durch den Empfänger entfällt.

Soll allerdings der gemeinsame Speicher (Shared Memory), zum Beispiel bei Multi-Tasking-Systemen als Benachrichtigungsform genutzt werden, ergibt sich ein ähnliches Vorgehen. Der eine Prozess sperrt den zu nutzenden Bereich, schreibt die Daten dort hinein und hebt anschließend die Sperrung wieder auf. Der andere Prozess kann daraufhin die Daten mit erneuter Sperrung auslesen.

Eine dritte Form der Benachrichtigung ist die Nutzung von Benachrichtigungsobjekten. Diese werden in dem Sender erzeugt und mit Daten bzw. gegebenenfalls sogar mit vorher definierten Funktionen versehen. Dieses Objekt wird dem Empfänger übermittelt und anschließend dort verarbeitet.

Weiterhin können zur Übertragung auch serielle Verbindungen (zum Beispiel die RS232-Schnittstelle) genutzt werden. Der Sender übergibt die Daten an seinen Sendepuffer, die Schnittstellenhardware sorgt für den Transport zum Empfänger und dieser kann abschließend die Daten aus seinem Empfangspuffer auslesen.

Neben der Nutzung von gemeinsamen Ressourcen, ist aber auch der Aufruf von Funktionen eine Möglichkeit Benachrichtigungen zu übertragen. Entweder werden in einem Programm gemeinsame Funktionen zwischen den verschiedenen Threads genutzt oder es werden externe Funktionen zum Beispiel mit RPC[2] aufgerufen.

2.2.5 Modellierung ereignisbasierter Systeme

In der Entwurfsphase zur Modellierung eines ereignisbasierten Systems sollten unter anderem die folgenden Fragen gestellt werden:

- Können Benachrichtigungen geändert werden?
- Können Benachrichtigungsanforderungen geändert werden?
- Können Benachrichtigungsanforderungen abgebrochen werden?
- Wie viele verschiedene Benachrichtigungsanforderungen darf ein einzelner Empfänger dem Sender übermitteln?

Die Beantwortung der Fragen beeinflusst das Systemkonzept maßgeblich, da der Grad der Dynamik für das ereignisbasierte System an dieser Stelle festgelegt wird.

[2]Das Remote Procedure Protocol ist eine Technik zur Netzwerkkommunikation des ISO-OSI-Modells, mit deren Hilfe Funktionsaufrufe auf entfernten Rechnern ausgeführt werden.

Ereignisbasierte Systeme werden vorwiegend mit UML-Diagrammen (Unified Modeling Language) modelliert. Es besteht auch die Möglichkeit, Petrinetze sowie Lollipop-, Espresso- und Catalysis-Diagramme einzusetzen. Die einzelnen Ansätze werden in [3] übersichtlich skizziert und erläutert.

Es existieren auch Ansätze zur Modellierung und Verifikation mit der Specification and Description Language (SDL), der Event-Driven Testing Model and Language (eTest ML [13]) und der Executable Architecture Definition Language (EADL [21]).

Die dreizehn UML-Diagramme werden prinzipiell in Struktur- und Verhaltensdiagramme unterschieden, um die statischen und dynamischen Aspekte darstellen zu können. Eine besondere Bedeutung wird den folgenden UML-Verhaltensdiagrammen bei der Modellierung ereignisbasierter Systeme zugesprochen:

- Mit den **Aktivitätsdiagrammen** lassen sich einfache Anwendungsfälle darstellen, ohne den internen Ablauf spezifizieren zu müssen.
- Die **State Charts** repräsentieren die Objektzustände der modellierten Komponente mit dazugehörigen Zustandsübergängen.
- Die **Sequenz-** und **Kommunikationsdiagramme** zeigen die grafische Darstellung des Nachrichtenaustausches, wobei bei den Sequenzdiagrammen die zeitliche Reihenfolge der Kommunikation berücksichtigt wird.
- **Interaktionsübersichtsdiagramme** kombinieren den Nachrichtenaustausch und die damit verbundenen Aktionen in einem Diagramm. Durch Modularisierung wird die Übersichtlichkeit bei komplexen Systemen in diesem Diagrammtyp verbessert.

Im Fokus dieser Arbeit steht die Modellierung mit den UML State Charts, da hierfür ein umfangreicher Tool Support gegeben ist und UML zur Spezifikation bereits genutzt wird. Eine weitere Arbeit[3] in dem DLR-Institut für Flugsystemtechnik befasst sich, parallel zu dieser Arbeit, mit der Modellierung über Petrinetze.

[3] In Zusammenarbeit mit dem Institut für Verkehrssicherheit und Automatisierungstechnik der TU Braunschweig.

2.3 UML State Charts

UML State Charts sind Zustandsautomatendiagramme mit einer endlichen Menge an Zuständen, wobei zu jedem Zeitpunkt genau ein Zustand in jedem State Chart aktiv ist. Die State Charts wurden erstmals von D. Harel 1988 in [6] spezifiziert. Das Anwendungsgebiet der State Charts umfasst vor allem Verhaltens- und Testspezifikation. Eine Klassifikation erfolgt in abstrakte und konstruktive State Charts, wobei deren Übergänge fließend sind. Abstrakte State Charts werden zu Dokumentationszwecken eingesetzt, konstruktive State Charts zur Code-Generierung.

Wichtige Elemente des State Charts sind die Zustände selbst, mit ihren internen Aktivitäten, Pseudozustände wie der Initial-, der End- und der History-Zustand sowie die Zustandsübergänge, welche auch als Transitionen bezeichnet werden.

2.3.1 Die Syntax

Zunächst sollen die wichtigsten Begriffe definiert werden. Die Definitionen 2.8 bis 2.11 entstammen dem Grundlagenkapitel agiler Modellierung mit UML [17]:

Definition 2.8 *Ein Zustand repräsentiert eine Teilmenge der möglichen Objektzustände und wird durch einen Zustandsnamen und je eine optionale entry-Aktion, exit-Aktion und do-Aktivität modelliert. Dieser Zustand kann mehrere Teilzustände enthalten, hierarchisches Modellieren ist daher möglich.*

Definition 2.9 *Ein Zustand kann eine entry-Aktion beinhalten, die ausgeführt wird, wenn der Zustand betreten wird. Analog dazu kann ein Zustand eine exit-Aktion beinhalten, die ausgeführt wird, wenn der Zustand verlassen wird.*

Definition 2.10 *Ein Zustand kann eine permanent andauernde Aktivität beinhalten, die do-Aktivität genannt wird. Verschiedene Mechanismen zur Auslösung können dabei eingesetzt werden.*

Definition 2.11 *Eine Transition führt von einem Quellzustand in einen Zielzustand und beinhaltet eine Beschreibung der Aktivierung sowie deren Reaktion in Form einer Aktion.*

Der angesprochene eindeutige Name eines Zustandes befindet sich meist im Kopf der grafischen Darstellung (Bild 2.14). Weitere interne Aktivitäten werden durch Transitionen gekennzeichnet, deren Quell- und Zielzustand identisch sind. Die Aktivierung einer Transition erfolgt optional durch einen Benachrichtigungstrigger für ein Ereignis (Event) oder durch eine Wächterbedingung (Guard), die ebenfalls eine booleschen Bedingung, z.B. für die Werte lokaler Variablen des Objektes, prüft.

Zusätzlich kann die Transition mit einer weiteren Aktivität verbunden werden, welche zwischen exit-Aktivität des einen und der entry-Aktivität des folgenden Zustandes geschaltet ist und somit diesen Zustandsübergang spezialisiert. Weiterhin wird in Bild 2.14 der Initialzustand (linke Seite) bzw. der optionale Endzustand (rechte Seite) gezeigt.

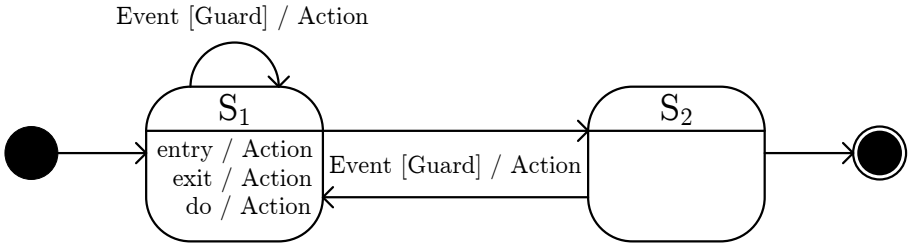

Bild 2.14: Grafische State Chart Syntax

Der interne Ablauf für einen Zustand wird in Bild 2.15 explizit dargestellt. Zunächst wird die entry-Aktion bei dem Betreten des Zustandes vollständig ausgeführt. Anschließend wird gewartet, bis der Trigger t eine Aktivierung der anderen internen Aktionen feststellt. Durch die Benachrichtigung t(e) wird die exit-Aktion dieses Zustandes und die entry-Aktion des Folgezustandes ausgeführt. Dagegen führt t(d) die do-Aktivität aus, wobei in diesem Fall derselbe Wartepunkt anschließend wieder erreicht wird. Transitionen mit dem „leeren Wort“ ε werden sofort geschalten, es bedarf in diesem Fall keiner separaten Aktivierung.

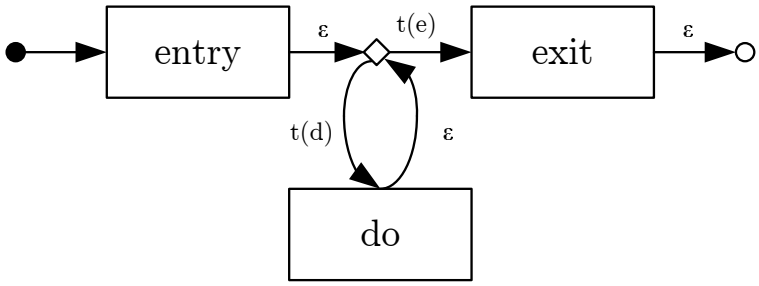

Bild 2.15: Interner Ablauf eines Zustandes

Wenn ein getakteter Trigger über eine Benachrichtigung Aktivitäten auslöst, muss folglich wieder ein Wartepunkt vor Beginn des Folgetaktes erreicht werden.

2.3.2 Zusammengesetzte Zustände

Durch Kombination der Grundelemente lassen sich zusammengesetzte Zustände bilden. So ist es möglich, jeden einzelnen Zustand wieder in Verbindung mit einem neuen State Chart zu bringen. Das Bild 2.16 zeigt diese Kombinationsmöglichkeit für den Zustand S_2. Beim Betreten von S_2 wird der interne Initialzustand aktiviert und die Default-Transition zu dem Teilzustand S_{21} ausgelöst.

Bei entsprechender Aktivierung kann anschließend intern zwischen S_{21} und S_{22} gewechselt werden. Löst ein Trigger die externe Transition $S_2 \rightarrow S_1$ aus, wird entweder von S_{21} oder von S_{22} zu S_1 gewechselt, wobei im selben Schritt die Exit-Aktivität von S_2 ebenfalls ausgeführt wird.

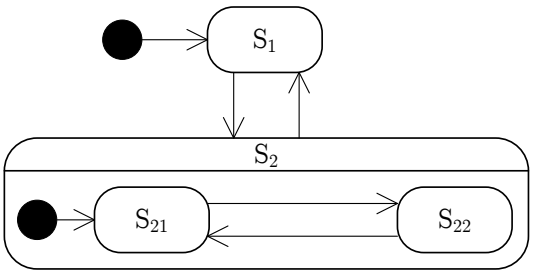

Bild 2.16: Zusammengesetzter Zustand am Beispiel

Für den zusammengesetzten Zustand S_2 ist der interne Ablauf in Bild 2.17 dargestellt. Zu beachten ist, dass der Trigger für die Teilzustände S_{21} und S_{22} auch auf Benachrichtigungen für S_2 erwartet und somit gegebenenfalls dessen Do-Aktivität ausführen kann. Die Teilzustände „erben“ somit die Transitionseigenschaften von dem Zustand S_2. Es wird daher kein expliziter Endzustand benötigt, um einen Teilzustand von S_2 in Richtung S_1 zu verlassen.

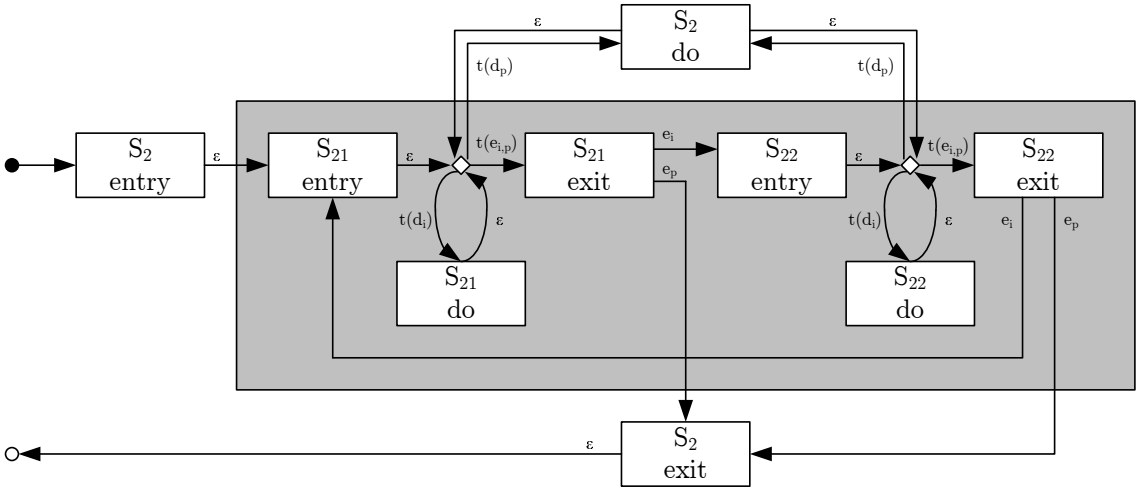

Bild 2.17: Interner Ablauf eines zusammengesetzten Zustandes (1)

Je nach getakteter Benachrichtigung und Wartepunkt ergeben sich somit die folgenden potentiellen Abläufe.

- ◇ S_1 exit → S_2 entry → S_{21} entry
- ◇ S_{21} exit → S_2 exit → S_1 entry
- ◇ S_{22} exit → S_2 exit → S_1 entry
- ◇ S_2 do
- ◇ S_{21} do
- ◇ S_{22} do
- ◇ S_{21} exit → S_{22} entry
- ◇ S_{22} exit → S_{21} entry

Mit jeder weiteren Hierarchieebene nimmt die Länge dieser Ablaufsequenzen zu. Diese Tatsache schränkt die Anwendung für getaktete Systeme ein, da ab einen gewissen Punkt nicht mehr garantiert werden kann, dass der nächste Wartepunkt tatsächlich vor Beginn des nächsten Taktes erreicht wird.

Wird der Zustand S_2 lediglich als logische Einheit modelliert, d.h. entry-, exit- und do-Aktivität sind leere Funktionen, ergibt sich der interne Ablauf aus Bild 2.18.

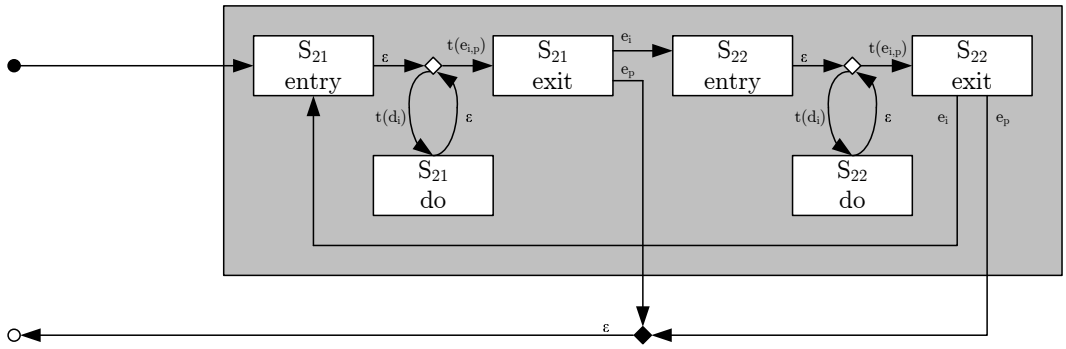

Bild 2.18: Interner Ablauf eines zusammengesetzten Zustandes (2)

Die Länge der Ablaufsequenzen nimmt in diesem Fall nicht zu.

- ◇ S_1 exit → S_{21} entry
- ◇ S_{21} exit → S_1 entry
- ◇ S_{22} exit → S_1 entry
- ◇ S_{21} do
- ◇ S_{22} do
- ◇ S_{21} exit → S_{22} entry
- ◇ S_{22} exit → S_{21} entry

Daher kann das hierarchische Beispiel aus Bild 2.16 auch nicht hierarchisch modelliert werden, wobei sich eine neue Transition in dem Modell ergibt (Bild 2.19).

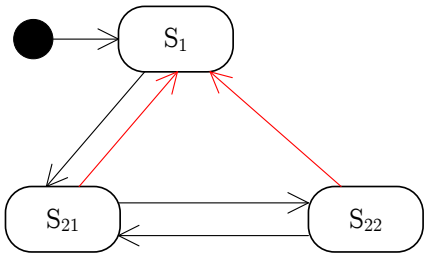

Bild 2.19: Projektion eines zusammengesetzten Zustandes am Beispiel

Zusammengesetzte Zustände besitzen daher bei der Modellierung Vor- und Nachteile. Der Vorteil liegt in der Reduktion der Komplexität, da weniger Transitionen in dem Modell benötigt werden. In dem Beispiel aus Bild 2.16 wurde gezeigt, wie die Teilzustände S_{21} und S_{22} die Transitionseigenschaften von S_2 „erben“. Weiterhin lassen sich in verschiedenen Programmen die Teilzustände je nach Bedarf in verschiedenen Sichten ein- und ausblenden.

Nachteilig ist insbesondere für das Testen, dass die Testfälle, welche nur auf Basis des Modells entworfen werden, nicht vollständig den generierten Code testen, da der interne Ablauf nicht berücksichtigt wird. Wenn beispielsweise von S_{21} zu S_1 gewechselt wird, ist die Transition $S_2 \rightarrow S_1$ getestet. Der Vollständigkeit halber fehlt allerdings der Testfall von S_{22} zu S_1.

Mit der Definition 2.12 wird der History-Zustand basierend auf [17] erläutert.

Definition 2.12 *Der History-Zustand ist ein Teilzustand, welcher bei einem Ersteintritt in einen zusammengesetzten Zustand als eigentlicher Initialzustand wirkt. Bei einem Wiedereintritt nimmt der History-Zustand jedoch die Rolle des zuletzt aktiven Zustandes ein.*

Das Beispiel für einen zusammengesetzten Zustand aus Bild 2.16 wird in dem Bild 2.20 erweitert, indem anstatt eines Initialzustandes der nun definierte History-Zustand in der zweiten Hierarchieebene verwendet wird. Es ist somit beispielsweise die folgenden Zustandssequenz möglich: $S_1 \rightarrow S_{21} \rightarrow S_{22} \rightarrow S_1 \rightarrow S_{22}$.

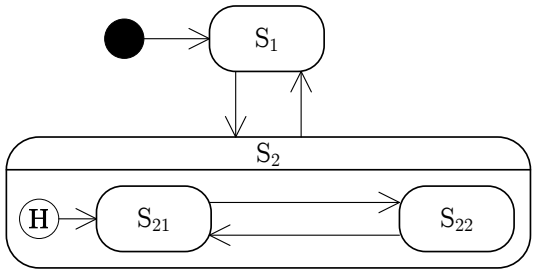

Bild 2.20: History-Zustand am Beispiel

Das heißt, dass bei Wiedereintritt in S_2 nicht zwangsläufig der Teilzustand S_{21} betreten werden muss. Intern wird dies durch eine lokale Variable realisiert, mit welcher bei dem Betreten von S_2 die entsprechende Transition zu der entry-Aktion des Teilzustandes ausgewählt werden kann (Bild 2.21).

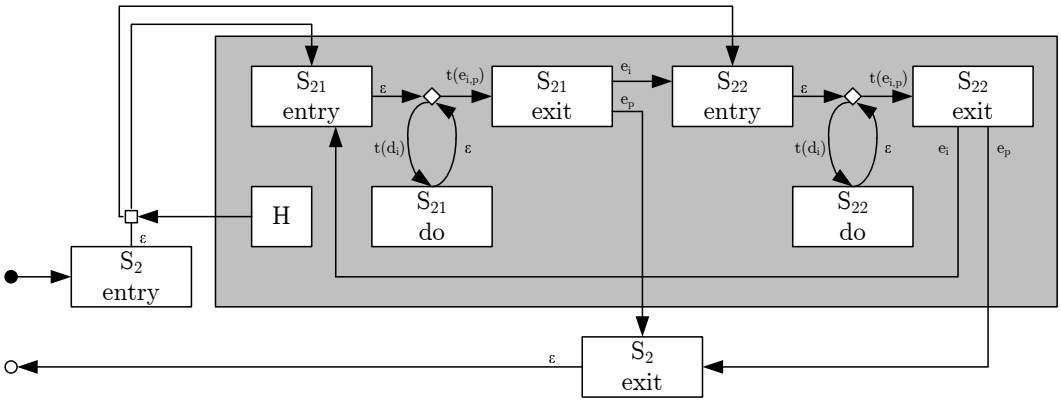

Bild 2.21: Interner Ablauf eines History-Zustandes

2.3.3 Code-Generierung

Die Code-Generierung erfolgt im modellbasierten Kontext in zwei unterschiedlichen Varianten (Bild 2.22). Zum Einen sind flache State Charts möglich, die sich auf Grund der optimierten Form für eingebettete Systeme eignen. Per Code-Generierung wird nur eine Klasse mit allen dazugehörigen Funktionen erzeugt. Dieser Klasse gehört ein globaler Benachrichtigungsverteiler an, der bei Aktivierung die entsprechende Funktion des aktuellen Zustands aufruft.

Bei flachen State Charts wird die modellierte Struktur weitgehend zusammengefasst. Dadurch reduziert sich die Anzahl der Funktionen, allerdings treten im Quellcode Redundanzen auf. Beispielsweise wird für die Exit-Aktion des einen und die Entry-Aktion des folgenden Zustands nur eine Funktion generiert. Diese Exit-Aktion erscheint später an mehreren Stelle im Code, da ein Zustand oft mit mehreren Folgezuständen verbunden ist.

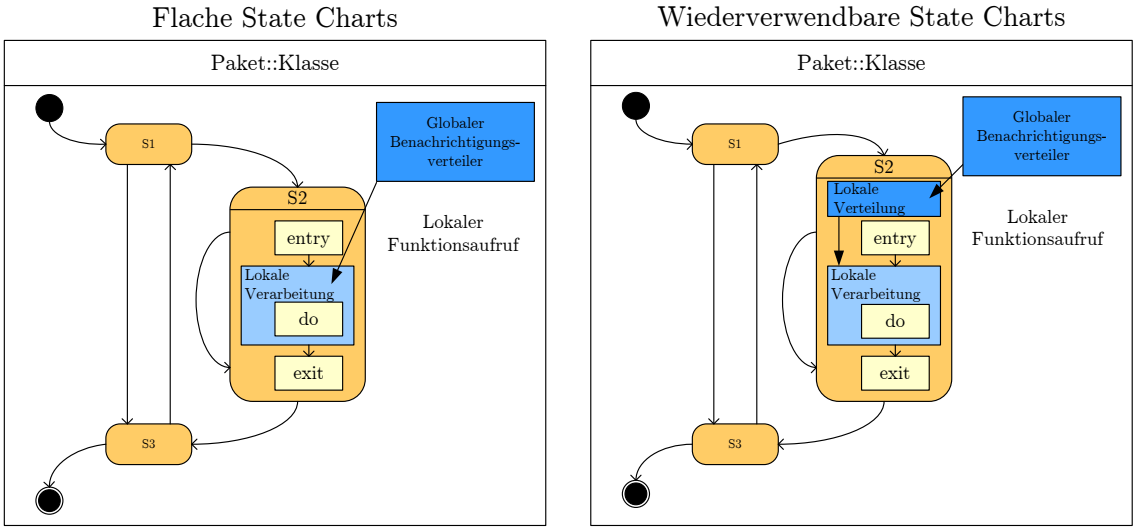

Bild 2.22: Flache versus wiederverwendbare State Charts

Wiederverwendbare State Charts sind durch ihr ausgeprägtes objektorientiertes Design gekennzeichnet. Es wird nicht nur eine Klasse, wie bei den flachen State Charts generiert, sondern eine Klasse für die State Chart Struktur sowie jeweils eine weitere Klasse für jeden einzelnen Zustand. Der globale Benachrichtigungsverteiler der State Chart Struktur übergibt bei einer Aktivierung das Benachrichtigungsobjekt an den lokalen Verteiler der aktiven Zustandsklasse. Dieser wählt daraufhin die entsprechende Funktion zur Bearbeitung des Objekts in der aktiven Klasse aus.

Die wiederverwendbaren State Charts sind für die Software-Entwickler nachvollziehbarer, da die Modellstruktur beibehalten wird. Änderungen können daher via Roundtrip-Verfahren, welche dazu dienen, den generierten Code bei manueller Veränderung wieder mit dem Modell abzustimmen, übersichtlich eingepflegt werden, ohne den Code auf Redundanzen, wie bei flachen State Charts, zu prüfen.

Bei der Design-Entscheidung muss somit festgelegt werden, ob der Schwerpunkt auf die Ausführungsgeschwindigkeit der flachen State Charts oder auf die Lesbarkeit von wiederverwendbaren State Charts gelegt wird. Werden die Schnittstellen abstrakt formuliert, können beide Varianten verwendet werden, so dass der wiederverwendbare State Chart zur Entwicklung und der flache State Chart zur Auslieferung des Systems eingesetzt werden kann.

2.3.4 Modellierungskonflikte

Schon während der Modellierung können verschiedene Fehler auftreten. An dem Beispiel aus Bild 2.23 sollen verschiedene Probleme erläutert werden. Es handelt sich dabei um das vereinfachte Modell eines CD-Players, welcher an- und ausschaltbar ist, Musik abspielt (Play) sowie vorwärts (Forward) und rückwärts (Reverse) spulen kann.

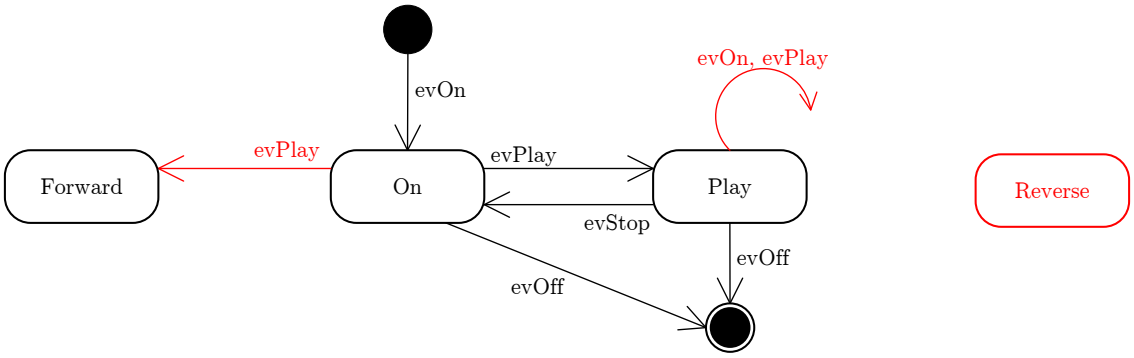

Bild 2.23: Probleme bei der State Chart Modellierung

Mögliche Modellierungsfehler sind unter anderem isolierte oder unerreichbare Zustände, die entweder mit keinem anderen Zustand verbunden sind, oder aber die Verbindung über Ereignisse und Wächterbedingungen nie schaltet, weil es sich um nie erfüllbare Ereignisse handelt und daher keine Benachrichtigung erfolgt. In dem Beispiel des CD-Players ist der Zustand Reverse mit keinem anderen Zustand verbunden, der Zustand Reverse ist somit isoliert.

Weiterhin können fehlende oder fehlerhafte Ereignistrigger und Wächterbedingungen potentielle Konflikte verursachen. Beispielsweise kann von dem Zustand On mit dem Benachrichtigung evPlay zu dem Zustand Play, aber auch zu dem Zustand Forward geschaltet werden. In diesem Fall ist der State Chart nicht deterministisch. Das Fehlen von einem Trigger kann zu einer Endlosschleife führen, da diese Transitionen sofort nach dem Betreten eines Zustandes aktiviert werden.

Ein generelles Grundproblem stellt das Testen dar, da theoretisch eine unendliche Menge an potentiellen Abläufen geprüft werden muss. In diesem Zusammenhang wird später auf die Zustands-, Transitions- und Pfadüberdeckung eingegangen.

Die Unvollständigkeit der Modellierung kann je nach Code-Generierungsvariante des Benachrichtigungsverarbeitungsmechanismus Konflikte verursachen. Es wird daher die Vollständigkeit mit der Definition 2.13 abgegrenzt:

Definition 2.13 *Ein State Chart heißt vollständig modelliert, wenn für jeden Zustand jede mögliche Benachrichtigung berücksichtigt wird.*

Eine vollständige Modellierung ist bei komplexen Systemen sehr aufwändig und nicht praktikabel. In dem Beispiel des CD-Players wird für den Zustand Play angedeutet, dass die Transitionen für die Benachrichtigungen evPlay und evOn fehlen. Das Beispiel ist daher nicht vollständig modelliert, so dass der Mechanismus der Benachrichtigungsverarbeitung nun von zentraler Bedeutung ist. Verhält sich der Mechanismus chaotisch, d.h. es wird in diesem Fall ein nicht modellierter Zustandswechsel vorgenommen (z.B. Zustandswechsel von Play zu Reverse bei Benachrichtigung evOn), ist der State Chart nicht deterministisch und für eingebettete Systeme unbrauchbar. Der Mechanismus muss daher für einen Zustand nicht modellierte Benachrichtigungen ignorieren und darf keinen Zustandswechsel durchführen. Diese Art der Code-Generierung ist gängige Praxis bei eingebetteten Systemen.

Kapitel 3

Re-Engineering des State Charts

Als Anwendungsbeispiel für die Modellierung und das spätere Testen dient die Ablaufsteuerung des Entscheidungssystems. Diese Komponente ist von zentraler Bedeutung bei der Missionssteuerung und erhält als Eingabe Statusinformationen des Modellhubschraubers, Direktkommandos zur Positions- und Geschwindigkeitssteuerung sowie Verhaltenssequenzen des Operators der mobilen Bodenstation. Diese Eingaben werden aufbereitet und an den nachgeschalteten Flugregler[1] als Ausgabe weitergereicht.

Bisher erfolgte die Entwicklung traditionell, d.h. diese Komponenten wurde zunächst separat modelliert und anschließend implementiert. Die Implementierung basiert dabei auf einem frei verfügbaren UML State Chart Template. Dieses traditionelle Vorgehen hat allerdings entscheidende Nachteile: Modell und Code driften auseinander, da keine Synchronisationsmöglichkeiten bestehen und der zeitliche Aufwand der Modellwartung oft zu groß ist. Das Modell stimmt daher selten mit dem Code überein und ist somit auch als Dokumentationsmittel ungeeignet.

Außerdem ist die UML State Chart Spezifikation in dem Template nicht vollständig umgesetzt, es fehlt an einem konsequenten hierarchischen Konzept, da zum Beispiel orthogonale oder zusammengesetzte Zustände nicht modellierbar sind. Da mit dem Template ohnehin schon sehr umfangreich Struktur und Ablauf per Hand festgelegt werden müssen, verursacht das fehlende hierarchische Konzept Redundanzen bei der Ereignisdetektierung. Strukturveränderungen sind daher praktisch sehr aufwändig umzusetzen.

Insgesamt ist die Ablaufsteuerung in der bisherigen Form sehr komplex, da Strukturinformationen, Ereignisdetektierung sowie der eigentliche funktionale Code in einer Klasse kombiniert sind. Das Testen der einzelnen genannten Teile ist somit nicht ohne weiteres möglich. Es kann bisher auch keine Aussage über den Überdeckungsgrad beim Testen bzw. in der Simulation getroffen werden. Die vorliegende Arbeit unternimmt den Versuch, die Komplexität durch ein Re-Engineering zu verringern.

[1] PID- bzw. PID2-Regler aus dem ARTIS-Forschungsbereich der Flugregelung.

Das traditionelle Vorgehen ist zwar für einfache Komponenten praktikabel, aber mit steigender Komplexität dominieren die genannten Nachteile den Vorteil einer schnellen Implementierung. Dieses Vorgehen ist kein (Industrie-)Standard, deshalb werden keine Lösungsmöglichkeiten für die Problematik im Zusammenhang mit dem Template angeboten. In diesem Kapitel wird daher beschrieben, wie das bisherige Vorgehen zu einen modellbasierten Ansatz durch Re-Engineering umstrukturiert wird.

3.1 Konzept

Bei der Umsetzung des Re-Engineerings können in diesem Fall keine Automatismen eingesetzt werden. Zwar unterstützen diverse Programme Reverse-Engineering, um zum Beispiel Klassendiagramme aus vorhandenem Code zu generieren, aber in Bezug auf State Charts und unter Verwendung dieses speziellen Templates sind solche Verfahren bisher nicht umgesetzt.

Ein automatisches Reverse-Engineering benötigt einen Textparser, welcher den Quelltext analysiert und vorher definierte Schlüsselwörter erkennt. Eine Klasse der Programmiersprache C++ ist nach ISO-Norm eindeutig definiert und weist bekannte Merkmale auf, die beispielsweise bei dem Reverse-Engineering für Klassendiagramme verwendet werden. Die State Charts können je nach deren Umsetzung nur schwer analysiert werden, da es hierfür keine eindeutige Normierung gibt und die einzelnen Definitionen der State Chart Komponenten erst für den Textparser aufbereitet werden müssen. Der Aufwand eines nicht-automatisierten Re-Engineerings wird in diesen Fall geringer eingeschätzt als die Vorbereitung und Umsetzung einer automatisierten Lösung.

Zu Beginn des Re-Engineerings wird mittels strukturierter Anforderungsanalyse die Spezifikationen an die zu modellierende Komponente zusammengetragen und als Anwendungsfälle formuliert. Die in den Studien [27] und [12] vorgestellte Methodik besteht dabei aus der Interviewtechnik (Projektleiter und Entwickler) und der Dokumentenanalyse von der ursprünglichen Spezifikation sowie dem aktuellen Quelltext. Eine ebenfalls vorgestellte Werkzeugunterstützung (z.B. Telelogic's DOORS) wird nicht verwendet, da die notwendigen Programme zu Beginn der Arbeit nicht vorlagen.

Nachfolgend sind als Beispiele zwei dieser Anwendungsfälle, in Zusammenhang mit dem Sicherheitspiloten der Bodenstation, beschrieben:

- *Der Missionsmanager darf nur dann aktiv werden, wenn der Sicherheitspilot auf „automatische“ Steuerung des Hubschraubers schaltet.*
- *Der Missionsmanager muss jederzeit in einen „Offline“ Zustand gebracht werden, sobald der Sicherheitspilot auf manuelle Steuerung umschaltet.*

Die Ergebnisse der Anforderungsanalyse werden mit dem in Kapitel 2.2 vorgestellten Konzept ereignisbasierter Systeme verknüpft, um die bekannten Schwächen des Templates durch strukturierte Modellierung zu eliminieren. Vor allem die hohe Komplexität soll mit diesem Schritt deutlich reduziert werden. Das Bild 3.1 zeigt schematisch das erstellte Modell, welches nachfolgend kurz erläutert wird.

Wenn der Sender bei den kontinuierlichen bzw. diskreten Eingabewerten aus den externen Datenquellen (z.B. GPS) Ereignisse detektiert, werden elementare Benachrichtigungen in Form gemeinsamer boolescher Variablen (Shared Memory) erstellt. Der Filter transformiert die elementaren Benachrichtigungen zu zusammengesetzten Benachrichtigungen in Form von Objekten (Translation Filtering), welche in dem State Chart einen Zustandswechsel auslösen können. Es werden dabei der aktuelle State Chart Zustand und die vordefinierte Priorität der zusammengesetzten Benachrichtigung beachtet. Erst dann werden die Benachrichtigungsobjekte an den Event Handler des State Charts übergeben. Dieser ruft die dazugehörige Prozedur für die Benachrichtigungsverarbeitung auf.

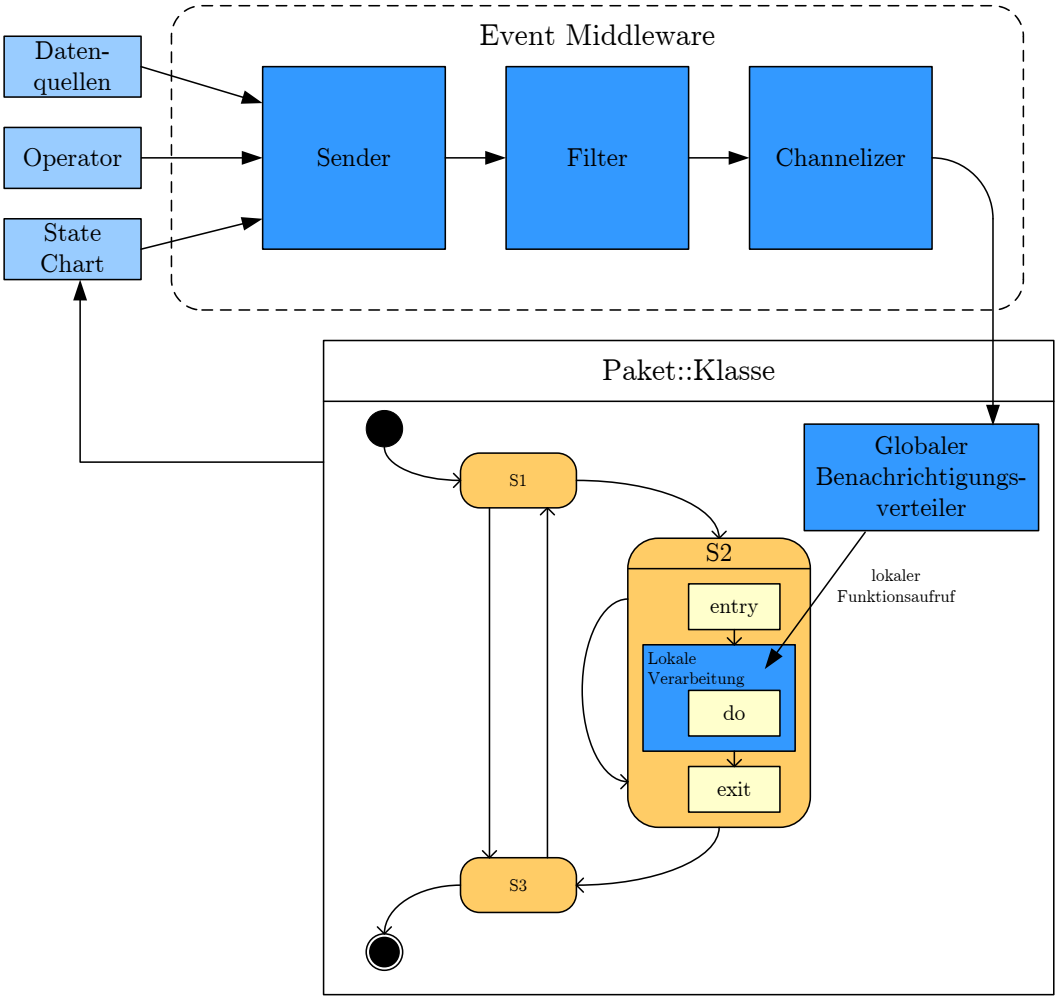

Bild 3.1: Schematische Darstellung des Modells

Das Modell aus Bild 3.1 wird mit dem Programm Rhapsody der Firma I-Logix umgesetzt. Rhapsody wurde dabei aus drei Gründen gewählt: Erstens ist dieses Werkzeug ein quasi Industrie-Standard, der unter anderem in Großunternehmen wie Airbus genutzt wird, und zweitens ermöglicht es, auf Basis des Modells, konstruktiv Code zu generieren. Ein weiterer Vorteil besteht darin, dass nun Modell und Code untereinander stets synchronisiert werden können, da mit Rhapsody auch Roundtrip-Verfahren für das Reverse-Engineering möglich sind. Ist das Re-Engineering des Templates abgeschlossen, müssen diese somit nicht mehr separat erarbeitet werden.

Die Umsetzung des Re-Engineerings erfolgt dabei nach dem erarbeiteten Schema aus Bild 3.2. Die strukturellen Informationen aus dem UML State Chart Template[2] werden mit Rhapsody grafisch modelliert und können jederzeit als Code-Rahmen generiert werden. Die in dem UML State Chart Template definierten Zustandsfunktionen werden auf die Entry-, Exit- und Do-Aktivitäten der State Chart Zustände verteilt. Des Weiteren wird die Ereignisdetektierung vollkommen in die bereits angesprochene Event Middleware verlagert und um einen Filter erweitert, welcher elementare Benachrichtigungen zu einem Benachrichtigungsobjekt zusammensetzt.

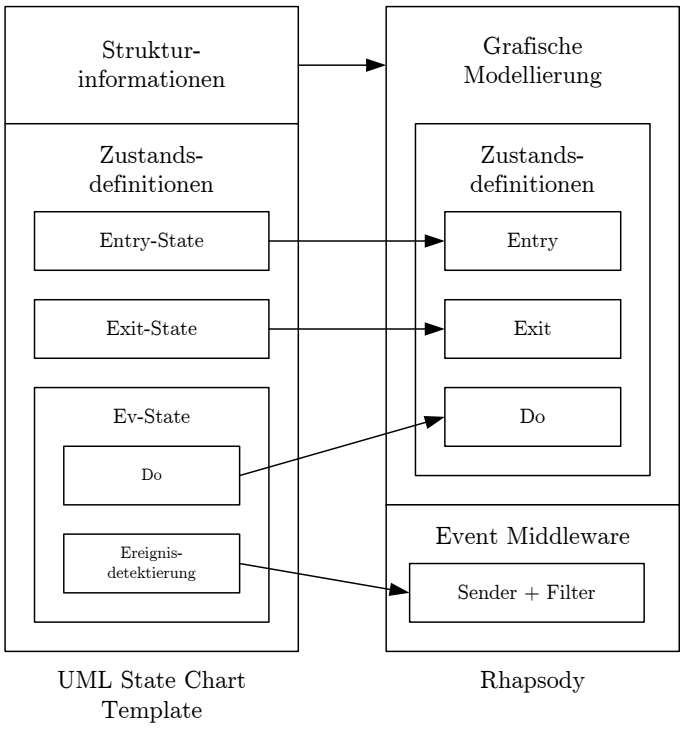

Bild 3.2: Schematisches Konzept des Re-Engineerings

[2]Eine ausführliche Beschreibung ist gedruckt in dem Anhang bzw. unter der ursprünglichen URL www.codeproject.com/samples/Statechart.asp zu finden.

3.2 Die Event Middleware

Die Event Middleware ist dazu konzipiert, die durch die Taktung diskretisierten kontinuierlichen Eingangsgrößen zu analysieren und anschließend Benachrichtigungsobjekte in Richtung des State Charts zu senden. Das Konzept der Ereignisanforderung ist in dieser Arbeit statisch umgesetzt, da dadurch Geschwindigkeitsvorteile im Vergleich zu einer dynamischen Realisierung erzielt werden. Darüber hinaus kann die Event Middleware auch auf mehrere State Charts, zum Beispiel für den „Supervisor“ aus Bild 2.4, erweitert werden.

3.2.1 Ereignisdetektierung und Transformation

Der Sender detektiert die notwendigen Ereignisse aus dem Eingabestrom für den Filter. Die interne Kommunikation zwischen Sender und Filter wird dabei über elementare Benachrichtigungen in Form gemeinsamer boolescher Variablen realisiert (Shared Memory), um die Ausführungszeit in der Event Middleware zu minimieren. Die Tabellen 3.1 und 3.2 zeigen die möglichen elementaren Benachrichtigungen, welche durch den Sender erstellt werden.

Elementare Benachrichtigung	**Bedeutung**
Flyable	Der Flugregler, die Sensorik sowie die Navigationslösung sind flugbereit.
ManualPilot	Der Sicherheitspilot übernimmt die Kontrolle.
AutoPilot	Der Sicherheitspilot übergibt die Kontrolle.
MissionModeOn	Die Bodenstation startet den Missionsmodus zur Abarbeitung einer Sequenz aus Basisverhalten.
MissionUpdated	Die Mission hat sich verändert.
MissionFinished	Alle Wegpunkte der Mission sind abgearbeitet.
StandStill	Der Hubschrauber befindet sich im Stillstand.
DriftXYorRot	Der Hubschrauber driftet bzw. rotiert.
CommandStop	Die Bodenstation fordert ein Anhalten bzw. ein Abbruch des aktuellen Verhaltens.
HeightReached	Der Hubschrauber ist auf die vorgegebene Höhe abgehoben.
HeightSafe	Der Abstand zwischen Hubschrauber und Boden ist ausreichend.

Tabelle 3.1: Elementare Benachrichtigungen des Senders (1)

Elementare Benachrichtigung	**Bedeutung**
GroundHeightAvailable	Das Sonar ist verfügbar.
HeadingReached	Das angestrebte Heading ist erreicht.
PositionReached	Die angestrebte Position ist erreicht.
PirouetteFinished	Die Pirouette ist abgeflogen.
Landed	Der Hubschrauber ist gelandet.
GCSlost	Die Datenlinks (WLAN, Funkmodem) zu der Bodenstation sind nicht verfügbar.
TimeOut	Eine vorher festgelegte Zeit ist abgelaufen.
DirectCommandValid	Das Direktkommando von der Bodenstation, dem Bildverarbeitungsrechner oder FLARM[3] ist gültig.
DirectCommandFinished	Das Direktkommando von der Bodenstation, dem Bildverarbeitungsrechner oder FLARM ist beendet.

Tabelle 3.2: Elementare Benachrichtigungen des Senders (2)

Der Filter transformiert diese elementaren Benachrichtigungen zu Benachrichtigungsobjekten (Translation Filtering) unter Berücksichtigung des aktuellen Zustandes und der vordefinierten Priorität. Die generierten Benachrichtigungsobjekte werden anschließend dem State Chart übergeben und aktivieren die Transitionen für einen Zustandswechsel. Durch die Teilung von Ereignisdetektierung und State Chart Schaltung ist die Funktionalität der Komponente auf zwei Klassen verteilt und damit einzeln testbar.

Die Tabelle 3.3 zeigt für die beiden ausgewählten Zustände MissionMode und HoverTo die dafür notwendigen Matrizen mit der Erweiterung von Verhaltenssequenzen (Behaviorsequence), Trajektorienplanung und direkten Kommandoeingaben (vdTarget). Im Anhang dieser Arbeit ist die Spezifikation für alle Zustände vollständig abgebildet. Die Benachrichtigungsobjekte sind in diesen Matrizen mit dem Präfix „ev“ gekennzeichnet, wobei sich die Prioritäten durch die Rangfolge in der Matrix bestimmen lassen. Von + ausgehend nimmt die Priorität der Benachrichtigungsobjekte in der Tabelle 3.3 ab. Wenn zum Beispiel der Sender die elementaren Benachrichtigungen ManualPilot und CommandStop für den Zustand MissionMode erstellt, generiert der Filter nur das Objekt evManualPilot, da dieses durch die gegebene Rangfolge eine höhere Priorität besitzt.

[3]FLARM ist eine Hardware-Lösung aus dem Segelflugbereich, welche frühzeitig vor gefährlicher Annäherung anderer mit FLARM ausgestatteter Flugzeuge warnt.

Das Objekt zur Taktung der Do-Aktivität (evPulse) wird erzeugt, wenn aus den elementaren Benachrichtigungen kein Objekt generiert werden kann. Ein Beispiel dafür ist die Kombination der Benachrichtigungen PositionReached und HeadingReached im Zustand HoverTo. Zur Generierung des Objekts evHoverToFinished fehlt die Benachrichtigung StandStill, daher wird in diesem Fall evPulse generiert.

MissionMode

	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evManualPilot			X																	
evDirectCommandValid															X					
evMissionModeStop				X																
EvDirectBehaviorCommand						X														
evPulse																				

HoverTo

	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evHoverToFinished								X			X	X								
evPulse																				

Tabelle 3.3: Ausgewählte Transformationsmatrizen

3.2.2 Algorithmen

Die Funktionsweise der Event Middleware wird nachfolgend erläutert. Bei jedem Aufruf über die Schnittstellenfunktion `CheckForEvents()`, welche in Listing 3.1 dargestellt ist, werden zunächst die aktuellen Eingangsgrößen (u. a. GPS-Position, Verhaltenssequenz der Bodenstation, aktiver Zustand im State Chart `ActiveState`) als lokale Kopie in die Event Middleware geladen (`GET`-Methodik). Diese Daten verwaltet der übergeordneten Missionsmanager aus Bild 2.4. Die Eingangsgrößen müssen vor der Beendigung der Event Middleware in den Missionsmanager zurück geschrieben werden (`SET`-Methodik), da eine Manipulation der Daten in der Event Middleware stattfindet. Es werden zum Beispiel Debug-Nachrichten hinzugefügt oder die Zielvorgaben für das Basisverhalten gesetzt.

Mit der Methode `ParentState()` wird geprüft, ob Benachrichtigungen für den Hauptzustand vorliegen, wenn der aktive Zustand ein zusammengesetzter Zustand ist. Anschließend findet erst eine Überprüfung für den eigentlichen aktiven Zustand mit der Methode `ActiveState()` statt. Die Verknüpfung von Haupt- und Teilzustand ist dabei statisch implementiert, um eine unnötige Kommunikation zwischen Event Middleware und State Chart zu unterbinden. Falls keine passende Benachrichtigung gefunden wird, wird die Do-Aktivität über das Objekt `evPulse` getaktet.

```
FUNCTION CheckForEvents()
{
    GET Data;

    SWITCH (ActiveState)
    {
        Flag = FALSE;
        IF (!Flag) THEN CALL Flag = ParentState();
        IF (!Flag) THEN CALL Flag = ActiveState();
    }

    IF (!Flag) THEN CALL CheckHandleEvent(evPulse);

    SET Data;
}
```

Listing 3.1: Schnittstellenfunktion der Event Middleware

Die Detektierung der Ereignisse wird mit einer separaten Klasse realisiert, welche aus den diskretisierten kontinuierlichen Eingabegrößen mit verschiedenen Methoden die elementaren Benachrichtigungen (`Element`) erzeugt. Die anschließende Transformation ist durch die vorgegebene Rangfolge aus den Matrizen priorisiert und wird bei dem ersten generierten Objekt (`Object`) abgebrochen. Zur Markierung wird die Variable `Flag` verwendet, welche signalisiert, ob bereits ein Objekt generiert wurde. Die verschiedenen Klassen für die instanziierten Objekte werden mit Rhapsody definiert und unterscheiden sich intern, durch eine eindeutige Identifizierung. Im Gegensatz zu dem UML Template treten keine Redundanzen auf, da eine klare Trennung zwischen `ParentState()` und `ActiveState()` erfolgt. Nachfolgend wird mit dem Listing 3.2 gezeigt, wie die Benachrichtigungen transformiert und weitergereicht werden.

```
FUNCTION ParentState()
{
    IF (!Flag AND Element1 AND Element2) THEN
    {
        CREATE Object1
        CALL CheckHandleEvent(Object1);
        RETURN TRUE;
    }
}
```

```
FUNCTION ActiveState()
{
    IF (!Flag AND (Element3 OR Element4)) THEN
    {
        CREATE Object2
        CALL CheckHandleEvent(Object2);
        RETURN TRUE;
    }
}
```

Listing 3.2: Trennung zwischen Haupt- und Teilzustand

Die Ereignisbenachrichtung für den State Chart wird in der Event Middleware mit der Methode `CheckHandleEvent()` aus Listing 3.3 überwacht. Treten in der Verarbeitung der Benachrichtigung Fehler auf, zum Beispiel wenn die übergebene Benachrichtigung nicht verarbeitet wird, wird dies in einem Objekt (`Output`) gespeichert und kann später ausgegeben werden.

```
FUNCTION CheckHandleEvent (Object)
{
    Result = StateChart.HandleEvent(Object);

    IF (Error == Result) Output.append(Result);
}
```

Listing 3.3: Benachrichtigungsüberwachung

3.2.3 Restriktionen

Bei der Umsetzung ergaben sich hinsichtlich einer schnellen Ausführungszeit die nachfolgenden Restriktionen: Die Taktung der Event Middleware ist konstant und wird in dem Flugsteuerungsrechner mit 100 Hz ausgeführt. Der State Chart darf die Methode `CheckForEvents()` der Event Middleware somit nicht separat, zum Beispiel über eine indirekte Verbindungen zu dem Missionsmanager bzw. über Makros aus dem Rhapsody OX-Framework, aufrufen.

Des Weiteren darf pro Takt lediglich eine Benachrichtigung generiert und in Richtung Ablaufsteuerung gesendet werden. Wird dies nicht eingehalten, können Sequenzketten als Folge einer Kettenreaktion auftreten, welche wichtige Berechnungen verzögern, so dass die für den Flugregler notwendigen Eingaben nicht rechtzeitig zusammengesetzt werden können.

3.3 Die Ablaufsteuerung

3.3.1 Modellierung

Das Rhapsody-Modell basiert auf der grundlegenden Struktur des UML State Chart Templates und wird, wie in dem Konzept (Bild 3.2) beschrieben, um zusammengesetzte Zustände erweitert. Die Transitionen zwischen den einzelnen Zuständen werden jedoch durch separate Benachrichtigungen aktiviert, welche, wie in Kapitel 3.2 erläutert (Tabelle 3.3), durch den Filter aus der Event Middleware erzeugt und dem State Chart übergeben werden. Um Kettenreaktionen zu vermeiden, wird bei jedem Aufruf der Event Middleware nur eine Benachrichtigung generiert. Dazu zählt auch die Benachrichtigung `evPulse`, welche die Do-Aktivität in jedem Zustand triggert. Findet durch eine Aktivierung einer Transition ein Zustandswechsel statt, wird dies protokolliert und kann bei Bedarf als Zustands- bzw. als Transitionsüberdeckung abgefragt werden.

Die oberste Hierarchieebene des State Charts ist in Bild 3.3 dargestellt und besteht aus sechs Zuständen und 18 Transitionen. Die zusammengesetzten Zustände MissionMode und CommandMode werden in dem Rhapsody-Modell durch ein entsprechendes Symbol in der rechten unteren Ecke dargestellt.

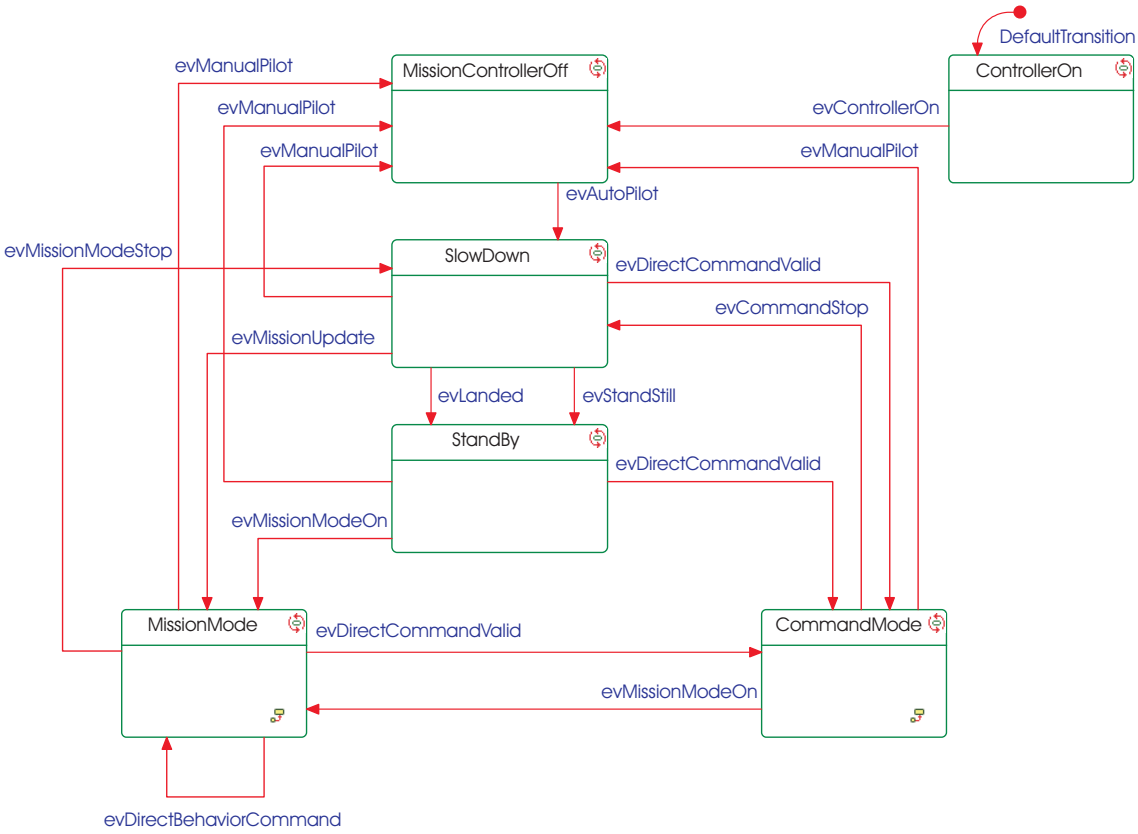

Bild 3.3: Oberste Hierarchieebene der Ablaufsteuerung

In dem State Chart ist es nicht notwendig, separate Fehlerzustände zu modellieren, da im Fehlerfall der externe Beobachter „Supervisor“ aus Bild 2.4 entsprechend reagiert und die Kontrolle übernimmt. Ein (optionalen) Endzustand ist unnötig, da der State Chart nur in Verbindung mit dem Hauptprogramm beendet wird.

Die Entry-Aktivität des Zustandes ControllerOn initialisiert den State Chart. Sobald der Hubschrauber flugbereit ist (Benachrichtigung `evControllerOn`), findet ein Zustandswechsel zu dem Zustand MissionControllerOff statt. In diesem Zustand übernimmt der Sicherheitspilot der Bodenstation die Steuerung, wobei die Ablaufsteuerung keinen Einfluss auf den nachgeschalteten Flugregler ausübt. Die Kontrolle wird von der Ablaufsteuerung erst übernommen, wenn der Sicherheitspilot den automatischen Flug aktiviert (Benachrichtigung `evAutoPilot`). Deaktiviert der Sicherheitspilot jedoch den automatischen Flug (Benachrichtigung `evManualPilot`), wird aus jedem anderen Zustand zu MissionControllerOff zurück gewechselt.

In dem Zustand SlowDown wird der Hubschrauber solange abgebremst, bis dieser keine Geschwindigkeit mehr in x-, y- und z-Richtung aufweist. Dies ist erfüllt, wenn ARTIS entweder gelandet ist oder in der Luft schwebt. Die dafür verwendeten Benachrichtigungen `evLanded` und `evStandStill` schalten den State Chart in den Zustand StandBy. Dort wird entweder das Schweben fortgesetzt, falls sich der Hubschrauber in der Luft befindet, oder es werden die regelungstechnischen Funktionen deaktiviert. Wenn bei einer Landung die Position von den Zielkoordinaten abweicht, verursacht ein nicht-deaktivierter PID-Regler ein Kippen des Hubschraubers, da der Soll-Wert des Integrators in dem Flugregler mit zunehmender Zeit weiter steigt.

Der eigentliche automatische Flug wird maßgeblich durch die beiden zusammengesetzten Zustände MissionMode und CommandMode bzw. durch deren Teilzustände bestimmt. In beiden State Charts wird kein expliziter Endzustand verwendet, da die Vorteile des internen Ablaufs eines zusammengesetzten Zustandes (Bild 2.17) ausgenutzt werden. Der Hauptzustand kann aus jedem Teilzustand verlassen werden, da diese die Benachrichtigungen für ihren „Vater“ ebenfalls triggern und somit zunächst die entsprechende Exit-Aktivität des Teilzustandes auslösen und anschließend die des Hauptzustandes. Damit werden Redundanzen aus dem UML State Chart Template vermieden und das System gestaltet sich übersichtlicher.

Im Kommandomodus werden Direktkommandos, die beispielsweise von dem Bodenstationsjoystick gesendet werden, verarbeitet. Der Missionsmodus dient dagegen dazu, eine definierte Verhaltenssequenz, wie sie in Bild 2.5 am Beispiel einer Mosaiking-Mission vorgestellt wurde, zu absolvieren. Das Bild 3.4 zeigt den dazugehörigen State Chart mit zehn Zuständen und 18 Transitionen.

Der Zustand GetCommand hat den Zweck, der Event Middleware bei Aktivierung des Missionsmodus den Vorteil zu bieten, das nächste Verhalten aus der Verhaltenssequenz im folgendem Takt auszulesen und damit eine entsprechende Benachrichtigung zu senden. Wenn beispielsweise das Verhalten zum Landen ausgeführt werden soll, muss die Benachrichtigung `evCmdLand` von der Event Middleware erzeugt werden. Ist das Verhalten in dem Folgezustand abgearbeitet, wenn in diesem Beispiel der Hubschrauber gelandet ist (Benachrichtigung `evLandingFinished`), wird wieder zu dem Zustand GetCommand gewechselt, um das nächste Verhalten durch die Event Middleware einlesen zu lassen.

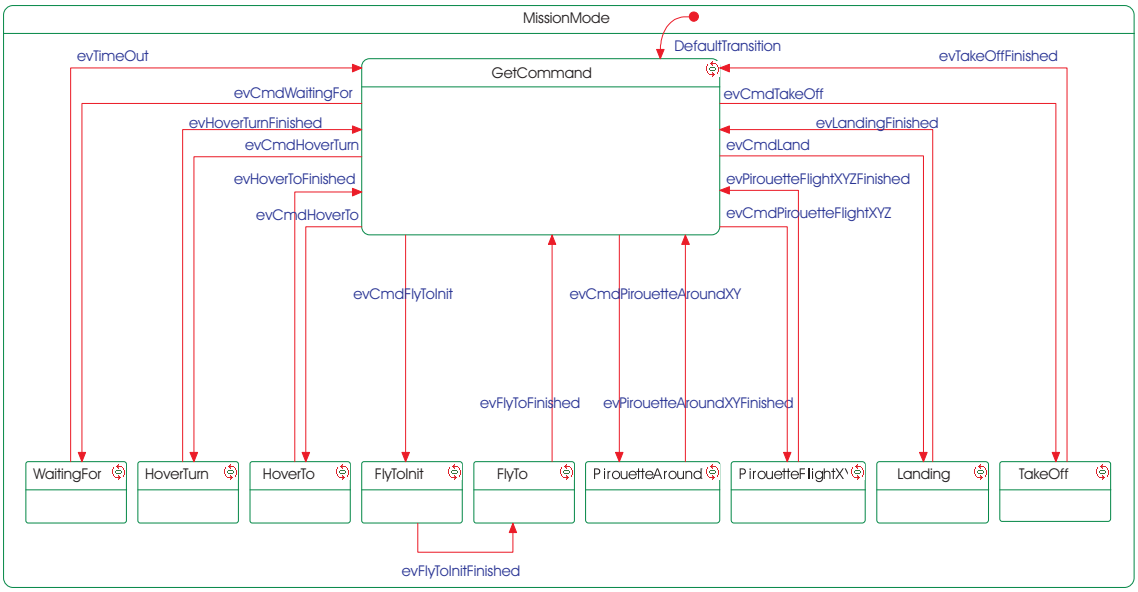

Bild 3.4: Zusammengesetzter Zustand MissionMode

Der Hubschrauber soll in dem Missionsmodus aus Bild 3.4 bei dem Basisverhalten:

- `WaitingFor`
  an der aktuellen Position für eine vorgegebene Zeit verweilen.

- `HoverTurn`
  an der aktuellen Position, um den angegebene Winkel, mit der vorgegebenen Winkelgeschwindigkeit drehen.

- `HoverTo`
  an die angegebene Position schweben und sich dabei um den angegebene Winkel drehen.

- `FlyToInit`
  aus mindestens drei Punkten eine Flugbahn (Trajektorie) berechnen.

- `FlyTo`
  die berechnete Trajektorie mit der vorgegebenen Geschwindigkeit abfliegen.

- `PirouetteAroundXY`
  um den angegebenen Punkt, bis zu einem vorgegebenen Winkel, drehen. Die Ausrichtung des Hubschraubers zu dem Punkt soll während der Pirouette beibehalten werden.

- `PirouetteFlightXYZ`
  zu dem angegeben Punkt fliegen und sich dabei wie vorgegeben drehen.

- `Landing`
  an der aktuellen Position langsam zu Boden sinken und dort landen.

- `TakeOff`
  vom Boden abheben, bis die vorgegebene Sicherheitshöhe für den Hubschrauber erreicht ist.

Nachfolgend soll an einem Beispiel erläutert werden, wie der zusammengesetzte Zustand verlassen werden kann. Wenn beispielsweise der Sicherheitspilot wieder die Kontrolle übernimmt (Benachrichtigung `evManualPilot`) und der aktive Teilzustand WaitingFor ist, wird zunächst dessen Exit-Aktivität ausgeführt und anschließend die des Hauptzustandes MissionMode. Danach findet der Zustandswechsel zu dem Zustand MissionControllerOff statt. Der interne Ablauf für einen zusammengesetzten Zustand ist in dem Kapitel 2.2 durch Bild 2.17 beschrieben.

In dem UML State Chart Template ist der Zustand CommandMode nicht hierarchisch implementiert. In dem Rhapsody-Modell wird jedoch analog zu dem Missionsmodus ein zusammengesetzter Zustand (Bild 3.5) verwendet, da vier verschiedene Direktkommandos (Positions- bzw. Geschwindigkeitskommandos verschiedener Herkunft) möglich sind. Auch für diesen State Chart erfolgt die Schaltung über Benachrichtigungen, welche von der Event Middleware erzeugt werden. Ein expliziter Endzustand ist durch den internen Ablauf des State Charts nicht notwendig.

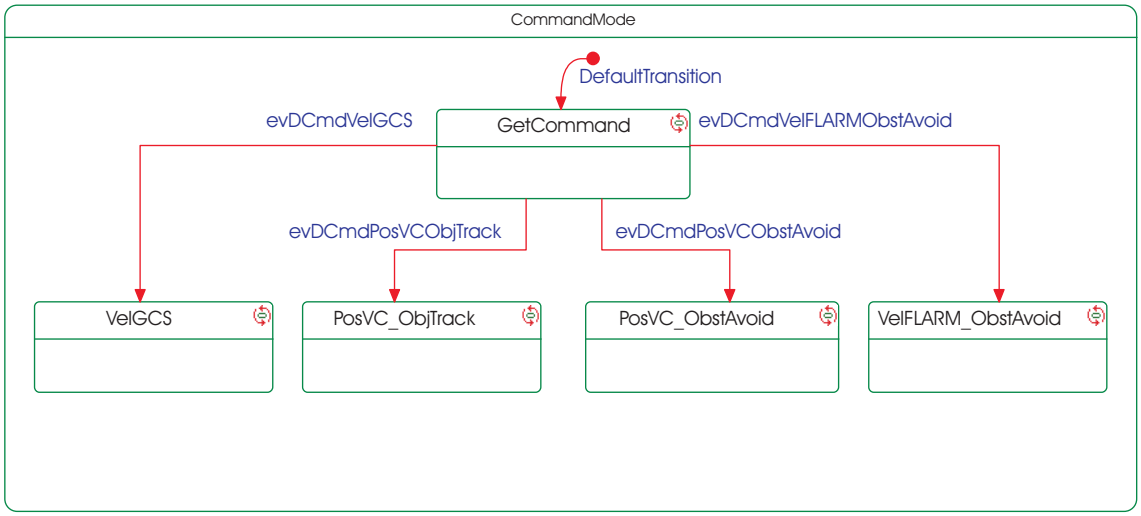

Bild 3.5: Zusammengesetzter Zustand CommandMode

Die Direktkommandos werden mit UDP-Paketen[4] übertragen und im Zustand GetDirectCommand eingelesen. Die Event Middleware generiert je nach Paket-ID eine Benachrichtigung, welche in den entsprechenden Teilzustand schaltet, um das Direktkommando auszuführen. In diesen Teilzuständen wird zunächst in der Do-Aktivität überprüft, ob sich der Hubschrauber in sicherer Entfernung zum Boden befindet. Trifft dies zu, werden die Direktkommandos wie nachfolgend beschrieben verarbeitet, andernfalls werden sie verworfen.

- `VelGCS`
  In diesem Teilzustand werden die von der Bodenstation übermittelten Geschwindigkeiten für die x-, y- und z-Richtung direkt an den Flugregler weitergeleitet.

- `PosVC-ObjTrack`
  Die von dem Bildverarbeitungsrechner übertragenen Geschwindigkeiten zur Verfolgung eines (beweglichen) Objektes, werden in eine Trajektorie transformiert und dem Flugregler übergeben.

- `PosVC-ObstAvoid`
  Die von dem Bildverarbeitungsrechner übertragenen Geschwindigkeiten zum Ausweichen eines (beweglichen) Hindernisses, werden in eine Trajektorie transformiert und dem Flugregler zur Weiterverarbeitung übermittelt.

- `VelFLARM-ObstAvoid`
  Die Do-Aktivität dieses Teilzustandes überträgt die von FLARM übergebenen Geschwindigkeiten zur Kollisionsvermeidung für die x-, y- und z-Richtung direkt an den Flugregler.

Beendet wird das Direktkommando, wenn es vollständig ausgeführt ist, abgebrochen werden soll oder eine bestimmte Zeitspanne für das erwartete direkte Folgekommando abgelaufen ist.

Abschließend soll ein letzter Unterschied zu der UML State Chart Template-Implementierung erläutert werden. Das Rhapsody-Modell ist so strukturiert, dass die Transitionen ausschließlich durch Benachrichtigungsobjekte aus der Event Middleware schalten. Optionale Wächterbedingungen finden in dem Rhapsody-Modell keine Berücksichtigung. Die einzigen Aktivitäten, welche mit den Transitionen verbunden sind und zwischen der exit-Aktion des einen und der entry-Aktion des folgenden Zustandes ausgeführt wird, dienen zur protokollieren der Zustandswechsel und werden in dem nachfolgenden Kapitel beschrieben.

[4]Das User Datagramm Protocol (UDP) ist ein verbindungsloses Netzwerkprotokol.

3.3.2 Zustands- und Transitionsüberdeckung

Wie bereits erläutert, werden alle Zustände und deren Übergänge bei Aktivierung geloggt und können bei Bedarf als Zustands- bzw. Transitionsüberdeckung in Form eines Reports abgefragt werden. Dabei findet das C++ Template Map aus der STL (Standard Template Library) in der Form

```
Map[String:Bezeichner] = Int:Wert
```

Verwendung. Das heißt, dass jeder Zustand und jede Transition einen eigenen Eintrag über ihren Namen als Bezeichner besitzen und bei Aktivierung der Wert inkrementiert wird. Der Report wird für die Zustands- bzw. die Transitionsüberdeckung je nach Benutzerwunsch ausführlich oder gekürzt erstellt. Der ausführliche Zustandsreport gibt für jeden Zustand somit an, wie oft dieser betreten wurde. Es wird zudem ein für den gesamten State Chart übersichtlicher Wert (StateCoverage in [%]) ermittelt. Nachfolgend ist ein solcher Report beispielhaft abgebildet.

```
SC: State-Entry (1) in CommandMode
SC: State-Entry (1) in ControllerOn
SC: State-Entry (0) in FlyTo
SC: State-Entry (0) in FlyToInit
SC: State-Entry (2) in GetCommand
SC: State-Entry (1) in GetDirectCommand
SC: State-Entry (0) in HoverTo
SC: State-Entry (0) in HoverTurn
SC: State-Entry (0) in Landing
SC: State-Entry (1) in MissionControllerOff
SC: State-Entry (1) in MissionMode
SC: State-Entry (0) in PirouetteAroundXY
SC: State-Entry (0) in PirouetteFlightXYZ
SC: State-Entry (0) in PosVC_ObjTrack
SC: State-Entry (0) in PosVC_ObstAvoid
SC: State-Entry (2) in SlowDown
SC: State-Entry (2) in StandBy
SC: State-Entry (1) in TakeOff
SC: State-Entry (0) in VelFLARM_ObstAvoid
SC: State-Entry (0) in VelGCS
SC: State-Entry (0) in WaitingFor
SC: StatesCoverage: 42.8571
```

Der gekürzte Report beinhaltet nur den übersichtlichen Wert, wobei dieser auch nur bei einer Veränderung ausgegeben wird. Nachfolgend ist beispielhaft dargestellt, wie sich dieser Wert für jeden Zustandswechsel in dem State Chart ändert.

```
1:
SC: MissionControllerOff: Entry (on ground)
SC: StatesCoverage: 9.5238
2:
SC: MissionControllerOff: Exit.
SC: SlowDown: Entry  (on ground)
SC: StatesCoverage: 14.2837
3:
SC: SlowDown: Exit.
SC: StandBy: Entry (on ground [...]).
SC: StatesCoverage: 19.0476
4:
SC: StandBy: Exit.
SC: MissionMode: Entry / On.
SC: GetCommand: Entry.
SC: StatesCoverage: 28.5714
```

Wie bereits beschrieben, kann auch der Report über die Transitionsüberdeckung wahlweise unterschiedlich ausgegeben werden. Der ausführliche Report gibt für jede Transition an, wie oft diese durchlaufen wurde. Zusätzlich wird, wie auch bei der Zustandsüberdeckung, ein übersichtlicher Wert (TransitionCoverage in [%]) ermittelt. Der gekürzte Report gibt diesen Wert ebenfalls nur bei einer Veränderung aus.

```
1:
SC: MissionControllerOff: Entry (on ground)
SC: TransitionCoverage: 5.0000 %
2:
SC: MissionControllerOff: Exit.
SC: SlowDown: Entry  (on ground)
SC: TransitionCoverage: 7.5000 %
3:
SC: SlowDown: Exit.
SC: StandBy: Entry (on ground [...).
SC: TransitionCoverage: 10.0000 %
```

Ein Beispiel für den ausführlichen Report über die Transitionsüberdeckung ist in dem Anhang abgebildet.

3.3.3 Komplexität

Eine Klassifikation von Strukturmetriken kann nach [29] wie folgt aussehen:

- Eigenschaften des Kontrollflussgraphen (Knoten, Kanten),
- Eigenschaften des Aufrufgraphen (Größe, Tiefe, Breite),
- Modulkohäsion, Modulkopplung (Abhängigkeiten),
- Objektorientierte Metriken und
- Daten bzw. Datenstrukturen.

Für die Ablaufsteuerung sollen an dieser Stelle die Eigenschaften des Kontrollflussgraphen betrachtet werden, da diese in Zusammenhang mit dem Konzept der State Charts einen intuitiven Zugang bilden. Thomas J. McCabe stellte 1976 ein, auf der Graphentheorie basierendes, strukturelles Software-Maß vor [10]. Mit Hilfe dieser so genannten zyklomatische Zahl (McCabe-Metrik) wird auf Basis eines Kontrollflussgraphen G dessen zyklomatische Komplexität $V(G)$ ermittelt. Mit der Gleichung 3.1 und den Parametern e (Anzahl der Kanten), n (Anzahl der Knoten) und p (Anzahl der unverbundenen Teile des Graphen) kann dieser Wert berechnet werden.

$$V(G) = e - n + 2 \cdot p \tag{3.1}$$

$$V(G) = \pi + 1 \tag{3.2}$$

Wird die zyklomatische Komplexität für eine zusammenhängende Komponente ermittelt, kann diese durch Gleichung 3.2 vereinfacht werden, wobei π die Anzahl der Bedingungen (Binärverzweigungen) ist. Die Aussagekraft seiner Metrik begründet McCabe damit, dass die Test- und Wartbarkeit eines Programms von der Anzahl seiner Ablaufpfade abhängt. In diesem Zusammenhang wird in der Studie [4] empirisch gezeigt, dass die Anzahl von Programmfehlern abhängig von der Anzahl der Programmzeilen sowie der zyklomatischen Komplexität des Programmes ist. Es wurde bei dieser Betrachtung festgestellt, dass mit steigender zyklomatischer Komplexität die Anzahl der Programmfehler zunahm. Nach [31] bzw. [10] ist es möglich, ein Programm auf Basis der McCabe-Metrik empirisch zu klassifizieren (Tabelle 3.4).

V(G)	**Risiko**
1-10	Einfacher Graph, geringes Risiko
11-20	komplexerer Graph, erträgliches Risiko
21-50	komplexer Graph, hohes Risiko
> 50	untestbarer Graph, extrem hohes Risiko

Tabelle 3.4: Empirische Klassifikation durch die McCabe-Metrik

Die McCabe-Metrik wird auch bei Programmkonstrukten angewandt, da Verzweigungen innerhalb des Kontrollflussgraphen den Entscheidungen im Programm (wie if-Abfragen oder while-Schleifen) entsprechen. Nachfolgend soll die strukturelle Komplexität für ein Programmbeispiel (Bild 3.6) berechnet werden. In diesem Fall liegt eine zusammenhängende Komponente mit zwei Verzweigungen bzw. sechs Knoten und sieben Transitionen vor. Es ergibt sich somit $V(G) = 3$.

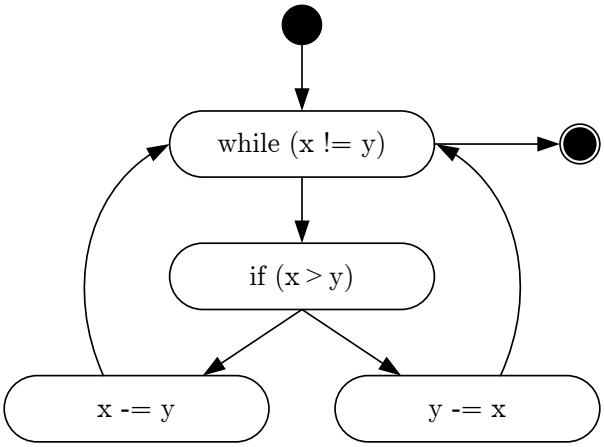

Bild 3.6: Beispiel struktureller Komplexität

Neben dem Vorteil, dass die McCabe-Metrik intuitiv zu berechnen ist, und durch die empirische Klassifikation einen Anhaltspunkt für eine Umstrukturierung bietet, kann sie auch für die Bestimmung der benötigten Anzahl von Testfällen herangezogen werden. Die zyklomatische Komplexität stellt die Obergrenze für die bei einem Zweigüberdeckungstest benötigten Testfälle dar [11], da diese die maximale Anzahl linear unabhängiger Wege durch den Kontrollflussgraphen von Initialknoten ausgehend angibt. Dies soll an den vier Beispielen aus Bild 3.7 erläutert werden.

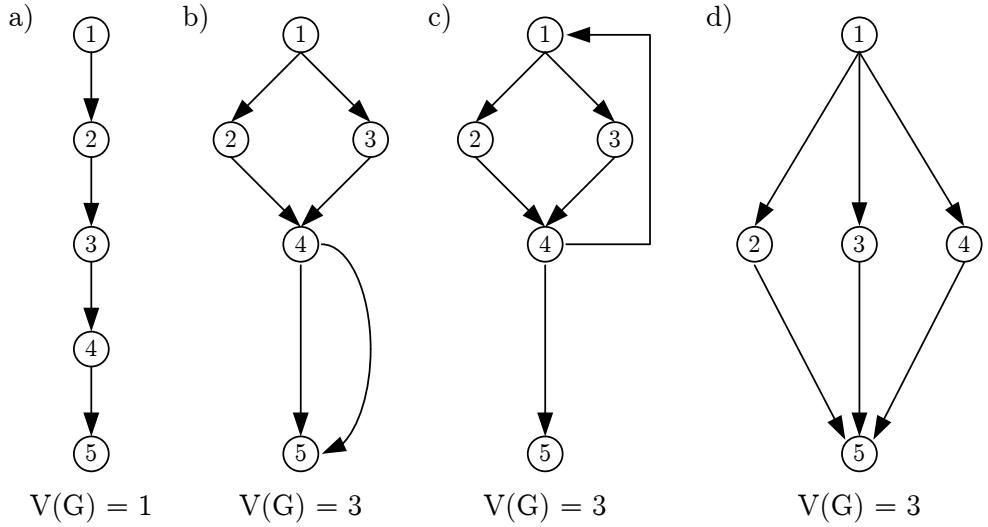

Bild 3.7: Beispiele für die Testanzahl auf Basis der McCabe-Metrik

In dem Beispiel a) wird nur ein Testfall $V(G) = 1$ benötigt, da der Ablauf zwischen den einzelnen Knoten sequentiell ist und keine Verzweigung aufweist. Die Beispiele b) und c) zeigen, dass $V(G)$ der maximalen Anzahl von Testfällen entspricht, da mit Modellwissen das Testen optimiert werden kann. Für den Fall b) sind zwei Testfällen ausreichend und in dem Fall c) wird sogar nur ein Testfall:

$$1 \rightarrow 2 \rightarrow 4 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 5$$

benötigt. Das Beispiel d) besitzt die gleiche Anzahl von Knoten und Kanten wie die Beispiele b) und c), wobei in diesem Fall tatsächlich drei Testfälle erstellt werden müssen.

Neben den genannten Vorteilen weist die McCabe-Metrik allerdings auch zwei Nachteile auf. Zum Einen wird nur der Kontrollfluss berücksichtigt und dabei der Datenfluss vernachlässigt und zum Anderen wird die Komplexität der Anweisungen nicht beachtet. Diese Nachteile werden jedoch nicht weiter berücksichtigt, da sie in der modellierten Ablaufsteuerung selten zutreffen.

Die McCabe-Metrik ist für das UML State Chart Template als auch für das in Rhapsody erstellte Modell berechnet. Durch das Re-Engineering ist die Komplexität der Ablaufsteuerung von $V(G) = 64$ um 22 Einheiten auf $V(G) = 39$ gefallen. Allerdings wird die Ablaufsteuerung immer noch als komplex mit hohem Risiko bewertet, jedoch nicht mehr als untestbar mit extrem hohen Risiko. Ein weitere Teilung von Missions- und Commandomodus in separate Klassen ist ein möglicher Schritt, die Komplexität zu reduzieren.

	Knoten	**Transition**	**Komponenten**	**V(G)**
Template	17	62+17	1	64
Rhapsody	24	40+21	1	39

Tabelle 3.5: Zyklomatische Komplexität des Modells

Wird in dem State Chart der Ablaufsteuerung in Endzustand eingesetzt, kann die McCabe-Metrik, wie beschrieben, auch als Kriterium für die maximale Anzahl von Testfällen für einen Strukturtest verwendet werden.

3.4 Rhapsody OX-Framework

Das Rhapsody OX-Framework (Object Execution Framework) bietet die in Bild 3.8 dargestellten Grundfunktionen an, dazu gehören State Chart, TimeOut, Thread und Event Management. In diesem Kapitel sollen die wichtigsten Funktionen aus diesen Bereichen kurz erläutert werden. Eine ausführliche Dokumentation wird mit den Anleitungen von Rhapsody [8] und OXF [7] gegeben.

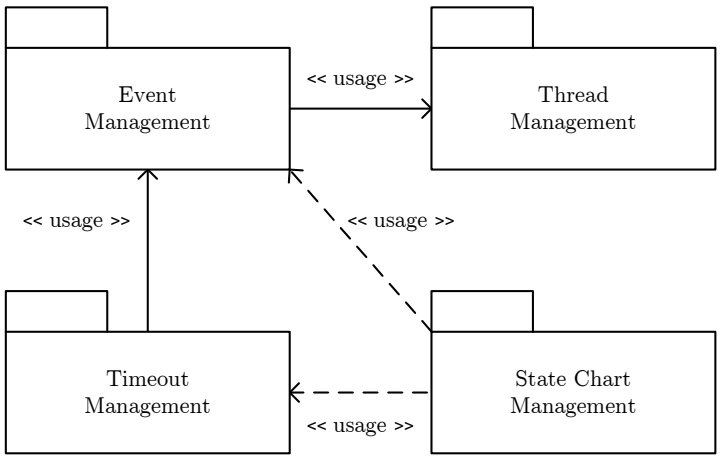

Bild 3.8: Grundfunktionen von OXF

3.4.1 State Chart Management

Das State Chart Management wird durch die Methoden der Superklasse `OMReactive` realisiert, von welcher alle State Chart Klassen abgeleitet sind. Dazu gehören Methoden zur Initialisierung `initStatechart()`, zur Ausführung `startBehavior()` und zur Terminierung `endBehavior()`. Über die Methode `setSupportRestartBehavior()` kann in Verbindung mit der Ausführungsmethode ein Restart realisiert werden, bei welchem der State Chart wieder in den Ausgangszustand versetzt wird.

Die Methoden `isActive()` bzw. `isBusy()` ermöglichen es anderen Klassen, abzufragen, ob der State Chart entweder initialisiert und aktiviert ist bzw. ob dieser zu dem Abfragezeitpunkt gerade eine Benachrichtigung verarbeitet. Des Weiteren ist es möglich, Endlosschleifen über die Methode `setMaxNullSteps()` automatisch detektieren zu lassen. Es wird dabei festgelegt, wie viele Transitionen ohne Benachrichtigungen bzw. Wächterbedingungen geschaltet werden dürfen. Die Zählung erfolgt stets automatisch in dem State Chart, wobei bei dem voreingestellten Wert von zwei abgebrochen und das Programm beendet wird. Mit der genannten Methode wird dieser Wert entweder verändert bzw. die Zählung mit dem Wert null deaktiviert.

Abschließend sollen zwei weitere Funktionen des State Chart Managements genannt werden. Das Makro `gen()` ermöglicht das Senden einer beliebigen Benachrichtigung zum Schalten einer Transition und mit der Methode `State-IN()` kann für flache State Charts überprüft werden, welcher Zustand aktiv ist. Die wiederverwendbaren State Charts nutzen dafür die Methode `in()` der Klasse `OMState`. Diese Funktionen werden für einfache Tests bzw. in der Event Middleware genutzt.

3.4.2 Event Management

Der Begriff Event wird in dem OX-Framework, im Gegensatz zu dem Begriff aus Kapitel 2.2, für die eigentliche Benachrichtigung als Objekt verwendet. Der Sender instanziiert das gewünschte Objekt einer Klasse, welche von der Superklasse `OMEvent` abgeleitet ist. Anschließend wird das Objekt direkt dem Handler mit der Methode `handleEvent()` übergeben und verarbeitet.

Die indirekte Möglichkeit besteht darin, das instanziierte Objekt des Senders, über die Methode `sendEvent()`, dem Dispatcher zu übergeben. Dieser nimmt die Objekte in eine Liste auf und benachrichtigt anschließend den Handler, welcher die Objekte schrittweise verarbeitet. Diese Liste kann separat mit der Methode `cancelEvents()` geleert werden. Die indirekte Benachrichtigung wird in der Ablaufsteuerung nicht verwendet, da die Taktung durch die Event Middleware direkt erfolgen soll und der Zwischenschritt des Dispatchers unerwünscht ist.

Die Methode `setDeleteAfterConsume()` des instanziierten Objektes aktiviert das Löschen nach dessen Verarbeitung. Wird das Objekt allerdings nicht verarbeitet, wenn beispielsweise kein Trigger für das Objekt spezifiziert ist, wird dies über den Rückgabewert `TakeEventStatus` mitgeteilt. Die Verarbeitung kann anschließend mit der Methode `handleEventNotConsumed()` erneut gestartet werden, wobei in der Ablaufsteuerung drauf verzichtet wird, da dieses Benachrichtigungsobjekt durch den Folgetakt seine Gültigkeit verliert.

Benachrichtigungsobjekte der Klasse `OMEvent` bieten noch weitere Funktionen an: Zum Einen besteht die Möglichkeit, jede Klasse mit einer eigenen Identifizierungsnummer (ID) zu versehen. Bei der Code-Generierung erstellt Rhapsody automatisch eine separate ID für jede Klasse, allerdings kann dies bei Bedarf auch manuell mit den Methoden `getId()` und `setId()` wiederholt werden. Zum Anderen können über die Methoden `getDueTime()` und `getDelayTime()` die Verweil- bzw. die Verarbeitungszeit ermittelt werden, um beispielsweise eine Aussage über die Rechenzeit in Abhängigkeit von Zustand und Benachrichtigung zu ermitteln.

3.4.3 Timeout Management

Mit dem Timeout Management ist es möglich, zeitabhängige Benachrichtigungen der Klasse `OMTimeout`, die ebenfalls von der Superklasse `OMEvent` abgeleitet sind, zu erzeugen und zu verwalten. Es wird dazu eine Zeitspanne definiert, die bei Ablauf ein Objekt der Klasse `OMTimeout` instanziiert, welches direkt von dem Handler mit der Funktion `handleEvent()` oder aber über die Liste des Dispatchers mit der Funktion `sendEvent()` verarbeitet wird. Wenn beispielsweise eine Transition das Makro `tm()` als Trigger verwendet, wird bei der Aktivierung des Zustands solange gewartet, bis die angegebene Zeitspanne abgelaufen ist, erst dann schaltet die Transition.

Das Timeout Management wird durch einen Scheduler umgesetzt, der mehrere Timeouts parallel verwalten kann. Der Scheduler wird über die Funktion `scheduleTimeout()` aufgerufen. Sollen zur Laufzeit im Modell definierte Timeouts abgeschaltet werden, können entweder, mit der Funktion `cancelTimeout()`, ein spezifischer Timeout bzw. über `cancelTimeouts()` alle Timeouts gelöscht werden.

Alle Aktivitäten der Ablaufsteuerung werden durch die Event Middleware mit 100 Hz getaktet, daher ist es nicht notwendig separate Scheduler einzusetzen, welche zusätzliche Rechenzeit benötigen. Das Timeout Management wird somit nicht verwendet, sollte aber als Grundfunktion des OX-Frameworks kurz vorgestellt werden.

3.4.4 Thread Management

Das Thread Management dient zur Verwaltung aller State Charts in verschiedenen Threads. Dazu wird die Klasse `OMThreadManager` verwendet, mit welcher die einzelnen Threads mit den Methoden `registerThread()` aufgenommen bzw. mit `deregisterThread()` entnommen werden. Zusätzlich können die Threads mit den Methoden `stopAllThreads()` angehalten, mit `wakeup()` einzeln wieder aktiviert und mit `cleanupAllThreads()` gelöscht werden.

Das Thread Management wird in der Ablaufsteuerung ebenfalls nicht verwendet, da bisher auf eine parallele Modellierung in dem State Chart verzichtet wird. Sollte der Missionsmanager allerdings vollständig in Rhapsody modelliert werden, bietet es sich an, ausgewählte Funktionen (zum Beispiel die Aufgabenplanung) durch verschiedenen Threads zu realisieren.

3.5 Die Integration

3.5.1 Strukturierung der ARTIS-Systemintegration

Mit diesem Kapitel wird versucht, den agilen Entwicklungsprozess der ARTIS-Systementwicklung in Bezug auf die Systemintegration zu strukturieren, da diese auch für das Rhapsody-Modell durchgeführt werden muss. Problematisch bei der ARTIS-Systementwicklung bzw. bei UAV-Forschungsprojekten allgemein ist, dass die verschiedenen Systemkomponenten parallel entwickelt werden und durch ständige Innovationen einen unterschiedlichen Entwicklungsgrad besitzen. Vor allem vor Flugversuchen müssen die einzelnen Systemkomponenten jedoch integriert werden, um eine funktionierende und zugleich stabile Flugsoftware zu erhalten. Aus diesem Grund werden die Integrationstage, deren Ablauf in Bild 3.9 dargestellt ist, eingeführt.

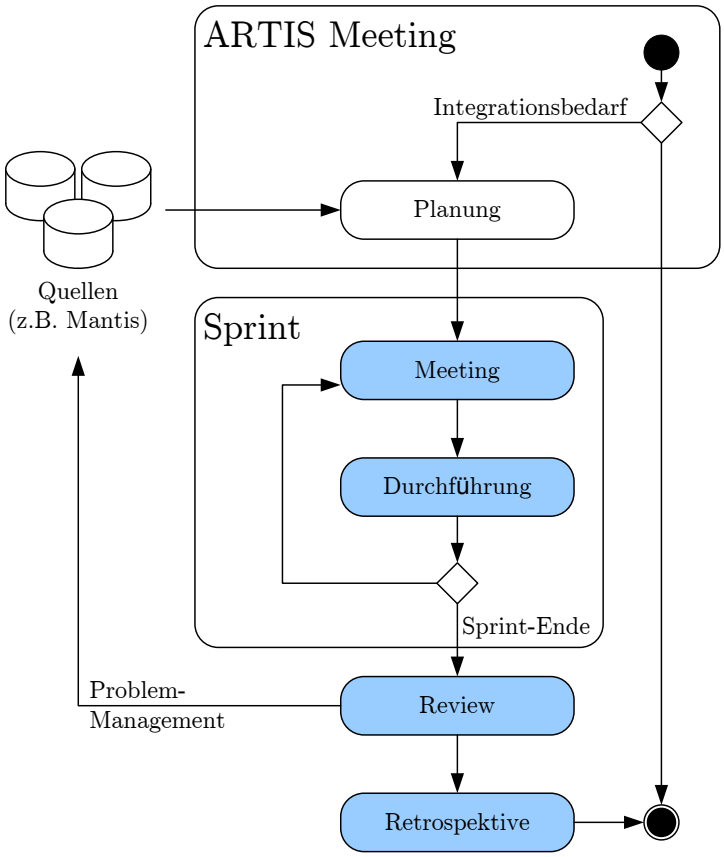

Bild 3.9: Strukturierter Ablauf der Integrationstage

Ein Ziel der Integrationstage ist es, die Integration geplant und damit strukturiert durchzuführen, dazu gehören qualitative und fehlerfreie Integration der einzelnen Systemkomponenten, kurze Dokumentationen der Vorgänge und Problemanalysen der durchgeführten Systementwicklung.

Der dafür notwendige Ablauf mit den fünf Aktivitäten (Planung, Meeting, Durchführung, Review und Retrospektive) basiert auf dem Scrum-Vorgehensmodell, welches vorwiegend zur Strukturierung der Implementierungsphase eingesetzt wird. Die Veröffentlichungen [9], [19] und [25] beschreiben den Einsatz von Scrum auch unter der Verwendung von Extreme Programming (XP) für kleine Projektteams.

Wenn bei dem wöchentlich stattfindenden ARTIS-Meeting der Bedarf zur Integration der einzelnen Systemkomponenten aufgezeigt wird, erfolgt die Integrationsplanung. Es werden dabei die einzelnen Aufgaben und deren Abhängigkeiten notiert, bewertet und nach dem Eisenhower-Prinzip (Bild 3.10), welche eine von vielen vorgestellten Methodiken aus [18] ist, priorisiert. Als Quellen können zum Beispiel das Mantis Bugtracking-System oder das CVS-Repository genutzt werden, da diese Informationen über behobene Fehler oder die Änderungshäufigkeit des Quellcodes beinhalten. Der Aufwand jedes einzelnen Arbeitspakets sollte dabei nicht größer sein als acht Stunden, größere Arbeitspakete werden dafür in kleinere Teilpakete zerlegt. Anschließend erfolgt eine grobe Zeitplanung für den Integrationszeitraum, in Bild 3.9 wird dieser als Sprint bezeichnet und sollte drei bis vier Tage dauern.

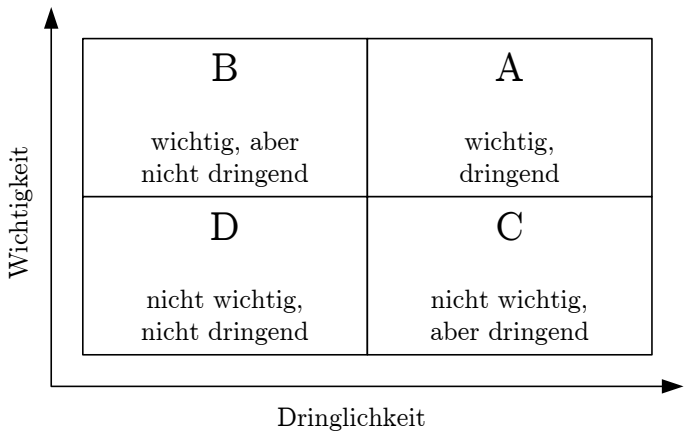

Bild 3.10: Das Eisenhower-Prinzip

Während des Sprints findet täglich ein Meeting statt, in welchem die geplanten Aufgaben eindeutig den jeweiligen Teilnehmer zugeordnet werden. Durch diese Zuordnung soll eine Verpflichtung zur Bearbeitung entstehen, da nicht verteilte Aufgaben schnell ignoriert werden. Weiterhin berichtet jeder Teilnehmer in dem Meeting über den bisherigen Fortschritt des jeweiligen Arbeitspakets und macht Zusagen über die nächsten Arbeitsschritte. Aufgetretene Probleme werden benannt und in einer kleineren Gruppe diskutiert.

Die einzelnen Aufgaben werden in kleinen Gruppen möglichst parallel bearbeitet. Ein Ziel dabei ist es, die Effektivität zu steigern, indem die Integration in dem ARTIS-Labor durchgeführt wird und nicht wie bisher auf mehrere Räume verteilt. Weiterhin stärkt die räumliche Nähe die Kommunikation untereinander und erhöht damit die Produktivität.

Paarweises Arbeiten ist dabei nach dem Extreme Programming-Vorbild, wie es in in [30] und [16] vorgestellt wird, angedacht und wird wenn möglich durchgeführt.

Nach dem Sprint erfolgt das Review, dazu werden informell die Ergebnisse vorgeführt bzw. die vorgenommenen Änderungen präsentiert. Änderungswünsche oder offene Probleme werden z.B. in dem Mantis-System (Problem Management) festgehalten. Abschließend findet die Retrospektive statt, in der die Teilnehmer den durchgeführten Prozess, jedoch nicht die Änderungen der Soft- bzw. Hardware, bewerten. Eine kurze Diskussion über die Vor- und Nachteile der Durchführung ermöglicht eine Anpassung der nächsten Integration durch neu gewonnene Erkenntnisse. Technische, personelle oder organisatorische Veränderungen im Projekt können somit aufgenommen und verarbeitet werden.

Die Steuerung und die Kontrolle über die Durchführung der Integrationstage obliegt dem Integrations- bzw. dem Projektleiter, welcher den jeweiligen Fortschritt an den einzelnen Tagen in einem einfachen Protokoll zur Überprüfung vermerkt. Es werden die folgenden Größen für eine Gesamtbetrachtung als sinnvoll erachtet:

- die Anzahl der geplanten Arbeitspakete und -stunden zu Beginn,
- die Anzahl der fertigen Arbeitspakete und -stunden am Ende sowie
- die Anzahl der Teilnehmer.

Das Bild 3.11 zeigt einen typischen Verlauf für die Integrationstage in dem ARTIS-Projekt. Trotz strukturierter Planung zu Beginn (acht Arbeitspakete) ergaben sich im weiteren Verlauf zusätzliche Aufgaben (sechs Arbeitspakete). Insgesamt wurden während der drei Integrationstage circa 70% der Arbeitspakete insgesamt und 100% der Arbeitspakete mit der Priorität A fertiggestellt. Die strukturierte Integration erzielt damit eine höhere Erfolgsquote als das bisherige Vorgehen.

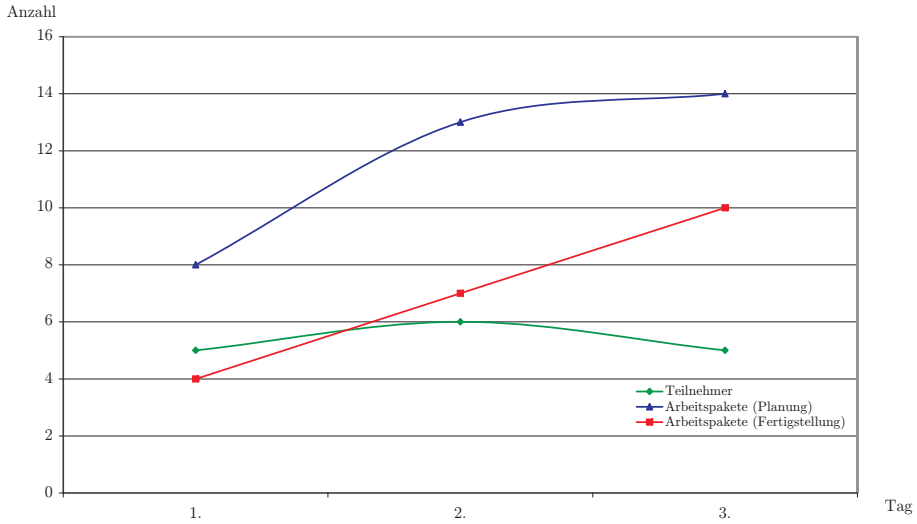

Bild 3.11: Typischer Verlauf der Integrationstage

3.5.2 Modellintegration

Ein Problem bei der Modellintegration ist die Bedingung, dass sowohl die UML State Chart Template-Implementierung als auch der von Rhapsody generierten Code kompatibel zu dem Missionsmanager sein soll, ohne aufwändige Umstrukturierungen vornehmen zu müssen. Das bedeutet zunächst, dass die für das UML State Chart Template entworfenen Schnittstellen in dem Rhapsody-Modell implementiert werden müssen. Die Schnittstellenfunktionen heißen `initSC()` zur Initialisierung sowie `runSC()` zur Ausführung des State Charts. Beiden Funktionen werden als Argumente verschiedene Ein- und Ausgabewerte übergeben. Es stellt sich nun die Frage, wie die Managementfunktionen aus dem OX-Framework mit den Schnittstellenfunktionen kombiniert werden. Eine mögliche Lösung wird nachfolgend erläutert:

1) Die OXF-Initialisierungsfunktion `initStatechart()` wird mit dem Konstruktor der State Chart Klasse verknüpft, da das Instanziieren der Ablaufsteuerung den Initialzustand folgerichtig aktivieren muss.

2) Die OXF-Funktion `startBehavior()` wird in der Funktion `initSC()` eingesetzt, so dass bei deren Ausführung der Zustand ControllerOn gestartet wird.

3) Weiterhin wird die OXF-Funktion `handleEvent()` mit der Schnittstellenfunktion `runSC()` verknüpft, um die generierten Benachrichtigungen aus der Event Middleware zu verarbeiten und somit den State Chart zu durchlaufen.

Ein weiteres Problem ist, dass in der Komponente des Missionsmanagers in einigen Funktionen der aktive Zustand der Ablaufsteuerung abgefragt wird. Diese Aufrufe könnten konsequent dem Rhapsody-Modell angepasst werden, allerdings ist dadurch die weitere Verwendung des UML State Chart Templates nicht mehr möglich. Des Weiteren muss in diesem Fall auch zwischen der flachen bzw. der wiederverwendbaren Konstruktionsmethode für den State Chart gewählt werden. Es wird daher versucht, mit der Methode `GetActiveState()` aus Bild 3.12, welche die genannten Nachteile nicht besitzt, einen Kompromiss zu finden.

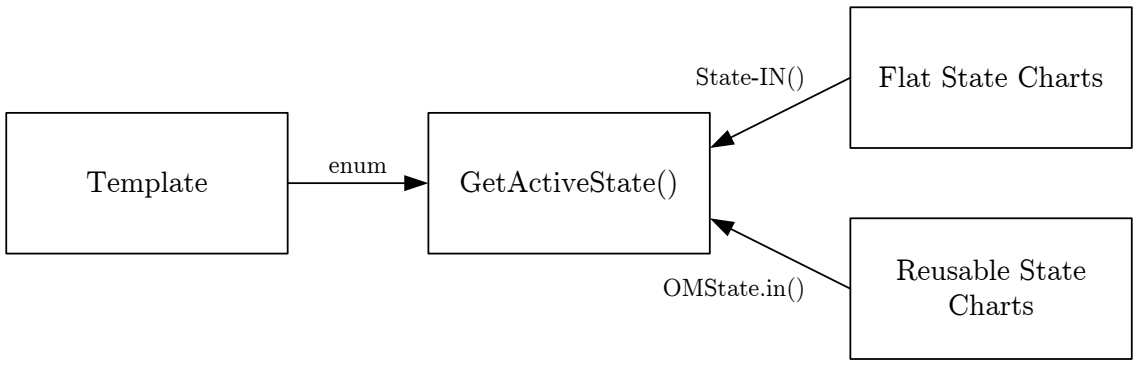

Bild 3.12: Zustandstransformation bei der Integration

Das Ziel dieses Ansatzes ist es, die Zustandsdefinitionen aus dem UML State Chart Template weiter zu verwenden, damit die Template-Implementierung kompatibel bleibt. Realisiert wird dies für die flache Konstruktionsmethode, indem die `State-IN()` Funktionalität aus dem OX-Framework auf die bisherigen Zustandsdefinition projiziert wird. Bei der wiederverwendbaren Konstruktionsmethode sind die einzelnen Zustände von der Superklasse `OMState` abgeleitet, welche die Methode `in()` besitzt. Somit ist auch in diesem Fall eine Projektion auf die Zustandsdefinitionen aus dem UML State Chart Template möglich. Die erforderliche doppelte Implementierung wird durch Präprozessor-Direktiven realisiert, welche je nach Definition (`FLAT` versus `REUSABLE`) den passenden Teil der Methode `GetActiveState()` übersetzen.

Rhapsody unterstützt vorwiegend den Microsoft Visual Compiler, um das Modell in eine eigenständige Applikation zu übersetzten. Die OXF-Bibliothek muss den verwendeten Compilern für die Software-in-the-Loop-Simulation, die Hardware-in-the-Loop-Simulation und den eigentlichen Flugversuch angepasst werden, da die Ablaufsteuerung in den Missionsmanager integriert werden soll, um als Systemkomponente und nicht als eigenständige Applikation ausgeführt zu werden. Das Bild 3.13 zeigt schematisch, wie diese Erweiterungen vorgenommen wurden. In dem ARTIS-Projekt werden die Compiler qcc (QNX Compiler Collection) und gcc (GNU Compiler Collection) zur Übersetzung verwendet, wobei der gcc zur SITL-Simulation benötigt wird und der qcc zur Integration auf dem Bordrechner (HITL-Simulation, Flugversuch). Das heißt, dass die OXF-Bibliothek für beide Compiler erstellt werden muss. Vorteilhaft ist in diesem Zusammenhang, dass die gesamten notwendigen OXF-Dateien mit Rhapsody ausgeliefert werden und somit verfügbar sind.

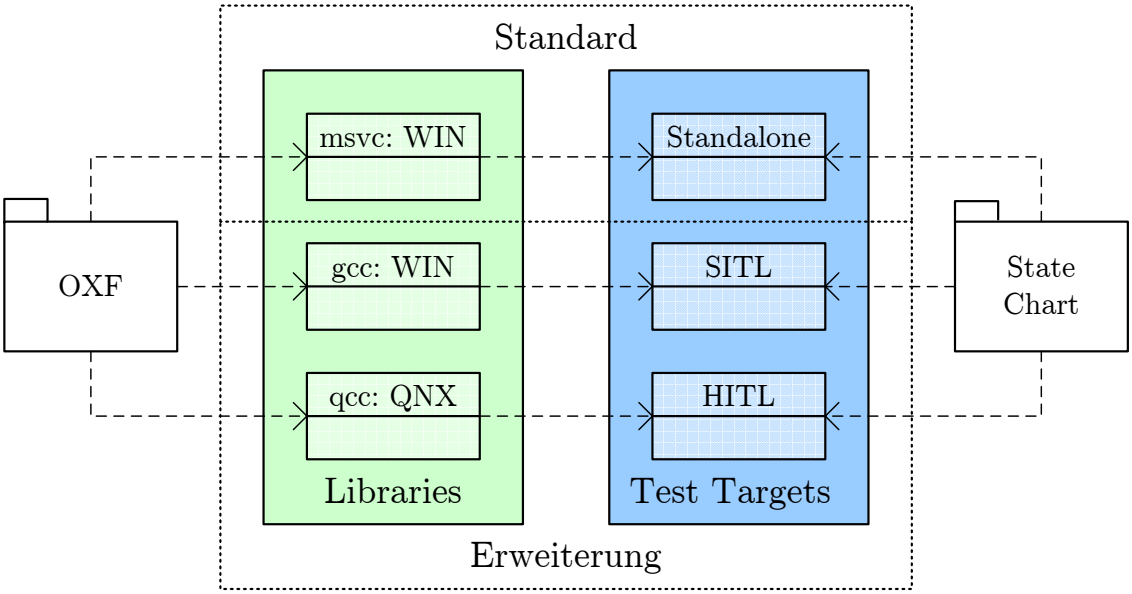

Bild 3.13: Erweiterung des OX-Frameworks

Zunächst wird mit der freien Cross-Compiler Plattform Code::Blocks ein neues Projekt für eine statische Bibliothek angelegt und die notwendigen OXF-Dateien zusammengetragen, welche sich aus den beschriebenen OXF-Managementfunktionen (zum Beispiel `OMReactive`, `OMEvent`, usw.) und weiteren Definitionen, die stets in Abhängigkeit zu dem verwendeten Betriebssystem stehen, zusammensetzen. Die Konfiguration muss in dem Projekt den beiden Zielbetriebssystemen Windows und QNX angepasst werden, so dass der Compiler jeweils die korrekten Betriebssystemdefinitionen verwendet.

Der Ansatz aus der OXF-Dokumentation [7], vorkompilierte Header-Dateien zu verwenden, wird verworfen, da zum Einen die Bibliothek statisch verwendet werden soll und zum Anderen vorkompilierte Header die Bibliothek deutlich vergrößern. Die PThread-Bibliothek [14] muss zusätzlich mit dem Projekt verknüpft werden, da das Thread Management aus dem OX-Framework auf diesem Quellcode basiert.

Ein merklicher Unterschied zwischen dem Visual Compiler von Microsoft und dem gcc bzw. qcc ist die Interpretation von Typedef-Operationen in Verbindung mit dem `Typename` Schlüsselwort. Der Visual Compiler unterstützt diese Operation vollständig, gcc und qcc nicht. Da das OX-Framework hauptsächlich auf Microsoft ausgerichtet ist, ergibt sich an dieser Stelle ein Konflikt, welcher allerdings mit der nicht dokumentierten Präprozessor-Definition `OM_NO_TYPENAME_SUPPORT` lösbar ist. Die Rhapsody-Entwickler berücksichtigten bekannte Compiler-Unterschiede durch die Implementierung Betriebssystem-spezifischer Konfigurationsdateien, welche allerdings wie angesprochen, unvollständig dokumentiert sind. Gut dokumentiert ist jedoch der Einsatz von der Standard Template Library (STL) über die Präprozessor-Definition `OM_USE_STL`.

Durch die in diesem Kapitel erläuterten drei Maßnahmen:

- Integration der Schnittstellenfunktionen,
- Zustandstransformation durch `GetActiveState()` und
- Anpassung der OXF-Bibliotheken

ist es möglich, das UML State Chart Template bzw. die aus dem Rhapsody-Modell generierten Dateien ohne einen Funktionsverlust auszutauschen.

Kapitel 4

Testverfahren

4.1 Vorbetrachtungen

Das Testverfahren wird im Bereich Software-Qualitätsmanagement [5] als grundlegendes Verfahren definiert, mit welchem einzelne Eigenschaften des zu testenden Programms bzw. der zu testenden Komponente durch eine geeignete Anzahl von Testfällen untersucht werden. Das Bild 4.1 zeigt eine Klassifikation dieser Verfahren in Strukturtests, funktionale Tests und vergleichende Tests auf, wobei jedes dieser Verfahren statische und dynamische Aspekte berücksichtigen kann.

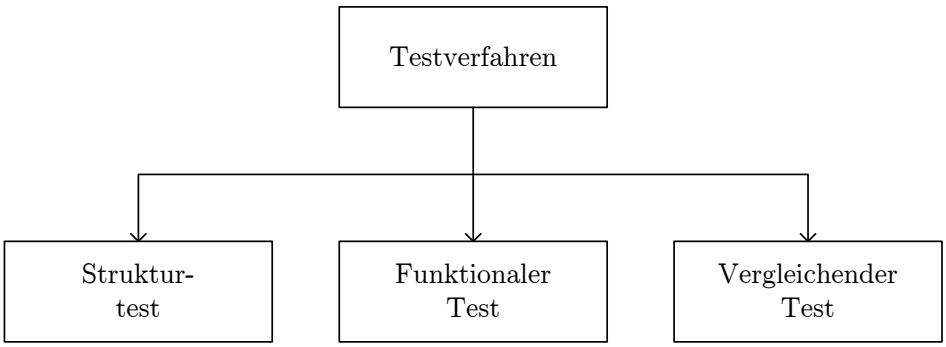

Bild 4.1: Klassifikation testender Verfahren

Die funktionalen Tests prüfen das Programm gegenüber der Spezifikation unter funktionalen Gesichtspunkten. In diesem Zusammenhang ist es auch möglich, Äquivalenzklassen zu bilden und ausgewählte Vertreter zu testen. Vergleichende Tests dienen dazu, die Ergebnisse verschiedener Programmversionen zu vergleichen. Es wird dadurch vermieden, dass die Fehler, die in einer Programmversion behoben wurden, in einer anderen Version übersehen werden. Die Strukturtests werden kontrollfluss- oder datenflussorientiert durchgeführt und basieren auf einem gegebenen Modell bzw. auf dem vorliegenden Quellcode.

Um das Modell der Ablaufsteuerung zu testen, sollen neben den funktionalen Tests auch die Strukturtests auf Kontrollflussbasis mit dazugehöriger Zustands-, Transitions- und Pfadüberdeckung verwendet werden. Ein Testdurchlauf für 100% Zustandsüberdeckung besitzt für jeden erreichbaren Zustand einen Testfall. Dieser wird als Sequenz von Eingaben ab dem Initialzustand modelliert. Die Zustandsüberdeckung ist ein notwendiges, aber nicht hinreichendes Kriterium, da nicht jede ausführbare Funktion geprüft wird. Die Zustandsüberdeckung weist nach [5] und [24] somit die niedrigste Fehleridentifizierungsquote auf .

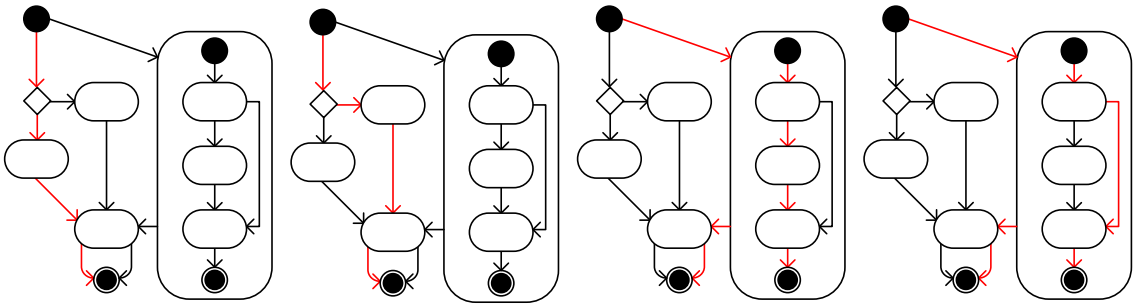

Bild 4.2: Transitionsüberdeckung

Ein Testdurchlauf für 100% Transitionsüberdeckung zeichnet sich durch Testfälle aus, die jede Transition in dem State Chart auslösen, beispielhaft in Bild 4.2 dargestellt. Die Transitionsüberdeckung gilt nach [5] als minimales Testkriterium , da diese zumindest die Nachteile von der Zustandsüberdeckung ausgleicht und jede Funktion mindestens einmal ausführt. Allerdings ist die Transitionsüberdeckung unzureichend bei dem Testen von Schleifen.

Die Pfadüberdeckung zeichnet sich durch Testfälle von jedem beliebigen Anfangszustand zu jedem möglichen Endzustand aus, die Zustands- und die Transitionsüberdeckung sind folglich deren Teilmenge. Die Pfadüberdeckung hat praktisch keine Relevanz und wird auf die minimale Schleifenüberdeckung reduziert, da unter Berücksichtigung von Schleifen die Menge der Pfade in einem State Chart theoretisch unendlich ist. Bei der minimale Schleifenüberdeckung werden die Schleifen mindestens einmal durchlaufen. Dieses Kriterium hat sich nach [17] in der Vergangenheit als stark und dennoch praktikabel erwiesen.

Da die modellierte Ablaufsteuerung ein konstruktives Modell darstellt, eignet sich der Einsatz von White-Box-Tests, wenn diese den aus dem Modell generierten Quellcode testen sollen [28]. Die Tests sollen in dieser Arbeit auf dem CppUnit-Testframework basieren und sowohl flache als auch wiederverwendbare State Charts gleichermaßen testen. Bei der Auswahl der Testumgebung muss darauf geachtet werden, dass alle Transitionen schaltbar und somit alle Zustände erreichbar sind. In einer Simulationsumgebung wie SITL oder HITL ist dies nicht immer möglich und muss daher in den zu konzipierenden Tests berücksichtigt werden.

4.1.1 Die Teststrategie

Da die Funktionalität der Ablaufsteuerung durch das Re-Engineering auf mehrere Klassen verteilt ist, besteht die Möglichkeit, diese separat mit drei Test Suiten zu testen (Bild 4.3). Die erste Test Suite (EMT) überprüft die Funktion der Event Middleware, in dem die Eingabewerte unterschiedlich kombiniert werden, so dass die Detektierung der Ereignisse von dem Sender ausgehend getestet werden kann. Weiterhin findet mit dieser Test Suite eine Überprüfung des Filters statt, dazu wird der State Chart durch ein abgeleitetes Modell ersetzt, welches nur die Grundfunktionen zum Setzen und Auslesen von aktuellem Zustand und Ereignis bietet.

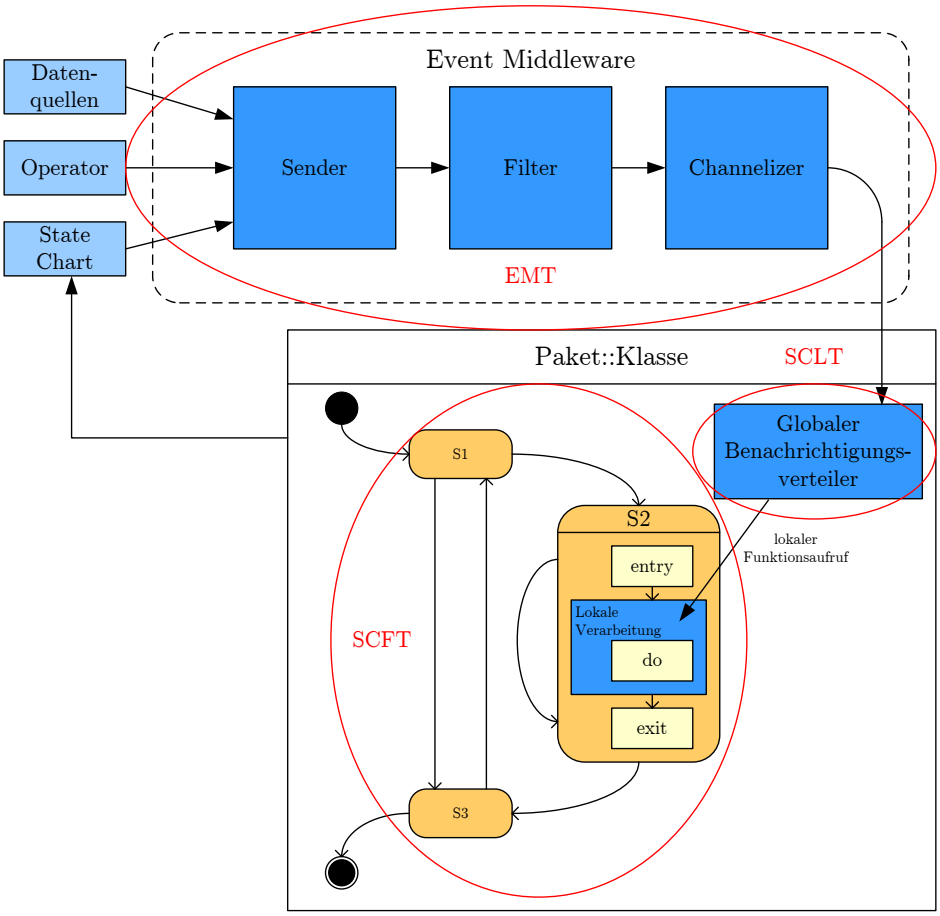

Bild 4.3: Testansatz für die drei Test Suiten

Im zweiten Schritt wird die logische Struktur des State Charts und der damit verbundene Benachrichtigungsverteilter überprüft. Die dafür verwendete Test Suite (SCLT) ersetzt die Event Middleware durch einen Event Generator, welcher Benachrichtigungsketten beliebiger Länge generiert, die anschließend von dem Benachrichtigungsverteilter sequentiell abgearbeitet werden. Das Hauptaugenmerk dieser Test Suite fokussiert sich dabei auf die korrekt ausgeführten Zustandswechsel. Die Entry-, Exit- und Do-Aktivitäten werden durch Präprozessor-Direktiven deaktiviert, um Seiteneffekte zu vermeiden.

Erst mit der dritten Test Suite (SCFT) werden die internen Aktivitäten getestet, dazu wird der State Chart in einen beliebigen Zustand versetzt, die Do-Aktivität ausgelöst und die interne Variablenveränderung überwacht. Anschließend kann über einen Zustandswechsel die Exit-Aktivität des einen und die Entry-Aktivität des folgenden Zustands überprüft werden.

4.1.2 Der Testautomat

Die Teststrategie wird mit einem Testautomaten als modellbasierter Ansatz realisiert, da der Testautomat, wie die eigentliche Systemkomponente auch, mit Rhapsody als hierarchischer State Chart modelliert werden kann. Dieser State Chart besteht aus den drei zusammengesetzten Zuständen, welche in Bild 4.4 dargestellt sind. Der erste Zustand (Test Key Generator) generiert einen (binären) Testschlüssel, welcher für die beiden nachfolgenden Zustände (Build und Run Test Case) die Abfolge der Transitionen zwischen den internen Teilzuständen festlegt. Dieser Schlüssel bestimmt beispielsweise für die Event Middleware die Konfiguration der Eingabewerte zur Detektierung der Ereignisse.

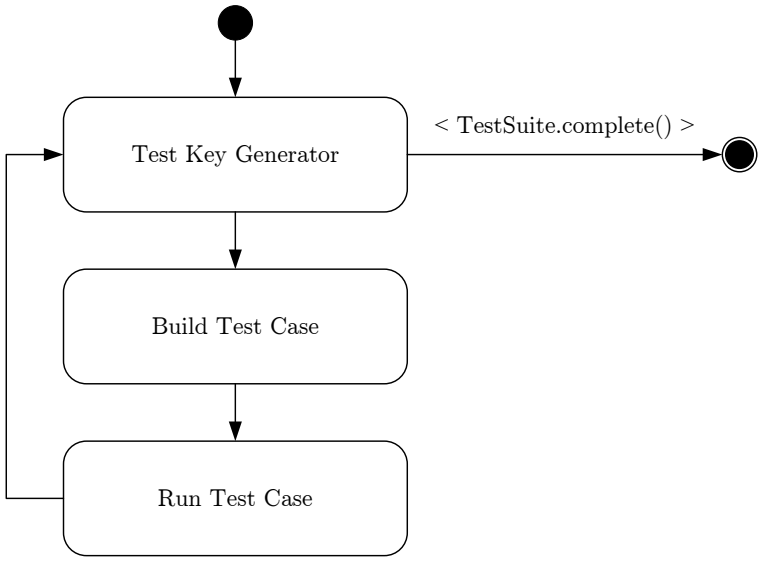

Bild 4.4: Der Testautomat

Der zweite Zustand (Build Test Case) wird anschließend nach Vorgabe des Testschlüssels durchlaufen, es werden dabei die notwendigen Eingabewerte für den Testfall festgelegt. In der zweiten Test Suite wird beispielsweise in diesem Zustand die angesprochene Benachrichtigungskette für den Strukturtest generiert.

Abschließend werden in dem dritten Zustand (Run Test Case) die vorbereiteten Testfälle ausgeführt und die Ergebnisse ausgewertet. Im Fehlerfall wird ein ausführlicher Report über den Testfall selbst, die eigentliche Spezifikation sowie das tatsächliche Systemverhalten der Ablaufsteuerung erstellt. Der Testautomat wechselt

in den Endzustand, wenn entweder 100% Zustands- und Transitionsüberdeckung erreicht sind bzw. die berechnete Anzahl der möglichen Testfälle ausgeführt wurden.

Bei der Code-Generierung eines State Charts wird, wie in Kapitel 3.4 beschrieben, die State Chart Klasse von der Superklasse OMReactive abgeleitet. Bei den Testautomaten werden durch eine zusätzliche Ableitung von dem CppUnit-Testframework zusätzliche Methoden eingebunden. Diese umfassen Methoden zum Aufsetzen, zur Ausführung und zur Auflösung einer Testumgebung. Für den Vergleich von Erwartung und Ergebnis werden u.a. die Makros aus Listing 4.1 verwendet.

```
CPPUNIT_ASSERT_EQUAL(expected, actual)
CPPUNIT_ASSERT_MESSAGE(message, expected, actual)
```

Listing 4.1: Verwendete CppUnit-Makros

Die Testautomaten sind einer manuell implementierten Test Suite durch den modellbasierten Ansatz und die zusätzliche Integration des CppUnit-Testframeworks in zwei Punkten überlegen: Zum Einen wird das Testszenario in einer abstrakteren Sprache übersichtlich modelliert und zum Anderen werden alle erforderlichen Testfälle kombinatorisch abgedeckt. Die Testautomaten ermöglichen somit das automatisierte Testen in einer sehr ausgeprägten Form.

4.1.3 Der Fehlerreport

Damit ein Gesamtbild über die getestete Komponente gebildet werden kann, ist es sinnvoll, die Testdurchläufe zu protokollieren. In einem später Flugtest dürfen beispielsweise nur die Funktionen ausführbar sein, die ausgiebig überprüft wurden.

Alle drei Test Suiten sollen daher im Fehlerfall einen Report erstellen, welcher als Datei speicherbar und damit in das Mantis-Bugtracking-System einstellbar ist. Dieser Report besteht aus den folgenden drei Abschnitten:

1) Zunächst wird der Testfall eindeutig beschrieben, dazu gehören zum Beispiel die verwendeten Eingabegrößen, die Benachrichtigungsketten oder der Initialzustand. Aus diesem Abschnitt muss sich präzise erschließen lassen, wie der Testfall erstellt wurde.

2) Im zweiten Abschnitt wird der erzeugte Fehler des CppUnit-Testframeworks dargestellt. Es wird der Erwartungswert des Testfalls und damit die Spezifikation der Ablaufsteuerung beschrieben.

3) Im abschließenden Abschnitt des Reports wird der tatsächliche Systemwert dargestellt. Dieser Wert weicht von der Spezifikation aus dem zweiten Abschnitt ab, da es sich um einen Fehlerfall handelt. Die Debug-Nachrichten aus der Ablaufsteuerung werden zur Darstellung des Systemwertes verwendet.

4.2 Test Suite für die Event Middleware

Mit Hilfe dieser Test Suite soll die Funktionalität der Event Middleware überprüft werden. Hierzu wird zunächst das State Chart Modell durch ein abgeleitetes Modell ersetzt, in welchem die Methoden des Benachrichtigungsverteilers und der Zustandstransformation überladen werden (Function Overloading). Zusätzlich wird eine Methode für das Setzen und Auslesen von Zustand und Benachrichtigung eingefügt.

Nach vorrausgehender Analyse der verschiedenen Auslösebedingungen für die detektierbaren Ereignisse des Senders werden die notwendigen Eingabewerte für die Event Middleware gesetzt. An dieser Stelle kann erstmals die Detektierung der Auslösebedingungen überprüft werden. Anschließend wird der gewünschte Zustand gewählt, da die Zustandsabfrage durch das abgeleitete Modell manipulierbar ist, und die Event Middleware aufgerufen. Diese verarbeitet die gegebenen Eingabewerte sowie den manipulierten Zustand und sendet ein Benachrichtigungsobjekt in Richtung des abgeleiteten Modells.

Diese Benachrichtigung wird durch den überladenen Verteiler abgefangen und kann abschließend ausgewertet werden. Um eine möglichst hohe Testabdeckung der Funktionalität zu erreichen, wird in dieser Test Suite jede mögliche Kombination von Eingabewerten und State Chart Zuständen geprüft.

4.2.1 Analyse

Die Anzahl der möglichen Testfälle (N_α) ergibt sich für diese Test Suite kombinatorisch aus der Anzahl der Zustände in dem State Chart (N_s) und der Anzahl der dafür detektierbaren Ereignisse in dem Sender der Event Middleware (N_ϕ).

$$N_\alpha(N_s, N_\phi(s)) = \sum_{s=1}^{N_s} 2^{N_\phi(s)} \tag{4.1}$$

Um die Robustheit der Komponente zu gewährleisten, werden alle Ereignisse des Senders für jeden einzelnen Zustand des State Charts geprüft. Die Gleichung 4.1 wird somit zu Gleichung 4.2 umgeformt:

$$N_\alpha(N_s, N_\phi) = N_s \cdot 2^{N_\phi} \tag{4.2}$$

Zunächst wurde jedoch analysiert, welche Ereignisse in dem Sender detektiert werden können und welche Eingabegrößen dies beeinflussen. Dieser Zusammenhang ist in der Tabelle 4.1 dargestellt und verdeutlicht, dass zur Detektierung einiger Ereignisse die gleichen Eingangsgrößen, aber mit einer unterschiedlichen Intention, überprüft werden. Die Variable vdTarget, die eigentlich als Zwischenspeicher (u.a. für die Zielposition) gedacht ist, wird mehrfach überlagert und kann in seltenen Fälle zum Systemabsturz führen. Die Analyse führte dazu, die Indizierung der Eingangsgrößen neu zu definieren.

	arDirectRC	arDirectGCS	asSysHealth	arNEuler	argNPos	arNHeight	bMissionMode	bMissionUpdated	arNVel	behaviorsequence/behavior	vdTarget	SysDeltaT	arCPath
Flyable			X										
AutoPilot	X												
ManualPilot	X												
CommandStop		X											
MissionModeOn							X						
MissionUpdated								X					
MissionFinished										X			
StandStill									X				
Landed						X						X	
HeightReached					X						X		
PositionReached					X						X		
HeadingReached				X							X		
PirouetteFinished													X
TimeOut											X	X	
DirectCommandValid						X					X		
DirectCommandFinished					X	X					X	X	

Tabelle 4.1: Beziehung zwischen Eingabewerten und Ereignissen

Die getakteten Eingabewerte werden prinzipiell in zwei Kategorien eingeteilt: diskretisierte reelle bzw. binäre Werte. Beispielsweise basiert die Positionsangabe des GPS-Systems (`argNPos` aus Tabelle 4.1) auf reellen Werten, welche sich mit jedem Takt des Systems diskret ändern. Eine Ereignisdetektierung auf Basis reeller Werte findet stets innerhalb vorgegebener Schranken statt. Es wird zum Beispiel mit den Ungleichungen 4.3 überprüft, ob die tatsächliche Position ($\vec{v}_{GPS}$) und die Zielposition ($\vec{v}_{Target}$) unter Berücksichtigung eines Schwellwertes (δ) identisch sind.

$$\begin{aligned} \delta &> |x_{Target} - x_{GPS}| \\ \delta &> |y_{Target} - y_{GPS}| \\ \delta &> |z_{Target} - z_{GPS}| \end{aligned} \tag{4.3}$$

Binäre Eingaben, wie beispielsweise das Umschalten auf den automatischen Betriebsmodus durch den Sicherheitspiloten (`arDirectRC` aus Tabelle 4.1), werden dagegen ohne Schwellwert geprüft (Bild 4.5). In diesem Fall kann somit nicht das Verhalten von einem Zustand bei dem Ereignis AutoPilot und ManualPilot getestet werden, da sich beide gegenseitig ausschließen und somit nicht gleichzeitig überprüft werden müssen.

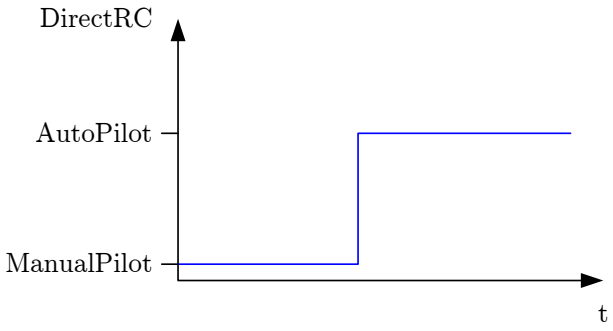

Bild 4.5: Binäre Eingabegrößen am Beispiel arDirectRC

Es werden mit dieser Test Suite jeweils alle Kombinationen der 15 Ereignisse aus Tabelle 4.1 für jeden der 23 Zustände geprüft. Es ergibt sich daher aus Gleichung 4.2 die folgende Anzahl von Testfällen:

$$N_\alpha(23, 15) = 23 \cdot 2^{15} = 753.664 \ . \tag{4.4}$$

4.2.2 Test Key Generation

In der Komponente „Test Key Generation“ wird für diese Test Suite der Testschlüssel generiert, welcher aus zwei Abschnitten (Bild 4.6) besteht. Im ersten Abschnitt („State Key“) wird festgelegt, welcher Zustand im abgeleiteten Modell für den Testfall gesetzt werden soll. Es ist in der Test Suite möglich, die Ereignisdetektierung durch den Sender für einen, mehrere oder alle Zustände nacheinander zu prüfen.

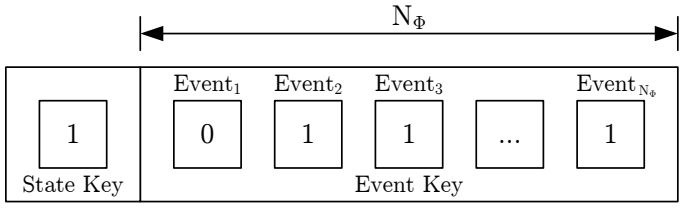

Bild 4.6: Testschlüssel in EMT

Der zweite Testschlüsselabschnitt („Event Key“) legt für die N_ϕ Ereignisse fest, ob sie detektiert (True) oder nicht detektiert (False) werden sollen. In diesem Abschnitt werden die Kombinationen nach jedem Testfall vom linken Wert ausgehend verändert. In der Tabelle 4.2 ist der Ablauf beispielhaft für einen Zustand mit drei Ereignissen abgebildet. Die Test Suite gilt als vollständig durchlaufen und wird beendet, wenn die berechnete Anzahl der Testfälle N_α absolviert ist.

Testcase	State	$Event_1$	$Event_2$	$Event_3$
1	1	0	0	0
2	1	1	0	0
3	1	0	1	0
4	1	1	1	0
5	1	0	0	1
6	1	1	0	1
7	1	0	1	1
8	1	1	1	1

Tabelle 4.2: Beispiel für einen Testschlüssel mit $N_s = 1$ und $N_\phi = 3$

4.2.3 Build Test Case

In der Komponente „Build Test Case“ werden für jedes Ereignis die Eingabegrößen entsprechend des Testschlüssels gesetzt. An dieser Stelle kann erstmals die Detektierung der Auslösebedingungen mit dem CppUnit-Testframework überprüft werden, ohne die Event Middleware oder das abgeleitete Modell zu verwenden.

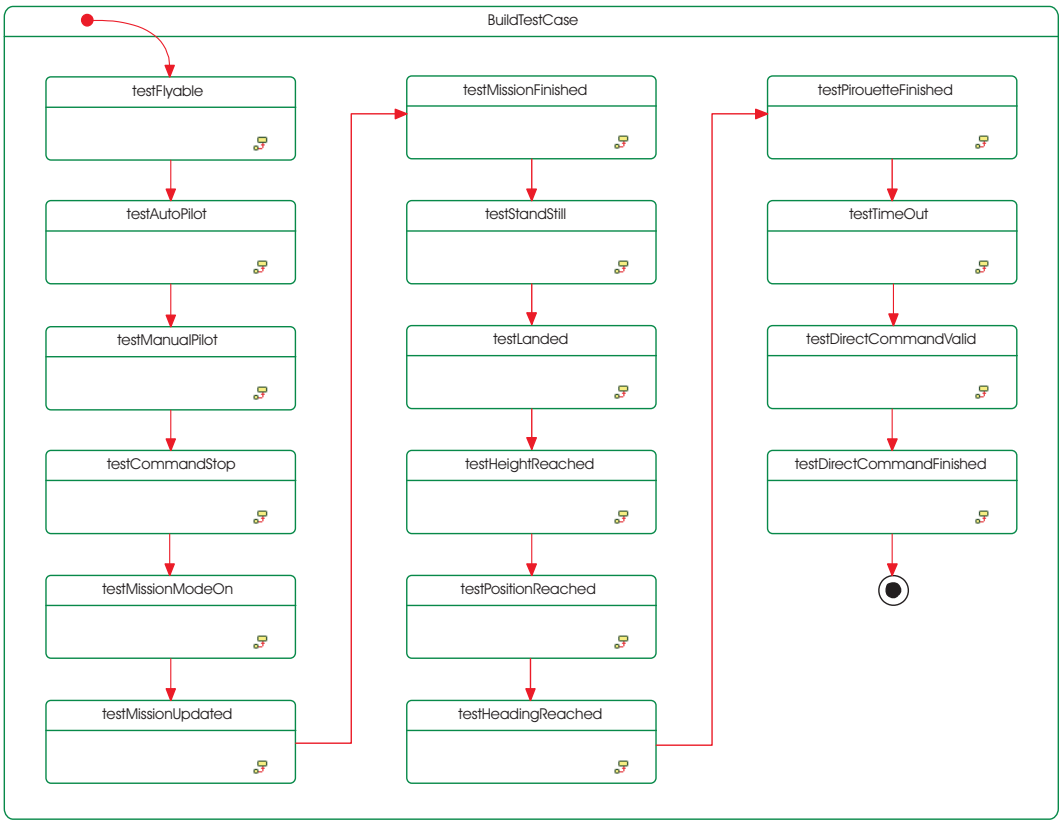

Bild 4.7: Build Test Case in EMT

Die Zustände „testFlyable“ bis „testDirectCommandFinished“ aus Bild 4.7 sind jeweils zusammengesetzte Zustände und bestehen aus dem in Bild 4.8 beispielhaft gezeigten State Chart zur Detektierung des Ereignisses PositionReached. Ist der Testschlüsselwert für das entsprechende Ereignis 1 (True), wird der linke Pfad durchlaufen und die Eingabegrößen so gesetzt, dass das Ereignis detektiert wird, andernfalls wird der rechte Pfad ohne Ereignisdetektierung absolviert. Die Pfadauswahl wird in globalen Variablen der Testklasse gespeichert und erleichtert die Spezifikation für die dritte Komponente dieser Test Suite.

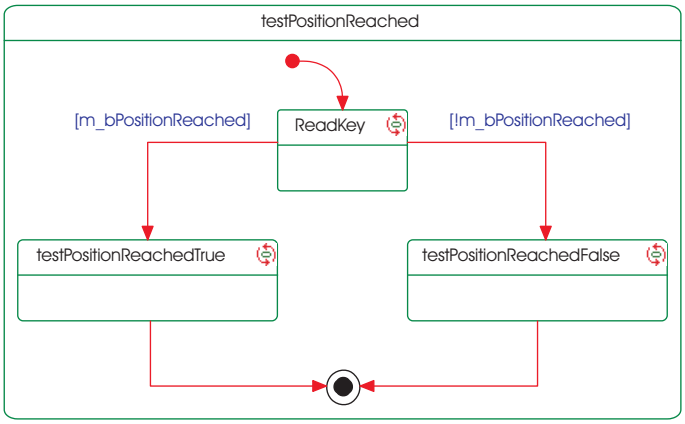

Bild 4.8: Beispiel für das Ereignis PositionReached

Das Listing 4.2 zeigt, wie dieser Vorgang für den Testschlüsselwert 1 (True) bei dem Ereignis PositionReached umgesetzt wird. Zunächst wird der Schwellwert δ (`threshold`) aus der Konfiguration geladen, da dieser für jeden einzelnen Hubschrauber (ARTIS/maxiARTIS) separat festgelegt ist. Anschließend wird dieser Schwellwert in jede Koordinate des Zielvektors $\vec{v}_{Target}$ und der tatsächlichen Position $\vec{v}_{GPS}$ (`argNPos`) geschrieben, wobei für die tatsächliche Position mit der Genauigkeitskonstante ε (`eps`) skaliert wird. In diesem Fall wird mit einer Genauigkeit von $\frac{1}{1000}$ getestet, die tatsächliche Position ist somit geringfügig kleiner als die Zielposition.

```
threshold = config->getThres_SuccessPosXYZ();

vdTarget.at(X) = vdTarget.at(Y) = vdTarget.at(Z) = threshold;

// case true, threshold*(1-eps) < threshold
input->set_argNPos( 2*threshold*(1-eps),
                    2*threshold*(1-eps),
                    2*threshold*(1-eps) );

CPPUNIT_ASSERT_MESSAGE ("testConditionPositionReached", true ==
    condition->PositionReached( vdTarget.at(X), vdTarget.at(Y),
    vdTarget.at(Z), input->get_argNPos(), threshold ) );
```

Listing 4.2: Testfall PositionReached (True)

Abschließend wird überprüft, ob das Ereignis tatsächlich detektiert wird. Ist dies nicht der Fall, bricht die Test Suite mit einer Fehlermeldung ab. Unter Berücksichtigung der Ungleichung 4.3 und $\varepsilon = \frac{1}{1000}$ ergibt sich jedoch für diesen Fall jeweils für x, y und z die folgende wahre Aussage, ohne dass der Schwellwert δ tatsächlich gegeben sein muss.

$$\begin{aligned} \delta &> |\delta - \delta \cdot (2 - 2 \cdot \varepsilon)| \\ 1 &> |-1 + 2 \cdot \varepsilon| \\ 1 &> \tfrac{998}{1000} \end{aligned} \tag{4.5}$$

Ist der Schlüsselwert gleich 0 (False) wird für $\vec{v}_{GPS}$ jeweils $1 + \varepsilon$ anstatt $1 - \varepsilon$ für die Skalierung eingesetzt. Somit ergibt sich für x, y und z die Ungleichung 4.6, das Ereignis PositionReached wird nicht detektiert.

$$\begin{aligned} \delta &\not> |\delta - \delta \cdot (2 + 2 \cdot \varepsilon)| \\ 1 &\not> |-1 - 2 \cdot \varepsilon| \\ 1 &\not> \tfrac{1002}{1000} \end{aligned} \tag{4.6}$$

4.2.4 Run Test Case

In der Komponente „Run Test Case“ wird der erstellte Testfall abschließend ausgeführt, dazu wird in dem abgeleiteten Modell der Zustand, welcher durch den Testschlüssel bestimmt wird, über die Methode `SetActiveState` gesetzt. Die generierten Eingabegrößen werden anschließend der Event Middleware übermittelt. Diese besitzt folglich alle notwendigen Informationen (einschließlich korrektem Zustand) und kann ausgeführt werden. Die von der Event Middleware erzeugte Benachrichtigung wird durch den überladenen Benachrichtigungsverteiler im abgeleiteten Modell abgefangen und kann mit der Spezifikation verglichen werden.

Die Spezifikation wird für jeden Zustand separat in dem State Chart aus Bild 4.9 formuliert, indem mit einer elementaren Syntax angegeben wird, bei welchen Eingabegrößen die Ablaufsteuerung welche Benachrichtigungen erwartet. Das Listing 4.3 zeigt die gekürzte Spezifikation für den Zustand HoverTo.

```
if ( m_dTestCaseState == eStateHoverTo )
{
    m_sTestCaseMsg.append("In testStateHoverTo ");

    if ( m_bManualPilot )
    {
        m_sTestCaseMsg.append("expected evManualPilot!\n");
        m_evID_TC = evManualPilot_LIB_MISSION_CORE_id;
    }

    ...
```

```
    if ( !m_bManualPilot && !m_bCommandStop &&
         !m_bMissionUpdated && !(m_bPositionReached &&
          m_bHeadingReached && m_bStandStill) )
    {
        m_sTestCaseMsg.append("expected evPulse!\n");
        m_evID_TC = evPulse_LIB_MISSION_CORE_id;
    }
}
```

Listing 4.3: EMT-Spezifikation für den Zustand HoverTo

Durch den State Chart aus Bild 4.8 ist durch die globalen Variablen bekannt, welche Ereignisse (z.B. PositionReached) der Sender detektieren soll, somit muss lediglich spezifiziert werden, wie die Prioritäten für die erwarteten Benachrichtigungen in der Event Middleware modelliert sein sollten. Anschließend wird die spezifizierte Identifizierungskennzahl (`evID-TC`) der erwarteten Benachrichtigung mit dem tatsächlichen Wert (`evID-SC`) über die Methode `GetCurrentEvent()` im abgeleiteten Modell verglichen.

Des Weiteren wird für jede Spezifikation gleichzeitig eine Textausgabe für den Fehlerfall vorbereitet, in welcher exakt beschrieben wird, welcher Zustand bei welchen Eingabegrößen welche Benachrichtigung in dem State Chart der Ablaufsteuerung erwartet.

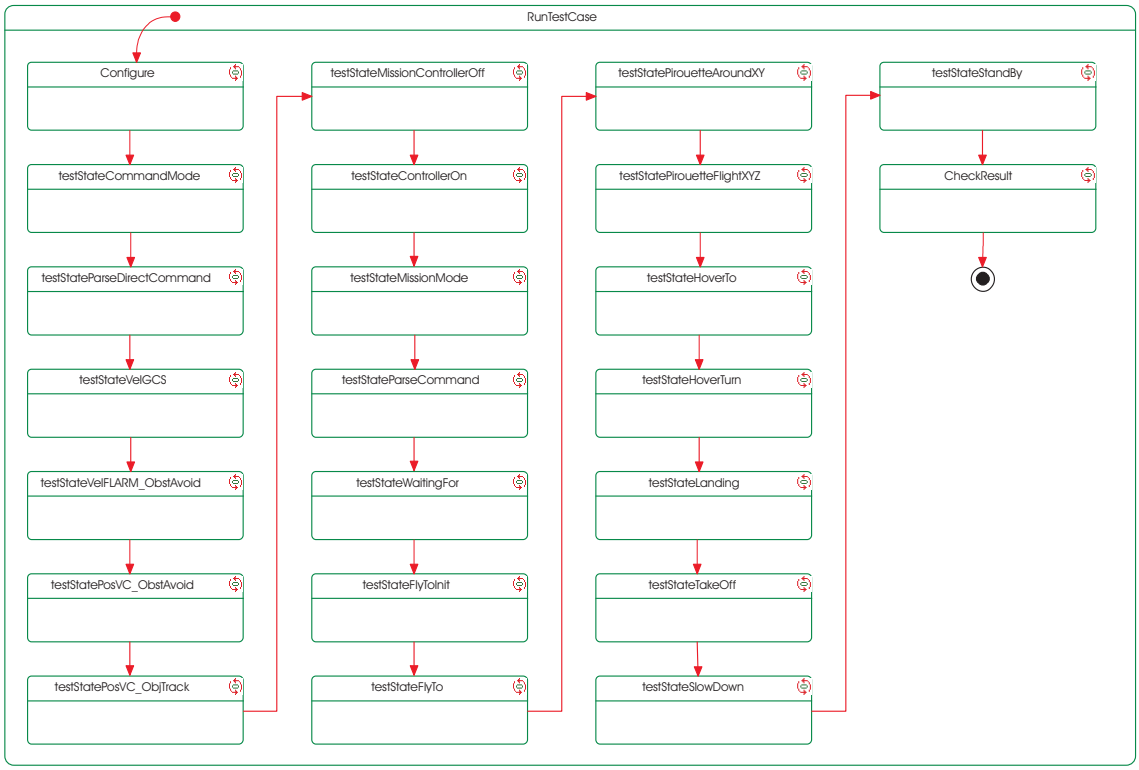

Bild 4.9: Run Test Case in EMT

4.2.5 Report

Falls in dieser Test Suite das CppUnit-Testframework einen Fehler während eines Testfalls registriert, wird der nachfolgende Report ausgegeben. Dieser Report gliedert sich dabei in drei wesentliche Bereiche. Der erste Abschnitt beschreibt, welche Ereignisse im Sender detektiert werden sollen. Durch diese Beschreibung ist nachvollziehbar, wie die Eingabegrößen für die Event Middleware konstruiert wird. Im zweiten Abschnitt wird die Fehlerquelle beschrieben, das heißt, welcher Zustand durch den Testschlüssel in der Ablaufsteuerung aktiv ist und welche Benachrichtigungserwartung dieser besitzt. Abschließend kann im dritten Abschnitt verfolgt werden, welchen Ablauf die Komponente aufweist, das heißt, welche Benachrichtigung tatsächlich in Richtung der Ablaufsteuerung gesendet wird.

```
*********************************************
Test Case
*********************************************

AutoPilot
CommandStop

*********************************************
Test Case Error Message
*********************************************

In testStateHoverTo expected evPulse!

*********************************************
SequenceControlDummy Output
*********************************************

SC: Call EventMiddleware...
SC: EP: Load Arguments...
SC: EP: Load Arguments Finished...
SC: SCDummy: GetActiveState(): 9
SC: EP: MissionMode: Check For Event...
SC: SCDummy: GetActiveState(): 9
SC: EP: MissionMode: Found evMissionModeStop...
SC: EP: Store Arguments...
SC: EP: Store Arguments Finished...
SC: EP: Send Event To SC...
SC: Call EventMiddleware Finished...
```

Der abgebildete Report zeigt beispielhaft, wie eine solche Fehlermeldung für den Zustand HoverTo[1] aussehen kann. Zunächst werden die Eingabegrößen so beschrieben,

[1] Der Zahlwert für den Zustand HoverTo ist in dem Missionsmanager neun.

dass der Sender zum Einen das Signal für den automatischen Flug vom Sicherheitspiloten erhält und zum Anderen die Nachricht von der Bodenstation empfängt, den Missionsmodus abzubrechen. Als Spezifikation wird für diesen Fall das Taktsignal `evPulse` erwartet, wobei in diesem Beispiel die Spezifikation bewusst falsch formuliert ist. Im dritten Abschnitt zeigt die Systemausgabe das tatsächliche Verhalten der Event Middleware, in diesem Fall wird die Benachrichtigung `evMissionModeStop` zur Ablaufsteuerung gesendet. Dieses Verhalten ist bei der gewählten Bodenstationsnachricht unter Berücksichtigung von Zustand und Priorität korrekt.

4.2.6 Ergebnisse

Mit dieser Test Suite werden prinzipiell alle Veränderungen in der Event Middleware in Bezug auf Sender und Filter erkannt. Die Schwerpunkte liegen dabei auf:

- Ereignisdetektierung durch den Sender auf Basis der Eingabegröße,
- Auswahl des korrekten Haupt und Teilzustand sowie
- Beachtung der vordefinierten Prioritäten in dem Filter.

Der vollständige Durchlauf benötigt circa 120 Sekunden. Die Zeit Δt, die zur vollständigen Prüfung eines Zustands benötigt wird, ist mit circa fünf Sekunden nahezu konstant. Bei jedem weiteren Zustand vergrößert sich die Durchlaufzeit somit um Δt. Im Gegensatz dazu verdoppelt sich die Durchlaufzeit mit jedem weiteren Ereignis, das hinzugefügt wird. Diese Abhängigkeiten sind in Bild 4.10 dargestellt.

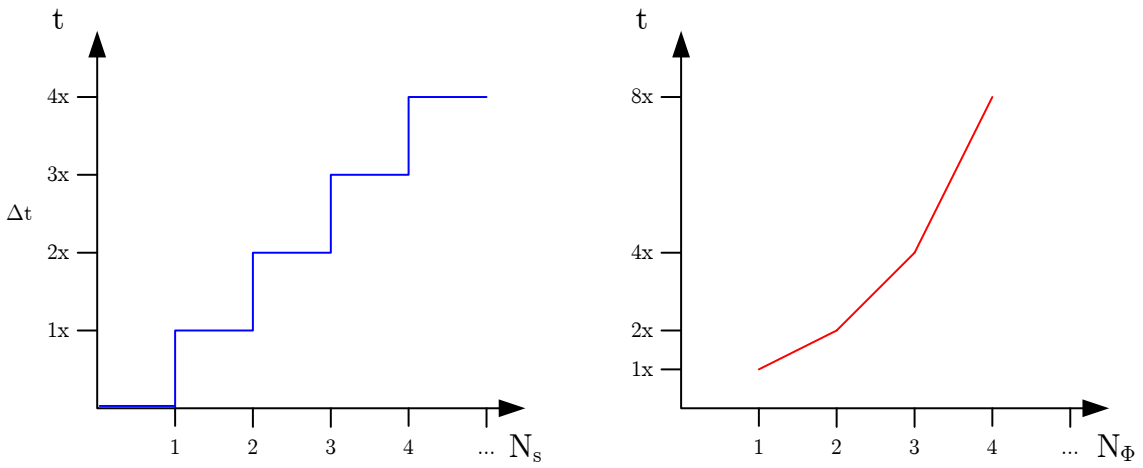

Bild 4.10: Abhängigkeit der Durchlaufzeit von N_s und N_ϕ

Aus der Anzahl der Testfälle N_α und deren Durchlaufzeit t lässt sich nun die durchschnittliche Zeit für einen Testfall errechnen. In diesem Fall ergibt sich:

$$\frac{t}{N_\alpha(23,15)} = \frac{120s}{753.664} = 1,59 \cdot 10^{-4}s. \tag{4.7}$$

Unter Berücksichtigung der Rechenleistung des verwendeten Computers (3400 MHz) und der des ARTIS-Bordrechners (1400MHz) wird die Rechenzeit für den Testfall auf des ARTIS-System projiziert:

$$1,59 \cdot 10^{-4}s \cdot \frac{3,4 \cdot 10^9 \frac{1}{s}}{1,4 \cdot 10^9 \frac{1}{s}} = 3,87 \cdot 10^{-4}s \tag{4.8}$$

Dieser Wert dient als grobe Abschätzung, da jedes Betriebssystem interne Anwendungen, wie die des Taskmanagers, parallel weiterführt. Durch den dreistufigen Aufbau des Testautomaten erfordert der Testfall etwas mehr Rechenzeit als die eigentliche Ausführung der zu testenden Komponente verbraucht. Für das System im Flugbetrieb lässt sich aber die Ausführungszeit für die Event Middleware mit einem Wert, der kleiner ist als $3,87 \cdot 10^{-4}$ Sekunden, abschätzen.

4.3 Test Suite für die logische Struktur

Damit die logische Struktur des State Charts getestet werden kann, wird in dieser Test Suite zunächst die Event Middleware durch einen Event Generator ersetzt. Dieser ermöglicht die Generierung verschiedener Benachrichtigungsketten beliebiger Länge. Des Weiteren wird die Funktionalität der internen Aktivitäten der Zustände deaktiviert, damit der Strukturtest nicht durch Seiteneffekte verfälscht wird. Es bieten sich dafür zwei Ansätze an: Zum Einen kann dies durch die Konfiguration vor der Code-Generierung umgesetzt werden, zum Anderen ist auch der Einsatz von Präprozessor-Direktiven (zum Beispiel `#ifndef`) möglich.

Ein Testfall besteht in dieser Test Suite somit aus der sequentiellen Abarbeitung der generierten Benachrichtigungskette. Es wird dabei von einem Initialzustand ausgehend jeder Folgezustand ausgewertet. Damit sich der Testaufwand reduzieren lässt, wird das Prinzip der minimalen Schleifenüberdeckung berücksichtigt. Hierfür werden Wiederholungen der Paare (Zustand, Benachrichtigung) detektiert.

Der Benachrichtigungsverteiler soll unbekannte Benachrichtigungen für einen Zustand ignorieren, da der State Chart unvollständig modelliert ist. Daher ist es folgerichtig, nicht erforderliche Benachrichtigungsketten mindestens einmal überprüfen zu lassen, um diese Art der Modellierung zu testen.

4.3.1 Analyse

Die Anzahl der möglichen Testfälle (N_β) ergibt sich für diese Test Suite kombinatorisch aus der Anzahl der möglichen Benachrichtigungsobjekten (N_φ) und der Anzahl der Zustände (N_s), aus denen die Anzahl unterschiedlich langer Benachrichtigungsketten (L) der Länge n beginnend abgearbeitet werden.

$$N_\beta(N_s, N_\varphi) = \sum_{s=1}^{N_s} \sum_{l=1}^{L(s)} (N_\varphi)^{n(s,l)} \tag{4.9}$$

Zur Vereinfachung der Gleichung 4.9 wird für jeden Durchlauf festgelegt, dass von jedem Initialzustand nur eine Benachrichtungskette konstanter Länge betrachtet werden soll. Es ergibt sich somit die Gleichung 4.10:

$$N_\beta(N_s, N_\varphi, n) = N_s \cdot (N_\varphi)^n \tag{4.10}$$

Wenn in der Test Suite nur von einem Startzustand ($N_s = 1$) ausgehend die Benachrichtigungskette verarbeitet wird, muss die Länge dieser Kette der minimalen Tiefe des Kontrollflussgraphen entsprechen, mit welchem 100% Transitionsüberdeckung möglich sind. In dem Beispiel der Ablaufsteuerung ist diese minimale Tiefe gleich sechs, das entspricht dem Pfad von Zustand ControllerOn bis Zustand FlyTo. Damit ergibt sich für diesen Fall die Anzahl der Testfälle wie folgt:

$$N_\beta(1, 33, 6) = 33^6 \approx 1,291 \cdot 10^9. \tag{4.11}$$

Auch mit modernen Rechnern ist das Prüfen von $1,291 \cdot 10^9$ Testfällen, unter Berücksichtigung der benötigten Rechenzeit, sehr aufwändig. Durch das vorhandene Modellwissen lassen sich allerdings Optimierungen vornehmen, indem der kritische Pfad geteilt wird. Es wird dabei mit zwei Startzuständen gearbeitet: Zum Einen mit dem ursprünglichen Startzustand ControllerOn und zum Anderen mit dem Zustand MissionMode, welcher sich auf der halben Strecke des kritischen Pfades befindet. Das Bild 4.11 zeigt die beiden hervorgehobenen Zustände und den kritischen Pfad.

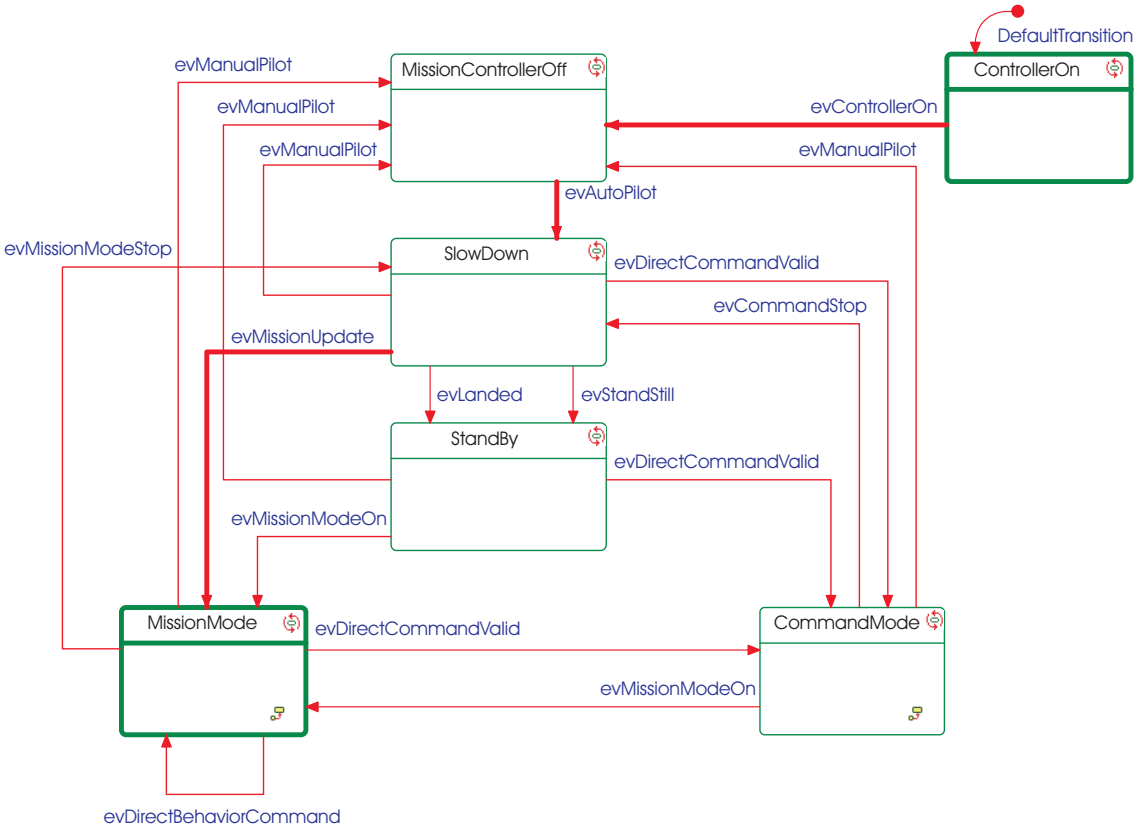

Bild 4.11: Kritischer Pfad der Ablaufsteuerung

Die Überlegung den kritischen Pfad zu teilen, reduziert die Anzahl der Testfälle:

$$N_\beta(2, 33, 3) = 2 \cdot 33^3 = 71874. \tag{4.12}$$

Dieser Wert ist in Bezug auf die bekannte Rechenzeit für die ersten Test Suite ausreichend. Eine letzte Optimierungsstufe ist dennoch der Einsatz von Benachrichtigungsketten der Länge 2 ausgehend von jedem Zustand des State Charts. Die Anzahl der Testfälle wird damit noch einmal reduziert:

$$N_\beta(21, 33, 2) = 21 \cdot 33^2 = 22869. \tag{4.13}$$

4.3.2 Test Key Generation

In dem Zustand „Test Key Generation“ wird der Testschlüssel für diese Test Suite generiert. Dieser Schlüssel besteht aus den zwei in Bild 4.12 dargestellten Abschnitten. Im ersten Abschnitt („State Key“) wird der Initialzustand für den Testfall gesetzt. Es ist möglich, einen, mehrere oder alle Zustände nacheinander als Initialzustand für die verschiedenen Testfälle in der Test Suite zu verwenden.

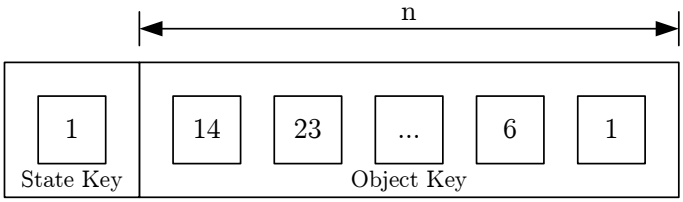

Bild 4.12: Testschlüssel in SCLT

Der zweite Schlüsselabschnitt („Object Key“) legt die Kombination der Benachrichtigungsobjekte für die später verwendete Benachrichtigungskette der Länge n fest. In diesem Abschnitt werden die Kombinationen nach jedem Testfall vom linken Wert ausgehend verändert. Nachfolgend ist der Ablauf in Tabelle 4.3 beispielhaft für eine Benachrichtigungskette mit vier Objekten abgebildet:

01-04	05-08	09-12	13-16
1 1	1 2	1 3	1 4
2 1	2 2	2 3	2 4
3 1	3 2	3 3	3 4
4 1	4 2	4 3	4 4

Tabelle 4.3: Beispiel für einen Testschlüssel mit $n = 2$ und $N_\varphi = 4$

Die Eigenschaft, dass die Änderungshäufigkeit sinkt, je weiter der Wert rechts im Schlüssel ist, wird bei dem Aufbau der Benachrichtigungskette berücksichtigt. Die Test Suite gilt als vollständig durchlaufen und wird beendet, wenn je nach Modus mindestens eine der nachfolgenden Bedingungen zutrifft:

- 100% Zustandsüberdeckung erreicht,
- 100% Transitionsüberdeckung absolviert,
- berechnete Anzahl der Testfälle N_β durchlaufen.

4.3.3 Build Test Case

In dem Zustand „Build Test Case“ wird der Testfall je nach generiertem Testschlüssel zusammengestellt. Man unterscheidet dabei zwei Abschnitte (Bild 4.13): „Load Event Queue“ erzeugt die benötigte Benachrichtigungskette und „Set Start State“ setzt den gewählten Startzustand in der Ablaufsteuerung.

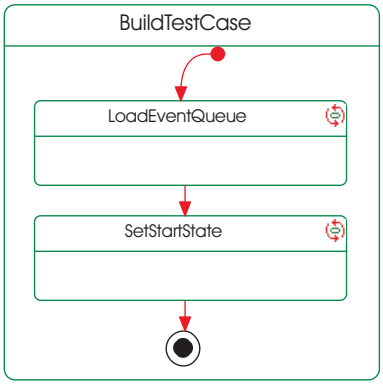

Bild 4.13: Zustand Build Test Case in SCLT

Die Benachrichtigungskette ist als Queue implementiert, es wird dabei das Template aus der STL (Standard Template Library) für C++ genutzt. Ausgehend von dem ersten Schlüsseleintrag wird für jeden Wert das entsprechende Objekt in die Benachrichtigungskette mit der Operation `push` geladen (Bild 4.14), entnommen wird es später in dem Zustand „Run Test Case“ mit der Operation `pop`.

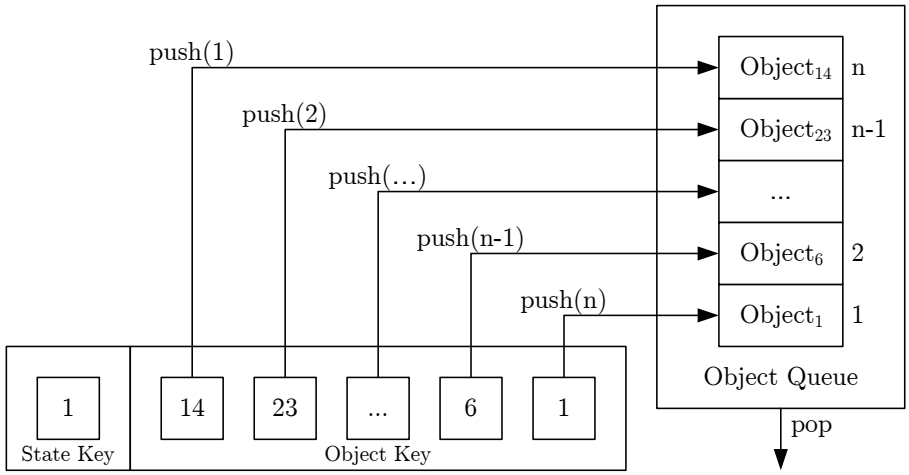

Bild 4.14: Generierung der Benachrichtigungskette

Zur Speicherung wird in der Queue das LIFO-Prinzip (Last In First Out) verwendet, welches ein Verfahren der Speicherung bezeichnet, bei dem diejenigen Elemente, die zuerst gespeichert wurden, auch zuerst wieder aus dem Speicher entnommen werden. Das gegenteilige Verfahren wird FIFO-Prinzip (First In First Out) genannt.

Der Grund für dieses Vorgehen basiert auf vorhandenem Modellwissen: Der letzte Schlüsseleintrag wird seltener geändert als alle anderen Einträge. Des Weiteren ist die Test Suite so modelliert, dass der Initialwert für diesen Schlüsseleintrag einer sinnvollen Benachrichtigung für den Initialzustand entspricht.

Im zweiten Abschnitt wird der Startzustand für den Testfall in der Ablaufsteuerung gesetzt. Die Idee dabei ist, dass diejenigen Pfade, die bereits als funktionierend erkannt wurden, genutzt werden. Wenn beispielsweise von dem Startzustand der Ablaufsteuerung (ControllerOn) ausgehend alle Pfade der Länge drei überprüft sind, kann der Initialwert in dem Testschlüssel für den Zustand bis zu dieser Entfernung verändert werden. In diesem Fall wird der Zustand MissionMode als nächster Startzustand betrachtet.

In dem CppUnit-Testframework werden für jeden Testfall die benötigten Objekte mit den Methoden (SetUp und TearDown) zu Beginn instanziiert und am Ende gelöscht. Dieser Vorgang benötigt pro Testfall nur eine geringfügige Zeitspanne (ca. $1 \cdot 10^{-5}$ Sekunden), die sich dennoch bei einer großen Anzahl von Testfällen bemerkbar macht. Aus diesem Grund wird für die Ablaufsteuerung eine Funktion aus dem OX-Framework genutzt, welche einen Neustart des State Charts vor dem Setzen des Anfangszustands ermöglicht. Die Testzeit wird somit weiter reduziert.

4.3.4 Run Test Case

In dem Zustand „Run Test Case“ wird der generierte Testfall ausgeführt. Zunächst werden der aktuelle Zustand und das erste Objekt bestimmt. Ist die Schleifendetektierung aktiviert, wird bei einer positiven Überprüfung die Benachrichtigungskette gelöscht und der Testfall beendet. Wird keine Schleife detektiert, findet ein Vergleich zwischen spezifiziertem und tatsächlichem Verhalten statt. Dazu wird für den aktuellen Zustand die Spezifikation eingelesen, das Objekt in der Ablaufsteuerung ausgelöst und aus der Benachrichtigungskette gelöscht. Der tatsächliche Folgezustand kann nun bestimmt und anschließend verglichen werden. Dieses Vorgehen wird solange wiederholt, bis die Benachrichtigungskette keine Objekte mehr enthält, erst dann ist der Testfall beendet. Der vollständige Ablauf ist in Bild 4.15 dargestellt.

Zur Detektierung einstufiger Schleifen[2] wird ein einfacher Index verwendet:

```
LoopIndex[CurrentState][CurrentObject].
```

Dabei wird das Prinzip der minimalen Schleifenüberdeckung berücksichtigt, da die Schleifen mindestens einmal durchlaufen werden müssen, um als Schleife erkannt zu werden. Spätere Benachrichtigungskette können dann über diesen Index auf einstufige Schleifen hin überprüft werden. Unnötiges Testen wird somit verhindert, wobei diese Schleifendetektierung auf mehrstufige Überprüfung ausgebaut werden sollte.

[2]Nach Auslösung eines Benachrichtigungsobjektes sind Anfangs- und Endzustand identisch.

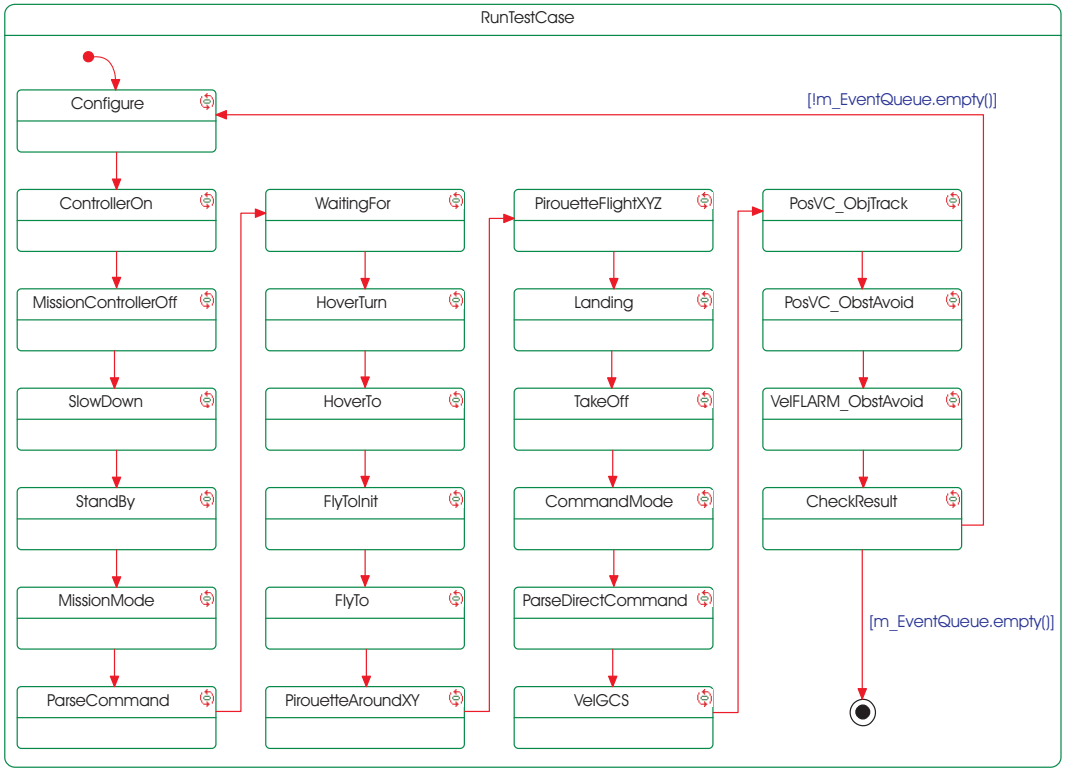

Bild 4.15: Zustand Run Test Case in SCLT

Die Spezifikation wird durch eine elementare Syntax formuliert, indem für den jeweiligen Zustand die Erwartungshaltung für eine Benachrichtigung eingetragen wird. Im Default-Fall werden für den Zustand unbekannte Benachrichtigungen als Schleife detektiert. Des Weiteren ist es sinnvoll, die jeweilige Spezifikation der Textausgabe anzuhängen, damit im Fehlerfall diese Informationen ausgelesen werden können. Nachfolgend ist in dem Listing 4.4 die Syntax für die Spezifikation abgebildet.

```
if( CurrentState == eState_1 )
{
    Message.append("eState_1");

    switch (CurrentObject)
    {
        case evObject1:
            Message.append("expected eState_2 next!\n");
            ExpectedState = eState_2;
            break;

        default:
            // Mark Loop
            LoopIndex[CurrentState][CurrentObject] = true;

            Message.append("expected eState_1 next!\n");
```

```
            ExpectedState = eState_1;
            break;
    }
}
```

Listing 4.4: Spezifikation in SCLT

4.3.5 Report

Falls in dieser Test Suite das CppUnit-Testframework einen Fehler während eines Testfalls registriert, wird der Fehlerreport ausgegeben. Dieser Report gliedert sich in drei Abschnitte, wie es in dem Konzept vorgesehen ist. Der erste Abschnitt benennt die generierte Benachrichtigungskette, der zweite Abschnitt erläutert den aktuellen Zustand und welcher Folgezustand für die Abarbeitung des Objektes erwartet wird. In dem dritten und letzten Abschnitt wird der tatsächliche Verlauf des State Chart ausgegeben.

In Bild 4.16 ist ein solcher Report für eine Benachrichtigungskette mit $n = 4$ beispielhaft dargestellt. In diesem speziellen Fall wird eine Benachrichtigungskette mit den Objekten evControllerOn, evAutoPilot, evStandStill und evControllerOn erzeugt. Die Ablaufsteuerung schaltet von dem Zustand ControllerOn ausgehend mit den Objekten bis zu dem Zustand StandBy. Diese Schaltung wird mit der entsprechenden Ausgabe in dem dritten Abschnitt nachvollzogen. Der Fehler wird durch die Ausgabe der Spezifikation sichtbar: Für den Zustand StandBy ist als Folgezustand MissionMode bei dem Objekt evControllerOn spezifiziert, dies löst den Fehler aus. Die Spezifikation ist in diesem Beispiel bewusst falsch gesetzt, um diesen Fehlerreport zu erzwingen.

```
*******************************************
Object Sequence
*******************************************

evControllerOn
evAutoPilot
evStandStill
evControllerOn

*******************************************
State Sequence
*******************************************

eStateControllerOn expected eStateMissionControllerOff next!
eStateMissionControllerOff expected eStateSlowDown next!
eStateSlowDown expected eStateStandBy next!
eStateStandBy expected eStateMissionMode next!
```

```
*****************************************
Sequence Control Output
*****************************************

SC: MissionControllerOff: Entry
SC: MissionControllerOff: Exit.
SC: SlowDown: Entry
SC: SlowDown: Exit.
SC: StandBy: Entry
```

Bild 4.16: Fehlerreport in SCLT

Mit dieser Test Suite werden somit jegliche Differenzen zwischen Spezifikation und Modell in Bezug auf die Schaltung der Transitionen aufgezeigt.

4.3.6 Ergebnisse

In Bild 4.17 ist der Zusammenhang zwischen der Anzahl der Testfälle und der damit verbundene Überdeckungsgrad für eine Test Suite mit einem Initialzustand und einer Benachrichtigungskette mit $n = 6$ dargestellt. Es lassen sich anhand des Diagramms verschiedene Aspekt ableiten: Zum Einen werden circa 0,1% der berechneten Testfälle ($1,291 \cdot 10^9$) benötigt, um 100% Überdeckungsgrad zu erreichen. Die Integration des Modellwissens in den Testschlüssel spiegelt sich an dieser Stelle wieder.

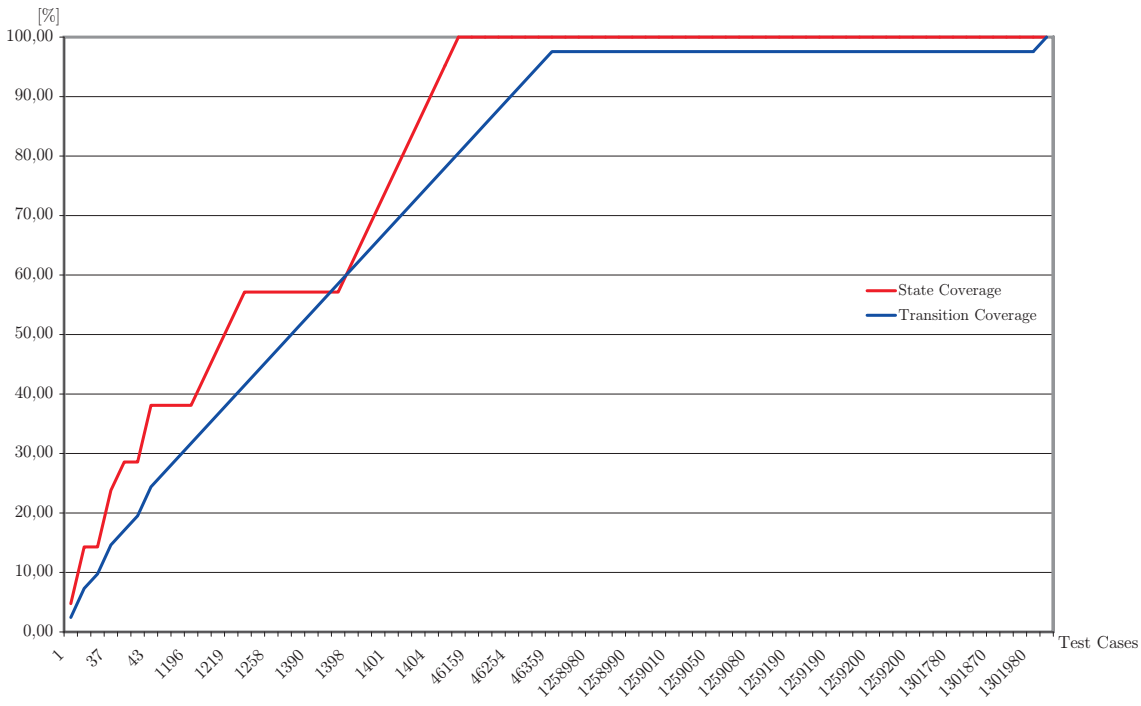

Bild 4.17: Sechsstufige Benachrichtigungskette von einem Initialzustand

Zum Anderen lässt sich aus der Zahl der tatsächlich benötigten Testfälle[3] (38) ableiten, dass die McCabe-Metrik mit einem Wert von 39 auch ohne Endzustand des State Charts für diese Test Suite stimmt. Nachteilig sind für diese Test Suite trotz allem zwei Faktoren. Einerseits werden circa 120 Sekunden Rechenzeit benötigt und andererseits ist die Distanz zwischen dem letzten und dem vorletzten benötigtem Testfall (circa $1,25 \cdot 10^6$ Testfälle) sehr groß.

Das nächste Diagramm (Bild 4.18) zeigt den Verlauf für dieselbe Test Suite mit dem Unterschied, dass in diesem Fall einstufige Schleifen erkannt und nicht verarbeitet werden. Die Rechenzeit beträgt 69 Sekunden bei 39 benötigten Testfällen. Es lässt sich weiter ableiten, dass die Benachrichtigungsketten aus dem vorherigen Beispiel, die zu einer schnellen Erhöhung des Überdeckungsgrads führten, einstufige Schleifen beinhalten und daher gelöscht werden. Die benötigten Testfälle beginnen erst nach über einer Million Durchläufen und enden bei dem gleichen Testfall wie in dem ersten Beispiel, da der entscheidende Pfad nur einmal in dem Testschlüssel vorkommt.

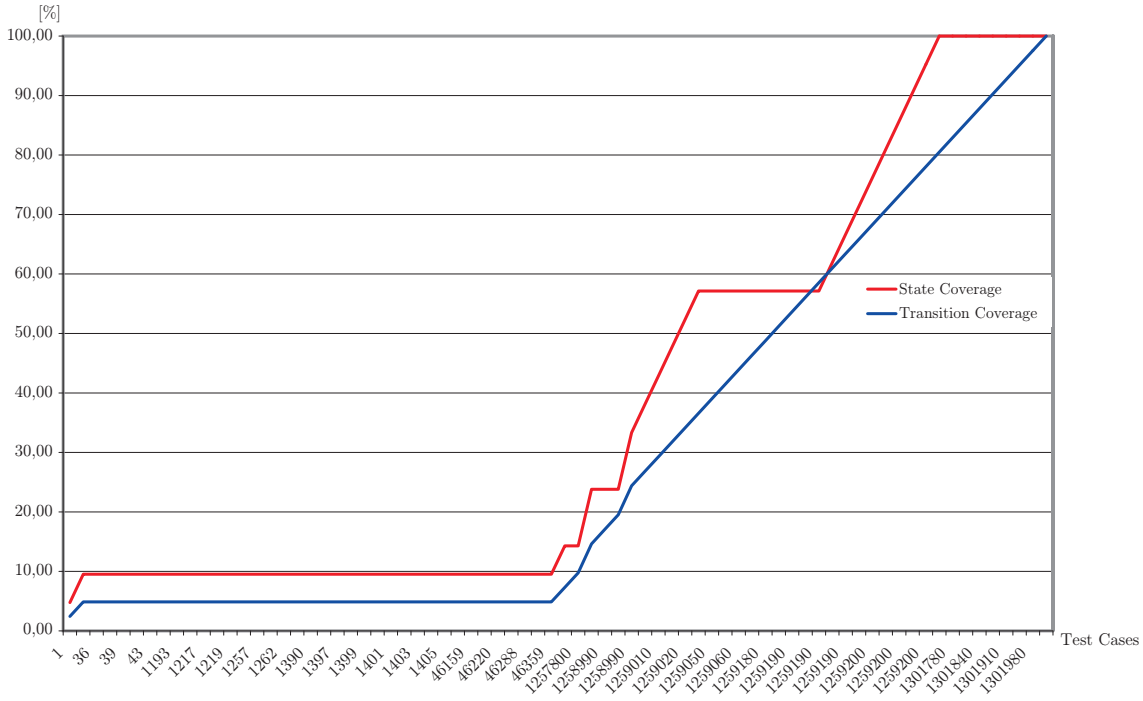

Bild 4.18: Detektierung von einstufigen Schleifen

Das letzte Diagramm (Bild 4.19) zeigt die in der Analyse angesprochene Teilung der Benachrichtigungskette. Es wird nacheinander von zwei Initialzuständen (Controller-On, MissionMode) ausgehend eine Benachrichtigungskette mit $n = 3$ abgearbeitet, wobei einstufige Schleifen zusätzlich detektiert werden. Insgesamt benötigt diese Test Suite 38 Testfälle bei einer Rechenzeit von circa zwei Sekunden. Die Transitionsüberdeckung erreicht im Gegensatz zu den vorherigen Beispiel merklich schneller ihren Zielwert.

[3] Jene Testfälle, die eine Erhöhung der Zustands- bzw. der Transitionsüberdeckung bewirken.

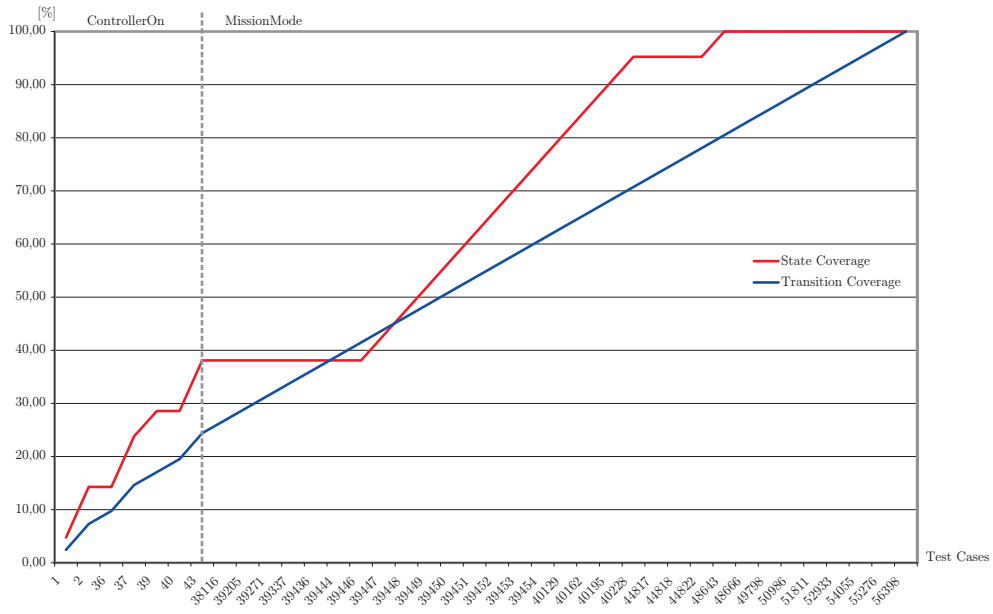

Bild 4.19: Teilung der Ereigniskette auf zwei Initalzustände

Aus der durchlaufenen Anzahl von Testfällen $\widetilde{N}_\beta \subseteq N_\beta$ aus Bild 4.17 und deren Durchlaufzeit t lässt sich nun die benötigte Zeit für einen Testfall ohne Schleifendetektierung errechnen. In diesem Fall ergibt sich:

$$\frac{t}{\widetilde{N}_\beta(1,33,6)} = \frac{120 \text{ s}}{1.522.080} = 7,88 \cdot 10^{-5} \text{ s}. \tag{4.14}$$

Es wird nun versucht, die Durchlaufzeit für einen Testfall auf dem Bordrechner (1400 MHz) in Beziehung zu dem Testcomputer (3400 MHz) zu setzen.

$$7,88 \cdot 10^{-5} \text{ s} \cdot \frac{3,4 \cdot 10^9 \frac{1}{s}}{1,4 \cdot 10^9 \frac{1}{s}} = 1,92 \cdot 10^{-4} \text{ s} \tag{4.15}$$

Dieser Wert dient als grobe Abschätzung, da das Betriebssystem interne Operation, wie die des Taskmanagers, parallel weiterführt und der Testfall durch die Generierung des Testschlüssels einen leichten Overhead besitzt. Für das System im Flugbetrieb lässt sich somit die Ausführungszeit für die logische Schaltung des State Charts der Ablaufsteuerung mit einem Wert, der kleiner ist als $1,92 \cdot 10^{-4}$ Sekunden, abschätzen.

4.4 Die Mustermission

Der Ansatz der Mustermission ist, mit einer möglichst kurzen Testsequenz gleichzeitig funktionale und strukturelle Tests während einer SITL- oder HITL-Simulation durchzuführen. Als Kriterium wird dabei die Zustands- bzw. die Transitionsüberdeckung verwendet, welche durch einen hohen Überdeckungsgrad besonders aussagekräftig sein soll. Das Sequenzdiagramm aus Bild 4.20 zeigt den vorgeschlagenen Ablauf schematisch. Es wird zunächst die Initialisierung des State Charts geprüft, um anschließend den Mission- und Kommandomodus zu testen. Abschließend findet eine Überprüfung des Wechsels zwischen manuellem und automatischem Flug statt.

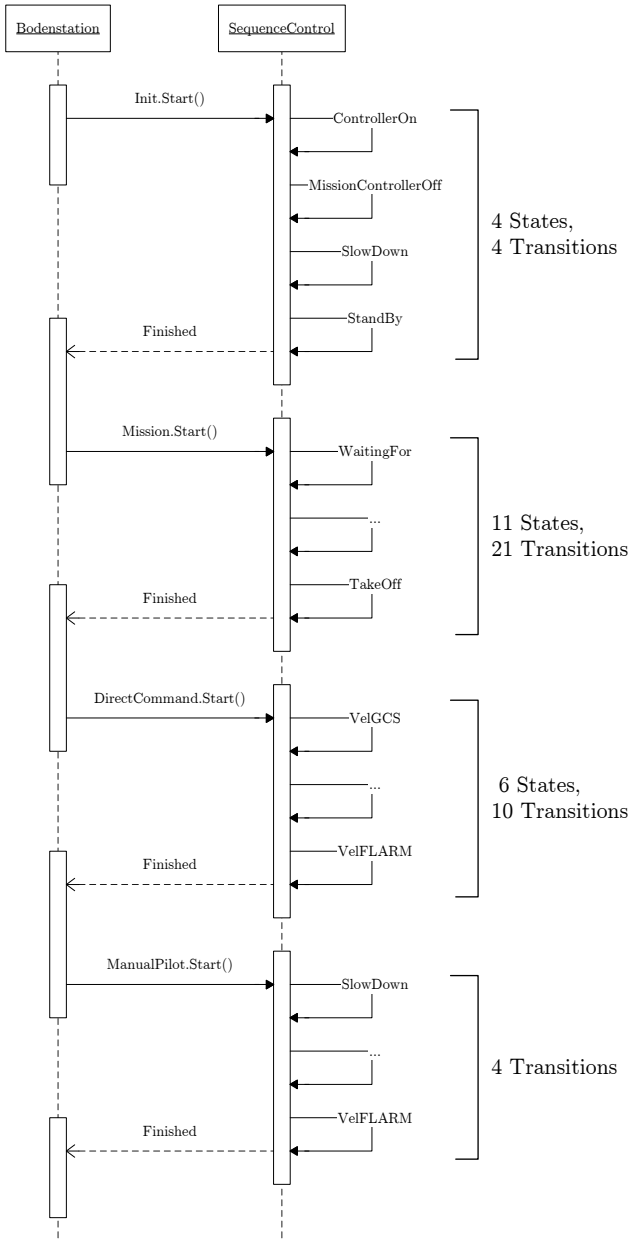

Bild 4.20: Sequenzdiagramm der Mustermission

In dem Bild 4.20 wird zusätzlich dargestellt, wie viele Zustände und Transitionen in jedem Schritt durchlaufen werden. Während der Simulation kann der Report jederzeit in MATLAB ausgegeben werden, um die bisherige Aussagekraft der Mustermission zu ermitteln.

Der Missionsmodus bietet acht verschiedene Basisverhalten für den Hubschrauber an, welche einzeln überprüft werden müssen. Das Sequenzdiagramm aus Bild 4.21 zeigt den dafür notwendigen Ablauf schematisch. Es wird dazu eine Wegpunktliste erstellt, welche von dem Zustand GetCommand eingelesen wird. In dieser Liste wird zunächst begonnen, den Hubschrauber ein Rechteck vom Boden startend mit dem Zustand FlyTo abfliegen zu lassen. Die dafür benötigte Trajektorie wird in dem Zustand FlyToInit erstellt. Dieses Rechteck muss möglichst groß sein, damit die Flugzeit ausreichend ist, auch Unterbrechungen stattfinden zu lassen.

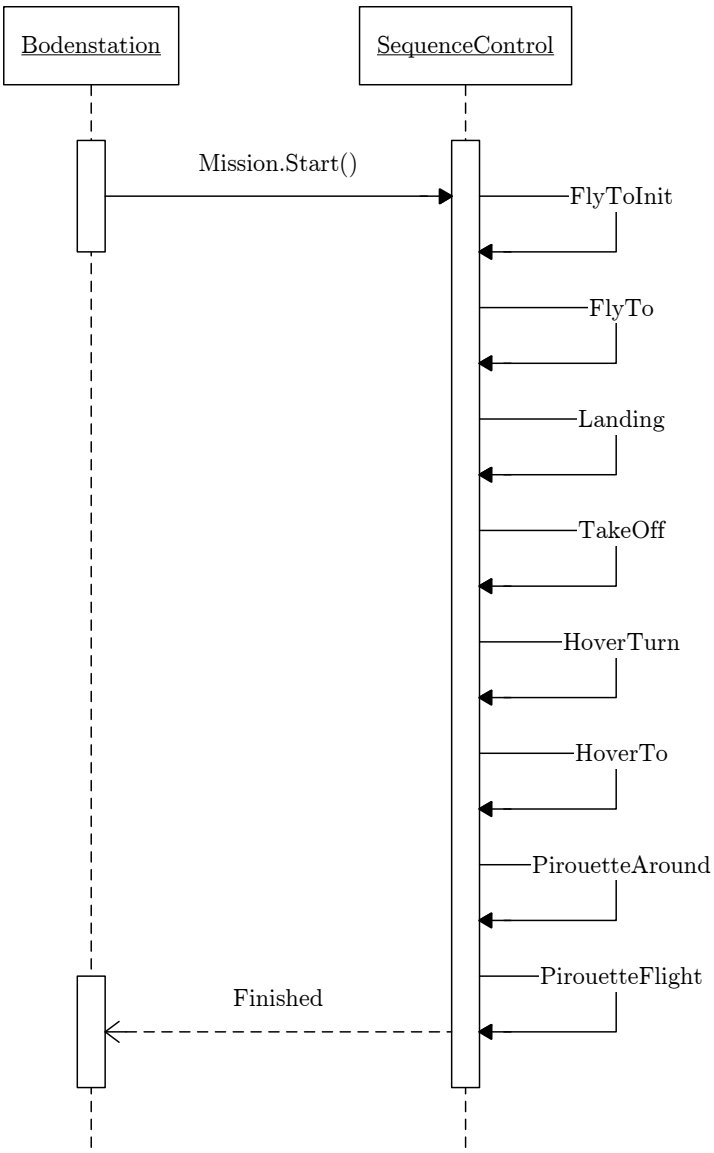

Bild 4.21: Sequenzdiagramm des Missionsmodus

Das Landen und das erneute Abheben wird überprüft, wenn der Hubschrauber die Trajektorie abgeflogen ist. Anschließend wird der Zustand HoverTurn getestet, in dem der Hubschrauber sich mehrmals im Uhrzeigersinn um 90° dreht. Mit dem Zustand HoverTo wird folglich das Schweben zu einer bestimmten Position mit verschieden Parametern mehrmals simuliert.

Abschließend werden die beiden Pirouettenfunktionen untersucht. Die Drehpunkte sollten nicht auf die x- bzw. die y-Achse im Koordinatensystem gelegt werden, damit die Pirouette möglichst umfangreich getestet werden kann (Bild 4.22). Es wird in diesem Beispiel eine Pirouette um den Punkt (-20; 20), ausgehend von dem Startpunkt (-90,71; -50,71) und der Anweisung 180° zu durchlaufen, geflogen. Zur Überprüfung sollten im Voraus Zwischen- und Endwerte (50,71; 90,71; -135°) berechnet werden.

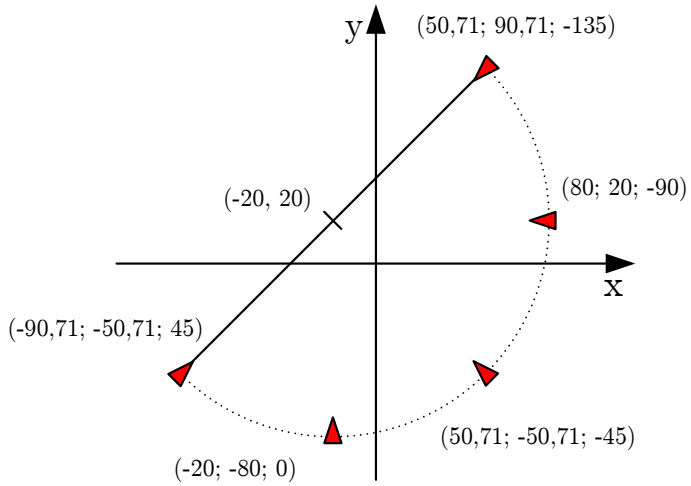

Bild 4.22: Pirouette in der Mustermission

Jedes Basisverhalten wird bei der Versuchsdurchführung mit einem WaitingFor verknüpft, damit auch dieser Zustand geprüft wird und es möglich ist, die jeweiligen Positions- und Ausrichtungswerte, zum Beispiel während einer Pirouette, abzulesen.

Um den Direktkommandomodus zu überprüfen, wird das Vorgehen des Sequenzdiagramms aus Bild 4.23 vorgeschlagen. Dabei wird zunächst der Missionsmodus gestartet, um die Transition zwischen Missionmodus und Direktkommandosmodus ausführen zu können. Befindet sich der State Chart im Zustand FlyTo findet die erste Unterbrechung statt. Es wird durch vordefinierte UDP-Pakete ein Direktkommando von FLARM zur Hindernisvermeidung simuliert. Ist das Direktkommando beendet, wird die Mission im Zustand FlyTo fortgesetzt. An dieser Stelle findet die nächste simulierte Unterbrechung durch ein Direktkommando des Bildverarbeitungsrechners (VC) statt. Auch dieser Befehl dient der Hindernisvermeidung und wird ebenfalls durch ein vordefiniertes UDP-Paket realisiert. Ist dieses Direktkommando abgearbeitet, wird der Missionmodus generell gestoppt, so dass der State Chart zunächst in den Zustand SlowDown schaltet und anschließend in den Zustand StandBy.

Ist der State Chart in dem Zustand StandBy angelangt, wird von der Bodenstation ein Geschwindigkeitskommando gesendet. Das Kommando muss in diesem Fall nicht unbedingt simuliert werden, da es möglich ist, auch der Joystick der Bodenstation einzusetzen.

Es ist jedoch zu beachten, dass das Kommando möglichst hohe Geschwindigkeiten beinhaltet, um anschließend möglichst lange im Zustand SlowDown, welcher den Modellhubschrauber wieder abbremst, verharren zu können. Wird dies umgesetzt, kann das Direktkommando des Bildverarbeitungsrechners zur Objektverfolgung mit einem vordefinierten UDP-Paket simuliert werden.

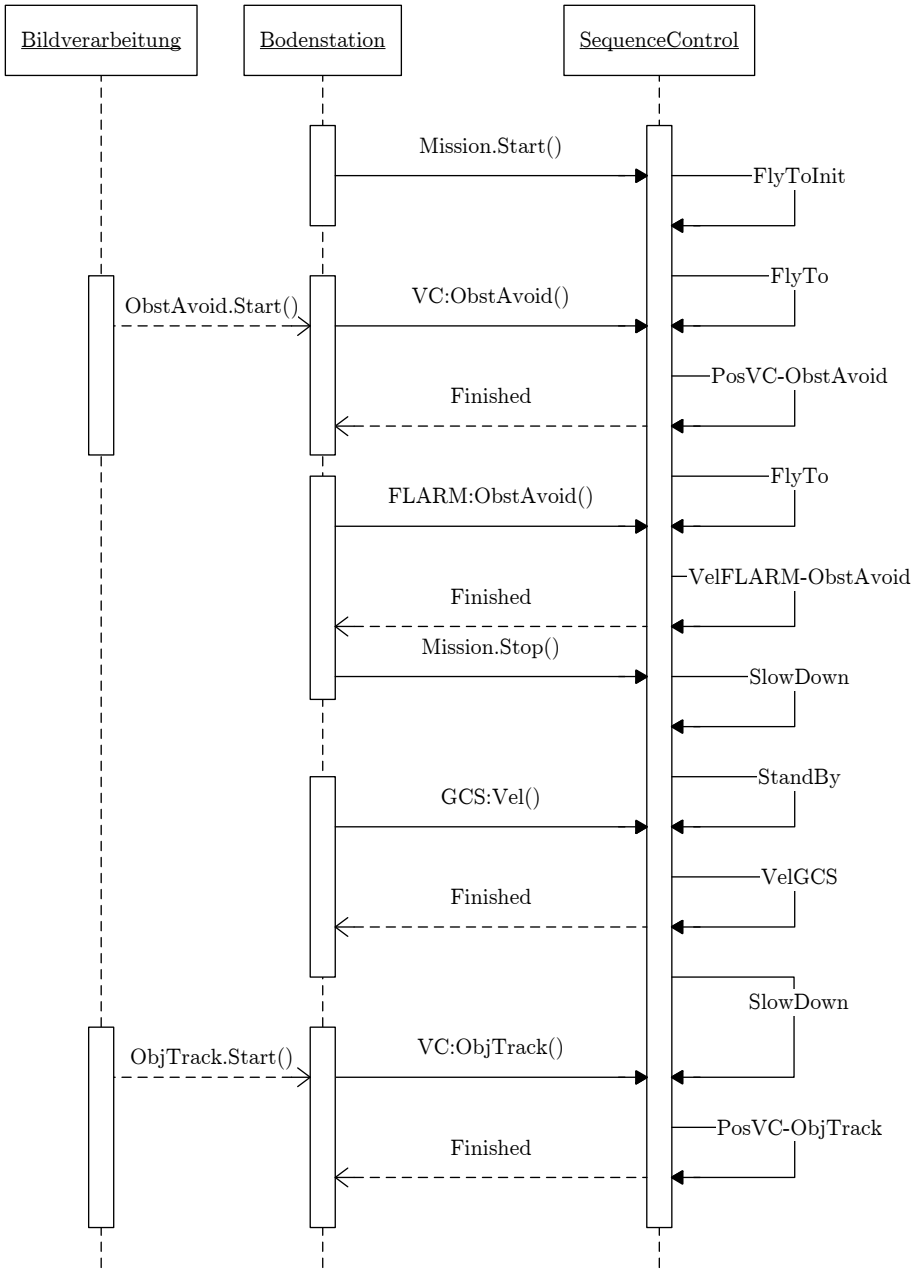

Bild 4.23: Sequenzdiagramm des Kommandomodus

Wenn das Sequenzdiagramm aus Bild 4.23 abgearbeitet ist, muss das Umschalten zwischen manueller und automatischer Steuerung durch den Sicherheitspiloten abschließend geprüft werden. Dazu wird wie folgt vorgegangen:

1) Zunächst wird der Missionsmodus gestartet und durch den Sicherheitspiloten unterbrochen.

2) Der Kommandomodus wird durch das Direktkommando VelGCS aufgerufen und durch den Sicherheitspiloten wieder gestoppt.

3) Anschließend wird der Kommandomodus erneut ausgeführt und mit einer hohen Geschwindigkeit verlassen, so dass der State Chart lange im Zustand Slow-Down verweilt. An dieser Stelle greift der Sicherheitspilot erneut ein.

4) Aus dem Zustand StandBy findet abschließend die letzte Unterbrechung durch den Sicherheitspiloten statt.

Dieses Vorgehen kann auf zwei unterschiedlichen Wegen realisiert werden: Zum Einen ist es möglich, die tatsächliche Funksteuerung zu nutzen und zum Anderen kann dieser Vorgang auch durch ein vordefiniertes UDP-Paket simuliert werden.

Zusammenfassend lässt sich die Mustermission wie folgt beschreiben: Es werden von allen Zuständen die entry-, exit- und do-Aktivitäten ausgeführt und können somit auf ihre Funktionsweise überprüft werden. Des Weiteren werden alle Transitionen aktiviert, somit erreicht diese Mustermission eine Zustands- und Transitionsüberdeckung von 100%. Der jeweilige Fortschritt kann dem Report in der entsprechenden Systemsimulation entnommen werden und bietet einen Ansatz für weitere Analysen, falls die 100% Überdeckungsgrad nicht erreicht werden.

Kapitel 5

Diskussion der Ergebnisse

Bereits während des Re-Engineerings wurden zwei Fehler entdeckt und korrigiert. Der erste Fehler befand sich in der Event Middleware (Bild 4.1) und war durch die Überlagerung verschiedener Eingabewerte mit unterschiedlicher Intention in einem Zwischenspeicher (`vdTarget`) gekennzeichnet. In seltenen Fälle konnte dieser Fehler zu einem Systemabsturz führen, was sich in der Test Suite der Event Middleware zeigte. Korrigiert wurde das Fehlverhalten durch die Einführung eines konstanten Index, welcher insgesamt 24 Werte umfasst. Dazu gehören beispielsweise die Geschwindigkeiten u, v, w an den Positionen eins bis drei oder die Koordinaten x, y, z an den Positionen fünf bis sieben.

Die zweite Unstimmigkeit, die schon während des Re-Engineerings entdeckt wurde, ist die unterschiedliche Überprüfung der Sicherheitshöhe bei einem Direktkommando. Diese Sicherheitsüberprüfung ist nun fester Bestandteil des Systems und verhindert somit risikoreiche Kommandos in Bodennähe von der Bodenstation (zum Beispiel über Joystick), von FLARM oder von dem Bildverarbeitungsrechner zur Objekt-Verfolgung oder zur Kollisionsvermeidung.

Die Test Suite für die logische Struktur deckte ebenfalls zwei Fehler in der Ablaufsteuerung auf. Zum Einen war die neu-eingeführte Methode `GetActiveState()` fehlerhaft, da die Zustandstransformation der Rhapsody Zuständen auf das UML State Chart Template unvollständig war. Die Ursache dafür ist, dass in einem hierarchischen State Chart von Rhapsody mehrere Zustände gleichzeitig aktiv sein können. Das folgende Beispiel soll dies näher erklären: Während der Initialisierung werden alle Default-Transitionen aktiviert, d.h. dass in der Ablaufsteuerung gleichzeitig die Zustände ControllerOn, GetCommand und GetDirectCommand aktiv sind, obwohl das formal falsch ist. Daher müssen zusätzlich die zusammengesetzten Zustände MissionMode und CommandMode überprüft werden, welche bei der Initialisierung inaktiv sind. Somit ist der einzige gültige Zustand ControllerOn. Dieser Umstand wird folglich bei der Methode `GetActiveState()` berücksichtigt.

Weiterhin empfiehlt es sich, die ID's der Benachrichtigungsobjekte zu überprüfen, wenn diese per Code-Generierung durch Rhapsody automatisch und nicht manuell erstellt werden. Bei der Test Suite für die logische Struktur werden einstufige Schleifen via Index (Zustand, Benachrichtigungsobjekt) registriert. Dieser Index muss somit dem ID-Intervall (minimaler bis maximaler Wert) der Benachrichtigungsobjekte angepasst werden, damit keine Speicherzugriffsfehler entstehen können. Eventuell ist sogar eine dynamische Lösung in diesem Fall von Vorteil. Zusätzlich wird empfohlen die Detektierung von Endlosschleifen im Flugbetrieb zu deaktivieren, da diese sofort die Ausführung der Anwendung bei einer Detektierung beendet. Während des Testens oder der Simulationen ist es allerdings sinnvoll, diese Funktionalität zu nutzen.

Es wurde gezeigt, dass der Testaufwand mit der Länge der Benachrichtigungsketten in den Test Suiten stark ansteigt. Auf den ersten Blick ist es eher sinnvoll, kurze Sequenzen von vielen Initialzuständen zu testen. Im Fall der Test Suite für die logische Struktur heißt das, von jedem Zustand eine Benachrichtigungskette mit zwei Objekten abzuarbeiten. Mit der Fallstudie [2] wird jedoch gezeigt, dass die meisten Fehler zwar mit kurzen Sequenzen erkannt werden, aber dass längere Sequenzen auch weitere Fehler finden, wobei die Effizienz abnimmt. Getestet wurde die modellierte Ablaufsteuerung des Programms RealJukebox (RJB) von der Firma RealNetworks (Tabelle 5.1). Mit zweistufige Testfällen wurden insgesamt 44 Fehler festgestellt. Drei- und vierstufige Testfälle fanden jeweils 12 weitere Fehler. Das Verhältnis zwischen Testaufwand und gefundene Fehler sinkt jedoch stark.

Länge der Ereigniskette	Entdeckte Fehler	Effizienz pro Testfall
2	44	$4,8 \cdot 10^{-2}$
3	56	$2,3 \cdot 10^{-2}$
4	68	$0,9 \cdot 10^{-2}$

Tabelle 5.1: Fallstudie für Ereignisketten am Beispiel von RJB

Es empfiehlt sich daher, für die ersten Modultests kurze Benachrichtigungsketten einzusetzen. Vor einem Flugtest sollten jedoch auch längere Benachrichtigungsketten überprüft werden.

Kapitel 6

Zusammenfassung und Ausblick

Die modellbasierte Software-Entwicklung wurde durch das Re-Engineering der Ablaufsteuerung mit dem Werkzeug Rhapsody beispielhaft vorgestellt und kann auf andere Komponenten (z.B. den „Supervisor“) des ARTIS-Systems übertragen werden. Der allgemeine Lösungsansatz der modellbasierten Software-Entwicklung wurde somit einem speziellen Bereich der UAV-Forschungsrichtung angepasst und lässt sich auf weitere ereignisbasierte Software-Systeme angleichen.

Der zunächst wichtigste Vorteil ist, dass das Modell und der Quellcode miteinander verbunden sind. In diesem Zusammenhang ist es prinzipiell möglich, den Quellcode aus dem Modell zu generieren bzw. in einem gewissen Umfang via Roundtrip-Verfahren das Modell mit manuell geändertem Quellcode abzugleichen. Strukturänderungen können in dem Modell fortan mit hohem Komfort grafisch durchgeführt werden und müssen nicht mehr manuell in dem Quellcode nachgetragen werden, wenn die Möglichkeit der Code-Generierung eingesetzt wird.

Mit Rhapsody ist ein Werkzeug eingeführt, welches als quasi Industriestandard gilt und von kooperierenden Unternehmen wie Airbus genutzt wird. Das zu Rhapsody dazugehörige OX-Framework bietet umfangreiche Funktionen, die bei einer konsequenten Nutzung weitere Vorteile bieten. Grundlegend kann nun eine grafische Modellierung durchgeführt werden, welche mit strukturellen Überprüfungen (z.B. fehlerhafte Transitionen bei State Charts) die Modellierungsphase verfeinert.

Die Code-Generierung ist eine wesentliche Funktion von Rhapsody und lässt sich auf die speziellen Anforderungen skalieren. Dies umfasst die Transformation in mehrere Sprachen (C/C++/Java) unter Verwendung verschiedener Methodiken (flache bzw. wiederverwendbare State Charts). Die Integration von Matlab/Simulink-Modellen ist zudem auch durchführbar und erweitert somit den ersten Ansatz für die modellbasierte Software-Entwicklung aus [20]. Rhapsody ist damit eine Alternative zu dem Werkzeug Stateflow (Mathworks), welches nur zur Modellierung von State Charts gedacht ist und keine weiteren Modellierungsfunktionen (z.B. Sequenzdiagramme) unterstützt.

Durch das Re-Engineering ist es möglich, während der Systemsimulation Zustands- und Transitionsüberdeckungen abzufragen, um eine Aussage über die Vollständigkeit des Versuchsdurchlaufs zu treffen. Die Idee der Mustermission, also eine möglichst kurze Versuchsdurchführung mit einem aussagekräftigem Überdeckungsgrad, wurde in dieser Arbeit vorgestellt und dient in diesem Zusammenhang der Unterstützung. Verfeinert wird die Systemsimulation durch die Debug-Nachrichten, welche den internen Verlauf in dem State Chart exakt wiedergeben. Diese sind für jede modellierte Klasse ausgearbeitet und gruppiert, sodass in jeder Klasse festgelegte Verbose-Level der einzelnen Ausgaben bestehen. Diese Verbose-Level sind in dem Gesamtrahmen des Missionsmanagers eingeordnet und leicht veränderbar.

Mit den beiden vorgestellten Test Suiten auf Basis eines Testautomaten können sehr ausführliche Tests durchgeführt werden, deren Ergebnisse in einen Benchmark-Vergleich münden können. Dazu zählen zum Beispiel beliebig lange und beliebig viele Benachrichtigungsketten, unterschiedlich kombinierte Eingabegrößen sowie verschiedene Konfigurationen des State Charts. Weiterhin beinhalten die beiden Test Suiten ein dreiteiliges Report-Konzept, welches deutlich den Testfall, den Fehler und die Situation in der zu testenden Komponente beschreiben. Ein solcher Report kann im Fehlerfall in das Mantis-System (Bugtracker) eingestellt und damit dem verantwortlichen Entwickler übermittelt werden. Eine zukünftige Erweiterung wäre die Umstellung auf einen XML-basierten Report, um zum Beispiel die Suchfunktionen in dem Report zu verbessern.

Neben den genannte Vorteilen, zeigen sich allerdings auch Nachteile auf: Es besteht nun eine Grundabhängigkeit von dem Werkzeug Rhapsody, da keine Export-Möglichkeiten des UML-Modells in ein anderes Dateiformat gegeben sind. Die Abhängigkeit erweitert sich durch die Möglichkeit, bei der Code-Generierung das OX-Framework zu nutzen. Tritt der Fall ein, dass die Ausführungszeit des OX-Frameworks nicht befriedigend ist oder dass das OX-Framework andere Fehler bei der Nutzung aufweist, besteht jedoch die Möglichkeit, ein anderes (eventuell freies) Framework einzubinden, da die Code-Transformation mit Rhapsody in diesem Zusammenhang sehr gut skalierbar ist. Es muss für diese Umstellung jedoch der zeitliche Aufwand für die Analyse und die Umstellung berücksichtigt werden.

Es stellt sich allgemein die Frage, ob sich die Investition in Rhapsody im Rahmen der ARTIS-Plattform amortisiert. Es muss daher genau analysiert werden, welche Kosten den Vorteilen der Nutzung gegenüber stehen. Des Weiteren ist zu untersuchen, welche Funktionen von Rhapsody generell genutzt werden. Andererseits sind auch freie alternative Lösungen (zum Beispiel Fujaba) zu betrachten, wenn für diese zertifizierte Code-Generatoren vorliegen.

Wird Rhapsody und das damit verbundene OX-Framework weiter verwendet, empfiehlt es sich, die Event Middleware zu überarbeiten. In dieser Arbeit lag der Fokus auf einer möglichst geringen Ausführungszeit, sodass die Implementierung statisch erfolgte. Nachteilig ist jedoch, dass zur Laufzeit keine Veränderungen vorgenommen werden können und dies dem Prinzip der dynamischen Ereignisanforderung wider-

spricht. Wenn beispielsweise der Bildverarbeitungsrechner in dem Flugversuch nicht eingesetzt wird, ist es unnötig, entsprechende Direktkommandos statisch detektieren zu wollen.

Die Event Middleware kann daher versuchsweise erweitert werden, indem für die einzelnen Zustände des State Charts (Empfänger E) zum Beispiel über eine XML-Bibliothek spezifiziert wird, welche Benachrichtigungsobjekte benötigt werden und wie sich diese zusammensetzen. In diesem Fall ist es sinnvoll, mehrere Gruppen und einen Binder B (Bodenstation) einzusetzen (Bild 6.1), um während eines Flugversuchs die Ereignisbenachrichtigung des Filters (F) vollkommen neu zu konfigurieren.

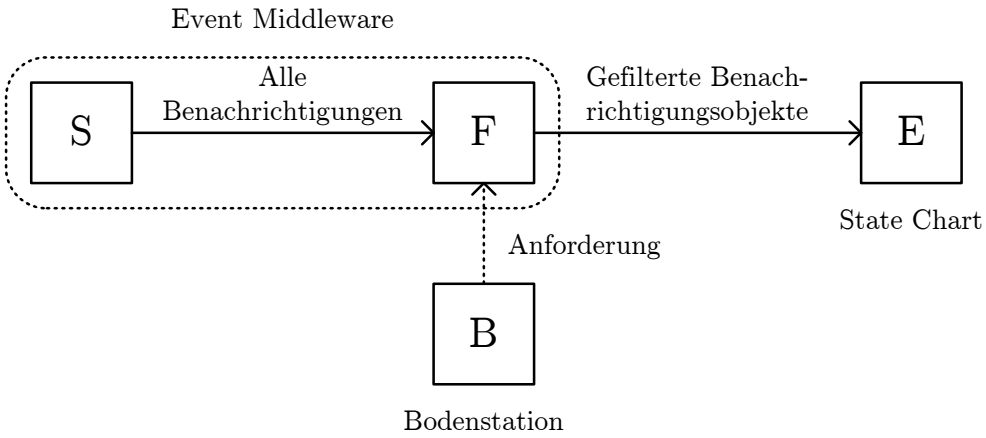

Bild 6.1: Dynamischer Ansatz der Event Middleware

Wie sich dieser dynamische Prozess auf die Laufzeit auswirkt, bleibt zu untersuchen.

Literaturverzeichnis

[1] J. Dittrich. *Das VTOL-UAV ARTIS: Konzepte und Nutzung.* CCG-Seminar UAV-Führungssysteme, ARTIS Projekt, Abteilung Systemautomation, Institut für Flugsystemtechnik, Deutsches Zentrum für Luft- und Raumfahrt e.V., 2006.

[2] C. J. Budnik und L. White F. Belli. *Event-based modelling, analysis and testing of user interactions: approach and case study.* Paper, Software testing, verification and reliability, Wiley InterScience, 2006.

[3] T. Faison. *Event-Based Programming: Taking Events to the Limit.* 1st Edition, Apress, Berkely, CA, 2006.

[4] N. Fenton and M. Neil. *Software Metrics and Risk.* Paper, FESMA 99, 2nd European Software Measurement Conference, 1999.

[5] H.-G. Gräbe. *Software-Qualitätsmanagement.* Vorlesungsskript, Forschungsgruppe Betriebliche Informationssysteme, Institut für Informatik, Universität Leipzig, 2006.

[6] David Harel. *On Visual Formalisms.* Communications of the ACM, Vol. 31, No. 5, 1988.

[7] I-Logix. *The OXF Model.* Technical Manual, 2005.

[8] I-Logix. *Rhapsody in C++.* Technical Manual, 2006.

[9] H. Kniberg. *Scrum and XP from the Trenches.* Technical Manual, Crisp AB, Stockholm, 2006.

[10] Th. J. McCabe. *A Software Complexity Measure.* IEEE Trans. Software Engineering SE-2(4), 308-320, 1976.

[11] C. Mertens. *Metriken im Qualitätsmanagement.* Seminararbeit Qualitätsmanagement in der Softwaretechnik, Institut für Wirtschaftsinformatik, Universität Münster, 2004.

[12] A. Russo; R. Miller; B. Nuseibeh and J. Kramer. *An Abductive Approach for Analysing Event-Based Requirements Specifications.* Paper, University College London, ICLP-2002, SpringerVerlag, 2002.

[13] OASIS. *Event-driven Testing Model and Language (eTest ML).* Technical Manual, Working Draft, OASIS Open Committees, 2006.

[14] Redhat. *POSIX Threads for Windows API Reference W32 2.8.0.* Manual, http://sources.redhat.com/pthreads-win32, Dezember 2006.

[15] C. Robinson-Mallett. *Modellbasierte Modulprüfung für die Entwicklung technischer, softwareintensiver Systeme mit Real-Time Object-Oriented Modeling.* Dissertation, Mathematisch-Naturwissenschaftlichen Fakultät der Universität Potsdam, 2005.

[16] B. Rumpe. *Extreme Programming - Back to Basics?* Paper, Technische Universität München, Modellierung 2001 - Workshop der Gesellschaft für Informatik e.V., 2001.

[17] B. Rumpe. *Agile Modellierung mit UML.* Xpert.press, Springer Verlag, 2005.

[18] C. Rupp. *Requirements-Engineering und -Management: Professionelle, iterative Anforderungsanalyse für die Praxis.* Hanser Fachbuchverlag, 4. Auflage, 2006.

[19] K. Schwaber. *Advanced Development Methods: SCRUM Development Process.* Paper, 2006.

[20] R. Schwarzer. *Prozesskette zur automatischen Generierung von echtzeitfähiger Software.* Diplomarbeit, Abteilung für Mathematische Verfahren und Datentechnik, Institut für Flugsystemtechnik, Deutsches Zentrum für Luft- und Raumfahrt e.V., September 2004.

[21] Rapide Design Team. *Rapide: Executable architecture definition language (EADL).* Technical Manual, Standford University, Program Analysis and Verification Group, Stanford Rapide Project, 1998.

[22] R. Petrasch und D. Macos. *MDA für Embedded Systems: Pros und Cons - Eine Anregung zur Diskussion.* Modellbasierte Software-Entwicklung für eingebettete Systeme, 2. Workshop der Special Interest Group, Model-Driven Software-Engineering (SIG-MDSE), Juni 2007.

[23] F. M. Adolf und F. Thielecke. *A Sequence Control System for Onboard Mission Management of an Unmanned Helicopter.* Paper No. AIAA-2007-2769, ARTIS Projekt, Abteilung Systemautomation, Institut für Flugsystemtechnik, Deutsches Zentrum für Luft- und Raumfahrt e.V., 2006.

[24] E. Dustin; J. Rashka und J. Paul. *Software automatisch testen.* Xpert.press, Springer Verlag, 2001.

[25] L. Rising und N. S. Janoff. *The Scrum Software Development Process for Small Teams.* Paper, IEEE Software Volume 17, Number 4, 2000.

[26] F. Dabek; N. Zeldovich; F. Kaashoek; D. Mazi‘eres und R. Morris. *Event-driven Programming for Robust Software*. Paper, MIT Laboratory for Computer Science, Proceedings of the 10th ACM SIGOPS European Workshop, 2002.

[27] C. Hood; A. Kreß; R. Stevenson; G. Versteegen und R. Wiebel. *Anforderungsmanagement: Methoden und Techniken, Einführungsszenarien und Werkzeuge im Vergleich*. iX Studie, Heise Zeitschriften Verlag, 2005.

[28] W.K. Chan; T.Y. Chen und T.H. Tse. *An Overview of Integration Testing Techniques for Object-Oriented Programs*. Paper, International Conference on Computer and Information Science, 2002.

[29] G. Vogel. *Software-Reengineering*. Vorlesungsskript, Fakultät für Informatik, Elektrotechnik und Informationstechnik, Universität Stuttgart, 2006.

[30] A. Zeller. *Extreme Programming*. Vorlesungsskript, Lehrstuhl Softwaretechnik, Universität des Saarlands, 2002.

[31] A. Zeller. *Software-Metriken*. Vorlesungsskript, Lehrstuhl Softwaretechnik, Universität des Saarlandes, Saarbrücken, 2002.

Abbildungsverzeichnis

1.1 Transformation in der modellbasierten Entwicklung 2

2.1 Die ARTIS Familie 4
2.2 Das ARTIS-Systemkonzept 5
2.3 Wegpunktplanung mit Maestro 5
2.4 Konzept der Missionssteuerung in ARTIS 6
2.5 Geomosaiking-Verhaltenssequenz 6
2.6 Software-in-the-Loop-Simulation 7
2.7 Hardware-in-the-Loop-Simulation 8
2.8 Beziehung zwischen Sender und Empfänger 9
2.9 Methodik der Ereignisanforderung 11
2.10 Anforderungsklassifikation 11
2.11 Teilung von Sender und Filter 12
2.12 Überlappende Gruppen 13
2.13 Klassifikation der Benachrichtigungsübertragung 14
2.14 Grafische State Chart Syntax 18
2.15 Interner Ablauf eines Zustandes 18
2.16 Zusammengesetzter Zustand am Beispiel 19
2.17 Interner Ablauf eines zusammengesetzten Zustandes (1) 19
2.18 Interner Ablauf eines zusammengesetzten Zustandes (2) 20
2.19 Projektion eines zusammengesetzten Zustandes am Beispiel 20

2.20 History-Zustand am Beispiel 21
2.21 Interner Ablauf eines History-Zustandes 21
2.22 Flache versus wiederverwendbare State Charts 22
2.23 Probleme bei der State Chart Modellierung 23

3.1 Schematische Darstellung des Modells 27
3.2 Schematisches Konzept des Re-Engineerings 28
3.3 Oberste Hierarchieebene der Ablaufsteuerung 34
3.4 Zusammengesetzter Zustand MissionMode 36
3.5 Zusammengesetzter Zustand CommandMode 37
3.6 Beispiel struktureller Komplexität 42
3.7 Beispiele für die Testanzahl auf Basis der McCabe-Metrik 42
3.8 Grundfunktionen von OXF 44
3.9 Strukturierter Ablauf der Integrationstage 47
3.10 Das Eisenhower-Prinzip 48
3.11 Typischer Verlauf der Integrationstage 49
3.12 Zustandstransformation bei der Integration 50
3.13 Erweiterung des OX-Frameworks 51

4.1 Klassifikation testender Verfahren 53
4.2 Transitionsüberdeckung 54
4.3 Testansatz für die drei Test Suiten 55
4.4 Der Testautomat 56
4.5 Binäre Eingabegrößen am Beispiel arDirectRC 60
4.6 Testschlüssel in EMT 60
4.7 Build Test Case in EMT 61
4.8 Beispiel für das Ereignis PositionReached 62
4.9 Run Test Case in EMT 64

4.10 Abhängigkeit der Durchlaufzeit von N_s und N_ϕ 66
4.11 Kritischer Pfad der Ablaufsteuerung 69
4.12 Testschlüssel in SCLT 70
4.13 Zustand Build Test Case in SCLT 71
4.14 Generierung der Benachrichtigungskette 71
4.15 Zustand Run Test Case in SCLT 73
4.16 Fehlerreport in SCLT 75
4.17 Sechsstufige Benachrichtigungskette von einem Initialzustand 75
4.18 Detektierung von einstufigen Schleifen 76
4.19 Teilung der Ereigniskette auf zwei Initalzustände 77
4.20 Sequenzdiagramm der Mustermission 78
4.21 Sequenzdiagramm des Missionsmodus 79
4.22 Pirouette in der Mustermission 80
4.23 Sequenzdiagramm des Kommandomodus 81

6.1 Dynamischer Ansatz der Event Middleware 87

A.1 Notation for states and transition 99
A.2 Notation for composite state, default arrow and internal events 99
A.3 Notation for a transition with history 100

C.1 Ausführlicher Transitionsüberdeckungsreport 112

Abkürzungsverzeichnis

ARTIS	Autonomous Rotorcraft Testbed For Intelligent Systems
CVS	Concurrent Versions Systems
DLR	Deutsches Zentrum für Luft- und Raumfahrt
EMT	Event Middleware Test
GCC	GNU Compiler Collecition
GNU	GNU is not UNIX
GPS	Global Positioning System
GUI	Graphical User Interface
HITL	Hardware-in-the-Loop-Simulation
IMU	Inertial Measurement Unit
MATLAB	Matrices Laboratory
MCP	Master Control Program
MDA	Model-Driven Architecture
OXF	Object Execution Framework
PID	Proportional-Integral-Differential
PIM	Platform Independent Model
PSI	Platform Specific Implementation
PSM	Platform Specific Model
QCC	QNX Compiler Collection
RPC	Remote Procedure Call
SCFT	Sequence Control Functional Test
SCLT	Sequence Control Logic Test
SDL	Specification and Description Language
SITL	Software-in-the-Loop-Simulation
SRTM	Shuttle Radar Topography Mission
STL	Standard Template Library
TCP	Transmission Control Protocol
UAV	Unmanned Aerial Vehicle
UDP	User Datagram Protocol
UML	Unified Modeling Language
USB	Universal Serial Bus
WLAN	Wireless Local Area Network
XP	Extreme Programming

Listings

2.1 Detektierung mit Benachrichtigungsmechanismus 10

3.1 Schnittstellenfunktion der Event Middleware 32

3.2 Trennung zwischen Haupt- und Teilzustand 32

3.3 Benachrichtigungsüberwachung 33

4.1 Verwendete CppUnit-Makros 57

4.2 Testfall PositionReached (True) 62

4.3 EMT-Spezifikation für den Zustand HoverTo 64

4.4 Spezifikation in SCLT 73

A.1 Enumeration of states 101

A.2 Struct of states 101

A.3 Class with statechart engine 102

A.4 Create and destroy statechart engine 102

A.5 Event processing with statechart engine 102

A.6 Event checking 103

A.7 Transition to a composite state with history 103

A.8 Internal transition 104

Symbolverzeichnis

State Chart

ε	Leere Wort
d	Benachrichtigungsobjekt (Do-Aktivität) für den Zustand
d_i	Benachrichtigungsobjekt (Do-Aktivität) für den Teilzustand
d_p	Benachrichtigungsobjekt (Do-Aktivität) für den Hauptzustand
e	Benachrichtigungsobjekt (Exit-Aktivität) für den Zustand
e_i	Benachrichtigungsobjekt (Exit-Aktivität) für den Teilzustand
e_p	Benachrichtigungsobjekt (Exit-Aktivität) für den Hauptzustand

Komplexität

e	Anzahl der Kanten des Kontrollflussgraphen G
n	Anzahl der Knoten des Kontrollflussgraphen G
p	Anzahl der unverbundenen Teile des Kontrollflussgraphen G
π	Anzahl der Binärverzweigungen des Kontrollflussgraphen G
V	Zyklomatische Komplexität des Kontrollflussgraphen G

Test Suite

δ	Schwellwert der Ereignisdetektierung
L	Anzahl der Variationen der Kettenlänge n
n	Länge der Benachrichtigungskette
N_α	Anzahl der Testfälle für die Event Middleware
N_β	Anzahl der Testfälle für die logische Struktur des State Charts
N_s	Anzahl der Zustände des State Charts
N_ϕ	Anzahl der detektierbaren Ereignisse des Senders
N_φ	Anzahl der Benachrichtigungsobjekte des Filters
t	Ausführungszeit
Δt	Ausführungszeit pro Zustand
ε	Genauigkeitsfaktor

Tabellenverzeichnis

2.1 Technische Daten der ARTIS-Plattform 4

3.1 Elementare Benachrichtigungen des Senders (1) 29
3.2 Elementare Benachrichtigungen des Senders (2) 30
3.3 Ausgewählte Transformationsmatrizen 31
3.4 Empirische Klassifikation durch die McCabe-Metrik 41
3.5 Zyklomatische Komplexität des Modells 43

4.1 Beziehung zwischen Eingabewerten und Ereignissen 59
4.2 Beispiel für einen Testschlüssel mit $N_s = 1$ und $N_\phi = 3$ 61
4.3 Beispiel für einen Testschlüssel mit $n = 2$ und $N_\varphi = 4$ 70

5.1 Fallstudie für Ereignisketten am Beispiel von RJB 84

B.1 Transformationsmatrizen des Filters (1) 105
B.2 Transformationsmatrizen des Filters (2) 106
B.3 Transformationsmatrizen des Filters (3) 107
B.4 Transformationsmatrizen des Filters (4) 108
B.5 Transformationsmatrizen des Filters (5) 109
B.6 Transformationsmatrizen des Filters (6) 110

Anhang A

UML State Chart Template

By GPSchultz (www.codeproject.com/samples/Statechart.asp).

This lightweight class allows you to easily implement a UML statechart in C++.

A.1 Introduction

This statechart "engine" (implemented as a C++ template class) implements the most commonly used aspects of UML statecharts. All you need to do is define an array of states, and implement event checking and handling methods. Then call the engine when an event occurs. The engine calls your event checking and handling methods in the correct order to figure out what event happened, and tracks the current state.

This is a lightweight implementation, which compiles under VC++ 6.0. The Boost statechart library does not compile under VC++ 6.0. This implementation requires less statechart-related housekeeping code than other C++ implementations. (See Miro Samek's, for example.)

A.2 Background

Statecharts were developed by David Harel, to add nesting and other features to flat state machines. (See David Harel, "On Visual Formalisms", Communications of the ACM, Vol. 31, No. 5, pp 514-530, 1988.) They were later added to the Unified Modeling Language (UML) and standardized. They are an excellent tool for modeling classes, sub-systems and interfaces that have many distinct states and complex transitions among them. My personal need for them arose while implementing a software interface between two real-time, embedded systems that controlled separate

machines requiring physical synchronization. I have also used them for modeling and implementing user interfaces that featured many modes.

The following is a brief summary of the notation and behavior of statecharts. For a full presentation of the UML statechart notation, see the UML 2.0 specification, available here.

Graphically, UML shows states as boxes with rounded corners, and transitions as arrows between boxes. (See Figure A.1.) The transitions are labeled with the event that causes the transition, optionally followed by a forward slash, and the action(s) that will be taken upon transition. A condition, called a "guard", can be indicated in square brackets. If the event happens, the guard condition is evaluated, and the transition is taken only if the condition is true.

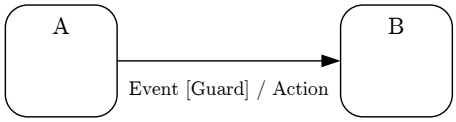

Bild A.1: Notation for states and transition

States can be nested (See Figure A.2.), which allows high-level events to invoke transitions that leave any of several states with just one arrow. The use of so-called compositeßtates keeps statechart diagrams much simpler than what flat state machines would require. Even though states can be nested, the system must always transition to some simple (i.e., non-composite) state.

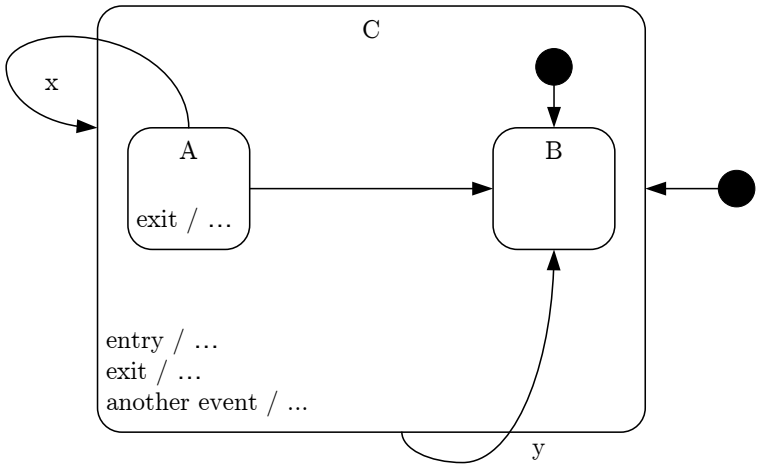

Bild A.2: Notation for composite state, default arrow and internal events

A solid circle at the beginning of an arrow indicates a default start state. A statechart must have at least one default start state designated, indicating the initial state of the system. If any transition–including a default one–ends on a compound state, then there must be a default state designation inside that compound state, and so on, eventually leading to a simple state. (One exception to this is described below.)

States can have internal transitions, which do not take the system to another state, yet do have actions associated. These are shown inside the state where they are handled. Compound states may also have internal transitions. Two special internal transitions are "entry" and "exit". These are executed upon entry to, and exit from, the corresponding state. This allows common actions such as initialization and destruction to be expressed one time, rather than as actions on every event leading to/from a state. Custom internal events can also be specified.

If a transition takes the system across several state boundaries, the various actions are executed in the following order:

1) The exit action(s) of all states that must be exited.

2) The action specified on the transition arrow.

3) The entry action(s) for all states that are entered.

Statecharts allow a transition to return to a previous state within a composite state, without requiring that you specify what state you were in previously. This is represented by a transition arrow ending in an encircled "H", for "history". (See Figure A.3.) For example, say an event can be handled from any of two simple states within a compound state. If you show a transition from the compound state, back inside it to the encircled "H" this means "handle the event and return to the state you most recently left". That is much simpler than showing two (or more) transitions with identical event/action labels.

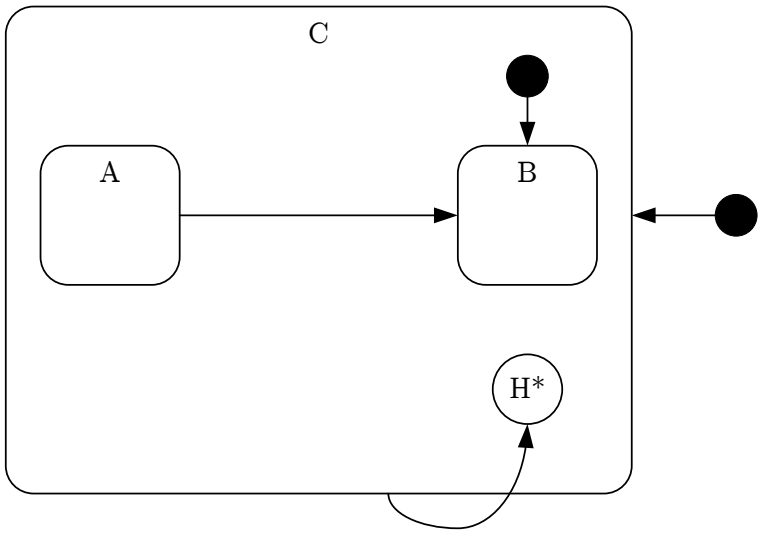

Bild A.3: Notation for a transition with history

There are two kinds of history returns in UML statecharts: shallow and deep. Shallow transitions (indicated by an "H") return to the most recently exited state at the level where the "H" is shown. If that does not lead to a simple state, then it is an error.

Deep history (indicated by an "H*") means that the system will return to the most recent simple state within that compound state that it most recently exited. So if the system in Figure 3 was in state A when event x happened, the system would return to state A.

A.3 Using the code

This implementation supports state nesting, entry, exit and custom internal events, default states and "generous" deep history transitions. (By "generous", I mean that I went beyond the UML specification: if a history return cannot find a recently exited state at a given level, it will try to use default state designations to get the system to a simple state. Only failing that, will it be considered an error.) This implementation does not currently support orthogonal states, factored transition paths, forks, joins, synch states or message broadcasting. Many of those unsupported features depend heavily on the system you are integrating this code into, and can be simulated in your event handlers.

Once you have designed your statechart graphically, do the following. First, define an enumeration of states. (The following comes from the included sample application, which illustrates all of the above features. This application performs a path-cover test of the statechart engine.)

```
enum eStates
{
    eStateA,
    eStartState = eStateA,
    eStateB,
    eStateC,
    eStateD,
    eStateE,
    eNumberOfStates
};
```

Listing A.1: Enumeration of states

Here I name the start state, and I also designate the number of states by the last enum value, so it will always be correct. Your specific state names will likely be more meaningful than these. Next, I allocate an array of the following:

```
typedef struct
{
    int32   m_i32StateName;
    int32   m_i32ParentStateName;
    int32   m_i32DefaultChildToEnter;
    uint32  (T::*m_pfu32EventChecker)(void);
```

```
    void      (T::*m_pfDefaultStateEntry)(void);
    void      (T::*m_pfEnteringState)(void);
    void      (T::*m_pfLeavingState)(void);
} xStateType;
```

Listing A.2: Struct of states

The structs must be initialized in the same order as the states in the enumeration above. Only the top-most state, here eStateA, will have -1 for its parent designation. States without a default sub-state (which will include all simple states) must specify -1 for the default sub-state. Every state must have an event checking/handling method, but need not have the last three fields filled in. Specify 0 for those if they are not defined.

The engine is referenced internally via a void pointer, so the class using the statechart must have a void pointer for its use:

```
class CStateClass
{
    .
    .
    .
    void *engine;
};
```

Listing A.3: Class with statechart engine

The engine must be created and destroyed in your class. (These macros and the ones below hide some of the necessary details. The engine name appears in all of them so that you may have more than one declared in the same client class.)

```
CStateClass::CStateClass(void)
{
    CREATE_ENGINE(CStateClass, engine, xaStates, eNumberOfStates,
                  eStartState);
}
CStateClass::~CStateClass(void)
{
    DESTROY_ENGINE(CStateClass, engine);
}
```

Listing A.4: Create and destroy statechart engine

At the point where events happen, place the following call:

```
PROCESS_EVENT(CStateClass, engine)
```

Listing A.5: Event processing with statechart engine

Since an event may be far more complex than a mere scalar value, the engine does not pass the event around to your event checking/handling methods. Rather, you must store the event in a member variable before calling `PROCESS_EVENT`, so that the event checking/handling methods can test for it. Thus, it does not appear in the above call.

For each state in your statechart diagram, an event checking/handling method must be defined that checks for all events that can be handled by that state. Simply test for each event that can happen while you are in the given state, as in the following:

```
uint32 CStateClass::evStateA(void)
{
    // Checking for event in state A.
    // include any guard conditions here
    if ('g' == m_cCharRead)
    {
        BEGIN_EVENT_HANDLER(CStateClass, engine, eStateA);
        // Put the transition action code here.
        END_EVENT_HANDLER(CStateClass, engine);
        return (u32HandlingDone);
    }
    return (u32NoMatch);
}
```

Listing A.6: Event checking

This arrangement allows any guard conditions to be tested at the same point in the code as the event itself, simplifying the code. Return u32NoMatch from a handler that does not find an event that it is supposed to handle.

The BEGIN_EVENT_HANDLER macro lets the engine know that you have found a match for an event. It records this fact, and executes the exit handlers for every state that must be exited to get to the destination state. It also records the fact that eStateA will be the state the system goes to after executing the handler code. (If the given state is a composite state, then of course you will end up in some simple state inside that composite state.) Control then returns to this method, where any transition actions are carried out. The END_EVENT_HANDLER macro executes any state entry handlers for states that you must enter, to end up in the correct simple state.

If you wish to transition to a composite state with history, ÖRtthe history flag onto the state name in the BEGIN_EVENT_HANDLER macro:

```
BEGIN_EVENT_HANDLER(CStateClass, engine, eStateA |
                    u32WithHistory);
```

Listing A.7: Transition to a composite state with history

For an internal transition, use the nno state changefflag:

```
BEGIN_EVENT_HANDLER( CStateClass , engine ,
                     u32NoStateChange );
```

Listing A.8: Internal transition

That's it!

A.4 Internals

Internally, the state definition array is parsed upon initialization, and more sophisticated data structures are created from it. Those data structures plus the state definition array are referenced when an event is being processed. (Don't even think about changing the state definition array at run-time!)

Because your code must call the statechart engine, it calls your event handlers back in order to find out who will be handling the event and where to go next. Hence, when executing an event handler, you are on the same thread that initially called the engine.

The code contains numerous assert() statements, which check for a variety of simple mistakes that can be made when defining the array of states. Examples are failure to have an initial default state, and a state not having a valid parent state. Such errors are caught during initialization, rather than at run-time.

Anhang B

Event Middleware Filter

Die Tabellen B.1, B.2, B.3, B.4, B.5 und B.6 zeigen die zustandsabhängigen Transformationsmatrizen des Filters der Event Middleware. Die Prioritäten werden durch die vorgegebene Rangfolge bestimmt, wie in Tabelle 3.3 aus Kapitel 3.2 dargestellt.

CommandMode	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evManualPilot			X																	
evCommandStop				X																
evMissionModeOn						X														
evPulse																				

ControllerOn	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evControllerOn	X																			
evPulse																				

Tabelle B.1: Transformationsmatrizen des Filters (1)

FlyTo

	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evFlyToFinished								X			X									
evPulse																				

FlyToInit

	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evFlyToInitFinished																			X	
evPulse																				

HoverTo

	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evHoverToFinished								X			X	X								
evPulse																				

HoverTurn

	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evHoverTurnFinished								X				X								
evPulse																				

Tabelle B.2: Transformationsmatrizen des Filters (2)

Landing	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evLandingFinished									X											
evPulse																				

MissionControllerOff	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evAutoPilot	X	X																		
evPulse																				

MissionMode	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evManualPilot			X																	
evDirectCommandValid															X					
evMissionModeStop				X																
evDirectBehaviorCommand						X														
evPulse																				

Tabelle B.3: Transformationsmatrizen des Filters (3)

GetCommand	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evMissionModeOn					X	X	!X													
evCmdTakeOff																	X	X		
evCmdLand																	X	X		
evCmdWaitingFor																		X		
evCmdHoverTo																		X		
evCmdHoverTurn																		X		
evCmdPirouetteAroundXY																		X		
evCmdPirouetteFlightXYZ																		X		
evCmdFlyToInit																		X		
evPulse																				

GetDirectCommand	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evDCmdVelGCS																				X
evDCmdPosVCObjTrack																				X
evDCmdPosVCObstAvoid																				X
evDCmdVelFLARMObstAvoid																				X
evPulse																				

PirouetteAroundXY	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evPirouetteAroundXYFinished													X							
evPulse																				

Tabelle B.4: Transformationsmatrizen des Filters (4)

PirouetteFlightXYZ	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evPirouetteFlightXYZFinished											X	X								
evPulse																				

PosVC-ObjTrack VelGCS	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evCommandStop																X				
evPulse																				

PosVC-ObstAvoid VelFLARM-ObstAvoid	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evMissionModeOn																X				
evPulse																				

WaitingFor	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evTimeOut														X						
evPulse																				

Tabelle B.5: Transformationsmatrizen des Filters (5)

SlowDown

	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evManualPilot			X																	
evDirectCommandValid															X					
evMissionUpdate						X														
evLanded									X											
evStandStill								X												
evPulse																				

StandBy

	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evManualPilot			X																	
evDirectCommandValid															X					
evMissionModeOn					X	X	!X													
evPulse																				

TakeOff

	Flyable	AutoPilot	ManualPilot	CommandStop	MissionModeOn	MissionUpdated	MissionFinished	StandStill	Landed	HeightReached	PositionReached	HeadingReached	PirouetteFinished	TimeOut	DirectCommandValid	DirectCommandFinished	GroundHeightAvailable	Behaviorsequence	initSplineAsyncFinished	vdTarget[CmdType]
evTakeOffFinished										X		X								
evPulse																				

Tabelle B.6: Transformationsmatrizen des Filters (6)

Anhang C

Transitionsüberdeckung

Das Bild C.1 zeigt beispielhaft einen ausführlichen Report der Transitionsüberdeckung mit allen Transitionen des State Charts und deren Aktivierungen.

```
SC: Transition-Entry (1) in rootState:DefaultTransition
SC: Transition-Entry (0) in StandBy:evDirectCommandValid
SC: Transition-Entry (1) in StandBy:evManualPilot
SC: Transition-Entry (0) in StandBy:evMissionModeOn
SC: Transition-Entry (0) in SlowDown:evDirectCommandValid
SC: Transition-Entry (0) in SlowDown:evManualPilot
SC: Transition-Entry (1) in SlowDown:evStandStill
SC: Transition-Entry (0) in SlowDown:evLanded
SC: Transition-Entry (0) in SlowDown:evMissionUpdate
SC: Transition-Entry (0) in MissionMode:DefaultTransition
SC: Transition-Entry (0) in MissionMode:evManualPilot
SC: Transition-Entry (0) in MissionMode:evDirectCommandValid
SC: Transition-Entry (0) in MissionMode:evMissionModeStop
SC: Transition-Entry (0) in MissionMode:evDirectBehaviorCommand
SC: Transition-Entry (0) in WaitingFor:evTimeOut
SC: Transition-Entry (0) in TakeOff:evTakeOffFinished
SC: Transition-Entry (0) in PirouetteFlightXYZ:evPirouetteFinished
SC: Transition-Entry (0) in PirouetteAroundXY:evPirouetteFinished
SC: Transition-Entry (0) in GetCommand:evCmdWaitingFor
SC: Transition-Entry (0) in GetCommand:evCmdHoverTo
SC: Transition-Entry (0) in GetCommand:evCmdTakeOff
SC: Transition-Entry (0) in GetCommand:evCmdLand
SC: Transition-Entry (0) in GetCommand:evCmdPirouetteAroundXY
SC: Transition-Entry (0) in Landing:evLandingFinished
SC: Transition-Entry (0) in HoverTurn:evHoverTurnFinished
```

```
SC: Transition-Entry (0) in HoverTo:evHoverToFinished
SC: Transition-Entry (0) in FlyToInit:evFlyToInitFinished
SC: Transition-Entry (0) in FlyTo:evFlyToFinished
SC: Transition-Entry (1) in MissionControllerOff:evAutoPilot
SC: Transition-Entry (1) in ControllerOn:evControllerOn
SC: Transition-Entry (0) in CommandMode:DefaultTransition
SC: Transition-Entry (0) in CommandMode:evCommandStop
SC: Transition-Entry (0) in CommandMode:evManualPilot
SC: Transition-Entry (0) in CommandMode:evMissionModeOn
SC: Transition-Entry (0) in GetDirectCommand:evDCmdVelGCS
SC: Transition-Entry (0) in GetDirectCommand:evDCmdVelFLARMObst
SC: Transition-Entry (0) in GetDirectCommand:evDCmdPosVCObjTrack
SC: Transition-Entry (0) in GetDirectCommand:evDCmdPosVCObstAvoid
```

Bild C.1: Ausführlicher Transitionsüberdeckungsreport

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, den 3. Juli 2007 Robert Schwarzer