Martin Mutz

Eine durchgängige modellbasierte Entwurfsmethodik für eingebettete Systeme im Automobilbereich

Cuvillier Verlag Göttingen

Von der Carl-Friedrich-Gauß-Fakultät für Mathematik und Informatik
der Technischen Universität Braunschweig

genehmigte Dissertation

zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

Martin Mutz

Eine durchgängige modellbasierte Entwurfsmethodik für eingebettete Systeme im Automobilbereich

Datum der Promotion: 14. November 2005

1. Referentin: Prof. Dr. Ursula Goltz
2. Referent: Prof. Dr. Stefan Kowalewski

eingereicht am: 30. August 2005

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.
1. Aufl. - Göttingen : Cuvillier, 2005
Zugl.: (TU) Braunschweig, Univ., Diss., 2005
ISBN 3-86537-684-3

© CUVILLIER VERLAG, Göttingen 2005
Nonnenstieg 8, 37075 Göttingen
Telefon: 0551-54724-0
Telefax: 0551-54724-21
www.cuvillier.de

Alle Rechte vorbehalten. Ohne ausdrückliche Genehmigung des Verlages ist es nicht gestattet, das Buch oder Teile daraus auf fotomechanischem Weg (Fotokopie, Mikrokopie) zu vervielfältigen.
1. Auflage, 2005
Gedruckt auf säurefreiem Papier

ISBN 3-86537-684-3

Für meine Familie

Kurzfassung

Der zurzeit stattfindende Technologiewandel in der Automobiltechnik ist dadurch gekennzeichnet, dass Fahrzeugfunktionen zunehmend durch Software in Form von Steuergeräten realisiert werden, anstatt wie bisher durch Elektrik, Mechanik oder Hydraulik. Die Erstellung Software-basierter Funktionen hat unter anderem den Vorteil, dass neben einer kostengünstigeren Anpassung, Erweiterung und Vervielfältigung bestehender Fahrzeugfunktionen, eine einfachere Erstellung und Validierung dieser ermöglicht wird. Des Weiteren wird durch den zunehmenden SW-Anteil die Wiederverwendung vorhandener Funktionen gefördert. Das signifikante Wachstum der Funktionen im Kfz-Bereich führt jedoch zu einem Anstieg der Steuergeräte und somit zur Erhöhung der SW-Komplexität im Gesamtfahrzeug, so dass die Fahrzeugfunktionen mit klassischen Entwicklungsmethoden kaum noch zu realisieren sind. Vielmehr ist eine durchgängige Entwicklungsmethodik basierend auf einem strukturierten Entwicklungsprozess und dem Einsatz moderner CASE Tools, unter Beachtung internationaler Normen und Standards, erforderlich.

Ziel der vorliegenden Arbeit ist es daher, eine derartige Entwurfsmethodik zur Beherrschung der SW-Komplexität im Automobilbereich vorzustellen. Dabei bildet der modellbasierte Ansatz den Rahmen dieser Entwurfsmethodik. Im Gegensatz zu existierenden modellbasierten Methoden, wie z. B. CARTRONIC, BMW-ROOM und BASEMENT, deckt die hier vorgestellte Methodik den gesamten Entwicklungsprozess von der Anforderungsbeschreibung über den Entwurf bis hin zur Implementierung ab. Denn gerade die fehlenden methodischen und werkzeugtechnischen Schnittstellen zwischen den verschiedenen Phasen verursachen einen Bruch im gesamten Entwicklungsprozess und führen somit nur zu suboptimalen Ergebnissen. Ein weiteres Ziel der Entwurfsmethodik ist die Sicherstellung der SW-Qualität. Dazu werden die entwickelten Modelle mit einem eigens entwickelten Analysewerkzeug automatisch auf Einhaltung von Modellierungsrichtlinien überprüft, die zuvor vom Benutzer aus einem Richtlinienkatalog entnommen und projektspezifisch angepasst werden.

Die im Rahmen dieser Arbeit entwickelte Entwurfsmethodik eignet sich in erster Linie zur Entwicklung verteilter, reaktiver Steuerungsfunktionen mit weicher Echtzeit für eingebettete Systeme im Automotive-Umfeld. Dazu gehören beispielsweise Systeme aus dem Bereich Body- und Komfortelektronik. Die Erprobung der Methodik erfolgt anhand eines prototypischen Reengineerings der Subsysteme Zentralverriegelung, Fensterheber und Spiegelsteuerung eines VW Polos.

Danksagungen

An dieser Stelle möchte ich all jenen Personen danken, die zum Gelingen dieser Arbeit beigetragen haben.

Zuerst danke ich meiner Mentorin Prof. Ursula Goltz für die Ermöglichung meiner Arbeit als wissenschaftlicher Mitarbeiter und für den nötigen wissenschaftlichen Freiraum. Ihre Diskussionsbeiträge und Fragen haben der Arbeit immer wieder wichtige Impulse verliehen und sie in die richtige Richtung gelenkt. Ebenfalls möchte ich mich bei Herrn Prof. Stefan Kowalewski für die Übernahme des Zweitberichts und seine konstruktiven Anregungen bedanken.

Die Arbeit in der Abteilung Programmierung und im STEP-X Team war durch den freundschaftlichen Umgang zwischen den Kollegen und durch ein sehr anregendes Arbeitsklima geprägt. Beides hat einen wesentlichen Teil zum Gelingen der Arbeit beigetragen. Mein besonderer Dank gilt Michaela Huhn und Bastian Florentz, die mir in den zahlreichen Diskussionen immer wieder neue Anregungen für meine Arbeit gaben.

Weiterhin möchte ich allen Diplomanden, Studienarbeitern und wissenschaftlichen Hilfskräften für Ihren Einsatz und ihre Leistungsbereitschaft danken. Insbesondere erwähnen möchte ich die Arbeiten von Gerald Beitzen-Heineke, Christoph Knieke, Steffen Pietsch, Lars Potratz, Stefan Schulze und Jens Steiner. Besonderer Dank gilt Frank Samir Attia und Andreas König für die sorgfältige Durchsicht und die hilfreichen Kommentare zu dieser Arbeit.

Ohne die Unterstützung meiner Eltern während meines gesamten beruflichen Werdegangs wäre diese Arbeit nie möglich gewesen. Sie ließen mir alle notwendigen Freiheiten und förderten mich auch in schwierigen Situationen. So legten sie den Grundstein für diese Arbeit. Zu großem Dank bin ich auch meiner Frau Antje verpflichtet. Sie musste in den letzten Jahren viel Verständnis aufbringen und oftmals ihre eigenen Wünsche und Ziele zurücknehmen. Sie hat mir immer den nötigen Rückhalt gegeben und mir in schwierigen Zeiten immer wieder neuen Mut zugesprochen. Sie hat nie am Gelingen meines Vorhabens gezweifelt und mich so immer wieder neu ermutigt. Dafür möchte ich mich ganz herzlich bei ihr bedanken.

Braunschweig, im November 2005 Martin Mutz

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AML	Automotive Modeling Language
API	Application Programming Interface
ASAM	Association for Standardization of Automation and Measuring Systems
AUTOSAR	Automotive Open System Architecture
BMBF	Bundesministerium für Bildung und Forschung
CASE	Computer Aided Software Engineering
CAN	Controller Area Network
CDIF	CASE Data Interchange Format
CRC	Class-Responsibility-Collaboration
DL	Digitales Lastenheft
DTD	Dokumenttypdeklaration
ECU	Electronic Control Unit
FH	Fensterheber
FIFO	First In First Out
GUI	Graphical User Interface
HAL	Hardware Abstraction Layers
HIL	Hardware in the Loop
HMI	Human Machine Interface
HTML	Hyper Text Markup Language
HW	Hardware
KM	Konfigurationsmanagement
LIN	Local Interconnect Network
MAAB	MathWorks Automotive Advisory Board
MEGMA	MSR Engineering Graphic Model Exchange
MISRA	Motor Industry Software Reliability Association
MMI	Mensch Maschine Interface
MSR	Manufacturer Supplier Relationship

MOST	Media Orientated Systems Transport
OEM	Original Equipment Manufacturers
OIL	OSEK Implementation Language
OMG	Object Management Group
OSEK	Offene Schnittstellen für die Elektronik im Kraftfahrzeug
PM	Projektmanagement
QM	Qualitätsmanagement
QS	Qualitätssicherung
SE	Systemerstellung
SEU	Software-Entwicklungsumgebung
SIL	Software in the Loop
STEP-X	Strukturierter Entwicklungsprozess für Automobil Anwendungen
SVG	Scalable Vector Graphics
SW	Software
SysML	Systems Modeling Language
TT	Technisches Tailoring
UML	Unified Modeling Language
VDX	Vehicle Distributed eXecutive
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
ZV	Zentralverriegelung

Kapitel 1

1 Einleitung

„Das Auto hat keine Zukunft, ich setze aufs Pferd."

Wilhelm II. (1859 - 1941) deutscher Kaiser und König von Preußen

Im zunehmenden Kampf um Marktanteile sind die Automobilhersteller bestrebt, durch eine Modelloffensive individueller und schneller auf Kundenwünsche einzugehen. Die Folge davon ist eine stetig wachsende Anzahl von Fahrzeugvarianten. So stieg beispielsweise in Deutschland die Zahl der Fahrzeugmodelle in den letzten 20 Jahren von 140 auf 250 an; die Zahl der Varianten nahm dabei um 80% zu [SSDC+04]. Um auch weiterhin die Wettbewerbsfähigkeit sicherzustellen, werden die Automobilhersteller ihre Variantenvielfalt zukünftig noch weiter ausbauen bei gleichzeitiger Reduktion von Entwicklungszeit und -aufwand [Schl02].

Werden in diesem Zusammenhang die Neuerungen in den verschiedenen Fahrzeugklassen genauer betrachtet, so ist festzustellen, dass insbesondere die Fahrzeugfunktionen einen erheblichen Teil der Innovationen ausmachen. Angefangen bei Motor- und Getriebesteuerung über Fahrdynamik-, und Navigationssysteme bis hin zu Komfort- und Assistenzsystemen, sind Fahrzeugfunktionen in allen Bereichen eines modernen Mittelklassefahrzeugs zu finden. Die Realisierung derartiger Funktionen erfolgt durch eine Kombination von Software und Hardware in Form von *Steuergeräten* (engl.: electronic control unit, ECU), die als *Eingebettete Systeme* fungieren. Diese sind wie folgt definiert:

Eingebettetes System/Steuergerät

Ein Eingebettetes System ist eine abgegrenzte (SW-/HW-)Einheit, die über Sensoren und Aktuatoren mit einem Gesamtsystem verbunden ist und darin Überwachungs-, Steuerungs- bzw. Regelungsaufgaben übernimmt. Ein Steuergerät ist in der Automobiltechnik die physikalische Implementierung eines eingebetteten Systems [BBK98].

Schätzungen von [Schl02, Vect04] prognostizieren, dass künftig auf die Elektronik 90 Prozent aller Innovationen im Automobil entfallen, von denen wiederum 80 Prozent durch Software und somit durch Steuergeräte realisiert werden. Das signifikante Wachstum des SW-Anteils wird auch einen Anstieg der Software-Komplexität zur Folge haben [Gres01]. Die Realisierung derartig komplexer Systeme ist mit klassischen Entwicklungsmethoden kaum noch möglich. Vielmehr ist eine durchgängige Entwicklungsmethodik basierend auf einem strukturierten Entwicklungsprozess und dem Einsatz moderner CASE Tools, unter Beachtung internationaler Normen und Standards, erforderlich. Ein derartiges Vorgehen wird im Rahmen dieser Arbeit vorgestellt.

In Abschnitt 1.1 wird auf den derzeitigen Wandel in der Automobiltechnik detailliert eingegangen. Die sich daraus neu ergebenden Anforderungen an die SW-Entwicklung werden abgeleitet und Maßnahmen zur Lösung der Problematik aufgezeigt. Anschließend werden existierende Forschungsprojekte vorgestellt, deren Ansätze zur Lösung einzelner Problemfelder beitragen, nicht aber den gesamten SW-Entwicklungsprozess begleiten. In diesem Zusammenhang wird auch das STEP-X Projekt vorgestellt, das einen ganzheitlichen Lösungsweg darstellt. Abschnitt 1.2 stellt die Zielsetzung der vorliegenden Arbeit vor. Abschließend wird im Abschnitt 1.3 der Aufbau der Arbeit dargestellt.

1.1 Der Wandel im Automobilbereich und dessen Folgen für die SW-Entwicklung

Der zurzeit stattfindende Technologiewandel in der Automobiltechnik ist dadurch gekennzeichnet, dass Funktionen im Kfz-Bereich zunehmend durch Software in Form von Steuergeräten realisiert werden, anstatt wie bisher durch Elektrik, Mechanik oder Hydraulik. Die Erstellung Software-basierter Funktionen hat unter anderem den Vorteil, dass neben einer kostengünstigeren Anpassung, Erweiterung und Vervielfältigung bestehender Fahrzeugfunktionen, eine einfachere Erstellung und Validierung dieser ermöglicht wird. Des Weiteren wird durch den zunehmenden SW-Anteil die Wiederverwendung vorhandener Funktionen gefördert.

Die Strukturen und Architekturen derartiger elektronischer Systeme haben in den letzten Jahren, bedingt durch ständig steigende Forderungen nach mehr Sicherheit, Komfort und Informationsmöglichkeiten, starke Veränderungen erfahren. Während die Kraftfahrzeug-Elektronik vor einigen Jahren von einigen wenigen Steuergeräten geprägt war, ist die Zahl der Steuergeräte in heutigen Mittelklassefahrzeugen bereits um den Faktor 2-3 auf etwa 40 Steuergeräte gestiegen [Grim03]. In einigen Premiumfahrzeugen beläuft sich die Anzahl auf ca. 70 Steuergeräte mit über 400 Einzelfunktionen [Broy01, SB05, Rapp04]. Abbildung 1.1 verdeutlicht den Anstieg der Steuergeräteanzahl in Fahrzeugen der Marke Volkswagen.

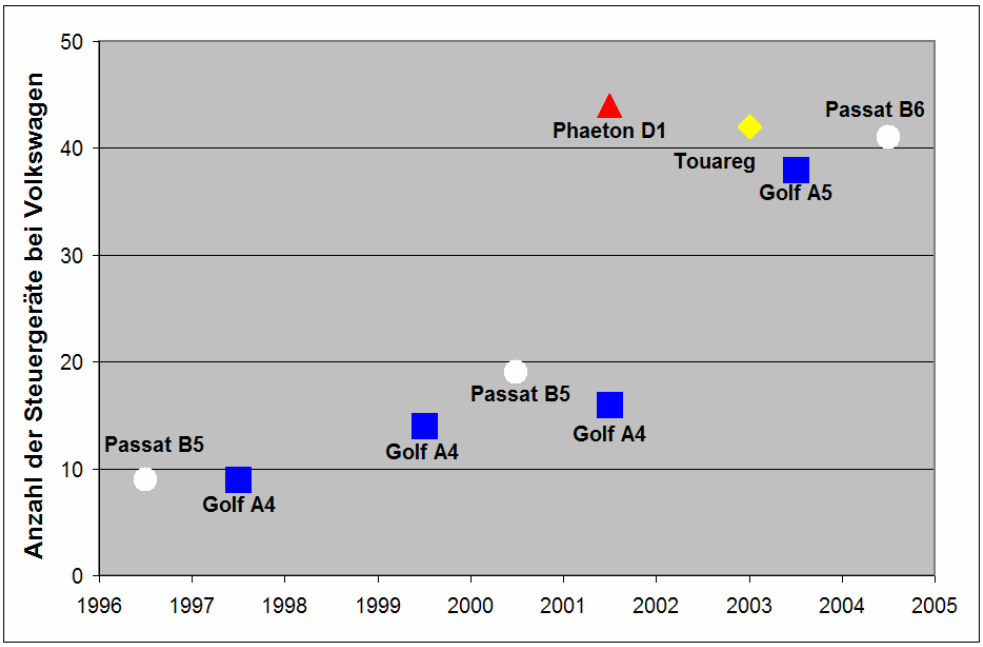


Abbildung 1-1: Anstieg der Fahrzeugsteuergeräte bei der Volkswagen AG [Scha05]

Mit dem Anstieg der Steuergeräteanzahl wurde zugleich die Verteilung der immer komplexer werdenden Funktionalität verfolgt. Während anfänglich an sich autonom arbeitende Komponenten wie Motor- und Getriebesteuerung auf die Steuergeräte verteilt wurden, ist heute eine Verteilung der einzelnen Funktionen auf einem Steuergerätenetzwerk mit verschiedenen Bussystemen, wie CAN, LIN und MOST zu beobachten. Durch die Vernetzung können einerseits redundante Funktionen reduziert werden, da dieselbe Funktion auf einem zentralen Steuergerät implementiert werden kann. Andererseits führt der Ausfall eines Steuergerätes nicht zwingend zu einem Systemausfall, da sicherheitsrelevante Funktionen durch andere Steuergeräte aufrechterhalten werden können.

Durch die Möglichkeit, Steuergeräte zu vernetzten und somit Funktionen auf das gesamte Fahrzeug zu verteilen, entstanden neuartige Fahrzeugfunktionen, wie z. B. Komfort- und Assistenzsysteme[1], die in den vergangenen Jahren die wesentlichen Treiber für den starken Anstieg der Steuergeräteanzahl waren.

Die hohe Anzahl der Steuergeräte sowie der Vernetzungsgrad zwischen den einzelnen Funktionen lässt die Komplexität der eingebetteten Systeme erahnen. Traditionelle Vorgehensweisen und Methoden reichen kaum aus, um die Komplexität der SW-Systeme zu beherrschen. Die Folge war die Vergabe der SW-Entwicklung an die Zulieferer, womit der Anspruch auf Innovationen in der Funktionsentwicklung stark zurückging. Heute beträgt der Zuliefererenteil zwischen 70% und 100% [BBK98]. Um zukünftig wettbewerbsdifferenzierende Merkmale erlangen zu können, ist sowohl ein Umstieg auf neue Technologien als auch ein Know-how-Transfer in Richtung Automobilhersteller notwendig.

[1] Bekannte Assistenzsysteme sind beispielsweise *Anti Blockier System*, *Dynamische Stabilitätskontrolle*, *Elektronische Stabilitätsprogramm* und *Abstandsregeltempomaten*.

1.1.1 Anforderungen an die künftige SW-Entwicklung

Zur Beherrschung der SW-Komplexität ist es erforderlich, zunächst die Schwachstellen herkömmlicher Entwicklungsprozesse zu identifizieren und diese daraufhin durch moderne Methoden und Werkzeuge zu beseitigen. In den Arbeiten von [BBK98, BBEF+98, Gres01, Schl02, SPHP02, Rau02] werden unterschiedliche Problemfelder traditioneller SW-Entwicklungsprozesse im Automobilbereich identifiziert. Einige der sich dabei ergebenden Anforderungen an die zukünftige SW-Entwicklung sind nachfolgend aufgelistet:

Einheitliche Vorgehensweise bei der SW-Entwicklung

Die Erstellung der Fahrzeugfunktionen findet gewöhnlich in Zusammenarbeit mit den Zulieferern statt. Dabei werden die Fahrzeugfunktionen in Form von Steuergeräten entsprechend den Anforderungen der Automobilhersteller (engl.: Original Equipment Manufacturers, OEM) ausgeliefert. Anschließend verläuft die Abnahme und Integration der Steuergeräte durch den Auftraggeber. Aufgrund der nicht vorhandenen standardisierten Vorgehensweise zur Erstellung von Steuergeräte-Software, sind die Lieferanten an ihren jeweiligen firmeninternen Entwicklungsprozess gebunden [BBK98]. Die daraus resultierende Problematik spiegelt sich z. B. dadurch wieder, dass für den Austausch von Lasten- und Pflichtenheften sowie von Artefakten[2] keine zentrale Datenablage mit geeigneten Werkzeugen existiert. Ein Datenaustausch auf Grundlage von Word™-Dokumenten ist im Gegensatz zu einem datenbankorientierten Ansatz eher von Nachteil, da spezielle Funktionen eines Anforderungsmanagement-Tools, wie z. B. Nachverfolgbarkeit, Zugriffskontrolle, Versions- und Änderungsmanagement, durch ein Textverarbeitungsprogramm nicht abgedeckt sind. In [HHM03] wird auf diese Probleme näher eingegangen. Des Weiteren stellt das Fehlen eines standardisierten Austauschformats sowie einheitlicher Werkzeugschnittstellen zwischen OEMs und Zulieferern eine große Herausforderung dar [WM99].

Eine industrieweit einheitliche Vorgehensweise mit entsprechend standardisierten Formaten und Schnittstellen würde eine arbeitsteilige SW-Entwicklung zwischen Herstellern und Zulieferern verbessern [LBB+01].

Verbesserung der Lastenhefte

In der Arbeit von [Spre96] ergab die Analyse vorhandener Entwicklungsprozesse im Automobilbereich, dass der Fehleranteil bereits in frühen Phasen beträchtlich ist. Diese Problematik ist typisch für den heutigen Entwicklungsablauf. Die Ursache liegt vor allem in der mangelnden Qualität der Lastenhefte, die das zu entwickelnde System nur unzureichend beschreiben.

[2] Unter Artefakten werden alle Ergebnisse einer SW-Entwicklung verstanden.

Typische Mängel in diesem Zusammenhang sind:

- ***Unvollständigkeit:*** Fehlen wichtiger Informationen oder ungenügende Darstellung der Informationen
- ***Inkonsistenz:*** mangelnde Beschreibung von Beziehungen und Querverweisen
- ***Mehrdeutigkeit:*** missverständlich beschriebene Informationen
- ***Redundanz:*** gleicher Informationsinhalt mehrfach vorkommend
- ***Überspezifikation:*** falsche, irrelevante oder undurchführbare Anforderungen
- ***Informationsvermischung:*** keine Trennung zwischen funktionalen Anforderungen und bauteilspezifischen Informationen

Derartige Schwächen in den Anforderungsspezifikationen haben zur Folge, dass es häufig zu Unstimmigkeiten zwischen Auftraggebern und Auftragnehmern kommt. Die ausgelieferten Steuergeräte entsprechen oftmals nicht den Wünschen, wodurch ständige Nachbesserungen notwendig sind. Ebenso bieten die Lastenhefte beim abschließenden Abnahmetest seitens der Automobilhersteller keine Möglichkeit, eine direkte Zuweisung der Testfälle auf die vorhandenen Anforderungen vorzunehmen. Da die Lastenheftqualität nicht befriedigend ist, eignen sich die bestehenden Lastenhefte nur begrenzt als Vertragsgrundlage [Spre96].

Ziel muss es daher sein, die Lastenhefte so zu verbessern, dass die Zulieferer präzise Anforderungen erhalten, in denen das gewünschte Funktionsverhalten genau und vollständig spezifiziert ist.

Einsatz modellbasierter Techniken

Die Entwicklung eingebetteter Systeme in der Automobiltechnik wird bisher im Wesentlichen durch informelle Techniken unterstützt. Das bedeutet, dass die Lastenhefte hauptsächlich in textueller Form vorliegen und die anschließende Umsetzung in Quellcode manuell erfolgt. Ein derartiges Vorgehen, ausschließlich natürlichsprachliche Beschreibungsmittel zur Formulierung der Anforderungen zu verwenden, bringt verschiedene Nachteile mit sich. So ist beispielsweise in vielen Fällen eine Vermischung unterschiedlicher Sichten (z. B. Struktur, Kommunikation, Verhalten) auf das zu entwickelnde System gegeben, so dass bestimmte Aspekte nicht separiert werden können. Ebenso ist eine Abstrahierung bzw. Detaillierung der Systembeschreibung nur unter enormen Aufwand möglich. Darüber hinaus ist der Zusammenhang vieler komplexer Anforderungen auf der rein textuellen Ebene nur schwer zu überschauen. Außerdem lassen sich die Eigenschaften der auf diese Weise entwickelten Software nur schwer validieren und verifizieren.

Zur Lösung der Schwierigkeiten bieten sich formale Techniken zwar an, jedoch ist der Einsatz dieser stark theoretisch geprägten Techniken in der Praxis durch den beträchtlich hohen Einarbeitungsaufwand relativ gering. Vielmehr muss die Möglichkeit bestehen, die textuellen Anforderungen durch grafische Modelle zu präzisieren, um sowohl eine Erhöhung der Lesbarkeit und des Verständnisses zu erzielen, als auch eine ausführbare Spe-

zifikation zu erstellen und somit eine frühzeitige Testbarkeit zu erreichen. Unter dem Begriff Modell wird folgendes verstanden:

Modell

Ein Modell ist eine Abstraktion eines Systems mit der Zielsetzung, das Nachdenken über ein System zu vereinfachen, indem irrelevante Details ausgelassen werden [BD00].

Durchgängige werkzeuggestützte Entwurfsmethodik

Einen besonderen Stellenwert zur Beherrschung der hohen SW-Komplexität haben Werkzeuge. So unterstützen die verschiedenen CASE Tools bereits heute die meisten Aktivitäten eines Entwicklungsprozesses, angefangen beim Anforderungsmanagement über Modellierung und automatische Codegenerierung bis hin zum Testen.

Das Hauptproblem bei der werkzeuggestützten Entwicklung von Steuergeräte-Software ist nicht etwa der Funktionsumfang der CASE-Tools, sondern die nicht vorhandene Integration zwischen den Werkzeugen. Durch das Fehlen standardisierter Schnittstellen und einheitlicher Datenformate kann ein durchgängiger werkzeugbegleiteter Entwicklungsprozess nur schwer erzielt werden. Ein weiteres Problem sind die in vielen Fällen fehlenden Vorgaben für den Werkzeugeinsatz in den Unternehmen. Stattdessen sind verschiedene Werkzeuge in unterschiedlichen Abteilungen im Einsatz, die miteinander nicht kompatibel sind bzw. über die geforderte Funktionalität nicht verfügen. Beispielsweise werden immer noch bei vielen Automobilherstellern gängige Textverarbeitungsprogramme für das Requirements-Engineering und -management verwendet.

Sieht man einmal von der Inkompatibilität ab, so liegt das Problem nicht allein in der Kopplung existierender Werkzeuge. Vielmehr muss zunächst die Methodik vertieft werden, um eine integrierte Werkzeugkette zu definieren, die den Steuergeräte-Entwicklungsprozess realisiert [BBK98]. Dabei ist der Begriff Methodik wie folgt definiert:

Methodik/Methode

Der Begriff Methodik beschreibt ein planmäßiges Vorgehen unter Einfluss mehrerer Methoden und entsprechender Hilfsmittel. Unter einer Methode wird eine planmäßig angewandte, begründete Vorgehensweise zur Erreichung von festgelegten Zielen verstanden [Balz01].

Frühzeitige Fehlererkennung

In den gegenwärtig existierenden Entwicklungsprozessen für Steuergeräte-Software ist die Forderung nach Korrektheit und Zuverlässigkeit noch nicht ausreichend berücksichtigt. Beispielsweise werden die Testvorgänge im Normalfall erst am Ende der SW-Entwicklung

durchgeführt, so dass Spezifikations- und Entwurfsfehler erst in der Implementierungs- oder Integrationsphase aufgedeckt werden können. Da im Allgemeinen der Kostenaufwand für notwendige Iterationsschleifen exponentiell mit dem Zeitpunkt der Fehlererkennung steigt, besteht die dringende Notwendigkeit, die spezifizierten Anforderungen bereits zu einem frühen Zeitpunkt der SW-Entwicklung zu überprüfen. Dies ist unter anderem durch Simulation des Verhaltens möglich, sofern ausführbare Spezifikationen vorliegen.

Ein weiteres Problem liegt in der Form der erbrachten Ergebnisse seitens der Zulieferer. Die Fahrzeugfunktionen werden in Form von Steuergeräten an die Automobilhersteller geliefert. Das anschließende Testen dieser Fahrzeugfunktionen kann dann nur durch einen Black-Box-Test erfolgen.

1.1.2 Maßnahmen zur Lösung der Problematik

Die im vorherigen Abschnitt genannten Anforderungen an die zukünftige SW-Entwicklung können im Wesentlichen durch einen strukturierten und optimierten Entwicklungsprozess unter Verwendung von geeigneten Entwurfsmethoden, Techniken, Werkzeugen und Standards erfüllt werden.

Da die Probleme bei der SW-Entwicklung im Automobilbereich vor allem in den frühen Entwicklungsphasen durch unzureichende Lastenhefte entstehen, müssen insbesondere rechnergestützte Systemspezifikationen stärker in den künftigen Entwicklungsprozess einbezogen werden. Dadurch wird die Qualität der Spezifikationsdokumente verbessert. Dies führt wiederum zu einer Reduktion der Folgefehler.

Unter rechnergestützter Systemspezifikation ist folgendes zu verstehen:

Rechnergestützte Systemspezifikation

Die rechnergestützte Systemspezifikation umfasst Methoden, Beschreibungsmittel und Werkzeuge zur Entwicklung einer vollständigen und konsistenten Spezifikation des Zielsystems. Sie unterstützt die Anforderungsermittlung und ihre Aufbereitung zur Realisierung in Hardware und Software [Bort94].

Aus der obigen Definition ist ersichtlich, dass zur Realisierung einer qualitativ hochwertigen Systemspezifikation ein massiver Einsatz moderner CASE Tools notwendig ist, um beispielsweise die funktionalen Anforderungen frühzeitig modellieren und (halb-)automatisch validieren zu können.

Neben der Verbesserung der Anforderungsdokumente ist ein hohes Maß an Wiederverwendbarkeit von existierenden SW-Komponenten, wie sie gerade durch die Vererbung in der Objektorientierung unterstützt wird, notwendig. Wiederverwendbarkeit bedeutet dabei, dass bestehende SW-Komponenten, die in einem bestimmten Kontext bereits eingesetzt wurden, in einem neuen Bereich ebenfalls verwendet werden können. Darüber hinaus ermöglicht die Vererbung die Variantenbildung von SW-Komponenten. Ziel der Varianten-

bildung ist es, aufbauend auf bestehenden SW-Komponenten neue SW-Funktionen zu definieren, wobei einige Details geändert oder ergänzt werden. Aus diesen Gründen hat der Einsatz von objektorientierten Methoden und Techniken einen großen Vorteil bei der Erstellung komplexer SW-Strukturen.

Eine weitere Maßnahme zur Lösung der Problematik ist die Verwendung eines standardisierten und zugleich flexiblen Vorgehensmodells, mit dem die Aktivitäten und die Art der Ergebnisse größtenteils feststehen aber an spezifische Gegebenheiten angepasst werden können. Ein derartiges Vorgehensmodell ist beispielsweise das V-Modell [DW00], das in Deutschland einen sehr breiten Anwenderkreis aufweist und in der Automobilindustrie eine immer größere Bedeutung erlangt.

Die genannten Maßnahmen zur Lösung der Problematik sind bekannte und erprobte Ansätze aus dem Bereich Systems Engineering [Spre96] und bilden daher den Ansatz vieler Arbeiten im Bereich der Software- und Systementwicklung im Automobilumfeld. Die hier vorgestellte modellbasierte Entwurfsmethodik basiert ebenfalls auf diesen Konzepten, differenziert sich jedoch an vielen Stellen von anderen Arbeiten, um beispielsweise bislang nicht beantwortete Problemstellungen aufzugreifen. Der Mehrwert und die Unterschiede zwischen der hier vorgestellten Entwurfsmethodik und einigen bekannteren Arbeiten mit ähnlichem Forschungsschwerpunkt werden im folgenden Abschnitt erläutert und machen zugleich die Notwendigkeit für einen neuen Ansatz in der Entwicklung von Software im Automobilbereich deutlich.

1.1.3 Existierende Forschungsprojekte

Die Automobilindustrie hat erkannt, dass zur Lösung der Problematik moderne werkzeuggestützte Entwicklungsmethoden notwendig sind. Aus diesem Grund entstanden in den letzten Jahren viele derartige Methoden unabhängig voneinander, die in bestehende Entwicklungsprozesse integriert wurden. Einige Arbeiten auf diesem Gebiet werden in [Müll99b, BBK98] vorgestellt und miteinander verglichen. Dabei ist festzustellen, dass die meisten dort untersuchten Methoden, wie *CARTRONIC*, *BMW-ROOM*, *KorSys* und *BASEMENT*, nur einen bestimmten Teilbereich des Entwicklungsprozesses abdecken.

Zur Beherrschung der Komplexität ist jedoch eine ganzheitliche Betrachtung des Entwicklungsprozesses erforderlich. Denn gerade die nicht vorhandenen methodischen und werkzeugtechnischen Schnittstellen zwischen den Entwicklungsphasen verursachen einen Bruch in dem gesamten Vorgehen und führen somit nur zu suboptimalen Resultaten. Nach [BBK98] sind durchgängige Ansätze kaum vorhanden, da sie nur schwer zu realisieren sind. Ein durchgängiger Ansatz zur Entwicklung von Steuergeräte-Software wurde in dem Projekt *Strukturierter Entwicklungsprozess für Anwendungen im Automobilbereich* (STEP-X) realisiert. Dabei wurde die im weiteren Verlauf dieser Arbeit vorgestellte Entwurfsmethodik in STEP-X integriert und so auf Realisierbarkeit und Effizienz überprüft.

Nachfolgend wird ein kurzer Überblick über das STEP-X Projekt gegeben. Des Weiteren werden andere Arbeiten mit ähnlichem Forschungsschwerpunkt aufgezeigt.

STEP-X

Im April 2004 wurde das dreijährige STEP-X Kooperationsprojekt zwischen der Volkswagen AG und der Technischen Universität Braunschweig erfolgreich abgeschlossen. Das Ziel von STEP-X war die Realisierung eines durchgängigen modellbasierten Entwicklungsprozesses für den Automobilbereich. Dabei wurden sowohl alle Entwicklungsphasen, von der Anforderungsbeschreibung über die grafische Spezifikation bis hin zur automatischen Codegenerierung, als auch die dazugehörige Qualitätssicherung berücksichtigt. In den Entwicklungsprozess, der sich am V-Modell orientiert (vgl. Abbildung 1-2), wurden auch die beiden Themenbereiche Test und Diagnose einbezogen, um die Anforderungen an die Qualitätssicherung erfüllen zu können.

Um die gesamte Vorgehensweise im STEP-X Projekt beherrschbar zu machen, wurden orientiert am Vorgehensmodell die fünf Arbeitsgruppen *Digitales Lastenheft*, *Steuergeräte/Busse*, *Test*, *Diagnose* und *Toolkopplung* gebildet, die wiederum durch drei Institute der Technischen Universität Braunschweig geleitet wurden.

Das *Institut für Programmierung und Reaktive Systeme* (ips[3], FB Informatik) war für die Erstellung eines Digitalen Lastenheftes und für die Definition einer modellbasierten Entwurfsmethodik sowie für den dazugehörigen Entwicklungsprozess verantwortlich. Die in dieser Arbeit vorgestellte Entwurfsmethodik bildete die Kernaufgabe dieser Arbeitsgruppe. Die Implementierung der Software und die anschließende Integration auf die Hardware erfolgten durch die Arbeitsgruppe Steuergeräte/Busse.

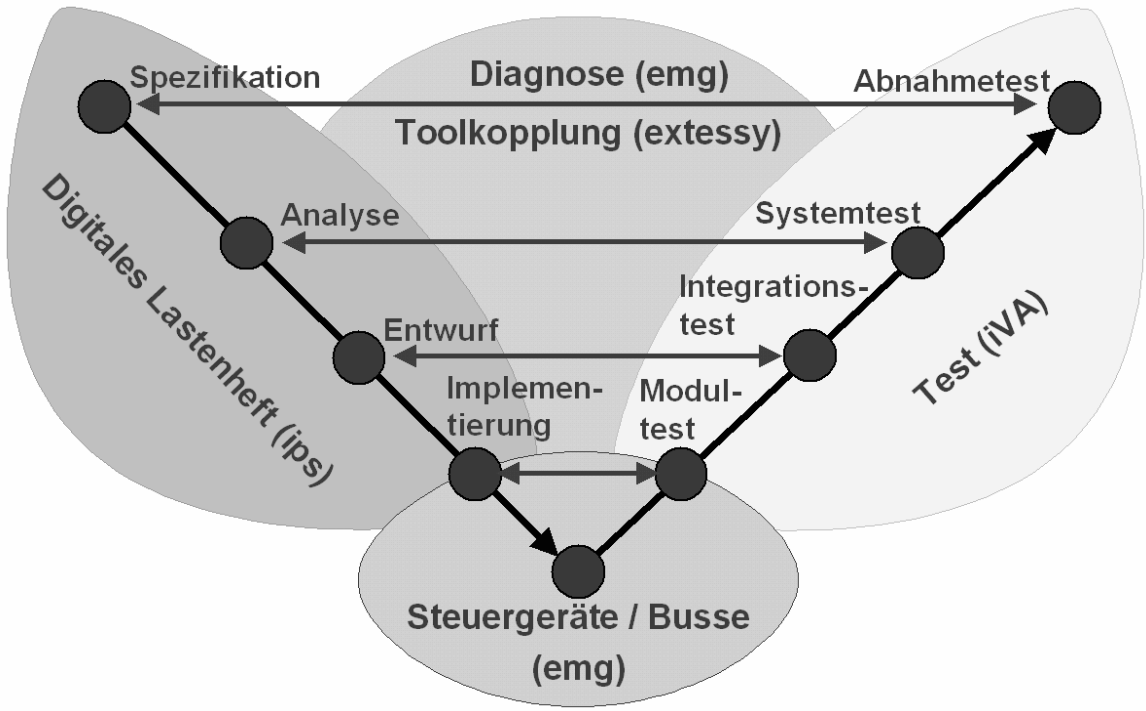


Abbildung 1-2: Zuständigkeiten der Arbeitsgruppen im V-Modell

[3] Vgl. http://www.cs.tu-bs.de/ips

Das Erkennen auftretender Fehlerzustände im laufenden Betrieb sowie das einwandfreie Identifizieren der betroffenen HW-Komponenten wurden durch die Arbeitsgruppe Diagnose unterstützt. Die beiden letztgenannten Arbeitsgruppen wurden durch das *Institut für Elektrische Messtechnik und Grundlagen der Elektrotechnik* (emg[4], FB Elektrotechnik) geleitet. Parallel zu der SW-Entwicklung wurde vom *Institut für Verkehrssicherheit und Automatisierungstechnik* (iVA[5], FB Maschinenbau) ein Testverfahren entworfen, das Tests auf unterschiedlichen Abstraktionsniveaus ermöglicht. Um die gewünschte Durchgängigkeit im gesamten Entwicklungsprozess zu gewährleisten, war sowohl eine vertikale als auch eine horizontale Kopplung der unterschiedlichen Werkzeuge im V-Modell notwendig. Die Durchführung dieser Tool-Integration erfolgte durch die Firma Extessy AG[6].

Ziel des derzeit laufenden Nachfolgeprojektes STEP-X II ist es, die Arbeiten der Arbeitsgruppen Test und Diagnose weiter auszubauen sowie die Funktionsverteilung auf logische Steuergerätenetzwerke zu optimieren und die logische HW-Architektur zu bewerten. Dazu wurde das STEP-X Team um das *Institut für Datentechnik und Kommunikationsnetze* (iDA[7], FB Elektrotechnik) erweitert, das mit dem Institut für Programmierung und Reaktive Systeme die Arbeitsgruppe *Entwicklungsmethodik und Architektur* bildet. Auf die Arbeit von STEP-X II soll an dieser Stelle jedoch nicht näher eingegangen werden, da sich das Projekt zurzeit in der Entwicklungsphase befindet. Die Projektlaufzeit endet im April 2006.

FORSOFT II – Automotive

1998 wurde das Projekt FORSOFT II – Automotive gestartet, welches zum Ziel hatte, einen modellbasierten Entwicklungsprozess für die frühen Phasen der Systementwicklung zu definieren und eine dazu geeignete durchgängige Werkzeugkette zu realisieren [BBEF+98]. Durchgeführt wurde das Projekt von der Technischen Universität München, den Automobilherstellern BMW AG und Adam Opel AG, den Zulieferern Robert Bosch GmbH und ZF Friedrichshafen AG sowie den Werkzeugherstellern ETAS GmbH und Telelogic GmbH.

Zum Erreichen der Ziele wurde das Projekt in drei Arbeitspakete unterteilt: Im ersten Arbeitspaket wurde eine Methode für das Requirements Engineering erarbeitet, indem die textuellen Anforderungen durch die standardisierte Beschreibungssprache *Unified Modeling Language* (UML) präzisiert wurden. Aufgrund der Mächtigkeit der UML wurde diese auf die im Automotive-Umfeld nötigen Notationen beschränkt. Beispielsweise wurden zur Beschreibung des dynamischen Verhaltens in der Analysephase nur UML-Sequenzdiagramme verwendet. Ein langfristiges Ziel von FORSOFT II – Automotive ist es, diese Methode zur Spezifikation von Anforderungen unter dem Namen *Automotive Modeling Language* (AML) für die Entwicklung im Automobilbereich zu standardisieren [BRS00]. Das zweite Arbeitspaket umfasste die Methodik des Requirements Management, welche

[4] Vgl. http://www.emg.ing.tu-bs.de
[5] Vgl. http://www.iva.ing.tu-bs.de
[6] Vgl. http://extessy.be-efficient.de
[7] Vgl. http://www.ida.ing.tu-bs.de

durch den Einsatz des Werkzeuges DOORS (vgl. Kapitel 3.2.3 auf Seite 54) unterstützt wurde. Das dritte Arbeitspaket beschäftigte sich mit der Integrität der verwendeten Werkzeuge. Dabei wurden die Schnittstellen zwischen DOORS und dem für die Analyse eingesetzten Werkzeug UML Suite der Firma Telelogic sowie dem für die Spezifikation verwendeten Werkzeug ASCET-SD der Firma ETAS [ETAS00] vereinheitlicht, um eine Werkzeugkopplung zu ermöglichen. Durch Vereinheitlichung sämtlicher Informationen des Funktions- und Steuergerätenetzwerks und der zentralen Datenhaltung wurde ein phasenübergreifender Einsatz der Entwicklungswerkzeuge in den frühen Phasen ermöglicht.

CARTRONIC

CARTRONIC wurde 1997 von der Firma BOSCH als Konzept für die Modellierung von Steuerungs- und Regelungssystemen in Kraftfahrzeugen entwickelt [LFSK+00]. Es basiert auf einem angepassten V-Modell und verfolgt eine hierarchische Dekomposition eines Gesamtsystems in mechatronische Subsysteme, die jeweils durch eine Kombination verschiedener technischer bzw. physikalischer Prinzipien realisiert werden.

Zur Beschreibung dieses Vorgehens werden Strukturen und Verhalten benötigt. Die Strukturbeschreibung, die das wesentliche Gerüst für den weiteren Entwicklungsprozess darstellt, erfolgt zunächst durch die hierarchische und modulare CARTRONIC-Funktionsstruktur. Eine solche Struktur repräsentiert eine funktionsorientierte Sicht auf die Systemarchitektur des Gesamtfahrzeugs. Unter Hinzunahme von nicht-funktionalen Anforderungen wird diese Struktur zu einer vollständigen Domänenarchitektur, die die Skalierbarkeit, Wiederverwendung und Austauschbarkeit von Systemkomponenten unterstützt. Die Erstellung der Funktionsstrukturen basiert auf den Strukturierungsregeln, welche Muster für die Lösung mehrfach vorkommender ähnlicher Problemstellungen darstellen. Beispielsweise existieren Muster für die Bereitstellung von fahrzeugglobalen Systemzuständen. Das Bindeglied zwischen der Anforderungs- und der Entwurfsphase ist die Abbildung der Domänenarchitektur in CARTRONIC UML-Modelle, in denen die Präzisierung der Funktionskomponenten mit ihren Kommunikationsbeziehungen erfolgt.

Derzeit existiert kein speziell entwickeltes Werkzeug zur Unterstützung des Konzepts. Das Prinzip kann aber grundsätzlich mit gängigen UML-Werkzeugen genutzt werden, um z. B. eine objektorientierte Klassenhierarchie für die zu modellierende Funktion aufzubauen. Für die Realisierung der Fahrzeugfunktionen wird das Werkzeug ASCET-SD vorgeschlagen.

COMTESSA

COMTESSA war ein vom Wirtschaftsministerium Baden-Württemberg gefördertes Forschungsprojekt zur Einführung und Optimierung von CASE-Technologie im durchgängigen SW-Entwurfsprozess von elektronischen Steuergeräten im Automobilbereich [MS00]. Die im Rahmen des Projektes entwickelte Methodik umfasst sowohl die Unterstützung von frühen Entwurfsphasen als auch den Einsatz von Werkzeugen im Bereich Modellierung, Simulation und Code-Generierung.

Der Fokus in COMTESSA war die Einführung von Verfahren und Methoden, die einen durchgängigen SW-Entwicklungsprozess nachhaltig unterstützen, frühzeitige Validierung ermöglichen und fehlerträchtige und aufwändige Routinetätigkeiten in den Implementierungsphasen automatisieren, ohne dabei die Echtzeiteigenschaften und die Ressourceneffizienz der so entwickelten Systeme zu beeinträchtigen.

In der ersten Phase des Projektes wurden zunächst kommerzielle Modellierungswerkzeuge evaluiert und für einen eigenen Entwicklungsprozess geeignete CASE Tools ausgewählt. Kriterien waren dabei unter anderem die Domänen- und Ebenenabdeckung unter Berücksichtigung des V-Modells, der Umfang der bereitgestellten Sprachelemente, die Benutzerfreundlichkeit sowie die Möglichkeiten weitere Werkzeuge in den Prozess zu integrieren. Im weiteren Projektverlauf wurde eine Integrationstechnologie auf Basis einer proprietären Sprache entwickelt, die es ermöglicht, ein System mittels mehrerer Entwurfswerkzeuge zu entwickeln, die Modelle anschließend zusammenzuführen und kommerzielle Code-Generatoren für die Code-Erzeugung zu nutzen. Zusätzlich wurde zu einem durchgängigen Entwicklungsprozess die Integration weiterer modellbasierter Werkzeuge, z. B. zur modellbasierten Diagnose, unterstützt.

Vergleicht man die einzelnen Projekte, wie in Tabelle 1-1 abgebildet, so ist festzustellen, dass keine der vorhergehenden Methoden, außer STEP-X, den gesamten Entwicklungsprozess von der Anforderungsbeschreibung über Analyse und Entwurf bis hin zur Implementierung und zum Testen abdeckt.

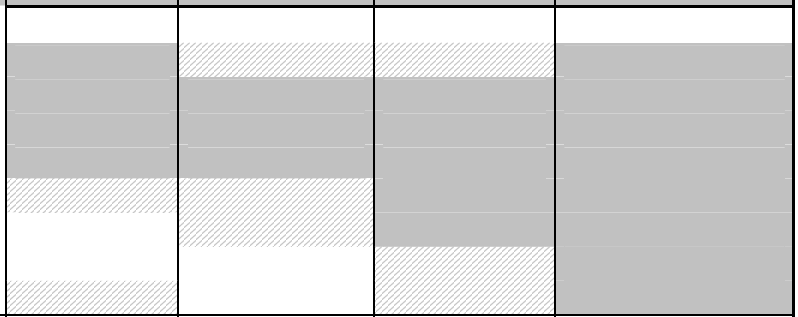
	FORSOFT II	CARTRONIC	COMTESSA	STEP-X
Entwicklungsphasen				
Anforderungsbeschreibung				
Analyse				
Systementwurf				
Grobentwurf				
Feinentwurf				
Implementierung				
Integration				
Test				
Durchgängige Werkzeugkette	ja	nein	ja	ja
Modellierungstechniken	UML, AML	Fkt-Strukturen, UML	Statecharts, Blockdiagramme	Blockdiagramme, UML
Tools	DOORS, ASCET-SD	Rational Rose, ASCET-SD	Statemate, Matlab, Rodon	DOORS, Artisan, ExITE, MATLAB, CTE XL, Rodon
formale Verifikation	nein	nein	nein	bedingt
Sicherheitskritische Systeme	bedingt	bedingt	bedingt	bedingt
HMI	nein	nein	nein	ja
Codegenerierung	nein	nein	ja	ja
Verteilungsaspekte (SW/HW)	ja/nein	ja/nein	nein	ja/nein
Werkzeugentwicklung	ja	nein	ja	ja
Validierung (Simulation)	ja	nein	ja	ja
Co-Simulation	nein	nein	nein	ja
Hardware-in-the-Loop	nein	nein	nein	ja

Tabelle 1-1: Entwicklungsprojekte im Automobilbereich

Darüber hinaus kann festgehalten werden, dass sich die Methoden, außer bei STEP-X, auf unterschiedliche Phasen konzentrieren, welches wiederum Auswirkungen auf die Auswahl der Beschreibungssprachen und Werkzeuge hat. In STEP-X dagegen wurde eine Entwurfsmethodik erarbeitet, die alle Phasen des V-Modells gleichermaßen abdeckt.

1.2 Zielsetzung der Arbeit

Wie bereits in den vorherigen Kapiteln beschrieben, reichen einzelne Methoden zur Lösung bestimmter Probleme, wie in vielen Projekten realisiert, nicht mehr aus. Vielmehr ist zur Beherrschung der heutigen SW-Komplexität im Automobilbereich ein durchgängiger Entwicklungsprozess notwendig, der sowohl werkzeuggestützte Methoden zur Erstellung als auch automatisierte Verfahren zur Überprüfung der Software bereitstellt.

Ziel der vorliegenden Arbeit ist es daher, eine werkzeuggestützte Entwurfsmethodik und einen entsprechenden durchgängigen Entwicklungsprozess für den Automobilbereich vorzustellen. Gleichermaßen wird die Entwicklung der Methodik an einigen Stellen der Arbeit exemplarisch aufgezeigt. Die Ergebnisse der Arbeitsgruppe *Digitales Lastenheft*[8] bilden das Grundgerüst der hier vorgestellten Methodik, wobei identifizierte Schwachstellen, beispielsweise die unzureichende Integration des V-Modells, behoben und einzelne Modellierungsprozesse optimiert wurden. Bei der Entwicklung der Methodik mussten folgende Anforderungen berücksichtigt werden:

- ***Schaffung eines durchgängigen Entwicklungsprozesses:*** Das Verfahren soll eine ganzheitliche Sicht einer SW-Entwicklung entlang des V-Modells darstellen. Um eine durchgängige Entwicklung gewährleisten zu können, muss durch Konzepte der Werkzeugkopplung ein phasenübergreifender Datenaustausch ermöglicht werden.
- ***Erstellung eines Digitalen Lastenheftes:*** Ein Digitales Lastenheft soll eine strukturierte, eindeutige und vollständige Beschreibung sowohl softwarespezifischer als auch hardwarespezifischer Systemkomponenten ermöglichen. Dabei sollen die textuellen Funktionsbeschreibungen bisheriger Lastenhefte weitestgehend durch eine graphische Darstellung präzisiert werden. Der Informationsgehalt soll dabei nicht nur auf die reine Funktionalität beschränkt sein, sondern auch nichtfunktionale Anforderungen wie Zuverlässigkeit und Sicherheit erfassen können. Das Digitale Lastenheft soll für den gesamten Entwicklungsprozess eine zentrale Datenbasis darstellen, so dass beispielsweise auch Modellierungsrichtlinien, Gesetzestexte, Testfälle und andere Artefakte aller Entwicklungsphasen beinhaltet werden können.
- ***Verwendung standardisierter Beschreibungsmittel:*** Im Bereich des objektorientierten Entwurfs hat sich in den letzten Jahren die graphische Beschreibungssprache Unified Modeling Language in vielen Bereichen des SW-Engineerings durchgesetzt. Es ist zu prüfen, inwieweit sich diese objektorientierte und standardisierte

[8] Der Autor dieser Arbeit war Leiter der Arbeitsgruppe *Digitales Lastenheft* und somit maßgeblich für die Konzeption und Realisierung der Entwurfsmethodik verantwortlich.

Sprache als wesentliche Säule Digitaler Lastenhefte eignet bzw. ob sie um andere Beschreibungsmittel ergänzt werden muss.

- ***Durchführung von Qualitätssicherungsmaßnahmen:*** Um die SW-Qualität zu wahren, sollen Modellierungsrichtlinien als konzeptionelle Maßnahmen entworfen und in einem Regelkatalog zusammengefasst werden. Dadurch erhält der Entwickler eine dokumentierte Vorgehensweise zur Erstellung qualitativ hochwertiger Modelle. Darüber hinaus soll ein analytisches Verfahren zur automatischen Überprüfung der Einhaltung dieser Regeln entstehen und an einer Fallstudie erprobt werden.
- ***Definition und Erprobung der modellbasierten Methodik:*** Nach der Festlegung der Vorgehensweise und der Bildung einer geeigneten Tool-Kette soll die Tauglichkeit der Methodik und des Werkzeugeinsatzes erprobt werden. Die Erprobung soll anhand eines prototypischen Reengineerings eines Kfz-Komfortsystems des VW Polos mit den Subsystemen Zentralverriegelung, Fensterheber und Spiegelsteuerung erfolgen.

Die im Rahmen dieser Arbeit vorgestellte Entwurfsmethodik soll in erster Linie die Entwicklung verteilter, reaktiver Steuerungssysteme mit weicher Echtzeit im Automotive-Umfeld gewährleisten. Dabei wird unter einem reaktiven System folgendes verstanden:

Reaktives System

Ein reaktives System reagiert während seiner Ausführung kontinuierlich auf zeitlich nicht vorhersagbare Ereignisse aus seiner Umgebung. Derartige Systeme weisen häufig einen hohen Grad an Nebenläufigkeit auf, da zum einen die Teilsysteme orthogonal ausgeführt werden und zum anderen das Gesamtsystem parallel zu seiner Umgebung abläuft.

Mit der Entwurfsmethodik können beispielsweise Systeme aus dem Bereich Body-Elektronik entwickelt werden. Für sicherheitskritische Systeme ist die Entwurfsmethodik nicht geeignet, da hierfür formale Überprüfungsmechanismen, wie das Model Checking notwendig sind [Peuk97]. Aus Gründen der nicht kompatiblen Schnittstellen zwischen den eingesetzten Modellierungswerkzeugen und vorhandenen Model Checkern ist eine diesbezügliche Werkzeugkopplung sehr aufwendig und daher im Automotive-Bereich derzeit nur schwer durchführbar [MD02].

1.3 Kapitelübersicht

Die vorliegende Arbeit gliedert sich in folgende Kapitel:

Kapitel 2 vermittelt Grundlagen der modellbasierten SW-Entwicklung. Dabei werden zunächst verschiedene Sprachansätze zur Beschreibung von SW-Systemen aus Sicht der modellbasierten Entwicklung diskutiert und die standardisierte Beschreibungssprache UML detailliert vorgestellt. Der letzte Abschnitt stellt Anforderungen an CASE-Werkzeuge für die SW-Entwicklung im Automobilbereich dar.

Kapitel 3 gibt einen Überblick über die Organisationsform und Vorgehensweise des V-Modells 97, das als Vorgehensmodell für die Entwurfsmethodik dient. Abschließend werden die projektspezifischen Anpassungen des Vorgehensmodells erläutert sowie die ausgewählten CASE Tools beschrieben.

In *Kapitel 4* erfolgt die Beschreibung der Entwurfsmethodik anhand ihrer Phasen Anforderungsbeschreibung, Analyse, Grobentwurf und Feinentwurf. Die Implementierungsphase sowie der gesamte rechte Ast des V-Modells bzw. die Integration und Testdurchführung stehen dagegen nicht im Fokus dieser Arbeit.

Parallel zum Entwicklungsprozess bilden im STEP-X Projekt konstruktive und analytische Maßnahmen die Qualitätssicherung. Dabei ist die wichtigste konstruktive Maßnahme ein Modellierungsrichtlinienkatalog, mit dem die Entwicklung qualitativ hochwertiger Modelle ermöglicht wird. Zur Durchführung der analytischen Maßnahmen wurde ein prototypisches Werkzeug entwickelt, mit dem die Einhaltung der Modellierungsregeln automatisch überprüft werden kann. Sowohl der Richtlinienkatalog als auch das Analysewerkzeug werden in *Kapitel 5* detailliert vorgestellt.

Kapitel 6 fasst die erarbeitete Entwurfsmethodik zusammen und bewertet diese mitsamt dem dazugehörigen Entwicklungsprozess, der Werkzeugkette und der eingesetzten Beschreibungssprache. Abschließend erfolgt ein kurzer Ausblick.

Kapitel 2

2 Grundlagen der modellbasierten Software-Entwicklung

„Siehe, es ist einerlei Sprache unter ihnen, nun wird ihnen nichts mehr verwehrt bleiben können von allem, was sie sich zu tun vorgenommen haben.“

1. Mose 11.6 (Turmbau zu Babel)

Eine Voraussetzung für die modellbasierte SW-Entwicklung sind geeignete Beschreibungssprachen. Daher wird in diesem Kapitel zunächst ein Überblick über die wichtigsten Spezifikationssprachen gegeben. Daraufhin erfolgt eine vertiefte Betrachtung der *Unified Modeling Language* (UML), die im Zusammenhang mit der objektorientierten Entwicklung derzeit die bedeutendste Beschreibungssprache darstellt [NHF98].

Zur Umsetzung der Beschreibungssprachen im Rahmen eines werkzeuggestützten Entwicklungsprozesses sind unterschiedliche CASE-Werkzeuge notwendig. Letztere unterstützen darüber hinaus weitere, für den SW-Entwicklungsprozess wichtige Aufgaben, wie z. B. Modellüberprüfung, Modellsimulation und automatische Codegenerierung für verschiedene Zielplattformen. In diesem Kapitel werden neben Anforderungen an die einzusetzenden Werkzeuge auch unterschiedliche Kategorien von CASE Tools dargestellt.

2.1 Beschreibungssprachen

Während traditionelle Ingenieurwissenschaften wie z. B. Maschinenbau und Elektrotechnik eine kleine Menge von Standardnotationen benutzen, steht den SW-Entwicklern eine breite Auswahl von Beschreibungssprachen zur Verfügung. Nachfolgend werden die für das SW-Engineering wichtigsten Sprachansätze kurz vorgestellt.

2.1.1 Der natürlichsprachliche Ansatz

Der *natürlichsprachliche Ansatz* ist rein textuell und wird daher in der Software- und Systementwicklung am häufigsten zur Beschreibung von Anforderungen verwendet. Der Vorteil dieses Ansatzes ist, dass er nicht zusätzlich erlernt werden muss und von allen beteiligten Personen im Projekt gleichermaßen beherrscht wird. Zudem sind natürlichsprachliche Beschreibungen in der Regel einfach und schnell zu erstellen [Joos00].

Diese Vorzüge können jedoch nicht darüber hinwegtäuschen, dass die Verwendung dieser Spezifikationssprache sehr problematisch ist. Gründe hierfür liegen unter anderem im informalen Charakter einer natürlichen Sprache, welcher zu Interpretierungsdifferenzen und Inkonsistenzen führen kann. Verschiedene Konzepte zur Behebung dieser Probleme werden in [Rupp00] diskutiert. Die Umsetzung einiger solcher Konzepte, wie z. B. *semantische* und *linguistische Analysen* oder die Verwendung von *Anforderungsschablonen*, wurde für das STEP-X Projekt in [GHMP02] realisiert.

Trotz dessen sind auch nach obigen Anpassungen immer noch wesentliche Schwachpunkte des natürlichsprachlichen Ansatzes ersichtlich. Beispielsweise lassen sich die verschiedenen Detaillierungsgrade der Systemfunktionalität nur sehr schwer realisieren. Darüber hinaus können die natürlichsprachlichen Anforderungen in späteren Entwurfsphasen nicht wieder verwendet, ausgeführt oder formal verifiziert werden.

Daher wurde der natürlichsprachliche Ansatz durch *grafische Beschreibungssprachen* ergänzt. Grafische Beschreibungssprache bedeutet dabei, dass die grundlegende Struktur grafisch beschrieben wird, wobei die Detailbeschreibung jedoch textuell erfolgt. Grafische Spezifikationssprachen lassen sich allgemein in die Kategorien[9] *funktionsorientierte, verhaltensorientierte* und *objektorientierte Ansätze* unterscheiden. Diese Ansätze orientieren sich an präzise vorgegebenen Paradigmen, welche syntaktische und semantische Regeln zur Strukturierung von Spezifikationen vorgeben.

2.1.2 Die funktionsorientierten Sprachansätze

Die *funktionsorientierten Sprachansätze* zerlegen ein Problem in eine Menge interagierender SW-Komponenten mit jeweils klar definierter Funktionalität. Zustand und Daten eines Systems werden im Wesentlichen zentral abgelegt und sind für andere Komponenten sichtbar und änderbar. Des Weiteren besitzen die Komponenten einen lokalen Zustandsraum, der jedoch nur innerhalb der Ausführung einer Komponente gültig ist und daher Informationen nur zeitlich begrenzt halten kann. Beispiele für funktionsorientierte Spezifikationssprachen sind *Structured Analysis and System Specification* [DeMa78] und *Structured Analysis and Design Technique* [Ross77].

2.1.3 Die verhaltensorientierten Sprachansätze

Bei *verhaltensorientierten Beschreibungssprachen* unterscheidet man zwischen diskreten und kontinuierlichen Ansätzen zur Verhaltensmodellierung.

Diskrete Ansätze stellen das zeitliche Verhalten eines Systems dar, indem aufzählbare, diskrete Zustände des Systems modelliert werden. Das Systemverhalten wird durch Zustandsübergänge spezifiziert, d. h. wann und unter welchen Bedingungen ein Zustandsübergang erfolgt und welche funktionalen Auswirkungen mit diesem Zustandsübergang verbunden sind. Beispiele für diskrete verhaltensorientierte Spezifikationssprachen sind *Statecharts* [Hare87], *Petrinetze* [Reis86] und die *Specification & Description Language*

[9] Eine detaillierte Klassifizierung von Spezifikationssprachen erfolgt in [Joos00].

(SDL) [SSR90]. Einen besonderen Stellenwert haben Statecharts, die daher gesondert in Kapitel 2.2.2.4 betrachtet werden.

Kontinuierliche Ansätze modellieren das zeitliche Systemverhalten, indem skalare Größen des Systems und die Wirkungen zwischen diesen Systemgrößen durch stetige Funktionen abgebildet werden. Um eingebettete SW-Systeme und ihre Umgebung so realistisch wie möglich zu simulieren, werden in der Praxis, wie auch im STEP-X Projekt beide Ansätze für die Beschreibung eines Systems verwendet. Man spricht dann von einem *hybriden System*, das wie folgt definiert ist.

Hybrides System

Ein System, das diskrete und kontinuierliche (analoge) Datenanteile verarbeitet und/oder sowohl über kontinuierliche Zeiträume, als auch zu diskreten Zeitpunkten mit seiner Umgebung interagiert [BBK98].

2.1.4 Die objektorientierten Sprachansätze

Objektorientierte Spezifikationssprachen beschreiben ein System durch eine Menge von Objekten, die mit Hilfe von Nachrichten miteinander kooperieren. Dabei ist ein *Objekt* wie folgt definiert:

Objekt

Ein Objekt ist ein individuelles Exemplar von Dingen, Personen oder Begriffen der realen Welt oder Vorstellungswelt [Balz01]. Es hat ein definiertes Verhalten (Operationen, Methoden), einen inneren Zustand (Attribute) und eine eindeutige Identität.

Durch das Zusammenführen von logisch zusammengehörigen Daten und Funktionen bilden Objekte hohe Kohäsionen. Eine Klasse beschreibt die Struktur und das Verhalten einer Menge gleichartiger Objekte durch Attribute und Operationen.

Die wichtigsten Prinzipien der Objektorientierung sind Kapselung, Vererbung und Polymorphie. Kapselung bedeutet, dass die Datenstruktur eines Objekts dem Nutzer verborgen bleibt und der Zugriff auf das Objekt nur über fest definierte Schnittstellen und Methoden möglich ist. Das Prinzip der Vererbung ermöglicht die Weitergabe von Daten und Methoden einer übergeordneten Klasse an andere Klassen. Durch Polymorphie können unterschiedliche Methoden mit denselben Methodennamen unterschiedliche Daten verarbeiten. Weitere Grundlagen zur Objektorientierung sind in [Balz96, Balz01, Oest98] nachzulesen.

Es existiert eine Vielzahl von Sprachen, die auf dem objektorientierten Paradigma aufsetzen [Ivan99]. Die verbreitetsten Vertreter dieses Sprachansatzes[10] sind die *Object Oriented Analysis* [CY91]*, Object Oriented Design with Applications* [Booc94]*, Object Oriented Technique* [RBPE+91]*, Object Oriented Software Engineering* [JCJ+92], *Real-Time Object Oriented Modeling* [SGW94] und die *Unified Modeling Language* [OMG03a]. Von besonderer Bedeutung ist die Unified Modeling Language als jüngste Beschreibungssprache. Sie wird im folgenden Kapitel näher betrachtet.

2.2 Die UML für die objektorientierte Software-Entwicklung

Die Unified Modeling Language ist eine standardisierte Beschreibungssprache, die für den gesamten Verlauf einer SW-Entwicklung einsetzbar ist. Sie bietet unterschiedliche Diagramme und Techniken zur Anforderungserfassung, Systemstruktur- und Verhaltensbeschreibung und spezifiziert implementierungsnahe Details. Sie wird von der *Object Management Group*[11] (OMG) wie folgt definiert:

Unified Modeling Language

Die Unified Modeling Language ist eine grafische Modellierungssprache zur Spezifikation, Visualisierung, Konstruktion und Dokumentation von SW-Systemen [OMG03a].

Nach der obigen Definition handelt es sich nicht um eine spezielle Vorgehensweise oder ein Prozessmodell, sondern um ein Konglomerat von grafischen Notationen mit den dazugehörigen semiformalen Semantiken. Diese Entscheidung wurde bewusst getroffen, um die Verwendbarkeit der UML zu erhöhen. Existierende Vorgehensmodelle wurden speziell an die UML angepasst, wie z. B. [AKZ96, DW98, Müll98, Hrus98, LFSK+00] sowie neue Vorgehensmodelle für den Einsatz der UML entwickelt [JBR99, Hunt03, MC00, Hörf02, WBP98].

In den folgenden Abschnitten wird einerseits die Historie der UML kurz wiedergegeben, andererseits werden die für die Entwurfsmethodik relevanten Diagramme näher erläutert. Abschließend werden Erweiterungen der UML dargestellt.

2.2.1 Entwicklung der UML

Nach dem Vormarsch der objektorientierten Programmiersprachen wurde in den letzten 15 Jahren versucht, die objektorientierten Ansätze auch für den SW-Entwicklungsprozess nutzbar zu machen. Es entstand eine Vielzahl von Methoden und Notationen für die Analyse und den Entwurf von objektorientierten SW-Systemen (vgl. Kapitel 2.1). Die Folge

[10] Diese Ansätze stehen stellvertretend für eine Vielzahl weiterer konzeptionell ähnlicher Ansätze. Ein Vergleich zwischen objektorientierten Ansätzen wird unter anderem in [Ste94] vorgenommen.

[11] Die OMG ist eine internationale Organisation mit über 600 Mitgliedern aus verschiedenen Bereichen wie System- und Softwareentwicklung. Vgl. http://www.omg.org.

war, dass bis 1994 über 50 objektorientierte Methoden erschienen [BRJ00], die sowohl in der Notation als auch in der Vorgehensweise mehr oder weniger stark voneinander abwichen. Dadurch wurde zum einen die modellbasierte SW-Entwicklung erschwert, zum anderen führten die vielfältigen Notationen zur Fragmentierung auf dem objektorientierten Werkzeugmarkt.

Die Standardisierung der Modellierungssprachen begann, als Grady Booch und James Rumbaugh ihre Entwicklungsmethoden *Object Oriented Design with Applications* (OOD) und *Object Management Technology* (OMT) bei der Firma Rational zusammenbrachten. 1995 war die gemeinsame Entwicklungsmethode *Unified Method v. 0.8* fertig gestellt. Ein Jahr später stieß Ivar Jacobson, der Entwickler von *Object Oriented Software Engineering* (OOSE), dazu und brachte die von ihm geprägten Anwendungsfälle mit. 1996 entstand die gemeinsame Modellierungssprache UML 0.9 [BJR96], in die weitere Modellierungsansätze (z. B. *Zustandsdiagramme* von Harel, *Aktivitätsdiagramme* von Martin & Odell oder *Stereotypen* von Wirfs-Brock) integriert wurden. Die stärkere Fokussierung auf die Syntax und Semantik und zugleich die fehlende Prozessunterstützung führten ab dieser Version zu der Namensänderung *Unified Modeling Language* [SW97]. Eine Übersicht über die historische Entwicklung der UML ist der Abbildung 2-1 zu entnehmen.

Im weiteren Verlauf errichtete Rational ein UML-Konsortium, das aus verschiedenen Organisationen[12] bestand, um eine einheitliche und komplette Modellierungssprache zu entwerfen. Im Januar 1997 wurde eine überarbeitete Version (UML 1.0 [OMG97a]) an die *Object Management Group* ausgehändigt, mit der Intention, diesen Vorschlag zu standardisieren. Im selben Jahr wurde die endgültige UML-Version 1.1 [OMG97b] bei der OMG eingereicht und als objektorientierter Modellierungsstandard verabschiedet. Die Weiterentwicklung wurde von der *OMG Revision Task Force* (RTF) vorgenommen. Eine rein redaktionelle Version 1.2 der UML ohne signifikante technische Veränderungen wurde 1998 herausgebracht.

Die RTF reagierte auf zahlreiche Änderungswünsche aus der Industrie und Wirtschaft und verbesserte beispielsweise die unvollständige Semantik der Aktivitätsdiagramme. Als Ergebnis kam die UML 1.3 [OMG99b] Mitte 1999 heraus. Dieser Standard wird als die „erste ausgereifte UML-Version“ (z. B. [Kobr99]) bezeichnet. Die UML-Version 1.4 [OMG01b] wurde im September 2001 verabschiedet und beinhaltet zahlreiche Verbesserungen und Erweiterungsmechanismen (vgl. Kapitel 2.2.3.2). Im Jahre 2003 kam die UML Version 1.5 [OMG03a] heraus. In dieser wurden unter anderem die *action language* und *action semantics* [OMG01a] aufgenommen, mit denen eine Modellierung unabhängig von der Implementierungssprache ermöglicht wird.

12 Digital Equipment Corporation, Hewlett-Packard, I-Logix, IBM, Microsoft, Oracle, Rational und Texas Instruments, um einige zu nennen.

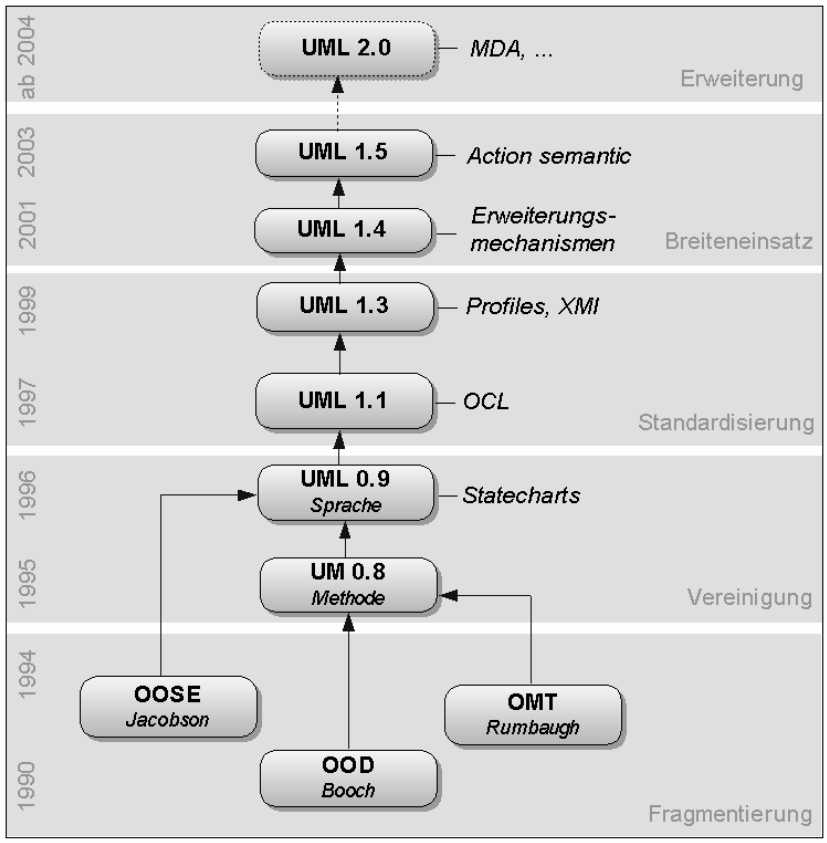


Abbildung 2-1: Entwicklung der UML

Infolge der Erfahrungen mit der UML 1.x wurde die Wunschliste für eine neue UML-Version immer länger. Da sich einige Anforderungen nicht sinnvoll in die aktuelle UML integrieren ließen, wurde entschieden, die UML von Grund auf neu aufzusetzen. Der am besten ausgereifte Änderungsvorschlag für die UML 2.0 wurde von den U2 Partners[13] eingereicht. Aus den vorliegenden Ideen entstanden im Wesentlichen die zwei sich ergänzenden und aufeinander verweisenden Dokumente *Infrastructure* [OMG03b] und *Superstructure* [OMG02b]. In der *Infrastructure* sind grundlegende Sprachkonstrukte und die Basisarchitektur festgehalten, wohingegen die *Superstructure* auf dieser Architektur aufbauende Diagrammnotationen und Semantik enthält.

In dieser Arbeit wurde auf die UML Version 1.4 zurückgegriffen, da zum Zeitpunkt des STEP-X Projektes keine aktuelleren UML-Werkzeuge zur Verfügung standen.

2.2.2 Sichten und Diagramme der UML

Bei der Modellierung großer SW-Systeme ist eine Unterteilung des Gesamtmodells in kleinere Submodelle ein erster Schritt zur Beherrschung der Komplexität. Des Weiteren ist es von Vorteil, wenn die zu entwickelnden Modelle nach verschiedenen Aspekten (z. B. Funktionalität, Kommunikation und Verteilung) modelliert werden können. Die UML be-

[13] Ein Konsortium, bestehend aus Alcatel, Computer Associates, Ericsson, Hewlett-Packard, IONA, Motorola, Oracle, Rational Software, SOFTEAM, Telelogic, Unisys und diversen unterstützenden Teilnehmern wie z. B. DaimlerChrysler.

zeichnet solche Aspekte als Sichten, die durch ein oder mehrere UML-Diagramme gleichen oder unterschiedlichen Typs repräsentiert werden. In der UML werden fünf unterschiedliche Sichten unterstützt (vgl. Tabelle 2-1).

Sicht	Diagramme	Beschreibung
Benutzer (statisch)	• Anwendungsfalldiagramme	Stellt eine grobe Systemfunktionalität aus Sicht des Benutzers dar.
Struktur (statisch)	• Klassendiagramme • Objektdiagramme • Paketdiagramme	Definiert die Verbindung zwischen verschiedenen statischen Teilen des Systems und beschreibt ihr Zusammenspiel.
Verhalten (dynamisch)	• UML Statecharts • Aktivitätsdiagramme	Beschreibt das gewünschte Verhalten des Systems oder einzelner Komponenten.
Interaktion (dynamisch)	• Sequenzdiagramme • Kollaborationsdiagramme	Beschreibt eine Folge von Botschaften zwischen Objekten.
Implementierung (statisch)	• Verteilungsdiagramme • Komponentendiagramme	Stellt den Bezug zwischen SW-System und HW-Plattformen dar.

Tabelle 2-1: Sichten und Diagramme der UML

Der Detaillierungsgrad sowie die Namenskonventionen für die UML-Sichten sind je nach Autor und Anwendungsgebiet unterschiedlich. So werden in einigen Literaturquellen und Vorlesungen zum Thema UML weitere Sichten hinzugefügt [BRJ00] oder abstrahiert [JRHZ+04]. Für eine rein qualitative Aussage genügt es häufig, die Sichten wie in Tabelle 2-1 in *statische* und *dynamische* Bereiche zu unterteilen. Im Folgenden werden nur diejenigen UML-Diagramme grundlegend vorgestellt, die für die Entwurfsmethodik in Kapitel 4 relevant sind. Für eine vollständige Beschreibung aller UML-Diagramme wird auf [EP98, HK99, Oest98, OMG03a] verwiesen.

2.2.2.1 Anwendungsfalldiagramme

Ein Anwendungsfalldiagramm (engl.: use case diagram) beschreibt hauptsächlich die Zusammenhänge zwischen einer Menge von Anwendungsfällen und den daran beteiligten Akteuren. Dabei können Akteure Personen, externe Systeme oder Hardware darstellen. Anwendungsfälle (vgl. Abbildung 2-2) beschreiben hauptsächlich die Funktionen der zu entwickelnden Software und deren Berührungspunkte zum Umfeld. Ein Rahmen um die Anwendungsfälle symbolisiert die Systemgrenzen. Mit der *include*-Beziehung lässt sich darstellen, dass innerhalb eines Anwendungsfalles ein anderer Anwendungsfall vorkommt. Die *extend*-Beziehung hingegen drückt aus, dass ein Anwendungsfall unter bestimmten Umständen durch einen weiteren erweitert wird.

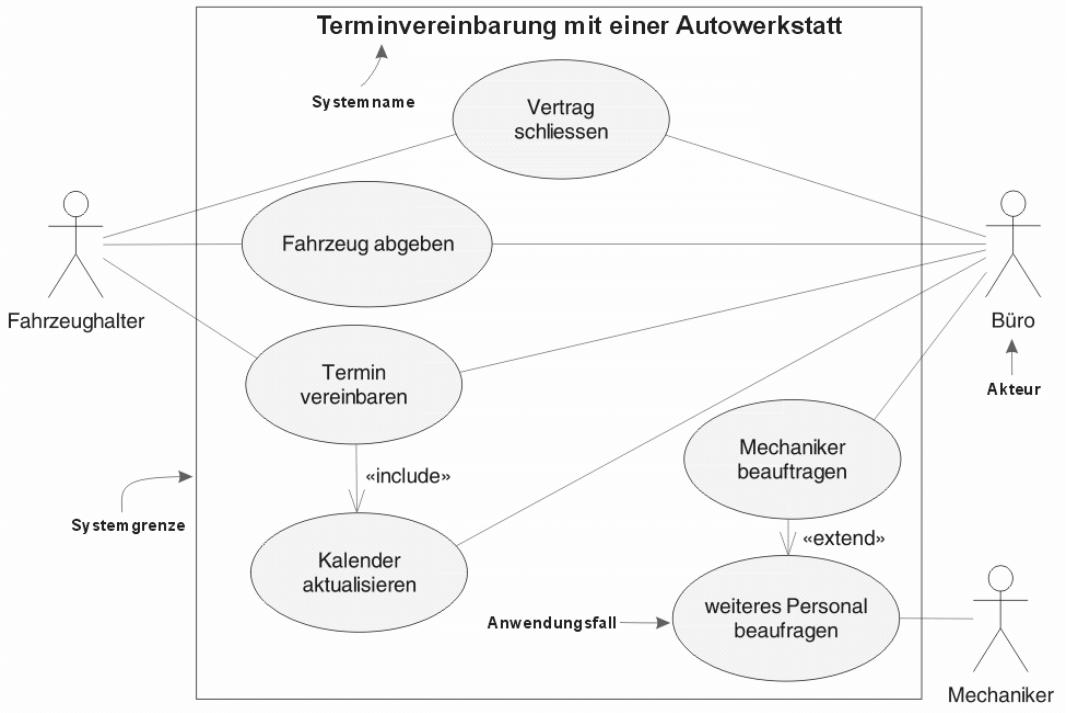


Abbildung 2-2: Ein Anwendungsfalldiagramm der UML

2.2.2.2 Klassendiagramme

Ein Klassendiagramm (engl.: class diagram) beschreibt mit Hilfe von Klassen und Beziehungen die statische Struktur des Systems. Eine Klasse besitzt im Regelfall Attribute (Daten) und Operationen (Verhalten). Mit Hilfe der Symbole + (public), – (private) und # (protected) werden die Zugriffsberechtigungen für Attribute und Operationen gekennzeichnet. Beispielsweise kann in Abbildung 2-3 auf das Attribut *AnzahlRäder* der Klasse *Fahrzeug* von außen nicht zugegriffen werden, da das Attribut als „private“ markiert worden ist. Bestimmte Klassenoperationen dienen als Kommunikationsmechanismen. Dabei wird eine Beziehung zwischen diesen Klassen vorausgesetzt.

Die UML stellt dafür verschiedene Arten von Beziehungen zur Verfügung:

- ***Assoziation:*** Eine Assoziation drückt eine allgemeine Beziehung zwischen Klassen aus und impliziert, dass die Klassen strukturell oder funktional zusammenhängen. Ein Austausch von Nachrichten ist dadurch grundsätzlich möglich. Eine Kardinalität an einer Assoziation wird in Form von Min – Max angegeben und begrenzt die Instanzbildung von Klassen. Zum Beispiel wird in Abbildung 2-3 dem Objekt der Klasse *Werkstatt* mindestens ein (1..*) Objekt *Mitarbeiter* zugeordnet. Im Gegensatz dazu wird dem Objekt der Klasse *Mitarbeiter* genau ein (1) Objekt der Klasse *Werkstatt* zugeordnet.
- ***Aggregation/Komposition:*** Eine Aggregation ist eine Spezialform der Assoziation, die eine Ganzes-Teile-Beziehung zwischen dem Aggregat und den Bestandteilen beschreibt. Eine solche Beziehung besteht beispielsweise zwischen den Klassen *Pkw* und *Rad*. Allerdings ist diese Beziehung nicht zwingend, denn ein Rad könnte auch losgelöst von einem Pkw betrachtet werden. Bei der Komposition

dagegen können die Bestandteile ohne das Aggregat nicht existieren. Wird beispielsweise die *Werkstatt* aufgelöst, verliert die *Abteilung* ihre Berechtigung (vgl. Abbildung 2-3).

- ***Generalisierung:*** Eine Generalisierung verbindet eine Oberklasse mit einer Unterklasse. Das bedeutet, dass eine Subklasse alle Attribute, Operationen und Assoziationen ihrer Oberklasse übernimmt und diese dann ergänzen kann. Zum Beispiel erbt die Klasse *Pkw* in der Abbildung 2-3 alle Eigenschaften von der Klasse *Fahrzeug*. D. h. die Klasse *Pkw* besitzt die Assoziation zur Klasse *Mitarbeiter*, auch wenn diese nicht explizit modelliert worden ist.

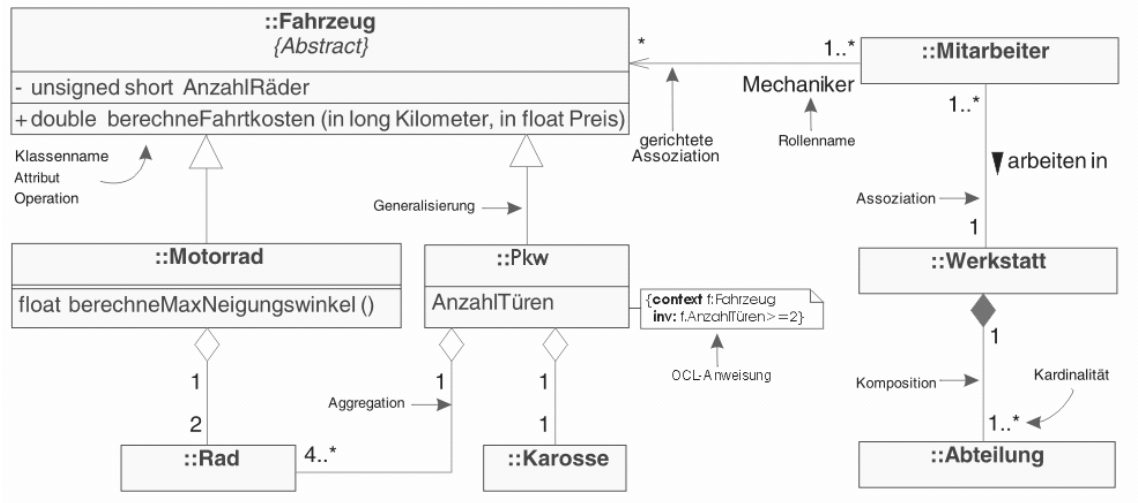


Abbildung 2-3: Ein Klassendiagramm der UML

Klassen bilden somit den Kern der UML sowie des objektorientierten Entwurfs. Ein SW-System hat typischerweise mehrere Klassendiagramme. Eine Klasse kann in unterschiedlichen Klassendiagrammen auftreten.

2.2.2.3 Objektdiagramme

Ein Objektdiagramm (engl.: object diagram) ist eine Variante des Klassendiagramms. Es stellt Objekte und ihre Relationen zu einem bestimmten Zeitpunkt so dar, dass die Attributwerte eines Objekts ersichtlich sind. Die Notation ähnelt den Klassendiagrammen mit der Ausnahme, dass jedes Objekt einer Klasse mit einem Namen gekennzeichnet wird und alle Instanzen einer Relation grafisch abgebildet werden.

Objektdiagramme haben in der objektorientierten Modellierung einen geringeren Stellenwert als Klassendiagramme. Daher werden diese beiden Notationen in einigen Werkzeugen zu einer gemeinsamen Diagrammart zusammengefasst. Als Beispiel sei hier das Tool Rhapsody (vgl. Kapitel 3.2.3 auf Seite 54) genannt, welches dafür den Diagrammbegriff *Object Model Diagram* definiert.

2.2.2.4 UML Statecharts

Statecharts wurden Mitte der 80er Jahre von Harel [Hare84, Hare87] als ein visueller Formalismus zur Beschreibung reaktiver Systeme vorgeschlagen und stellen eine Weiterentwicklung der allgemein bekannten endlichen Automaten (Zustandsdiagramme) dar. Der Statechart-Formalismus behebt zahlreiche Schwachpunkte der Zustandsdiagramme, die in [HPSS87] diskutiert werden. Die Statecharts ermöglichen unter anderem das Schachteln von Zuständen, Zusammenfassen von komplexen Transitionen und die Darstellung von Nebenläufigkeiten mit Hilfe der Broadcast-Kommunikation. Zusammenfassend können Statecharts wie folgt definiert werden:

Statecharts
Statecharts = Zustandsdiagramme + Tiefe + Orthogonalität + Broadcast-Kommunikation [Hare87]

Wegen der Einfachheit und Mächtigkeit dieser Beschreibungssprache, wurde das Statechart-Konzept sehr früh in die UML aufgenommen (vgl. Abbildung 2-1 auf Seite 21) und an das objektorientierte Paradigma semantisch angepasst. Ein Vergleich zwischen den Statecharts von Harel und den UML Statecharts wird unter anderem in [Proc03, OMG03a] vorgenommen.

Die Beschreibung der UML Statecharts erfolgt hier nur informell, da die UML-Spezifikation [OMG03a] für die Statecharts einigen Freiraum (engl.: semantic variation) lässt und somit nicht präzise ist. Eine formale Beschreibung von Statecharts ist beispielsweise aus [Beec01, BCR02, CIS00, HN96, HP96, JEJ02] zu entnehmen. Für den Einsatz von Statecharts in der Objektorientierung sei auf [CHB92, HG96] verwiesen.

Der strukturelle Aufbau eines UML Statecharts entspricht der Struktur von herkömmlichen Statecharts. Es besteht aus hierarchisch aufgebauten Zuständen und Transitionen.

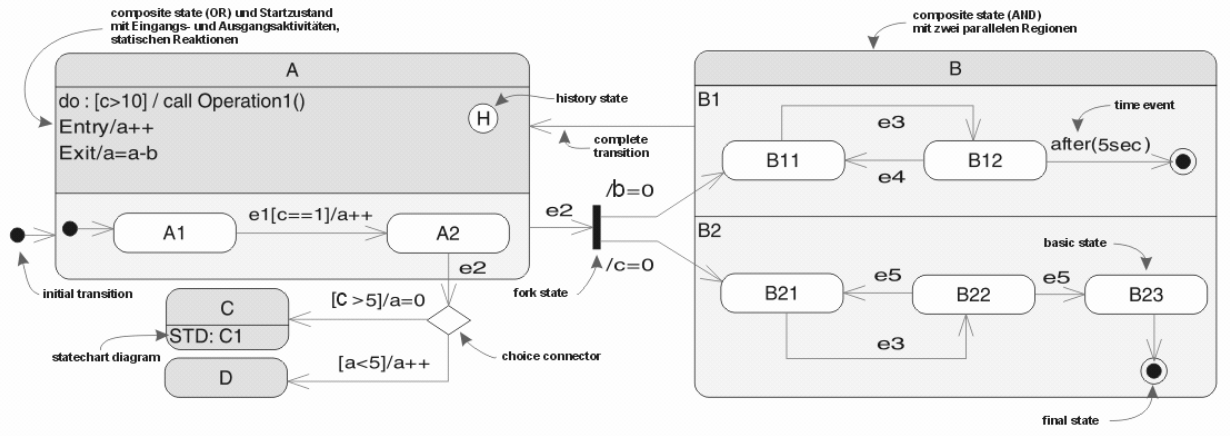


Abbildung 2-4: Ein Zustandsdiagramm der UML

Zustände

Jedes Zustandsdiagramm enthält eine endliche Menge von *Zuständen*, *Pseudozuständen*, einen *Startzustand* (engl.: initial state) und eine Menge von *Endzuständen* (engl.: final states). Das Eintreten in den Startzustand beschreibt den Zeitpunkt der Instanziierung eines Objekts, wohingegen das Betreten aller Endzustände in einem Statechart die Zerstörung des Objekts bedeutet. Der hierarchische Aufbau eines Statecharts wird durch eine Baumstruktur abgebildet[14]. Eine solche Struktur erlaubt eine schrittweise Verfeinerung eines komplexen Systems. Zustände können dahingehend verfeinert werden, dass sie Unterzustände enthalten. Dies ist bis zu einer beliebigen Verschachtelungstiefe möglich. Zustände, die Unterzustände enthalten, werden *zusammengesetzte Zustände* (engl.: composite states) genannt. *Basiszustände* (engl.: basic states) dahingegen besitzen keine Unterzustände. Ein Zustand kann entweder in nebenläufige Regionen (AND-Zustände) oder in sich gegenseitig ausschließende Unterzustände (OR-Zustände) aufgegliedert werden. Im Falle eines OR-Zustands[15] kann sich das System zu einem Zeitpunkt nur in einem Zustand pro Ebene befinden. Wird in der Abbildung 2-4 der Zustand *A* betreten, so ist nur einer der beiden Unterzustände *A1* oder *A2* aktiv. Zur Kennzeichnung eines Startzustands wird dieser mit einer *Initial-Transition* gekennzeichnet. Beim Verlassen des Oberzustands werden alle seine Unterzustände mitverlassen. Zustände können komplette Zustandsautomaten mit einem Startzustand und einem oder mehreren Endzuständen enthalten. Dies wird im Zustand *C* in der Abbildung 2-4 durch das Akronym *STD* (engl.: statechart diagram) gekennzeichnet.

Als Hilfsmittel wurden Pseudozustände eingeführt, mit denen komplexere Modellkonstrukte einfacher und übersichtlicher dargestellt werden können. Ein solcher Pseudozustand ist beispielsweise der *History-Konnektor* (engl.: history state). Mit ihm werden beim Wiedereintritt in einen zusammengesetzten Zustand alle diejenigen Unterzustände betreten, die vor dem Verlassen aktiviert waren. UML Statecharts kennen zwei Arten von History-Konnektoren: der „flache“ History-Konnektor (engl.: shallow history) bezieht sich nur auf die aktuelle Ebene. Der „tiefe“ History-Konnektor (engl.: deep history) beeinflusst zusätzlich alle darunterliegenden Ebenen. In der Abbildung 2-4 enthält der Zustand *A* einen flachen History-Konnektor. Beim Wiedereintritt wird somit nicht der Startzustand *A1* betreten, sondern der zuletzt gespeicherte Unterzustand, also *A1* oder *A2*.

Transitionen

Eine Transition verbindet genau zwei (Pseudo-)Zustände miteinander und ist für einen Zustandswechsel verantwortlich. Sie kann mit einem Ereignis *e*, einer Bedingung *c* und/oder einer Liste von Aktionen *a* in der Form *e[c]/a* beschriftet werden. Ein Zustandsübergang kann nur dann stattfinden, wenn das Ereignis triggert und die Bedingung an der Transition *wahr* ist. In der UML sind verschiedene Typen von Ereignissen vordefiniert. So sind beispielsweise synchrone (Methodenaufrufe), asynchrone (Signale aus der Umge-

[14] Der Ausgangszustand (Top-Level-Zustand) selbst wird üblicherweise nicht grafisch abgebildet.
[15] In einigen Literaturquellen wird deshalb von XOR-Zuständen gesprochen, da ein *logisches ODER* in diesem Zusammenhang eine falsche Interpretation erlaubt [John01, LL98, MK98].

bung) oder zeitbehaftete Ereignisse (Schalten nach Verstreichen einer Zeit) möglich. Mit dem Schalten werden alle Aktionen der Transition ausgeführt, wodurch beispielsweise Variablen geändert und neue Ereignisse ausgelöst werden können[16]. Anschließend wird der Zielzustand betreten. Im Allgemeinen kann der Quell- und Zielzustand auf unterschiedlichen Ebenen eines Statecharts liegen (Interlevel-Transition[17]).

UML Statecharts unterstützen drei Arten von Transitionen. *Einfache Transitionen* (engl.: simple transitions) führen direkt von einem Zustand in den anderen. *Zusammengesetzte Transitionen* (engl.: compound transitions) dagegen beinhalten zwei oder mehrere einfache Transitionen, die durch Pseudozustände verbunden sind. So führt beispielsweise eine zusammengesetzte Transition in der Abbildung 2-4 aus dem Zustand *A2* über einen Pseudozustand in die beiden Zustände *C* und *D*. Im Gegensatz zu „normalen" Zuständen darf in Pseudozuständen nicht verweilt werden. Ein Schalten ist daher nur möglich, wenn alle einfachen Transitionen einer zusammengesetzten Transition aktiviert sind. Bei der dritten Art der Transition handelt es sich um die so genannten *abschließenden Transitionen* (engl. completion transitions), die mit keinem expliziten Trigger-Ereignis beschriftet sind. Sie werden erst dann getriggert, wenn ein oder mehrere Endzustände betreten werden. Des Weiteren können zur besseren Übersicht Aktionen an einen Zustand gebunden werden, wie bereits von Moore-Automaten bekannt. Die Zustände können so über drei interne Aktionsarten verfügen. Die Eingangsaktion (engl.: entry action) wird erst dann ausgeführt, wenn der Zustand betreten wird. Beim Verlassen eines Zustands wird die Ausgangsaktion (engl.: exit action) ausgeführt. Mit Hilfe der statischen Reaktion (engl.: static reaction), die in der UML auch als do-Aktivität bekannt ist, wird eine Aktivität solange ausgeführt, wie in diesem Zustand verweilt wird. Im Gegensatz zu Aktionen beanspruchen Aktivitäten eine gewisse Ausführungszeit und können somit unterbrochen werden.

Transitionen können von außen direkt in Unterzustände führen bzw. aus diesen herausführen. Wenn eingehende Transitionen am Rand des übergeordneten Zustands enden, ist das gleichbedeutend mit dem Eintreffen in die Startzustände der Unterzustände. In Abbildung 2-4 führt beispielsweise die Transition von Zustand *B* in den Zustand *A*, dabei wird automatisch der Unterzustand *A1* betreten. Bei einer Transition, die von einem zusammengesetzten Zustand ausgeht, werden alle Unterzustände nach dem Schalten der Transition mitverlassen. Dementsprechend führt das Eintreten von Ereignis *e2* zum Verlassen der beiden Unterzustände *A1* und *A2*.

Transitionsverläufe können durch Verzweigungspunkte beeinflusst werden. Diese Verzweigungspunkte werden durch *Auswahlkonnektoren* (engl.: choice connector) dargestellt, in denen eine Transition hineinführt und mehrere, mit booleschen Bedingungen versehene, Transitionen herausführen. In Abbildung 2-4 kann der Zustandsübergang vom Zustand *A2* je nach Bedingung entweder in den Zustand *C* oder *D* erfolgen.

[16] Nach der UML-Spezifikation benötigt die Ausführung der Aktionen keine Zeit. Dies ist ein wesentlicher Kritikpunkt bei der Modellierung von Echtzeitsystemen mit der UML 1.x.

[17] Einige Statechart-Ausprägungen, wie z. B. [Hui90] verbieten eine Interlevel-Transition.

Ausführungssemantik

Das Prinzip eines Statecharts basiert auf der in [HN96] angegebenen Semantik. Für die Beschreibung der Verhaltensdynamik einer komplexen Zustandsmaschine verwendet die UML eine hypothetische Ausführungsmaschine, die aus einer Ereignisschlange und einem Ereignisprozessor besteht. Die Ereignisschlange wird hierbei als serieller Eingabespeicher für eingetretene Ereignisse nach dem FIFO-Prinzip[18] benutzt und der Ereignisprozessor als operationelle Einheit, die diese eingetretenen Ereignisse aus der Ereignisschlange herausnimmt und entsprechend verarbeitet. Dadurch kann das Reaktionsverhalten einer Zustandsmaschine (Wahrnehmung von Ereignissen und damit verbundenen Zustandsübergängen) durch die Verarbeitungsschritte einer Ausführmaschine interpretiert werden [PS91]. Mit Hilfe eines Schritts wechselt das System aus einer aktuellen Konfiguration in eine Folgekonfiguration[19], wobei die Konfiguration folgende Punkte beinhaltet:

- die Menge aller Zustände, in denen das System zur Zeit verweilt,
- die Menge laufender Aktivitäten,
- die aktuellen Werte der Bedingungen und Variablen,
- die Menge interner generierter Ereignisse und
- relevante Informationen über die Systemgeschichte.

Je nach Anwendungsgebiet und Werkzeug unterscheidet sich die Ausführungssemantik der Statecharts erheblich [EW00]. So gibt es für asynchrone Zeitmodelle z. B. den *micro-step* und den *super-step* oder für synchrone Zeitmodelle den *sync-step* [HPSS87, John03]. Die UML schlägt als Ausführungsmodell den *run-to-completion-step* (rtc-Schritt) vor. In einem rtc-Schritt werden vom Ereignisprozessor folgende Teilschritte ausgeführt:

- ***Selektionsschritt:*** Aus der aktuellen Zustandskonfiguration werden alle Transitionen der Zustandsmaschine bestimmt, die bezüglich des akzeptierten Ereignisses schaltfähig sind und die nicht zueinander in Konflikt stehen.
- ***Verarbeitungsschritt:*** Die neue Zustandskonfiguration der Zustandsmaschine wird berechnet. Hierzu werden alle Zustandsübergänge der Transitionen, die im Selektionsschritt als schaltfähig und konfliktfrei ermittelt wurden, in einer nicht festgelegten Reihenfolge durchgeführt.
- ***Abschlussschritt:*** Wenn die im Verarbeitungsschritt neu berechnete Zustandskonfiguration instabil[20] ist, wird vom Ereignisprozessor ein *Abschlussereignis* akzeptiert und mit dem Selektionsschritt fortgefahren. Ansonsten wird der rtc-Schritt abgeschlossen.

[18] First-In-First-Out: Die Elemente werden hierbei in der Reihenfolge von der Queue zurückgeliefert, in der sie hinzugefügt wurden.

[19] Ein System wird allgemein als nichtdeterministisch bezeichnet, wenn durch Auslösen eines Ereignisses aus derselben Konfiguration mehrere Folgekonfigurationen existieren.

[20] Besitzt eine Zustandskonfiguration mindestens eine Transition die schaltfähig ist, so wird sie als instabil bezeichnet, da sie in dieser Konfiguration ohne eingetretenes Ereignis in eine neue Zustandskonfiguration übergehen kann.

Nach Angaben der UML-Spezifikation [OMG03b] kann der Vorgang eines rtc-Schritts nicht unterbrochen werden. Alle Ereignisse, die zwischendurch eintreten, werden in der Ereignisschlange erfasst und zwischengespeichert. Allerdings variiert die Implementierung des run-to-completion Algorithmus je nach Werkzeughersteller. In [HG96, John01, MPT03, Porr01] wird ausführlich auf Schrittsemantiken und ihre Unterschiede eingegangen.

Wenn nach dem Auslösen eines Ereignisses aus derselben Konfiguration unterschiedliche Folgekonfigurationen möglich sind, so spricht man hier von einem Konflikt. Eine Konfliktsituation liegt beispielsweise dann vor, wenn mehrere aktive Transitionen aus einem zusammengesetzten Zustand auf unterschiedliche Ebenen führen. In der Abbildung 2-4 auf Seite 25 wird dies durch den Zustand *A* und das Ereignis *e2* dargestellt. Für diesen Fall wird in der UML eine Prioritätsreihenfolge definiert, die dafür sorgt, dass Konflikte, die zwischen gleichzeitig aktivierten Transitionen auftreten, behoben werden können. Nach [OMG03a] hat diejenige Transition eine höhere Priorität, die von einem tiefer liegenden Zustand ausgeht. Eine Konfliktsituation entsteht auch, wenn mehrere aktive Transitionen aus einem Zustand führen. Der Zustand *B22* aus Abbildung 2-4 enthält zwei Transitionen mit demselben Ereignis *e5*. In diesem Fall haben beide Aktionen dieselbe Priorität. Die UML gibt für diesen Fall keine Lösung vor. Jedes Werkzeug löst einen solchen Konfliktfall auf eigene Art und Weise, entweder durch interne Prioritätenvergabe oder durch Warnmeldungen beim Simulationslauf.

Nebenläufigkeit

Das Konzept der Nebenläufigkeit wird durch orthogonale Regionen (AND-Zustände) repräsentiert. In der Abbildung 2-4 besitzt der AND-Zustand *B* die beiden Regionen *B1* und *B2*. Beim Aktivieren eines AND-Zustands wird keine Aussage bezüglich der Eintrittsreihenfolge in den Regionen gemacht. Wird ein solcher AND-Zustand betreten, so werden alle seine nebenläufigen Startzustände aktiviert. Soll dahingegen von einem anderen Zustand in einer Region gestartet werden, so ist ein Synchronisationsbalken zu verwenden. Je nachdem, ob ein Synchronisationsbalken mehrere Nachzustände oder mehrere Vorzustände besitzt, wird er als *Fork-Konnektor* oder als *Join-Konnektor* bezeichnet. Es können auch Mischformen auftreten [HK99]. Für Fork-Konnektoren gilt, dass ihre Nachzustände unterschiedlichen Subzuständen eines AND-Zustands angehören müssen und ihr Vorzustand außerhalb dieses AND-Zustands liegen muss. Bei Join-Konnektoren müssen die Vorzustände unterschiedlichen Subzuständen eines AND-Zustands angehören, während sich ihr Nachzustand außerhalb des AND-Zustands befinden muss. Beispielsweise wird in der Abbildung 2-4 der OR-Zustand *A* mittels des Ereignisses *e2* verlassen und mit Hilfe des Fork-Konnektors in die Basiszustände *B11* und *B21* des AND-Zustands *B* gewechselt, wobei die beiden Attribute *b* und *c* jeweils auf Null gesetzt werden. Transitionen in Regionen schalten grundsätzlich unabhängig voneinander. Wird beispielsweise aus der momentanen Basiskonfiguration das Ereignis *e3* ausgelöst, so wechselt das System in die Folgezustände *B12* und *B22*.

2.2.2.5 Sequenzdiagramme

Ein Sequenzdiagramm (engl.: sequence diagram) stellt eine oder mehrere Sequenzen eines bestimmten Szenarios dar. Dabei ist unter einer Sequenz folgendes zu verstehen:

Sequenz

Eine Sequenz zeigt eine Reihe von Nachrichten, die eine ausgewählte Menge von Objekten in einer zeitlich begrenzten Situation austauscht, wobei der zeitliche Ablauf betont wird [Oest98].

Ein Sequenzdiagramm besitzt zwei Dimensionen (vgl. Abbildung 2-5). Während die vertikale Dimension die Zeit repräsentiert, werden auf der Horizontalen die Objekte in einer beliebigen Reihenfolge eingetragen. Jedes Objekt wird durch eine Lebenslinie im Diagramm dargestellt. Diese repräsentiert die Existenz eines Objekts während einer bestimmten Zeit. Die Linie beginnt nach dem Erzeugen des Objekts und endet mit dem Löschen desselben. Nachrichten zwischen Objekten werden als Pfeile notiert.

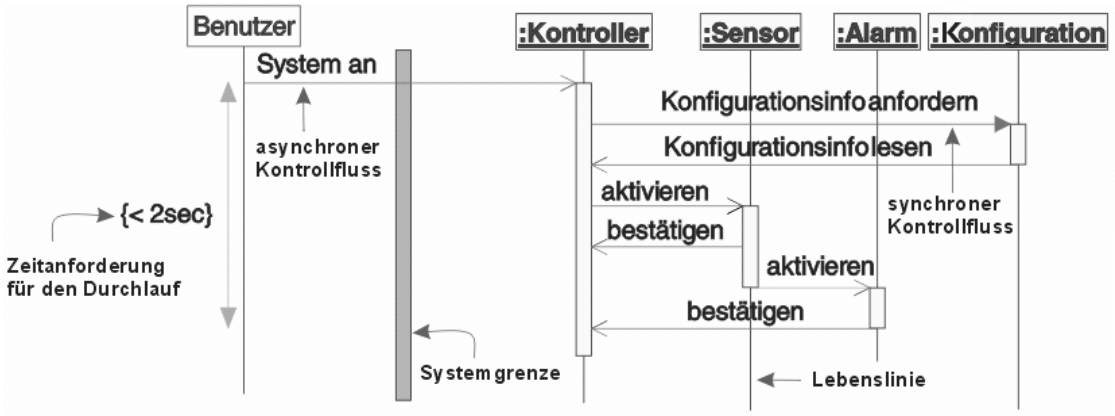


Abbildung 2-5: Ein Sequenzdiagramm der UML

In Sequenzdiagrammen ist es möglich, die Aktivierung von Objekten nachzuvollziehen und die Art der Kommunikation zu spezifizieren (synchron oder asynchron). Im Gegensatz zu UML Statecharts werden mit Sequenzdiagrammen nur beispielhaft interessante Szenarien beschrieben.

2.2.2.6 Paketdiagramme und Komponentendiagramme

Paketdiagramme bestehen aus Paketen, wobei Pakete Ansammlungen von Diagrammen und UML-Modellelementen (z. B. Klassen, Objekten, Assoziationen und Akteuren) sind, mit denen das Gesamtmodell in kleine überschaubare Einheiten oder nach bestimmten Kriterien gegliedert werden kann. Da Pakete selbst Modellelemente sind, können sie wiederum Pakete enthalten. Dadurch ist eine Hierarchiebildung möglich.

Komponentendiagramme (engl.: component diagrams) dienen zur Darstellung von Strukturen und Abhängigkeiten zwischen SW-Komponenten. Dies wird benötigt, um beispiels-

weise Compiler- und Laufzeitabhängigkeiten zu notieren oder zusammenhängende Klassen zu gruppieren, die selbstständig nicht ausgeführt werden können (vgl. Abbildung 2-6).

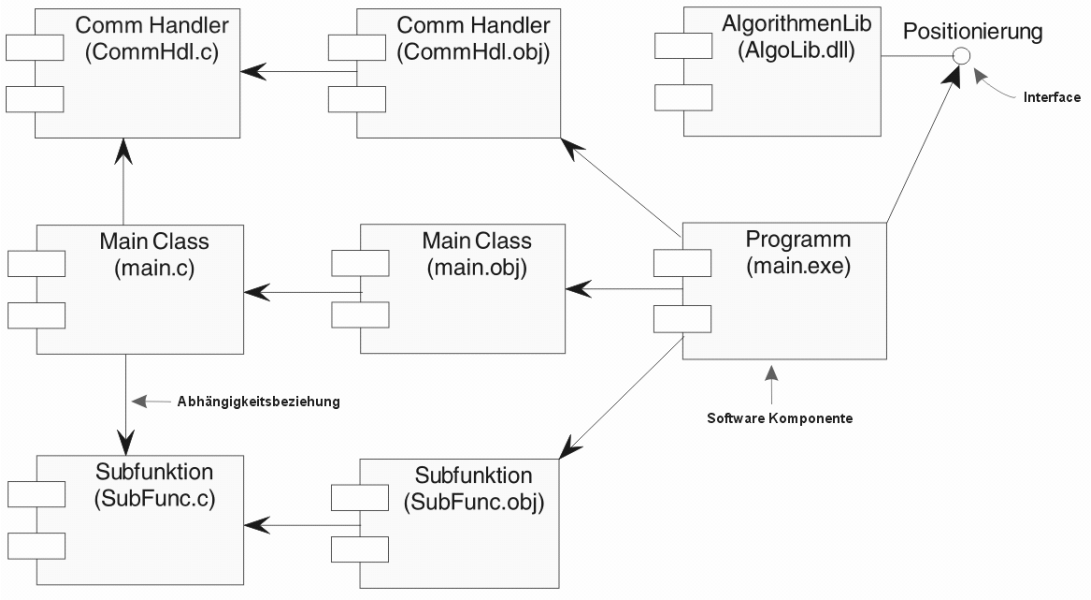


Abbildung 2-6: Ein Komponentendiagramm der UML

Komponenten sind Paketen sehr ähnlich. Sie definieren Grenzen und gruppieren eine Menge einzelner Elemente. Während Pakete eine mehr logische Sicht darstellen, betonen Komponenten die physische Sicht.

In der UML-Spezifikation wird die Komponente wie folgt beschrieben:

Komponente

Eine Komponente ist ein ausführbares SW-Modul mit Identität und definierter Schnittstelle. Mögliche Realisierungsformen der Komponenten sind Quellcode, Bytecode, Binärcode oder ausführbares Programm [OMG03a].

2.2.2.7 Verteilungsdiagramme

Für die Darstellung der Zuordnung von Prozessen und Objekten zu Rechnern bzw. Prozessoren wurden in der UML Verteilungsdiagramme (engl.: deployment diagrams) eingeführt. Dazu werden die SW-Komponenten aus dem Komponentendiagramm den physischen Knoten zugeordnet, die jeweils eine Hardware-Ressource repräsentieren. Zwischen den Knoten existieren Verbindungen, die physikalische Kommunikationspfade darstellen (vgl. Abbildung 2-7). Der Einsatz von Verteilungsdiagrammen ist besonders bei der Modellierung der physikalischen Architektur von verteilten SW-Systemen sinnvoll.

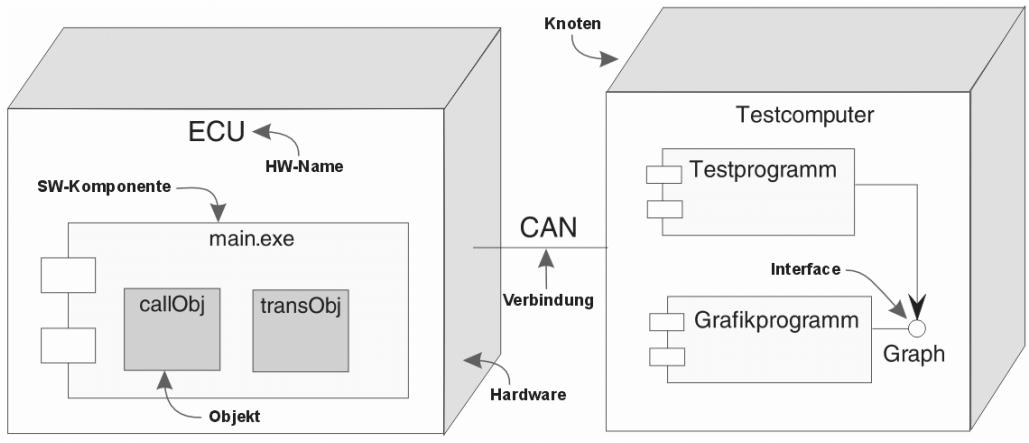


Abbildung 2-7: Ein Verteilungsdiagramm der UML

2.2.3 Erweiterungskonzepte der UML

Soll eine Modellierungstechnik für eine Vielzahl verschiedenster Anwendungsbereiche geeignet sein ohne gleichzeitig zu umfangreich zu werden, so können keine speziellen, auf bestimmte Anwendungsbereiche zugeschnittenen Konzepte aufgenommen werden. Viele Anwendungsbereiche sind jedoch durch spezifische Besonderheiten charakterisiert, die sich stark von den Eigenschaften eines gewöhnlichen SW-Systems unterscheiden. Dies gilt besonders für die hier betrachteten *Eingebetteten Systeme* im Automobilbereich.

Vor diesem Hintergrund stehen der UML verschiedene Konzepte zur Verfügung, mit denen projektspezifische Anpassungen und Erweiterungen ermöglicht werden. So lässt sich mit Hilfe der Metamodellierung die Grundstruktur der UML persistent modifizieren und erweitern. Dahingegen bilden Erweiterungsmechanismen einen leichtgewichtigeren Ansatz, bei dem vorhandene Modellkonstrukte zur Erweiterung der gängigen UML-Spezifikation verwendet werden können. Zur Beschreibung von formalen Sachverhalten verfügt die UML über die formale Beschreibungssprache OCL. Darüber hinaus ist ein Austausch von Metadaten durch ein standardisiertes Austauschformat grundsätzlich möglich. In den nachfolgenden Abschnitten werden diese Erweiterungskonzepte genauer vorgestellt.

2.2.3.1 Metamodellierung

Die Modellierungskonzepte der UML, die beispielsweise bei der Spezifikation eines Klassenmodells zum Einsatz kommen, sind unter anderem Klassen und Assoziationen. Werden die in einem Modell benutzten Sprachkonzepte mit den Konzepten derselben Sprache beschrieben, so spricht man von *Metamodellierung*.

Diese Art der Modellierung wird von der UML durch das *4-Schichten-Modell* (vgl. Tabelle 2-2) unterstützt. Die Sprachbeschreibung der UML wird ebenenweise angegeben, wobei jede Ebene die Menge der möglichen Ausprägungen der nächst tieferen Ebene bestimmt und dabei selbst aus Instanziierungen von Elementen der nächst höheren Ebene besteht.

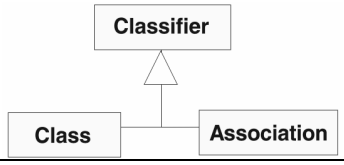
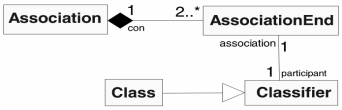
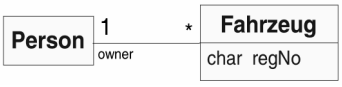
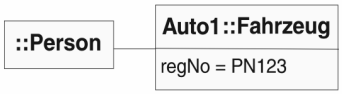
Ebene	Beschreibung	Beispiel
M3: Meta-Meta-Modellebene	Definiert formal die Syntax und Semantik der Meta-Modellebene.	
M2: Meta-Modelle-bene	Beschreibt formal mit Hilfe von Klassen und Assoziationen die grundlegenden Sprach-konzepte der UML (z. B. das Konzept der Klasse).	
M1: Modellebene	Macht von den im Metamodell spezifizierten Sprachkonzepten Gebrauch und nutzt sie zur Modellierung.	
M0: Benutzerebene	Entspricht der Instanziierung einer Klasse.	

Tabelle 2-2: Das 4-Schichten-Modell der UML

Auf der *Benutzerebene* werden konkrete Daten des Systems dargestellt. Als Beispiel sei hier das Objektdiagramm genannt, welches die momentanen Attributwerte seines Klassendiagramms zu einem bestimmten Zeitpunkt enthält. In der nächst höheren Schicht, der *Modellebene* kommen die bereits vorgestellten UML-Diagramme zur Beschreibung der erforderlichen Systemaspekte zum Einsatz. Die *Meta-Modellebene* spezifiziert Elemente, die zur Modellierung auf der Ebene M1 eingesetzt werden können. Durch dieses Meta-Modell ist die abstrakte Syntax der Modellierungssprache formal definiert. Alle Elemente, die auf der Ebene M2 zur Angabe des Meta-Modells verwendet werden, definiert man schließlich auf der *Meta-Meta-Modellebene*. Diese Ebene ist fest für alle Modellierungssprachen, die nach der *Meta-Object Facility* (MOF)[21] definiert sind.

Die Vorteile der Metamodellierung nach [HK99] sind:

- ***Formale Spezifikation*:** Durch die Definition eines Metamodells sind alle Modellierungskonzepte mit ihren Merkmalen, Einschränkungen und erlaubten Beziehungen untereinander formal beschrieben.
- ***Einheitliche Austauschformate*:** Alle erzeugten Modelle werden als Instanzen von Metamodellklassen modelliert und dargestellt. Damit ist die Basis für die Definition eines standardisierten Austauschformats vorgegeben.
- ***Erweiterbarkeit*:** Ein definiertes Metamodell kann mit den Mitteln der Modellierungssprache selbst erweitert werden.

2.2.3.2 Erweiterungsmechanismen

Die Anpassung der UML ist prinzipiell durch die Metamodellierung möglich. Dies ist aber aus vielerlei Gründen nachteilig. So führt eine Änderung in der Meta-Modellebene z. B. zur Inkompatibilität zwischen den sich daraus ergebenden Modellen und den Modellen aus der UML-Spezifikation. Daher wird diese Erweiterungsmöglichkeit von den meisten

[21] Die MOF ist ein Standard der OMG [OMG03], die die Modellierung von Meta-Informationen erlaubt.

Werkzeugherstellern nicht angeboten. Einen anderen, leichtgewichtigeren Ansatz bieten die drei UML-Erweiterungsmechanismen:

- ***Stereotypen*:** Ein Stereotyp ist ein neues Element, das auf einem vorhandenen Element der UML aufbaut. Dazu werden der grafischen Notation Winkelklammern (Guillemots) zur Kennzeichnung als Stereotyp hinzugefügt: *<<Name>>*. Einer getrennten Dokumentation ist zu entnehmen, wie sich die Syntax und Semantik des neuen Elements von der des bekannten UML-Elements unterscheiden.
- ***Einschränkungen:*** Einem Modellelement können beliebig viele Einschränkungen (engl.: constraints) zugeordnet werden. Eine Einschränkung gibt eine Eigenschaft vor, die ein UML-Element erfüllen muss. Diese Einschränkung kann in natürlicher, formaler oder einer beliebigen anderen textuellen Sprache definiert werden, da sie innerhalb der UML nicht interpretiert wird. Eine Einschränkung wird immer in geschwungene Klammern gesetzt.
- ***Eigenschaftswerte*:** Dies sind spezielle Schlüsselwörter, die einem Element zugeordnet werden und somit die Semantik des Elements detaillieren. Sie sind den Einschränkungen ähnlich, indem sie ebenfalls in einer beliebigen Sprache definierbar sind. Bekannte Eigenschaftswerte (engl.: tagged values) aus der objektorientierten Programmierung sind *abstract* (kennzeichnet abstrakte Klassen und Operationen) und *private* (verwehrt direkte Zugriffe auf Attribute oder Operationen anderer Objekte).

In der UML existieren bereits einige vordefinierte Stereotypen, Einschränkungen und Eigenschaftswerte. Eine ausführliche Liste befindet sich unter anderem in [BRJ00].

Seit der Version 1.3 verfügt die UML über das Konzept der Benutzerprofile (engl.: profiles). Ein Benutzerprofil ist eine Sammlung von Stereotypen, Einschränkungen und Eigenschaftswerten. Es dient zur Anpassung bzw. Erweiterung der UML an fachliche oder technische Domänen. Beispielsweise stellt die UML das *Profile for Schedulability, Performance and Time* [OMG02a] zur Verfügung, mit dem unter anderem Echtzeitaspekte in die Modellierung aufgenommen werden können.

2.2.3.3 Object Constraint Language

Mit Hilfe der *Object Constraint Language* (OCL) können bestimmte Einschränkungen und Ergänzungen an Modellelemente in Form von Invarianten sowie Vor- und Nachbedingungen hinzugefügt werden. Diese Informationen lassen sich häufig nicht in Diagrammen ausdrücken. Die OCL [WK98] basiert auf mathematischer Mengenlehre und Prädikatenlogik und besitzt eine formale Semantik. Mathematische Symbole, wie $\forall$ oder $\exists$, werden in ASCII-Form (*forAll* bzw. *exists*) repräsentiert. Obwohl es auf den ersten Blick so scheint, als sei eine präzise, eindeutige Notation eine gute Wahl, lässt sich eine mathematische Notation nicht als weit verbreitete Standardsprache anwenden. Der Grund hierfür ist, dass eine solche formale Notation in der Praxis nur von wenigen Entwicklern akzeptiert wird.

Charakteristisch für die OCL ist des Weiteren, dass sie eine deklarative Sprache ist. Somit beschreibt ein OCL-Ausdruck lediglich was getan werden muss, aber nicht wie. OCL-Ausdrücke haben daher keine Seiteneffekte[22]. Das bedeutet, dass der Status eines Systems durch die Auswertung eines OCL-Ausdrucks nicht verändert wird. Die Syntax ist angelehnt an die Programmiersprachen C++ und Java. Wegen dieser und vieler anderer positiver Eigenschaften ist sie seit der UML 1.1 [OMG97b] ein fester Bestandteil der OMG.

Das folgende Beispiel für einen einfachen OCL-Ausdruck bezieht sich auf die Abbildung 2-3 auf Seite 24.

```
context f:Pkw
inv: f.AnzahlTüren>=2;
```

Invarianten gelten immer für alle Instanzen eines Typs, dem sie zugeordnet sind. Im obigen Beispiel ist die eigentliche Invariante in der zweiten Zeile zu finden, während die erste Zeile mit dem Schlüsselwort *context* und dem nachfolgenden *Fahrzeug* den Typ bezeichnet, dem die Invariante zugeordnet ist, wobei diese durch *inv* als solche gekennzeichnet ist. Die Semantik des Ausdrucks ist, dass das Attribut *AnzahlTüren* für alle Instanzen vom Typ *Pkw* einen Wert von größer oder gleich 2 haben muss.

2.2.3.4 Das UML Metadata - Austauschformat

Die Speicherung der Daten einzelner Anwendungen erfolgt teilweise in proprietären Formaten, die einen Datenaustausch verhindern. Daraus erwächst der Bedarf nach einem einheitlichen Standard zum Datenaustausch von UML-Modellen. Dieses Problems hat sich die OMG angenommen und versucht, eine standardisierte Möglichkeit zum plattform- und werkzeugunabhängigen Austausch von Metadaten zu schaffen. Das Ergebnis ihrer Bemühungen ist die XMI-Spezifikation (XML Metadata Interchange). Zusammenfassend lässt sich die XMI-Spezifikation [OMG03c] als eine Darstellungsform von UML-Modellen unter Verwendung der Sprachmittel beschreiben, die der *eXtensible Markup Language*-Standard (XML) zur Verfügung stellt.

Die XML [Midd99] ist eine allgemeine Methode, um Daten in textueller Form auf flexible Weise zu verwalten. Jedes XML Dokument ist konzeptionell ein Baum mit verschiedenen Typen von Knoten, welche textuell repräsentiert werden durch ein Paar von geöffneten und geschlossenen Etiketten (engl.: tags*)*. Daher sind sie der *Hyper Text Markup Language* (HTML) [23] sehr ähnlich. Im Gegensatz zur HTML können für ein XML-Dokument die zu verwendenden Elemente frei definiert werden. Die Definition eines Elementes umfasst neben dem Elementnamen auch eine strukturelle Beschreibung des Inhalts, die Beziehung zu anderen Elementen, eine Liste der möglichen Attribute und vieles mehr. Diese Festlegungen werden in der Dokumenttypdeklaration (DTD) beschrieben. Dabei werden die Namen gültiger Elemente und Attribute sowie ihre Struktur ähnlich einer Grammatik

[22] Ausdrücke wie z. B. *a>b++* dürfen in der OCL nicht verwendet werden, da unklar ist, ob der Vergleich oder die Zuweisung als erstes ausgeführt wird.
[23] Vgl. http://www.w3c.org.

festgelegt. Neben der Schachtelung werden Konstruktoren, Sequenzen, Alternativen und Iterationen verwendet. Die Semantik der XML-Elemente wird nicht in der DTD definiert. Die Bedeutung der verwendeten Elemente muss in externen Dokumenten beschrieben oder aus dem Zusammenhang entnommen werden.

2.3 CASE-Werkzeuge für den Einsatz in der Automobil-Software-Entwicklung

Der Erfolg einer Methodik für die SW-Entwicklung in der industriellen Praxis wird maßgeblich von der Qualität und Durchgängigkeit der Werkzeugunterstützung bestimmt. Denn eine möglichst weit reichende Unterstützung der Werkzeuge kann die Effektivität einer Methode entscheidend erhöhen [Raus01]. Entsprechende CASE-Werkzeuge sind in nahezu allen Bereichen der SW-Entwicklung sinnvoll und kommerziell verfügbar. Der Einsatz solcher CASE Tools in der Automobilindustrie erfolgt derzeit schwerpunktmäßig in den Gebieten des Testens, der Anforderungserfassung, Modellierung, Diagnose und automatischen Code-Generierung [BBK98, Hofm02]. Dabei wird unter dem Begriff CASE folgendes verstanden:

CASE

CASE (Computer Aided Software Engineering) bezeichnet die aufeinander abgestimmte und durchgängige Anwendung von Werkzeugen im SW-Entwicklungsprozess.

Die folgenden Abschnitte stellen zunächst dar, welchen Anforderungen CASE-Werkzeuge im Hinblick auf den Entwicklungsprozess von eingebetteten Systemen genügen sollen. Darüber hinaus wird die tatsächliche Realisierung dieser Anforderungen in den heutigen Werkzeugen erörtert. Abschließend erfolgt eine Klassifizierung der Tools nach Aufgabenbereichen.

2.3.1 Anforderungen an Werkzeuge

Durch den Einsatz geeigneter CASE-Werkzeuge erhoffen sich SW-Entwickler einen effizienteren Entwicklungsprozess, mit dem die Software hochwertiger und gleichzeitig kostengünstiger entwickelt werden kann. Um diese Erwartungen zu erfüllen, müssen CASE Tools bestimmten Anforderungen genügen.

In [EP98, Ivan99] werden allgemeine Anforderungen an moderne Modellierungswerkzeuge diskutiert. [BBCP+03] und [BBK98] untersuchen ausgewählte Anforderungen an SW-Entwicklungswerkzeuge im Automobilbereich. Darüber hinaus sind in [OBJ02] Werkzeugeigenschaften aufgelistet, die eher einen Innovationscharakter haben und in den meisten Werkzeugen bislang nicht implementiert worden sind. Für die Modellierung reaktiver Systeme stellt [Katt97] zusätzliche Anforderungen an zustandsbasierte Modellierungswerkzeuge auf.

Die nachfolgenden Abschnitte geben einen Überblick über typische Anforderungen an moderne Werkzeuge für die modellbasierte Entwicklung von eingebetteten Systemen im Automobilbereich.

Beschreibungstechniken

Zur Spezifikation von reaktiven Steuergerätefunktionen sollen grafische und ausführbare Beschreibungsmittel verwendet werden. Die grafische Darstellung dient zur strukturierten Beschreibung komplexer Systeme, die Simulationsfähigkeit unterstützt zusätzlich eine schnelle Einarbeitung und Transparenz in komplexe Abläufe. Weitere Erleichterungen aufgrund der grafischen Darstellung ergeben sich bei der Erweiterung bzw. Änderung von bestehenden (Sub-)Systemen, in denen Teilfunktionen leichter lokalisiert und ausgetauscht werden können. Wie in [SZ03] beschrieben, bestehen elektronische Steuerungssysteme im Fahrzeug aus diskreten, reaktiven Steuerungen und kontinuierlichen Regelungen. Hierfür eignen sich beispielsweise Zustandsautomaten zur Beschreibung von diskreten, reaktiven Systemanteilen und Blockdiagramme für kontinuierliche Funktionen. Diese Beschreibungssprachen haben sich als Industriestandard etabliert und sind in vielen Modellierungswerkzeugen integriert. Bei Zustandsautomaten sollte auf erweiterte Formalismen wie z. B. Statecharts (vgl. Kapitel 2.2.2.4 auf Seite 25) zurückgegriffen werden [Katt97]. Sie bieten unter anderem Mechanismen und Konzepte zur Erstellung von komplexen Systemfunktionen. Durch Verwendung von Hierarchie, Nebenläufigkeit und speziellen Modellkonstrukten kann eine Zustandsexploration vermieden werden. Darüber hinaus unterstützen temporale Kontrollstrukturen die Einbindung von zeitlichen Aspekten. Transitionsprioritäten dagegen verhindern Konflikte bei hierarchischen Strukturen.

Algorithmen zur Minimierung von Zuständen in einem Automaten können den Ressourcenverbrauch (Speicherkapazität und Prozessorlast) eines Mikrocontrollers minimieren und somit die Effektivität des zu entwerfenden SW-Systems erhöhen. Solche Algorithmen sind in kommerziellen Werkzeugen derzeit leider nicht auf der Konstruktionsebene implementiert. Stattdessen findet die Optimierung der Modelle erst bei der Code-Generierung statt. Dies hat zunächst den Vorteil, dass das Modell vom Werkzeug nicht verändert wird und somit eine evtl. Anpassung des SW-Systems entfallen kann. Des Weiteren können im Normalfall weitere Optimierungsparameter erst dann betrachtet werden, wenn die Zielplattform (engl.: target) und das Betriebssystem angebunden worden sind. Der Nachteil bei der späten Optimierung wird ersichtlich, wenn bei der Code-Generierung die maximal erlaubte Code-Größe überschritten wird. Dem Entwickler bleibt dann in der Regel eine manuelle Anpassung des Codes nicht erspart. Im Extremfall muss sogar auf ein größeres Steuergerät zurückgegriffen werden oder die gesamte SW-Architektur neu konzipiert werden. Beide Maßnahmen haben enorme Kosten zur Folge. Um solchen Extremfällen vorzubeugen, verzichtet beispielsweise der Werkzeughersteller ETAS in seinem Modellierungswerkzeug ASCET-SD [ETAS00] auf Nebenläufigkeiten in einem Zustandsautomaten und den damit verbundenen hohen Speicherverbrauch. Wünschenswert wäre eine Kombination aus den beiden vorgestellten Optimierungsprozessen.

Für objektorientierte SW-Entwicklungen eignet sich die standardisierte Beschreibungssprache UML (vgl. Kapitel 2.2.2 auf Seite 21). Dabei sollten die UML-Modellierungswerkzeuge zumindest annähernd den gesamten Sprachumfang der UML aufweisen. Aufgrund der Diagrammvielfalt und ungenauen Spezifikationen kann zurzeit eine vollständige und einheitliche Umsetzung von keinem Werkzeug erfüllt werden.

Einen besonderen Stellenwert hat die grafische Darstellung von Informationen, mit der eine vereinfachte Modellierung möglich sein muss. Beispielsweise sollen Teile eines Diagramms und bestimmte Merkmale eines Modells durch den Einsatz von Filtern hervorgehoben oder versteckt werden können. Diese Anforderungen sind in vielen Werkzeugen bereits gut umgesetzt.

Zentrale Steuerung und Datenhaltung

Die gemeinsame Nutzung von Informationen aller Modellkomponenten ist für eine konsistente und verteilte SW-Erstellung sehr wichtig. Eine zentrale Datenbasis und eine Steuerungseinheit ermöglichen die parallele Abarbeitung von Prozessen, vereinfachen die Navigation zwischen Diagrammen und verbessern die Kontrolle der Zugriffsrechte sowie die Prioritätenvergabe. Um diese Verbesserungen zu realisieren, verfügen die meisten Modellierungswerkzeuge über ein *Repository*. Mit Hilfe eines solchen Repositorys können zusätzliche Aufgaben wie das Sicherstellen der Datenintegrität und Gewährleistung eines Datenaustauschs erleichtert werden. In einigen Fällen sind Mechanismen für das Konfigurations- und Versionsmanagement vorhanden.

Wiederverwendung

Die Wiederverwendung von Modellen, Programm-Codes und anderen Artefakten spielt in einer modellbasierten SW-Entwicklung eine zentrale Rolle und muss daher durch ein Werkzeug unterstützt werden. Hierfür sind verschiedene Maßnahmen denkbar. Für die Wiederverwendung von Modellen aus anderen Projekten, bieten alle gängigen CASE Tools Import- und Exportfunktionen an. Die meisten Werkzeuge können jeweils nur eigene Modelltypen einlesen, wodurch die Wiederverwendbarkeit begrenzt wird und die Abhängigkeit von einem Toolhersteller zunimmt. Eine weitere Maßnahme besteht in der Nutzung von objektorientierten Methoden und Notationen, mit denen Vererbung und Kapselung möglich sind. Diese Maßnahme wird durch alle objektorientierten Werkzeuge umgesetzt. Moderne Lösungskonzepte wie z. B. Design-Muster (engl.: design patterns) sind trotz des ansteigenden Interesses [Hunt03, Quib99] nur in wenigen Werkzeugen vorhanden. Ein pragmatischer und bewährter Ansatz ist die Verwendung von Modellbibliotheken, in denen häufig verwendete Funktionen und Funktionsgruppen bereitgestellt werden. Diese lassen sich durch wohldefinierte Schnittstellen in das Gesamtsystem einbinden. In den letzten Jahren wurden durch verschiedene Organisationen, wie ASAM[24], MISRA[25] und verschiedene Toolhersteller einige Bibliotheken wie MAAB [MATH01] und MSR-MEGMA [WM99] für Automotive-Anwendungen definiert.

[24] Vgl. http://www.asam.net
[25] Vgl. http://www.misra.org.uk

Qualitätssicherung

Komplexe SW-Entwicklung kann mit Unterstützung der Qualitätssicherung (QS) effektiver und somit kostengünstiger durchgeführt werden. Dabei kann man zwischen konstruktiven und analytischen QS-Maßnahmen unterscheiden. Unter konstruktiven Maßnahmen werden beispielsweise Verfahren zur Erstellung von Modellierungs- und Qualitätsrichtlinien sowie Bewertungskriterien verstanden, die in den letzten Jahren in der SW-Entwicklung zugenommen haben [BM96, Rau01b]. Mit diesen können beispielsweise projektspezifische Bedingungen und Modellierungsregeln definiert werden. SW-Metriken dagegen bewerten die Qualität eines Modells, indem sie dessen Eigenschaften auf Sollwerte überprüfen. Eine nähere Beschreibung der SW-Metriken erfolgt in Kapitel 5.1.2. In der Automobilindustrie wurde die Wichtigkeit von konstruktiven Maßnahmen erkannt [MISRA04a, MISRA04b, RA95a], ihre Umsetzung aber nur in wenigen Werkzeugen realisiert.

Die konstruktiven Verfahren können durch analytische Maßnahmen unterstützt werden. Durch diese Maßnahmen lässt sich die Einhaltung der vorgegebenen Qualitätsanforderungen überprüfen. Die wichtigsten analytischen Verfahren sind Simulation, Anforderungs-Tracing, Konsistenzüberprüfung und (formale) Tests von Systemeigenschaften. Die analytischen Maßnahmen werden in der heutigen SW-Entwicklung erfolgreich durch entsprechende Werkzeuge unterstützt und daher in Kapitel 5.2 näher vorgestellt.

Round-Trip Engineering

Round-Trip Engineering bezeichnet die Vereinigung der beiden Verfahren *Forward Engineering* und *Reverse Engineering*. Beim Forward Engineering wird aus Modellen automatisch Code generiert. Diese Maßnahme ist sehr vorteilhaft, da bei der späteren Realisierung die manuelle Umsetzung vom Entwurf zur Implementierung größtenteils entfällt. Eine Reduzierung von Implementierungsfehlern ist die Folge. Beim Reverse Engineering ist das Gegenteil der Fall. Aus dem eingelesenen Code wird das Modell automatisch erzeugt. Das Round-Trip Engineering ist ein Verfahren zur Synchronisation von Code und Modellen und vereinfacht somit das Reengineering. Unter Reengineering wird die Überarbeitung, Weiterentwicklung oder Wartung eines bestehenden Systems verstanden.

Die meisten Modellierungswerkzeuge für den Automobilbereich unterstützen den Forward Engineering Prozess, indem sie aus Klassen- und Blockdiagrammen bzw. aus Zustandsautomaten Code erzeugen. Dabei ist die Zielsprache und das Target frei wählbar, wobei Ada, C/C++ sowie Java von den meisten Generatoren unterstützt werden. Der generierte Code eignet sich entweder für einen Prototypen oder für die Serienfertigung. Während der prototypische Code hauptsächlich zur Simulation auf einem PC verwendet wird und somit keine Überprüfung im Hinblick auf die Einhaltung von echtzeitrelevanten Eigenschaften enthält, wird der Serien-Code zur Ausführung in Echtzeitumgebungen optimiert und kann beispielsweise in der Hardware-in-the-Loop-Umgebung (HIL) eingesetzt werden. Das Reverse Engineering dagegen ist nur in wenigen Werkzeugen realisiert und derzeit nur auf Systemstrukturen begrenzt.

Dokumentation

In komplexen SW-Projekten ist die Dokumentation von großer Bedeutung. Eine reine Kommentierung des Codes reicht für die modellbasierte Entwicklung nicht mehr aus. Die umfangreichen Analyse- und Designergebnisse in Form von Diagrammen, Tabellen und Code müssen in einheitlicher Form zur Verfügung stehen. Dabei müssen schriftliche Anmerkungen der Entwickler, Tester, Projektleiter und weiterer Projektteilnehmer mit aufgenommen werden. Dies verlangt nicht nur flexible Abstraktionsstufen der grafischen Informationen sondern auch einheitliche und ggf. standardisierte Formatvorlagen. Zusätzlich müssen Modelle für weitere Zwecke in die gängigsten Grafikformate umgewandelt werden können. Dabei sind Vektorgrafiken wie SVG [Adam02] zu bevorzugen, um die Modelle ohne visuelle Seiteneffekte beliebig vergrößern und durch zusätzliche Grafikprogramme einfach bearbeiten zu können.

Um diesen Anforderungen zu entsprechen, enthalten Modellierungswerkzeuge sogenannte Reportgeneratoren. Diese stellen alle verfügbaren Modellinformationen in einem sinnvollen Zusammenhang und übersichtlicher Form zur Verfügung. Zur weiteren Verarbeitung lassen sich die Modelle in übliche Formate wie z. B. HTML, SVG oder DOC exportieren.

Die Möglichkeit, große Diagramme auf mehreren Seiten auszudrucken und mit individuellen Kommentaren zu versehen, ist derzeit noch die Ausnahme.

Werkzeugintegration

Die Verwendung mehrerer kooperierender Werkzeuge ist in einem Entwicklungsprozess unvermeidlich. Die Gründe hierfür sind vielfältig. Einerseits erfüllt kein Werkzeug allein die bisher genannten Anforderungen, andererseits wird aus Sicht der Fahrzeughersteller eine Werkzeugunabhängigkeit angestrebt. Für eine durchgängige und modellbasierte Entwicklung sind demnach standardisierte Austauschformate und offene Schnittstellen erforderlich. Durch offene Schnittstellen soll die Integration benutzerspezifischer Funktionen ermöglicht werden.

Die meisten Werkzeuge verfügen leider nur über proprietäre Datenstrukturen und Schnittstellen, so dass ein Modellaustausch zwischen solchen Entwicklungswerkzeugen nicht möglich ist [LBB+01]. Die OMG erkannte dieses Problem und definierte die standardisierte Schnittstelle XMI (vgl. Kapitel 2.2.3.4 auf Seite 35). Die Durchführbarkeit eines Modellaustauschs ist jedoch sehr begrenzt, da Inkompatibilitäten zwischen Versionen von XMI und UML zu verzeichnen sind. Experimente zeigen, dass es sogar beim Modellaustausch zwischen Werkzeugen eines Herstellers zu Datenverlusten kam [BBCP+03]. Eine weitere Schwierigkeit beim Modellaustausch ist die Positionierung der eingelesenen Modellelemente wie z. B. Klassen und deren Relationen, da im Meta-Modell der UML keine Grafikinformationen enthalten sind. Intelligente Algorithmen zur automatischen Positionierung (z. B. Vermeidung von Transitionsüberschneidungen) sind nur in den seltensten Fällen implementiert. Dies wurde in einer Evaluierungsstudie [Eich02] mit 42 Werkzeugen an der Universität Würzburg untersucht. Eine Integration von benutzerspezifischen Funktio-

nen ist mittlerweile in fast allen Werkzeugen durch Bereitstellung von entsprechenden Schnittstellen möglich.

2.3.2 Klassifizierung von Werkzeugen

Abbildung 2-8 zeigt die möglichen Teilaufgaben eines modellbasierten SW-Entwicklungsprozesses. Für alle diese Teilaufgaben besteht in der Praxis die Möglichkeit der Werkzeugunterstützung. In [BBK98] wird für diese Teilaufgaben eine Vielzahl von Werkzeugherstellern und Produkten aufgelistet, die sich für den Embedded-Bereich, insbesondere den Automobilbereich eignen.

Abbildung 2-8: Mögliche Aufgaben in einem modellbasierten SW-Entwicklungsprozess

Aus dem vorhergehenden Abschnitt wird deutlich, dass ein einzelnes Werkzeug nicht alle Aufgaben unterstützen kann. Die verschiedenen Werkzeughersteller haben sich daher auf bestimmte Kerngebiete der SW-Entwicklung spezialisiert [BBCP+03]. Die auf dem Markt befindlichen Werkzeuge lassen sich in vier verschiedene Werkzeuggruppen einteilen[26]. In Abhängigkeit von Projektumfang, Organisationsstruktur und der zu entwickelnden Systemdomäne (z. B. Steuerungs-, oder Regelungssysteme mit und ohne Verteilungs-, Echtzeit- und Sicherheitsaspekten) variiert die Anzahl der Werkzeugtypen in einem Projekt. Nachfolgend werden die vier Werkzeuggruppen erläutert.

Werkzeuge zur Projektplanung und Projektsteuerung

Zu jedem erfolgreichen Projekt gehört einerseits die Planung von Mitarbeitern, Finanzen und Sachmitteln, andererseits die Anpassung des Prozesses an die Betriebsorganisation [West91]. Ein Ergebnis der Projektplanung ist unter anderem ein Projektplan mit definierten Meilensteinen, Aktivitäten und Richtlinien. Durch die Projektsteuerung können beispielsweise Aktivitäten und Verantwortlichkeiten bestimmt und regelmäßige Sitzungen organisiert werden. Ergebnisse der Projektsteuerung sind z. B. Protokolle und Zwischenberichte. Weitere Tätigkeiten und Artefakte sind aus [Wien97] zu entnehmen.

Für diese Aufgabenbereiche gibt es eine Vielzahl von bewährten CASE Tools. In [Star94] und [Tich85] werden einige Werkzeuge zur Projektplanung und -steuerung vorgestellt.

[26] [BBK98] gliedern diese vier Gruppen in weitere Teilgruppen auf.

Werkzeuge für das Requirements Engineering und Management

Darunter werden alle Werkzeuge verstanden, die sich mit dem Analysieren, Erfassen und Organisieren von Anforderungen befassen. Diese Art von Werkzeugen hilft beim Erstellen eines Lastenheftes. Im Gegensatz zu konventionellen Methoden, bei denen die Anforderungen an ein (Teil-)System in einem gängigen Textdokument zusammengefasst werden, besitzen solche Werkzeuge eine Datenbank mit speziellen Zugriffs- und Bearbeitungsoperationen. Dies bietet eine Vielzahl von Vorteilen, wie z. B. Verlinkungen von Anforderungen, Filtern von Informationen und teilweise Verhinderung von Redundanzen und Inkonsistenzen zwischen textuellen Anforderungen und grafischen Modellen. In Kapitel 3.2.3 auf Seite 54 wird das Werkzeug *DOORS* vorgestellt. Weitere Werkzeuge zum Requirements Engineering und Management sind beispielsweise aus [Broy97, Joos00, Knet01] zu entnehmen. In der Regel sind Schnittstellen zu Dokumentationswerkzeugen und Datenbanken möglich.

Modellierungswerkzeuge

Modellierungswerkzeuge bilden mit ihren umfangreichen Funktionen die Grundlage eines modellbasierten SW-Entwicklungsprozesses. Mit Ihnen können sowohl Modelle zur Struktur als auch zum Verhalten in unterschiedlichen Abstraktionsstufen erstellt werden. Dabei unterscheiden sich die Werkzeuge durch ihre Modellierungssprachen. So eignet sich beispielsweise *Statemate MAGNUM* [HLNP+90] mit seinen Statecharts vor allem für diskrete, ereignisgetriggerte Steuerungssysteme und *Simulink* mit seinen Blockdiagrammen für kontinuierliche, zyklische Regelungssysteme. Einige Werkzeuge wie die *MATLAB Toolbox* (vgl. Kapitel 3.2.3 auf Seite 54) oder *ASCET-SD* verknüpfen beide Laufzeitmodelle. Die UML als Beschreibungssprache ist domänenneutral und daher universell anwendbar. Aus diesem Grund existiert eine Vielzahl von UML-Werkzeugen auf dem Software-Markt.

Für ereignisgesteuerte Systeme [Kien97] werden zustandsbasierte Modelle wie Statecharts verwendet. Solche Verhaltensmodelle lassen sich durch integrierte Simulatoren schrittweise ausführen. In manchen Fällen kann die Simulation durch einfache Animationen wie in Statemate MAGNUM veranschaulicht werden.

Die heutigen Modellierungswerkzeuge haben bereits eine Möglichkeit, ihre Verhaltensmodelle zu simulieren und teilweise auch zu animieren. Leider gibt es bei der direkten Simulation einige Nachteile. Zum einen ist die Simulation unkomfortabel, da alle Ereignisse nur manuell und sequentiell eingegeben werden können. Außerdem ist es schwierig, die Systemausgaben während der Simulation richtig zu interpretieren und zu erfassen, wenn alle Ausgaben textuell über ein Fenster dargestellt werden. Zum anderen ist es nicht möglich, mehrere Eingaben gleichzeitig auszulösen, um ein paralleles Systemverhalten zu überprüfen. Wegen dieser Nachteile besteht neben der direkten Simulation die Möglichkeit, sogenannte *Front-End Tools* zu verwenden, mit denen die Simulation bedeutend komfortabeler durchgeführt werden kann (vgl. Kapitel 3.2.3).

Eine Konsistenzüberprüfung der Modelle kann in den meisten Fällen bereits während der Modellierung durchgeführt werden. Häufige Modellierungsfehler werden von den Entwurfswerkzeugen automatisch überprüft. Dies verhindert später eine langwierige Fehlersuche im Programm-Code. Die meisten CASE Tools dieser Art besitzen einen Code-Generator für verschiedene Zielsprachen und Targets. UML-Werkzeuge erzeugen im Allgemeinen einen prototypischen Code zur Simulation, wohingegen Werkzeuge wie ASCET-SD und TargetLink[27] einen Serien-Code generieren und diesen für verschiedene Mikrocontroller optimieren.

Einige Modellierungswerkzeuge wie z. B. *ARTiSAN Real-time Studio* [AGM03] verfügen über integrierte Versionisierungsmechanismen und sind daher von fremden Konfigurationswerkzeugen unabhängig. Für bestimmte Teilprozesse, wie das bereits oben genannte Projektmanagement, Requirements Engineering und Testen, werden Schnittstellen angeboten, so dass eine Integration unterschiedlicher CASE Tools in einen Entwicklungsprozess unproblematisch ist.

Werkzeuge zur Qualitätsüberprüfung

Wie im vorherigen Kapitel erwähnt, wurde eine Vielzahl von analytischen Maßnahmen zur SW-Qualitätssicherung entwickelt. Zur besseren Unterscheidung, können diese in Validations- und Verifikationstechniken aufgegliedert werden. Dabei wird Validation wie folgt definiert:

Validation

Unter Validation wird der Prozess zur Beurteilung eines Systems oder einer Komponente verstanden mit dem Ziel festzustellen, ob der Einsatzzweck oder die Benutzererwartung erfüllt werden [SZ03].

Bei der Validation von Spezifikationen, Modellen und Programmen kommen Methoden wie Simulation/Animation und Tracing zum Einsatz. Reaktive Systeme, die mit Hilfe von Zustandsautomaten oder Petri-Netzen modelliert worden sind, lassen sich durch komfortable Simulatoren validieren. Der Benutzer kann mit dem System interagieren, indem er den momentanen Systemstatus abfragt, Ereignisse ausführt oder Variablen ändert. Durch Einfügung von kontinuierlichen Fahrzeugmodellen ist eine gemeinsame Simulation der Steuergerätefunktionen mit ihrer Systemumgebung möglich (*Software-in-the-Loop*). Teilumfänge des Fahrzeugs können im Echtzeitbetrieb mit realen Steuergeräten gekoppelt werden, um beispielsweise einen Steuergeräteverbund mit seiner Umgebung testen zu können.

Bei der Verifikation werden dagegen andere Techniken wie Konsistenzüberprüfungen, Tests und formale Beweise eingesetzt. Dabei wird unter Verifikation folgendes verstanden:

[27] Vgl. http://www.dspace.de.

Verifikation

Verifikation ist der Prozess zur Beurteilung eines Systems oder einer Komponente mit dem Ziel festzustellen, ob die Resultate einer gegebenen Entwicklungsphase den Vorgaben für diese Phase entsprechen [SZ03].

Mit Hilfe von Konsistenzüberprüfungen können Inkonsistenzen und Redundanzen in Modellen in Form von Regeln analysiert und behoben werden [Glin00]. In der UML lassen sich solche Konsistenzregeln mit Hilfe von OCL formal definieren (vgl. Kapitel 2.2.3.3 auf Seite 34). Konsistenzsicherungen sind zwar in vielen Modellierungswerkzeugen integriert, haben aber den Nachteil, dass sie statisch und unflexibel sind. Sie können deshalb weder deaktiviert noch modifiziert oder ergänzt werden. Aus diesem Grund lässt sich eine benutzerspezifische Überprüfung eigener Modellierungsregeln nicht durchführen. Eine Ausnahme bilden die Modellierungswerkzeuge Simulink/Stateflow, für die das Tool *mint* [GT04] entwickelt wurde. Mit diesem Programm sind eigene Regeldefinitionen und -überprüfungen möglich, die in Kapitel 5.1.1 auf Seite 118 diskutiert werden. Andere kommerzielle Werkzeuge zur Definition und Überprüfung eigener Modellierungsregeln sind dem Autor dieser Arbeit nicht bekannt. Jedoch beschäftigen sich im Automotive-Umfeld zahlreiche Gruppen mit der Realisierung derartiger Modellüberprüfer [MKF00, MH03].

Das Testen auf SW-Ebene und das Diagnostizieren auf HW-Ebene sind die häufigsten Verifikationsmaßnahmen in der SW-Entwicklung. Hierfür gibt es zahlreiche Werkzeuge, die eine automatische Testfallerstellung und Testdurchführung ermöglichen. Von DaimlerChrysler wurde beispielsweise das Werkzeug CTE XL [LW00] entworfen, mit dem eine solche Überprüfung möglich ist. Im Bereich der Diagnose dagegen gab es nur zwei nennenswerte Toolhersteller, die ein Diagnosewerkzeug für eingebettete Systeme anboten. Dabei handelt es sich um die Hersteller OCC'M mit dem Werkzeug *Raz'r* [SSW00] und R.O.S.E. Informatik *mit dem Werkzeug Rodon* [Buro01].

Die formale Verifikation von Modelleigenschaften wie Widerspruchsfreiheit, Vollständigkeit, Erreichbarkeit oder Deadlock wird durch sogenannte Model Checker [Peuk97] realisiert. Ein bekannter Vertreter der Model Checker ist UPPAAL [BLLPY96], mit dem zustandsbasierte Systeme (Timed Automata) grafisch erstellt und verifiziert werden können. Das Problem der bisherigen Model Checker ist, dass sie keine Anbindungen zu den Modellierungswerkzeugen haben. Eine manuelle Umsetzung der Modelle in UPPAAL wurde beispielhaft in [MD02] anhand einer Fallstudie aus dem Kfz-Komfortbereich durchgeführt.

Eine weitere Maßnahme zur Überprüfung der SW-Qualität ist die bereits erwähnte Bewertung durch SW-Metriken. Obwohl die Anwendung von Metriken in der Programmierung seit über 20 Jahren erfolgreich praktiziert wird, sind sie in der modellbasierten Entwicklung kaum vertreten (vgl. Kapitel 5.1.2). Daher sind hier sehr wenige Werkzeuge verfügbar. Ein bekanntes Werkzeug zur Modellbewertung ist das Werkzeug *Modeling Metric* [Hosa04], das aber nur auf die Werkzeuggruppe von MathWorks beschränkt ist.

Kapitel 3

3 Das STEP-X Vorgehensmodell

„Good projects need both ingredients: A dream and a vision to excite developers and customers alike plus the resources and the discipline to make it come true.“

Andreas Rau, 2002

Die Standardisierung der UML zeigt, dass die objektorientierte Modellierung für die Entwicklung qualitativ hochwertiger Anwendungen einen immer größeren Stellenwert erlangt. Im selben Maße, wie sich eine durchgängige Betrachtungsweise der objektorientierten SW-Entwicklung durchsetzt, steigt die Forderung nach geeigneten Vorgehensmodellen. Ein Vergleich derartiger Prozessmodelle findet sich unter anderem in [BK02, NS99, Wolf00]. Der hier verwendete Begriff Prozess ist wie folgt definiert:

Prozess

Der Ablauf eines Vorhabens mit der Beschreibung der Schritte, der beteiligten Personen, der für diesen Ablauf benötigten Informationen und der dabei entstehenden Informationen wird Prozess genannt [Glin02b].

Bei der in dieser Arbeit vorgestellten Entwurfsmethodik liegt das *V-Modell 97* als Vorgehensmodell zugrunde, welches sich bereits bei vielen Automobilherstellern etabliert hat. Des Weiteren besitzt das V-Modell 97 eine Anbindung zu objektorientierten Techniken, so dass die Nutzung von UML-Notationen problemlos möglich ist [SW00, Vers00b].

Bevor in Kapitel 4 auf die modellbasierte STEP-X Entwurfsmethodik näher eingegangen wird, erfolgt in diesem Kapitel zunächst ein Überblick über die Organisationsform und Vorgehensweise des V-Modells 97, in dem die einzelnen Submodelle und die Erzeugnisstruktur vorgestellt werden. Abschließend werden die projektspezifischen Anpassungen des Vorgehensmodells sowie die Werkzeugauswahl und -integration in das V-Modell 97 beschrieben.

3.1 Charakteristik des V-Modells 97

Das deskriptive und präskriptive V-Modell 97 (im Weiteren nur V-Modell) beschreibt einen Entwicklungsprozess mit den dazugehörigen Aktivitäten, Produkten[28] und Rollen. Zusätzlich umfasst es die zur Entwicklung notwendigen Tätigkeitsbereiche Qualitätssicherung, Konfigurations- und Projektmanagement. Zudem lässt sich das V-Modell unter Berücksichtigung einer Entwicklungsstrategie (inkrementell, sequentiell usw.) an die Rahmenbedingungen eines Projektes anpassen. Um eine möglichst universelle Lösung zur SW-Entwicklung anzubieten, wird das V-Modell in drei Ebenen eingeteilt [DW00]:

- Die erste Ebene, das eigentliche Vorgehensmodell [AU250], beschreibt auf Grundlage der Aktivitäten und Produkte welche Tätigkeiten im Rahmen der SW-Entwicklung durchgeführt werden müssen und welche Ergebnisse dabei entstehen sollen (vgl. Kapitel 3.2.1).
- Die zweite Ebene, die Methodenzuordnung [AU251], schlägt für die einzelnen Aktivitäten vor, welche Techniken und Notationen verwendet werden können, um diese Aktivitäten sinnvoll durchzuführen (vgl. Kapitel 3.2.2).
- Die Werkzeuganforderungen [AU252] stellen die dritte Ebene dar. In ihr werden die Anforderungen an die zur Entwicklung benötigten Tools dargestellt, um dem Anwender die Werkzeugauswahl zu erleichtern (vgl. Kapitel 3.2.3).

3.1.1 Submodelle des V-Modells

Für einen erfolgreichen SW-Entwicklungsprozess müssen außer der eigentlichen System-/SW-Erstellung (SE) auch begleitende Projekttätigkeiten, wie Qualitätssicherung (QS), Konfigurations- (KM) und Projektmanagement (PM) berücksichtigt werden. Diese Prozesse werden im V-Modell als eigenständige, sich gegenseitig beeinflussende Submodelle betrachtet. In Abbildung 3-1 ist das Zusammenspiel dieser Submodelle vereinfacht dargestellt. Die Submodelle im V-Modell werden durch Kernaktivitäten beschrieben, die wiederum Teilaktivitäten enthalten. Die Aktivitäten werden durch eine Kombination von Buchstaben und Ziffern eindeutig gekennzeichnet. Dabei geben die Buchstaben die Zuordnung zu den einzelnen Submodellen wieder, und die Ziffern stellen die Bearbeitungsreihenfolge in dem jeweiligen Submodell dar.

[28] Alle Ergebnisse einer Aktivität wie z. B. die zu erstellende Software aber auch die Dokumente, die im Entwicklungsverlauf entstehen, werden im V-Modell als Produkte bezeichnet.

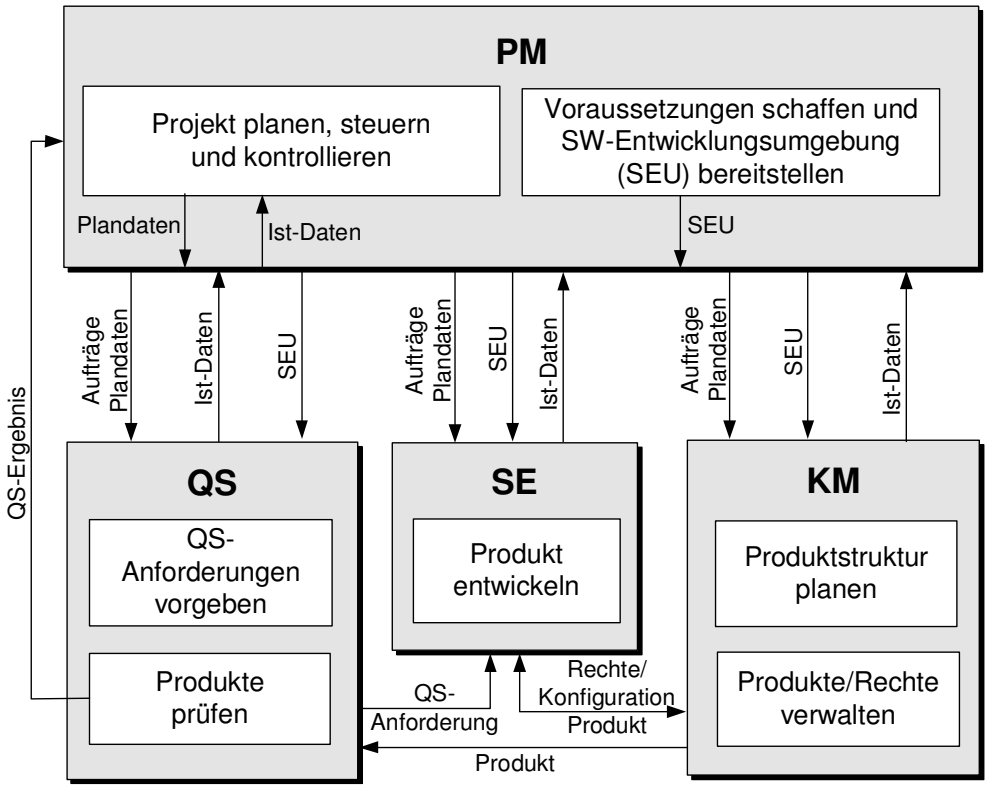


Abbildung 3-1: Zusammenspiel der vier Submodelle im V-Modell [DW00]

Projektmanagement

Das Projektmanagement ist für die Planung und Durchführung des Projekts verantwortlich. Es steuert und kontrolliert die einzelnen Arbeitsschritte und stellt Ressourcen (Hardware, Software, Personal usw.) bereit. Durch den Vergleich von Plan- und Ist-Daten wird der Fortgang des Projekts in regelmäßigen Zeitabständen bewertet. Eine weitere Aufgabe des PMs ist das *Toolset Management*. Hierbei werden alle Werkzeuge und Methoden bereitgestellt, die zur Unterstützung des Prozesses notwendig sind. Das Projektmanagement hat somit die Aufgabe, die Aktivitäten der anderen Submodelle zu triggern und sie mit den notwendigen Informationen zu versorgen. Im Rahmen dieser Arbeit wird insbesondere auf das Toolset Management eingegangen, welches in Kapitel 3.2.3 vorgestellt wird.

Qualitätssicherung

Die Qualitätssicherung umfasst die Planung von präventiven und konstruktiven Maßnahmen zur Verbesserung bzw. Sicherstellung der SW-Qualität. Der Nachweis der SW-Qualität erfolgt durch analytische Maßnahmen, wie z. B. Konsistenzüberprüfung oder Testdurchführung. Die hier verwendeten und entwickelten präventiven und analytischen Maßnahmen zur Qualitätssicherung werden ausführlich in Kapitel 5 vorgestellt.

Konfigurationsmanagement

Im Submodell Konfigurationsmanagement wird sichergestellt, dass alle Produkte während der gesamten Entwicklung entsprechend ISO 12207 gespeichert, verwaltet und zu Konfigurationen zusammengestellt werden können [KMP01]. Durch die automatische Versionierung aller Konfigurationen ist die Erkennung von Zusammenhängen und Unterschieden zwischen früheren und aktuellen Konfigurationen sichergestellt [HMFP+03]. Die individuelle Vergabe von Zugriffsrechten verhindert unerwünschte Datenmanipulationen. Ein regelmäßiges Daten-Backup schützt zusätzlich vor Datenverlust.

Im STEP-X Projekt werden diesbezüglich alle Produkte, wie Dokumente, Modelle oder Quelltexte mit Versionsnummern, Datum und Bearbeiter versehen und anschließend in ein zentrales Repository für alle Beteiligten zur Verfügung gestellt, wobei Zugriffsrechte individuell vergeben werden müssen. Das Versionierungswerkzeug *WinCVS*[29] dient als gemeinsames Repository, bei dem alle Änderungen automatisch protokolliert werden.

An dieser Stelle wird auf das KM nicht weiter eingegangen, da die oben beschriebenen Konfigurations- und Verwaltungsmaßnahmen bereits von den Entwicklungswerkzeugen automatisch vorgenommen werden.

Systemerstellung

Das Submodell Systemerstellung bildet den Kern des V-Modells, in dem es den eigentlichen SW-Entwicklungsprozess in einer V-förmigen Anordnung von Aktivitäten beschreibt. Durch diese Anordnung bekam das V-Modell seinen Namen. Zum besseren Verständnis wird nachfolgend unter dem Begriff V-Modell das Submodell SE verstanden.

In der Abbildung 3-2 ist das V-Modell mit seinen Kernaktivitäten dargestellt. Es beschreibt auf der linken Seite eine stufenweise Verfeinerung des Gesamtsystems durch das Top-Down-Verfahren. Der erste Schritt bei der Systemerstellung ist die Systemanforderungsanalyse (SE 1), bei der die Erwartung des Anwenders in ein Anforderungsdokument (Lastenheft) niedergeschrieben wird. Dieses dient als Basis für das weitere Vorgehen. Beim Systementwurf (SE 2) wird das System in weitere Teilstrukturen (vgl. Kapitel 3.1.2) zerlegt und durch technische Anforderungen präzisiert (SE 3). Von hier ab spaltet sich der weitere Fortgang in die SW-Entwicklung und ggf. in die HW-Entwicklung. Die Erstellung der Software wird in den Phasen SW-Grobentwurf, SW-Feinentwurf und SW-Implementierung (SE 4 bis SE 6) durchgeführt.

Die rechte Seite des V-Modells umfasst die Integrationsphasen, bei denen die einzelnen Strukturelemente durch das Bottom-Up-Verfahren schrittweise zu einem Gesamtsystem zusammengesetzt werden (SE 7 und SE 8). Während dieser Tätigkeiten müssen auch Informationen über die Hardware berücksichtigt werden. Zusätzlich erfolgen in jedem Integrationsschritt eine Verifikation sowie eine Validation. Die Überleitung in die Phase Nutzung (SE 9) beschreibt alle Tätigkeiten, die notwendig sind, um ein fertig gestelltes System an der vorgesehenen Einsatzstelle zu installieren und in Betrieb zu nehmen.

[29] Vgl. http://www.wincvs.org

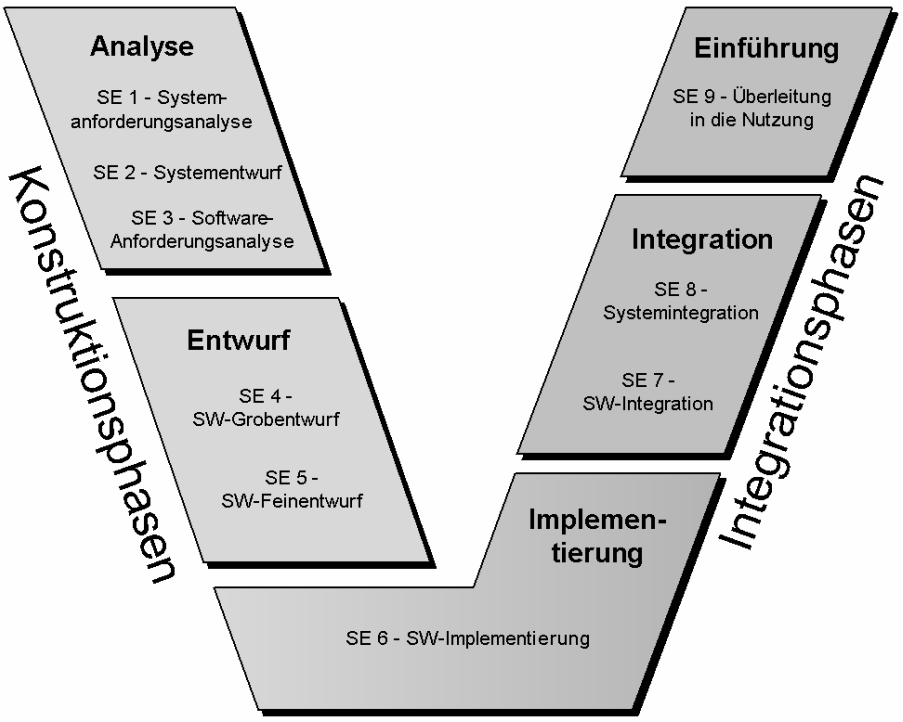


Abbildung 3-2: Das Submodell Systemerstellung

Die in dieser Arbeit vorgestellte Entwurfsmethode ist speziell auf die Konstruktionsphasen des V-Modells ausgerichtet. Eine Auflistung aller benötigten Aktivitäten für diese Phasen erfolgt in Kapitel 3.2.1. Zum besseren Verständnis werden im späteren Entwicklungsprozess (vgl. Kapitel 4) die ersten drei Kernaktivitäten in den Phasen Anforderungsbeschreibung und Analyse zusammengefasst. Die Integrationsphasen mit den Aktivitäten zur Fertigstellung des Systems sollen in dieser Ausarbeitung nur eine untergeordnete Rolle einnehmen, da sie sonst den Rahmen der Arbeit sprengen würden.

3.1.2 Erzeugnisstruktur des V-Modells

Mit Hilfe der Erzeugnisstruktur des V-Modells wird festgelegt, aus welchen generischen Bausteinen das zu entwickelnde System bestehen soll. Die wichtigsten Elemente zur Bildung von SW-Architekturen stellt die Erzeugnisstruktur dar:

- ***SW-Module:*** SW-Module sind gekennzeichnet durch die Umsetzung der typischen Modularisierungskriterien wie Abgeschlossenheit, schwache Kopplung zwischen Modulen und enge interne Bindung zwischen Funktionen. Sie entsprechen in der Objektorientierung den Klassen, die mittels Schnittstellen die interne Struktur kapseln und nach außen ein genau definiertes Verhalten repräsentieren.
- ***SW-Komponenten:*** Besteht eine Funktionalität aus mehreren SW-Modulen, so können diese gemeinsam zu einer SW-Komponente gruppiert werden. Genauso wie ein SW-Modul, verfügt eine SW-Komponente über eine oder mehrere wohldefinierte Schnittstellen. Durch diese Art von Kapselung können Funktionsgruppen von mehreren SW-Einheiten verwendet werden, wodurch die Forderung nach Wiederverwendung erfüllt wird.

- ***SW-Einheiten:*** Eine SW-Einheit stellt eine ausführbare Funktionseinheit dar, welche sich im Allgemeinen aus mehreren SW-Modulen bzw. SW-Komponenten zusammensetzt. Ein (Sub-)System besteht aus mindestens einer Einheit.

Abbildung 3-3 stellt den Zusammenhang der Erzeugnisstrukturelemente grafisch dar. Das V-Modell liefert im Wesentlichen nur diese grobe Erzeugnisstruktur als Architekturbeschreibung. Eine Zuordnung von Aktivitäten zur Erstellung eines Architekturmodells mit entsprechenden Detaillierungsgraden muss daher zusätzlich durchgeführt werden. Ein derartiges Vorgehen zur Gliederung einer SW-Architektur wird in Kapitel 4.3.1 vorgestellt.

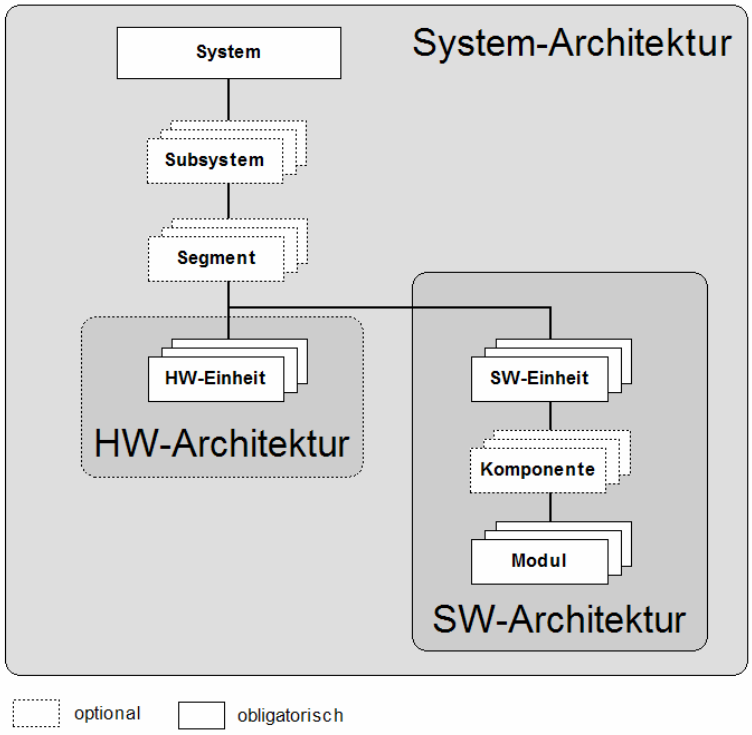


Abbildung 3-3: Erzeugnisstruktur des V-Modells

3.1.3 Vorgehensweise im V-Modell

Die hier vorgestellte Vorgehensweise im V-Modell bezieht sich auf den objektorientierten SW-Entwicklungsprozess, bei dem Notationen und Techniken der Objektorientierung verwendet werden. Die zum Einsatz kommenden Aktivitäten werden vorwiegend inkrementell durchgeführt [Burk98]. Die inkrementelle Vorgehensweise basiert auf der Grundidee, dass ein System in seiner Gesamtheit geplant und die Realisierung dann in mehreren Stufen (Inkrementen) durchgeführt wird, so dass die Funktionalität des Systems schrittweise anwächst. Diese Vorgehensweise ermöglicht frühzeitig den Anwendern eine erste Systemversion zur Verfügung zu stellen, die bereits die Grundfunktionalität des Systems abdeckt. Des Weiteren wird das inkrementelle Konzept durch folgende Ansätze erweitert:

- ***Anwendungsgetriebener Ansatz:*** Es werden aufgrund von Anforderungen des Auftraggebers Anwendungsfälle definiert. Auf diese Weise erhält man die Möglichkeit, den zu betrachtenden Ausschnitt des Gesamtsystems einzuschränken.

- ***Evolutionärer Ansatz:*** Dabei wird darauf geachtet, dass möglichst bald sichtbare Zwischenergebnisse erzielt werden, die entweder einen schmalen Ausschnitt nahezu komplett umsetzen oder das nahezu komplette Anwendungsgebiet überblickartig realisieren.
- ***Ereignisorientierter Ansatz:*** Die SW-Entwicklung läuft im Normalfall nicht nach einer fest vorgegebenen Reihenfolge, sondern muss auf Ereignisse (z. B. neue Anforderungen des Auftraggebers) reagieren können.
- ***Architekturzentrierter Ansatz:*** Dieser Ansatz beruht auf der Wiederverwendung und dem Einsatz bekannter Architekturen wie logische Architektur (Klassendiagramme) und physische Architektur (Verteilungsdiagramme).

Eine detailliertere Erläuterung zu diesen und weiteren Ansätzen ist [Burk98, MFW97] zu entnehmen. Darüber hinaus werden in [KM00, Müll98, Müll99a] UML-basierte Verfahren beschrieben, mit denen sich eingebettete Systeme in V-Modellen entwickeln lassen.

3.2 Projektspezifische Anpassung des V-Modells

Das V-Modell definiert für die Durchführung eines Entwicklungsprozesses eine Vielzahl von Aktivitäten und beschreibt die Produkte, die daraus erzeugt werden. Für eine konkrete Anwendung ist jedoch in der Regel nur eine Teilmenge der zur Verfügung stehenden Aktivitäten erforderlich. Die restlichen, nicht benötigten Aktivitäten und Produkte können durch sinnvolle Begründung gestrichen werden. Eine solche Anpassung des V-Modells an das konkrete Projekt wird Tailoring[30] genannt. Bei der Auswahl der projektspezifischen Aktivitäten wird unter dem Aspekt einer hinreichenden Qualität nach ISO 9001 vorausgesetzt, dass bestimmte Mindestanforderungen eingehalten werden. Bei der durchgeführten Streichung muss geprüft werden, ob die Mindestanforderungen eingehalten sind und ob die Streichung ausreichend begründet ist. Nach der Auswahl von geeigneten Aktivitäten müssen diesen passende Methoden mit den dazugehörigen Werkzeugen zugeordnet werden. Nachfolgend werden sowohl die Ergebnisse des Tailorings als auch der Methodenzuordnung sowie die Werkzeugauswahl in Bezug auf die zu entwickelnde Methodik präsentiert. Da das Tailoring-Verfahren ein aufwendiger Prozess ist, soll im Rahmen dieser Arbeit das Tailoring auf das Submodell SE begrenzt werden.

3.2.1 Tailoring der Aktivitäten im SE-Modell

Wie bereits in Kapitel 3.1.1 erwähnt, erfolgt im Submodell SE die eigentliche Systemerstellung, für die das V-Modell Aktivitäten und dazugehörige Produkte definiert. Tabelle 3-1 enthält einen Ausschnitt der für die STEP-X Methodik benötigten Kern- und Teilaktivitäten des Submodells SE. Eine vollständige Auflistung aller Aktivitäten des SE-Modells sowie die vom V-Modell geforderten Streichbegründungen befinden sich im Anhang A.

[30] Tailoring bedeutet begründetes Weglassen.

Aktivitäten im SE-Modell
SE 1 Systemanforderungsanalyse
Stellt die Grundlage für das zu erstellende System dar.
SE 1.2 Anwendungssystem beschreiben Dies geschieht in Form einer groben Systembeschreibung, die den Rahmen für alle weiteren Verfeinerungen und Ergänzungen der Anwenderforderungen bildet.
SE 1.5 System fachlich strukturieren Die fachliche Systemstruktur wird nach Gesichtspunkten des Anwenders modelliert. Für die Darstellung der Funktionsweise des Systems werden Ablaufbeschreibungen festgelegt, die das Zusammenwirken des Anwenders mit dem System darstellen.
SE 2 Systementwurf
Auf der Basis der fachlichen Anforderungen wird das Gesamtsystem grob erstellt.
SE 2.1 System technisch entwerfen Es wird eine technische Architektur erstellt, in der das System in SW-Einheiten zerlegt wird.
SE 2.5 Schnittstellen beschreiben Die Beschreibung der Systemarchitektur endet mit der Identifikation aller Schnittstellen des Systems zu seiner Umgebung.
SE 2.6 Systemintegration spezifizieren Im Integrationsplan wird festgelegt, wie und wann die in der Systemarchitektur definierten Elemente zu einem Gesamtsystem integriert werden.
SE 3 Anforderungsanalyse für SW-Einheiten
Die SW-Einheiten des Gesamtsystems werden spezifiziert.
SE 3.3 Anforderungen an die Funktionalität definieren Die von der SW-Einheit zu realisierende Funktionalität wird identifiziert.
SE 3.4 Anforderungen an die Qualität definieren Auf der Basis DIN ISO 9126 werden Anforderungen bzgl. der nichtfunktionalen Qualitätsmerkmale festgelegt.
SE 4 SW-Grobentwurf
Die SW-Einheiten werden weiter in ihre Bestandteile zerlegt und durch Verhalten ergänzt.
SE 4.1 SW-Architektur entwerfen Hier wird die Zerlegung in SW-Komponenten und SW-Module durchgeführt und diese mit internen Schnittstellen versehen. Neben der Beschreibung der statischen Systemarchitektur wird durch Verhaltensmodellierung die Systemdynamik spezifiziert.
SE 5 SW-Feinentwurf
Die SW-Komponenten und SW-Module werden hinsichtlich ihrer Umgebung, der Realisierung ihrer Funktionalität, der Datenhaltung und Ausnahme- und Fehlerbehandlung spezifiziert.
SE 5.1 SW-Modul/SW-Komponenten beschreiben Neben der Spezifizierung der Programmiervorgaben wird in erster Linie das physikalische Design betrieben. Die zuvor gebildete grobe SW-Struktur und das grobe Systemverhalten werden weiter verfeinert.
SE 6 SW-Implementierung
Die SW-Vorgaben werden in Programme umgesetzt und geprüft.
SE 6.1 SW-Module kodieren Aus den Modulen wird automatisch Code generiert. Einige Code-Fragmente müssen evtl. manuell angepasst und erweitert werden. Dies wird im Programm sowie in externen Dokumenten kommentiert.
SE 6.3 Selbstprüfung des SW-Moduls durchführen Vom Entwickler sind Selbstprüfungen der von ihm realisierten SW-Module durchzuführen. Nach der Prüfung erfolgt die Auswertung der Ergebnisse, die den weiteren Entwicklungsverlauf bestimmt. Nach einer erfolgreichen Überprüfung erfolgt die Weitergabe des Prüfgegenstandes an die QS zum Zweck der formellen Produktprüfung.

Tabelle 3-1: Ausgewählte Aktivitäten des SE-Modells

3.2.2 Zuordnung von elementaren Methoden

Durch das Tailoring wurde zunächst eine sinnvolle Auswahl von Aktivitäten für den SW-Entwurf vorgenommen. Im nächsten Schritt sind geeignete Methoden für diese Aktivitäten zu ermitteln und zuzuordnen. Hierfür stellt das V-Modell eine Methodenzuordnung [AU251] zur Verfügung. Diese definiert keine Methoden, sondern regelt den Einsatz von Methoden in allgemeingültiger Weise. Um diese Allgemeingültigkeit zu erreichen, beschränken sich die Festlegungen der Methodenzuordnung auf elementare Methoden.

Elementarmethode

Als Elementarmethoden werden die Vorgehensweisen bezeichnet, die eine spezifische, abgegrenzte Sicht des Systems bzw. eine bestimmte Phase der Systementwicklung beschreiben.

Durch die Beschränkung auf elementare Methoden ist sichergestellt, dass das V-Modell unabhängig von technischen Randbedingungen, wie Notationen und Werkzeugen ist.

Bereits zu Beginn der UML wurde im V-Modell-Umfeld [Rein97] die Bedeutung von UML als Sammlung wichtiger objektorientierter Elementarmethoden[31] erkannt. Die UML-Elementarmethoden flossen daher in die Methodenzuordnung ein. Beispiele hierfür sind *Use-Case-Modellierung* und *Klassenmodellierung*. Die im V-Modell aufgeführten Elementarmethoden beeinflussen sich gegenseitig im Modellierungsprozess. Demzufolge werden im V-Modell Schnittstellen zwischen diesen objektorientierten Elementarmethoden definiert (vgl. Abbildung 3-4).

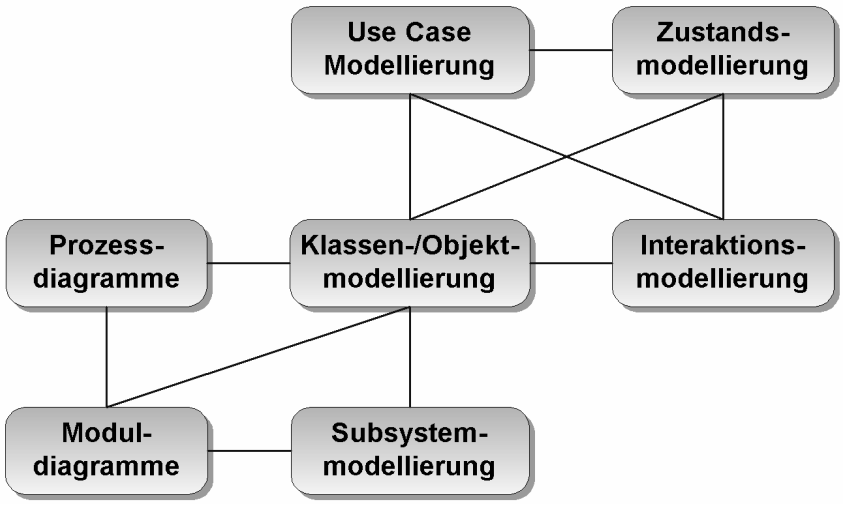


Abbildung 3-4: Schnittstellen objektorientierter Methoden im V-Modell

Werden beispielsweise in einem Entwicklungsprozess die Elementarmethoden Use Case-Modellierung, Klassenmodellierung und Zustandsmodellierung verwendet, so können die-

[31] Obwohl die UML keine Methode im eigentlichen Sinne ist, enthält sie eine Sammlung von Elementarmethoden.

se nicht unabhängig voneinander betrachtet werden. Vielmehr muss eine Gesamtsicht der voneinander abhängigen Elementarmethoden bestehen, um die Konsistenz zwischen den Modellen zu bewahren. Andere Kombinationen von Elementarmethoden dagegen wie beispielsweise *Interaktionsmodellierung* und *Moduldiagramme* beeinflussen sich im Modellierungsprozess nicht und können gänzlich unabhängig voneinander verwendet werden.

In Tabelle 3-2 sind den bereits ausgewählten Aktivitäten elementare Methoden zugewiesen. Eine detaillierte Zuweisung erfolgt in Kapitel 4.

Aktivität	Methode
SE 1 Systemanforderungsanalyse	
SE 1.2 Anwendungssystem beschreiben	KM, SM, UM
SE 1.5 System fachlich strukturieren	KM, SM, IM, UM
SE 2 Systementwurf	
SE 2.1 System technisch entwerfen	SM, IM KM, PD
SE 2.5 Schnittstellen beschreiben	SM, KM, IM
SE 2.6 Systemintegration spezifizieren	Balkendiagramm
SE 3 Anforderungsanalyse für SW-Einheiten	
SE 3.3 Anforderungen an die Funktionalität definieren	KM, IM, CRC-Karten
SE 4 SW-Grobentwurf	
SE 4.1 SW-Architektur entwerfen	KM, SM, IM, ZM
SE 5 SW-Feinentwurf	
SE 5.1 SW-Modul/SW-Komponenten beschreiben	KM, OM, ZM, MD

IM = Interaktionsmodellierung, KM = Klassenmodellierung, MD = Moduldiagramme, OM = Objektmodellierung, PD = Prozessdiagramme, SM = Subsystemmodellierung, UM = Use Case-Modellierung, ZM = Zustandsmodellierung

Tabelle 3-2: Zuweisung von Elementarmethoden im V-Modell

3.2.3 Integration von CASE Tools

Für die SW-Entwicklung von eingebetteten Systemen existiert auf dem Markt eine Vielzahl von CASE Tools[32]. Eine Auswahl geeigneter Werkzeuge ist daher ein langwieriger Prozess. Zur Vereinfachung der Werkzeugauswahl wurden im STEP-X Projekt zunächst Werkzeuge selektiert, die sich bereits in der Entwicklung von eingebetteten Systemen [BBK98] und insbesondere im Automobilbereich [Hofm02] erfolgreich etabliert haben. Zusätzlich wurden frühere Evaluierungsergebnisse anderer Projekte bei der Werkzeugauswahl berücksichtigt. Beispielsweise erfolgt im EvaLUM Projekt [KF03] eine Analyse zahlreicher UML-Modellierungswerkzeuge anhand eines umfangreichen Kriterienkatalogs [Kirc01]. In [KKPS99] wurden unterschiedliche Beschreibungstechniken mit den dazugehörigen Werkzeugen untersucht, die sich zur Spezifikation reaktiver und verteilter Systeme eignen. Des Weiteren evaluierte [SHPR+03] nur solche CASE Tools, die im Automobilbereich bereits erfolgreich zum Einsatz kamen. Eine weitere Evaluierungsstudie wurde im COMTESSA Projekt durchgeführt. Dabei stand die Kopplung zwischen Modellier-

[32] Bereits heute sind über 100 unterschiedliche UML Tools verfügbar, die unter anderem auf den Web-Seiten *http://www.jeckle.de/umltools.html* und *http://www.objectsbydesign.com* beschrieben sind.

ungs- und Diagnosewerkzeugen im Vordergrund (vgl. Kapitel 1.1.3).

Grundsätzlich muss bei früheren Evaluierungsergebnissen darauf geachtet werden, dass die Versionen der untersuchten Werkzeuge möglichst aktuell sind, da durch neuere Updates viele Mängel behoben und zusätzliche Funktionen hinzugefügt werden können.

Das V-Modell bietet mit dem Dokument *Funktionale Werkzeuganforderungen* [AU252] eine Hilfestellung zur Auswahl von geeigneten Werkzeugen für den projektspezifischen SW-Entwicklungsprozess. Zu Beginn des STEP-X Projekts wurde somit eine Vorauswahl der in Frage kommenden Modellierungswerkzeuge getroffen, die in Tabelle 3-3 zu sehen ist.

Hersteller	Werkzeug	Paradigma
Artisan Software	Artisan Real-time Studio™	diskret
ETAS	ASCET-SD™	kontinuierlich/diskret
I-Logix	Statemate MAGNUM™	diskret
	Rhapsody in Micro C™	diskret
	Rhapsody in C/C++™	diskret
Rational	Rational Rose RT™	diskret
Telelogic	Telelogic Tau Suite™	diskret
The MathWorks	MATLAB™/Simulink™/Stateflow™	kontinuierlich/diskret

Tabelle 3-3: Mögliche Modellierungswerkzeuge für den Entwicklungsprozess

In der Evaluierung [BEHH+03] erfolgt eine nähere Betrachtung dieser acht Modellierungswerkzeuge. Dabei wurden die meisten Werkzeugeigenschaften aus Kapitel 2.3.1 mit einbezogen und die folgenden Mindestanforderungen berücksichtigt:

- Realisierung eines objektorientierten Ansatzes mit UML,
- Berücksichtigung aller notwendigen UML-Diagramme,
- Modellierung von zustandsorientierten und regelungstechnischen Systemen,
- Simulation und Animation des Systemverhaltens,
- Datenaustausch bzw. Modellkopplung über standardisierte Schnittstellen,
- Schnittstellen zu anderen Tools (z. B. Anforderungserfassung, Konfiguration),
- Codegenerierung für Mikrocontroller C167 und
- Nachvollziehbarkeit des SW-Entwurfs über alle Entwicklungsphasen.

Eine detaillierte Auflistung aller berücksichtigten Anforderungen befindet sich in Anhang B, wobei die kategorisierten Kriterien je nach Relevanz mit entsprechenden Gewichtungen versehen worden sind. Bei der durchgeführten Evaluierung stellten sich signifikante Unterschiede zwischen den Modellierungswerkzeugen heraus. So eignen sich bestimmte Tools mehr für die früheren Phasen Analyse und Entwurf, andere dagegen mehr für die Implementierung und das Testen. Des Weiteren unterliegen die Modellierungswerkzeuge bestimmten Paradigmen, so dass beispielsweise entweder funktionsorientierte oder objektorientierte Techniken eingesetzt werden können. Diese Techniken haben wiederum

Einfluss auf die Verhaltensmodelle, die entweder eine kontinuierliche/zeitgetriebene oder diskrete/ereignisgetriebene Ausführungssemantik aufweisen.

Diese und weitere grundsätzliche Unterschiede zwischen den Werkzeugen führten zu der Schlussfolgerung, dass der komplette Entwicklungsprozess nur von mehreren miteinander kooperierenden Werkzeugen zu realisieren ist. In Abbildung 3-5 ist der linke Zweig des V-Modells mit seinen Konstruktionsphasen abgebildet, in die die ausgewählten CASE Werkzeuge integriert sind.

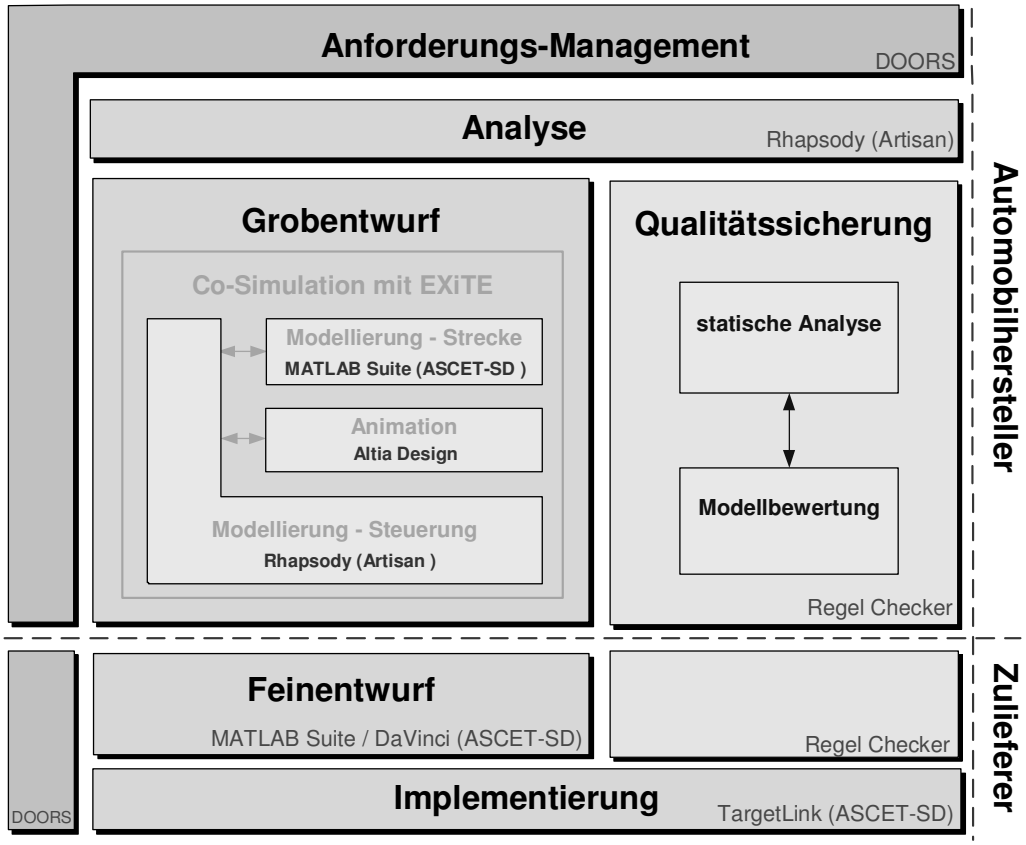


Abbildung 3-5: Die Werkzeugkette im Vorgehensmodell

Die Anforderungserfassung und -analyse erfolgt durch das Werkzeug *DOORS*[33]. In ihm werden die textuellen Anforderungen strukturiert und zueinander in Beziehung gesetzt. Ferner dient DOORS als Back Bone für den gesamten Entwicklungsprozess, wobei die Produkte der einzelnen Phasen mit den Anforderungen verlinkt werden. In den Phasen Analyse und Grobentwurf kommen objektorientierte Techniken und Notationen zum Einsatz. Daher wurde für diese Phasen das UML-Werkzeug Rhapsody ausgewählt. Für die Modellierung von kontinuierlichen regelungstechnischen Systemen reichen die UML-Notation und die UML-basierten Werkzeuge nicht aus [PBLF+00]. Aus diesem Grund erfolgte zusätzlich die Integration der Werkzeuggruppe MATLAB Toolbox, die Blockdiagramme und Datenflüsse bereitstellt. Die Validierung der diskreten Systemanteile wird durch animierte Simulationsläufe der Verhaltensmodelle realisiert. Zu diesem Zweck steht

[33] Für das Requirement Engineering/Management wurde eine weitere Evaluierung [GHMP02] durchgeführt.

das Front-End-Tool *Altia Design* zur Verfügung. Um eine Simulation des gesamten Systems zu ermöglichen, ist eine Kopplung zwischen der diskreten Steuerung, der kontinuierlichen Strecke und ggf. der grafischen Simulations- und Animationsumgebung notwendig. Die Kopplung und Co-Simulation der verteilten Anwendungen ermöglicht das Werkzeug ExITE.

Bis zu diesem Punkt der SW-Entwicklung ist das Ergebnis eine ausführbare Spezifikation des SW-Systems, die als *Digitales Lastenheft* bezeichnet wird. Letzteres enthält textuelle untereinander verlinkte und strukturierte Anforderungen sowie eine Anbindung an die Modelle. Die für die Erstellung des Lastenhefts notwendigen Aktivitäten im Vorgehensmodell sollen mittelfristig von den Automobilherstellern eigenständig durchgeführt werden, da einerseits die Anforderungen an das zu entwickelnde System beim Hersteller entstehen und andererseits an dieser Stelle der Entwicklung kein spezielles Wissen über Hardware, Kommunikation und Betriebsysteme notwendig ist.

Die Weitergabe des Digitalen Lastenhefts an die Zulieferer geschieht in Form einer Freigabe auf ein gemeinsames Repository. Diese zentralen Daten dienen als Grundlage für den Feinentwurf, bei dem die Modelle verfeinert, an spezifische Steuergeräte angepasst und auf verschiedene logische Steuergeräte partitioniert werden. Im Gegensatz zu MATLAB-Modellen, sind die UML-Werkzeuge momentan für den Feinentwurf ungeeignet. Einerseits können keine regelungstechnischen Systemanteile dargestellt werden, andererseits ist die Code-Generierung für Mikrocontroller ungenügend. Code-Optimierungen und Anbindung an verschiedene Betriebssysteme sowie die Definition von Busnachrichten sind nur einige Punkte, die derzeit nicht möglich sind. In einigen Arbeiten, wie [FM01, HMP04, Kühl01, Proc03, Schul04] wird bereits an möglichen Lösungsansätzen gearbeitet, bei denen unterschiedliche Werkzeuge gemeinsam betrieben werden können.

Bei dem hier vorgestellten Vorgehen wird nur aus den MATLAB-Funktionsblöcken ein geeigneter C-Code generiert, der in das Werkzeug DaVinci integriert wird. Anschließend werden notwendige Betriebssystemroutinen und Treiber angebunden sowie die Kommunikationsmatrix definiert. In der Implementierungsphase wird mit Hilfe des Werkzeugs TargetLink der gesamte C-Code für das gewünschte Target aus dem DaVinci-Modell automatisch generiert. Bei Bedarf können diese Aktivitäten von den OEMs ebenso durchgeführt werden wie die vorherigen, wobei langfristig eine totale Unabhängigkeit von den Zulieferern mit deren Know-how nicht möglich sein wird und von Herstellerseite auch nicht gewünscht ist.

Neben dem eigentlichen Modellierungsprozess verläuft parallel die SW-Qualitätssicherung, in der die zustandsbasierten Verhaltensmodelle beispielsweise auf bestimmte statische Eigenschaften überprüft werden können. Derartige analytische Maßnahmen werden von dem Werkzeug *Regel Checker* übernommen (vgl. Kapitel 5.2).

Die Möglichkeit, unterschiedliche CASE Tools im Entwicklungsprozess einsetzen zu können, um eine möglichst hohe Flexibilität und Unabhängigkeit von Werkzeugherstellern zu erreichen, ist eine wesentliche Eigenschaft der Entwicklungsmethodik. Daher können in diesem Vorgehensmodell prinzipiell weitere UML-Werkzeuge eingesetzt werden (vgl.

Tabelle 3-3 auf Seite 55). In STEP-X wurden in zwei Fallstudien die Modellierungswerkzeuge MATLAB Toolbox und Rhapsody durch die beiden Tools ASCET-SD von ETAS [ETAS00] und Artisan Real-time Studio von ARTiSAN [MC00] ersetzt, um die Flexibilität der Methodik darzustellen. In dieser Arbeit sollen sie aber nur am Rande betrachtet werden. Für Interessierte sei auf [Dett03, Stei02, Wern03] verwiesen.

Im Folgenden werden die für die Entwurfsmethodik ausgewählten CASE Tools näher erläutert.

DOORS für das Requirements Engineering/Management

DOORS™ 6.0 (im Weiteren DOORS) von Telelogic ist in der Automobilindustrie ein verbreitetes Werkzeug für das Requirements Engineering/Management. Es dient der Erfassung, dem Nachvollziehen und der Verwaltung von Anforderungen. DOORS basiert auf einem datenbankgestützten Repository, das über Zugriffskontrollmechanismen verfügt sowie das Versions- und Konfigurationsmanagement unterstützt. Die Erfassung der natürlichsprachlichen Anforderungen geschieht in textueller Form. Durch Verlinkung der Anforderungen an die Modelle wird das Tracing während des gesamten Entwicklungsprozesses ermöglicht. Mit der von DOORS eigenen Skriptsprache DXL (DOORS eXtension Language) können individuelle add-ons, wie z. B. CASE Tool Interfaces realisiert werden. Ein für das STEP-X notwendiges add-on zum automatischen Testen der Benutzeranforderungen wurde in STEP-X von [Nien04] implementiert. Hierbei wurde eine Schnittstelle zum Austausch von Daten zwischen MATLAB und DOORS realisiert. Des Weiteren ist es möglich verschiedene benutzerspezifische Filter zu definieren, um bestimmte Sichten auf die Anforderungsstruktur zu erhalten. Die hierarchische Gliederung sowie die Einbindung von Tabellen und Grafiken tragen zum besseren Verständnis der Lastenhefte bei. In Kapitel 4.1 werden methodische Aktivitäten zur Erfassung und Analyse von Benutzeranforderungen vorgestellt.

Das UML Tool Rhapsody zur Modellierung von diskreten Systemen

Rhapsody™ in C++ 4.3 (im Weiteren Rhapsody) ist ein UML-basiertes Modellierungswerkzeug für objektorientierte Analyse und Design [Hare02]. Es bietet eine graphische Entwicklungsumgebung, in der die Struktur und das Verhalten des SW-Systems für eingebettete Systeme analysiert, entworfen und simuliert werden können. Dabei unterstützt es die von der OMG vorgegebenen UML-Diagramme [OMG01b]. Die gesamten Modelldaten eines Projekts werden konsistent in einem Repository verwaltet. Für die Validierung der SW-Systeme, können in Rhapsody die Statecharts schrittweise simuliert werden, wobei den Benutzern debuggingähnliche Mechanismen zur Verfügung stehen. Für das Forward Engineering bietet der integrierte Code-Generator mehrere Zielsprachen für die PC-Plattform als auch für einige wenige Mikrokontroller an. Der Anwender muss sich allerdings schon zu Beginn der Entwicklung auf eine Sprache festlegen, da Modelle zwischen den verschiedenen Rhapsody-Varianten (Rhapsody in C, Rhapsody in C++ und Rhapsody in Java) nicht kompatibel sind. Eine XMI-Schnittstelle für den Import/Export ist genauso vorhanden, wie ein Reportgenerator zur Dokumentation von Modellen. Eine automatische

Überprüfung der Modelle auf häufige Syntaxfehler im Modell wird angeboten, ist allerdings vom Benutzer nicht erweiterbar. Eine Schnittstelle zu DOORS und einigen Konfigurations-Tools ist vorhanden.

Die MATLAB Toolbox zur Erstellung des Streckenmodells

Bei der MATLAB Toolbox von The MathWorks handelt es sich um eine Entwicklungsumgebung mit einer erweiterbaren Sammlung von Funktionen für die Modellierung von kontinuierlichen Regelungssystemen. Zusätzlich können reaktive Systemanteile mit einfachen ausführbaren Zustandsautomaten spezifiziert werden. Für die Erstellung solcher hybrider Systeme, sind in der MATLAB Toolbox drei kooperierende Werkzeuge integriert:

- MATLAB™ 7.0 ist eine Skriptsprache und zugleich eine grafische Umgebung zur Algorithmenentwicklung, Visualisierung und Analyse von Daten sowie für numerische Berechnungen. Es bildet somit eine Basis für weitere Anwendungen.
- Simulink™ 6.0 ist speziell für die Simulation von dynamischen Systemen entwickelt worden. Es implementiert eine grafische Sprache in Form von Datenflussgraphen mit Blockdiagrammen. Eine funktionale Dekomposition wird durch das Konzept der Subsysteme ermöglicht. Ein Subsystem ist ein kontinuierlicher, diskreter oder hybrider Funktionsblock, der andere Blöcke kapselt und sich nach außen als Black-Box mit wohldefinierten Ein- und Ausgängen präsentiert. Die Dekomposition ist rein zur Visualisierung gedacht und hat weder Einfluss auf das Systemverhalten noch auf die Code-Generierung. Eine detaillierte Beschreibung zu Simulink befindet sich in [ABRW04, Hoff98].
- Stateflow™ 6.0 erweitert die Funktionalität von Simulink durch das Hinzufügen einfacher Statecharts, mit denen diskrete Steuerungs-Software modelliert und simuliert werden kann. Obwohl die Stateflow-Statecharts über hierarchische und parallele Zustände verfügen, unterscheidet sich die Semantik sowie das Ausführungsmodell von Stateflow wesentlich von den UML Statecharts. Die in [HR04] erschienene formale Semantik gibt einen Überblick über das Verhalten der Stateflow-Modelle.

Mit dem separat erhältlichen TargetLink™ von *dSPACE* kann C-Code aus der MATLAB Toolbox für alle marktrelevanten Mikrocontroller generiert [SHPR+03] und durch spezielle Hardware unter Echtzeitanforderungen getestet werden.

Validierung des Systems mit dem Front-End-Tool Altia Design

Altia Design™ 5.0 (im Weiteren Altia) der Firma Altia ist eine Entwicklungsumgebung zur Erstellung von interaktiven, grafisch animierbaren Benutzeroberflächen (engl.: GUI, Graphical User Interface). Eine solche GUI dient entweder als virtueller Prototyp oder wird direkt in ein eingebettetes System implementiert. Um möglichst schnell und komfortabel erste Ergebnisse zu erzielen, verfügt Altia über eine Vielzahl vorgefertigter Grafikelemente, die in Bibliotheken zusammengefasst sind. Darüber hinaus werden domänenspezifische Bibliotheken wie z. B. Kfz-Komfortsysteme, Flugzeugsimulatoren oder Multime-

dia-Geräte zur Verfügung gestellt. Die bereits vordefinierten Elemente in einer solchen Automotive-Bibliothek reichen von einfachen Schaltern, Tastern und Kontrolllämpchen bis hin zu komplexeren Elementgruppen, wie Außenspiegel, Autoradio und verschiedenen Armaturenanzeigen. Alternativ können benutzerspezifische Elemente mit individuellen Eigenschaften versehen werden, so dass unter bestimmten Bedingungen die Farbe, Position und Größe der jeweiligen Elemente zur Laufzeit verändert werden kann. Grundsätzlich lassen sich die Elemente durch *drag & drop* definieren, jedoch kann für komplexeres Systemverhalten eine dafür vorgesehene Skriptsprache verwendet werden. Für die Kopplung der GUI mit den zu simulierenden Modellen besitzt Altia Schnittstellen zu verschiedenen Modellierungswerkzeugen. In Kapitel 4.3.3 wird eine Benutzeroberfläche für das Komfortsystem erstellt und mit den Steuerungsmodellen aus Rhapsody verknüpft.

Co-Simulation mit dem Werkzeug ExITE

ExITE™ 1.3.2 (Extessy Inter Tool Engineering, im Weiteren ExITE) ist eine universelle Simulationsplattform der Firma Extessy[34], die die Datenübertragung zwischen den Simulationsteilnehmern definiert und kontrolliert. Durch Verwendung von CORBA als Standard-Middleware [OHE98] unterliegt ExITE keinen Einschränkungen bezüglich der Plattformauswahl. Durch die Anbindung von Hardware via CAN-Bus können die SW-Funktionen auf verschiedenen Steuergeräten betrieben und validiert werden.

Durch eine derartige Co-Simulation kann das Zusammenwirken mehrerer Subsysteme auf unterschiedlichen Anwendungen untersucht werden. ExITE unterstützt hierfür die Kopplung zwischen den oben genannten Modellierungswerkzeugen und erlaubt somit eine durchgängige Entwicklung. Die Kommunikation zwischen den Simulationsteilnehmern erfolgt über quasikontinuierliche Signale, die zu diskreten äquidistanten Zeitpunkten [KJ02] abgetastet werden. Da die UML keine Darstellung von kontinuierlichen Signalen kennt, wird die Simulation von einer übergeordneten Klasse gesteuert, die entsprechende Methoden mit einer festen Abtastrate zyklisch aufruft. Für weitere technische Einzelheiten wird auf [Schu04] verwiesen.

Verteilung der Fahrzeugfunktionen mit DaVinci

Mit der DaVinci Tool Suite™ 1.0 (im Weiteren DaVinci) von Vector-Informatik ist die Entwicklung eines logischen Steuergerätenetzwerks mit verteilten Funktionen möglich. Dazu sind in DaVinci die beiden kooperierenden Werkzeuge DaVinci DEV (Developer) und DaVinci SAR (System Architect) integriert.

DaVinci DEV ist eine Entwicklungs- und Integrationsumgebung für die Erstellung von Fahrzeugfunktionen für ein einzelnes Steuergerät. Für die Funktionserstellung werden zunächst leere Funktionsrümpfe mit Ein- und Ausgängen definiert. Dafür stehen Datenflussgrafen mit Blöcken zur Verfügung. Durch die Anbindung zur MATLAB Toolbox und dem Code-Generator TargetLink, kann der generierte C-Code aus den vorherigen Modellierungswerkzeugen in die jeweiligen Blöcke implementiert werden. Anschließend erfolgen

[34] Vgl. http://extessy.be-efficient.de

Maßnahmen zur Anbindung des Betriebs- und Kommunikationssystems.

DaVinci SAR erlaubt einen Gesamtentwurf von verteilten Systemen für mehrere Steuergeräte. Hiefür wird die HW-Topologie mit Steuergeräten, Sensoren, Aktuatoren und CAN-Bussen spezifiziert. Anschließend lassen sich die Funktionsblöcke aus DaVinci DEV auf die logischen Steuergeräte verteilen.

Mit dem *DaVinci Target-Package* kann C-Code für unterschiedliche Targets aus dem Gesamtmodell, einschließlich Netzwerkkommunikation über CAN generiert werden.

Werkzeuge zur Sicherstellung der SW-Qualität

Zur Sicherstellung der SW-Qualität werden die Modelle während des gesamten Entwicklungsprozesses auf Korrektheit, Inkonsistenz, Redundanz und Vollständigkeit durch verschiedene Werkzeuge überprüft. Die Überprüfung der Korrektheit der SW-Funktionen geschieht durch Simulation und Animation. Auf eine vollständige Überprüfung der Systemeigenschaften, beispielsweise durch Model Checking, wird in STEP-X bewusst verzichtet. Der Grund hierfür ist, dass der Einsatz formaler Analysewerkzeuge in der Praxis zurzeit nur schwer und kostenintensiv zu realisieren ist [JW96, HMF04]. Diese Erkenntnis konnte auch in [MD02] gewonnen werden, indem ein Model Checker in den STEP-X Entwicklungsprozess integriert wurde. Aufgrund der zum Teil fehlenden Schnittstellen sowie der unterschiedlichen Semantiken zwischen dem Model Checker und den Modellierungswerkzeugen, konnte nur eine prototypische Anbindung an die bereits vorgestellte Werkzeugkette realisiert werden.

Da durch die Simulation nur ein Teilaspekt überprüft werden kann, wird zusätzlich auf statische Analyseverfahren zurückgegriffen. Mit Hilfe der analytischen Maßnahmen kann das Verhaltensmodell vor der Ausführung geprüft werden. Dadurch sind Inkonsistenzen, Redundanzen und Modellierungsfehler frühzeitig lokalisierbar.

Zur Durchführung der Analyse wurden im Rahmen des STEP-X Projekts am Institut für Programmierung und Reaktive Systeme eigene Tools entwickelt. In [HMDF+04] werden diese näher beschrieben. Das wichtigste Werkzeug zur statischen Analyse ist der Regel Checker [MP03, Mutz03], mit dem außer einer Modellüberprüfung auch eine Modellbewertung durch SW-Metriken [Mutz04a, Mutz04b, Mutz05] möglich ist und somit eine quantitative Bewertung des SW-Systems gewährleistet wird. In Kapitel 5.2.1 auf Seite 122 befindet sich eine detaillierte Beschreibung des Regel Checkers.

Der Nachweis der vollständigen Umsetzung aller Anforderungen wird am Ende des Entwicklungsprozesses durch die AG Testen vorgenommen. Die Testdurchführung und Integration der Testwerkzeuge CTE XL und DMI [Nien04] in den Entwicklungsprozess wird unter anderem in [HHKM03, EHHM+03] vorgestellt.

Kapitel 4

4 Die STEP-X Entwurfsmethodik

„There is a big difference between Methodology and methodology[35].”

Tom DeMarco, 1987

Mit Hilfe des Tailorings wurde im letzten Kapitel ein für den STEP-X Entwicklungsprozess spezifisches Vorgehensmodell definiert. Die hierfür ausgewählten Elementarmethoden sind allerdings nur grob beschrieben und haben nicht den Anspruch einer durchgängigen methodischen Vorgehensweise. Beispielsweise werden für die Erstellung eines SW-Modells statische und dynamische Diagramme vorgeschlagen. Auf welche Weise diese zu erstellen sind, welche Detaillierungsstufen notwendig sind oder wie die Konsistenz zwischen diesen gewährleistet wird, bleibt allein dem Methodiker überlassen. In diesem Kapitel wird daher eine modellbasierte Entwurfsmethodik vorgestellt, die auf dem zuvor definierten Vorgehensmodell aufbaut.

Dabei wird vor allem auf die Konstruktionsphase des V-Modells eingegangen, die von der Anforderungserfassung bis zur automatischen Code-Generierung reicht. Den Schwerpunkt dieses Kapitels bildet das *Digitale Lastenheft*, das neben den textuellen Anforderungen auch grafische Modelle enthält, mit denen komplexere Sachverhalte präzisiert werden können. Zusätzlich ermöglicht ein Digitales Lastenheft eine frühzeitige Validierung des SW-Systems. Zur Verdeutlichung des modellbasierten Vorgehens wird das Komfortelektroniksystem eines VW Polos rekonstruiert. Auf die beiden Themen Testen und Diagnose, die sich auf dem rechten Ast des V-Modells befinden, soll in dieser Arbeit nicht eingegangen werden.

Wie bereits im vorherigen Kapitel erwähnt, unterscheiden sich die Phasenbezeichnungen in diesem Kapitel von den Phasenbezeichnungen im V-Modell, da die Kernaktivitäten SE 1 bis SE 3 in den beiden Phasen Anforderungsbeschreibung und Analyse zusammengefasst werden.

[35] *Methodology* ist der Versuch, ein generelles Verfahren zur Durchführung von komplexen Aufgaben zu etablieren. Alle wichtigen Entscheidungen sind bereits durch die Methode vorgegeben. Unter *methodology* dahingegen wird ein genereller Ansatz für eine geordnete Projektabwicklung verstanden. Dieser ist nirgendwo dokumentiert, sondern befindet sich in den Köpfen der projektbeteiligten Personen.

Kapitel 4.1 erläutert zunächst die Anforderungsbeschreibung. Daraufhin beschäftigt sich Kapitel 4.2 mit der Analyse der Systemarchitektur und den sich daraus ergebenden SW-Einheiten. Anschließend wird in Kapitel 4.3 der funktionale Grobentwurf betrachtet, in dem sowohl die Funktionsmodelle als auch die Umgebungsmodelle spezifiziert und simuliert werden. Abschließend erläutert das letzte Unterkapitel den Feinentwurf.

4.1 Anforderungsbeschreibung

Die Entwicklung umfangreicher SW-Systeme beginnt mit der Anforderungsbeschreibung, die sich hauptsächlich mit dem Auffinden, Erfassen und Überprüfen von Anforderungen befasst. Diese Tätigkeiten tragen zur Erstellung eines Lastenhefts bei. In der Automobilindustrie dienen diese Anforderungsdokumente vorwiegend als Kommunikationsgrundlage zwischen Herstellern und Zulieferern und sind in vielen Fällen vertragsbindend.

Die Qualität der Lastenhefte ist ausschlaggebend für die erfolgreiche Projektdurchführung, da fehlerhafte Anforderungen einen Summationseffekt nach sich ziehen und dadurch die Gefahr eines exponentiellen Fehleranstiegs zunimmt [Böhm81]. Die Standish Group [Stan95] evaluierte diesbezüglich zahlreiche SW-Projekte und kam zu dem Ergebnis, dass durch fehlerhafte Anforderungen 13% aller getesteten Projekte nicht erfolgreich abgeschlossen werden konnten.

Für einen durchgängigen Entwicklungsprozess ist es daher erforderlich, dass die Anforderungsdokumente vollständig, korrekt, verständlich und in sich konsistent sowie frei von Widersprüchen und Redundanzen sind.

Ziel im Rahmen des STEP-X Projektes war es, die Anforderungsbeschreibungen durch den Einsatz geeigneter Requirements Engineering-Methoden[36] zu verbessern. Als methodische Grundlage dienten die Ansätze von [Davi93, HJD02, Roma85, RR99, Rupp01], Zur Realisierung wird das in Kapitel 3.2.3 beschriebene Werkzeug DOORS eingesetzt. Als Informationsgrundlage dienten bestehende Lastenhefte eines Komfortelektroniksystems [BBHS96, Burn97, Burn01, Köhl01] mit den dazugehörigen Subsystemen *Zentralverriegelung*, *Elektrischer Fensterheber* und *Spiegelverstellung*.

Zusammenfassend wurden folgende Erkenntnisse erlangt. Eine erste Verbesserung der Anforderungsdokumente wird durch die Verwendung einer einheitlichen Struktur erzielt. Diese legt obligatorische Inhalte eines Moduls fest und erhöht die Lesbarkeit, Vollständigkeit und Konsistenz der Lastenhefte. Des Weiteren müssen die erfassten Anforderungen hinsichtlich ihrer Semantik genau analysiert werden, um unvollständige, widersprüchliche und missverständliche Spezifikationen aufzudecken. Zur Verbesserung der Lesbarkeit sind die textuellen Anforderungen anschließend mit Hilfe von Modellen zu visualisieren. Durch „intelligente“ Verlinkungsstrukturen können die Anforderungen abschließend hinsichtlich der Aufgabenstellung validiert werden. Diese im Kurzen vorgestellte Vorgehensweise wird im Folgenden näher erläutert.

[36] Auf den Austausch von Lastenheften zwischen Hersteller und Zulieferer wird nicht näher eingegangen. Die notwendigen Schritte für den Datenabgleich im STEP-X sind [HHM03] zu entnehmen.

4.1.1 Strukturierung von Anforderungsdokumenten

Unter Einbeziehung der in den Lastenheften bereits vorgefundenen Struktur und unter Ausnutzung der zusätzlichen Möglichkeiten, die DOORS bietet, wurde eine neue Strukturierung der Anforderungsdokumente vorgenommen [HHM03]. Diese ist speziell auf technische Systeme zugeschnitten, bei denen die Aspekte Echtzeitfähigkeit und Sicherheit für den Entwicklungsvorgang irrelevant sind.

Der erste anzuwendende Schritt zur Verbesserung der Lastenhefte ist die Aufteilung der gesamten Spezifikation in mehrere Dokumente, die den einzelnen Subsystemen entsprechen. In jedem Lastenheft besteht die Möglichkeit, die Anforderungen unterschiedlich zu detaillieren. So können beispielsweise die vom Benutzer direkt erfassten Wünsche bzgl. des Systems zunächst in die Benutzeranforderungen strukturiert abgelegt werden, ohne technische Details zu berücksichtigen. Nach einer gründlichen Analyse werden die Benutzeranforderungen präzisiert, durch externe Anforderungen ergänzt und als Systemanforderungen abgelegt. Dabei können die beiden Dokumente Benutzer- und Systemanforderungen als Produkte der Kernaktivitäten SE 1 bis SE 3 des V-Modells angesehen werden (vgl. Kapitel 3.2.1). Abbildung 4-1 veranschaulicht die Zusammenhänge der verschiedenen Anforderungsdokumente.

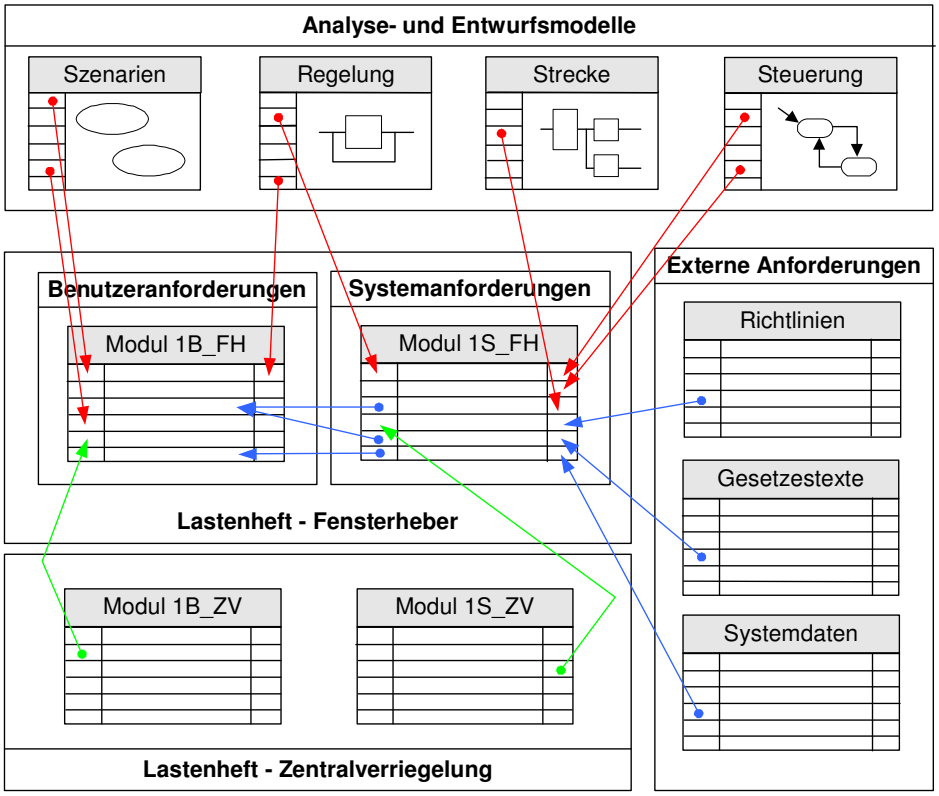


Abbildung 4-1: Das Verlinkungskonzept

Das Festlegen einer einheitlichen Struktur für alle Lastenhefte einer gemeinsamen Domäne führt unter anderem zur Erhöhung der Lesbarkeit und Wiederverwendbarkeit. In dem IEEE 830 Standard [IEEE91a] wird ein Vorschlag zur Errichtung einer Anforderungsstruktur unterbreitet, abhängig davon, ob Daten oder Funktionen im Vordergrund stehen.

Unter Zuhilfenahme dieser Strukturvorgabe, ergänzt durch [Rupp01], entstand eine einheitliche Lastenheftstruktur. In Abbildung 4-2 ist ein grobes Schema der Struktur dargestellt.

Zunächst erfolgt eine Auflistung aller beteiligten Bedienstellen (Schalter, Taster), Sensoren und Aktuatoren (Motoren und akustische oder optische Elemente). Diese werden dann näher unterteilt. Zum Beispiel erfolgt eine Einteilung der Sensorarten nach Türsensoren (einschließlich der Heckklappe), Fenstersensoren (z. B. Blockieren), Sensoren für allgemeine Fahrzeugfunktionen (Zündung, Geschwindigkeit usw.) und Sensoren für die Sicherheit (z. B. Wegfahrsperre).

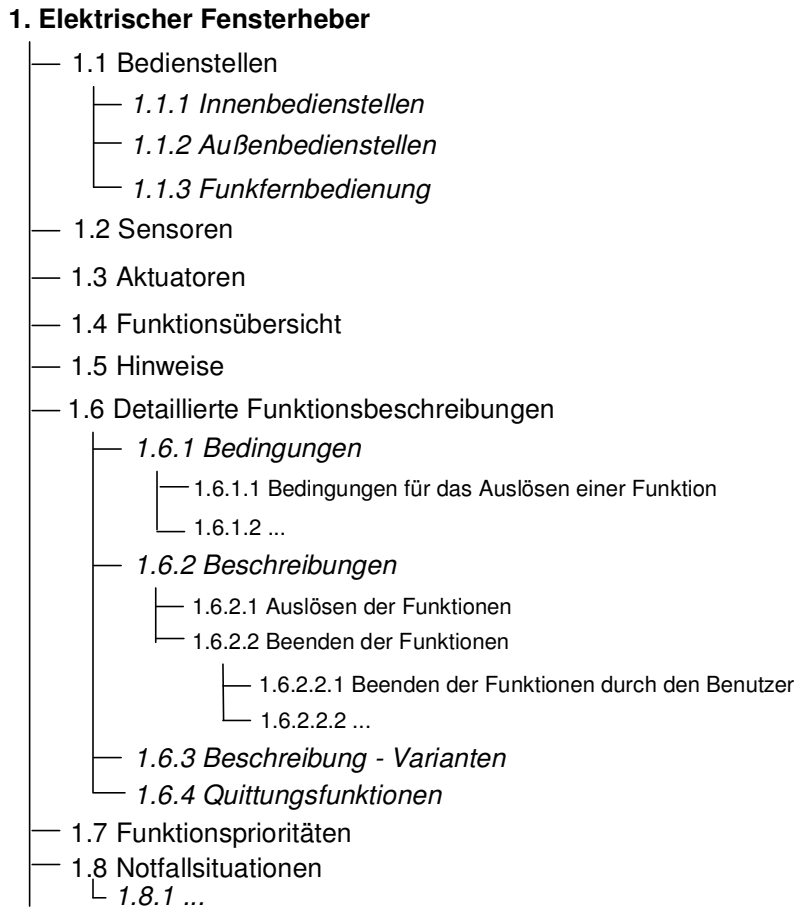


Abbildung 4-2: Struktur des Anforderungsdokuments Elektrischer Fensterheber

Die nächsten beiden Gliederungspunkte, Funktionsübersicht und Hinweise, bilden keinen Bestandteil der eigentlichen Funktionsbeschreibung, sondern dienen lediglich zur näheren Erläuterung. Anschließend erfolgt die detaillierte Beschreibung der einzelnen Funktionen, wobei diese wiederum unterteilt werden in Bedingungen, Beschreibungen, Varianten und Quittierungsfunktionen. In den Bedingungen sind alle Funktionsbedingungen aufgelistet, die grundsätzlich vor dem Ausführen der eigentlichen Funktion erfüllt sein müssen. Hierzu zählen beispielsweise der Status der Bordnetzfreigabe und die Positionierung der Türen. Unter dem Gliederungspunkt Beschreibungen wird das Auslösen der jeweiligen Subfunktion beschrieben. Handelt es sich bei der Funktion nicht um eine punktuelle Aktion (z. B.

Einschalten des Lichts) muss ebenfalls eine Erläuterung zur Beendigung der Funktion erfolgen. Im Normalfall wird das Beenden der Funktion entweder durch den Benutzer oder durch das System ausgelöst. Falls mehrere Varianten einer Funktion existieren, sind diese nacheinander mit verständlicher Namensbezeichnung und Erläuterung aufzulisten. Bei Funktionen, die z. B. durch einen Blinker oder ein Horn zu quittieren sind, sollte außer der eigentlichen Quittungsfunktion auch der zu quittierende Aktuator beschrieben werden. Ist die Bedienung eines Aktuators über mehrere Bedienstellen möglich, muss eine genaue Prioritätenreihenfolge z. B. durch Tabellen definiert werden. Um die Funktionalität eines Systems vollständig zu beschreiben, müssen auch Anforderungen für das Verhalten im Ausnahmefall aufgenommen werden. Daher sind abschließend mögliche Notfallsituationen (z. B. Zusammenstoß und Ausfall der Spannungsversorgung) zu beschreiben.

Nach der vertikalen Strukturierung des Lastenhefts durch Überschriften, erfolgt als nächstes eine horizontale Strukturierung durch das Anordnen von Attributen. Mit Hilfe der Attribute können übergreifende Informationen bzgl. der Anforderungen eingebracht werden. Hierfür stehen drei Arten von Attributen zur Verfügung:

- ***Funktionsspezifische Attribute:*** Bei komplexen Funktionen können funktionsspezifische Attribute die Zuordnung von Anforderungen zu bestimmten Varianten erleichtern. Ein Beispiel stellt der Ländercode dar, mit dem der Funktionsumfang eines Fahrzeugs ersichtlich wird. Je nach Land kann sich der Funktionsumfang beispielsweise durch Gesetzesvorgaben unterscheiden.
- ***Technische Attribute:*** Technische Attribute kennzeichnen einzelne Anforderungen oder Anforderungsgruppen mit zusätzlichen technischen Informationen. Dadurch können den Benutzern bestimmte Randbedingungen, wie Zugriffsrechte, Werkzeugnotwendigkeit und Bearbeitungsprioritäten mitgeteilt werden.
- ***Organisatorische Attribute:*** Obwohl in den meisten Requirements Engineering Tools die Möglichkeit besteht, das Änderungsmanagement automatisch vorzunehmen, erkennen diese nicht, ob die getätigten Änderungen relevant sind, oder ob nur ein Tippfehler korrigiert worden ist. Daher sind Änderungen vom Benutzer explizit vorzunehmen. Hierfür werden bestimmte Attribute, wie Benutzername, Änderung und Datum vorgegeben.

4.1.2 Ermittlung von Anforderungen

Nach der Fertigstellung der Dokumentenstruktur kann mit der Ermittlung von Anforderungen begonnen werden. Die hier vorgestellte Methode zur Anforderungsermittlung basiert im Wesentlichen auf einem natürlichsprachlichen Ansatz, der von den meisten Experten befürwortet wird [Lude93, HDK93, Jack95, SS97].

Die Ermittlung der Anforderungen in STEP-X orientiert sich an gängigen Requirements Engineering Methoden und ist in die Arbeitsschritte *Akteure definieren*, *Anwendungsfälle identifizieren* und *Anforderungen erfassen* gegliedert.

Akteure definieren

Im ersten Schritt werden Akteure (Anwender, externe Systeme) definiert, die das zu entwickelnde System bedienen sollen. Dabei ist insbesondere darzustellen, welche Ziele ein Akteur mit dem System erreichen will. In vielen Fällen macht es Sinn, mehrere Akteure mit gleicher Zielvorstellung zu gruppieren, um Überspezifikationen zu vermeiden und somit die Lesbarkeit zu erhöhen. Am Beispiel des Fensterhebers können beispielsweise die hinteren Insassen zu einem gemeinsamen Akteur zusammengefasst werden, da alle Personen im hinteren Fahrzeugraum die gleichen Funktionen auslösen können (z. B. manuelles runterfahren/hochfahren der jeweiligen Fenster).

Anwendungsfälle identifizieren

Bei der Identifizierung von Anwendungsfällen ist grundsätzlich zwischen funktionalen und nichtfunktionalen Anforderungen zu unterscheiden. Dabei beschreiben funktionale Anforderungen die eigentliche Funktionalität des Systems, während nichtfunktionale Anforderungen für die Qualitätsmerkmale und Randbedingungen der Software, wie z. B. Bedienbarkeit, Zuverlässigkeit und Wiederverwendbarkeit, verantwortlich sind.

Funktionale Anforderungen haben beim Erstellen eines Lastenhefts eine höhere Relevanz und sollten daher als erstes erfasst werden. Hierbei geht man von den identifizierten Akteuren aus. Für jedes erkannte Ziel eines Akteurs wird ein entsprechender Anwendungsfall abgeleitet, wobei zunächst die Hauptfunktionalität und anschließend die Variationen sowie mögliche Ausnahmefälle beschrieben werden.

Im Folgenden sind die wichtigsten Anwendungsfälle des elektronischen Fensterhebers aufgelistet, um die Verständlichkeit der begleitenden Fallstudie zu erhöhen:

- ***Türen öffnen/schließen:*** Über sogenannte Drehfallsensoren erkennt das System das Öffnen/Schließen einer Tür. Die Türposition hat wiederum Einfluss auf die Bordnetzfreigabe, die für die Aktivierung/Deaktivierung des Fensterlaufs verantwortlich ist.
- ***Zündung ein/aus:*** Durch das Betätigen der Zündung wird das Bordnetzsignal freigegeben, so dass Fensterläufe grundsätzlich möglich sind.
- ***Manueller Fensterlauf hoch/runter:*** Durch Betätigung eines Tasters an der Tür erfolgt ein manueller Fensterlauf. In Abhängigkeit von der Tasterstellung kann dies ein Öffnen oder Schließen des Fensters bewirken.
- ***Automatischer Fensterlauf hoch/runter:*** Für die vorderen Fenster ist ein Automatiklauf vorgesehen, welcher nur vom Fahrer ausgelöst werden kann.
- ***Komfortbedienung hoch/runter:*** Alle Fenster können gemeinsam über die Funkfernbedienung oder über die Schlüsselschalter geöffnet bzw. geschlossen werden.
- ***Kindersicherung ein/aus:*** Der Fahrer kann durch Betätigen der Kindersicherung die Bedienung der hinteren Fenster über die dortigen Taster deaktivieren/aktivieren.
- ***Schutzfunktionen ein/aus:*** Die Schutzfunktionen überwachen den Öffnungs- bzw.

Schließvorgang der Fenster, um beispielsweise Beschädigungen des Fensters zu vermeiden (Softstopp, Sanfteinlauf) oder Personen beim automatischen Schließen nicht zu gefährden (Einklemmschutz). Außerdem sollen die Fenstermotoren vor einem Schaden bewahrt bleiben. Bewegt sich zum Beispiel nach Ablauf einer definierten Zeit trotz Aktivierung das entsprechende Fenster nicht, so registriert das System eine Blockierung des Fensters und beendet die Bestromung (Blockiererkennung). Wird während eines Fensterlaufs der Motor überhitzt, so wird der Thermoschutz aktiviert und für eine bestimmte Zeit das Fenster nur noch zu öffnen sein.

Nichtfunktionale Anforderungen spielen bei der Erstellung von Lastenheften ebenfalls eine wichtige Rolle und müssen daher genauso ermittelt werden. Eine detaillierte Beschreibung vieler in der Praxis notwendiger nichtfunktionaler Anforderungen ist unter anderem in [IEEE91b] zu finden. Bezüglich der Bedeutsamkeit für das hier beschriebene System werden die drei Anforderungen Zuverlässigkeit, Sicherheit und Prüfbarkeit exemplarisch betrachtet.

Die Aspekte Zuverlässigkeit und Sicherheit müssen für die einzelnen Funktionen differenziert geprüft werden. Funktionen, wie z. B. der Einklemmschutz, erfordern einen hohen Grad an Zuverlässigkeit und Sicherheit. Für derartige Funktionen existieren gesetzliche Vorschriften, die mit den entsprechenden Funktionen verknüpft werden müssen. In Abbildung 4-1 wird dies durch das Modul Gesetzestexte verdeutlicht.

Den Anforderungen an die Prüfbarkeit des Systems kommt eine hohe Bedeutung zu. Dies gilt vor allem für zeitliche Anforderungen. So werden beispielsweise folgende Zeitbedingungen in das Anforderungsdokument eingeführt:

- ***Ansteuerungszeiten:*** Wie lange muss der Aktuator mit einem Signal angesteuert werden, damit die Funktion ausgelöst wird?
- ***Antwortzeiten:*** Wie viel Zeit darf maximal/minimal von der Funktionsauslösung durch eine Bedienstelle bis zum tatsächlichen Ausführen der Funktion vergehen?
- ***Ausführungszeiten:*** Wie viel Zeit darf die Funktionsausführung maximal/minimal benötigen?

Die Werte für die Zeitbedingungen sind parametrisierbar, d. h. für jede Zeitangabe wurde eine Variable eingeführt. Der Anfangswert sowie ein, für das Testen wichtiger, Toleranzbereich für jede Variable sind in dem Modul Systemdaten (vgl. Abbildung 4-1) aufgeführt.

Anforderungen erfassen

Die letzte Aktivität in der Anforderungsermittlung befasst sich mit der Erfassung der identifizierten Benutzeranforderungen. Die ermittelten Anforderungen werden in der entsprechenden Struktur von DOORS abgelegt und erhalten eine eindeutige Identifizierung.

Zur Erleichterung der Erfassung wurden die einzelnen Kapitel gesondert im Repository abgelegt, um somit eine höhere Modularisierung der Anforderungsdokumente zu erreichen. Diese Dezentralisierung hat den Vorteil, dass die Anforderungserfassung durch

mehrere Anwender getätigt werden kann. In dem Beispiel aus Abbildung 4-2 auf Seite 65 existieren für alle Kapitel der zweiten Ebene gesonderte Module in DOORS. Unter einem Modul wird ein in sich konsistentes und abgeschlossenes DOORS-Dokument verstanden.

4.1.3 Analysieren der Anforderungsdokumente

In diesem Arbeitsschritt werden die ermittelten Benutzeranforderungen durch den Systemanalytiker textuell überarbeitet. Dabei hat er die Aufgabe, aus den noch grob strukturierten Anforderungen eine vollständige Sammlung von qualitativ hochwertigen Anforderungen zu erstellen. Hierfür wird zunächst jeder Satz auf Vollständigkeit, Konsistenz, Verständlichkeit und Korrektheit überprüft.

Zur Durchführung eines solchen Analyseprozesses stellt die Literatur [GR00, Rupp01, Schr01] verschiedene Ansätze und Methoden bereit. Als besonders nützlich erwies sich in der Praxis die Kombination aus semantischer und linguistischer Analyse.

Mit Hilfe der semantischen Analyse lässt sich der Inhalt der Anforderungen auf Verständlichkeit, Konsistenz und Korrektheit überprüfen. Beispielsweise wird untersucht, ob Synonyme und doppelte Begriffsbelegungen in den Spezifikationen verwendet wurden, die unweigerlich zu Missverständnissen und Unklarheiten führen würden. Diese Defekte treten insbesondere dann auf, wenn mehrere Personen an der Erstellung der Lastenhefte beteiligt sind. Für eine reibungslose SW-Entwicklung ist es daher notwendig, ein Glossar mit einer projektspezifischen Terminologie anzulegen. Das STEP-X Glossar kann [GHMP02] entnommen werden. Des Weiteren ist die Umsetzung der komplementären Funktionen[37] zu kontrollieren.

Die linguistische Analyse ist ein weiteres Verfahren zur Überprüfung natürlichsprachlicher Anforderungsdokumente. Sie basiert auf dem linguistischen Ansatz von [Rupp00], die Methoden aus der Linguistik, Informatik und Psychologie vereinigt. Die in einem Anforderungsdokument enthaltenen Sätze werden bestimmten linguistischen Auswertungen unterzogen. Typische Mängel, nach denen gesucht wird, sind Tilgungen, Generalisierungen und Verzerrungen, die in [Melc00] beschrieben und in STEP-X aufgegriffen wurden:

- ***Tilgung:*** Unter Tilgung wird das Weglassen von Informationen verstanden. Eine Möglichkeit zur Eingrenzung des Informationsverlustes ist das Formulieren der Anforderungen im Aktiv. Dies hat den Vorteil, dass der Akteur in die Anforderungsbeschreibung mit einbezogen wird. Zusätzlich ist jede Anforderung durch ein Vollverb auszudrücken, damit Adjektive die Wichtigkeit der geforderten Funktionalität nicht verringern.
- ***Generalisierung:*** Die Verallgemeinerung von funktionalen Informationen wird als Generalisierung bezeichnet. Dabei tragen Universalquantoren, wie „*nie*, *immer*, *kein*, *jeder*, *alle*, *irgendeiner*“ eine wesentliche Rolle zur Generalisierung bei. Durch das Analysieren jeder Anforderung, die über einen Universalquantoren verfügt, soll

[37] Unter einer komplementären Funktion wird eine Funktion verstanden, die die Auswirkungen einer anderen Funktion rückgängig macht (z. B. einschalten/ausschalten).

bestimmt werden, ob das geforderte Verhalten tatsächlich für alle Objekte aus der Menge gelten soll. Zusätzlich sollen die in einer Anforderung vorkommenden Substantive stets im Singular verwendet werden. Eine Formulierung im Plural ist nur dann zulässig, wenn der geforderte Sachverhalt sich auf eine Gruppe in ihrer Gesamtheit bezieht.

- ***Verzerrung:*** Wenn zeitlich zusammenhängende Informationen in einer Anforderung verallgemeinert werden, spricht man von einer Verzerrung. Eine Form der Verzerrung ist das Funktionsverbgefüge, das meist Kombinationen aus Modalverben, wie „*machen*, *können*, *haben*, *sein*“ und sinngebenden Substantiven bildet. Solche Funktionsverbgefüge sind nach Möglichkeit durch einfache und direkte Vollverben zu ersetzen.

Nach der Durchführung der beiden Analysemethoden erhält der Anwender ein verständliches, korrektes und vollständiges Anforderungsdokument aus der Sicht des Benutzers. Dieses Lastenheft hat jedoch den Nachteil, dass durch die stark modularisierte und textuelle Struktur ein Zusammenhang zwischen den unterschiedlichen Anforderungen fehlt. Dieses Problem wird im folgenden Kapitel durch eine Verlinkungsstruktur gelöst.

4.1.4 Visualisierung der Benutzeranforderungen

Zur Erreichung eines hochwertigen Lastenheftes reicht eine strukturelle Überarbeitung der textuellen Benutzeranforderungen allein nicht aus. Zusätzlich müssen schwerverständliche Anwendungsfälle durch grafische Notationen ergänzt werden, um einen schnelleren Einblick in das zu entwickelnde System zu erhalten. Die Lesbarkeit und das Verständnis der Anforderungen sind die Voraussetzung für einen effektiven Entwicklungsprozess.

Als Beschreibungssprache dienen Anwendungsfalldiagramme, mit denen die identifizierten Anwendungsfälle zunächst übersichtlich dargestellt werden. Zusätzlich erfolgt eine Verknüpfung der Anwendungsfälle mit den dazugehörigen Akteuren. Das Ein- und Ausgabeverhalten des reaktiven Systems wird durch Szenarien visualisiert, die den zeitlichen Ablauf eines Anwendungsfalls näher beschreiben. Jedem Anwendungsfall wird mindestens ein Szenario für die Hauptfunktion sowie für Varianten und Ausnahmesituationen zugewiesen. Die Vorgehensweise zur Erstellung von sinnvollen Anwendungsfällen und Szenarien basiert auf praxisrelevanten Erfahrungen von [JH02]. Sie wird nachfolgend erläutert.

Erstellung der Funktionsübersicht

Für die Erstellung einer grafischen Übersicht der zu entwickelnden Systemfunktionen eignen sich vor allem Anwendungsfalldiagramme. Diese fassen alle in Kapitel 4.1.2 ermittelten Akteure und Anwendungsfälle zusammen und bilden diese in einer übersichtlichen Form ab. Ausgangspunkt bei der Anwendungsfallmodellierung ist zunächst ein grobes Anwendungsfalldiagramm, das iterativ in mehreren Zyklen verfeinert wird.

Bei der hier angewendeten Anwendungsfallmodellierung wird zwischen internen und ex-

ternen Akteuren unterschieden. In der STEP-X Methodik stellen interne Akteure die Sensorik und die externen Akteure die Bediener bzw. Aktuatoren des Systems dar. Im Gegensatz zu [KM00] werden Aktuatoren ausschließlich als externe Akteure angesehen. Die daraus resultierenden Vorteile, wie z. B. getrennte Betrachtung von Steuerung und restlichen Systemkomponenten sowie ein höheres Abstraktionsniveau der Hardware, vereinfachen den Modellierungsvorgang. Der gleiche Ansatz wird auch von [Oest99] verfolgt. Als Konvention wurde vereinbart, die Benutzer auf die linke und die Aktuatoren auf die rechte Seite der Systemgrenzen zu platzieren.

Abbildung 4-3 zeigt das erste Anwendungsfalldiagramm, in dem die sechs Anwendungsfälle dargestellt sind, die den Kern des Fensterhebers aus Sicht der Benutzer repräsentieren. Alle Anwendungsfälle lassen sich von den Benutzern initiieren. Dabei wird grundsätzlich zwischen verschiedenen Anwendern des Systems unterschieden, die alle über einen benutzerspezifischen Funktionsumfang verfügen. So verfügt beispielsweise der Fahrer durch die Vererbungsbeziehung über dieselben Rechte wie der Beifahrer, kann aber weitere Funktionen wie z. B. den automatischen Fensterlauf auslösen.

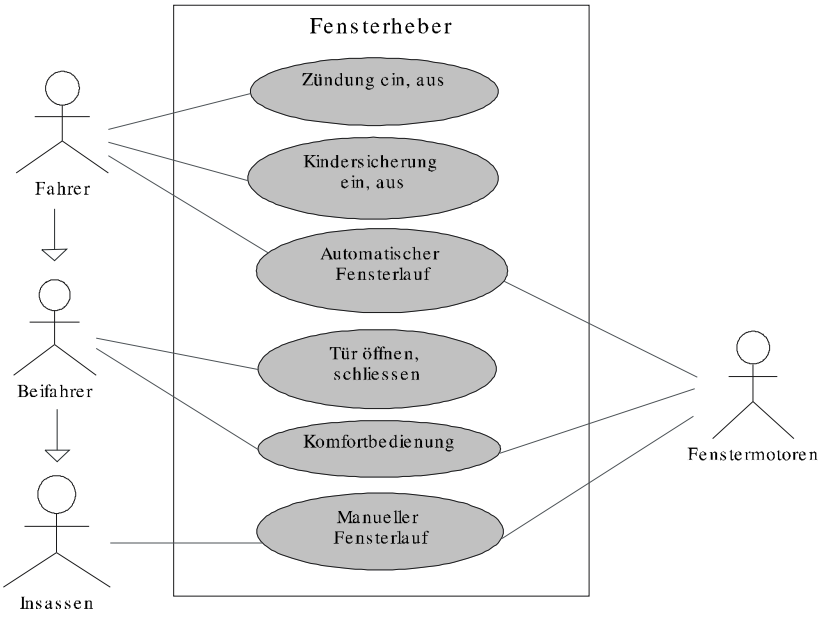


Abbildung 4-3: Kernfunktionen des Fensterhebers

Zur Darstellung der vollständigen Fensterheberfunktionalität müssen neben den eigentlichen Benutzerfunktionen auch systeminterne Anwendungsfälle mitbetrachtet werden. Als Beispiel dient der Anwendungsfall *Schutzfunktionen*, der in Abbildung 4-4 dargestellt ist. Aus Gründen der Übersichtlichkeit wurde in dieser Abbildung auf Akteure und nicht näher zu betrachtende Anwendungsfälle verzichtet.

Dabei verdeutlicht der Stereotyp *<<include>>* die Kommunikationsbeziehungen zwischen den Fensterläufen und Schutzfunktionen. Wird beispielsweise ein Fensterlauf ausgelöst, so sind auch die Schutzfunktionen auszuführen. Um den Begriff Schutzfunktionen in Anlehnung an Kapitel 4.1.2 näher zu spezifizieren, wird dieser mit Hilfe der Vererbungsbeziehung in weitere Anwendungsfälle gegliedert. Diese Verfeinerung ist grundsätzlich für alle komplexeren Anwendungsfälle durchzuführen.

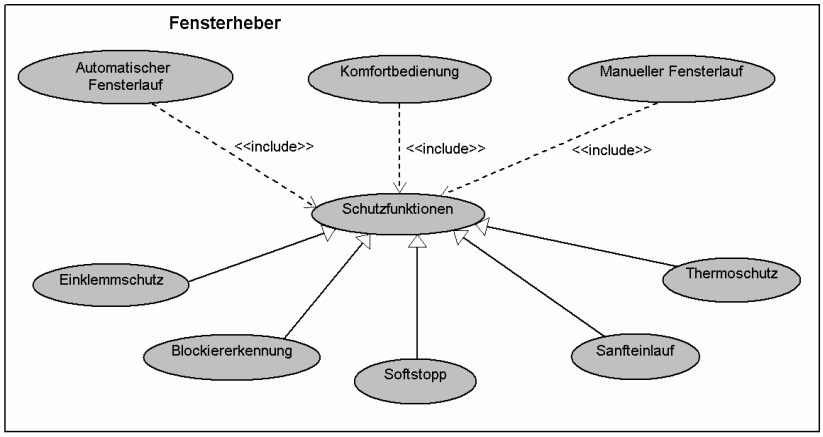


Abbildung 4-4: Anwendungsfalldiagramm des Fensterhebers mit Schutzfunktionen

Bei einer zu detaillierten Anwendungsfallspezifikation kann es jedoch zu Unübersichtlichkeiten kommen. In solchen Fällen, bei denen nicht auf das höhere Detailniveau verzichtet werden kann, bleibt dem Anwender nur die Möglichkeit, auf den von der UML vorgegebenen Abstraktionsmechanismus zurückzugreifen. Im Sinne einer Bottom-Up-Vorgehensweise werden Anwendungsfälle, die ein gemeinsames Ziel verfolgen, jeweils zu einem Paket gruppiert, das sich nach außen wie ein eigenes Anwendungsfalldiagramm verhält.

Abschließend ist jeder Anwendungsfall bzw. jedes Paket im UML-Werkzeug mit einer zusätzlichen Kurzbeschreibung zu versehen. Als Beschreibungshilfe können die von [HK99, KM00] vorgegebenen Beschreibungsschemata verwendet werden.

Definition von Szenarien

Durch die Anwendungsfallmodellierung wird zunächst ein Überblick über das zu entwickelnde System gegeben. Informationen über zeitliche Abläufe, Prioritäten oder den Kommunikationsaustausch zwischen Benutzern, System und Aktuatoren sind aber weiterhin unspezifiziert. Daher werden in den darauffolgenden Schritten die abstrakten Anwendungsfälle näher spezifiziert, wobei der zeitliche Ablauf der Funktionen im Vordergrund der Modellierung steht. Zur Abbildung dieser Szenarien werden Sequenzdiagramme verwendet.

Jedes Szenario beginnt mit der Aktion des Akteurs und endet mit einer nach außen hin sichtbaren Reaktion des Systems. Dabei wird spezifiziert, in welcher Reihenfolge die beteiligten Objekte (Akteure, Anwendungsfälle, Aktuatoren) miteinander kommunizieren müssen, um die gewünschte Funktion erfolgreich durchzuführen. Die Benennung und Typisierung der später tatsächlich zu realisierenden Objekte und Signale ist an dieser Stelle zunächst irrelevant, sollte allerdings allgemeinverständlich sein. Die Beschriftung und Typenzuweisung erfolgt in der nachfolgenden Analysephase.

Abbildung 4-5 zeigt beispielhaft ein Sequenzdiagramm für das *automatische öffnen* eines Fensters durch den Fahrer. Der *Automatiklauf* wird ausgelöst, indem der Fahrer den Taster für die Fensterhebersteuerung über einen Widerstand (Doppeldrucktaster) hinaus betätigt. Der Taster leitet das Signal an die Funktion *Automatiklauf* weiter, die dann dem

Fenstermotor den Befehl zum Öffnen des Fensters erteilt. Lässt der Fahrer denselben Taster wieder los, so wird der Fensterlauf ohne Unterbrechung fortgesetzt. Ein erneutes Betätigen des Tasters in eine beliebige Richtung bewirkt ein Stopp des dazugehörigen Fenstermotors. Da der Automatiklauf für alle Fenster dasselbe Verhalten aufweist, bedarf es keiner näheren Erläuterung jedes einzelnen anzusteuernden Fenstermotors.

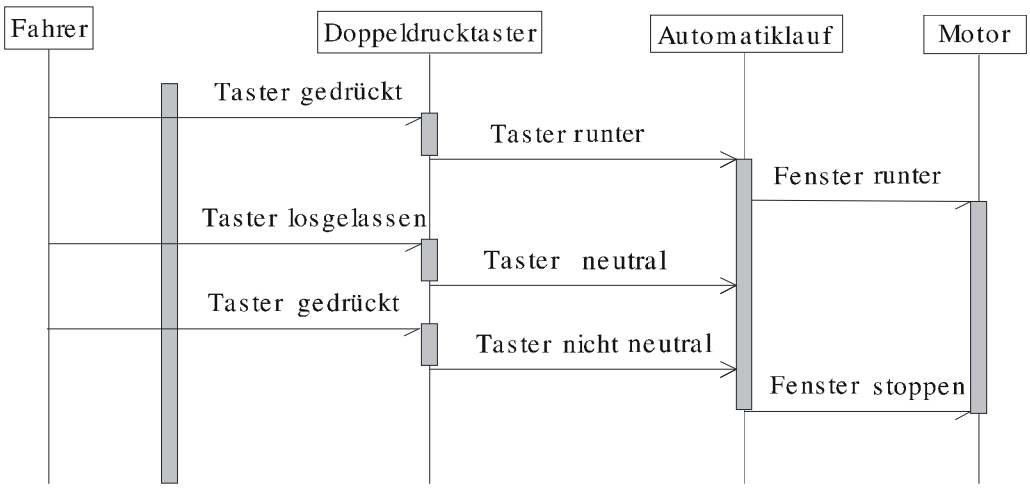


Abbildung 4-5: Sequenzdiagramm für den Automatiklauf

Wurden alle ermittelten Anwendungsfälle durch Szenarien näher erläutert, so müssen als nächstes Notfallsituationen und Varianten in gleicher Weise spezifiziert werden. Abbildung 4-6 veranschaulicht dazu am Beispiel des Einklemmschutzes das Vorgehen: Zunächst löst der Fahrer das automatische Schließen eines Fensters aus. Dabei entspricht der Schließvorgang einem ähnlichen Ablauf wie in Abbildung 4-5. Wird das Fenster während des Schließens von einem Insassen blockiert, so sendet ein Fenstersensor ein Blockiersignal zum Automatiklauf, der anschließend den Fenstermotor zum Reversieren auffordert.

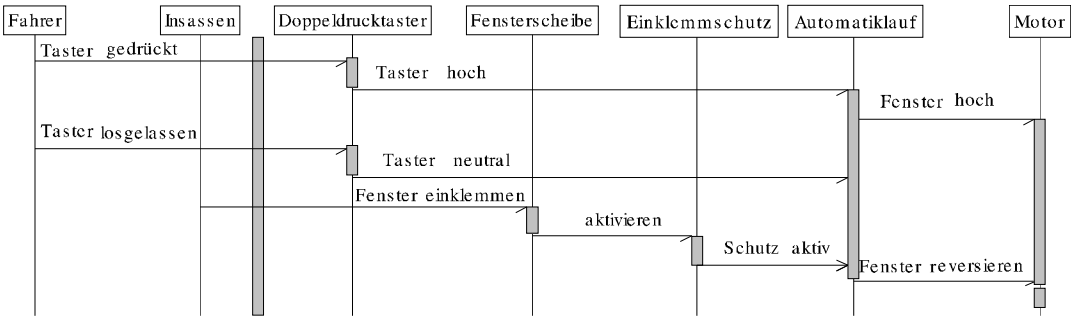


Abbildung 4-6: Sequenzdiagramm für den Automatiklauf mit Einklemmschutz

4.1.5 Verlinkung von Anforderungen

Lastenhefte in der Fahrzeugentwicklung umfassen gegenwärtig mehrere hundert Seiten. Um einen besseren Überblick zu gewährleisten, werden neben der bereits vorgestellten Strukturierung und Modularisierung auch Zusammenhänge zwischen den einzelnen Anforderungen durch Verlinkung hergestellt. Dabei ist eine Verlinkung sowohl zwischen Anforderungen desselben Moduls als auch zwischen verschiedenen Modulen möglich. Es

besteht die Konvention, dass Links (Pfeilrichtung) von den abgeleiteten Anforderungen auf ursprüngliche Anforderungen verweisen müssen. Dabei kann entlang der verlinkten Anforderungen in jede Richtung navigiert werden.

Mit Hilfe einer durchgängigen Link-Struktur kann außerdem eine automatische Analyse aller Anforderungsdokumente eines Projekts erfolgen. Hierbei sind zwei grundsätzliche Analysearten möglich. Durch die *Auswirkungsanalyse* (engl.: impact) werden alle Anforderungen, die von einer ausgewählten Anforderung direkt oder indirekt abhängen, untersucht. Diese Analyseart eignet sich insbesondere, um die Kosten von Änderungsvorschlägen zu bewerten oder nicht abgedeckte Anforderungen zu identifizieren, d. h. Anforderungen, für die in nachfolgenden Modulen keine zugeordneten Anforderungen vorhanden sind. Bei der Rückverfolgung (engl.: traceability) werden dagegen Link-Ketten in Richtung der ausgehenden Links verfolgt, um den Ursprung abgeleiteter Anforderungen zu erhalten. Wird beispielsweise im späteren Grobentwurf eine Änderung vorgenommen, so kann durch die Rückverfolgung automatisch ermittelt werden, welche übergeordneten Anforderungen zu berücksichtigen sind. Rückverfolgung eignet sich auch, um unnötige Anforderungen zu identifizieren, die auf einer bestimmten Ebene hinzugefügt wurden, ohne dass sie sich aus einer Anforderung oder technischen Randbedingung eines übergeordneten Dokuments ergeben.

Im STEP-X Entwicklungsprozess wurde eine derartige Verlinkungsstruktur erarbeitet, die in Abbildung 4-1 auf Seite 64 schematisch dargestellt ist. Als Erweiterung der oben beschriebenen Verlinkungsstruktur, ist hinzuzufügen, dass textuelle Anforderungen auch mit grafischen Modellen verlinkt werden. Wesentliche Vorteile der Verlinkung zwischen textuellen Anforderungen und Modellen sind zum einen die verbesserte Übersicht über die gesamten (Sub-)Funktionen und zum anderen die Detaillierung der textuellen Beschreibung [BR00, Heim01]. Die Granularität der Verlinkungsstruktur orientiert sich allerdings eher an pragmatischen Gegebenheiten, wie Effektivität, Übersichtlichkeit und Aufwand, so dass nur schwerverständliche Anforderungen über eine Modellverlinkung verfügen. Ein derartiges Vorgehen ist auch in der Praxis zu erkennen, da eine detaillierte Verlinkung einen hohen Aufwand bedeutet, der Nutzen aber nicht im gleichen Maße ansteigt.

4.2 Analyse

Mit der im vorherigen Kapitel vorgestellten methodischen Vorgehensweise sind die SW-Entwickler in der Lage, vollständige und konsistente Lastenhefte für eingebettete SW-Systeme zu erstellen. Diese strukturierten Anforderungsdokumente bilden den Ausgangspunkt für die hier vorgestellte Analysephase. Ziel dieser Phase ist das Konstruieren eines Analysemodells, mit dem alle benötigten Systemfunktionen und Schnittstellen identifiziert werden können. Des Weiteren bildet das Konzept des Analysemodells die Grundlage für die spätere Entwurfsphase. Daher wird bereits in der Analyse auf objektorientierte Elementarmethoden zurückgegriffen (vgl. Tabelle 3-2), wobei Klassen, Pakete und Komponenten die wesentlichen Modellierungselemente darstellen.

Gemeinsam mit dem Analysemodell bilden die textuellen Spezifikationen die Systemanforderungen, mit denen eine statische Sicht des Gesamtsystems dargestellt wird. Eine Verlinkungsstruktur erhöht das Verständnis der Abhängigkeiten zwischen den Benutzer- und Systemanforderungen (vgl. Abbildung 4-1 auf Seite 64).

Die Maßnahmen zur Erstellung des Analysemodells werden im Folgenden anhand des Subsystems Fensterheber näher erläutert. Dabei sind die anschließenden Abschnitte als inkrementelle zyklische Modellierungsprozesse zu verstehen, die im V-Modell den Kernaktivitäten SE 2 und SE 3 entsprechen.

4.2.1 Analyse der logischen Systemarchitektur

Das V-Modell fordert mit der Aktivität SE 2 die SW-Architektur des Systems in übersichtliche Struktureinheiten zu zerlegen. Dabei ist auf die in Kapitel 3.1.2 erläuterte Erzeugnisstruktur zurückzugreifen, in der die SW-Einheiten die gröbste Granularität eines SW-Systems aufweisen. Nach der anschließenden Identifikation aller ersichtlichen Schnittstellen zwischen System und Umgebung ist diese Kernaktivität beendet.

In der STEP-X Entwurfsmethodik wird dies wie folgt umgesetzt. Zunächst erfolgt eine Zerlegung des Fensterhebersystems in vier SW-Einheiten. Die Aufteilung des Systems ist in Abbildung 4-7 in Form von UML-Paketen dargestellt, wobei das Paket *Fensterheber* das eigentliche Steuerungsmodell darstellt. Durch den Stereotype <<subsystem>> wird darauf hingewiesen, dass diese SW-Einheit als ein eigenständig funktionierendes Subsystem fungiert. Die zweite zu entwickelnde SW-Einheit bildet die *Strecke*, die über die notwendige Aktuatorik verfügt und somit von der Fensterhebersteuerung getriggert wird. Für die Bedienung der Fenstersteuerung und eine möglichst realitätsnahe Simulation des Gesamtsystems ist eine spezielle Bedienoberfläche notwendig. Zu diesem Zweck werden Bedienelemente sowie Sensor- und Aktuatorwerte im *GUI*-Paket zusammengefasst. In vielen Fällen hat das Ausführen einer Funktion Auswirkungen auf weitere Fahrzeugfunktionen. Beispielsweise müssen Daten ausgetauscht, Funktionen gestartet oder Systemläufe unterbrochen werden. Aus diesem Grund werden alle Funktionen, die das System beeinflussen bzw. vom System beeinflusst werden in die SW-Einheit *Fremdsysteme* aufgenommen.

Nach der Darstellung der SW-Einheiten ist im nächsten Schritt eine grobe Analyse der Paketabhängigkeiten notwendig. Die Kommunikation zwischen den SW-Einheiten verläuft grundsätzlich über Schnittstellen [Rau01a].

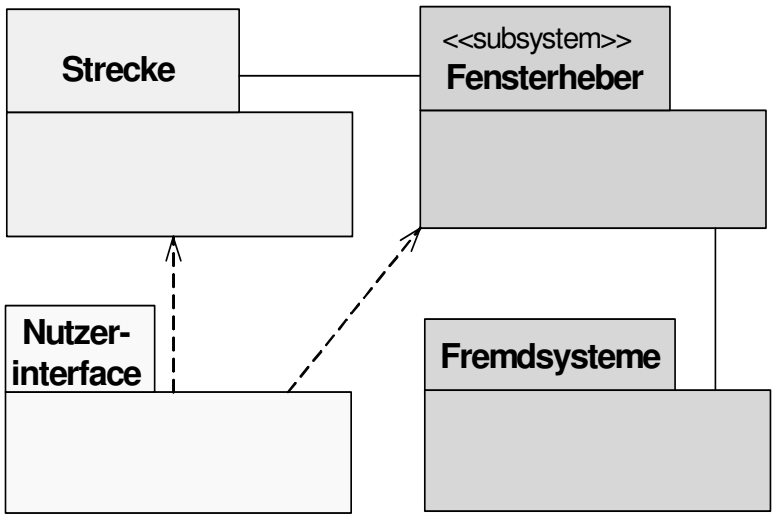


Abbildung 4-7: Ausschnitt der SW-Architektur

Durch die Einführung von Schnittstellenklassen wird der strukturelle Aufbau des Systems verdeckt und somit das Geheimnisprinzip bewahrt.

Geheimnisprinzip

Kriterium zur Gliederung eines Gebildes in SW-Einheiten/SW-Komponenten, so dass jede SW-Einheit/SW-Komponente eine Leistung (oder eine Gruppe logisch eng zusammenhängender Leistungen) vollständig erbringt, und zwar so, dass außerhalb der SW-Einheit/SW-Komponente nur bekannt ist, was diese leistet. Wie sie ihre Leistung erbringt, wird nach außen verborgen [Glin04].

Als Darstellungsmittel dient zunächst die UML Lollipop-Notation, mit der die Attribute und Methoden ausgeblendet werden. Es wird gefordert, die Lollipop-Schnittstelle mit einem ausdrucksstarken Namen zu versehen.

Die in Abbildung 4-7 dargestellten SW-Einheiten werden im nächsten Schritt mit Hilfe der Kernaktivität SE 3 des V-Modells genauer spezifiziert, indem die SW-Einheiten wiederum in kleinere Strukturelemente zerlegt werden. Der nachfolgende Abschnitt erläutert das dabei anzuwendende Vorgehen.

4.2.2 Spezifizierung der SW-Einheiten

Eine nähere Spezifizierung der SW-Einheiten erfolgt durch Bildung geeigneter SW-Komponenten. Aus den daraus resultierenden Strukturen werden Aufgaben zwischen den SW-Entwicklern verteilt sowie eine Zuweisung der später zu verwendenden Beschreibungssprachen und Werkzeuge getroffen. Des Weiteren lassen sich frühzeitig die ersten Zusammenhänge zwischen Subsystemen und Funktionen identifizieren. Eine derartige Gliederung durch Pakete und Komponenten hat sich bereits in zahlreichen UML-Projekten [BRS00, DKKM+02, HK99, Hörf02] durchgesetzt. Lediglich wenige UML-Autoren haben

Einwände gegen die frühzeitige Einteilung in Subsysteme bereits in der Analysephase:

> *„ ... Da aber zu diesem Zeitpunkt nicht deutlich entscheidbar ist, welche Objekte nun Subsysteme werden und als Paket auftauchen, raten wir dazu einfach sofort mit Klassendiagrammen zu beginnen...“* [SW00].

Diese Befürchtung, identifizierte Objekte nicht richtig einordnen zu können, ist an dieser Stelle nicht nachvollziehbar. Die frühzeitige Zerlegung des Systems in kleinere funktionale Einheiten gibt einen ersten Überblick über das zu entwickelnde SW-System und vereinfacht die damit verbundenen organisatorischen Maßnahmen [EFS01, JBAG97, SP00].

Durch Verringerung der Kopplung zwischen den SW-Komponenten wird eine strikte Modularisierung und somit eine erhöhte Flexibilität des Systems angestrebt, so dass Pakete und Komponenten ohne großen Aufwand wieder entfernt werden können.

Dabei wird unter Kopplung folgendes verstanden:

Kopplung

Die Kopplung ist ein qualitatives Maß für die Abhängigkeit zwischen zwei SW-Komponenten. Die Modularisierung ist umso besser, je geringer die wechselseitige Kopplung zwischen den SW-Komponenten ist [Glin04]. Dabei spielen der Kopplungsmechanismus, die Schnittstellenbreite und die Kommunikationsart eine wesentliche Rolle.

Je nach Größe und Komplexität des zu entwickelnden Systems können SW-Einheiten über weitere Komponenten und detailliertere Abhängigkeitsbeziehungen verfügen. In Abbildung 4-8 ist ein Beispiel für eine mögliche Strukturierung der SW-Einheiten dargestellt.

Das Paket *Fensterheber* enthält zwei SW-Komponenten. In der STEP-X Methode wird davon ausgegangen, dass ein Subsystem keine weiteren SW-Komponenten enthalten darf, sondern nur aus SW-Modulen (Klassen) besteht. So enthalten beispielsweise die beiden SW-Komponenten *Fensterläufe* und *Schutzfunktionen* lediglich Funktionsansammlungen in Form von Klassendiagrammen.

Die Bildung von SW-Komponenten für die Fensterhebersteuerung richtet sich zunächst nach den Anwendungsfällen aus der Anforderungsbeschreibung (vgl. Abbildung 4-4 auf Seite 72). Hierbei muss sorgfältig abgewogen werden, ob mehrere Anwendungsfälle ein Subsystem bilden können (z. B. Automatiklauf und manueller Lauf) oder ob sie lediglich zur Beeinflussung einer Funktion oder eines Subsystems beitragen (z. B. Türen öffnen/schließen oder Zündung ein-/ausschalten). Alle weiteren SW-Einheiten der Systemarchitektur werden auf die gleiche Weise zerlegt, wobei als Vorgabe wiederum die Anwendungsfalldiagramme dienen.

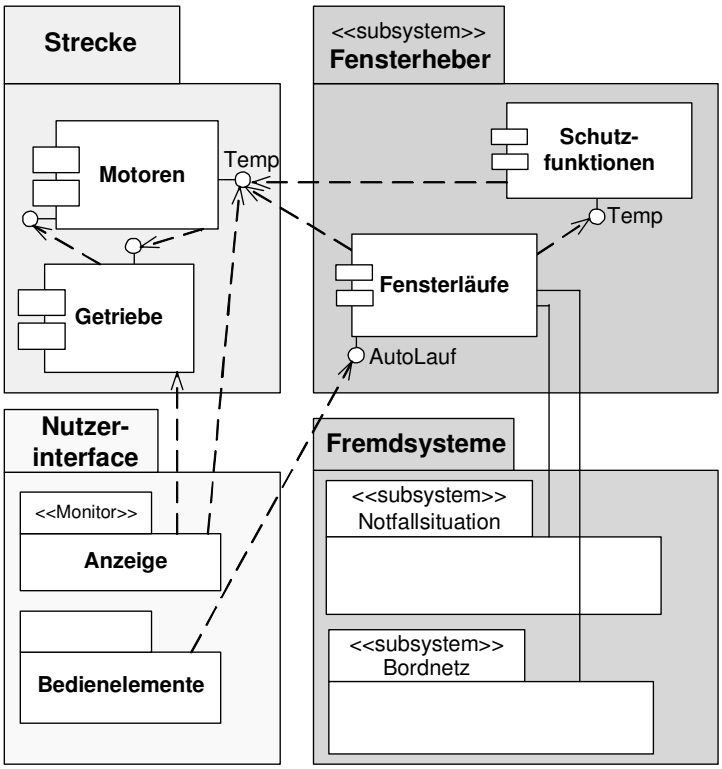


Abbildung 4-8: SW-Einheiten des Fensterhebers

In einigen Fällen müssen SW-Einheiten zur näheren Spezifikation mit Informationen erweitert werden, die sich nicht in den Lastenheften befinden. Beispielsweise verfügt das Streckenmodell über eine *Mechanik*-Komponente, die in einem Lastenheft normalerweise nicht beschrieben ist. Bei der Mechanik handelt es sich um eine Ansammlung von (zusammenhängenden) Systemteilen, die vom Motor angesteuert werden. Bei einem Fensterhebersystem sind dies beispielsweise die zu bewegende Fensterscheibe und die Führung. Letztere beeinflussen den Fensterlauf, je nach Beschaffenheit der Oberfläche und des Materials und wirken somit indirekt auf die Systemfunktionalität (z. B. Blockiererkennung). Je nach dem, wie hoch die Simulationsgenauigkeit sein soll, kann die Mechanik-Komponente entfallen. Für einen realistischen Simulationslauf einer sicherheitsrelevanten Funktion wie dem *Einklemmschutz*, ist die Modellierung eines detaillierten Umgebungsmodells erforderlich.

Ein weiteres Beispiel stellt das Paket *Anzeige* dar, das lediglich als Hilfspaket für die Simulation dient und durch den Stereotyp <<Monitor>> gekennzeichnet ist. Unter Monitorfunktionen ist eine Ansammlung aller für die Simulationsläufe notwendigen Signale zu verstehen. Diese sind explizit für den Testbetrieb geeignet und haben im späteren Serienfahrzeug keine Verwendung. Als Beispiele seien hier die Funktionen *Sanfteinlauf* und *Softstopp* genannt, die dem Tester den Status des Fensterlaufs mitteilen.

Die SW-Einheit Fremdsysteme beinhaltet die beiden Pakete *Notfallsituation* und *Bordnetz*, die genauso wie der Fensterheber als eigenständige Subsysteme agieren. Ziel dieser beiden Pakete ist nicht die Modellierung der jeweiligen Subsysteme, sondern lediglich die Beachtung möglicher Zugriffe von und nach außen. Die hierfür notwendigen Kommunika-

tionspfade zwischen Fremdsystemen und dem zu modellierenden System sind der Anforderungsbeschreibung zu entnehmen.

Je nach Größe der Systemkomplexität und des Funktionsumfangs kann eine Erweiterung der Struktur erfolgen, in der weitere SW-Komponenten und Schnittstellen definiert werden. Nach der Spezifikation aller benötigten SW-Komponenten werden diese im nächsten Schritt wiederum näher beschrieben. Obwohl die hier dargestellten SW-Einheiten im späteren Entwurf teilweise durch unterschiedliche Werkzeuge und Notationen realisiert werden, wird in dieser Phase die Erstellung eines einheitlichen Analysemodells mittels einer einzigen Beschreibungssprache angestrebt. Dadurch wird einerseits das Verständnis des Gesamtsystems erleichtert. Andererseits vereinfacht es die Verlinkung zwischen dem Analysemodell und den textuellen Anforderungen. Der folgende Abschnitt demonstriert die Verfeinerung der SW-Einheiten am Beispiel des Fensterhebers.

4.2.3 Erstellung der logischen SW-Struktur

Die logische SW-Struktur eines (Sub-)Systems dient als Grundlage für den späteren Grobentwurf. Für die Erstellung der logischen SW-Struktur werden zunächst die für den Entwurf benötigten Objekte identifiziert und in einem Klassendiagramm zusammengefasst. Dabei ist für einen erfolgreichen Analyseprozess nicht der Detaillierungsgrad ausschlaggebend, sondern eine abstraktere Darstellungsform der kooperierenden Objekte. Im Einzelnen bedeutet dies, dass bei der Erstellung von Klassendiagrammen den Attributen, Methoden und Sichtbarkeiten keinerlei Bedeutung zukommt. Vorrangig ist dagegen die Bildung zusammengehöriger Objekte, die durch Schnittstellen untereinander und nach außen kommunizieren.

Die Erzeugnisstruktur des V-Modells schreibt nicht vor, wie viele Klassen in einer SW-Komponente zu integrieren sind. Sinnvollerweise sollte eine SW-Komponente jedoch über möglichst wenig Module verfügen, um eine hohe Kohäsion zu erreichen, d. h. die Elemente jeder SW-Komponente sollten in einem engen Zusammenhang zueinander stehen. Unter Kohäsion wird folgendes verstanden:

Kohäsion

Kohäsion ist ein qualitatives Maß des inneren Zusammenhangs einer SW-Komponente. Je höher die Kohäsion ist, desto stärker der Zusammenhang [Glin04]. Es werden dazu die Beziehungen zwischen den SW-Komponenten betrachtet.

Module mit *zeitlicher Kohäsion* (zusammengefasst ist, was zeitlich miteinander auszuführen ist) sind aus softwaretechnischer Sicht schlecht und sollten daher vermieden werden [Glin04]. In der STEP-X Methodik wird die *funktionale Kohäsion* bevorzugt, in der die SW-Komponente eine in sich geschlossene Funktion realisiert. Dazwischen gibt es verschiedene Kohäsionsstufen wie logische, kommunikative oder sequentielle, die hier nicht näher diskutiert werden sollen. Für Details sei auf [Page88] verwiesen.

Um die logische SW-Struktur möglichst effizient zu erstellen, wird auf die bewährte Vorgehensweise von [Doug98] zurückgegriffen, bei der die folgenden Aktivitäten inkrementell durchgeführt werden müssen:

- ***Objekte identifizieren:*** Der erste Schritt ist das Ermitteln von Objekten. Eine Möglichkeit, das Auffinden der Objekte methodisch zu unterstützen, bietet der Einsatz von CC-Karten (engl.: class responsibility collaboration cards). Diese sind stark anwendungsfallgetrieben und ermöglichen ein einfaches, informelles Vorgehen zur Identifizierung geeigneter Objekte. CRC-Karten enthalten einen Klassennamen, Verantwortlichkeiten (Aufgaben) und besitzen eine Referenz auf weitere Klassen, die bei der Erfüllung der Verantwortlichkeiten notwendig sind. Eine nähere Beschreibung des Einsatzes von CRC-Karten ist in [WWW90] zu finden. Beim Einsatz von CRC-Karten stehen Anwendungsfälle im Vordergrund, da sie bereits über zahlreiche mehr oder weniger intuitiv ermittelte Objekte verfügen. Die Hauptaufgabe der Identifizierung besteht nicht in einer vollständigen Analyse aller Objekte. Vielmehr steht das Ermitteln einiger wichtiger Objekte im Mittelpunkt, mit denen eine erste SW-Struktur erstellt werden kann [SW00]. Fehlende Objekte werden im Verlauf der Analyse ermittelt und in das System integriert.
- ***Objekte strukturieren:*** In diesem Schritt erfolgt die Darstellung der ermittelten Objekte in Form von Klassendiagrammen. Dabei werden alle zusammengehörigen Objekte, die ein Subsystem bilden, in einem gemeinsamen Klassendiagramm zusammengefasst. Die einzelnen Objekte, die zunächst als Klassen dargestellt werden, enthalten jeweils einen aussagekräftigen Namen und falls nötig auch eine dazugehörige Interface-Klasse. Bei der Erstellung des Analyse-Klassendiagramms kann unter Verwendung von CRC-Karten das folgende einfache Verfahren nach [WWW90] angewendet werden:
    - Übertragen aller CRC-Karten als Klassen in das Klassendiagramm,
    - Bildung von Interface-Klassen als Lollipop-Notation,
    - Eintragen der Verantwortlichkeiten in einzelne Klassen,
    - Verbinden von Klassen mittels Assoziationen, falls eine Klasse eine weitere Klasse zur Erfüllung ihrer Verantwortlichkeiten benötigt.

  Nach Abarbeitung aller Punkte, erhält der SW-Entwickler ein oder mehrere Klassendiagramme des (Sub-)Systems.
- ***Klassendiagramme verfeinern:*** Der letzte Schritt befasst sich mit der Verfeinerung der Klassendiagramme. Hierfür werden die Klassen hinsichtlich ihrer Verantwortlichkeiten überprüft. Enthält eine Klasse verschiedenartige Verantwortlichkeiten, so werden diese in weitere Klassen unterteilt. Bei der Verteilung von Verantwortlichkeiten können die Analysemuster von [Fowl99, SW00] verwendet werden, nach deren Schema bestimmte Teilprobleme auf mehrere Klassen verteilt werden können. Zudem können Sequenzdiagramme zur Ermittlung von Verantwortlichkeiten herangezogen werden. Gegebenenfalls müssen aber die in der Anforderungs-

beschreibung noch grobstrukturierten Szenarien präzisiert werden. Zur Verdeutlichung dient ein Auszug aus dem Automatiklauf-Szenario aus Abbildung 4-5 auf Seite 73, in dem ein Doppeldrucktaster zum automatischen Öffnen betätigt wird.

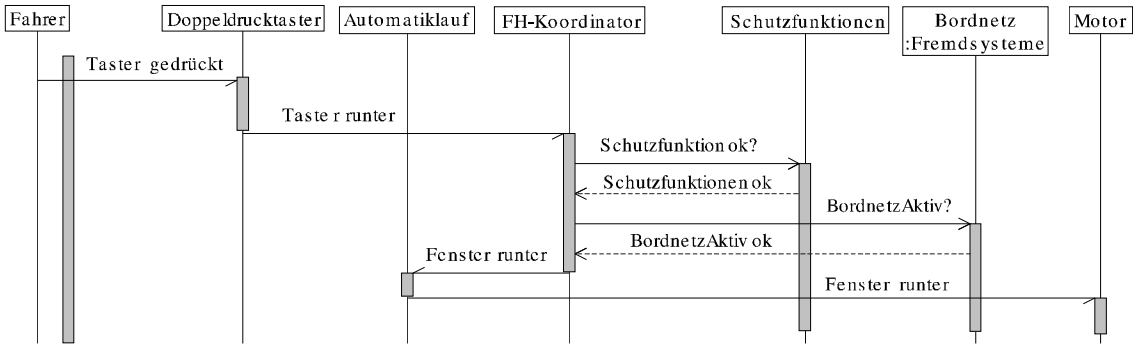


Abbildung 4-9: Detaillierter Automatiklauf

In dem neuen Szenario wird sofort ersichtlich, dass für den Automatiklauf weitere Objekte verantwortlich sind. So werden beispielsweise vor der Ausführung des Fensterlaufs der Systemstatus (Bordnetz) und die Schutzfunktionen überprüft. Erst wenn alle Vorbedingungen erfüllt sind, wird der Motor angesteuert.

Die im Szenario ermittelten Objekte und Beziehungen werden direkt in die zu erstellende SW-Architektur übernommen.

Abbildung 4-10 beinhaltet einen Auszug der logischen SW-Struktur des Fensterhebers. Zur Präzisierung der jeweiligen Funktionalität, erhalten die beiden SW-Komponenten Fensterläufe und Schutzfunktionen mehrere Funktionsklassen.

Durch das Betätigen der *Bedienelemente* werden Signale zur Aktivierung der unterschiedlichen *Fensterläufe* ausgelöst. Je nach Status des *Bordnetzes* ist einer der drei Fensterläufe aktiviert. Zur Senkung bzw. Hebung des Fensters werden die Signale an die jeweiligen *Motoren* weitergesendet. Zu Kontrollzwecken wird der Motorenverlauf in der *Anzeige* dargestellt. Außer den Fensterfunktionen, die vom Benutzer ausgeführt werden können, existieren verschiedene *Schutzfunktionen*, die die Fensterläufe überwachen. Tritt eine Schutzverletzung auf, so werden unmittelbar Gegenmaßnahmen ergriffen.

Nach der Fertigstellung der groben SW-Struktur müssen als nächstes alle benötigten Bedienstellen aus den Lastenheften beschrieben werden. Der hierfür notwendige Analysevorgang wird im folgenden Abschnitt näher erläutert.

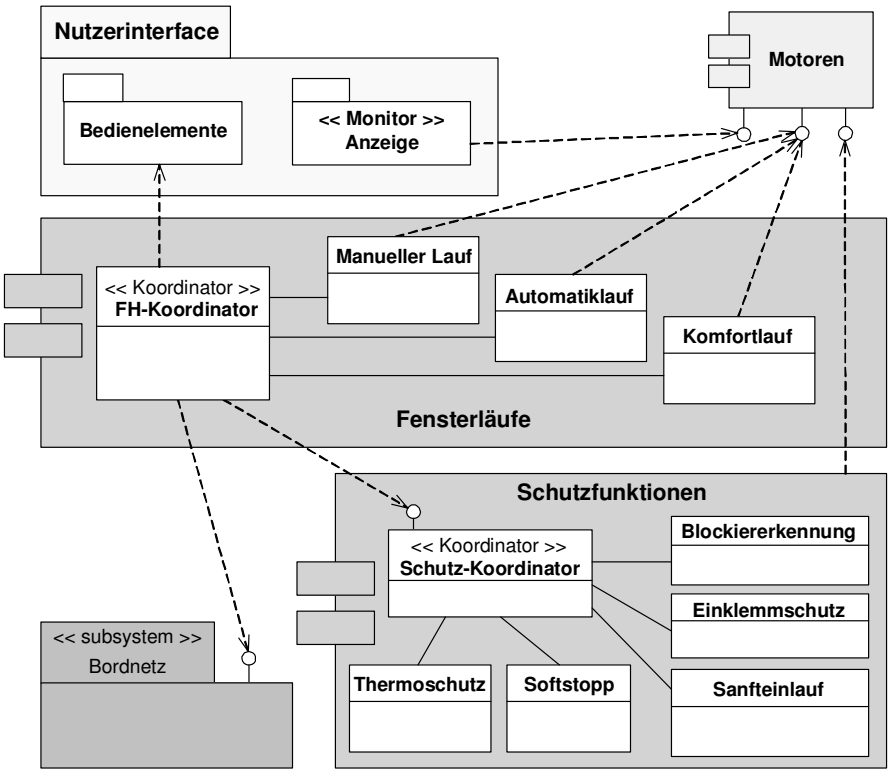


Abbildung 4-10: SW-Struktur des Fensterhebers

4.2.4 Signalanalyse

Wie in Abbildung 4-10 ersichtlich, werden die Fensterläufe des Fensterhebers mit Hilfe einer grafischen Oberfläche angesteuert. Hierzu betätigt der Akteur ein Bedienelement, das als Sensor fungiert. Dieser sendet ein Signal an die entsprechende Fensterheberfunktion, die daraufhin einen oder mehrere Aktuatoren auslöst. In einem Komfortelektroniksystem sind zahlreiche Bedienstellen verfügbar. Durch multifunktionale Eigenschaften können mit nur einem Eingabeelement entweder ein einzelner oder auch mehrere Aktuatoren auf verschiedene Weise angesteuert werden. Der dadurch gewonnene Bedienungskomfort und die Ersparnis von Hardware-Teilen haben eine komplexe Kommunikationsstruktur zur Folge.

Ziel der Signalanalyse ist es daher, nach dem Auslösen einer Bedienstelle alle kausalen Kommunikationsbeziehungen zwischen Akteuren, Bedienstellen, Funktionen und Aktuatoren zu ermitteln und anschließend in Form eines Eingabesignals im Paket *Bedienelemente* abzulegen. In Abbildung 4-11 ist die Zusammensetzung des Signalnamens farblich unterlegt.

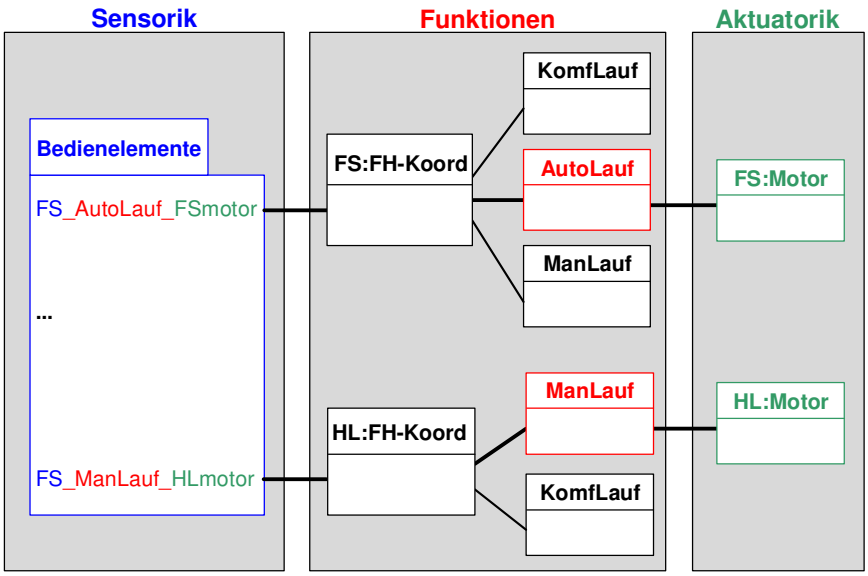


Abbildung 4-11: Verlauf von Signalen

Zur Erhöhung des Kommunikationsverständnisses werden die Eingabesignale in eine einheitliche Syntax überführt, bei der die folgenden Regeln anzuwenden sind:

- Ein Signalname beginnt mit der Bezeichnung des Ortes, in den das Bedienelement integriert ist. Eine Ausnahme bilden die Außenbedienstellen, die sich nicht einem einzelnen Akteur zuweisen lassen und daher direkt mit ihren Namen versehen werden. Die Kennzeichnung ist mit zwei Großbuchstaben abzukürzen.
- Der mittlere Teil des Namens wird durch zwei Unterstriche abgegrenzt und beschreibt die auszulösende Funktion. Sind für die Beschreibung mehrere Teilworte notwendig, so beginnen diese zur optischen Trennung mit einem Großbuchstaben und werden in verständlicher Weise abgekürzt (z. B. entspricht *EinklemmSch* dem Einklemmschutz). Es ist für das Verständnis wichtig, nur solche Abkürzungen zu verwenden, die in ihrer Bedeutung erkennbar bleiben. Ansonsten ist auf ein Abkürzungsverzeichnis im Lastenheft zu verweisen.
- Das Namensende kennzeichnet sowohl den Einbauort als auch den anzusteuernden Aktuator. Im Gegensatz zum Einbauort, der mit zwei Großbuchstaben abgekürzt wird, sollte der Name des Aktuators ein aussagekräftiger Ausdruck sein. Werden mehrere Aktuatoren angesteuert, so sind diese nacheinander aufzulisten. Zum Beispiel bedeutet die Abkürzung *FSBSmotor*, dass die Motoren der Fahrer- und Beifahrerseite angesprochen werden. Falls durch ein Signal alle Fenstermotoren gleichzeitig angesteuert werden, so kann statt einer Auflistung auch ein Sternchen verwendet werden. Beispielsweise bedeutet *FF_KomLauf_**, dass durch die Funkfernbedienung ein Komfortlauf ausgelöst wird, der alle Fenster gleichzeitig ansteuert. Hat das Betätigen einer Bedienstelle eher eine Auswirkung auf das Verhalten einer Funktion, so wird statt eines Aktuators der Funktionsname gewählt. Der Signalname *FS_KindSich_HLHRlauf* bedeutet in diesem Fall, dass die hinte-

ren Fensterläufe aktiviert bzw. deaktiviert sind, nachdem der Fahrer die Kindersicherung betätigt hat.

Formal ergibt sich für die Signalbeschriftung folgendes Syntaxdiagramm:

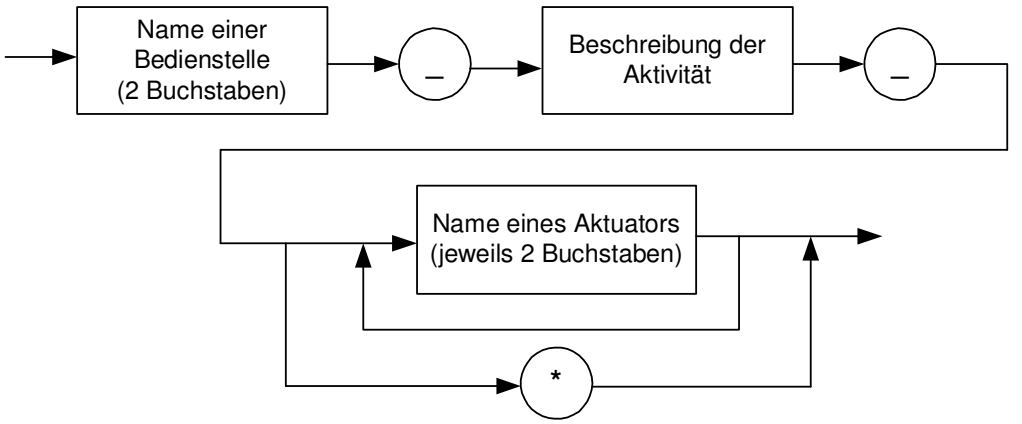


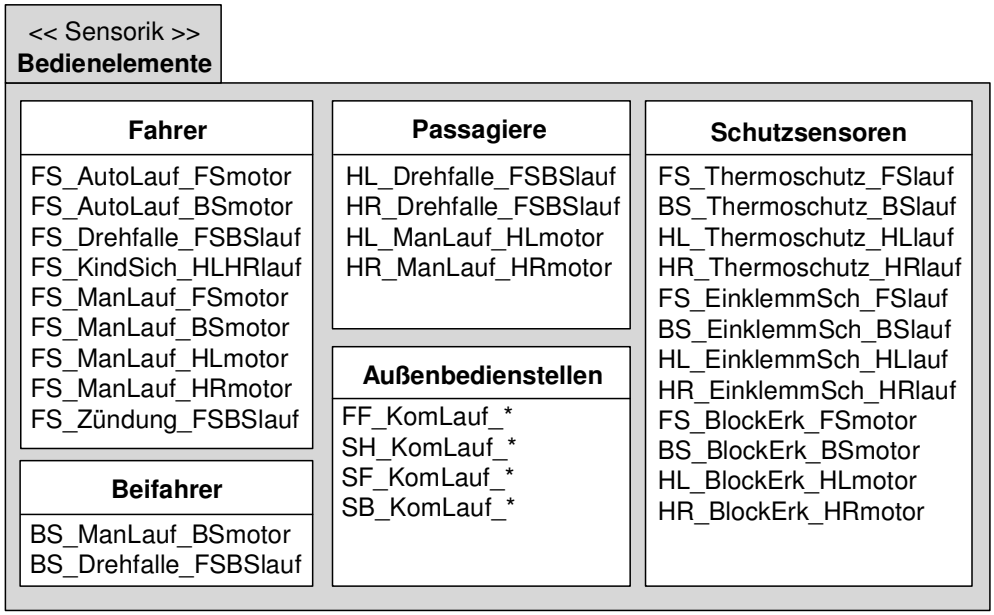
Abbildung 4-12: Syntaxdiagramm zur Beschriftung von Signalen in der Analyse

Eine detaillierte Signalspezifikation, wie Art (analog, diskret, asynchron, synchron), Richtung (Senden oder Empfangen) oder Zugriff (direkte Attributmodifizierung oder Methodenaufruf) ist zunächst genauso nebensächlich wie die Aufteilung der Signale in Steuerungs- und Datensignale. Eine derartige Detaillierung erfolgt im späteren Grobentwurf.

Neben der Ermittlung von Eingabesignalen, werden im Rahmen der Signalanalyse zusätzliche Funktionsrückgabewerte im Paket *Anzeige* gesammelt, die während der Simulation als Nachweis über die korrekten Funktionsläufe dienen. Eine syntaktische Definition ist für diese Art von Signalen irrelevant, da eine Kausalität nicht besteht.

Nach der Definition aller benötigten Signale, erfolgt eine Überprüfung der Signale auf Vollständigkeit. Hierfür dienen die in Kapitel 4.2.3 erstellten Sequenzdiagramme, die bereits die kausalen Signalläufe nach Auslösen einer Bedienstelle aufzeigen. Durch einen direkten Vergleich zwischen den ermittelten Signalen und den Sequenzdiagrammen werden unvollständige Spezifikationen ersichtlich. Eine zusätzliche Verlinkung zwischen Signalen und Szenarien schließt fehlende sowie redundante Informationen aus und erhöht somit die Qualität des Analysemodells.

Zur Erhöhung der Übersichtlichkeit über alle ermittelten Signale, wird die Struktur der beiden Pakete Bedienelemente und Anzeige durch Signalklassen verfeinert. Dabei wird unter einer Signalklasse eine Klasse verstanden, die nur über Signale in Form von Attributen verfügen darf. Als erstes wird für jeden Akteur, der eine Systemfunktion auslösen kann, eine eigene Signalklasse erstellt. Im nächsten Schritt werden Bedienstellen zusammengefasst, die sich nicht direkt einem Akteur zuordnen lassen. Als Beispiel sei hier auf die Außenbedienstellen verwiesen. Die Schutzsensoren, die nicht von Akteuren, sondern vom System ausgelöst werden, bilden die dritte Art von Signalklassen. Abbildung 4-13 zeigt ein vorläufiges Ergebnis der Signalanalyse.



FS = Fahrerseite, BS = Beifahrerseite, HL = Hinten Links, HR = Hinten Rechts, SF = Schlüsselschalter FS, SH = Schlüsselschalter Heck, SB = Schlüsselschalter BS, BlockErk = Blockiererkennung, FF = Funkfernbedienung, KindSich = Kindersicherung

Abbildung 4-13: Signalklassen für die Bedienelemente

Für Fahrzeugsysteme mit wenigen Bedienelementen kann alternativ auch eine gröbere Struktur gewählt werden. In der Praxis werden gelegentlich alle Eingabesignale aus der Umgebung in einer einzigen Klasse zusammengefasst.

Die Darstellung in Form von Signalklassen dient zusätzlich als Basis für den Grobentwurf, in dessen Verlauf eine weitere Verfeinerung der Signale vorgenommen wird. In der Analysephase ist eine Überspezifizierung von Signalnamen durch zusätzliche Adverbien zu unterbinden, da z. B. für jeden Status eines Fensters (runter, hoch, stoppen, reversieren) eine explizite Signaldefinition notwendig wäre (z. B. *FA_AutoLaufRunter_FAmotor*). Im Grobentwurf dahingegen können die Signale als Methodenaufrufe dargestellt werden, so dass beispielsweise die Richtungsangabe über Parameter erfolgen kann.

Mit der Spezifikation der Signale ist das Analysemodell fertiggestellt. Abschließend werden den einzelnen SW-Einheiten aus Abbildung 4-7 die in Kapitel 3.2.3 auf Seite 54 ausgewählten Werkzeuge und Notationen zugewiesen. Dies ist notwendig, um das weitere Modellierungsvorgehen im Grobentwurf mit den Fachexperten abzustimmen.

SW-Einheiten	Notationen	Werkzeuge
Fensterheber	UML-Diagramme	Rhapsody
Strecke	Blockdiagramme	MATLAB/Simulink
GUI	animierbare GUI-Elemente	Altia Design mit Automotive-Bibliothek
Fremdsysteme	hybride Beschreibungs-sprachen	UML Tools, MATLAB Toolbox, ASCET-SD

Tabelle 4-1: Zuweisung von Notationen und Werkzeugen

Zusammenfassend ist festzustellen, dass das Analysemodell eine gemeinsame Basis für Diskussionen zwischen OEMs und Zulieferern schafft. Dadurch kann einerseits frühzeitig auf Änderungswünsche der Auftraggeber reagiert werden und andererseits können die Systemanforderungen auf Vollständigkeit und Korrektheit überprüft werden. Zugleich bildet das Analysemodell die Grundlage für die weiteren Entwurfsphasen.

4.3 Funktionaler Grobentwurf

Die Aufgabe des funktionalen Grobentwurfs besteht in der Präzisierung des Analysemodells mit dem Ziel, ein konzeptionelles simulierbares Gesamtmodell zu erhalten. In diesem Entwicklungsstadium werden noch keine hardwarespezifischen Implementierungsentscheidungen, wie die Wahl einer geeigneten Programmiersprache, eines Betriebssystems oder der tatsächlich einzusetzenden Hardware, getroffen. Die dadurch erlangte Plattformunabhängigkeit hat den Zweck, die Entwicklung von Funktionen und deren Wiederverwendung, nicht aber deren Einbettung in das System, hervorzuheben [Seem00].

Das V-Modell enthält für den Grobentwurf nur wenige Vorgaben bezüglich der Entwicklung von SW-Funktionen. Mit der Kernaktivität SE 4 wird lediglich die Erstellung einer statischen SW-Architektur gefordert. Erst zu einem späteren Zeitpunkt soll die Entwicklung des Systemverhaltens vorgenommen werden. Durch einen iterativen Prozessverlauf ist der Grob- und Feinentwurf solange abwechselnd durchzuführen, bis die gesamte Funktionalität fertiggestellt ist.

Im Gegensatz zu dieser Vorgehensweise wird im Rahmen dieser Arbeit ein Vorgehen propagiert, welches die strukturellen und dynamischen Aspekte bereits im Grobentwurf vereinigt, ohne dabei auf implementierungsnahe Details wie z. B. Tasterentprellung[38] einzugehen. Im anschließenden Feinentwurf werden die einzelnen SW-Funktionen präzisiert und hardwarespezifische Anpassungen vorgenommen. Dies hat zunächst den Vorteil, dass das Modell vor der Implementierung ein gewisses Abstraktionsniveau behält und somit das Verständnis erleichtert. Darüber hinaus können bereits zu einem sehr frühen Zeitpunkt weitere Entwicklungsaspekte, wie die Kopplung semantisch unterschiedlicher Modelle, die Erstellung von grafischen Bedienoberflächen zu Testzwecken und Verteilungskriterien für logische Steuergerätenetzwerke, betrachtet werden.

Das Vorgehen im Grobentwurf basiert auf den in der Analysephase identifizierten Subsystemen. Diese werden zunächst mit Hilfe des Top-Down-Verfahrens verfeinert oder durch eventuelle technische Randbedingungen neu strukturiert sowie hinsichtlich ihrer Realisierbarkeit überarbeitet. Des Weiteren wird zur Beschreibung des Systemverhaltens die statische SW-Architektur durch Statecharts ergänzt und anschließend durch grafische Simulationswerkzeuge validiert. Um ein realistischeres Systemverhalten zu ermöglichen, erfolgt eine Anpassung der Aktuatorikklassen sowie der Systemumgebung, die jeweils durch regelungstechnische Modelle beschrieben werden. Die Co-Simulation dieser hybri-

[38] Das Loslassen eines gedrückten Tasters kann ein hardwaretechnisches Nachschwingen verursachen, wodurch mehrere Impulse ausgelöst werden. Dies muss von der Software erkannt und behoben werden.

den Modelle wird durch eine Werkzeugkopplung ermöglicht. Weist das Gesamtsystem während der Simulation keine Fehler auf, so erfolgt im nächsten Schritt eine Verteilung der Funktionen auf ein logisches Steuergerätenetzwerk. Abschließend werden die Ergebnisse des Grobentwurfs mit dem Lastenheft verlinkt. Die hier kurz vorgestellte Vorgehensweise wird im Folgenden näher erläutert.

4.3.1 Detaillierung der SW-Struktur

Bei der Detaillierung der SW-Struktur steht die Bildung und Kommunikation der SW-Komponenten im Vordergrund. Dazu wird auf der Basis des Analysemodells die SW-Architektur mittels der *funktionalen Dekomposition* in kleinere Funktionseinheiten hierarchisch zerlegt. Die hierarchische Strukturierung erhöht die Übersichtlichkeit der Signalverläufe, wobei die Anzahl der Hierarchieebenen bei der hier durchgeführten Fallstudie auf vier Ebenen begrenzt wird.

Zur Wahrung der in Kapitel 4.2.2 genannten Kopplung enthält jede SW-Komponente eine eigene Koordinatorklasse. Koordinatoren sind Kontrollmechanismen, die die jeweilige SW-Komponente unterstützen, indem sie eintreffende Signale aus der Umgebung aufnehmen und diese nach einer Signalbearbeitung an die betreffenden Funktionsklassen weiterleiten. Dadurch wird ein unerlaubter Eingriff von außen in das (Sub-)System verhindert. Des Weiteren dienen Koordinatorklassen zur Koordinierung der Kommunikation zwischen einzelnen Funktionen einer gemeinsamen SW-Komponente. Somit lassen sich beispielsweise Ausführungsprioritäten und Interrupts besser handhaben. Der Signalaustausch zwischen Funktionsklassen bzw. SW-Komponenten findet somit ausschließlich über Koordinatorklassen statt.

Um Signalverläufe zwischen Hierarchieebenen von Signalverläufen zwischen Funktionsklassen unterscheiden zu können, wurden in STEP-X die Begriffe *horizontale* und *vertikale Kommunikation* aus dem FORSOFT II Projekt [BBEF+98] aufgegriffen. Die horizontale Kommunikation stellt den Datenfluss zwischen den einzelnen Funktionsklassen auf gleicher Hierarchieebene dar, wogegen die vertikale Kommunikation in Form einer Assoziation den Signalfluss zwischen den Hierarchieebenen verdeutlicht.

In Abbildung 4-14 ist eine hierarchische Struktur mit den beiden Kommunikationsarten skizziert. Auf der obersten Ebene ist die Fensterheberfunktionalität als eine einzelne Klasse dargestellt. Die mittlere Ebene detailliert diese Funktionalität, indem sie die verschiedenen Fensterläufe von den Schutzfunktionen separiert. In der untersten Ebene werden alle Funktionsklassen wiederum in kleinere Funktionsklassen zerlegt, wobei die Schutzfunktionen als Beispiel dienen.

Das soeben vorgestellte Prinzip der Kommunikationsstruktur wurde bereits in [Stei02, BRS00, MHPK+04] anhand einer Fallstudie aus dem Automobilbereich umgesetzt und detailliert beschrieben. Im Rahmen dieser Arbeit wird diese Vorgehensweise aufgegriffen und projektspezifisch angepasst. Zur Darstellung der SW-Struktur sind drei Schritte notwendig.

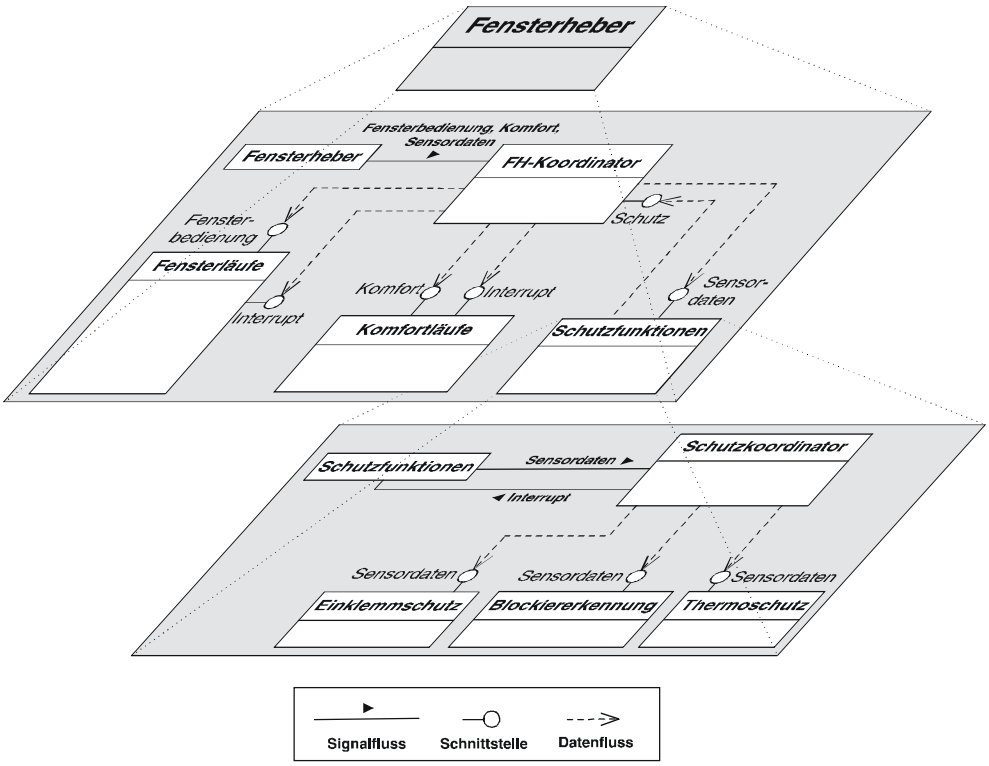


Abbildung 4-14: Prinzip der vertikalen und horizontalen Signalverläufe nach [BRS00]

Erstellung der Kommunikationsstruktur

Im ersten Schritt werden die aus dem Analysemodell identifizierten Funktionsklassen mit Koordinatorklassen zu Funktionseinheiten in Form von Klassendiagrammen zusammengesetzt. Im Gegensatz zu der SW-Struktur aus dem Analysemodell, wird hier nicht nur die reine Funktionalität betrachtet, sondern auch die Optimierung der Funktionszusammenlegung. Beispielsweise wurde in der Abbildung 4-10 der Analysephase der Komfortlauf als eine eigenständige Klasse realisiert. Bei genauerer Betrachtung der Funktionsweise eines Komfortlaufs wird schnell ersichtlich, dass eine eigene Komfortfunktion pro Fensterheber unnötig ist. Stattdessen kann diese durch einen zentralen Koordinator ersetzt werden, welcher die jeweiligen Fensterheber steuert. Die tatsächliche Umsetzung des Komfortlaufs im Fensterheber kann dann durch den manuellen Fensterlauf erfolgen. Obwohl diese Art von Maßnahmen zur Optimierung der SW-Struktur beitragen, sollte stets die Modularisierung und die damit verbundene Flexibilisierung nicht vernachlässigt werden. Ein Optimum zwischen Modularisierung und Optimierung durch Zusammenlegung von Funktionen ist schwer realisierbar und kann nur durch erfahrene Entwickler individuell erbracht werden.

Des Weiteren werden zusammengehörige Klassen durch Assoziationen mit Multiplizitätsangaben verbunden. Die Kommunikation zwischen Koordinator- und Funktionsklassen verläuft ausschließlich durch wohldefinierte Schnittstellen. Im Gegensatz zu [Rau01a], der sieben verschiedene Schnittstellentypen differenziert, wird in dem hier vorgestellten Grobentwurf, wie von [Bued00] vorgeschlagen, nur zwischen internen und externen Schnitt-

stellen unterschieden:

- Interne Schnittstellen sind das Resultat einer Dekomposition eines Systems. Durch sie kann eine Kommunikation zwischen Funktionen realisiert werden, ohne den inneren Aufbau der Funktion zu kennen. Die Darstellung von internen Schnittstellen erfolgt häufig in Form der Lollipop-Notation.

- Durch externe Schnittstellen kann ein (Sub-)System mit der Umgebung (externe Systeme, Werkzeuge) kommunizieren, ohne seinen Aufbau und seine Funktionsweise bekannt geben zu müssen.

Zur Sicherstellung der SW-Qualität werden Namenskonventionen für Klassen und SW-Komponenten eingeführt, die zur Verständlichkeit und Änderbarkeit der Modelle beitragen und somit die Abstimmung im Team erleichtern. Für die objektorientierte Modellierung wurden zahlreiche Richtlinien [Balz96, Balz97] unterbreitet, die in der STEP-X Methode aufgegriffen wurden. Insbesondere die Vorschläge von [KM00, SW97] fanden eine starke Resonanz. Die Namenskonventionen (Tabelle 4-2) wurden allerdings projektspezifisch angepasst, so dass beispielsweise Klassennamen mit drei Kleinbuchstaben und nicht mit einem Großbuchstaben beginnen.

Name	Typ	Beschreibung
akt	Aktuator	Aktuatorklassen stellen die im System anzusteuernden Aktuatoren dar, d. h. sie können nur Signale empfangen und somit andere Klassen nur passiv beeinflussen.
fkt	Funktion	Funktionsklassen stellen eine (Sub-)Funktion dar. Sie werden im Normalfall über Koordinatorklassen angesprochen bzw. erhalten von ihnen den benötigten Systemstatus.
ext	Extern	Externe Klassen symbolisieren weitere Programme, die das System beeinflussen oder vom System beeinflusst werden.
koo	Koordinator	Koordinatorklassen kontrollieren und steuern den internen und externen Signalverlauf zwischen Funktionen bzw. Systemen.
sen	Sensor	Sensorklassen sind das Gegenstück zu Aktuatorklassen. Sie stellen Ereignisse aus der Umgebung dar und übermitteln diese an das System. Sie selbst werden nicht aktiv beeinflusst.
sys	System	SW-Komponenten vom Typ System stellen einen Zusammenschluss von Funktions- und Koordinatorklassen dar und bilden ein eigenständiges System. Systemklassen repräsentieren die höchste Abstraktionsstufe und sind zu verwenden, wenn keine nähere Beschreibung des Systems benötigt wird.

Tabelle 4-2: Namenskonventionen für Klassen

Abbildung 4-15 stellt einen Ausschnitt einer vorläufigen SW-Struktur dar. Zur besseren Übersicht wurde die Darstellung lediglich auf einige wenige Koordinatoren und Funktionsklassen sowie eine Aktuatorklasse begrenzt. Die unterschiedlichen Signalflüsse werden

nicht explizit angegeben, da die Koordinatorklassen die Kommunikationsstruktur hinreichend erläutern.

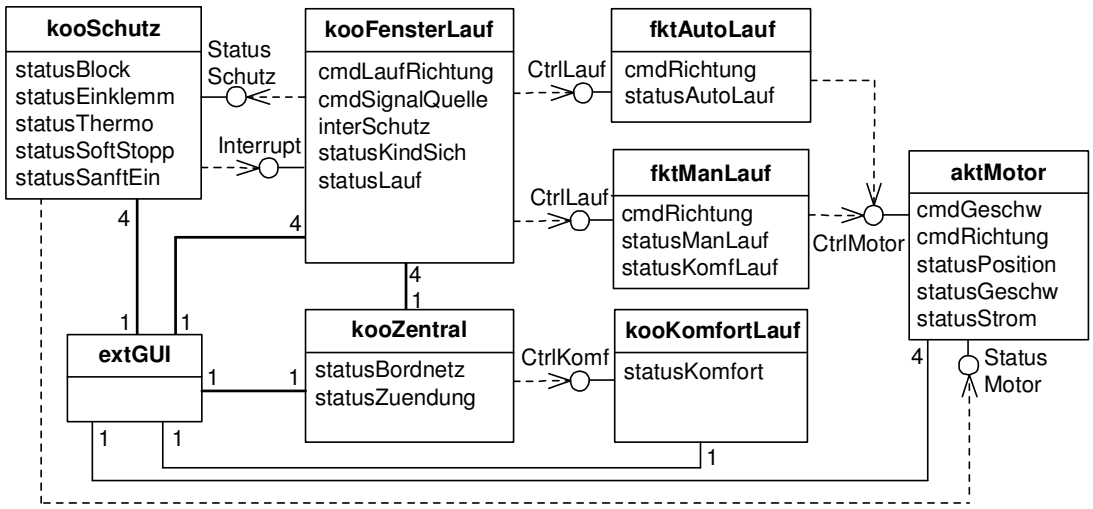


Abbildung 4-15: Ausschnitt der Kommunikationsstruktur für ein 4-türiges Fahrzeug

Das Zusammenwirken der oben dargestellten Klassen ist wie folgt zu verstehen. Für jedes anzusteuernde Fenster existiert ein entsprechender *Fensterlaufkoordinator* (kooFensterLauf), der die Steuerung der Fensterläufe übernimmt. Die vier Fensterlaufkoordinatoren erhalten Signale vom *Zentralkoordinator* (kooZentral), den *Bedienstellen* (extGUI) und dem *Schutzkoordinator* (kooSchutz).

Der Zentralkoordinator als oberste Steuerungseinheit im Body Electronic System regelt unter anderem die Bordnetzfreigabe, die für das Aktivieren der unterschiedlichen Fensterläufe verantwortlich ist. So ist z. B. bei fehlender Bordnetzfreigabe nur ein Komfortlauf über die Außenbedienstellen möglich. Eine solche Freigabe erfolgt beim Einschalten der Zündung, die in diesem Fall durch die externe Klasse *extGUI* realisiert wurde.

Sofern keine Schutzverletzung vorliegt, kann in Abhängigkeit der Bordnetzfreigabe entweder der *Automatiklauf* (fktAutoLauf) oder der *manuelle Lauf* (fktManLauf) durchgeführt werden, die wiederum den dazugehörigen *Fensterhebermotor* (aktMotor) ansteuern. Der Komfortlauf wird vom *Komfortkoordinator* (kooKomfort) vorgenommen. Die dazugehörigen Funktionsklassen sind nicht notwendig, da der Komfortlauf von der Klasse *fktManLauf* übernommen wird.

Die Kommunikation zwischen den einzelnen Klassen geschieht hauptsächlich durch Attribute, die durch Accessoren und Mutatoren[39] ausgelesen bzw. modifiziert werden. Die Attribute und Methoden werden nach Möglichkeit mit Sichtbarkeiten, Datentypen, Parametern und Defaultwerten versehen.

Die Namenskonvention für die Benennung von Attributen hat einen genauso wichtigen Stellenwert wie die Benennung von Klassen. Tabelle 4-3 zeigt die wichtigsten Attributtypen.

[39] Mutator = set-Methode, Accessor = get-Methode

Name	Typ	Beschreibung
cmd	Kommando	Kommandoattribute beeinflussen die zugehörige Klasse durch ihren Wertebereich. Sie werden grundsätzlich von anderen Klassen geändert.
inter	Interrupt	Interruptattribute kennzeichnen eine Ausnahmesituation, die beispielsweise durch Schutzfunktionen ausgelöst wird. Sie haben in der Regel dadurch die höchste Priorität.
status	Status	Statusattribute liefern den momentanen Status eines Systems, um beispielsweise ein bestimmtes Funktionsverhalten aktivieren zu können.

Tabelle 4-3: Namenskonventionen für Attribute

Status- und Interruptattribute nehmen gewöhnlich einen Booleschen Wert an. Dabei bedeutet *true*, dass die dazugehörige Funktion aktiviert ist. Ein *false* entspricht einer Deaktivierung der momentan aktiven Funktion. Beispielsweise besagt das auf true gesetzte Attribut *statusAutoLauf*, dass ein Automatiklauf zurzeit möglich ist.

Kommandoattribute steuern eine Funktion durch ihren Wertebereich, der normalerweise über einen einfachen Boolean hinausragt. Beispielsweise werden durch das Attribut *cmdRichtung* die Richtung sowie die Art des Fensterlaufs angegeben. Das Manipulieren und Abfragen eines Attributs wird stets durch vordefinierte Konstanten durchgeführt, wobei bei gegenteiligen Aussagen möglichst das entsprechende Antonym und nicht die Negation anzuwenden ist. Beispielsweise wird für das Gegenteil der Konstante AKTIV anstatt des Wortes NICHT_AKTIV das Antonym INAKTIV verwendet. Am Beispiel des Fensterlaufs werden in Tabelle 4-4 die zu verwendenden Konstanten angegeben.

Wert	Bedeutung	Name der Konstante
-2	automatisches Öffnen	AUTO_OPEN
-1	manuelles Öffnen	MAN_OPEN
0	Stopp	STOP
1	manuelles Schließen	MAN_CLOSE
2	automatisches Schließen	AUTO_CLOSE

Tabelle 4-4: Konstanten für die Fensterläufe

Diese und weitere Konstanten werden tabellarisch in dem Lastenheft gespeichert und dienen als Referenz für alle Phasen der Entwicklung. Somit kann auf Basis dieser Werte ein zu der Entwicklung parallel stattfindender Test erfolgen.

Bildung des Gesamtsystems

Nach der Erstellung der einzelnen Subsysteme, die eine Kombination von Funktions- und Koordinatorklassen darstellen, werden im zweiten Schritt mit Hilfe der *funktionalen Komposition* die einzelnen Subsysteme zu einem Gesamtsystem integriert. Abbildung 4-16 stellt einen Auszug der gesamten SW-Struktur des Kfz-Komfortsystems dar, wobei in die-

ser Ansicht die Klassen nur aus den Namen besteht.

Für eine intakte Kommunikationsstruktur zwischen den verschiedenen (Sub-)Systemen müssen ggf. weitere Koordinatoren eingebunden werden. So wurde beispielsweise die Koordinatorklasse *kooTuer* eingeführt, um die Kommunikation zwischen den drei Teilsystemen Fensterhebersteuerung, Spiegelsteuerung und Zentralverriegelung auf einem Steuergerät zu managen.

Die Angabe von Kardinalitäten an den Assoziationen erleichtert das spätere Instanziieren der Klassen und gibt einen Einblick in die Architektur des Fahrzeugs. Beispielsweise wird ersichtlich, dass der Türkoordinator *kooTuer* entweder zweimal oder viermal instanziiert wird, so dass bereits zu diesem Zeitpunkt von einem Zwei- bzw. Viertürer ausgegangen werden kann. Ähnlich verhält es sich beispielsweise mit der Spiegelsteuerung.

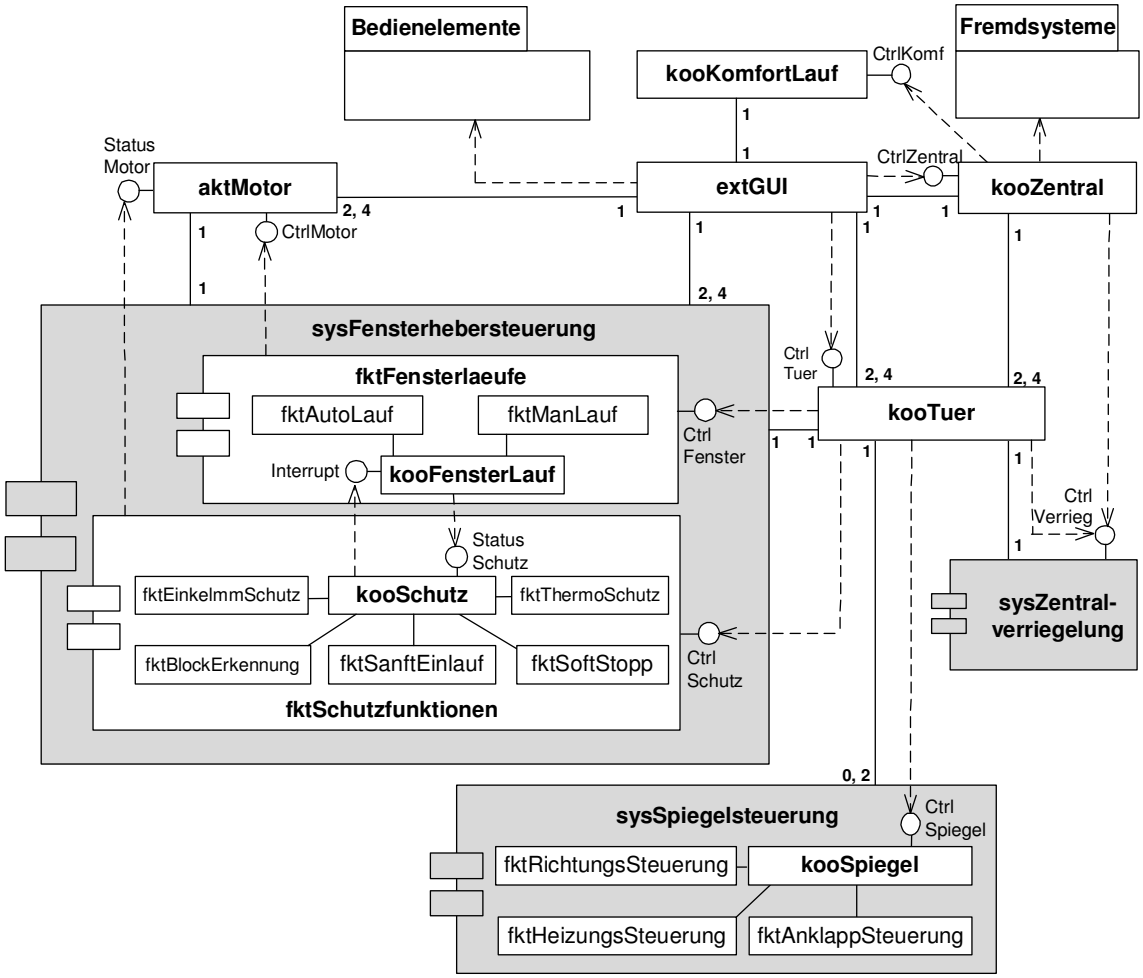


Abbildung 4-16: Die SW-Struktur des Kfz-Komfortsystems

Objektbildung

Im letzten Schritt wird auf Basis der SW-Struktur aus Abbildung 4-16 die Instanziierung vorgenommen. Zur Durchführung der Objektbildung ist es ratsam, jede SW-Komponente, die ein Subsystem darstellt, einzeln zu behandeln und anschließend die übrigen Klassen zu instanziieren. Abbildung 4-17 zeigt beispielhaft das Objektdiagramm der SW-Komponente *sysFensterhebersteuerung* für die Fahrerseite. Zur besseren Übersicht wird auf die

Darstellung der Attribute, SW-Komponenten und Schnittstellen verzichtet.

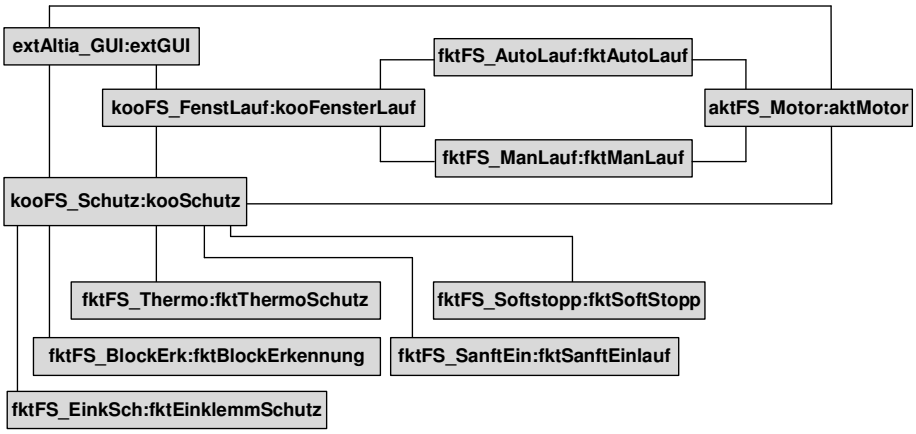


Abbildung 4-17: Objektdiagram der Fensterhebersteuerung für die Fahrerseite

Genau wie bei Klassen und Attributen, leistet die Namenskonvention bei der Bildung von Objektnamen einen wichtigen Beitrag zur Beibehaltung der Übersichtlichkeit und zur Erhöhung des Verständnisses. Der Objektbezeichner setzt sich zunächst aus dem Typ der Klasse zusammen, wie in Tabelle 4-2 bereits vorgestellt. Anschließend folgt eine Abkürzung für den Einbauort der Funktion. Die hier benötigte Abkürzung setzt sich wie in der Analysephase (vgl. Abbildung 4-13 auf Seite 85) aus zwei Großbuchstaben zusammen. Eine Ausnahme bilden externe Klassen, die keinen spezifischen Einbauort aufweisen. Stattdessen werden diese durch den Namen des Programms bzw. des Systems näher präzisiert. Nach einem Unterstrich folgt die Funktionsbezeichnung, die in der Regel einer Abkürzung des Klassennamens entspricht. Ein Doppelpunkt trennt die Objektbezeichnung vom Klassennamen. Abbildung 4-18 veranschaulicht die formale Namensgebung für ein Objekt anhand eines Syntaxdiagramms.

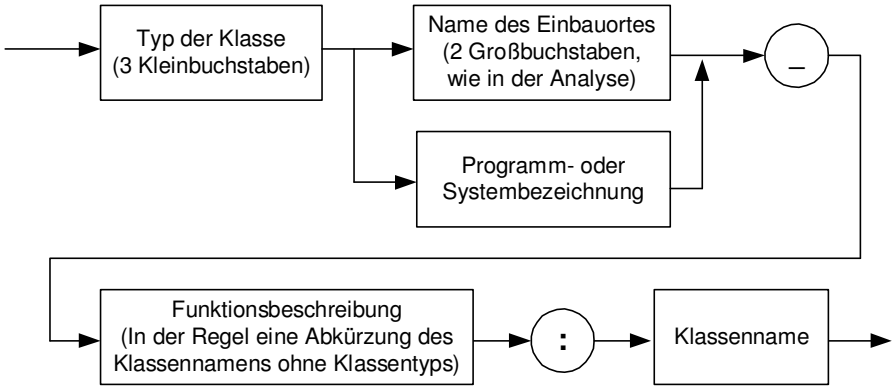


Abbildung 4-18: Syntaxdiagramm zur Beschriftung von Objektnamen

Analog zur Fahrerseite müssen Objektdiagramme für alle Fensterheber und ggf. weitere Subsysteme erstellt werden. Dabei ist zu beachten, dass für die Fensterhebersteuerungen an den Fondtüren standardmäßig keine Automatikläufe vorgesehen sind und somit nur manuelle Läufe instanziiert werden dürfen.

Mit der Bildung aller notwendigen Objektdiagramme ist der Entwurf der Systemstruktur abgeschlossen.

4.3.2 Modellierung des Systemverhaltens

Nach der Fertigstellung der statischen Kommunikationsarchitektur wird die SW-Struktur durch dynamische Aspekte ergänzt. Zur Darstellung des Systemverhaltens in den Koordinatoren und Funktionsklassen werden hauptsächlich Statecharts verwendet.

Allerdings erschweren zustandsbasierte Beschreibungssprachen aufgrund ihrer Variantenvielfalt[40] das Verständnis der zugrunde liegenden Semantik. Trotz der Bestrebungen der OMG, einen einheitlichen Standard zu schaffen, konnte bislang noch keine interpretationsfreie UML Statechart-Semantik definiert werden. Dies liegt vor allem an dem domänenneutralen Verständnis der OMG und an der semiformalen Beschreibung der UML-Spezifikation [OMG03a]. Um trotzdem die geforderte Qualität des Modellierungsprozesses zu wahren, müssen alle an dem Modellierungsprozess beteiligten Personen die verwendete Statechart-Semantik kennen. Zum Verständnis der hier betrachteten Statecharts genügen die Grundlagen aus Kapitel 2.2.2.4 auf Seite 25.

Zur Beschreibung eines komplexen Systemverhaltens reichen Statecharts allein nicht aus. In vielen Fällen ist die Implementierung von bestimmten Klassen (zumeist Hilfsklassen und externe Klassen) mit Hilfe von Programmiersprachen unabdingbar. Die Programmierung ist der Verhaltensmodellierung insbesondere dann vorzuziehen, wenn die Umsetzung durch Statecharts einen inakzeptablen Arbeitsaufwand oder Ressourcenverbrauch zur Folge hätte. In STEP-X fließen daher sowohl Statecharts als auch objektorientierter Programm-Code in die SW-Entwicklung ein. Hierfür wird zunächst das grundsätzliche Systemverhalten mit all seinen anzunehmenden Systemzuständen und Ereignissen aus der Umgebung durch Zustandsdiagramme dargestellt. Funktionen, die zur Laufzeit nur administratorische Arbeiten, wie z. B. Initialisierungsmaßnahmen, Kommunikationsaufbau oder Treiberansteuerung verrichten, werden entweder durch interne Methoden (private) in derselben Funktionsklasse oder durch externe Klassen realisiert.

Bei der Modellierung von Statecharts ist ein methodisches Vorgehen erforderlich, um möglichst einheitliche und verständliche Verhaltensmodelle zu erhalten. In [DKKM+02, DKKP03] wird ein pragmatisches Vorgehen zur Erstellung von Statecharts anhand einer Fallstudie aus dem Automobilbereich vorgestellt. Der dortige Modellierungsprozess basiert allerdings auf einer SW-Architektur, die sich von der hier vorgestellten SW-Struktur (vgl. Abbildung 4-16) weitestgehend unterscheidet. So verläuft beispielsweise die Signalweitergabe ohne Koordinatoren direkt zwischen Funktionsklassen. Darüber hinaus weist die SW-Struktur nur eine schwache Modularisierung auf, wodurch die wenigen Statecharts eine komplexere Struktur erhalten. Eine Funktionserweiterung kann somit einen hohen Modellierungsaufwand nach sich ziehen. Aus diesem Grund wird hier ein eigenes Vorgehen verfolgt, dass auf der Vorgehensweise von [Otto98] basiert. Dabei beginnt die Modellierung der Statecharts mit der Auswahl einer einfachen Schablone (engl.: templates), mit der zunächst eine grobe Strukturierung des Systems vorgenommen wird. Abbildung 4-19 stellt zwei solche Templates dar.

[40] Auf die Unterschiede einiger Statechart-Varianten wird in Kapitel 5.1 auf Seite 117 eingegangen.

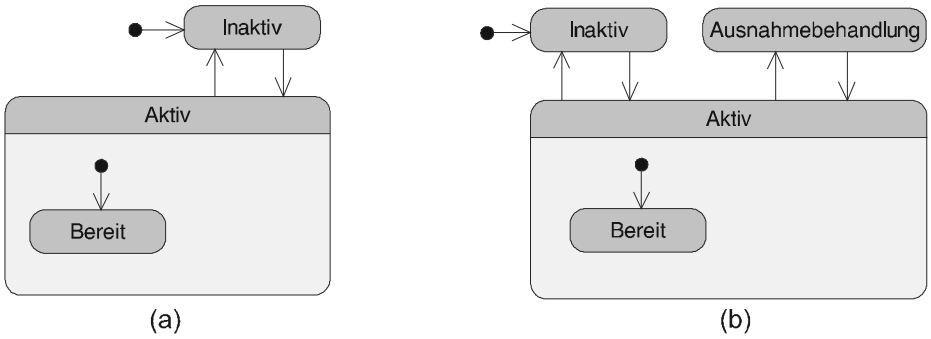


Abbildung 4-19: Zwei einfache Templates a) ohne und b) mit Ausnahmebehandlung

Beide Templates unterliegen einem ähnlichen Aufbau, indem sie über einen aktiven und einen inaktiven Bereich verfügen. Im aktiven Zustandsbereich wird das eigentliche Verhalten abgebildet, wogegen der inaktive Bereich das Ausführen verhindert. Müssen Ausnahmebehandlungen bei der Verhaltensmodellierung berücksichtigt werden, so wird dies durch einen expliziten Zustand gekennzeichnet. Um beim Initialisieren des Statecharts keine Funktion versehentlich auszulösen, ist der inaktive Bereich als Startzustand vorzusehen. Durch die Trennung von aktivem und inaktivem Bereich wird der Modellierungsvorgang vereinfacht, indem die Integration eines Statecharts in das Gesamtsystem ermöglicht wird, ohne dass dessen Funktionsweise fertiggestellt ist. Im Einzelnen bedeutet dies, dass ein Statechart nur dann eine Aktivierungsfreigabe erhalten darf, wenn die dazugehörige Funktionalität fertiggestellt wurde. Zusätzlich wird auch der Integrationstest vereinfacht, da die zu testenden Funktionen durch Setzen einer einzigen Variable dem Gesamtsystem zur Verfügung stehen.

Im Folgenden werden am Beispiel der Koordinatorklasse *kooFensterLauf* aus Abbildung 4-15 die notwendigen Modellierungsschritte zur Darstellung des dazugehörigen Statecharts aufgezeigt. Als erstes wird das ausgewählte Template durch Hinzufügen von weiteren Zuständen detailliert. Wie bereits oben beschrieben, koordiniert der Fensterlaufkoordinator unter anderem die unterschiedlichen Funktionsläufe, d. h. zu jedem Zeitpunkt darf an einem Fensterheber nur eine Betriebsart aktiviert sein. Aus diesem Grund erhält das Template zusätzlich zu dem Initialzustand die drei Zustände *KomfortLauf*, *AutomatikLauf* und *ManuellerLauf*. Diese können je nach Systemstatus und Benutzereingabe einzeln betreten und wieder verlassen werden. Außerdem ist ein Sofortwechsel zwischen den drei Fensterläufen möglich. Die Anordnung der Transitionsverläufe sollte einem orthogonalen Graphen[41] entsprechen und nur bei komplexeren Transitionsverbindungen einen 90° Verlauf annehmen. Eine Ausnahme bilden Transitionen, die zu einem Zeitpunkt nicht fertiggestellt werden konnten. Sie werden mit einem Knick versehen, um sie von abschließenden Transitionen (vgl. Kapitel 2.2.2.4 auf Seite 25) zu unterscheiden. Alternativ können diese Transitionen farblich gekennzeichnet werden, um die abschließende Neuanordnung zu vermeiden. Allerdings wird diese Option nur von wenigen UML-Werkzeugen unterstützt, so dass aus Kompatibilitätsgründen die erstgenannte Maßnahme zu bevorzugen ist (vgl. Abbildung 4-20).

[41] Ein orthogonaler Graph enthält ausschließlich horizontale und vertikale Kanten [EFK01].

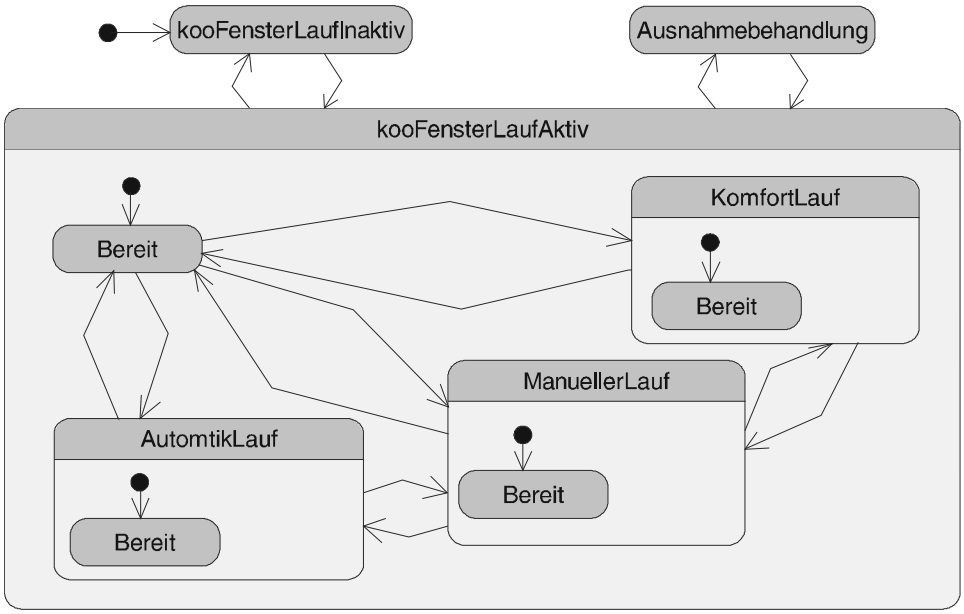


Abbildung 4-20: Der Koordinator *kooFensterLauf* nach der ersten Entwicklungsstufe

Im nächsten Schritt werden die Transitionen beschriftet und die Zustände mit internen Ereignissen versehen. Abbildung 4-21 stellt das Statechart für den Koordinator Fensterlauf dar. Zu Beginn befindet sich die Klasse im Initialzustand *Bereit*, bei dem durch die Methode *kooFensterlaufInit()* zunächst alle benötigten Variablen initialisiert und Konstanten definiert werden. In Abhängigkeit von dem Attribut *statusLauf*, welches wiederum vom Zentralkoordinator beeinflusst wird, findet ein Schalten in den betreffenden Zustand statt. Beim Eintritt in einen der drei Zustände wird die dazugehörige Funktionsklasse (*fktManLauf* bzw. *fktAutoLauf*) aktiviert. Da für den Komfortlauf keine zusätzliche Funktionsklasse vorgesehen ist, wird durch die beiden Attribute *statusManLauf* und *statusKomfLauf* dementsprechend der Betriebsmodus gesetzt. Zugleich wird durch das Attribut *cmdRichtung* die Verlaufsrichtung in der entsprechenden Funktionsklasse angegeben. Des Weiteren werden die zum Initialzustand zurückführenden Transitionen beschriftet.

Sind in einem Statechart alle Zustände erstellt und durch Transitionen verbunden, so wird zur Erhöhung der Lesbarkeit die Anordnung der Modellelemente im Statechart optimiert. Darüber hinaus sollte überprüft werden, in wie weit die Zustandsmenge zu reduzieren ist, um Speicherressourcen zu minimieren. Im Gegensatz zur Anordnung der Elemente, welche mit Hilfe von Modellierungsregeln (vgl. Kapitel 5.1.1 auf Seite 118) bewerkstelligt werden kann, ist das Reduzieren von Zuständen bei gleichzeitiger Beibehaltung der Lesbarkeit ein schwieriger Prozess. In dem Beitrag von [Jin98] wird z. B. ein Vorgehen zur Verringerung von Zuständen vorgestellt.

Ist die Hauptfunktionalität fertiggestellt, so werden anschließend die Schutzfunktionen integriert. Hierfür werden die Beschriftungen der Transitionen angepasst und der Zustand *Ausnahmebehandlung* verfeinert.

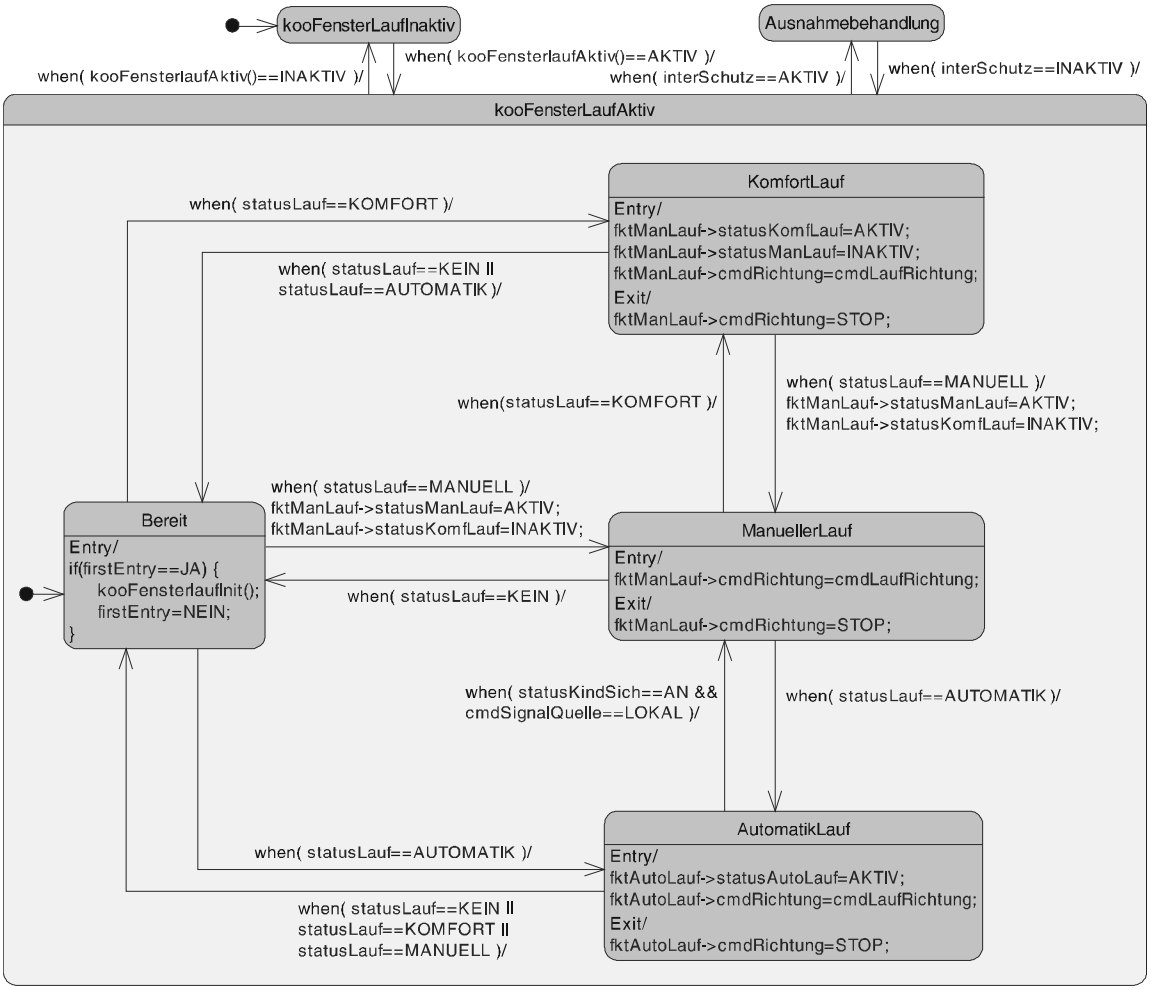


Abbildung 4-21: Statechart der Klasse *kooFensterlauf*

Wie in Abbildung 4-21 dargestellt, erfolgt der Zugriff noch über Klassen und nicht auf die daraus instanziierten Objekte. Diese notwendige Anpassung erfolgt im letzten Modellierungsschritt entweder manuell oder (halb)automatisch durch die Unterstützung des gewählten Entwicklungswerkzeugs. Für das Objektdiagramm der Fensterhebersteuerung aus Abbildung 4-17 würde im Statechart jede aufgerufene Klasse durch einen entsprechenden Objektnamen ersetzt werden. In diesem Falle würde beispielsweise der Klassenname *fktManLauf* in den Objektnamen *fktFS_ManLauf* umbenannt.

4.3.3 Simulation der Steuergeräte-Software durch eine GUI

Nach der Modellierung des dynamischen Verhaltens erfolgt mit Hilfe eines im Modellierungswerkzeug integrierten Simulators die Validierung der Zustandsdiagramme. Dabei können zur Durchführung des Simulationsvorgangs sowohl Ereignisse ausgelöst als auch Attribute modifiziert werden. Als Reaktion des Systems wird der momentan auftretende Zustandswechsel visuell hervorgehoben. Des Weiteren werden die beobachteten Variablenwerte aktualisiert. Ein derartiger Simulationslauf ähnelt dem aus der Programmierung bekannten Debugging, bei dem während der schrittweisen Programmausführung Variablen manipuliert werden können, um die Reaktion des Systems zu überprüfen.

Der Einsatz der oben beschriebenen Simulatoren bietet in erster Linie eine einfache Möglichkeit das Verhaltensmodell frühzeitig zu validieren und somit erste auftretende Fehler zu beheben. Dennoch sind mit dieser Vorgehensweise auch einige Schwächen verbunden. Zum einen ist die Bedienbarkeit der meisten in den Modellierungswerkzeugen integrierten Simulatoren unzureichend. Eine Möglichkeit, mehrere hintereinander auftretende Ereignisse in Form einer Skriptsprache auszulösen, um somit vollständige Szenarien automatisch durchzuführen, ist in den meisten Fällen nicht gegeben. Zum anderen können Zwischenergebnisse während des Simulationsvorgangs nur schwer nachvollzogen werden. Dies liegt einerseits an den Variablen, deren Werte nicht sofort ersichtlich sind und die daher meistens einer Zuordnung in dem bestehenden Kontext bedürfen, andererseits hat ein häufiger Signalaustausch zwischen Statecharts während der Simulation eine schnell wechselnde Ansicht der Zustandsdiagramme zur Folge, so dass nicht alle Zustandsübergänge wahrgenommen werden können. Eine genauere Überprüfung des Systemverhaltens ist dann nur noch durch das Nachvollziehen der protokollierten Daten nach dem Simulationsvorgang möglich. Der größte Nachteil liegt jedoch in der Einarbeitung des Testers in das zu überprüfende Modell. Wenn dieser nicht im Modellierungsprozess involviert ist, so bedarf es zunächst einer ausgiebigen Analyse der Verhaltensmodelle, um sowohl die Struktur als auch die Semantik zu verstehen. Erst dann kann eine sinnvolle Validierung vorgenommen werden.

Durch die Verwendung eines Front-End Tools, wie z. B. dem Altia Design (vgl. Kapitel 3.2.3), werden die oben genannten Schwächen relativiert, da sich eine grafische Bedienoberfläche optisch von den technischen Verhaltensmodellen abkoppelt. Somit steht dem Tester eine Sicht zur Verfügung, die dem realen System ähnelt.

Die Bedienung des Systems erfolgt wahlweise durch die Maus oder Tastatur, wobei das parallele Auslösen mehrerer Funktionen möglich ist. Zusätzlich kann mit Hilfe eines Skripts ein vordefinierter Simulationsvorgang durchgeführt werden. Die Reaktion des Systems erfolgt nicht mehr primär durch das Färben von Zuständen und Anzeigen von Variablenwerten, sondern wird durch animierte Modellierungselemente, die das zukünftige System in Form und Verhalten abbilden, dargestellt. Somit interessiert den Funktionstester nicht mehr das Verhaltensmodell oder das zu verwendende Werkzeug mit der dazugehörigen Semantik, sondern viel mehr, ob das funktionale Verhalten den gestellten Anforderungen genügt. Aus Sicht des Projekt- und Qualitätsmanagements steigt durch den Einsatz derartiger Werkzeuge die Effektivität der SW-Entwicklung [Eder02].

Abbildung 4-22 stellt eine Bedienoberfläche des entwickelten Komfortsystems dar, welches die drei Teilsysteme *elektronischer Fensterheber*, *elektrische Spiegelverstellung* und *Zentralverriegelung* integriert. In der Mitte der GUI sind die Eingabeelemente der Türen in vier einzelne Segmente unterteilt. Diese enthalten neben den Bedienstellen für die Steuerung des jeweiligen Fensters einen Taster zum Öffnen und Schließen der dazugehörigen Türen. Des Weiteren wird die aktuelle Position der jeweiligen Fensterscheibe durch vertikale Balken dargestellt. Zur Simulation des Einklemmschutzes steht jedem Bedienfeld ein entsprechender Taster zur Verfügung. Darüber hinaus enthalten alle Bediensegmente

einen Lock-/Unlock Taster zum Ver- bzw. Entriegeln der jeweiligen Türen. Das Segment der Fahrertür enthält zudem Bedienelemente zur Ansteuerung der anderen Fenster, einen Taster für die Kindersicherung und ein Zündschloss. Des Weiteren ist ein Joystick zur Bedienung der beiden Spiegel vorhanden sowie ein Drehknauf zur Auswahl verschiedener Spiegelfunktionen, wie z. B. eine Spiegelheizung. Die momentan aktive Spiegelfunktion wird innerhalb der beiden Spiegelelemente angezeigt. Durch die oberhalb der Spiegel angebrachten Ausrastschalter kann der Normierungslauf für die beiden Spiegel initialisiert werden. Zudem ist eine Aktivierung der synchronen Spiegelverstellung möglich.

Außerhalb der vier Bedienfelder sind die Anzeigen für die Fensterpositionen angebracht. Diese dienen der Steuerung als Ist-Werte, die von den tatsächlichen Fensterpositionen abweichen können, beispielsweise durch Gewalteinwirkung oder längeren Gebrauch.

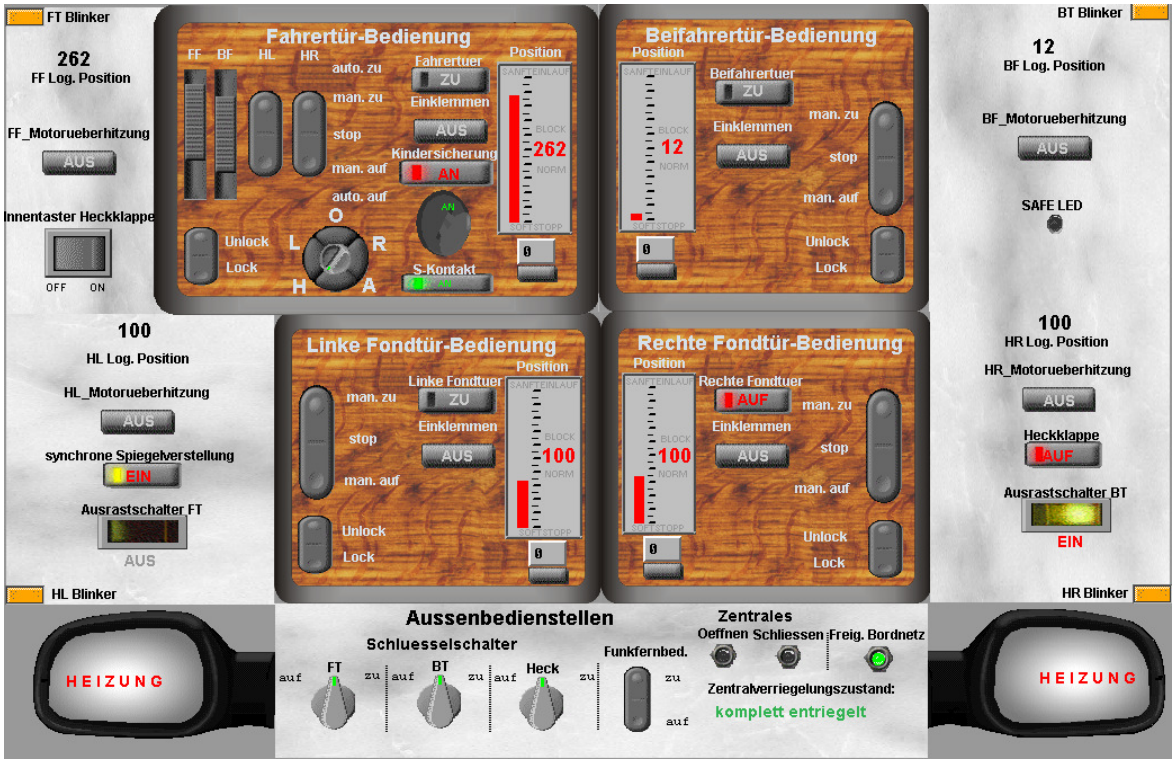


Abbildung 4-22: Graphische Benutzeroberfläche zur Simulation des Komfortsystems

Um jedoch eine Blockiererkennung auszuführen, bedarf es zunächst einer Normierung der Fensterscheiben, die eine Anpassung zwischen den logischen und tatsächlichen Positionsangaben vornimmt. Hierfür lässt sich die tatsächliche Fensterposition mit einem Taster unterhalb der Positionsanzeige einstellen.

Im unteren Bereich der Benutzeroberfläche befinden sich die Schlüsselschalter, die Funkfernbedienung sowie zahlreiche Kontrolllämpchen für die verschiedenen Systemfunktionen, wie z. B. den Status der Schutzläufe und der Kindersicherung. Zusätzlich enthält die Benutzeroberfläche Taster und Kontrolllämpchen zur Steuerung der Zentralverriegelung sowie Blinker zur Quittierung der Eingaben der Zentralverriegelung. Der Zustand der Zentralverriegelung wird zudem im unteren Bereich der Benutzeroberfläche angezeigt.

Für die gemeinsame Simulation zwischen GUI und Zustandsdiagrammen ist eine bidirek-

tionale Kommunikation notwendig. Diese händelt einerseits die Benutzereingaben aus Altia Design und leitet diese an die entsprechenden Teilmodelle weiter, und andererseits müssen Bestätigungssignale vom Modell an die GUI gesendet werden. Eine detaillierte Beschreibung zur Anbindung von UML-Modellen an die Altia-Oberfläche ist [KM03] zu entnehmen, so dass hier nur eine kurze Erläuterung erfolgt.

Das gesamte Kommunikationsmanagement zwischen den Verhaltensmodellen und der GUI wird durch eine einzelne Klasse realisiert, so dass langwierige Änderungen am Verhaltensmodell nicht erforderlich sind. Die Klasse *extGUI* enthält, wie in Abbildung 4-23 dargestellt, nur einen einzigen Zustand (*infAltia_GUI*). Über die Aktionen an der Initial-Transition wird zunächst die Verbindung zu Altia Design hergestellt. Anschließend wird durch zwei Methoden in der do-Aktivität das Kommunikationshandling gesteuert und überwacht. Der Methodenaufruf *EingabenVonUmweltAnModell()* überprüft, ob Ereignisse aus der Umgebung (Altia Design) auftreten und leitet diese an die entsprechende Koordinatorklasse weiter. Die darauffolgende Methode *StatuswerteVomModell()* sendet eventuelle Rückgabewerte des Systems an die GUI. Durch das im Zustandsdiagramm verwendete Timer-Ereignis werden diese beiden Methoden in regelmäßigen Zeitabständen[42] aufgerufen.

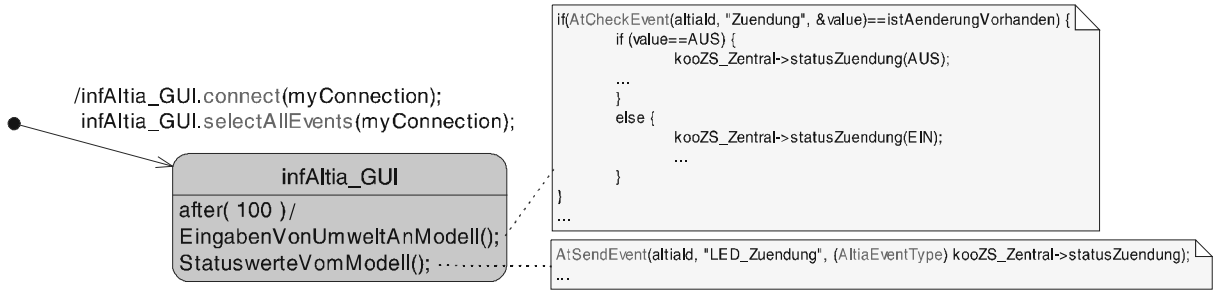


Abbildung 4-23: Das Zustandsdiagramm der externen Klasse *extGUI*

Durch den Einsatz eines Front-End-Tools ist es grundsätzlich möglich, ohne technische Kenntnis über den Aufbau des Modells das Systemverhalten zu simulieren und dementsprechend zu überprüfen. Mit dieser Lösung ist allerdings bislang nur die später zu implementierende Steuergeräte-Software simulierbar. Um das Systemverhalten auch unter realeren Bedingungen überprüfen zu können, bedarf es einer zusätzlichen Integration von Umgebungsmodellen. In STEP-X wird dies mit Hilfe einer Co-Simulation der verwendeten Modellierungswerkzeuge realisiert, die im nächsten Abschnitt erläutert wird.

4.3.4 Integration und Simulation kontinuierlicher Umgebungsmodelle

Das Verhalten hardwarelastiger Funktionen ist stark von den physikalischen Gegebenheiten, wie dem Reibungswiderstand der Fensterscheibe oder der Temperatur, abhängig. Daher ist zur Überprüfung der korrekten Funktionsweise eines eingebetteten SW-Systems ein präzises Strecken- bzw. Umgebungsmodell erforderlich. Die Begriffe Strecke und Umgebung werden wie folgt definiert:

[42] Für einen Windows PC mit 1,6 GHz und 512 MByte RAM hat sich eine Abtastrate von 100ms bewehrt.

Strecke/Umgebung

Ein Streckenmodell bildet das dynamische Verhalten eines realen physikalischen Prozesses ab, das durch eine Steuerungs- oder Regelungsfunktion gezielt beeinflusst wird. Werden in einem Streckenmodell Sollwertgeber, Sensoren und Aktuatoren integriert, so wird dieses als ein Umgebungsmodell bezeichnet [SZ03].

Abbildung 4-24 stellt das Konzept der Integration einer SW-Steuerung und eines Umweltmodells in Form eines Regelkreises dar. Die Eingaben des Fahrers (Sollwerte *W**) werden vom Sollwertgeber aufgenommen und als Führungsgröße *W* an die Steuerung bzw. den Regler weitergeleitet. Diese steuern die Aktuatoren durch die Ausgangsgröße *U*, die wiederum Einfluss auf die Strecke (Stellgröße *Y*) hat. Als Rückmeldung erhält die Steuerung bzw. der Regler die Messgröße *R*, die mit der Führungsgröße *W* verglichen und abhängig vom Ergebnis dieses Vergleichs im Sinne einer Angleichung an die Führungsgröße *W* beeinflusst wird. Der sich dabei ergebende Wirkungsablauf findet in einem geschlossenen Regelkreis statt. Außer der direkten Eingabe des Fahrers, kann das Gesamtsystem zusätzlich von der Umwelt (Störgröße *Z*) beeinflusst werden, welches wiederum eine Korrektur nach sich zieht.

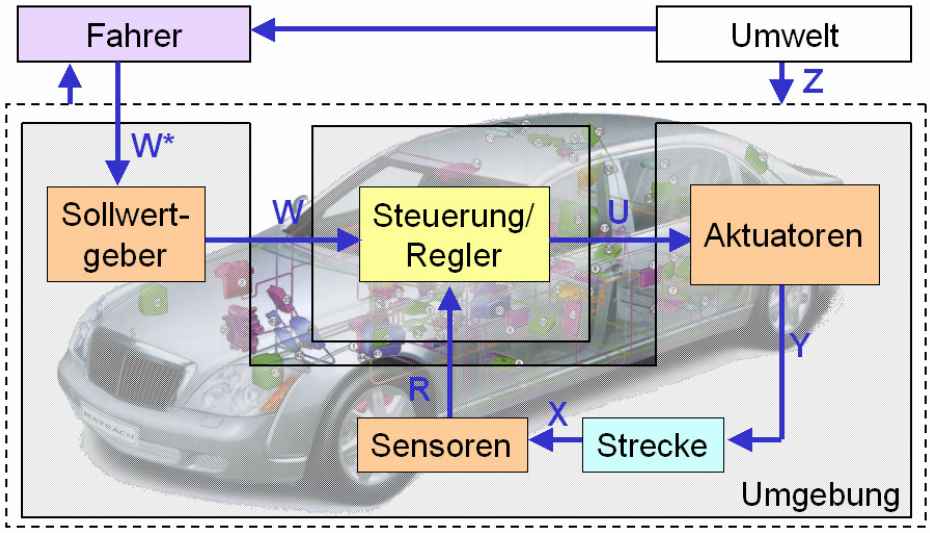


Abbildung 4-24: Das Konzept eines Regelkreises im Fahrzeug nach [SZ03]

Die in STEP-X entwickelte SW-Steuerung verfügt über zustandsbasierte Modelle und weist somit ein diskretes Verhalten [Kien97, BRS00] auf. Dem gegenüber stehen Umgebungsmodelle, die einen physikalischen Prozess abbilden, der grundsätzlich nur durch kontinuierliche Verhaltensmodelle [Unbe02] modelliert werden kann. Diese Vorgehensweise, bei der anstelle von diskreten Modellen ausschließlich kontinuierliche Modelle zur Erstellung von Umgebungsmodellen verwendet werden, findet unter anderem in [FHV04] anhand einer Fallstudie aus der Regelungstechnik Bestätigung. Da der neue UML-Standard [OMG02b, OMG03b] weiterhin keine Möglichkeit bietet regelungstechnische Prozesse zu erstellen, wird bei der STEP-X Entwurfsmethodik auf zusätzliche Beschrei-

bungssprachen zurückgegriffen.

Die bereits in Kapitel 3.2.3 vorgestellte MATLAB Toolbox ist ein Werkzeug, das in der Praxis sehr häufig zur Modellierung von regelungstechnischen Prozessen eingesetzt wird. Es verfügt über vorgefertigte Regelungsfunktionen, die in Funktionsbibliotheken zusammengefasst sind. Abbildung 4-25 stellt einen Ausschnitt der Fensterheberumgebung dar.

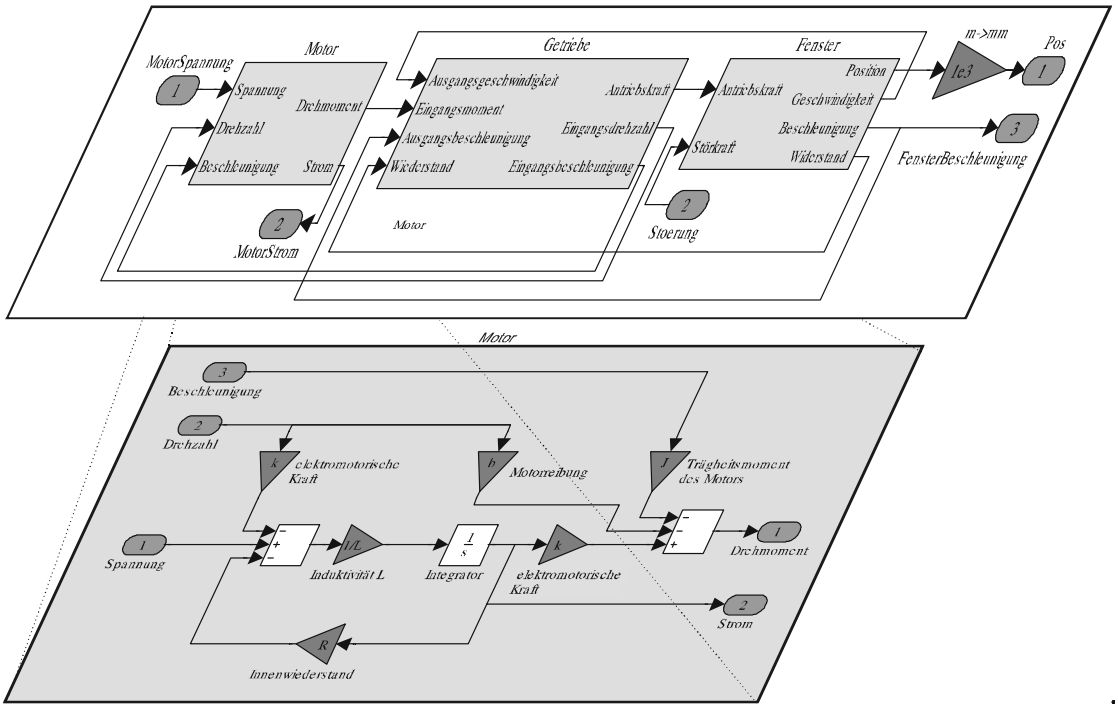


Abbildung 4-25: Ein Ausschnitt des Fensterheberumgebungsmodells

Das Umgebungsmodell ist gemäß der gegebenen physikalischen Struktur in die drei Teilsysteme *Motor*, *Getriebe* und *Fenster* gegliedert, die sich durch wohldefinierte Schnittstellen Signale sowohl untereinander als auch mit weiteren Teilsystemen austauschen. Die zunächst noch grobe Aufteilung auf der obersten Ebene ermöglicht einen schnelleren Überblick über das Gesamtsystem und definiert zudem Verantwortlichkeiten. Des Weiteren werden durch die Kopplung die Wiederverwendbarkeit und die Effektivität durch parallele Entwicklung gesteigert [CWM98]. Die Verfeinerung der einzelnen Blöcke geschieht durch Dekomposition des Systems in weitere Teilsysteme, wobei ein Teilsystem bzw. eine Subfunktion wiederum als Block dargestellt wird. Die hierarchische Struktur des Systems erhöht lediglich die Lesbarkeit, hat jedoch keine semantische Bedeutung, so dass beispielsweise keine Ausführungsprioritäten zwischen Ebenen existieren.

Im Gegensatz zu der Modellierung von Steuerungs-Software ist die Erstellung eines Umgebungsmodells in vielen Fällen einfacher. Dies liegt vor allem an der klar definierten Semantik der eingesetzten Modellierungselemente, die auf mathematischen und regelungstechnischen Grundsätzen basieren. Des Weiteren ist eine Änderung und Anpassung der Umgebungsmodelle in der Praxis nur selten vorzunehmen, da hauptsächlich der Steuerungsanteil eines eingebetteten Systems modifiziert wird, während die mechanische/ hydraulische Umgebung nahezu unverändert bleibt.

Die Validierung des Systemverhaltens erfolgt unter Berücksichtigung des Umgebungsmodells mit Hilfe einer Co-Simulation der im Modellierungsprozess verwendeten CASE Tools. Besonders vor dem Hintergrund einer schnelllebigen Tool-Landschaft ist die Werkzeugunabhängigkeit ein wichtiges Kriterium, um die Durchgängigkeit im Entwicklungsprozess auch zukünftig sicherzustellen. Um die Werkzeugunabhängigkeit zu gewährleisten bedarf es einer flexiblen Werkzeugkopplung. Zu diesem Zweck kommt in STEP-X die ExITE Toolbox der Firma Extessy AG (vgl. Kapitel 3.2.3) zum Einsatz, die eine Integration der hier verwendeten Werkzeuge ermöglicht. Darüber hinaus können weitere Werkzeuge mittels der ExITE C/C++ API angebunden werden [Schu04].

In Abbildung 4-26 ist die in STEP-X realisierte Co-Simulation auf der Grobentwurfsebene dargestellt. Dabei findet eine Kopplung zwischen einer in Rhapsody modellierten Steuerung sowie einer in MATLAB /Simulink erstellten Umgebung statt. Optional kann die Visualisierung der HMI (Human Machine Interface) durch Altia Design vorgenommen werden.

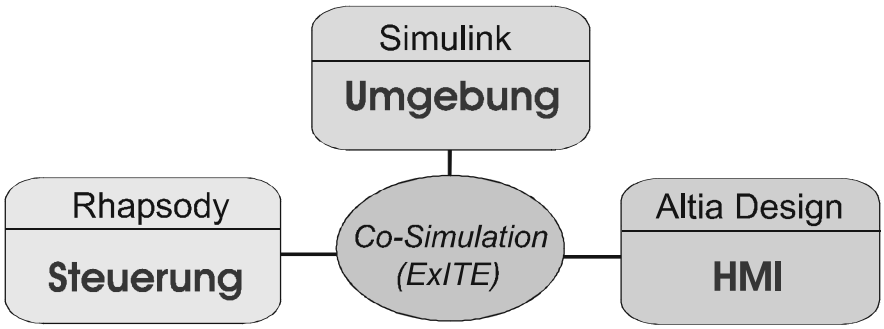


Abbildung 4-26: Tool-Kopplung der verwendeten Modellierungswerkzeuge

Als Kommunikations-Framework dient ein übergeordnetes MATLAB/Simulink-Modell, das als Master-Modell fungiert. Es hat die Aufgabe, den Signalaustausch zwischen den hybriden Modellen zu koordinieren, indem es modellübergreifende Signale zu einer zentralen Stelle, dem ExITE-Server, übermittelt. Analog zur vorherigen Abbildung besteht das Master-Modell aus den drei Blöcken *HMI*, *Steuerung* und *Umgebung*. In Abbildung 4-27 ist ein Ausschnitt des Master-Modells in Form eines Regelkreises dargestellt.

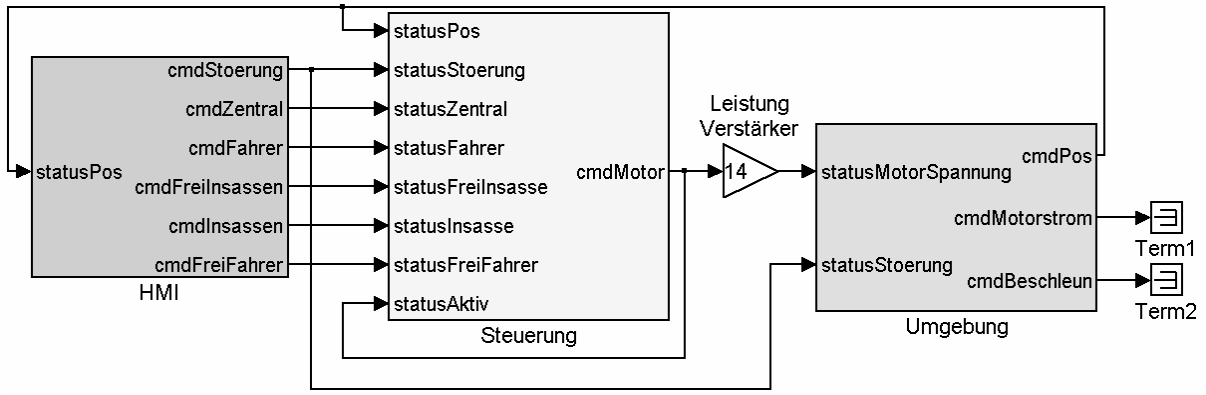


Abbildung 4-27: Master-Modell der Fensterhebersteuerung in MATLAB/Simulink

Die beiden Blöcke HMI und Steuerung repräsentieren hierbei die Modelle aus den jeweiligen Werkzeugen. Um die Co-Simulation erfolgreich ausführen zu können, muss eine An-

bindung zwischen diesen Blöcken und den dazugehörigen Referenzmodellen in deren Werkzeugen durch speziell dafür vorgesehene ExITE-Kommunikationsblöcke erfolgen. Diese werden zwischen den eingehenden und ausgehenden Signalen in den jeweiligen Blöcken integriert. Abbildung 4-28 demonstriert beispielhaft eine Anbindung im HMI-Block.

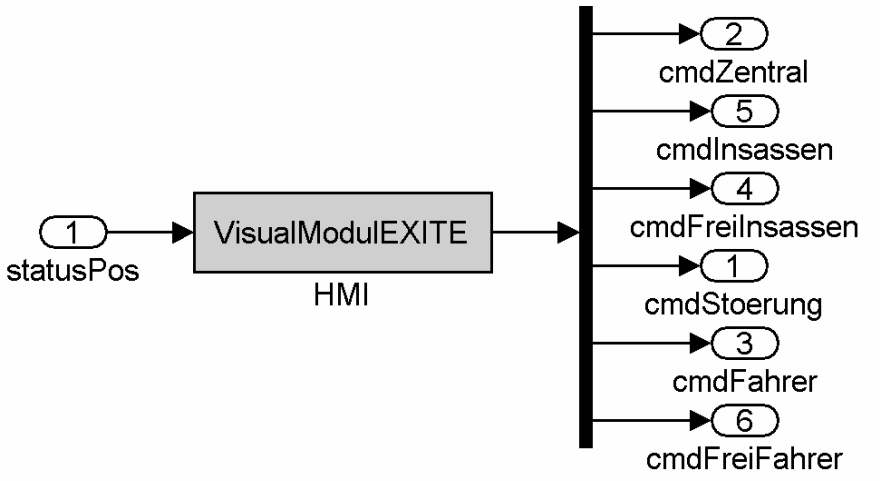


Abbildung 4-28: Die innere Struktur des HMI-Blocks

Ein derartiger ExITE-Kommunikationsblock hat die Aufgabe, Signale aus dem Master-Modell an das jeweilige Modellierungswerkzeug weiter zu leiten bzw. ankommende Signale aus den Werkzeugen an das Master-Modell zu übergeben. Dabei müssen zunächst die Signale transformiert und anschließend an den richtigen Schnittstellenports zugewiesen werden. Eine Ankopplung der Umgebung an den ExITE-Server ist nicht notwendig, da das Umgebungsmodell bereits in MATLAB/Simulink erstellt ist und somit direkt vom Master-Modell ausgeführt werden kann. Grundsätzlich ist aber eine Realisierung der Umgebung ebenso in einem anderen Werkzeug, wie z. B. ASCET-SD [ETAS00], denkbar.

Der generelle Ablauf der hier beschriebenen Co-Simulation ist wie folgt: Ein betätigter Taster in der Altia-Umgebung erzeugt ein Signal, welches von dem ExITE-Server durch den *VisualModulEXITE*-Block (vgl. Abbildung 4-28) aufgefangen und an das Steuerungsmodell in Rhapsody weitergeleitet wird. Die hierbei auftretenden Reaktionen der Steuerungs-Software werden an das Umgebungsmodell in Form von Signalen gesendet. Die Rückmeldung der Aktuatoren und eventueller Umwelteinflüsse erfolgt wiederum über den ExITE-Server an die HMI.

Zur Erreichung einer derartigen Co-Simulation müssen die Modelle der Steuerung und Visualisierung an das MATLAB-Master-Modell angepasst werden. Nachfolgend wird die grobe Vorgehensweise bei der Kopplung aufgezeigt, ohne genauer auf technische Einzelheiten einzugehen. Eine detaillierte Beschreibung befindet sich im Handbuch der Extessy AG [Schu04].

Zur Kopplung der Modelle an den ExITE-Server durch Kommunikationsblöcke steht ein Klassendiagramm zur Verfügung, das die grundsätzliche Anbindungsstruktur aufzeigt (vgl. Abbildung 4-29).

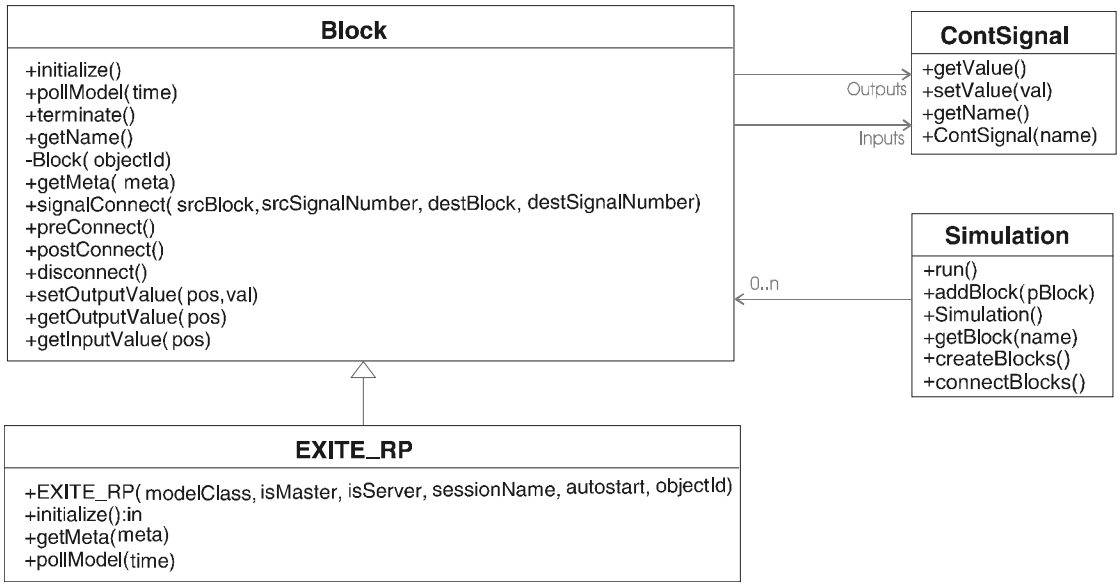


Abbildung 4-29: Das ExITE-Klassendiagramm für die Kommunikationsdefinition

Es besteht zunächst aus vier Klassen. Die Klasse *Block* beschreibt eine allgemeingültige Simulationskomponente, die als Grundbaustein für die Kopplung an die Simulation zur Verfügung steht. Die von ihr abgeleiteten Klassen (z. B. Steuerungsklassen von Rhapsody) werden initialisiert, über den Simulationsfortschritt benachrichtigt und am Ende der Simulation aus dem Speicher entfernt. Des Weiteren beinhaltet sie Schnittstellen zu den anderen kommunizierenden Klassen vom gleichen Typ. Jede Instanz dieser Klasse definiert eine eigene Abtastrate, die von der Simulation individuell berücksichtigt wird. Die hier abgeleitete Klasse *EXITE_RP* steht speziell für die Anbindung von Rhapsody-Modellen zur Verfügung.

Die Klasse *ContSignal* sorgt für die Sicherstellung der korrekten Kommunikationsverbindung zwischen den kommunizierenden Block-Klassen. Dabei muss ein Eingangssignal immer mit einem Ausgangssignal verbunden werden, während ein Ausgangssignal keine Verbindung benötigt.

Die Überwachung und Steuerung des Simulationsvorgangs erfolgt durch die Klasse *Simulation*. Sie ist dafür verantwortlich die Reihenfolge der Abarbeitung der Blöcke festzulegen und enthält daher auch die aktuelle Simulationszeit. Sie erstellt die notwendigen Instanzen der Blöcke, fügt diese in die Simulation ein und führt anschließend den Simulationsvorgang durch.

Mit Hilfe der ExITE-Applikation ist es ebenso möglich, die Steuerungsmodelle in eine reale HW-Umgebung einzubetten, um somit eine Möglichkeit zu schaffen, eine HIL-Simulation durchzuführen. Hierfür wurden in STEP-X durch die AG Diagnose und die AG Testen zwei unterschiedliche HW-Demonstratoren [BHHM03, MHHH+03] entwickelt, die in der Abbildung 4-30 zu sehen sind.

Bei dem linken HW-Demonstrator handelt es sich um eine Fahrzeugtür eines VW Golfs, die über einen elektrischen Spiegel und einen elektrischen Fensterheber verfügt.

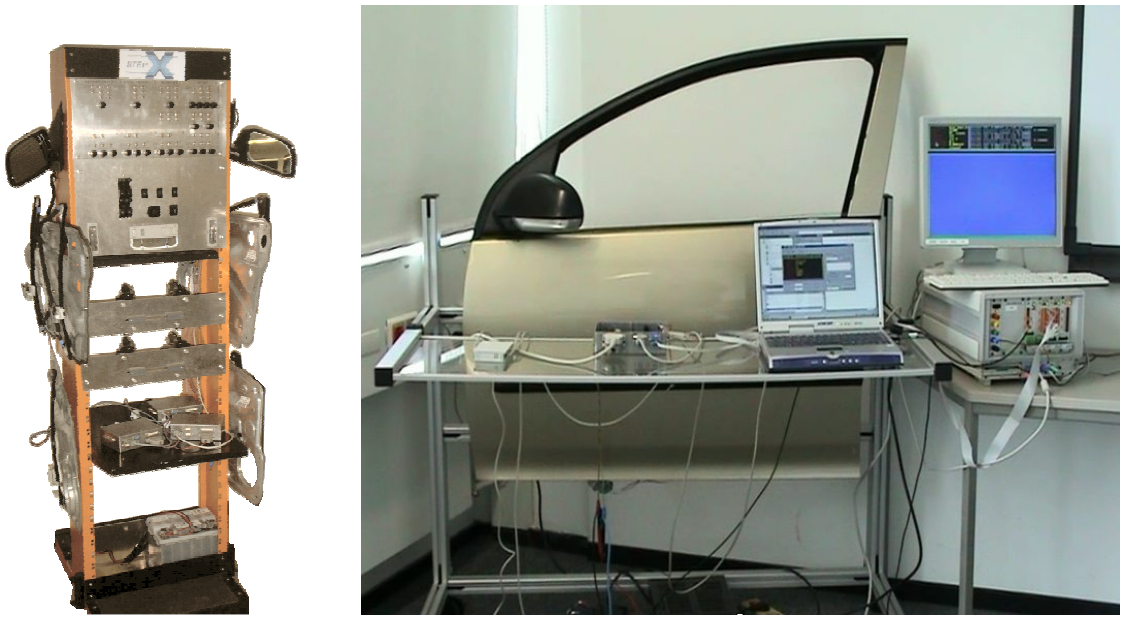


Abbildung 4-30: Die verschiedenen Ausprägungen einer HW-Umgebung

Der rechte HW-Demonstrator repräsentiert zwei elektrische Spiegel, eine Zentralverriegelung und vier elektrische Fensterheber, die allerdings keine Fensterscheiben bewegen, sondern nur die Fensterführung ansteuern, d. h. im Gegensatz zu dem linken HW-Demonstrator wird in dem Rechten das Zusammenspiel der verteilten HW-Komponenten überprüft.

Ein weiterer Vorteil des gemeinsamen Einsatzes von HW-Demonstratoren und dem ExI-TE-Server ist die Möglichkeit des flexiblen Austauschs von Hard- und Software-Komponenten im Gesamtsystem. So kann beispielsweise nicht nur das Umgebungsmodell ausgetauscht werden, sondern es besteht zusätzlich die Möglichkeit, den HW-Demonstrator auch als Eingabebedienfeld zu nutzen, um somit die SW-Steuerung von außen zu bedienen. Daher stellt ein HW-Demonstrator in STEP-X (vgl. Abbildung 4-30 auf Seite 106) sowohl eine reale Umgebung als auch eine HMI dar.

4.3.5 Funktionsverteilung auf ein logisches Steuergerätenetzwerk

Die bisherigen Tätigkeiten im funktionalen Grobentwurf beziehen sich vorwiegend auf die Erstellung und Validierung der Funktions- und Umgebungsmodelle. Um die modellierte Steuergeräte-Software erfolgreich in die Fahrzeugumgebung integrieren zu können, bedarf es einer grafischen Abbildung der physikalischen Fahrzeugarchitektur. Dabei wird zunächst von einer groben Granularität dieser HW-Architektur ausgegangen, die lediglich die Anzahl der Steuergeräte, der Sensoren und der Aktuatoren sowie die Verbindung zwischen diesen aufzeigt.

Auf der Basis eines dezentralen Komfortelektroniksystems, wie in [BBHS96] beschrieben, besteht die HW-Architektur eines 4-türigen Volkswagen-Fahrzeugs demzufolge aus einem Zentralmodul und vier Türsteuergeräten. Während in den Türsteuergeräten die lokalen Funktionen integriert sind, beinhaltet das Zentralmodul türsteuergeräteübergreifende

Funktionen. Als Beispiel seien hier die Zentralverriegelung, Komfortfunktionen, Innenlichtsteuerung und Heckdeckelansteuerung genannt. Im Allgemeinen sind die Steuergeräte direkt mit der Sensorik und der anzusteuernden Aktuatorik verbunden. Die Kommunikation zwischen den Steuergeräten in der Domäne Komfortelektronik verläuft dagegen über einen *Low Spee CAN*-Bus.

In Abbildung 4-31 ist die soeben beschriebene Fahrzeugarchitektur mit Hilfe eines UML-Verteilungsdiagramms dargestellt. Dabei werden die Steuergeräte durch die großen Knoten und die Sensoren, Eingabeelemente sowie Aktuatoren durch die kleineren Knoten repräsentiert.

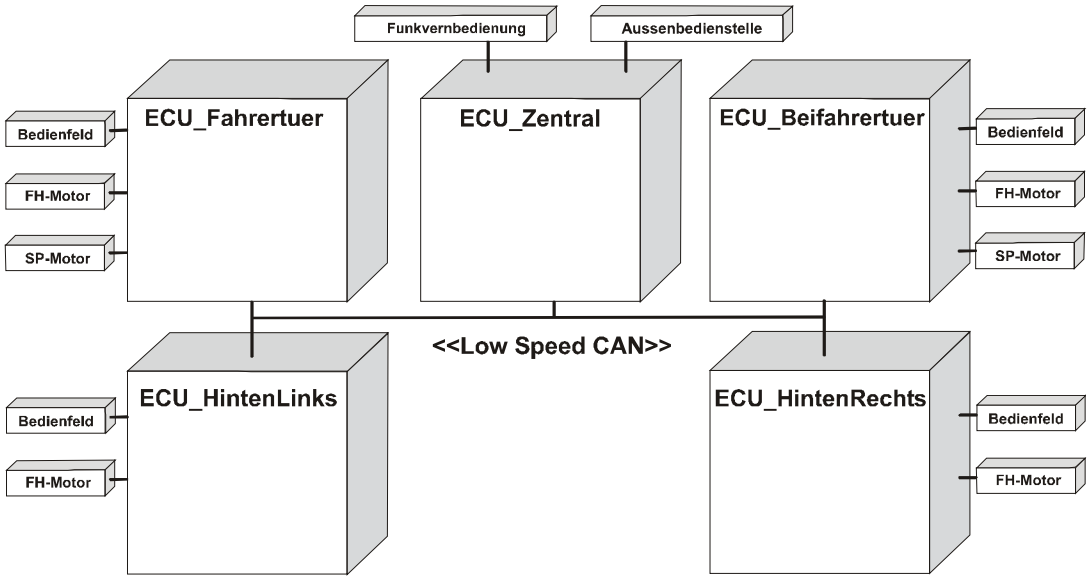


Abbildung 4-31: Die logische Fahrzeugtopologie für das Kfz-Komfortsystem

Die Anordnung der physikalischen Bauteile dient in STEP-X einerseits zur frühzeitigen Erkennung des geplanten Steuergerätenetzwerks, andererseits als Plattform für die Verteilung der Funktionen auf die logischen Steuergeräte. Zur Durchführung der Funktionsverteilung werden die SW-Komponenten und Koordinatorklassen aus der SW-Struktur der Abbildung 4-16 auf Seite 92 den Knoten zugewiesen. Die Bedienelemente und Aktuatoren, die zuvor als eigenständige Pakete bzw. Klassen fungierten, werden in der Fahrzeugarchitektur als eigenständige Knoten abgebildet.

Obwohl die Funktionsverteilung grundsätzlich keiner Beschränkung unterliegt, ist die physikalische Anordnung der Bauteile zu berücksichtigen. So sollten beispielsweise die Funktionen so auf den Steuergeräten platziert werden, dass sie möglichst direkt die dazugehörigen Aktuatoren ansteuern können. Dadurch wird ein überhöhter Signaltransfer auf dem CAN-Bus vermieden. Eine mögliche Verteilung der SW-Komponenten und Koordinatorklassen ist in dem unteren Verteilungsdiagramm dargestellt, wobei aus Platzgründen nur die Steuergeräte abgebildet sind.

Alle vier Türsteuergeräte verfügen über eine Fensterhebersteuerung, die als ein eigenständiges SW-System dargestellt ist. Darüber hinaus ist in der Fahrertür und der Beifahrertür die Spiegelfunktionalität integriert. Um evtl. Konflikte zwischen den beiden SW-

Systemen zu vermeiden, erhalten die beiden Steuergeräte jeweils einen Türkoordinator. Das zentrale Steuergerät umfasst sowohl das Subsystem Zentralverriegelung als auch zwei Koordinatorklassen. Mit Hilfe des Koordinators *kooZentral* werden die Signale von der Funkfernbedienung aufgenommen und an die betreffenden Türsteuergeräte weitergeleitet. Die Koordinatorklasse *kooKomfortLauf* nimmt Signale von den Außenbedienstellen entgegen und synchronisiert den Fensterlauf während des Komfortöffnens bzw. Komfortschließens.

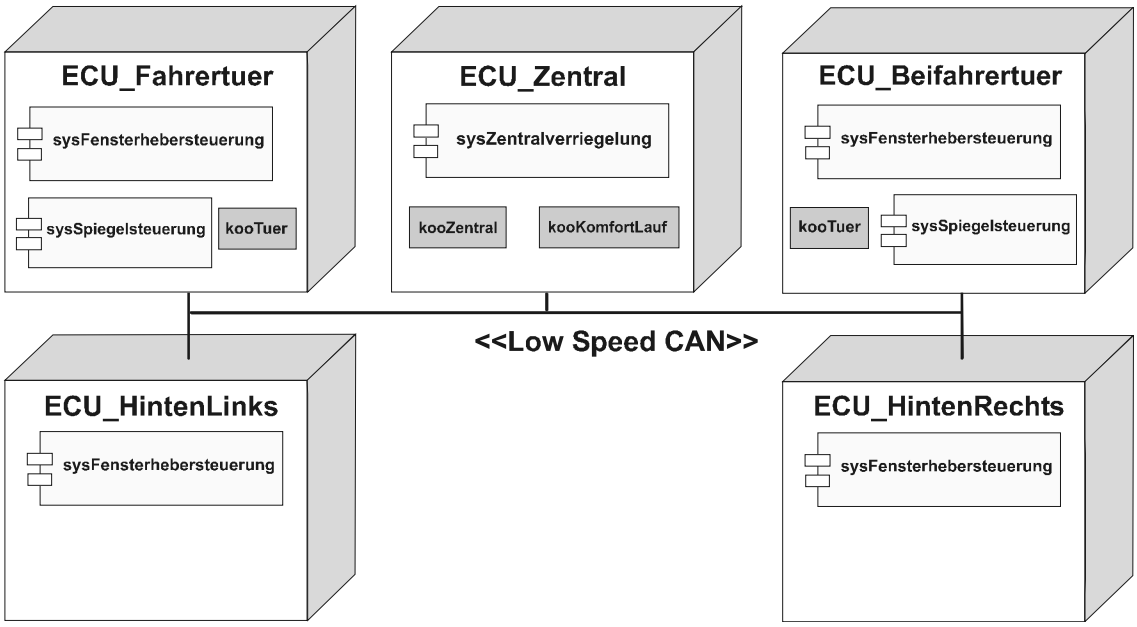


Abbildung 4-32: Komponentenverteilung auf logische Steuergeräte

Die Granularität der Darstellung in Abbildung 4-32 ist für das grobe Verständnis der Funktionsverteilung zunächst ausreichend. Ist ein höherer Detaillierungsgrad der verteilten Funktionen gewünscht, so werden die einzelnen Knoten in separate Diagramme unterteilt, um somit die Übersichtlichkeit zu erhöhen.

Mit der hier beschriebenen logischen HW-Architektur und den darauf verteilten SW-Komponenten haben die architekturverantwortlichen Entwickler einen Anhaltspunkt, auf welchen Steuergeräten die Funktionen zu integrieren sind. Zu bemerken ist allerdings, dass die vorgegebene Funktionsverteilung lediglich als Vorschlag zu interpretieren ist, da aufgrund der gegenwärtig fehlenden Möglichkeit in den UML-Werkzeugen die Auswirkung der Funktionsverteilung nicht analysierbar ist. Das bedeutet, dass hardwarespezifische Aussagen über die Speicherbelegung auf den einzelnen Steuergeräten oder über den Signaltransfer auf dem Kommunikationsbus nicht getroffen werden können. Eine Verschiebung der SW-Komponenten ist daher im späteren Entwicklungsverlauf nicht auszuschließen.

Nach der Durchführung aller notwendigen Aktivitäten im funktionalen Grobentwurf ist das Digitale Lastenheft fertiggestellt. Die darauf basierende Weiterentwicklung des SW-Systems sowie die anschließende Integration in das Gesamtsystem erfolgen in den meisten Fällen durch Zulieferfirmen. Die Vorgabe einer methodischen Vorgehensweise aus Sicht

der OEMs an die Zulieferer kann nicht ernsthaft betrieben und realisiert werden. Um die Durchgängigkeit des STEP-X Entwicklungsprozesses trotzdem einmal aufzuzeigen, wird die Entwicklung des SW-Systems beispielhaft im anschließenden funktionalen Feinentwurf fortgeführt.

4.4 Feinentwurf

Wie im vorherigen Abschnitt bereits erwähnt, erfolgt die anschließende Implementierung und Integration der Steuergeräte-Software derzeit meist durch Zulieferer. Da jeder Lieferant seinen eigenen Entwicklungsprozess definiert, der durch spezielle Methoden und Werkzeuge unterstützt wird, ist eine einheitliche, firmenübergreifende Vorgehensweise auf die vielen Lieferanten nicht übertragbar. Daher liegt die Verantwortung in der Umsetzung der Anforderungen allein bei den Fremdfirmen. Durch die Verlinkungen zwischen Lasten- und Pflichtenheften sowie den zu erstellenden Produkten wird die Nachvollziehbarkeit während des gesamten Entwicklungsprozesses gewährleistet. Somit besteht die Möglichkeit, die Ergebnisse der Zulieferer jederzeit zu validieren und außerdem auf Anforderungsänderungen seitens der OEMs frühzeitig zu reagieren.

Um die Durchgängigkeit des STEP-X Entwicklungsprozesses von der Anforderungserfassung bis zur Implementierung trotz dessen aufzeigen zu können, erfolgt an dieser Stelle eine beispielhafte Weiterentwicklung des SW-Systems aus Sicht der Zulieferer. Die hierfür notwendigen Aktivitäten unterscheiden sich von dem in [DW00] beschriebenen funktionalen Feinentwurf dadurch, dass der Schwerpunkt dieser Aktivitäten nicht auf der alleinigen Beschreibung der SW-Komponenten (vgl. Tabelle 3-1 auf Seite 52) liegt. Vielmehr ist darunter eine Kombination aus den Hauptaktivitäten Feinentwurf (SE 5), Implementierung (SE 6) und SW-Integration (SE 7) zu verstehen. Um die starke Verzahnung zwischen den arbeitsgruppenübergreifenden Aktivitäten im STEP-X Projekt verständlich repräsentieren zu können, werden diese zu einer Phase zusammengefasst.

Die Hauptaufgabe des funktionalen Feinentwurfs in STEP-X ist die Modellierung einer fahrzeugspezifischen Architektur mit den darauf befindlichen hardwarenahen Funktionen. Zur Durchführung eines derartigen Modellierungsprozesses müssen weitere Aspekte, wie die Definition der Buskommunikation, Anbindung des Betriebssystems, Erstellung steuergerätespezifischer Schnittstellen und Partitionierungskriterien herangezogen werden. Die UML-basierten Werkzeuge können diese hardwarespezifischen Aspekte nicht ausreichend bewerkstelligen. Daher findet an dieser Stelle des Entwicklungsprozesses ein Werkzeugwechsel statt. Während die Funktionsmodellierung mit Hilfe der MATLAB Toolbox vorgenommen wird, erfolgen die Architekturerstellung, CAN-Nachrichtendefinition und Funktionsverteilung durch das Werkzeug DaVinci (vgl. Kapitel 3.2.3).

Abbildung 4-33 skizziert die vierstufige Vorgehensweise des STEP-X Feinentwurfs zur Erstellung der fahrzeugspezifischen Architektur und Funktionen.

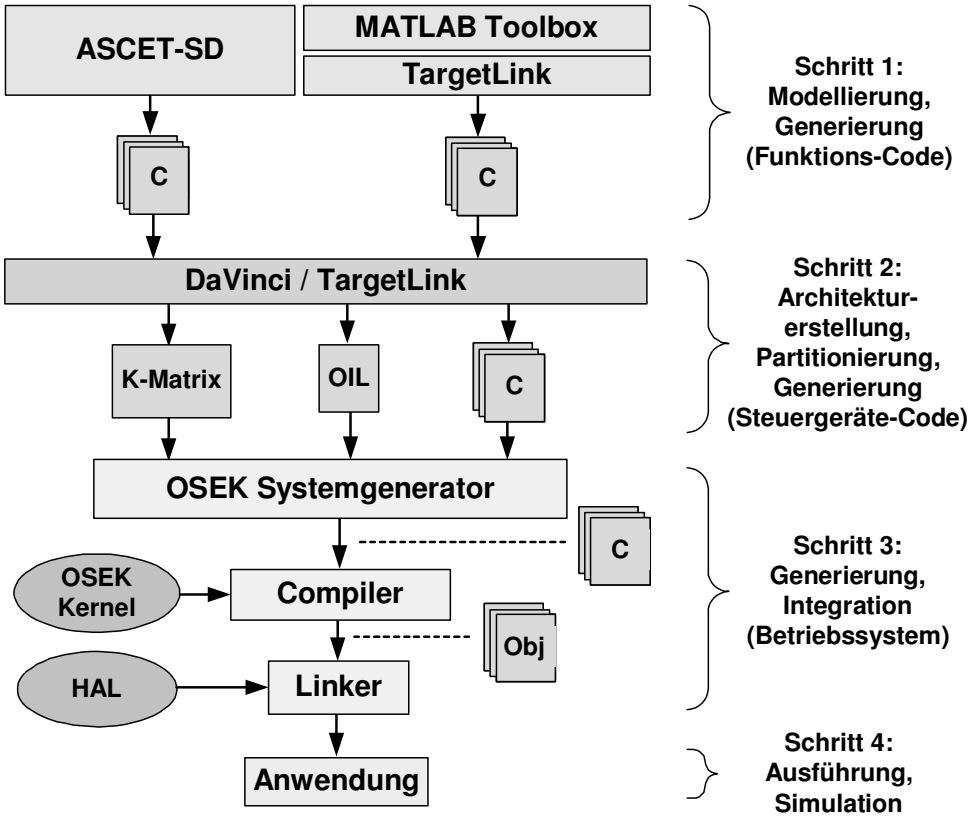


Abbildung 4-33: Die Vorgehensweise im Feinentwurf

Als erstes erfolgt ein Reengineering der Funktionsmodelle aus dem Grobentwurf, wobei fahrzeugspezifische Informationen in die Modellierung mit einfließen. Die aus der anschließenden Code-Generierung erzeugten Dateien werden an DaVinci übergeben. Anhand der Partitionierungsvorgaben wird zunächst ein logisches Steuergerätenetzwerk entworfen, das als Ausgangspunkt für die Funktionsverteilung dient. Des Weiteren werden Busnachrichten definiert und Systemroutinen eingebunden. Auf Basis dieser Informationen werden im nächsten Schritt durch den Systemgenerator die notwendigen Betriebssystemaufrufe generiert und die Kommunikationspfade optimiert. Abschließend kann der auf den Steuergeräten befindliche Code ausgeführt und simuliert werden.

Die hier kurz vorgestellte Vorgehensweise wird in den folgenden Abschnitten näher erläutert.

4.4.1 Entwurf fahrzeugspezifischer Funktionen

Der erste Schritt im funktionalen Feinentwurf ist die Erstellung detaillierter Funktionsmodelle, die auf einer fahrzeugspezifischen Architektur verteilt und ausgeführt werden können. Um dieses zu realisieren, reicht eine implementierungsunabhängige Beschreibung der Funktionalität nicht mehr aus. Vielmehr müssen hardwarenahe Informationen über die verwendeten Steuergeräte, den Kommunikationsbus und das Betriebssystem in das Gesamtmodell mit einbezogen werden. Die zu diesem Zweck notwendige Anbindung von Funktionen an Bauteile erfordert eine implementierungsnahe Werkzeuglösung, die mit den gängigen UML-Werkzeugen nicht realisierbar ist. Die Gründe hierfür liegen vor allem

in der für die Serienentwicklung unzureichenden Code-Qualität. Einerseits ist der Umfang des generierten Codes für die gängigen Mikrocontroller zu groß, andererseits haben objektorientierte Modelle nach [Romb00] den Nachteil, dass durch die höhere Anzahl der Operationsaufrufe pro Modellschritt die Rechenzeit ansteigt. Daher wird in STEP-X ausschließlich auf die beiden Werkzeuge MATLAB-Suite und ASCET-SD zurückgegriffen, die einen optimierten Code für eine Vielzahl unterschiedlicher Steuergeräte generieren können.

Die Nutzung derartiger Modellierungswerkzeuge erfordert allerdings auch einen Wechsel von objektorientierten Strukturen hin zu signalflussorientierten hierarchischen Blockdiagrammen. Dieser Vorgang ist zurzeit nicht automatisiert und kann daher nur manuell erfolgen. In Abbildung 4-34 ist eine Spiegelsteuerung in MATLAB/Simulink dargestellt, die der SW-Komponente *sysSpiegelsteuerung* aus dem Grobentwurf (vgl. Abbildung 4-16 auf Seite 92) entspricht. Auf der obersten Ebene besteht sie aus drei Blöcken, die jeweils eine eigenständige Funktionalität darstellen. Durch den hierarchischen Aufbau bestehen die Funktionsblöcke aus weiteren Subfunktionen, die entweder durch kontinuierliche oder durch diskrete Modellkonstrukte abgebildet werden. Auf den Aufbau und die Funktionsweise der einzelnen Blöcke soll an dieser Stelle jedoch nicht näher eingegangen werden. Viel mehr sollen durch die signalflussorientierte Darstellungsform die Ein- und Ausgänge der Funktionsblöcke und deren Verbindungen sowohl untereinander als auch nach außen verdeutlicht werden.

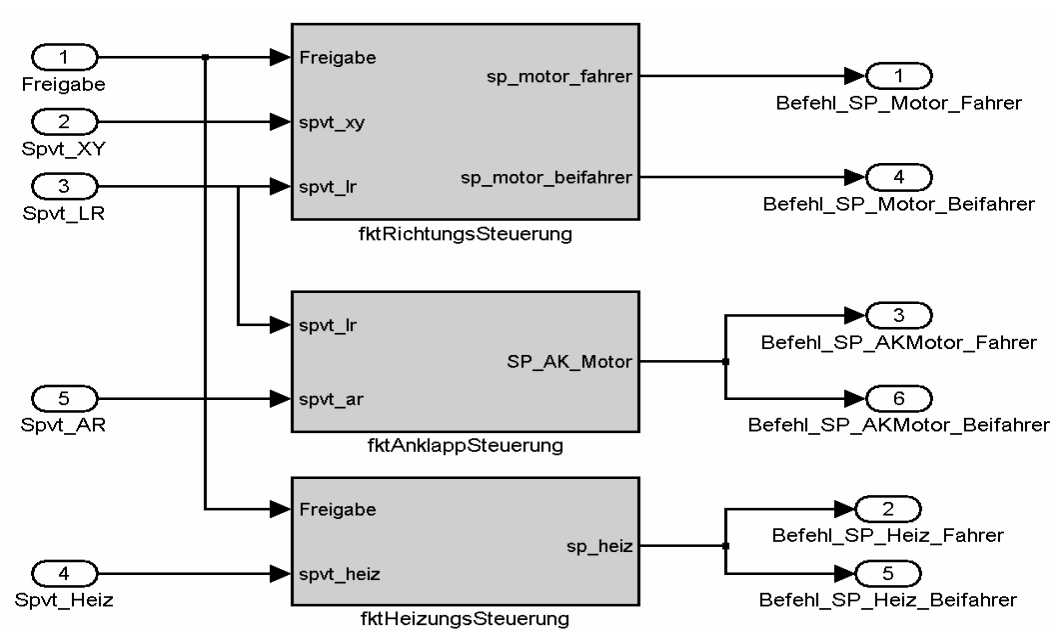


Abbildung 4-34: Spiegelsteuerung in MATLAB/Simulink

Die Nutzung einer Funktionsbibliothek mit wiederverwendbaren SW-Komponenten ermöglicht die Erhöhung der Effektivität bei der Modellierung der Systeme. Allerdings ist die Verwendung eines derartigen zentralen Repositories erst dann sinnvoll, wenn diesem darüber hinaus Sicherheits-, Versions- und Konfigurationsmechanismen zur Verfügung stehen, mit denen die zahlreichen Funktionsvarianten verwaltet werden können [BBK98].

Da bei den meisten OEMs keine Funktionssammlungen mit zentraler Datenhaltung und intelligenten Verwaltungsmechanismen vorliegenden, gibt es derzeit diverse Bestrebungen, diese Problematik durch methodische Ansätze und Werkzeugunterstützung zu beheben. Beispielsweise werden in den Arbeiten von [WM99, Scha05, SB05] mögliche Vorgehen propagiert, die Funktionskomponenten so zu strukturieren und abzulegen, dass sie eine möglichst hohe Wiederverwendbarkeit und Flexibilität erlangen.

Nachdem die Funktionen mit ihren Schnittstellen spezifiziert bzw. aus der Funktionsbibliothek ausgewählt wurden, erfolgt als nächstes die Code-Generierung aus den Modellen mit Hilfe des Code-Generators TargetLink. Die Code-Generierung ist an dieser Stelle aus zwei Gründen notwendig. Einerseits werden dadurch die Funktionen gekapselt und stehen jeder neueren SW-Entwicklung als wiederverwendbarer *legacy code* zur Verfügung. Andererseits ist oftmals die Weitergabe der Funktionen an weitere Werkzeuge nur in Form eines Quelltextes möglich. Dies wird beispielsweise im Falle der anschließenden Funktionsverteilung ersichtlich, in der die autonomen Funktionen nur als Code-Fragmente an das Architekturwerkzeug DaVinci übergeben werden können.

4.4.2 Funktionspartitionierung auf logische Fahrzeugarchitekturen

Nach der Spezifizierung der hardwarenahen Funktionen, erfolgt im zweiten Schritt des Feinentwurfs (vgl. Abbildung 4-33 auf Seite 110) die Funktionspartitionierung. Unter Partitionierung wird die Abbildung des SW-Systems auf die zugehörige logische Fahrzeugtopologie verstanden. Der Begriff Mapping ist im Rahmen dieser Arbeit synonym zu verstehen.

Um den Partitionierungsprozess erfolgreich durchführen zu können, wird die Erstellung einer fahrzeugspezifischen Topologie vorausgesetzt. Erst dann kann mit der eigentlichen Funktionsverteilung auf die logischen Steuergeräte begonnen werden. Zur Durchführung des Partitionierungsvorgangs hat sich DaVinci als geeignetes Werkzeug herausgestellt, weil damit sowohl die Erstellung der Architektur als auch die Verteilung der Funktionen durchgeführt werden kann. Das Resultat bildet eine Fahrzeugarchitektur, bestehend aus einem logischen Steuergerätenetzwerk und den dazugehörigen Funktionseinheiten.

Der hier kurz dargestellte Verlauf ist in der unteren Abbildung skizziert und wird nachfolgend näher beschrieben.

Die Funktionspartitionierung beginnt zunächst mit der Nachbildung und Verfeinerung der sehr abstrakten HW-Architektur aus dem Grobentwurf (vgl. Abbildung 4-31 auf Seite 107). Dazu wird die Fahrzeugtopologie spezifiziert, die aus einer Vielzahl von Sensoren, Aktuatoren, Steuergeräten und dem Kommunikationsbus besteht. Der Detaillierungsgrad der HW-Komponenten ist allerdings noch abstrakt gehalten, da für die Modellierung der logischen Bauteile zunächst die Definition der Ein- und Ausgänge genügt. Hardwarespezifische Eigenschaften, wie Speicherkapazität oder Busbreite werden nicht betrachtet, da ein derartiger Detaillierungsgrad vorwiegend für hardwarenahe Analysen notwendig ist. Die in STEP-X vorgenommene Fahrzeugtopologie ist [MHHH+03, VHHH+04] zu entnehmen.

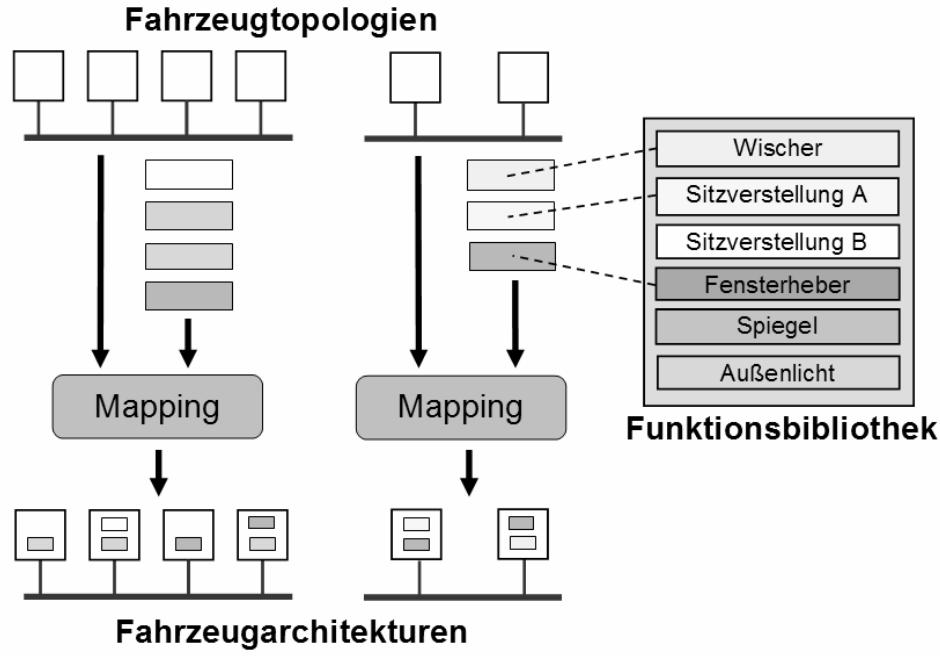


Abbildung 4-35: Funktionsorientierte Partitionierung

Für den anschließenden Partitionierungsprozess muss zunächst die Struktur der MATLAB/Simulink-Modelle in DaVinci manuell abgebildet werden. Dabei verwendet das Architekturwerkzeug genauso wie MATLAB/Simulink Funktionsblöcke zur Beschreibung von eigenständigen Funktionen. Zur Erreichung einer besseren Wiedererkennung erhalten die Ein- und Ausgänge in beiden Werkzeugen dieselbe Beschriftung. Als Beispiel soll das Spiegelmodell aus der Abbildung 4-34 dienen, bei dem die einzelnen Funktionsblöcke zu einer DaVinci-Funktionskomponente zusammengefasst werden. Das Ergebnis ist in Abbildung 4-36 zu sehen.

Anzumerken ist, dass bei unvorhersehbaren Änderungen im Feinentwurf, die eine andere Zusammenlegung der Funktionskomponenten erfordern, eine Inkonsistenz zwischen den Modellen in unterschiedlichen Phasen ausgeschlossen werden kann. Intelligente Impact-Analysen aus DOORS ermöglichen auftretende Änderungen sowohl im Lastenheft als auch in den Modellen zu analysieren und den Benutzer zu informieren. Somit wird eine konsistente Datenhaltung zwischen Grob- und Feinentwurf gewährleistet.

Es folgt die Partitionierung der DaVinci-Funktionskomponenten auf die zuvor definierte logische Fahrzeugtopologie. Dabei findet eine Zuordnung zwischen den Schnittstellen der Funktionen und der logischen Bauteile statt. Aus diesem Verteilungsszenario resultiert eine Kommunikationsmatrix, die für alle Steuergeräte bindend ist. Für jede ECU werden daraus entsprechende Signale erzeugt, um einen Nachrichtenaustausch über den Bus zu ermöglichen. Darüber hinaus kann der Anwender individuelle Botschaften definieren und den Bussignalen zuordnen.

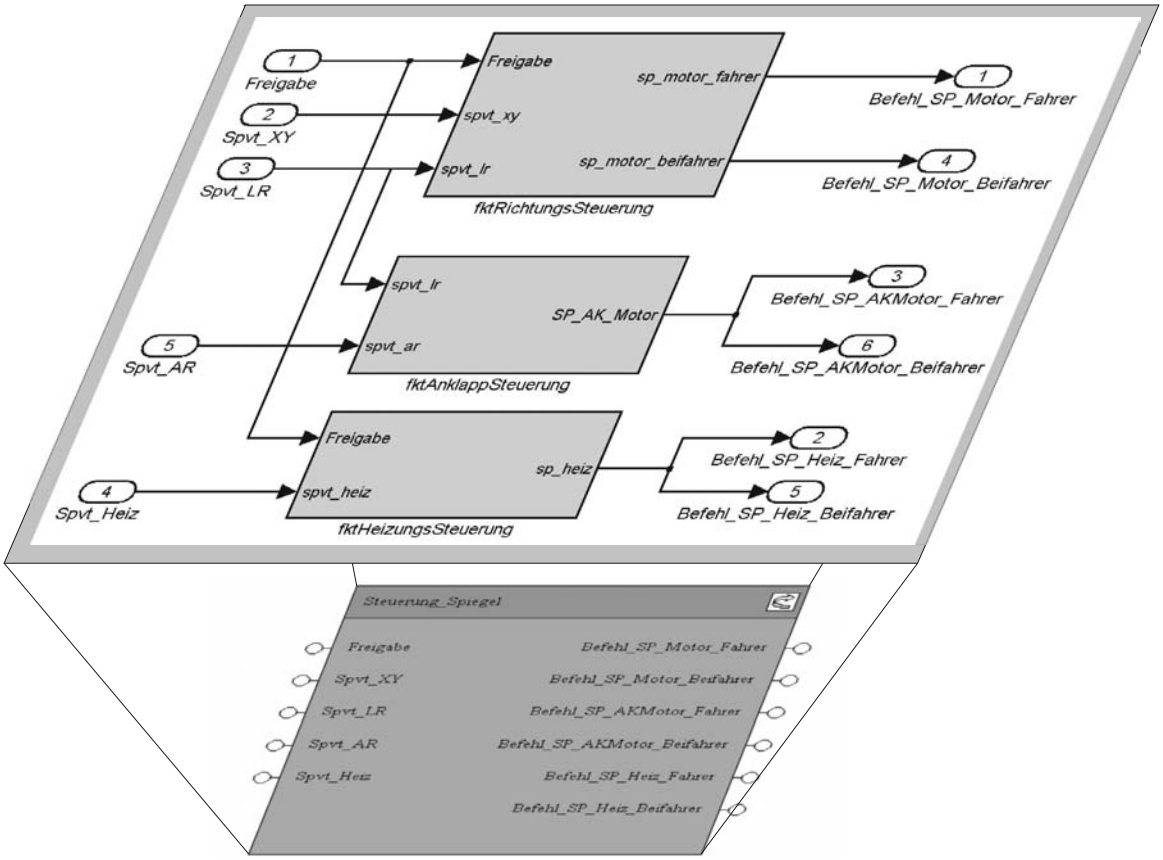


Abbildung 4-36: Spiegelsteuerung als DaVinci-Funktionskomponente

Um den Partitionierungsprozess gänzlich von der Software zu trennen, besteht die Möglichkeit aus der erzeugten Kommunikationsstruktur OIL-Daten (OSEK Implementation Language) mit Hilfe des Code-Generators zu generieren und in DaVinci zu integrieren (vgl. Abbildung 4-33). Mit Hilfe dieser standardisierten Sprache, zur Spezifizierung von Betriebssystemroutinen wird die notwendige Kommunikationsstruktur für das zugrundeliegende OSEK Betriebssystem (vgl. nächsten Abschnitt) definiert. Um eine Kommunikation zwischen SW-Komponenten herstellen zu können, ist zusätzlich eine Definition und Zuordnung der Tasks zu Funktionskomponenten erforderlich. Die Zuordnung kann entweder nach einem Standardprinzip automatisch oder vom Anwender manuell erfolgen. Üblicherweise werden alle periodischen Subsysteme gleicher Abtastrate als auch getriggerte Subsysteme mit derselben Triggerquelle vom Code-Generator zusammengefasst. Anschließend erfolgt die Umsetzung der Datenverbindungen zwischen den Tasks. Dabei muss berücksichtigt werden, dass sich Tasks gegenseitig unterbrechen können und demzufolge Datenzugriffe vor ungewünschten Interrupts geschützt werden müssen. Daher werden die Tasks mit Attributen, wie Priorität, Unterbrechbarkeit, Ressourcenbelegung, Ereigniszuordnung etc. versehen, um einen sicheren Prozessablauf zu gewährleisten. Die Abarbeitung der Tasks kann anschließend entweder vom Betriebssystem automatisch zu definierten Zeiten oder durch asynchrone Ereignisse erfolgen. Aus den Tasks wird mit Hilfe eines externen Code-Generators ein System-Code erzeugt, da im Gegensatz zu ASCET-SD in MATLAB/Simulink/Stateflow keine Tasks vorgesehen sind.

4.4.3 Generierung und Integration von Betriebssystemroutinen

Im dritten Schritt des Feinentwurfs werden die vom Code-Generator erzeugten, aber noch einzeln vorliegenden Systeminformationen (Quelltexte, Kommunikationsmatrix, OIL-Daten), zu einem Gesamtsystem zusammengefasst. Die Kopplung erfolgt durch einen Systemgenerator, der anhand von Konfigurationsdaten wie Timer, Nachrichten, Prioritäten, Interrupts usw. für den Betrieb notwendige Systemroutinen generiert und mit applikationsspezifischen Werten initialisiert. Anschließend werden die Betriebssystemroutinen zusammen mit dem Anwendungs-Code kompiliert (vgl. Abbildung 4-33).

Darüber hinaus sind Betriebssystemroutinen für das Task Scheduling, die Inter-Task-Kommunikation sowie das Ressourcen-Management erforderlich [Brau97]. Derartige Systemroutinen sind applikationsunabhängig und werden im OSEK-Kernel zusammengefasst. Die OSEK-Spezifikation [OSEK01] ist ein Standard für Betriebssysteme im Embedded-Bereich. Damit wurden unter anderem das *Application Programming Interface* (API) und der eigentliche Betriebssystemkern vereinheitlicht [KTSC01].

Um die Hardware in dem Steuergerätenetzwerk ansteuern zu können, benötigt man zusätzlich zum OSEK-Betriebsystem spezielle Interrupt Service Routinen sowie Gerätetreiber. Diese meistens in C und Assembler geschriebenen Funktionen sind sehr hardwarelastig und werden daher vom Betriebssystemkern getrennt. Die Verwaltung derartiger Systemroutinen wird von dem sogenannten Hardware Abstraction Layer (HAL) übernommen. Abschließend werden alle generierten Quelltextdateien in ausführbaren Maschinen-Code übersetzt und auf die Steuergeräte geflasht.

Im letzten Schritt des Feinentwurfs besteht die Möglichkeit einen Modul- bzw. Systemtest durchzuführen, indem das Zusammenspiel mit bereits existierenden Steuergeräten anhand des STEP-X Demonstrators validiert wird (vgl. Abbildung 4-30). Engpässe, die sich durch hardwareschwache Ressourcen ergeben, können somit festgestellt und möglicherweise durch Funktionsverlagerung auf andere Steuergeräte behoben werden. Durch eine derartige Optimierung des Verteilungsszenarios lässt sich eine Reduzierung der benötigten Botschaften und damit eine Verringerung der Buslast sowie eine verbesserte Ausnutzung der Steuergeräte-Ressourcen erzielen.

Mit Abschluss des Feinentwurfs ist die gesamte Konstruktionsphase des V-Modells durchgängig beschrieben. Auf die Phasen SW-Integration (SE 7), Systemintegration (SE 8) und Überleitung in die Nutzung (SE 9), die durch den rechten Ast des V-Modells abgebildet werden, wird in dieser Arbeit nicht näher eingegangen. Die Ergebnisse hierzu sind [HEVB03, HHK04] zu entnehmen.

Kapitel 5

5 Maßnahmen zur Sicherstellung der Software-Qualität

„Ohne natürliche Einsicht sind Regeln und Richtlinien wertlos.“

Quintillian, 30-96 n. Chr.

Die Sicherstellung der Qualität der entwickelten Steuergeräte-Software und der aus dem Entwicklungsprozess resultierenden Artefakte ist ein wesentlicher Bestandteil des V-Modells (vgl. Abbildung 3-1 auf Seite 47). Dabei ist der Begriff Qualität nach DIN 55350-11 wie folgt definiert:

Qualität

Qualität ist die Gesamtheit von Eigenschaften und Merkmalen eines Produktes oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung gegebener Erfordernisse bezieht.

Um der Forderung nach einer qualitativ hochwertigen Steuergeräte-Software gerecht zu werden, reicht die bisherige methodische Vorgehensweise mit ihren vordefinierten Templates, Simulationsmöglichkeiten und der Nachverfolgbarkeit von Anforderungen allein nicht aus. Vielmehr muss die Möglichkeit bestehen, sowohl allgemeingültige als auch projektspezifische Modellierungsrichtlinien zu definieren und diese den Entwicklern in Form eines Regelkatalogs zur Verfügung zu stellen. Des Weiteren muss eine automatische Überprüfung der Modelle auf Einhaltung dieser Richtlinien während des gesamten Entwicklungsprozesses ermöglicht werden.

Zur Erreichung dieser Ziele wird in STEP-X auf der einen Seite ein umfangreiches Regelwerk bereitgestellt, das vorwiegend für die zustandsbasierte Verhaltensmodellierung gilt. Auf der anderen Seite erfolgt mit Hilfe eines hierfür speziell entwickelten Analysewerkzeugs eine Überprüfung der Modelle hinsichtlich Inkonsistenz, Überspezifikation, Inkompatibilität und weiterer benutzerdefinierter Kriterien [Mutz04a] sowie eine Modellbewertung mittels SW-Metriken [Mutz04b].

Der hier genannte Lösungsansatz wird in den nachfolgenden Abschnitten detailliert vorgestellt. Kapitel 5.1 beginnt mit der Vorstellung der Regeln und SW-Metriken für den zustandsbasierten Modellierungsprozess. Abschließend wird in Kapitel 5.2 der Überprüfungsprozess vorgestellt, wobei auf das prototypische Analysewerkzeug näher eingegangen wird.

5.1 Konstruktive Qualitätssicherungsmaßnahmen

In größeren SW-Projekten bilden Modellierungsrichtlinien einen integralen Bestandteil des Entwicklungsprozesses. Diese konstruktiven Qualitätssicherungsmaßnahmen dienen unter anderem der Erreichung folgender Ziele:

- ***Fehlervermeidung*:** Durch die eingeschränkte Benutzung von schwer verständlichen, an sich fehleranfälligen oder unerwünschten Modellierungskonstrukten können Seiteneffekte und Fehler vermieden werden [BR01].
- ***Einheitlichkeit*:** Designregeln helfen einen konsistenten Modellierungsstil während des gesamten Projektverlaufs beizubehalten und verbessern somit die Lesbarkeit, Wartbarkeit und das Verständnis des Systems [PMPS01].
- ***Aufwandsverringerung*:** Wiederkehrende Arbeitsschritte lassen sich durch vorgegebene Funktionsbibliotheken, Algorithmensammlungen und Templates erleichtern [PMP01].

Zusätzlich können Regeln zur Modellkompatibilität und Steigerung der Effizienz bei Verhaltensmodellen beitragen. Zudem bietet der Einsatz von Regeln eine Möglichkeit, Modelle werkzeugunterstützt zu überprüfen.

Als eine weitere konstruktive Maßnahme eignen sich SW-Metriken [Fent91, KB02] zur Bewertung der Qualität eines SW-Systems. Dabei ist unter einer SW-Metrik folgendes zu verstehen:

SW-Metrik

Vorschrift zur Berechnung einer Kenngröße des betreffenden SW-Prozesses oder SW-Produkts auf der Basis eines Quellcodes oder eines Modells.

In der Automobilindustrie werden SW-Metriken bereits seit 10 Jahren erfolgreich eingesetzt und wurden in [MISRA95] zusammengestellt. Daher wird dieser Ansatz in STEP-X zur qualitativen und quantitativen Bewertung von zustandsbasierten Modellen herangezogen.

Im Folgenden werden diejenigen Modellierungsregeln und SW-Metriken vorgestellt, die sich im STEP-X Projekt bewährt haben. Zur besseren Übersicht wurden diese in verschiedene Kategorien gegliedert.

5.1.1 Modellierungsregeln für die zustandsbasierte SW-Entwicklung

Die Vielzahl der auf dem Markt befindlichen CASE Tools führt zu einer großen Variantenvielfalt [Beec94] der Statecharts, so dass bestehende Modellierungsregeln im Gegensatz zu Programmiervorschriften wie [MISRA04b] zumeist werkzeugspezifisch sind. Dies wird in den beiden Dokumenten [Krep01] und [MATH01] ersichtlich. Um die Forderung nach einem werkzeugunabhängigen Entwicklungsprozess dennoch erfüllen zu können, wurden in STEP-X vorwiegend werkzeugneutrale Regeln definiert.

Im Folgenden werden die unterschiedlichen Regelkategorien mit jeweils zwei Beispielen erläutert. Eine detaillierte Auflistung aller Regeln ist Anhang C zu entnehmen.

Konsistenzregeln

Mit Hilfe der Konsistenzregeln wird die korrekte syntaktische Nutzung der Modellierungselemente in einem Statechart gewährleistet. Da die Verletzung einer solchen Regel zu ernsthaften Problemen führen kann, ist diese Regelklasse zwingend einzuhalten. Die UML-Spezifikation [OMG03a] enthält ca. 30 Konsistenzregeln, die von den meisten auf dem Markt vorhandenen UML Tools unterstützt werden. Beispiele für Konsistenzregeln sind:

- Der Inhalt einer Bedingung muss syntaktisch korrekt sein. Um dieses zu überprüfen bedarf es eines Parsers, der den syntaktischen Ausdruck überprüft.
- Ein Join-Konnektor muss mindestens zwei eintreffende Transitionen und exakt eine ausgehende Transition besitzen.

Designregeln

Unter der Kategorie Designregeln werden all diejenigen Regeln zusammengefasst, die Modelle auf Lesbarkeit und Verständlichkeit hin überprüfen. Dadurch erhält der Entwickler Vorgaben, die zur Verbesserung des Verständnisses eines syntaktisch korrekten Modells beitragen. Beispiele hierfür sind:

- Miracle States sind zu vermeiden, d. h. Zustände, die zwar eine ausgehende, aber keine eingehende Transition besitzen, können nicht betreten werden und sind daher sinnlos.
- Beim Erstellen eines Zustands soll der Standardname in eine aussagekräftige Benennung umgewandelt werden.

Kompatibilitätsregeln

Die meisten Modellierungswerkzeuge unterstützen den Modellimport und -export nur durch ein proprietäres Datenformat, so dass ein Modellaustausch zwischen Werkzeugen demzufolge in den meisten Fällen nicht möglich ist. Einen Ansatz zum Austausch von Modellen bietet die UML mit dem bereits im Kapitel 2.2.3.4 vorgestellten XMI-Format. Leider ist dieses standardisierte Austauschformat noch nicht vollständig spezifiziert, so dass in der Praxis eine Anpassung des Formats notwendig ist. Dieses führt allerdings nicht zu der von der Industrie gewünschten Werkzeugkompatibilität [DHO01]. Eine weitere Prob-

lematik stellen die werkzeugspezifischen Modellierungskonstrukte dar. Beispielsweise enthält ASCET-SD an den Transitionen eine explizite Priorität, um Konflikte (vgl. Kapitel 2.2.3.4) zu vermeiden. Bei Stateflow hingegen kann durch eine wahre Aktionsbedingung (engl.: condition action) eine Folge von Aktionen auch beim Nichtschalten derselben Transition ausgeführt werden. Diese beiden Beispiele zeigen die Schwierigkeit bei der Definition eines gemeinsamen Austauschformats.

Daher werden in dieser Regelkategorie zwei wesentliche Kriterien überprüft. Einerseits soll die Kompatibilität unterschiedlicher Modelle gewährleistet werden, und andererseits soll eine Überprüfung der Modelle auf UML-Konformität erfolgen. Beispiele für diese Art von Regeln sind:

- Die Ausführungsreihenfolge orthogonaler Regionen muss in beiden Werkzeugen äquivalent sein. In dem Werkzeug Artisan RtS verläuft sie von oben nach unten, in Rhapsody ist sie undefiniert. ASCET-SD enthält keine Orthogonalität in einem Statechart.
- Ein shallow history-Konnektor ist in Rhapsody nicht implementiert, daher soll dieses Modellierungselement in der gesamten Werkzeugkette vermieden werden.

Layoutregeln

Eine homogene und übersichtliche Darstellung komplexer Modelle ist die Voraussetzung für eine einfache Einarbeitung in ein SW-System. Zusätzlich wird die Einarbeitung verkürzt, Fehler schneller erkannt sowie Test- und Simulationsläufe besser angepasst.

In einigen Literaturquellen, wie z. B. [CMT02, Eich02, HY99, Seke98] werden Verfahren zur grafischen Modellierung von zustandsbasierten Systemen vorgeschlagen. Diese und weitere Vorschläge dienen in STEP-X als Basis zur Strukturierung von wohlspezifizierten zustandsbasierten SW-Systemen.

Beispielsweise werden Zustände so platziert, dass die Zustandsdiagramme eine möglichst symmetrische Form erhalten. Die Transitionen im Modell sollen möglichst im Uhrzeigersinn verlaufen, wobei die Beschriftung der Transitionen links von der Pfeilrichtung der Transitionen zu positionieren ist. Außerdem soll der Verlauf der Transitionen entweder horizontal oder vertikal verlaufen, wobei ein Richtungswechsel nur um 90° erlaubt ist.

Zusätzliche Regeln sind beispielsweise:

- Eine Transitionsüberschneidung ist zu vermeiden. Sollte dies doch der Fall sein, so muss dies in einem möglichst großen Winkel geschehen.
- Der Startzustand sollte sich in der linken oberen Ecke eines zusammengesetzten Zustands befinden, und die Initial-Transition sollte stets links bzw. oben in den Zustand führen.

Optimierungsregeln

Trotz vorgegebener Templates zur Erstellung von Statecharts (vgl. Kapitel 4.3.2 auf Seite 94), können Modelle mit gleichem Verhalten auf unterschiedliche Art und Weise erstellt

werden. Dies kann sich jedoch auf die Lesbarkeit und Code-Größe erheblich auswirken. Optimierungsregeln werden eingesetzt, um die Effektivität eines zustandsbasierten Modells möglichst beizubehalten. Beispiele für Optimierungsregeln sind:

- Parallele Regionen mit nur einem Basiszustand können evtl. vermieden werden.
- Ein zusammengesetzter Zustand, der nur einen Unterzustand besitzt, kann unter bestimmten Umständen in einen Basiszustand umgewandelt werden.

5.1.2 SW-Metriken zur Bewertung der Modellqualität

Genau wie die Regeln, sind auch die SW-Metriken in verschiedene Kategorien eingeteilt. Aus Kompatibilitätsgründen erhalten die SW-Metriken einen zu den Modellierungsregeln adäquaten Kategorienamen [Mutz04b]. Allerdings erhalten die Kategorien und die einzelnen SW-Metriken zusätzlich eine Gewichtung, um eine quantifizierte Modellauswertung zu ermöglichen. Bevor auf die Gewichtung im nächsten Kapitel näher eingegangen wird, erfolgt zunächst eine Vorstellung der Metrikenkategorien.

Zustandsmetriken

Zustandsmetriken bewerten das Modell nach strukturellen Gesichtspunkten. Dabei spielen vorwiegend die Anzahl und die Verteilung der Zustände eine wesentliche Rolle. Um die ungefähre Komplexitätsverteilung der Funktionen ermitteln zu können, wird die minimale und die maximale Anzahl der Zustände jedes Statecharts analysiert und mit dem daraus resultierenden Mittelwert verglichen. Weicht die Anzahl der Zustände einer Funktionsklasse sehr stark vom Mittelwert ab, so ist das dazugehörige Statechart daraufhin zu überprüfen, ob es in weitere Funktionsklassen zu untergliedern ist. Durch Einbeziehung der Hierarchieebenen lässt sich die Zustandsverteilung auch in einem Statechart verfolgen, wobei eine adäquate Verteilung der Zustände auf die einzelnen Ebenen angestrebt wird.

Ein ähnliches Vorgehen wird auch für die Bestimmung der Länge eines Zustandsnamens angewendet, indem das Minimum, das Maximum und der Durchschnitt aller Zustandsnamen ermittelt werden. Eine gute Bewertung liegt dann vor, wenn die Namenslänge der Zustände einen bestimmten Bereich nicht unterschreitet aber auch nicht überschreitet. Das Analysieren der Strukturen, wie Hierarchietiefe oder Anordnung der parallelen Regionen ist ein weiteres Kriterium bei der Bewertung der Statecharts.

Pseudozustandsmetriken

Die Pseudozustandsmetriken sind den Zustandsmetriken sehr ähnlich, wobei die Bewertungskriterien jedoch auf die Anzahl der Pseudozustände reduziert werden. Analysiert wird beispielsweise, wie häufig Pseudozustände in einem Statechart vorkommen. Die Nutzung der Pseudozustände sollte gut abgewogen werden, da sie zwar komplexe Sachverhalte auf einfache Weise darstellen, jedoch in zu großer Zahl das Verständnis beeinträchtigen können.

Transitionsmetriken

Transitionsmetriken bewerten die Komplexität eines Statecharts, indem die Transitionsverläufe zwischen den (Pseudo-)Zuständen analysiert werden. Wie bereits bei den Zustandsmetriken, wird zunächst die Anzahl aller Transitionen in allen Statecharts bestimmt. Anschließend können das Minimum und Maximum sowie der Durchschnitt ermittelt werden. Dabei wird zwischen ein- und ausgehenden Transitionen unterschieden. Zusätzlich werden Transitionsverläufe, wie z. B. interlevel- oder self-loop-Transitionen analysiert. Zudem findet eine Analyse der Transitionsbeschriftung statt, wobei die Länge der Bedingungen und die Anzahl der Aktionen ein mögliches Bewertungskriterium darstellen.

Layoutmetriken

Layoutmetriken bewerten die Übersichtlichkeit und das Verständnis des zustandsbasierten Modells, indem unter anderem die Anordnung und Skalierung von Modellelementen analysiert werden. Wie bereits in den Layoutregeln erläutert, verbessert die symmetrische Anordnung der Statecharts die Lesbarkeit. Des Weiteren wird der Verlauf der Transitionen analysiert, wobei Überschneidungen und diagonale Transitionsverläufe zu einer Verschlechterung der Lesbarkeit führen.

Klassenmetriken

Klassenmetriken analysieren im Gegensatz zu den anderen SW-Metriken einzelne Funktionsklassen und nicht Statecharts. Dabei wird unter anderem die Anzahl der Attribute und Methoden jeder Klasse sowie deren Länge, Typ und Parameteranzahl bestimmt.

5.2 Analyse und Bewertung zustandsbasierter SW-Systeme

Obwohl formale Verifikationstechniken, wie das Model Checking [Peuk97] die Qualität auch im Automobilbereich nachweislich verbessern konnten [MD02, BBBD04], geht die Anwendung solcher Methoden in industriellen Projekten immer weiter zurück. Die Ursache für den Rückgang liegt vor allem in den hohen Kosten, die durch das zu schulende Personal, den längeren Spezifizierungsprozess und die notwendige Werkzeuganpassung [BDEH+97, BDEM+96] entstehen.

Um trotzdem auf analytische Methoden nicht gänzlich verzichten zu müssen, werden in STEP-X statische Analyseverfahren [JW96, PMHV03] verfolgt. Mit diesen können die im vorherigen Abschnitt erläuterten Modellierungsregeln und weitere Kriterien [HHM01] formal überprüft werden, ohne die textuellen Anforderungen formal spezifizieren zu müssen.

Die meisten kommerziellen Modellierungswerkzeuge bieten rudimentäre Konsistenzüberprüfungen anhand einer festdefinierten Regelbasis an, die sowohl standardisierte Regeln, z. B. aus der UML-Spezifikation, als auch werkzeugspezifische Abfragen beinhaltet. Es fehlt jedoch die grundsätzliche Option zur Erweiterung vorhandener Regeln. Somit können die in den letzten Jahren entworfenen Modellierungsrichtlinien für den Automobilbereich [FORD00, MISRA04a] nicht in den herkömmlichen Modellierungswerkzeugen verwendet werden. Aus diesem Grund wurden spezielle Analysewerkzeuge, wie z. B. das

kommerzielle Werkzeug *mint* [GT04] oder der prototypische *GDC* [MKF00], eine Kooperation zwischen DaimlerChrysler und der Fachhochschule Esslingen, entwickelt, die auch diese Art der Regeln überprüfen.

Sowohl bei mint als auch bei GDC haben sich folgende Nachteile herausgestellt:

- Begrenzung der Regelüberprüfung nur auf die MATLAB Toolbox,
- Starke Bindung zwischen Regeln und internen Datenstrukturen der Werkzeuge,
- Proprietäre Skriptssprachen erschweren die Einarbeitung,
- Regeln sind während der Ausführung nicht selektierbar und nicht modifizierbar.

Andere Analysewerkzeuge wie USE [GR02] dagegen sind nur auf UML-Klassendiagramme begrenzt.

Vor diesem Hintergrund wird das Werkzeug *Regel Checker*, welches im Rahmen des STEP-X Projektes von der AG Digitales Lastenheft entwickelt wurde, eingesetzt. Mit dem Prototyp wird ein genereller und konfigurierbarer Ansatz verfolgt, der sowohl UML Tools als auch andere zustandsbasierte Werkzeuge unterstützt. Konsequenterweise basiert diese Analyse auf einer flexiblen Datenstruktur, mit der sowohl Standardkonstrukte als auch werkzeugspezifische Modellelemente anhand von definierten Modellierungsregeln überprüft werden können. Die strukturelle Trennung der Regelbasis vom Analyseprogramm, ermöglicht einen flexiblen Einsatz des Werkzeugs. Durch die GUI-gestützte Regelanpassung und Verwendung standardisierter Beschreibungssprachen zur Regeldefinition wird ein komfortables Vorgehen bei der Erstellung anwendungsspezifischer Regeln gewährleistet. Zudem können durch SW-Metriken die Modelle qualitativ und quantitativ bewertet werden.

Im Folgenden wird auf das Werkzeug Regel Checker und den Analyseprozess näher eingegangen.

5.2.1 Der Regel Checker

Die Hauptaufgabe des Regel Checkers besteht in der Überprüfung der Einhaltung von benutzerdefinierten Regeln sowie in der Bewertung der Modellqualität anhand von SW-Metriken. Im Folgenden wird auf den Aufbau des Regel Checkers und die Definition von Regeln näher eingegangen.

5.2.1.1 Die innere Struktur des Regel Checkers

Der Regel Checker ist in sechs Module untergliedert, die in Abbildung 5-1 zu sehen sind.

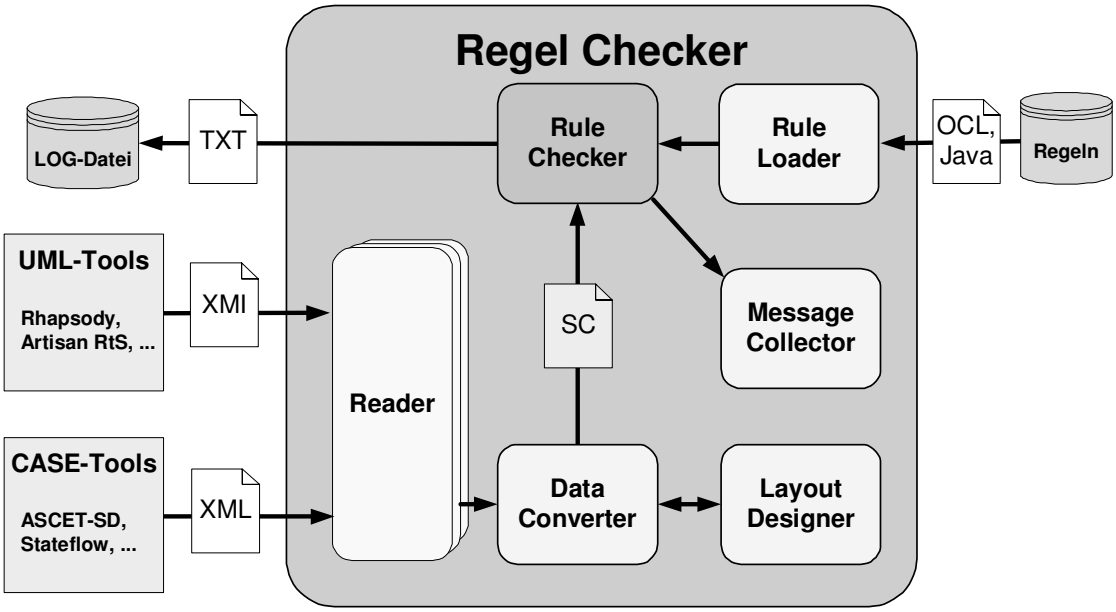


Abbildung 5-1: Vereinfachte Darstellung der inneren Struktur des Regel Checkers

Reader

Zum Einlesen von zustandsbasierten Modellen steht für jedes Werkzeug ein Reader-Modul zur Verfügung, mit dem die jeweiligen Modelle eingelesen werden können. Um das Importieren eines Modells möglichst komfortabel zu halten, ist eine automatische Auswahl des richtigen Reader-Moduls notwendig. Dazu werden Informationen, wie Werkzeugname und Version aus den jeweiligen Modelldateien identifiziert. Zurzeit ist der Modellimport aus den drei Werkzeugen Rhapsody [Hare02], ASCET-SD [ETAS00] und der MATLAB Toolbox [HR04] möglich. Eine Erweiterung um weitere Werkzeuge ist nur durch das Hinzufügen eines neuen Reader-Moduls möglich. Voraussetzung ist, dass die zu importierenden Modelle in Form einer lesbaren ASCII-Datei existieren. Eine Anpassung der Datenstruktur des Regel Checkers aus Abbildung 5-2 ist nur dann notwendig, wenn das Werkzeug über proprietäre Modellkonstrukte verfügt. Weitere Informationen zur Erstellung von Reader-Modulen können den Arbeiten von [Köck03] und [Piet03] entnommen werden.

DataConverter

Das *DataConverter*-Modul wandelt die eingelesenen Dateien in die Datenstruktur des Regel Checkers um. Hierfür wurde die in Abbildung 5-2 dargestellte Datenstruktur (SC ≡ Statechart) entwickelt, die einerseits flexibel genug ist, um unterschiedliche Statechart-Varianten intern abzubilden und andererseits einheitliche Strukturen vorgibt, anhand derer eine Regelüberprüfung möglich ist.

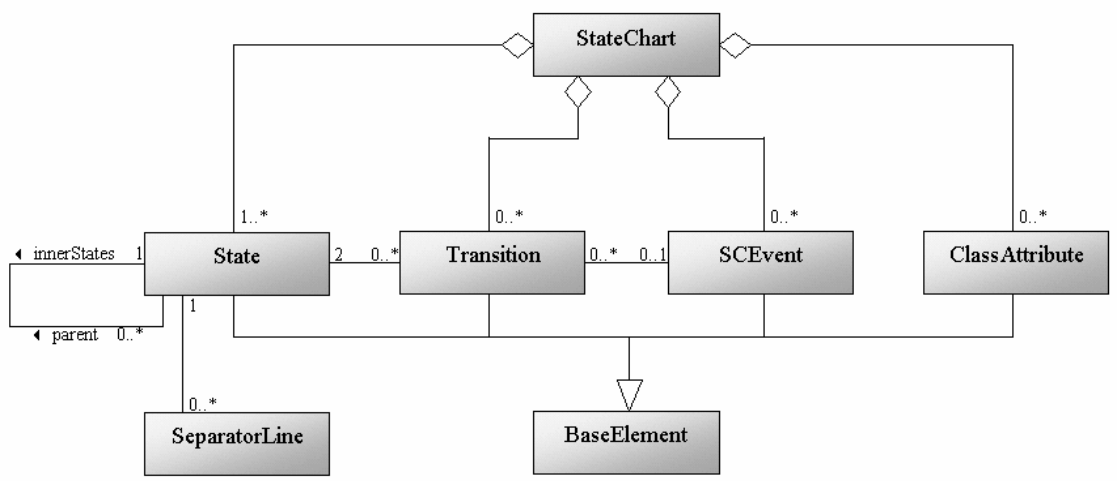


Abbildung 5-2: Ausschnitt der Datenstruktur des Regel Checkers [Piet03]

Ausgehend von einem Zustand aus der Klasse *State* erfolgt ein rekursiver Aufbau der Zustandshierarchie. Ein Zustand kann dabei beliebig viele direkte Unterzustände besitzen. Zur Abgrenzung der orthogonalen Regionen wird auf die Klasse *SeparatorLine* zurückgegriffen. Alle Zustände können auf eine beliebige Anzahl von Zustandsübergängen der Kasse *Transition* referenzieren. Dabei besitzt eine Transition immer genau einen Start- und einen Endzustand. Ausgelöst werden kann ein Zustandsübergang durch ein Ereignis, welches durch eine Instanz der Klasse *SCEvent* repräsentiert wird. Ein Ereignis kann mehrere Transitionen triggern, wohingegen eine Transition durch maximal ein Ereignis ausgelöst wird. Die Basisklasse *BaseElement* stellt fundamentale Eigenschaften der Modellelemente bereit. Darüber hinaus ist die Datenstruktur um grafische Angaben erweitert worden. Ein Modellkonstrukt besitzt somit Informationen über die Position, die Skalierung und die Farbe. Beispielsweise besitzt eine Transition zusätzlich zu den beiden Start- und Endpunkten eine beliebige Anzahl von Zwischenpunkten, die als Vorgabe für einen Linienzug dienen. Zudem existiert die Möglichkeit zur Angabe eines frei positionierbaren Beschriftungstextes.

LayoutDesigner

Im Gegensatz zu Modellen aus Stateflow und ASCET-SD, die ihre grafischen Daten abspeichern, gestaltet sich die Darstellung der importierten UML-Modelle problematischer. In diesem Format bleiben grafische Informationen vollständig unberücksichtigt, so dass beim Importieren von XMI-Dateien sämtliche Informationen zum grafischen Aufbau verloren gehen.

Mit Hilfe des *LayoutDesigners* besteht die Möglichkeit das Modell trotz dieser fehlenden Informationen durch spezielle Layoutalgorithmen [Mutz97, Mutz01, Klos03] zu visualisieren. In der Arbeit von [Bauh04] wurde für den Regel Checker ein Layout-Algorithmus implementiert, der auf dem Algorithmus von [Cast02] basiert. Durch zusätzliche Parameter kann der Algorithmus an bestimmte Layouts angepasst werden. Zu erwähnen ist aber, dass sich die grafische Darstellung maßgeblich vom originalen Modell unterscheidet, so dass Regelabfragen bezüglich des Layouts in diesem Fall nicht zielführend sind.

RuleLoader

Beim Start des Programms werden zunächst alle Regeln in Form einer Java-Klasse bzw. OCL-Datei in das dafür vorgesehene Verzeichnis vom *RuleLoader* eingelesen und in das Auswahlmenü des Regel Checkers aufgenommen. Diese Lösungsidee basiert auf der Möglichkeit von Java, dynamisch Klassen zu laden, um einen Eingriff in den Quelltext nicht vornehmen zu müssen. Darüber hinaus werden in Abhängigkeit des geladenen Modelltyps werkzeugspezifische Regeln automatisch deaktiviert, um unsinnige Fehlermeldungen zu vermeiden.

RuleChecker

Das Modul *RuleChecker* koordiniert den eigentlichen Prozess der Regelüberprüfung und der Modellbewertung. Zusätzlich ist er neben der Visualisierung der Ergebnisse auch für die Protokollierung der System- und Benutzeraktivitäten zuständig. Ein ausführliches Beispiel wird in Kapitel 5.2.2 beschrieben.

MessageCollector

Um die identifizierten Beanstandungen zu einem späteren Zeitpunkt aussagekräftig darstellen zu können, wurde mit dem *MessageCollector* ein Nachrichtensystem entwickelt, das die Aufbereitung der Überprüfungsergebnisse ermöglicht. Nachrichten repräsentieren das Ergebnis einer Überprüfung im Falle einer Beanstandung. Da nicht jede Beanstandung gleichbedeutend ist, wird bei der Ergebnispräsentation differenziert. Zu diesem Zweck existiert eine Klassifizierung nach Schweregraden. Hierbei wird eine Regel durch Belegung eines Attributs mit einer vordefinierten Konstante als Vorschlag, Warnung oder Fehler gekennzeichnet. Vorschläge sind Nachrichtentypen, die einen Hinweis bezüglich einer möglichen Verbesserung enthalten (z. B. sollten zwei oder mehrere Zustände mit gleichem Namen nicht existieren). Warnungen sind Nachrichten, die auf mögliche Inkonsistenzen hinweisen (z. B. jede deklarierte Variable sollte am Anfang initialisiert werden). Fehlermeldungen werden generiert, wenn ein Statechart fehlerhafte Eigenschaften aufweist (z. B. jede Transition muss einen Zustand betreten).

5.2.1.2 Regeldefinition

Nachdem ein Einblick in die Struktur des Regel Checkers gegeben wurde, stellt sich die Frage, auf welche Art und Weise die Regeln implementiert und angewendet werden können. Eine Antwort geben Buck und Rau in [BR01], in dem sie eine Vielzahl von Anforderungen an eine Regelbasis aufgestellt haben. Die wichtigsten davon sind:

- ***Verständlichkeit*:** Die Bedeutung der Regeln sollte für alle Entwickler verständlich sein.
- ***Atomar*:** Jede Regel sollte nur einen einzigen Aspekt umfassen. Dadurch kann eine gezielte Überprüfung erfolgen.
- ***Konsistenz*:** Die gesamte Regelmenge muss in sich konsistent sein. D. h., dass die Verwendung von Regel A nicht zu einer Verletzung der Regel B führen darf.

- ***Anpassbarkeit*:** Richtlinien sind nicht allgemeingültig und müssen daher an projekt- und werkzeugspezifische Gegebenheiten angepasst werden können.
- ***Effektivität*:** Richtlinien sind nur dann effektiv, wenn die Überprüfung dieser automatisch erfolgen kann.

Alle diese Anforderungen wurden in dem Konzept des Regel Checkers in Betracht gezogen: Die *Verständlichkeit* wird einerseits durch Handbücher [MH03, Piet03] mit detaillierten Beschreibungen und grafischen Beispielen erreicht. Andererseits steht dem Regel Checker ein dynamisches Hilfesystem zur Verfügung, dass durch das HTML-Format über einen Browser gelesen werden kann. Die zusätzliche Generierung von Berichten bezüglich der Analyseergebnisse, erleichtert die Fehlersuche und Dokumentation. Um die *Atomarität* der Regeln sicherzustellen, werden sie anhand von verschiedenen Kriterien strukturiert. Die Gewährleistung der *Konsistenz* erfolgt durch die Verhinderung gegenseitiger Regelaufrufe und die Unterdrückung von bereits analysierten Fehlerbeanstandungen. Um die Regeln flexibel an die projektspezifischen Eigenschaften anpassen zu können, wurde ein Mechanismus zur nachträglichen Konfiguration der Regeln durch den Anwender integriert. Diese Anpassung erfolgt ohne Eingriffe in den Quellcode der Regeln, um eine einfache Handhabbarkeit zu gewährleisten. Abbildung 5-3 zeigt beispielhaft die Konfiguration einer Regel, die die Länge des Zustandsnamens begrenzt. Dabei unterliegen die Parameter keiner Einschränkung bezüglich der Anzahl und Typen. Die getätigte Konfiguration wird automatisch gespeichert und beim nächsten Aufruf des Regel Checkers geladen.

Die *Effektivität* wird dadurch gewährleistet, dass die Überprüfung der Modellierungsregeln automatisch erfolgt. Dies erfordert eine formale, ausführbare Spezifikation der zu testenden Modellierungsrichtlinien.

Die meisten Regeln sind in Java implementiert. Jedoch gibt es die Möglichkeit Regeln auch im Standardformat OCL (vgl. Kapitel 2.2.3.3 auf Seite 34) zu formulieren. Die Möglichkeit, Regeln in OCL zu definieren, bringt eine Vielzahl von Vorteilen mit sich. Einerseits können die Regeln auf einem höheren Abstraktionslevel definiert werden und andererseits benötigt der Benutzer kein detailliertes Wissen über Java-Programmierung und keine Kenntnis der internen Datenstruktur des Regel Checkers. Darüber hinaus können mit Hilfe des hierfür entwickelten OCL-Interpreters [Potr05] die meisten Modellierungsregeln direkt von der UML-Spezifikation umgesetzt werden, da die dortigen Modellierungsregeln bereits in OCL vorhanden sind. Aus diesem Grund ist diese Art der Regeldefinition unter anderem auch in [Rich01, RG03] wiederzufinden, die insbesondere die Konsistenz von UML-Modellen überprüfen. Ein weiterer Vorteil von OCL ist, dass die Formulierung der Abfragen kompakt gehalten ist. In Abbildung 5-4 wird beispielhaft die Spezifikation einer Regel in Java und OCL dargestellt. Das Beispiel verweist auf die von der OMG well-formedness Regel, bei der ein Startzustand über keine eintreffenden Transitionen verfügen darf. Es ist ersichtlich, dass die zweizeilige OCL-Anweisung kompakter ist als die vergleichbare Implementierung in Java.

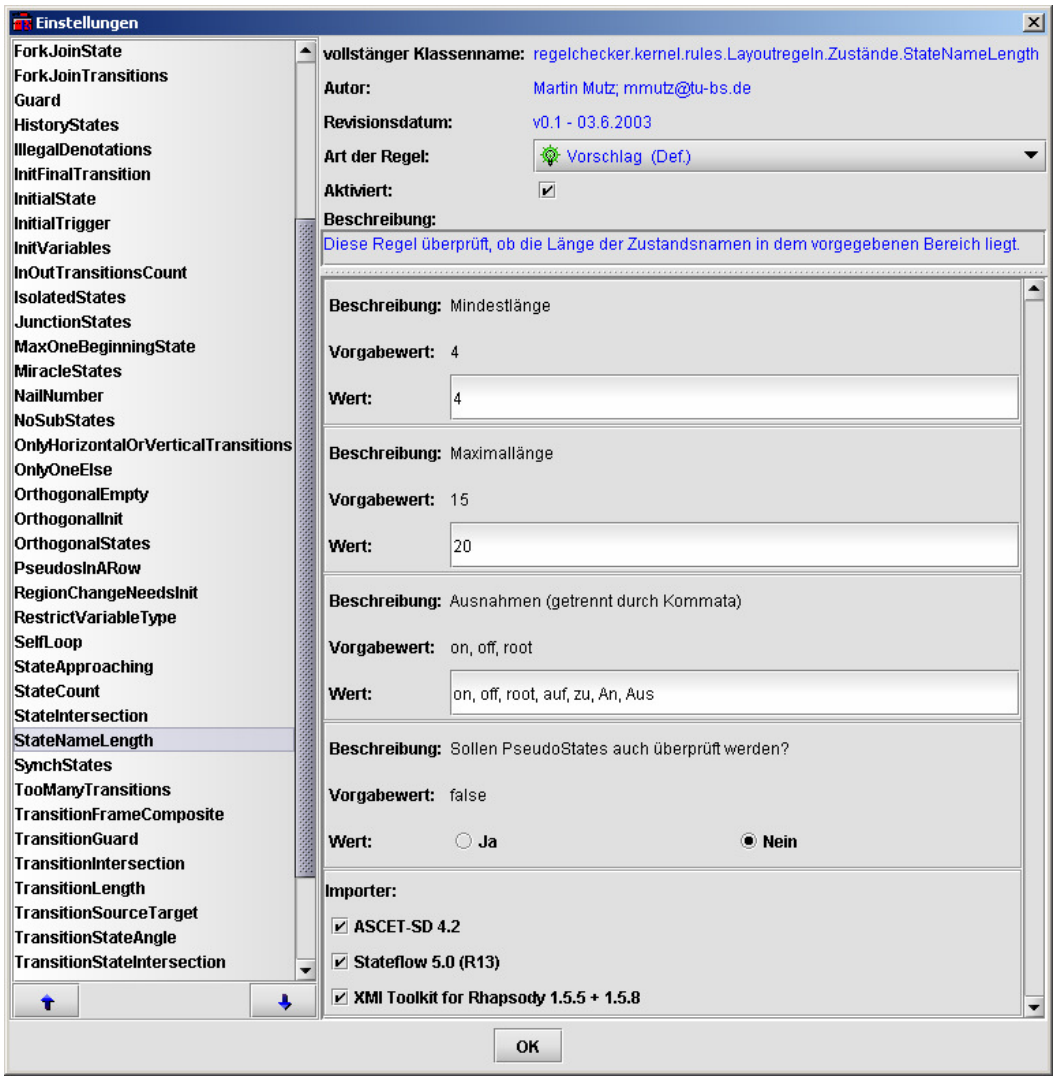


Abbildung 5-3: Parametrisierung der Regeln

Eine Schwäche von OCL ist allerdings, dass grafische Informationen nicht überprüft werden können, da das UML Meta-Modell (vgl. Kapitel 2.2.3.1 auf Seite 32) diese Angaben nicht beinhaltet. Eine Regelformulierung wie “Befindet sich der Startzustand in der oberen linken Ecke des Statecharts?” ist daher nicht möglich. Darüber hinaus stellt die OCL eine Menge an mathematischen Funktionen und Symbolen bereit, enthält aber nur wenige String-Operationen. Somit lässt sich beispielsweise eine Überprüfung von Bedingungen an Transitionen nur schwer zu realisieren.

```
public void check(StateChart sc) {
  boolean checkInits = getParameterValue(0).equalsIgnoreCase("true");

  Iterator it = sc.getStates().iterator();
  State state = null;

  while (it.hasNext()) {
    state = (State)it.next();
    if ((checkInits) && (state.getType() == State.INITIAL_STATE)) {
      if (state.getIncomingTransitionCount() != 0) {
          ...
```

```
context InitialState
inv: self.incoming->size()=0
```

Abbildung 5-4: Vergleich zwischen Java und OCL

In STEP-X wird das Problem in soweit gelöst, dass auf der einen Seite Java-Regeln zur Beschreibung graphischer und komplexer Ausdrücke dienen und auf der anderen Seite die Menge der vordefinierten OCL-Methoden um String-Methoden erweitert wird.

Es wurde eine Vielzahl von Modellierungsregeln aus existierenden Regelkatalogen wie [Ambl03, FORD00, Leve95, MISRA04a] sowie studentischen Arbeiten [Chkh04, Knie03, Koss00, Piet03] implementiert. Eine Auflistung aller im Regel Checker vorkommenden Regeln befindet sich im Anhang C.

5.2.2 Der Überprüfungsprozess am Beispiel

Die Hauptaufgaben des Regel Checkers bestehen, wie bereits erwähnt, in der automatischen Überprüfung der Einhaltung von Regeln sowie in der Bewertung der Modelle durch SW-Metriken. In Abbildung 5-5 ist der Überprüfungsprozess vereinfacht dargestellt. Er besteht aus den beiden Aktivitäten *Modellüberprüfung* und *Modellbewertung*, wobei die Modellüberprüfung als Blackbox der Abbildung 5-1 anzusehen ist.

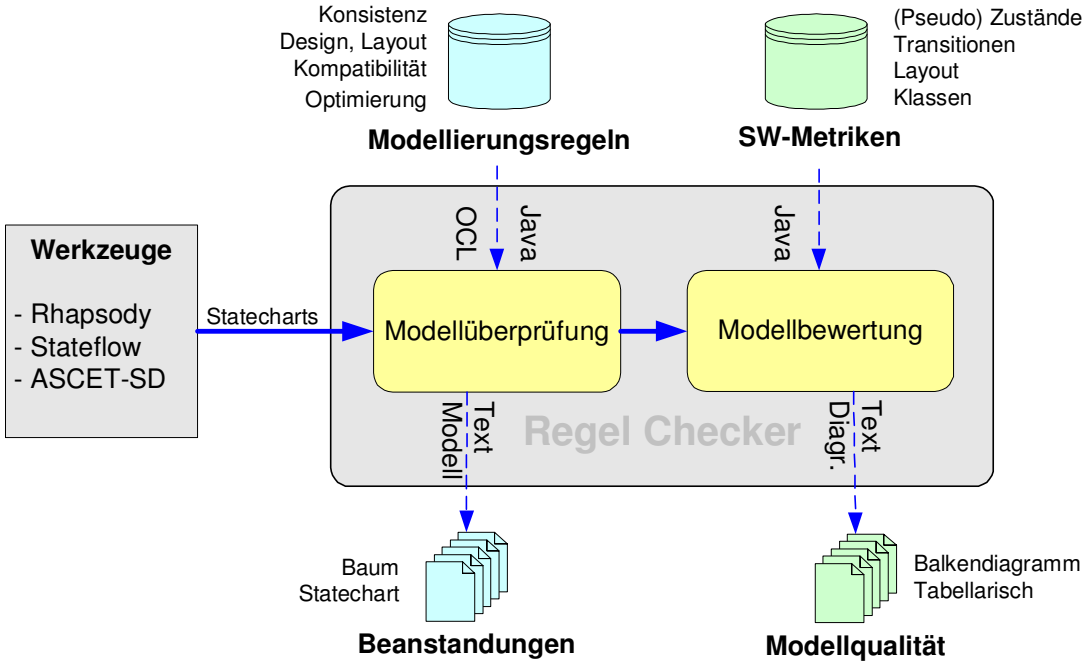


Abbildung 5-5: Das Konzept des Überprüfungsprozesses

Das Einlesen eines oder mehrerer Statecharts aus einem Werkzeug ist der erste Schritt der Analyse. Anschließend wird das Statechart anhand der ausgewählten Regeln überprüft und die Beanstandungen textuell oder grafisch dem Anwender mitgeteilt. Nach der Überprüfung des Modells kann eine Modellbewertung erfolgen, wobei auch hier die SW-Metriken wahlweise aktiviert und parametrisiert werden können.

Die beiden Überprüfungsarten können grundsätzlich unabhängig voneinander durchgeführt werden. Aus methodischem Gesichtspunkt ist die Modellbewertung nach der Regelüberprüfung durchzuführen, da eine Bewertung nur bei korrekten Modellen sinnvoll ist.

Das hier kurz dargestellte Vorgehen bei der Überprüfung und Bewertung der Modelle wird im Folgenden am Beispiel näher erläutert.

5.2.2.1 Modellüberprüfung

Anhand des Statecharts aus Abbildung 5-6 wird das Analyseergebnis des Regel Checkers vorgestellt, wobei das Zustandsdiagramm folgende Modellierungsfehler enthält:

- Die orthogonalen Regionen *Par1, par1* und *PAR1* haben die gleichen Namen, die sich nur durch die Großschreibung unterscheiden. Dies kann zu einer Erschwerung der Verständlichkeit führen.
- Die Region *par1* besitzt keinen Startzustand, so dass eine Ausführung des Modells nicht möglich ist.
- Der Zustand *sub_function* ist isoliert und kann daher nicht betreten werden.
- Beim Zustand *stop* handelt es sich um einen Miracle State, von dem eine Transition herausführt, der jedoch nie betreten wird.
- Die Zustandsbeschriftung *state_8* ist ein Default-Name, der durch ein entsprechendes Werkzeug automatisch erstellt wird. Zur besseren Lesbarkeit sollten Zustandsnamen explizit mit verständlichen Bezeichnern versehen werden.
- Die Transition von *Alarm* nach *Error* ist eine abschließende Transition. Diese sollte nur in Kombination mit einem Endzustand verwendet werden.

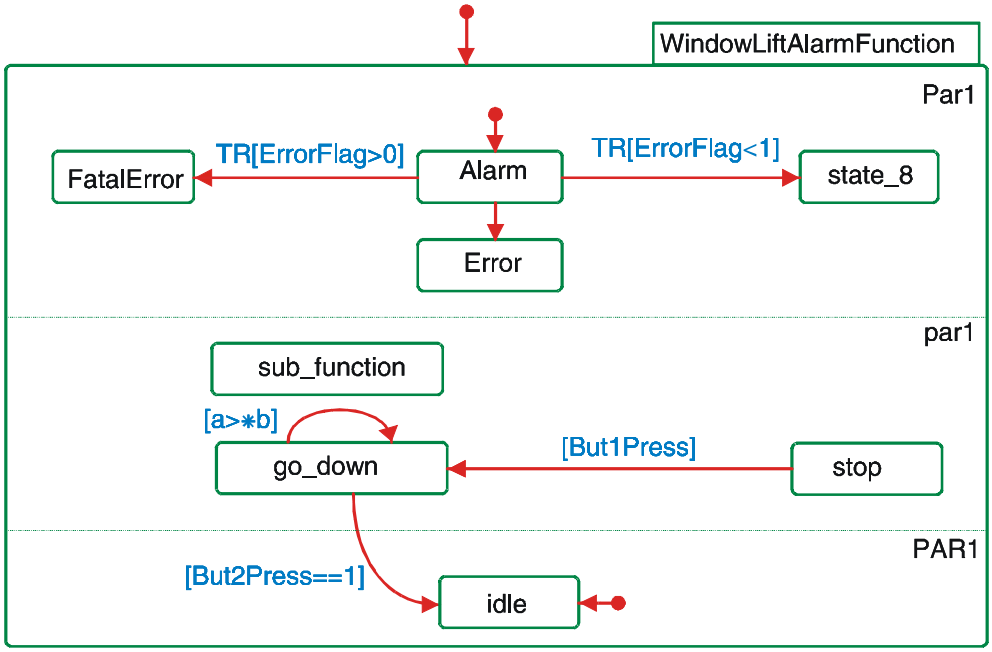


Abbildung 5-6: Statechart mit Modellierungsfehlern

Nach der Durchführung der Analyse wird im Regel Checker ein Baum (vgl. Abbildung 5-7) dargestellt, der dem eingelesenen Statechart entspricht. Dabei wird zwischen sequentiellen Zuständen (Kreis), parallelen Regionen (Quadrat) und Pseudozuständen (Dreieck) unterschieden.

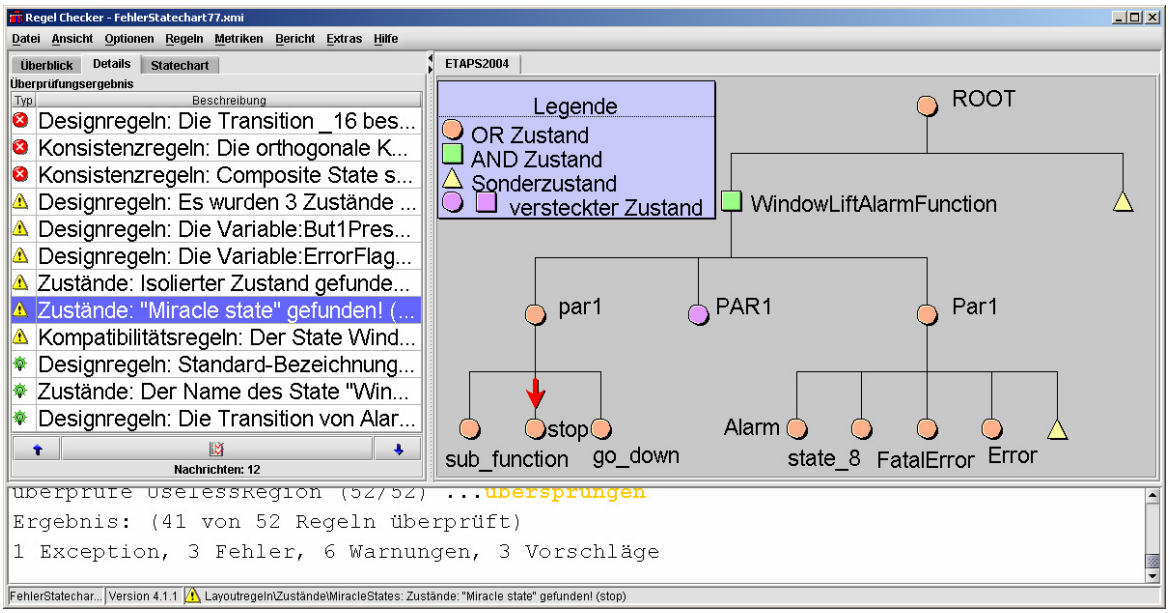


Abbildung 5-7: Hauptfenster des Regel Checkers

Die Wurzel im Baum bildet der Zustand *ROOT*, der selbst im Statechart nicht angezeigt wird. Er symbolisiert den Startknoten im Baum. Auf der zweiten Ebene ist der AND-Zustand *WindowLiftAlarmFunction* und ein dazugehöriger Pseudozustand (Startzustand) abgebildet. Dieser zusammengesetzte AND-Zustand besteht aus drei parallelen Regionen, die auf der nächsten Ebene zu sehen sind. Der lilagefärbte Zustand PAR1 symbolisiert einen Teilbaum, der nicht weiter ausgeklappt wurde. Damit kann die Lesbarkeit von umfangreichen Statecharts erhöht werden. Die Blätter stellen die Basiszustände im Modell dar.

Durch diese Art der Darstellung ist eine weitere Sicht auf das Statechart möglich. Sie betont gegenüber der herkömmlichen Abbildung die Hierarchien und die daraus entstehenden Abhängigkeiten, die innerhalb eines Zustandsdiagramms existieren. Zudem ist diese Darstellungsform unabhängig von einer konkreten Positionierung und Dimensionierung der Diagrammelemente in einem CASE Tool. Um strukturelle Ähnlichkeiten zwischen zwei Diagrammen festzustellen, liefert der hierarchische Aufbau eines Diagramms wichtige Anhaltspunkte. Vorwiegend wird der Baum jedoch zur Darstellung der aufgetretenen Beanstandungen im Modell verwendet.

Im linken Teil des Hauptfensters ist eine Liste der analysierten Beanstandungen, sortiert nach Schweregraden, zu sehen. Durch das Selektieren einer solchen Beanstandung, wird der entsprechende Knoten im Baum durch einen Pfeil markiert, wie in Abbildung 5-7 zu sehen ist. Alternativ kann die beanstandete Stelle auch im Modell markiert werden, dies hat den Vorteil, dass auch Beanstandungen hinsichtlich der Transition angezeigt werden können, die so im Baum nicht ersichtlich sind.

Im unteren Teil des Hauptfensters werden sowohl die Benutzer- und Systemaktivitäten protokolliert als auch die während des Analysevorgangs aktivierten Regeln aufgelistet. Aus Abbildung 5-7 ist ersichtlich, dass drei Fehler, sechs Warnungen und drei Vorschläge aus der Analyse hervorgehen. Die aufgetretene Exception symbolisiert einen Runtime-

Fehler, der von Java generiert wird. Dadurch erhält der Anwender eine Rückmeldung, ob die Regel korrekt ausgeführt wurde oder ob ein Programmfehler vorlag. Die Modellüberprüfung ergab zusätzlich zu den bereits ersichtlichen Regelverletzungen weitere Beanstandungen, die durch das alleinige Betrachten des Statecharts nicht ermittelt hätten werden können:

- ***Fehler:*** Der Typ der Variable *ErrorFlag* ist nicht erlaubt.
- ***Fehler:*** Die Variable *But1Press* wird im Statechart benutzt aber nicht als Klassenattribut deklariert.
- ***Warnung:*** Der Name vom Zustand *WindowLiftAlarmFunction* ist größer als die erlaubte Maximallänge (vgl. Abbildung 5-3 auf Seite 127).
- ***Vorschlag:*** Die Region PAR1 kann evtl. eingespart werden.

5.2.2.2 Modellbewertung

Wird das Statechart-Modell nach der ersten Analysephase so modifiziert, dass keine Beanstandungen vom Modul *RuleChecker* erzeugt werden, ist das Modell als korrekt einzustufen. Anschließend kann mit der Metrikenüberprüfung (vgl. Abbildung 5-5) begonnen werden. Wie bereits bei der Regelauswahl können Metriken je nach gewünschtem Detaillierungsgrad ausgewählt werden. Allerdings verläuft die Definition der SW-Metriken derzeit nur in Java. Die Arbeit von [BBA02] zeigt aber, dass die Definition von SW-Metriken auch mit Hilfe von OCL möglich ist.

Am Beispiel der Kategorie Zustandsmetriken aus Kapitel 5.1.2 soll der letzte Schritt des Überprüfungsprozesses, die Bewertung des Gesamtmodells, veranschaulicht werden. Zunächst wird die Anzahl aller Zustände ermittelt, um den Umfang des Gesamtmodells einzuschätzen [MN00]. Dabei werden Modelle bis 500 Zustände als klein, bis 2000 als umfangreich und darüber hinaus als groß definiert. Des Weiteren werden die minimale und die maximale Anzahl der Zustände jedes Statecharts ermittelt sowie der Mittelwert berechnet. Somit ist die Verteilung der Funktionskomplexität ersichtlich. Eine weitere Metrik analysiert die hierarchische Struktur der Statecharts. Dabei wird die Tiefe und Breite jedes Statecharts festgestellt und ins Verhältnis gesetzt. Dadurch können unterschiedliche Aussagen getroffen werden. Beispielsweise sind tiefe (viele Ebenen) und schmale (wenig Zustände pro Ebene) Statecharts genauso unübersichtlich, wie flache aber sehr breite Statecharts. Als letztes wird die Beschriftung der Zustände überprüft.

Das Analyseergebnis kann als Balkendiagramm, wie in Abbildung 5-8 zu sehen ist, dargestellt werden. In dem Screenshot sind drei Kategorien abgebildet, die jeweils Metriken enthalten. Da SW-Metriken je nach Verwendung unterschiedlich starke Ausprägungen haben, werden sie mit einer entsprechenden Gewichtung von 1 (unrelevant für die Gesamtbewertung) bis 10 (sehr wichtig für die Gesamtbewertung) versehen. Die Gewichtungsangaben stehen in Klammern hinter der Metrik bzw. der Metrikenkategorie. Die Balken zeigen den ermittelten Wert in Prozent, wobei die Färbung eine qualitative Bewertung wiedergibt.

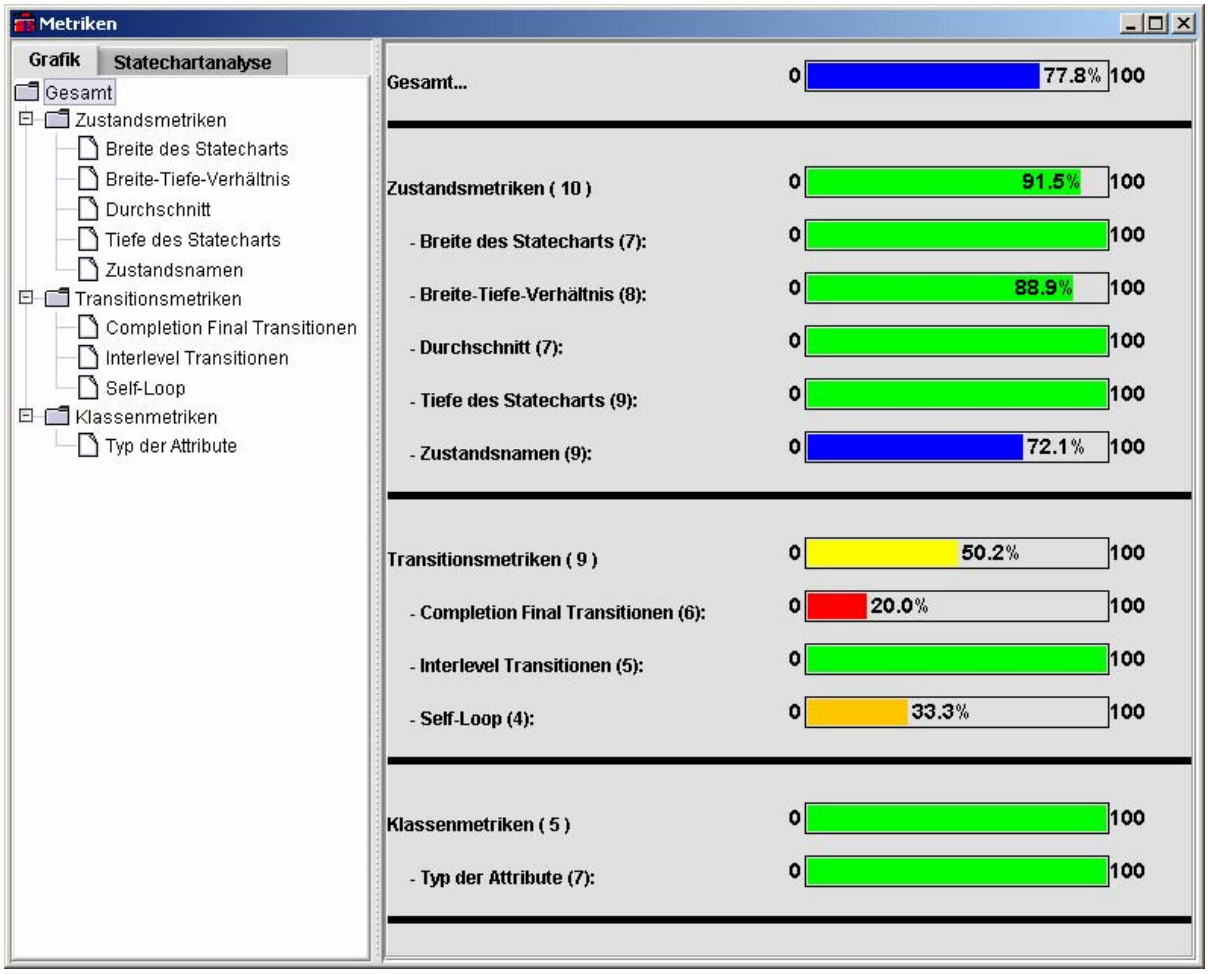


Abbildung 5-8: Ausgabe der Ergebnisse durch Metriken

Zur Berechnung der Gesamtbewertung B_{Modell}, in der obersten Zeile der Abbildung 5-8, wird die nachfolgende Formel aufgestellt:

$$B_{Modell} = \frac{1}{G_{K_{Sum}}} \sum_{j=1}^{K} \left(B_{K_j} * G_{K_j} \right) \quad \text{mit} \quad B_{K_j} = \frac{1}{G_{M_{Sum}}} \sum_{i=1}^{M_j} \left(B_{j,i} * G_{j,i} \right) \quad \text{für} \quad (i = 1,...,K), (j = 1,...,K)$$

Zunächst wird jede Bewertung einer Metrik $B_{j,i}$ mit deren Gewichtung $G_{j,i}/G_{MSum}$ multipliziert und mit den anderen Metrikenbewertungen der gleichen Kategorie aufsummiert, wobei M_j die Anzahl der Metriken in einer Kategorie darstellt. Als Ergebnis erhält man die Bewertung für diese Kategorie $B_{K,i}$. Zur Gesamtbewertung werden anschließend alle Kategorien auf dieselbe Weise berechnet und jeweils mit der dazugehörigen Kategoriengewichtung G_{Kj}/G_{KSum} multipliziert. Das K über dem Summenzeichen stellt die Anzahl der Kategorien dar. Da die Gewichtungssumme aller Kategorien G_{KSum} und die Gewichtungssumme G_{MSum} aller Metriken konstant ist, werden diese aus den jeweiligen Summen herausgezogen.

Kapitel 6

6 Schlussbemerkung

„Dass das Automobil praktisch die Grenzen seiner Entwicklung erreicht hat, wird daran deutlich, dass im vergangenen Jahr keine nennenswerten Verbesserungen erzielt wurden.“

Zeitschrift Scientific American, 2. Januar 1909

In diesem Kapitel wird die Arbeit zusammengefasst und die entwickelte Entwurfsmethodik bewertet. Abschließend erfolgt ein Ausblick über weitere Aktivitäten, die mit der Methodik in Zusammenhang stehen.

6.1 Zusammenfassung

In der vorliegenden Arbeit wurde eine Entwurfsmethodik für eingebettete Systeme im Automobilbereich vorgestellt. Das Ziel dabei war, einen Beitrag zur Verbesserung der SW-Entwicklung im Automotive-Umfeld mittels modellbasierter Techniken und einem durchgängigen Werkzeugeinsatz zu leisten.

Dazu wurde in Kapitel 2 zunächst eine grundlegende Übersicht über die modellbasierte SW-Entwicklung gegeben, wobei insbesondere objektorientierte Methoden und die standardisierte Beschreibungssprache Unified Modeling Language als Basis des methodischen Vorgehens näher vorgestellt wurden. Darüber hinaus erfolgte eine Anforderungsanalyse für CASE Tools, die für einen modellbasierten und werkzeuggestützten Entwicklungsprozess notwendig sind.

Zur Gewährleistung einer disziplinierten Vorgehensweise bei der Erstellung von Steuergeräte-Software, wurde die Entwurfsmethodik stark an das standardisierte V-Modell 97 angelehnt. Dazu wurde in Kapitel 3 mittels des Tailorings eine projektspezifische Anpassung der Konstruktionsphasen des Vorgehensmodells vorgenommen, wobei sowohl eine Auswahl geeigneter Strukturelemente und notwendiger Aktivitäten als auch eine Zuordnung von objektorientierten Elementarmethoden und CASE-Werkzeugen für den gesamten Entwicklungsprozess erfolgte.

Kapitel 4 stellte die Entwurfsmethodik anhand des linken Asts des V-Modells detailliert vor, wobei die Phasen Anforderungsbeschreibung, Analyse, Grob- und Feinentwurf im

Vordergrund der Arbeit standen.

Grundsätzlich umfasst die Entwurfsmethodik folgende Vorgehensweise. In der Anforderungsbeschreibung werden zunächst die Lastenhefte strukturiert und die Benutzeranforderungen durch die UML-Notation präzisiert. Die somit erreichte Funktionsübersicht bildet mit den definierten Szenarien die Basis für den, im V-Modell parallel verlaufenden, Abnahmetest. In der Analysephase wird die logische Systemarchitektur identifiziert. Anschließend werden in der Entwurfsphase mittels der funktionalen Dekomposition die SW-Einheiten mit wohldefinierten Schnittstellen gebildet. Die Detaillierung der SW-Einheiten erfolgt durch Statecharts, mit denen das frühzeitige Simulieren des Systemverhaltes ermöglicht wird. Die Zusammensetzung der einzelnen SW-Einheiten zum Gesamtsystem erfolgt nach dem Bottom-Up Prinzip. Anschließend findet eine Verteilung der Funktionen auf eine logische HW-Architektur statt. Die Integration von Betriebssystemroutinen bildet den abschließenden Schritt zur Realisierung von fahrzeugspezifischen Funktionen. Dieses Vorgehen wurde anhand mehrerer Fallbeispiele aus dem Komfortbereich eines VW Polos erprobt.

Einen großen Stellenwert bei der Definition der Entwurfsmethodik hatte die Qualitätssicherung. Sie wurde zunächst durch konstruktive Maßnahmen, in Form von Modellierungsrichtlinien, realisiert. Mit Hilfe dieser Modellierungsrichtlinien lassen sich sogenannte wellformedness Modelle erstellen, die bestimmte Kriterien, wie Konsistenzsicherung und Einhaltung eines einheitlichen Layouts berücksichtigen. Um die Einhaltung der Regeln und Modellierungsvorschriften gewährleisten zu können, wurde zusätzlich ein prototypisches Werkzeug entwickelt, das automatisch zustandsbasierte Modelle auf Regelverletzungen überprüft. Darüber hinaus werden Qualitätsmerkmale der Modelle durch SW-Metriken qualitativ und quantitativ bewertet. Die Darstellung einiger wichtiger Modellierungsrichtlinien und des Überprüfungswerkzeugs erfolgte in Kapitel 5.

6.2 Bewertung der Entwurfsmethodik

Im Rahmen dieser Arbeit wurde eine modellbasierte Methodik zur Erstellung von Steuergeräte-Software vorgestellt und die Durchführbarkeit dieser Methodik anhand einer Fallstudie aus dem Automobilbereich belegt. Es zeigte sich, dass der Einsatz modellbasierter Techniken bereits in einem frühen Stadium des Entwicklungsprozesses zu einer erheblichen Verbesserung der SW-Qualität führt. So können beispielsweise schwerverständliche Systemanforderungen durch animierbare Modellkomponenten aufgezeigt oder zusammenhängende Informationen grafisch abgebildet werden. Der gesamte SW-Entwicklungsprozess lässt sich außerdem in unterschiedliche Sichten (Kommunikation, Verhalten oder Struktur) gliedern, wobei die Spezifikation stufenweise verfeinert wird. Ebenso lassen sich Verhaltensmodelle schrittweise simulieren, um die ausführbare Spezifikation frühzeitig zu testen. Die geprüften SW-Module können in zukünftigen Systemen verwendet werden, wodurch die Wiederverwendung und damit die Effizienz des Entwicklungsprozesses steigt.

Allerdings bleibt die Frage offen, inwieweit die Entwurfsmethodik auch im industriellen Bereich praktizierbar ist. Diese Frage soll anhand der folgenden Bewertungen einiger Kernpunkte der Entwurfsmethodik diskutiert werden.

SW-Entwicklung mit der Unified Modeling Language

Obwohl die vielen Revisionen der Unified Modeling Language eine Reihe von Problemen in der SW-Entwicklung beseitigt haben, gibt es immer noch Kritik aus Forschung und Industrie [BBBR01, BS02, Glin02a, SG99]. So wird z. B. der Umfang der Sprache kritisiert. Einerseits umfasst allein die Sprachspezifikation der Version 1.5 [OMG03a] über 700 Seiten. Andererseits existiert eine Reihe von Konzepten, die in UML nicht enthalten sind. So wird z. B. die Beschreibung von kontinuierlichem Systemverhalten, die insbesondere für Regelungssysteme notwendig ist, nicht angemessen unterstützt. Die wohl am häufigsten geäußerte Kritik an der UML ist jedoch ihr Mangel an präziser Semantik. Aussagen dazu finden sich in fast allen hier zitierten Arbeiten. Grund für den Mangel an Präzision ist beispielsweise, dass die Semantik der UML hauptsächlich in natürlicher Sprache dargestellt wird. Das Problem einer solchen Darstellung ist nicht nur, dass einzelne Formulierungen mehrdeutig oder missverständlich sein können, sondern auch, dass keine Prüfung auf Vollständigkeit und Schlüssigkeit der Definition als Ganzes möglich ist [Haus01, RW99]. Ein weiteres Problem liegt darin, dass das XMI-Austauschformat ausschließlich strukturelle Informationen der Modelle berücksichtigt. Gänzlich unberücksichtigt bleiben grafische Informationen wie Position, Größe oder Farbe von Diagrammelementen. Darüber hinaus sind die verwendeten Tags der XMI-Datei zur Abbildung der Metainformation größtenteils sehr lang. Beim Erzeugen einer XMI-Datei ergibt sich ein Verhältnis der Metadaten zu den Nutzdaten von 15:1 und führt somit zu einem übermäßig hohen Datenanstieg [Span00].

Trotz der genannten Kritiken überwiegen die Vorteile der Unified Modeling Language als standardisierte Beschreibungssprache zur Erstellung von SW-Systemen (vgl. Kapitel 2.2 auf Seite 19). So wird beispielsweise durch die vorgegebene Notation ein einheitliches Verständnis der Modelle geschaffen und durch die standardisierte SW-Schnittstelle der Informationsaustausch zwischen den Arbeitsgruppen ermöglicht. Im STEP-X Ansatz werden mit Hilfe der UML einerseits die Systemanforderungen grafisch präzisiert und andererseits im weiteren Verlauf des Entwicklungsprozesses die Fahrzeugfunktionen spezifiziert. Ähnlich gute Erfahrungen mit dem Einsatz der UML wurden auch von anderen Projekten im automobilen Umfeld, wie [BBEF+98, BR00, GK01, KM00, LFSK+00] gesammelt, und durch ihre Mächtigkeit, Anpassbarkeit und Wiederverwendbarkeit wird die UML als notwendig für die Verbesserung der Prozesse und der Produktqualität angesehen.

Einen noch größeren Vorteil bei der Entwicklung zukünftiger SW-Systeme erhofft man sich von der neuen UML-Spezifikation 2.0 [OMG02b, OMG03b], die noch in 2005 von der OMG verabschiedet werden soll. Diese neu aufgesetzte Version behebt viele der oben genannten Probleme und erweitert den Standard um weitere Diagrammarten und Konzepte [JRHZ+04]. Das Grundproblem der UML, die umfangreiche Spezifikation detailliert beherrschen zu können, bleibt jedoch weiterhin bestehen und widerspricht dem schnellen Rhythmus der modernen SW-Entwicklung [EK99].

Standardisierte Vorgehensweise

Das V-Modell 97 stellt einen guten Ansatz dar, um anhand eines prozessorientierten Vorgehens Steuergeräte-Software zu entwickeln. Das Vorgehensmodell besitzt eine Anbindung zu objektorientierten Techniken und der UML-Notation. Es ist zudem im Automobilbereich sehr verbreitet, so dass eine Einarbeitung in das V-Modell und dessen Einsatz im Rahmen der Methodik unproblematisch waren.

Allerdings werden, trotz der ausführlichen und gut verständlichen Dokumentation [DW00], nicht alle Aspekte im V-Modell erläutert. So beschränkt sich beispielsweise die Beschreibung der Erzeugnisstruktur weitgehend auf ihre hierarchische Gliederung. Es wird keine klare Definition der Elemente der Erzeugnisstruktur angegeben, so dass Begriffe wie Segment und dessen Granularität undefiniert bleiben. Diese und weitere Definitionsprobleme sind auch in [Lang01] dokumentiert. Ein wesentlicher Kritikpunkt des V-Modells 97 ist auch, dass das Vorgehen durch die Aktivitäten und nicht durch die zu erzeugenden Produkte bestimmt wird [RHV04]. Die vorgegebene Reihenfolge der Aktivitäten zur Erzeugung der Produkte ist zwar in den meisten Fällen sinnvoll. Es kann aber auch vorkommen, dass eine andere Reihenfolge aus organisatorischen Gründen notwendig ist. Es ist nach Abschluss der Entwicklung nicht mehr interessant, in welcher Reihenfolge die Produkte entstanden sind, sondern nur, dass alle notwendigen Produkte erstellt wurden. Diese grundsätzliche Schwäche des derzeitigen V-Modells wurde im V-Modell XT[43] im Februar 2005 behoben. Durch die Modularisierung der Prozesse erhofft man sich zusätzlich eine Verringerung des Aufwands für das Tailoring sowie eine Verbesserung der Anwendbarkeit, Skalierbarkeit und Erweiterbarkeit des V-Modells.

Durchgängigkeit der Werkzeugkette

In der Automobilindustrie wird derzeit eine Vielzahl unterschiedlicher CASE Tools verwendet. So befinden sich z. B. Werkzeuge für Dokumentationsverwaltung, Versions- und Konfigurationsmanagement und diverse Programmierumgebungen im Repertoire eines durchschnittlichen SW-Entwicklers. Darüber hinaus wird die Beherrschung der meisten Microsoft Office™ Produkte, wie Word, Excel, Powerpoint, Project und weiterer, für den Arbeitsbereich spezifischer Werkzeuge vorausgesetzt. Eine zusätzliche Verlängerung der Werkzeugkette, wie für die Umsetzung der STEP-X Methodik erforderlich, kann die Anwender schnell demotivieren und die Effektivität mindern. Ziel muss es daher sein, den Mehrwert der zusätzlichen Werkzeuge aufzuzeigen und eine Möglichkeit zu schaffen, die notwendigen CASE Tools zu erlernen. Bereits heute erhalten die Ingenieure und SW-Entwickler in der Automobilindustrie Werkzeugschulungen und nehmen an Weiterbildungsmaßnahmen im Bereich der SW-Entwicklung teil. Das STEP-X Projekt hat gezeigt, dass sowohl Anwender aus der Industrie als auch aus dem universitären Bereich die Erlernbarkeit der neuen Werkzeuge durch gute Handbücher, interaktive Tutorials und professionellen Support als unproblematisch ansahen.

[43] Seit August 2005 ist das Release 1.1. unter http://www.kbst.bund.de/-,279/V-Modell.htm verfügbar.

Die für die Entwurfsmethodik vorgeschlagenen Werkzeuge sind für die Entwicklung von SW-Systemen im Automobilbereich gut geeignet. Dabei haben die CASE Tools DOORS, MATLAB/Simulink/Stateflow und ASCET-SD ihre Praxistauglichkeit auch schon in anderweitigen Automotive-Projekten bewiesen, wie beispielsweise aus [BRS00, MATH98, FNDR98] ersichtlich ist. Durch die existierenden Schnittstellen zwischen den Modellierungswerkzeugen und DOORS ist ein phasenübergreifendes Requirements Management gegeben. Ebenso wird durch die Kopplung der Modellierungswerkzeuge eine Co-Simulation in frühen Phasen der SW-Entwicklung ermöglicht. Dies gilt auch für den eigens entworfenen Regel Checker, mit dem eine werkzeugunabhängige Überprüfung der zustandsbasierten Modelle gewährleistet wird.

Jedoch ist ein Austausch von Modellen zwischen den Entwicklungswerkzeugen nicht möglich, da die Werkzeuge auf grundsätzlich unterschiedlichen Beschreibungssprachen basieren. Dadurch kann ein Export von Klassenmodellen in Blockdiagramme nicht erfolgen. Eine beispielhafte Modelltransformation von Rhapsody nach ASCET-SD im Rahmen von STEP-X ergab in [Proc03], dass eine zufrieden stellende Lösung einen Mehraufwand zur Folge hätte, der innerhalb des STEP-X Projekts nicht realisierbar wäre. Darüber hinaus zeigten die ersten Ergebnisse der Modelltransformation, dass die transformierten Modelle durch die unterschiedliche Ausführungssemantik der Statecharts einen vollkommen anderen Aufbau hatten, so dass ein Vergleich zwischen Originalmodell und dem Resultat nur schwer möglich war. Des Weiteren würde eine künftige Änderung oder Erweiterung der Werkzeuge einen großen Anpassungsaufwand zur Folge haben. Daher erscheint die Verfolgung dieser Strategie aus Sicht des Autors für den praxisrelevanten Einsatz nicht sinnvoll. Viel versprechende Ansätze sind die Entwicklung einer einheitlichen SW-Schnittstelle und die Erstellung von Standard-Software-Modulen. Im Automobilbereich wird dieses beispielsweise von der *Hersteller Initiative Software* [LBB+01], MSR-MEGMA [WM99] und inzwischen verstärkt durch AUTOSAR [Scha05] verfolgt.

Eine weitere Möglichkeit wäre, zunächst nur die Strukturinformationen in UML abzubilden und das Verhalten erst zu einem späteren Zeitpunkt im Entwicklungsprozess zu modellieren. Dieser Ansatz wird beispielsweise bei dem Automobilhersteller BMW verfolgt, wobei die UML für die Modellierung von Strukturen und Kommunikation Verwendung findet und das Verhalten zu einem späteren Zeitpunkt mit ASCET-SD modelliert wird. Dies hat jedoch den Nachteil, dass die Validierung und Verifikation erst am Ende der SW-Entwicklung möglich ist. DaimlerChrysler verzichtet gänzlich auf einen UML-gestützten Ansatz, nachdem das mehrjährige TITUS-Projekt [TITUS00, Lang01] gestoppt worden ist. Stattdessen wird die ganzheitliche Modellierung mit der MATLAB Toolbox durchgeführt. Nachteile hierbei sind allerdings, dass keine objektorientierten Methoden verwendet werden und ebenso keine Entwicklung mit Hilfe verschiedener UML-Sichten möglich ist. Darüber hinaus ist die SW-Entwicklung von einem einzigen Werkzeughersteller abhängig.

Betrachtet man die eben genannten Argumente, so ist aus Sicht von STEP-X zu verstehen, warum der Einsatz der UML auf diese Art und Weise erfolgt ist, auch wenn zurzeit keine Möglichkeit existiert, die Modelle aus den frühen Phasen in die späteren Phasen

werkzeugübergreifend und automatisch zu überführen. Aus Sicht der Methodik ist dies allerdings kein Problem, da mit Hilfe der UML die Entwicklung eines ausführbaren Lastenhefts im Vordergrund steht. Auf Basis dieses Digitalen Lastenhefts kann die Weiterentwicklung vom Zulieferer oder durch die OEMs erfolgen. In beiden Fällen steht dabei eine verständliche, vollständige und korrekte Spezifikation des zu entwickelnden Systems an erster Stelle.

Kurzfristig wird die Entwicklung der Steuergeräte-Software weiterhin durch Zulieferer erfolgen, die somit qualitativ hochwertige Lastenhefte erhalten. Mittelfristig erlangen die Automobilhersteller Know-how über die Entwicklung von Fahrzeugfunktionen und deren Verknüpfung mit den dazugehörigen Lastenheften. Langfristig wird durch die Weiterentwicklung der UML-Werkzeuge auch eine Code-Generierung für Steuergeräte ermöglicht, die sich für den Einsatz von Serien-Code eignet. Telelogic und andere UML-Werkzeughersteller haben bereits in 2002 gezeigt, dass eine prototypische Code-Generierung für Steuergeräte möglich ist.

Qualitätssicherung durch konstruktive und analytische Maßnahmen

Um einen möglichst effektiven Entwicklungsprozess zu erreichen, muss die Qualität der entwickelten Modelle sichergestellt werden. Hierfür wurden sowohl konstruktive als auch analytische Verfahren zur Sicherung der SW-Qualität in die Entwurfsmethodik integriert. Die Definition von Modellierungsrichtlinien, als konstruktive Maßnahme, ist eine Voraussetzung für eine disziplinierte Modellierungsweise und wurde in den letzten Jahren seitens der Automobilhersteller und Zulieferer intensiv verfolgt [MDHKK05]. Die hier vorgestellten Richtlinien, denen vor allem zustandsbasierte Modellen zu Grunde liegen, eignen sich zur Sicherstellung der Konsistenz und Werkzeugkompatibilität, zur Vereinheitlichung des Layouts sowie weiterer Designentscheidungen. Es konnte festgestellt werden, dass diese Maßnahmen zur Verbesserung des modellbasierten Entwicklungsprozesses beitragen, indem einerseits das gemeinsame Modellverständnis die Zusammenarbeit zwischen Projektmitgliedern verbessert und andererseits die Wiederverwendbarkeit von Modellkomponenten erhöht wurde. Denn nur durch ein gemeinsames Verständnis der abgebildeten Inhalte und kompatible SW-Schnittstellen kann eine systematische Wiederverwendung von SW-Komponenten erfolgen. Das prototypische Analysewerkzeug Regel Checker erlaubt, als analytische Maßnahme eine vollautomatische Überprüfung der zustandsbasierten Modelle anhand der zuvor definierten Richtlinien.

Um die Effizienz der Modellierungsrichtlinien zu überprüfen und den Einsatz eines statischen Analysewerkzeugs zu rechtfertigen, wurde eine Evaluierungsstudie durchgeführt. Anhand des Fallbeispiels Fensterheber wurden 14 Modelle ausgewertet. Dabei wurden die Modellierer in zwei Kategorien unterschieden. In der Ersten konnten gute Modellierungskenntnisse aufgewiesen werden, bei der Zweiten hatten die Probanden nur wenig oder gar keine Modellierungserfahrungen. Darüber hinaus wurde eine weitere Aufteilung in den beiden Kategorien vorgenommen. Eine Gruppe erhielt nur grobe Modellierungsvorgaben, wie Lesbarkeit und Verständlichkeit, und die andere Gruppe erhielt den vollständigen Regelsatz. Die Auswertung der erstellten Modelle erfolgte mittels SW-Metriken,

die mit dem Regel Checker erstellt und analysiert wurden. Als Ergebnis konnten zwei grundsätzliche Aussagen getroffen werden. Erstens konnte durch Verwendung der Richtlinien der Modellierungsstil verbessert werden. Zweitens führte die Verwendung des Richtlinienkatalogs allein nicht zum gewünschten Optimum der Qualität, was daraus ersichtlich wurde, dass der Regel Checker weitere Regelverletzungen aufdeckte. So wurden beispielsweise unnötige Variablen definiert, unerlaubte Modellkonstrukte verwendet und nur wenige Optimierungsmaßnahmen bezüglich der Modellstruktur vorgenommen. In Tabelle 6-1 sind die Ergebnisse der Evaluierung zusammengefasst. Es wird ersichtlich, dass die Verwendung des Regelkataloges die Modellqualität erheblich erhöht, das Analyseergebnis aber trotzdem noch suboptimal bleibt. Dies rechtfertigt die Nutzung eines statischen Analyseprogramms, wie beispielsweise den Regel Checker.

Modellierungs-erfahrungen	**Regeln vorhanden**	**Metriken in Prozent**				**Gesamt-bewertung**
		Z	T	L	S	
vorhanden	teilweise	62	53	57	62	59 %
	vollständig	87	86	92	96	91 %
keine/wenig	teilweise	49	47	51	43	48 %
	vollständig	73	79	87	78	79 %

Z = Zustandsmetriken, T = Transitionsmetriken, L = Layoutmetriken, S = Sonstige Metriken

Tabelle 6-1: Auswertung der Evaluierungsstudie

Zusammenfassend lässt sich feststellen, dass der Mehraufwand, der sich bei der Anwendung der vorgeschlagenen Vorgehensweise gegenüber konventionellen Methoden ergibt und zunächst in den Prozess investiert werden muss, sich rasch durch eine deutliche Fehlerreduktion im Systementwurf und gute Wiederverwendbarkeit der erstellten Modelle amortisiert. Dass sich die Verfolgung eines modellbasierten Ansatzes zur Beherrschung der SW-Komplexität in der Automobilindustrie als richtig und vernünftig gezeigt hat, wird durch derzeit laufende Forschungsarbeiten bestätigt. Beispielsweise fördert das Bundesministerium für Bildung und Forschung im Rahmen der Forschungsoffensive *Software-Engineering 2006* die Projekte IMMOS (Integrierte Methodik zur modellbasierten Steuergeräteentwicklung), AutoMoDe (Modellbasierte Entwicklung verteilter Steuergerätesoftware im Automobilsektor) und SiLEST (Entwicklung und Erprobung von Methoden und Werkzeugen für den Test und die Sicherheitsanalyse eingebetteter Echtzeitsoftware).

6.3 Ausblick

Die hier vorgestellte Methodik zur Entwicklung von Steuergeräte-Software ist für sich gesehen vollständig und abgeschlossen. Die Bestätigung des Verfahrens erfolgte bereits durch die Übernahme von Konzepten und Teilergebnissen der frühen Phasen des Entwicklungsprozesses in die ersten Fachabteilungen des industriellen Projektpartners. Allerdings lässt sich der Modellierungsprozess durch die massive Weiterentwicklung von Techniken und Beschreibungsmitteln noch deutlich verbessern. So verspricht beispielsweise der Wechsel von UML 1.5 auf die neuere Version 2.0 einige Vereinfachungen bei der Systemspezifikation und eröffnet gleichzeitig neue Möglichkeiten der SW-Erstellung.

Zur Realisierung dieses Wechsels müssen zunächst eine Evaluierung der neuen UML-Spezifikation sowie eine Schätzung des Aufwands und Nutzens für die Neustrukturierung erfolgen. In diesem Zusammenhang sollte ebenfalls das von der OMG neu definierte UML-Erweiterungskonzept *Systems Modeling Language*[44] (SysML) evaluiert werden, mit dem das Systems Engineering und nicht, wie bei UML, das Software Engineering im Fokus steht. So soll damit z. B. die Präzisierung der textuellen Anforderungen durch neue Diagramme vereinfacht werden. Der Standard wird nach Aussage der OMG noch in diesem Jahr verabschiedet und ist bereits teilweise in der Version 5.0 des UML-Werkzeugs Artisan RtS integriert.

Ein weiterer Schwerpunkt zukünftiger Arbeiten wird der Ausbau des inzwischen von Volkswagen zum Patent angemeldet Regel Checkers [Patent05] sein, da man die Notwendigkeit von Modellierungsrichtlinien [MDHKK05] sowie deren Überprüfung [Mutz04a, Mutz05] erkannt hat. Außerdem gestattet das Konzept des Regel Checkers nicht nur die Definition und Analyse von Modellierungsrichtlinien und SW-Metriken, sondern erlaubt darüber hinaus die Anbindung weiterer Applikationen. So wurde beispielsweise in [Beit04, Firl04] ein Slicing-Algorithmus für Statecharts implementiert. Dieser hat zum Ziel die Zustandsmenge so zu reduzieren, dass auch größere Statecharts in möglichst kurzer Zeit durch Model Checking verifiziert werden können.

In anderen Arbeiten [Flor03, FMH04, Kaya04] wurde ein Statechart-Simulator realisiert, mit dem eine benutzerspezifische Semantik konfiguriert und anschließend simuliert werden kann. Dadurch lässt sich nicht nur eine statische Überprüfung [HMDF+04], sondern auch eine dynamische Überprüfung der Modelle durchführen, indem z. B. die Modelle im Hinblick auf Deadlocks überprüft werden können. Die genannten Ansätze existieren derzeit als Prototypen für einen eingeschränkten Bereich. Um sie tatsächlich auch in die Praxis einzuführen, ist zusätzlicher Programmieraufwand nötig.

[44] Vgl. http://www.sysml.org

ANHANG A - Tailoring

Die im Rahmen des Tailoring gestrichenen Aktivitäten werden durch die Markierung „—“ in der Spalte TT gekennzeichnet. Die Markierung „❍“ symbolisiert, dass diese Aktivität in der Methodik Verwendung findet. Das „☆“ stellt dar, dass es sich bei dieser Aktivität um eine Maßnahme handelt, die unbedingt nach der ISO 9001 einzuhalten ist und dadurch in die Methodik einbezogen werden muss. In der zweiten Spalte sind die Streichbegründungen und die eingesetzten Elementarmethoden aufgelistet. Die dritte Spalte zeigt den Verantwortlichkeitsbereich.

TT	Aktivität/Begründung der Streichung	Verantwortung
	SE 1 Systemanforderungsanalyse	
—	SE 1.1 Ist-Aufnahme/Analyse durchführen Der Ist-Zustand hat keinen Einfluss auf das Reengineering.	Volkswagen AG
☆	SE 1.2 Anwendungssystem beschreiben	AG Digitales Lastenheft /Volkswagen AG
—	SE 1.3 Kritikalität und Anforderungen an die Qualität definieren Die Kritikalitätsbetrachtung entfällt für dieses System, da es sich aufgrund der Vorgaben als unkritisch einstufen lässt. Neben der Funktionalität des zu entwickelnden Systems sind keine weiteren Qualitätsanforderungen relevant.	Volkswagen AG
—	SE 1.4 Randbedingungen definieren Technische und organisatorische Randbedingungen kommen von extern.	Volkswagen AG
❍	SE 1.5 System fachlich strukturieren	AG Digitales Lastenheft /Volkswagen AG
—	SE 1.6 Bedrohung und Risiko analysieren Der Bedrohungsaspekt ist nicht relevant; bei Fehlverhalten des Systems werden nur geringe Risiken erwartet.	Volkswagen AG
—	SE 1.7 Forderungscontrolling durchführen Die Forderungen sind wirtschaftlich realisierbar. Es werden keine komplexen Funktionen gefordert.	Volkswagen AG
—	SE 1.8 Softwarepflege- und Änderungskonzept erstellen Entfällt, weil keine besonderen Wartbarkeitsanforderungen gestellt sind.	---
	SE 2 Systementwurf	
☆	SE 2.1 System technisch entwerfen	AG Digitales Lastenheft
—	SE 2.2 Wirksamkeitsanalyse durchführen Sicherheitsaspekte sind nicht relevant.	---
—	SE 2.3 Realisierbarkeit untersuchen Die Realisierbarkeit wurde bereits durch fremde Vorstudien ermittelt.	AG Digitales Lastenheft
—	SE 2.4 Anwenderforderungen zuordnen Ergibt sich aus der Beschreibung der Aktivität SE 2.1.	---
❍	SE 2.5 Schnittstellen beschreiben	AG Digitales Lastenheft
❍	SE 2.6 Systemintegration spezifizieren	AG Digitales Lastenheft
	SE 3 Anforderungsanalyse für SW-Einheiten	
—	SE 3.1 Allgemeine Anforderungen aus Sicht der SW-Einheit definieren Allgemeine Anforderungen sind bereits in Aktivität SE 2.5 ausreichend beschrieben.	---
—	SE 3.2 Anforderungen an externe Schnittstellen präzisieren Die Schnittstellen sind bereits in Aktivität SE 2.5 ausreichend beschrieben.	---
❍	SE 3.3 Anforderungen an die Funktionalität definieren	AG Digitales Lastenheft
☆	SE 3.4 Anforderungen an die Qualität der SW-Einheit definieren	AG Digitales Lastenheft

TT	Aktivität/Begründung der Streichung	Verantwortung
—	SE 3.5 Anforderungen an Entwicklungsumgebung definieren Entfällt, da diese Anforderungen von den Systemanforderungen nicht abweichen.	---
	SE 4 SW-Grobentwurf	
☆	SE 4.1 SW-Architektur entwerfen	AG Digitales Lastenheft
—	SE 4.2 SW-interne Schnittstellen entwerfen Entfällt, da die Schnittstellen bereits in den Aktivitäten SE 2.5 und SE 4.1 ausreichend beschrieben sind.	---
—	SE 4.3 SW-Integration spezifizieren Die Integration ist aus der Aktivität 4.1 ableitbar und benötigt keine weiteren Erläuterungen.	---
	SE 5 SW-Feinentwurf	
❍	SE 5.1 SW-Modul/SW-Komponenten beschreiben Wird auch von anderen Projektteilnehmern bearbeitet.	AG Digitales Lastenheft / AG Bussysteme
❍	SE 5.2 Betriebsmittel und Zeitbedarf analysieren Wird von anderen Projektteilnehmern durchgeführt.	AG Bussysteme
	SE 6 SW-Implementierung	
☆	SE 6.1 SW-Module kodieren	AG Bussysteme
—	SE 6.2 Datenbank realisieren Es liegt keine Datenbank zugrunde.	---
☆	SE 6.3 Selbstprüfung des SW-Moduls durchführen Wird auch von anderen Projektteilnehmern durchgeführt.	AG Digitales Lastenheft / AG Test
	SE 7 SW-Integration	
❍	SE 7.1 Zur SW-Komponente integrieren Wird auch von anderen Projektteilnehmern durchgeführt.	AG Digitales Lastenheft / AG Toolkopplung
❍	SE 7.2 Selbstprüfung der SW-Komponente durchführen Wird von anderen Projektteilnehmern durchgeführt.	AG Test
☆	SE 7.3 Zur SW-Einheit integrieren Wird auch von anderen Projektteilnehmern durchgeführt.	AG Digitales Lastenheft / AG Toolkopplung
❍	SE 7.4 Selbstprüfung der SW-Einheit durchführen Wird von anderen Projektteilnehmern durchgeführt.	AG Test
	SE 8 Systemintegration	
☆	SE 8.1 Zum System integrieren Wird auch von anderen Projektteilnehmern durchgeführt.	AG Digitales Lastenheft /Test
❍	SE 8.2 Selbstprüfung des Systems durchführen Wird von anderen Projektteilnehmern durchgeführt.	AG Test
❍	SE 8.3 Produkt bereitstellen Wird von anderen Projektteilnehmern durchgeführt.	AG Test
	SE 9 Überleitung in die Nutzung	
—	SE 9.1 Beitrag zur Einführungsunterstützung leisten Eine Einführungsunterstützung ist nicht erforderlich.	---
❍	SE 9.2 System installieren Wird von anderen Projektteilnehmern durchgeführt.	AG Bussysteme
❍	SE 9.3 In Betrieb nehmen Wird von anderen Projektteilnehmern durchgeführt.	AG Bussysteme

Tabelle 6-2: Streichungen im Submodell SE

ANHANG B – Kriterienkatalog

Kategorie	Kriterien	Gewichtung	Gewichtung (in %)
Herstellerbewertung	Herstellerrenommee	1	2 (0,81%)
	Erfahrung im Automotive Bereich		5 (2,02%)
	Aufwärtskompatibilität von Toolversionen		5 (2,02%)
	Support		5 (2,02%)
Beschreibungsmittel	Verhaltensdiagramme (z. B. Statecharts)	3	9 (3,64%)
	Beschreibung für analoges (hybr.) Systemverhalten		6 (2,43%)
	Strukturdiagramme (z. B. Klassendiagramme)		8 (3,24%)
	Interaktionsdiagramme (z. B. Sequenzdiagramme)		6 (2,43%)
Offenheit	Integration von Teilen anderer Toolketten	2	3 (1,21%)
	Standardisierte Schnittstellen		3 (1,21%)
	Schnittstelle zu ASCET-SD		6 (2,43%)
	Schnittstelle zu Diagnose-Tools		6 (2,43%)
	Schnittstelle zu MATLAB/Simulink		7 (2,83%)
Übergänge zwischen den Phasen	Requirements(DOORS) -> Analyse	4	9 (3,64%)
	Analyse -> Design		9 (3,64%)
	Design -> Implementierung		9 (3,64%)
Testen	Tracing zwischen Phasen	3	6 (2,43%)
	Tracing zwischen System- u. Komponentenebene		4 (1,62%)
	Vorgabe von wiederkehrenden Strukturen		7 (2,83%)
	Testspezifikation auf formaler Analyse		2 (0,81%)
Codeerzeugung	Erzeugung von C-Code	5	10 (4,05%)
	Codequalität		10 (4,05%)
	OSEK		7 (2,83%)
	Unterstützung verschiedener Zielhardware		5 (2,02%)
	Lesbarkeit des Codes		4 (1,62%)
	Verteilung, Generierung d. Kommunikation		7 (2,83%)
	Einbindung von Legacy Code		9 (3,64%)
	Rückführung von händischen Änderungen		4 (1,62%)
Ergonomie	Handhabbarkeit der Werkzeugkette	2	7 (2,83%)
	Graphische Darstellung		4 (1,62%)
	automatische Dokumentation		4 (1,62%)
	Modellnavigation		6 (2,43%)
Weitere Anforderungen	Konsistenzprüfung der Modelle	5	9 (3,64%)
	Wiederverwendbarkeit der Teilmodelle		10 (4,05%)
	Simulation auf PC		10 (4,05%)
	Animation		4 (1,62%)
	Versionsmanagement		5 (2,02%)
	Co-Simulation (Debugging von Modellen)		7 (2,83%)
	Multi-User Eignung		8 (3,24%)

Tabelle 6-3: Kriterienkatalog für die Evaluierung von Modellierungswerkzeugen

ANHANG C – Modellierungsregeln

Die nachfolgende Tabelle enthält alle im Regel Checker implementierten Modellierungsregeln. Diese sind zur besseren Übersicht in die fünf Kategorien Design-, Kompatibilitäts-, Konsistenz-, Layout- und Optimierungsregeln aufgegliedert. Unterschiedliche Beanstandungstypen kennzeichnen die Wichtigkeit der auftretenden Beanstandungen. Dabei bedeutet = Fehler, = Warnung und = Vorschlag.

Nr.	Name	Typ	Kurzbeschreibung
			Designregeln
1	CheckGuard		Ein Guard muss syntaktisch korrekt sein und darf keine Seiteneffekte, wie z. B. [a>b && b>c && c<a] haben .
2	CheckTrigger		Überprüft, ob Transitionen über einen Event, Guard, ActionGuard und/oder eine Action verfügen.
3	ClassAttributes		Klassenattribute sollten mit einem Grossbuchstaben beginnen.
4	CompletionTransition		Completion Transitions sollten nur im Zusammenhang mit final state verwendet werden.
5	ConstantNotWrite		Konstanten dürfen nicht überschrieben werden. Sie sind durch Großbuchstaben zu kennzeichnen.
6	DefaultStateName		Der Zustandsname sollte explizit vergeben werden.
7	Description		Beschreibungsfelder für das Projekt und einzelne Statecharts sind auszufüllen.
8	Determinismus		Transitionen aus einem Zustand dürfen nicht identische Triggerbedingung haben.
9	EqualNames		Zustände sollten sich namentlich unterscheiden (mit und ohne Berücksichtigung der Groß- und Kleinschreibung).
10	IllegalDenotations		Zur Bildung von Zustandsnamen wird nur auf einen reduzierten Zeichenvorrat zurückgegriffen. Dabei wird zwischen Anfangs- und Folgezeichen unterschieden.
11	InitVariables		Jede deklarierte Variable ist am Anfang des Statecharts (Init-Transition oder entry-Aktion) zu initialisieren.
12	InOutTransCount		Die Anzahl der ein- und ausgehenden Transitionen eines (Pseudo)States sollte nicht über- oder unterschritten werden.
13	MultiPseudoStates		Die Anzahl der hintereinander stehenden Pseudozustände ist zu begrenzen.
14	StateCount		Anzahl der Zustände pro Zustand bzw. Projekt begrenzen.
15	StateNameLength		Aussagekräftige Zustandsnamen sind zu bevorzugen. Dies impliziert eine Minimallänge für die Vergabe von Zustandsbezeichnungen. Es muss jedoch die Möglichkeit zur Definition von Ausnahmen gegeben sein.
16	IsolatedStates		Alle Zustände müssen betreten und verlassen werden können.
17	MiracleStates		Basiszustände, die zwar eine ausgehende, aber keine eingehende Transition besitzen sind zu vermeiden.

Nr.	Name	Typ	Kurzbeschreibung
Kompatibilitätsregeln			
18	ActionMethod		Vermeidung von Methodenaufrufen in Aktionen.
19	OrthogonalStates		Orthogonale Regionen dürfen nur horizontal/vertikal angeordnet werden.
20	RestrictPseudoStates		Bestimmte Pseudostates sind zu vermeiden.
21	RestrictRegions		Die Anzahl der Regionen sollte pro Statechart beschränkt werden.
22	RestrictVariableTypes		Bestimmte Variabeltypen sind zu vermeiden.
Konsistenzregeln			
23	CompositeState		Ein Composite Zustand sollte mindestens einen Basiszustand bzw. zwei orthogonale Regionen enthalten.
24	InitialState		Ein Statechart muss einen Initialzustand pro Level bzw. pro Region besitzen, falls keine Transition von außen direkt hineinführt.
25	OnlyeOneElse		Wird ELSE in einer Bedingung verwendet, muss es mindestens eine weitere Transition geben, die den Zustand verlässt.
26	RegionChangeNeedsInit		Wenn eine Transition von einer Region innerhalb eines AND-States in eine andere Region führt, muss die Quellregion einen Initial Zustand besitzen.
27-57	UMLCheck		Es werden alle well-formedness rules der OMG [OMG03a] aufgenommen.
Layoutregeln			
58	NailNumber		Transitionen sollten nicht zu viele Zwischenpunkte haben.
59	OnlyHorizontVertcal-Trans		Transitionen sollten entweder nur horizontal bzw. vertikal verlaufen.
60	StartPosition		Der Initialzustand sollte möglichst links oben anfangen.
61	StateApproaching		Zustände in der gleichen Ebene sollten einen Mindestabstand zueinander einhalten.
62	StateIntersection		Überschneidungen von Zuständen sind unzulässig.
63	StatesPerLevel		Die Anzahl von Unterzuständen pro Zustand (offene Hierarchie) ist zu begrenzen.
64	TransIntersection		Transitionen sollten sich nicht schneiden. Sollte dies doch der Fall sein, so muss dies in einem möglichst großen Winkel geschehen.
65	TransLength		Transitionen sind möglichst kurz zu zeichnen.
66	TransStateIntersection		Transitionen dürfen keine Zustände kreuzen.
Optimierungsregeln			
67	HistoryStateCheck		Verwendung eines Deep-History-Konnektors ohne dass es tiefe Hierarchien gibt.
68	InnerTrans		Besitzt ein Zustand keine Entry-/Exit-Aktionen, so sind Self-Loops durch innere Transitionen zu ersetzen.
69	OrthogonalEmpty		Leere orthogonale Komponenten sind zu vermeiden.
70	ReduceRegion		Eine orthogonale Region mit nur einem Basiszustand ohne interne Aktionen kann evtl. entfallen.
71	UselessCompositeState		Ein zusammengesetzter Zustand, der nur einen Unterzustand besitzt, kann evtl. eingespart werden.

Tabelle 6-4: Modellierungsregeln im Regel Checker

ANHANG D – Studien- und Diplomarbeiten

Folgende von mir betreute Studien- und Diplomarbeiten haben zur vorliegenden Arbeit beigetragen.

M. Bauhan. Entwurf und Implementierung eines Layout-Algorithmus zur Darstellung von Statecharts. Studienarbeit, TU Braunschweig, 2004.

G. Beitzen-Heinecke. Umsetzung von Eigenschaftserhaltenden Vereinfachungen von Statecharts, Studienarbeit, TU Braunschweig, 2004.

K. Chkhihvadze. Erstellung von Modellierungsrichtlinien für das UML-Tool Artisan. Studienarbeit, TU Braunschweig, 2004.

M. Dettmer. Modellierung und Implementierung eines Fensterhebers mit ASCET-SD. Studienarbeit, TU Braunschweig, 2003.

B. Florentz. Entwicklung und Implementierung einer modularen Statechartsemantik. Diplomarbeit, TU Braunschweig, 2003.

M. Kaya. Entwurf eines UML Statechart - Editors mit Java. Studienarbeit, TU Braunschweig, 2004.

C. Knieke. Evaluierung von Rhapsody in Java anhand einer Fallstudie aus dem Kfz-Komfortbereich. Studienarbeit, TU Braunschweig, 2003.

N. von Köckritz. Anpassung der ASCET-SD Struktur an den Designregel Checker. Studienarbeit, TU Braunschweig, 2003.

S. Nienaber. Konzeption und Realisierung einer Schnittstelle zwischen DOORS und Matlab zur automatisierten Testskriptgenerierung. Studienarbeit, TU Braunschweig, 2004.

S. Pietsch: Automatische Überprüfung von UML-Statecharts anhand definierbarer Design-Regeln. Diplomarbeit, TU Braunschweig, 2003.

J. Steiner. Entwurf und Implementierung von UML-Strukturmodellen für Komfortfunktionen im Kfz mit Artisan. Studienarbeit, TU Braunschweig, 2002.

T. Werner. Evaluierung von Artisan RtS anhand der STEP-X Methodik. Studienarbeit, TU Braunschweig, 2003.

ANHANG E – Eigene Veröffentlichungen

[BHHM03] G. Bikker, M. Harms, M. Horstmann, M. Mutz. Methodik und Werkzeugkette für den modellbasierten Entwicklungsprozess im Automobilbereich. GzVB 2003, Automatisierungs- und Assistenzsysteme für Transportmittel, S.134-144, Braunschweig, 2003.

[EHHM+03] T. Ehlers, M. Harms, M. Horstmann, M. Mutz, J.-U. Varchmin. Strukturierter Entwicklungsprozess für Automotive Anwendungen. 23. Tagung 'Elektronik im Automobil', Haus der Technik, Stuttgart, 2003.

[FMH04] B. Florentz, M. Mutz, M. Huhn. Avoiding Unpredicted Behaviour of Large Scale Embedded Systems by Design and Application of Modelling Rules. Specification Implementation and Validation of Embedded Systems - Model Design and Validation, Workshop at the 15th IEEE Intern. Symposium of Software Reliability Engineering, Saint-Malo, 2004.

[HBHM05] M. Huhn, J. Braam, M. Harms, M. Horstmann, M. Mutz. Structured Hardware/Software Development for Enhanced Quality and Safety in Automotive Systems. In: 6th Conference on Automation, Assistance and Embedded Real Time Platforms for Transportation, Braunschweig, 2005.

[HHKM03] M. Harms, M. Horstmann, C. Krömke, M. Mutz. STEP-X: Strukturierter Entwicklungsprozess für eingebettete Systeme im Automobilbereich. Veröffentlichung in Beilage Automotive Electronics 1/03 zur ATZ, MTZ und Automotive Engineering Partners, 2003.

[HHM03] M. Huhn, P. Hofmann, M. Mutz. Digitale Lastenhefte für die Softwareentwicklung vernetzter Steuergeräte. 23. Tagung 'Elektronik im Automobil', Haus der Technik, Stuttgart, 2003.

[HMDF+04] M. Huhn, M. Mutz, K. Diethers, B. Florentz, M. Daginnus. Applications of Static Analysis on UML Models in the Automotive Domain. FORMS/ FORMAT 2004, Braunschweig, 2004.

[HMF04] M. Huhn, M. Mutz, B. Florentz. A Lightweight Approach to Critical Embedded Systems Design using UML. Critical Systems Development with UML, Lissabon, 2004.

[MD02] M. Mutz, K. Diethers. Improving Software Development in the Automotive Area through Tool Supported Modelling and Formal Analysis. SDA 2002, System Design Automation, S.81-88, Pirna, 2002.

[MDHKK05] M. Mutz, M. Daginnus, P. Hofmann, T. Klein, H. Kleinwechter. Ein Richtlinienkatalog für die Erstellung von Simulink/Stateflow-Modellen im Automobilbereich. 3. Workshop: Automotive Software Engineering, Bonn, 2005.

[MH03] M. Mutz, M. Huhn. Automated Statechart Analysis for User-defined Design Rules. Technischer Bericht 2003-10, TU Braunschweig, 2003.

[MHGK03] M. Mutz, M. Huhn, U. Goltz, C. Krömke. Model Based System Development in Automotive. SAE 2003, Detroit, USA, 2003.

[MHHH+03] M. Mutz, M. Harms, M. Horstmann, M. Huhn, G. Bikker, C. Krömke, K. Lange, U. Goltz, E. Schnieder, J.-U. Varchmin. Ein durchgehender modellbasierter Entwicklungsprozess für elektronische Systeme im Automobil. Elektronik im Kraftfahrzeug, VDI-Berichte 1789, Baden-Baden, 2003.

[MP03] M. Mutz, S. Pietsch. Verbesserung des modellbasierten Softwareentwurfs im Automobilbereich durch einen Design-Regel Checker. 23. Tagung 'Elektronik im Automobil', Haus der Technik, Stuttgart, 2003

[Mutz03] M. Mutz. Ein Regel Checker zur Verbesserung des modellbasierten Softwareentwurfs. EKA 2003, Entwurf komplexer Automatisierungssysteme, S. 411 - 423, Braunschweig, 2003.

[Mutz04a] M. Mutz. Metriken und Regeln für eine durchgängige und modellbasierte SW-Entwicklung im Automobilbereich. 2. Workshop: Automotive Software Engineering, Ulm, 2004.

[Mutz04b] M. Mutz. Metriken für zustandsbasierte Software-Entwicklung. Metrics News, Journal of the SW Metrics Community, Volume 9, Number 2, ISSN 1431-8008, 2004.

[Mutz05] M. Mutz. Metriken und Regeln für eine zustandsbasierte SW-Entwicklung im Automobilbereich. In: Informatik – Forschung und Entwicklung, Band 19, Heft 4, S. 206 - 212, 2005.

[VHHH+04] J.-U. Varchmin, M. Harms, M. Horstmann, M. Huhn, M. Mutz, G. Bikker, C. Krömke, K. Lange, E. Schnieder. Strukturierter Entwicklungsprozess für eingebettete Systeme - vom Lastenheft zur automatischen Codegenerierung. Tagungsband des EUROFORUM Elektronik-Systeme im Automobil, 2004.

[Patent05] M. Mutz, U. Goltz, S. Prochnow, S. Pietsch, C. Krömke. Patentanmeldung auf Basis der DE 103 03 379.3 „Verfahren und Vorrichtung zur Weiterverarbeitung von Daten eines Zustandsautomaten“, Dezember 2003, Anmeldenummer 0278363001DE, Patentnummer DE 10358289 A1, Offenlegung 2005.

Literaturverzeichnis

[ABRW04] A. Angermann, M. Beuschel, M. Rau, U. Wohlfarth. Matlab-Simulink-Stateflow. 3. Auflage, Oldenbourg, ISBN 3-486-27602-6, 2004.

[Adam02] A. Adam. SVG - Scalable Vector Graphics. Franzis' Verlag, Poing, ISBN 3-7723-6190-0, 2002.

[AGM03] C. Angerer, M. Glaser, C. Merenda. Tool ARTiSAN Real-time Studio. White paper, 2003.

[AGSB+98] B. Aumann, W.-D. Gruhle, M. Sieger, O. Buchhold, F. König. Einsatz objektorientierter Methoden zur Entwicklung von Echtzeitsoftware. In: VDI-Berichte 1415, S. 435 - 461, 1998.

[AKZ96] M. Awad, J. Kusela, J. Ziegler. OCTOPUS: Object-Oriented Technology for Real-Time Systems. Prentice Hall, 1996.

[Ambl03] S. W. Ambler. The Elements of UML Style. Cambridge University Press, 2003.

[AU250] Entwicklungsstandard für IT-Systeme des Bundes. Vorgehensmodell. Allgemeiner Umdruck 250, Bundesministerium des Inneren, Bonn, 1997.

[AU251] Entwicklungsstandard für IT-Systeme des Bundes. Methodenzuordnung. Allgemeiner Umdruck 251, Bundesministerium des Inneren, Bonn, 1997.

[AU252] Entwicklungsstandard für IT-Systeme des Bundes. Funktionale Werkzeuganforderungen. Allgemeiner Umdruck 252, Bundesministerium des Inneren, Bonn, 1997.

[Balz96] H. Balzert. Methoden der objektorientierten Systemanalyse. 2. Auflage, Spektrum Akademischer Verlag, Heidelberg, 1996.

[Balz97] H. Balzert. Wie erstellt man ein objektorientiertes Analysemodell?. Springer Verlag, 1997.

[Balz01] H. Balzert. Lehrbuch der Software-Technik - Software-Entwicklung. 2. Auflage, Spektrum Akademischer Verlag, ISBN 3-8274-0480-0, 2001.

[Bauh04] M. Bauhan. Entwurf und Implementierung eines Layout-Algorithmus zur Darstellung von Statecharts. Studienarbeit, TU Braunschweig, 2004.

[BBA02] A. L. Baroni, S. Braz, F. B. e Abreu. Using OCL to Formalize Object-Oriented Design Metrics Definitions. In: Proc. of QUAOOSE'2002, Malaga, Spanien, 2002.

[BBBD04] R. Buschermöhle, M. Brörkens, I. Brückner, W. Damm, W. Hasselbring, B. Josko, C. Schulte, T. Wolf. Model Checking - Grundlagen und Praxiserfahrungen. Informatik Spektrum, (2), 2004.

[BBBR01] M. Broy, M. von der Beeck, P. Braun, M. Rappl. A fundamental critique of the UML for the specification of embedded systems. unpublished, 2001.

[BBCP+03] P. Braun, M. Broy, M. V. Cengarle, J. Philipps, W. Prenninger, A. Pretschner, M. Rappl, R. Sandner. The Automotive CASE. In: B. Westfechtel, M. Nagl. Modelle, Werkzeuge, Infrastrukturen zur Unterstützung von Entwicklungsprozessen, Seite 211-228, 2003.

[BBEF+98] M. Broy, B. Brügge, J. Eberspächer, G. Färber, G. Reinhart, H.-J. Schneider, H. Wildemann, M. Wirsing. FORSOFT Bayerischer Forschungsverbund Software-Engineering, Jahresbericht, 1998.

[BBHS96] D. Baumgarten, O. Burnus, C. Hoffmann, F. Schröder. Das dezentrale Komfortelektronik - System bei VW. 7. VDI-Tagung Elektronik im Kraftfahrzeug, Baden-Baden, 1996.

[BBK98] M. Broy, M. von der Beeck, I. Krüger. Softbed: Problemanalyse für ein Großverbundprojekt. Universität München, 1998.

[BCR02] E. Börger, A. Cavarra, E. Riccobene. A precise semantics of UML State Machines: making semantic variation points and ambiguities explicit. SFEDL02 - ETAPS 2002, 2002.

[BD00] B. Brügge, A. Dutoit. Object-Oriented Software Engineering. Prentice-Hall, 2000.

[BDEH+97] U. Brockmeyer, W. Damm, M. Eckrich, H.-J. Holberg, G. Wittich. Einsatz formaler Methoden zur Erhöhung der Sicherheit eingebetteter Systeme im KFZ. VDI/VW Gemeinschaftstagung Systemengineering in der Kfz-Entwicklung, VDI Berichte 1374, S. 349 - 366, 1997.

[BDEM+96] M. Broy, W. Damm, M. Eckrich, W. Mala, G. Venzl. Korrekte Software für sicherheitskritische Systeme. Das Projekt KORSYS im Überblick. In: Statusseminar des BMBF - Softwaretechnologie, Projektträger Informationstechnik des BMBF bei der DLR e. V., S. 337 - 346, Berlin, 1996.

[Beec94] M. von der Beeck. A comparison of statecharts variants. In: H. Langmaack, W. de Roever, J. Vytopil, editors, Formal Techniques in Real-Time and Fault-Tolerant Systems. Volume 863 of Lecture Notes in Computer Science, S. 128 - 148. Springer Verlag, Berlin, 1994.

[Beec01] M. von der Beeck. Formalization of UML-Statecharts. In: Lecture notes in computer science, Springer Verlag, S. 406 - 421, 2001.

[BEHH+03] G. Bikker, T. Ehlers, M. Harms, M. Horstmann, M. Huhn, M. Mutz. Auswahl der Entwicklungswerkzeuge - Evaluierungsbericht, unveröffentlicht, Interner Bericht, TU Braunschweig, Version 1.0, 2003.

[Beit04] G. Beitzen-Heinecke. Umsetzung von Eigenschaftserhaltenden Vereinfachungen von Statecharts, Studienarbeit, TU Braunschweig, 2004.

[BGJ99] S. Berner, M. Glinz, S. Joos. A Classification of Stereotypes for Object-Oriented Modeling Languages, Beitrag für den 7. Workshop des GI-Arbeitskreises GROOM der GI-Fachgruppe 2.1.9, Universität Koblenz-Landau, 1999.

[BJMF03] M. Boger, M. Jeckle, S. Mueller, J. Fransson. Diagram Interchange for UML. In: UML 2002, The Unified Modeling Language, Springer Verlag, 2003.

[BJR96] G. Booch, I. Jacobson, J. Rumbaugh. Unified Method for Object-Oriented Development. Rational Software Corporation, Addendum, Version 0.91, 1996.

[BK02] C. Bunse, A. von Knethen. Vorgehensmodelle kompakt. Spektrum Akademischer Verlag, Heidelberg, ISBN 3-8274-1203-X, 2002.

[BLLPY96] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, W. Yi. UPPAAL- A Tool Suite for the Automatic Verification of Real-Time Systems. In: Proceedings of Hybrid Systems III. Lecture Notes in Computer Science 1066, S. 232 - 243. Springer Verlag, 1996.

[BM96] L. B. Basili, W. Melo. A validation of object oriented design metrics as quality indicators. IEEE Transactions on Software Engineering, Vol.22, Nr. 10, S. 751 - 761, 1996.

[Böhm81] B. W. Boehm. Software Engineering Economics. Prentice Hall PTR, 1981.

[Booc94] G. Booch. Object-Oriented Analysis and Design with Applications. 2nd ed. The Benjamin/Cummings Publishing Company, Inc., 1994.

[Bort94] J. Bortolazzi. Untersuchungen zur rechnergestützten Erfassung, Verwaltung und Prüfung von Anforderungsspezifikationen und Einsatzbedingungen elektronischer Steuerungs- und Regelungssysteme. Dissertation, Universität Erlangen, 2004.

[BR00] P. Braun, M. Rappl. Model based Systems Engineering - A Unified Approach using UML. In: 2nd European Systems Engineering Conference, Herbert Utz Verlag, München, 2000.

[BR01] D. Buck, A. Rau. On Modelling Guidelines: Flowchart Patterns for Stateflow. In: Gesellschaft für Informatik, FG 2.1.1: Softwaretechnik Trends, ISSN 0720-8928, 2001.

[Brau97] R. Brause. Betriebssysteme – Grundlagen und Konzepte. Springer Verlag, ISBN 3-540-62929-7, 1997.

[BRJ00] G. Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language User Guide: The ultimate tutorial to the UML from the original designers. 6. Auflage, Addison-Wesley, 2000.

[Broy97] M. Broy. Requirements Engineering for embedded systems. In: Proc. FemSys'97, 1997.

[Broy01] M. Broy. Trends bei der Software im Automobil. 5. EUROFORUM Konferenz Software im Automobil, Stuttgart, 2001.

[BRS00] P. Braun, M. Rappl, J. Schüffele. Softwareentwicklung für Steuergerätenetzwerke - Eine Methodik für die frühe Phase. VDI-Berichte. Nr. 1547, 2000.

[BS02] M. Broy, J. Siedersleben. Objektorientierte Programmierung und Softwareentwicklung - Eine kritische Einschätzung. Informatik Spektrum, Band 25, Heft 1, S. 3 - 11, 2002.

[Bued00] D. M. Buede. The Engineering Design of Systems. A Wiley-Interscience Publication, 2000.

[Burk98] R. Burkhardt. Objektorientierte Modellierung: Der methodische Einsatz der Objekttechnologie, Habilitation an der TU Ilmenau, 1998.

[Burn97] O. Burnus. Einführung ins Komfortsystem Türsteuergeräte. Anforderungsbeschreibung - Elektrische Fensterheber, Version 2.0, internes Dokument der Volkswagen AG, 1997.

[Burn01] O. Burnus. Festlegung der umzusetzenden Funktionen in den Türsteuergeräten PQ24. internes Dokument, Volkswagen, 2001.

[Buro01] D. Burow. Tolerance Based Simulation & Automated FMEA generation. 4. Annual Systems Engineering Conference, Dallas, 2001.

[Cast02] R. Castello. A Framework for the Static and Interactive Visualization of Statecharts. In: Journal of Graphical Algorithms & Applications, 6 (3), S. 313 - 351, 2002.

[CHB92] D. Coleman, F. Hayes, S. Bear. Introducing Objectcharts or how to Use Statecharts in Object-Oriented Design. IEEE Transactions on Software Engineering 18(1), 1992.

[Chkh04] K. Chkhihvadze. Erstellung von Modellierungsrichtlinien für das UML-Tool Artisan. Studienarbeit, TU Braunschweig, 2004.

[CIS00] K. Compton, J. Illuggins, W. A. Shen. A Semantic Model for the State Machine in the Unified Modelling Language. In: UML 2000 Workshop, Dynamic Behaviour in UML Models: Semantic Questions, S. 25 - 31, 2000.

[CMT02] R. Castello, R. Mili, I. G. Tollis. Automatic Layout of Statecharts. Software Practice and Experience, 2002.

[CWM98] M. Conrad, M. Weber, O. Müller. Towards a Methodology for the Design of Hybrid Systems in Automotive Electronics. Daimler-Benz AG,1998.

[CY91] P. Coad, E. Yourdon. Object Oriented Analysis. Yourdon Press Computing Series, 1991.

[Davi93] A. M. Davis. Software Requirements: Objects, Functions & States. Prentice Hall, 1993.

[DeMa78] T. DeMarco. Structured Analysis and System Specification. Yourdan Press, New York, 1978.

[Dett03] M. Dettmer. Modellierung und Implementierung eines Fensterhebers mit ASCET-SD. Studienarbeit, TU Braunschweig, 2003.

[DHO01] B. Demuth, H. Hussmann, S. Obermaier. Experiments with XMI based Transformations of Software Models. Proceedings of the Workshop on Transformations in UML, 2001.

[DJVP03] W. Damm, B. Josko, A. Votintseva, A. Pnueli. A formal semantics for a UML kernel language. Technical Report IST/33522/WP 1.1/D1.1.2-Part1, Verimag, 2003.

[DKKM+02] C. Denger, D. Kerkow, A. von Knethen, M. Mora, B. Paech. Quasar - Von Use Cases zu Statecharts in 7 Schritten. IESE-Report 086.02/D, Kaiserslautern, 2002.

[DKKP03] C. Denger, Antje von Knethen, D. Kerkow, B. Paech: Beschreibung elektronischer Steuergeräte mit Use Cases und Statecharts. 8. Fachtagung Entwurf komplexer Automatisierungssysteme, 2003.

[Doug98] Bruce Powel Douglass. Real-Time UML - Developing Efficient Objects for Embedded Systems. Addison-Wesley, 1998.

[Dürr01] I. Dürrbaum. Objektorientierte Softwareentwicklung für Steuergeräte im Kraftfahrzeug am Beispiel der Funktionen einer Scheibenwischer- und Außenbeleuchtungssteuerung. Studienarbeit, Universität Duisburg, 2001.

[DW98] D. D'Souza, A. Wills. Objects, Components and Frameworks with UML: The Catalysis Approach. Addison-Wesley, 1998.

[DW00] W. Dröschel, M. Wiemers. Das V-Modell 97 - Der Standard für die Entwicklung von IT-Systemen mit Anleitung für den Praxiseinsatz. Oldenbourg Verlag, 2000.

[Eder02] K. Eder. Modellbasierte Entwicklung moderner Bediensysteme im Fahrzeug. Zeitschrift Automotive, Nr. 2, 2002.

[EFK01] M. Eiglsperger, S. P. Fekete, G. W. Klau. Drawing Graphs: Methods and Models. Springer Verlag, 2001.

[EFS01] M. Erben, J. Fetzer, H. Schelling. Software-Komponenten - Ein neuer Trend in der Automobilelektronik. Zeitschrift ATZ/MTZ-Special Automotive Electronics, 2001.

[Eich02] H. Eichelberger. Evaluation-Report on the Layout Facilities of UML Tools. Report No. 298, Universität Würzburg, 2002.

[EK99] A. Evans, S. J. Kent. Core Meta Modelling Semantics of the UML: the UML Approach. In: UML'99 - The Unified Modeling Language, Lecture Notes in

Computer Science 1723, Springer Verlag, 1999.

[EP98] H.-E. Eriksson, M. Penker. UML-Toolkit. Wiley Computer Publishing, ISBN 0-471-19161-2, 1998.

[ETAS00] ETAS. ASCET-SD User Guide 4.0. ETAS GmbH, Dokument EC010001 R4.0.1 EN, Stuttgart, 2002.

[EW00] R. Eshuis, R. Wieringa. Requirements-level Semantics for UML Statecharts. In: Formal Methods for Open Object-based Distributed Systems, S. 121 - 140, 2000.

[Fent91] N. E. Fenton. Software Metrics: A Rigorous Approach. Chapman and Hall, 1991.

[FHV04] K. Fischer, G. Hordys, B. Vogel-Heuser. Evaluation of an UML Software Engineering Tool by Means of a Distributed Real Time Application in Process Automation. In: Lecture Notes in Informatics, Modellierung 2004, S.135 - 148, Marburg, 2004.

[Firl04] T. Firley. Computing Abstract Models for Verifying Reactive Systems. Dissertationsschrift, TU Braunschweig, Shaker Verlag, ISBN 3-8322-4335-62004, 2005.

[Flor03] B. Florentz. Entwicklung und Implementierung einer modularen Statechartsemantik. Diplomarbeit, TU Braunschweig, 2003.

[FM01] G. Frick, K. D. Müller-Glaser. Information Management Concepts for Multi-Tool Modeling. In: Proceedings of the IASTED International Conference Modelling, Identification, and Control, Innsbruck, Österreich, S. 803 - 806, 2001.

[FNDR98] M. Fuchs, D. Nazareth, D. Daniel, B. Rumpe. BMW-ROOM - An Object-Oriented Method for ASCET, In: SAE 98 - Society of Automotive Engineers, Detroit, USA 1998.

[FORD00] FORD. Structured Analysis & Design Using Matlab/Simulink/Stateflow. In: SmartVehicle Challenge Problems, http://vehicle.me.berkeley.edu/mobies/, 2000.

[Fowl99] M. Fowler. Analysemuster. Addison-Wesley, Bonn, 1999.

[GHMP02] U. Goltz. M. Huhn, M. Mutz, B. Prammer. Methoden zur Anforderungserfassung. unveröffentlicht, Interner Bericht, Version 1.0, TU Braunschweig, 2002.

[GK01] M. Götze, W. Kattanek. Experiences with the UML in the Design of Automotive ECUs. In: Design, Automation and Test in Europe, München, 2001.

[Glin95] M. Glinz. An Integrated Formal Model of Scenarios Based on Statecharts. In: Lecture Notes In Computer Science, Vol. 989, Proceedings of the 5th Euro-

pean Software Engineering Conference, S. 254 - 271, London, UK, 1995.

[Glin00] M. Glinz. A Lightweight Approach to Consistency of Scenarios and Class Models. In: Proceedings of the 4th International Conference on Requirements Engineering, S. 49, Washington, DC, USA, 2000.

[Glin02a] M. Glinz. Problems and Deficiencies of UML as a Requirements Specification Language. In: International Workshop on Software Specifications & Design, Proceedings of the 10th International Workshop on Software Specification and Design, S. 11, 2000.

[Glin02b] M. Glinz. Statecharts for Requirements Specification - as Simple As Possible - As Rich As Needed. ICSE International Workshop on Scenarios and State Machines: Models, Algorithms and Tools, Orlando, USA, 2002.

[Glin04] M. Glinz. Software Engineering I. Vorlesungsskript, WS 04/05, Universität Zürich, 2004.

[GR00] B. Gebhard, M. Rappl. Requirements Management for Automotive Systems Development. SAE Technical Paper Series 2000-01-0716, Detroit, USA, 2000.

[GR02] M. Gogolla, M. Richters. Using the UML Specification Environment USE. http://www.db.informatik.uni-bremen.de/projects/USE/, 2002.

[Gres01] P. Gresch. Einfluss steigender Komplexität neuer Systeme auf den Entwicklungsprozess eines Automobilherstellers. EUROFORUM-Konferenz Software im Automobil, Stuttgart, 2001.

[Grim03] K. Grimm. Software technology in an automotive company - major chalenges. In: Proceedings of the 25th International IEEE Conference on Software Engineering, 2003.

[Hare84] D. Harel. Statecharts: A Visual Approach to Complex Systems. Technical Report CS84-05. The Weizmann Institute of Science, Israel, 1994.

[Hare87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 8(3), S. 231 - 274, 1987.

[Hare02] D. Harel. Rhapsody: A complete livecycle model-based development System. In: Proc. 3rd Int. Conf. on Integrated Formal Methods S. 1 - 10, 2002.

[Haus01] J. H. Hausmann. Dynamische Metamodellierung zur Spezifikation einer operationalen Semantik von UML. Diplomarbeit, Universitär Paderborn, 2001.

[HDK93] P. Hsia, A. M. Davis, D. C. Kung. Status Report: Requirements Engineering. IEEE Software, 10 (6), S. 75 - 79, 1993.

[Heim01] K. Heimannsfeld. Modellbasierte Anforderung in der Produkt- und Systementwicklung: von Dokumenten zu Modellen. Dissertationsschrift, Aachen, 2001.

[HEVB03] M. Harms, T. Ehlers, J.-U. Varchmin, A. Breuer. Diagnose im strukturierten Entwicklungsprozess STEP-X. Haus der Technik, Stuttgart, 23. Tagung Elektronik im Kraftfahrzeug, 2003.

[HG96] D. Harel, E. Gery. Executable Object Modeling with statecharts. In: Proceedings of 18th Int. Conf. On Software Engineering, S. 246 - 257, Berlin, 1996.

[HHK04] M. Harms, M. Horstmann, S. Kuhler. Einbettung von Test und Diagnose in den Entwicklungsprozess für Automotive Anwendungen. Haus der Technik, 2. Tagung - Simulation und Test in der Funktions- und Softwareentwicklung für die Automobilelektronik, Berlin, 2004.

[HHM01] B. Hnatkowska, Z. Huzar, J. Magott. Consistency Checking in UML Models. 4th Int. Conf. on Information Systems, Modeling ISM'01, 2001.

[HJD02] E. Hull, K. Jackson, J. Dick. Requirements Engineering. Springer Verlag, 2002.

[HK99] M. Hitz, G. Kappel. UML@Work - von der Analyse zur Realisierung. dpunkt Verlag, ISBN 3-932588-38-X, 1999.

[HLNP+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. IEEE Transactions on Software Engineering 16(4), 1990.

[HMFP+03] A. Heinrich, K. Müller, J. Fehrling, A. Paggel, I. Schneider. Versionsmanagement für Transparenz und Prozesssicherheit in der Steuergeräte-Entwicklung. VDI Berichte Nr. 1789, 2003.

[HMP04] J. Hooman, N. Mulyar, L. Posta. Validating UML models of Embedded Systems by Coupling Tools. In: Proceedings Workshop on Specification and Validation of UML models for Real-Time and Embedded Systems, 2004.

[HN96] D. Harel, A. Naamad. The STATEMATE semantics of statecharts. In: ACM Transactions on Software Engineering Methods, 5(4), 1996.

[HN99] W. Hermsen, K. J. Neumann. OMOS - Objektorientierte Modellierung von Embedded-Software. In: OMER-Workshop - Object-oriented Modeling of Embedded Real-Time Systems, Herrsching am Ammersee, 1999.

[Hoff98] J. Hoffmann. MATLAB und SIMULINK. ISBN 3-8273-1077-6, Addison-Wesley, 1998.

[Hofm02] P. M. Hofmann. Einsatz von CASE Tools in der Automobilindustrie. unveröffentlicht, Interner Bericht, Volkswagen, 2002.

[Hörf02] S. Hörfarter. S.P.E.E.D - Seamless Process for Efficient and Economic Development. Whitepaper, Berner & Mattner Systemtechnik GmbH, 2002.

[Hosa04] A. Hosagrahara. The Modeling Metric Tool Users Guide. The MathWorks,

Inc., 2004.

[HP85] D. Harel, A. Pnueli. On the Development of Reactive Systems. In: K. R. Apt. editor, Logics and Models of Concurrent Systems, volume F-13 of NATO ASI, S. 477 - 498, New York, USA, 1985.

[HP96] D. Harel, M. Politi. Modeling Reactive Systems with Statecharts: The STATEMATE Approach. Technical Report, I-Logix, Inc., 3 Riverside Drive, Andover, MA 01810, USA, 1996.

[HPSS87] D. Harel, A. Pnueli, J. P. Schmidt, R. Sherman. On the Formal Semantics of Statecharts. In: Proc. of IEEE Symposium on Logic in Computer Science, S. 54 - 64, 1987.

[HR04] G. Hamon, J. Rushby. An Operational Semantics for Stateflow. In: Lecture Notes in Computer Science 2984, 2004.

[Hrus98] P. Hruschka. Ein pragmatisches Vorgehensmodell für die UML. In: OBJEKTspektrum 2/98, S. 34 - 45, 1998.

[Hunt03] J. Hunt. Guide to the Unified Process - featuring UML, Java and Design Patterns, Springer Verlag, ISBN 1-85233-721-4, 2003.

[HY99] D. Harel, G. Yashchin. An Algorithm for Blob Hierarchy Layout, Institute of Science, Rehovot, Israel, 1999.

[IEEE91a] IEEE830. IEEE Guide to Software Requirements Specifications, In Software Engineering Standards Collection, New York, 1991.

[IEEE91b] IEEE730. IEEE Standard for Software Quality Assurance Plans, In Software Engineering Standards Collection, New York, 1991.

[Ivan99] E. Ivanov. Eine Methodik für die Entwicklung und Anwendung von objektorientierten Frameworks. Dissertationsschrift, TU Ilmenau, 1999.

[Jack95] M. Jackson. Requirements and Specifications: A Lexicon of Practice, Principles, and Prejudices. Addison-Wesley, 1995.

[JBAG97] S. Joos, S. Berner, M. Arnold, M. Glinz. Hierarchische Zerlegung in objektorientierten Spezifikationsmodellen. In: Softwaretechnik-Trends, Band 17, Heft 1, ISSN 0720-8928, 1997.

[JBR99] I. Jacobson, G. Booch, J. Rumbaugh. The Unified Software Development Process. Addison-Wesley, ISBN 0-201-57169-2, 1999.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard. Object-Oriented Software Engineering - A Use Case Driven Approach. Addison-Wesley, 1992.

[JEJ02] Y. Jin, R. Esser, J. W. Janneck. Describing the Syntax and Semantics of UML Statecharts in a Heterogeneous Modelling Environment, 2002.

[JH02] S. Jerala, M. Holl. Use-Case in der Praxis, OBJEKTspektrum, 29. Ausgabe,

2002.

[Jin98] M. Jin. Practical Rules for Reduction on the Number of States of a State Diagram. Proceedings of the Technology of Object-Oriented Languages and Systems, 1998.

[John01] S. John. Transition Selection Algorithms for Statecharts, In Informatik 2001: Wirtschaft und Wissenschaft in der Network Econcomy - Visionen und Wirklichkeit; Proc. GI/OCG Jahrestagung, Vol. 1, S. 622 - 627, Wien, Österreich, 2001.

[John03] S. John. Steps for statecharts - a tool-based, comparative study. Technischer Bericht 2003-4, TU Berlin, Fakultät IV, 2003.

[Joos00] S. Joos. ADORA-L - Eine Modellierungssprache zur Spezifikation von Software-Anforderungen. Dissertationsschrift, Universität Zürich, 2000.

[JRHZ+04] M. Jeckle, C. Rupp, J. Hanh, B. Zengler, S. Queins. UML 2 – glasklar. Hanser-Verlag, 2004.

[JW96] D. Jackson, J. Wing. Lightweight Formal Methods. IEEE Computer, 29 (4), 1996.

[Katt97] W. Kattanek. Anforderungen an FSM-Tools beim hierarchischen Entwurf paralleler digitaler Steuerungen. 42. IWK, Universität Ilmenau, 1997.

[Kaya04] M. Kaya. Entwurf eines UML Statechart - Editors mit Java. Studienarbeit, TU Braunschweig, 2004.

[KB02] H. Kim, C. Boldyreff. Developing Software Metrics Applicable to UML Models. Computer science department, City University London, UK, 2002.

[Kien97] U. Kiencke. Ereignisdiskrete Systeme. Oldenbourg Verlag, ISBN 3-486-24150-8, München, 1997.

[Kirc01] L. Kirchner. Entwicklung und Anwendung eines Bezugrahmens zur Evaluierung von UML-Modellierungswerkzeugen. Diplomarbeit, Universität Koblenz-Landau, 2001.

[KJ02] U. Kiencke, H. Jäkel. Signale und Systeme. Oldenbourg Verlag, 2. Auflage, ISBN 3-486-25959-8, 2002.

[KF03] L. Kirchner, U. Frank. Evaluierung von UML-Modellierungswerkzeugen. OBJEKTspektrum 01/03, 2003.

[KKPS99] E. Kamsties, A. von Knethen, J. Philipps, B. Schätz. Eine vergleichende Fallstudie mit CASE-Werkzeugen für objektorientierte und funktionale Modellierungstechniken. In: OMER1- Object-oriented Modeling of Embedded Real-Time Systems, UniBW München, 1999.

[Klos03] T. Kloss. Flexibles und Automatisiertes Layout von Statecharts. Studienarbeit, Christian-Albrechts Universität zu Kiel, 2003.

[KM00] A. von Knethen, J. Münch. Entwicklung eingebetteter Software mit UML - Der Do-it-Prozess. Version 1.0, SFB 5001, 2000.

[KMP01] V. Knaup, W. Müller, T. Pfeffer. Software Versions- und Konfigurations-Management. 10. Internationaler Kongress Elektronik, VDI, 2001.

[Knet01] A. von Knethen. Change-oriented requirements traceability support for evolution of embedded systems. Dissertationsschrift, Universität Kaiserslautern, 2001.

[Knie03] C. Knieke. Evaluierung von Rhapsody in Java anhand einer Fallstudie aus dem Kfz-Komfortbereich. Studienarbeit, TU Braunschweig, 2003.

[Kobr99] C. Kobryn. UML 2001: A Standardization Odyssey. Communications of the ACM, 42 (10), S. 29 - 37, 1999.

[Köck03] N. von Köckritz. Anpassung der ASCET-SD Struktur an den Designregel Checker. Studienarbeit, TU Braunschweig, 2003.

[Köhl01] U. Köhlke. Funktionsbeschreibung des Komfortsteuergeräts PQ24 / T5. Version 1.0, internes Dokument der Volkswagen AG, 2001.

[Koss00] K. Kossowan. Automatisierte Überprüfung semantischer Modellierungsrichtlinien für Statecharts. Diplomarbeit, Technische Universität, 2000.

[Krep01] T. Kreppold. Richtlinie Modellierung mit Statemate MAGNUM und Rhapsody in Micro C. Berner & Mattner Systemtechnik GmbH, Version 1.4, Ottobrunn, 2001.

[KTSC01] L. Köster, T. Thomsen, R. Stracke. Connecting Simulink to OSEK/VDX, Automatic Code Generation for Real-Time Operating Systems with Target-Link. SAE Technischer Bericht 2001-01-0024, USA, 2001.

[Kücü04] Ferit Kücükay. Assistenzsysteme in der Fahrzeugtechnik. In: 5. Braunschweiger Symposium, Automatisierungs- und Assistenzsysteme für Transportmittel, Braunschweig, 2004.

[Kühl01] M. Kühl. Eine durchgehende Entwicklungsmethodik für das Rapid Prototyping von Eingebetteten Systemen. Workshop im Rahmen der DFG-Schwerpunktprogramme, Modelltransformation und Werkzeugkopplung, Braunschweig, 2001.

[Lang01] T. Lange. Einsatz des Unified Software Development Process zur Entwicklung von Kfz-Steuerungssoftware. Diplomarbeit, TU Berlin, 2001.

[LBB+01] K. Lange, J. Bortolazzzi, P. Brangs, et al. Herstellerinitiative Software. 10. VDI-Tagung Elektronik im Kraftfahrzeug, Baden-Baden, 2001, VDI-Berichte 1646, VDI-Verlag, Düsseldorf, 2001.

[Leve95] N. Leveson. Safeware - System Safety and Computers. Addison-Wesley, ISBN 0-20-111972-2, 1995.

[LFSK+00] A. Lapp, P. T. Flores, J. Schirmer, D. Kraft, W. Hermsen, T. Bertram, J. Petersen. CARTRONIC-Domänenarchitektur auf Basis funktionaler Anforderungen in UML. In: Architektur-Workshop FORSOFT: Architektur eingebetteter Systeme, München, 2000.

[Lude93] J. Ludewig. Sprachen für das Software Engineering. Informatik-Spektrum, 16, S. 286 - 294, 1993.

[Lude97] J. Ludewig. Software Engineering: Vorläufiges Skript zur Vorlesung Software Engineering. Fakultät Informatik, Universität Stuttgart, 1997.

[LW00] E. Lehmann, J. Wegener. Test Case Design by Means of the CTE XL. Proceedings of the 8th European International Conference on Software Testing, Kopenhagen, Denmark, 2000.

[MATH98] MATHWORKS. Using simulink and stateflow in automotive applications. Technischer Bericht, https://tagteamdbserver.mathworks.com/ttserverroot /Download/559_9521v00_autoex_all.pdf, 1998.

[MATH01] MATHWORKS. Controller Style Guidelines For Production Intent Using MATLAB, Simulink and Stateflow. MathWorks Automotive Advisory Board, Version 1.0, 2001.

[MISRA95] Motor Industry Research Association (MISRA). Software Metrics. The Motor Industry Software Reliability Association, Report 5, Version 1.0, MIRA, 1995.

[MISRA04a] Motor Industry Research Association (MISRA). Development Guidelines For Vehicle Based Software. The Motor Industry Software Reliability Association, Version 1.1, MIRA, 0-952-415-607, 1994.

[MISRA04b] Motor Industry Research Association (MISRA). Guidelines For thr Use Of The C Language in Vehicle Based Software. Version 2, 2004.

[MC00] A. Moore, N. Cooling. Real-Time Perspective. ARTiSAN Whitepaper, 2000.

[Melc00] R. Melchisedech. Verwaltung und Prüfung natürlichsprachlicher Spezifikationen. Dissertationsschrift, Universität Stuttgart, 2000.

[MFW97] A. Mölders, W. Fengler, M. Wolf. Dynamic Object Oriented Modelling Using an Object Process Model and a Method for Verification. Object Oriented Petri Nets, 1997.

[Midd99] S. Middendorf. XML und Java. OBJEKTspektrum 5/99, 1999.

[MK98] S. Mann, M. Klar. A metamodel for object-oriented statecharts. 2. Workshop on Rigorous Object Oriented Methods, ROOM 2, 1998.

[MKF00] M. Moutos, A. Korn, C. Fisel. Guideline-Checker. Studienarbeit, FH Esslingen, 2000.

[MN00] F. Maier, F. Necker. Implementierung eines Programms zur Analyse der Komplexität von MATLAB/Simulink-Modellen. Studienarbeit, Fachhoch-

schule Essling, 2000.

[MPT03] A. Maggiolo-Schettini, A. Peron, S. Tini. A Comparison of Statecharts Step Semantics. Theoretical Computer Science Archive, 290 (1), S. 465 - 498, ISSN 0304-3975, 2003.

[MS00] K.-D. Müller-Glaser, E. Sax. COMTESSA - Einsatz der CASE-Technologie im durchgängigen Software.Entwurfsprozess von Steuergeräten, Technischer Bericht, Karlsruhe, 2000.

[Müll98] G. Müller-Ettrich. System Development with V-Model and UML. In: The Unified Modeling Language, Technical Aspects and Applications, Physica-Verlag, 1998.

[Müll99a] G. Müller-Ettrich. Objektorientierte Prozessmodelle. Addison-Wesley, 1999.

[Müll99b] A. Müller. Entwurfsmethodik und automatisierte Verteilung für Steuerungssoftware in einem verteilten Rechnersystem in der Automobilelektronik. Dissertationsschrift, Der Andere Verlag, 1999.

[Mutz97] P. Mutzel. Algorithmen zum automatischen Zeichnen von Graphen. im Rahmen des DFG-Schwerpunkts Effiziente Algorithmen für diskrete Probleme und ihre Anwendungen, GI Jahrestagung, S. 58 - 67, 1997.

[Mutz01] P. Mutzel. Automatic Layout and Labelling of State Diagrams. In: Mathematics - Key Technology for the Future, Springer Verlag, Berlin, 2001.

[NHF98] M. Nüttgens, M. Hoffmann, T. Feld. Objektorientierte Systementwicklung mit der Unified Modeling Language. Universität Saarbrücken, 1998.

[Nien04] S. Nienaber. Konzeption und Realisierung einer Schnittstelle zwischen DOORS und Matlab zur automatisierten Testskriptgenerierung. Studienarbeit, TU Braunschweig, 2004.

[NS99] J. Noack, B. Schienmann. Objektorientierte Vorgehensmodelle im Vergleich. In: Informatik-Spektrum 22, S. 166 - 180, 1999.

[OBJ02] OBJECT by DESIGN. Choosing a UML Modeling Tool. http://www.objectsbydesign.com, 2002.

[Oest98] B. Oestereich. Objektorientierte Softwareentwicklung - Analyse und Design mit der Unified Modeling Language. 4. Auflage, Oldenbourg Verlag, 1998.

[Oest99] B. Oestereich. Wie setzt man Use Cases wirklich sinnvoll zur Anforderungsanalyse ein?. OBJEKTspektrum 1/99, 1999.

[OHE98] R. Orfali, D. Harkey, J. Edwards. Instant CORBA. Addison-Wesley, 1998.

[OMG97a] Object Management Group. Unified Modeling Language v 1.0. OMG document ad/97-01-14, 1997.

[OMG97b] Object Management Group. Unified Modeling Language v 1.1. OMG document ad/97-08-11, 1997.

[OMG99a] Object Management Group. Request for information on UML 2.0 and responses, 1999.

[OMG99b] Object Management Group. OMG Unified Modeling Language Specification 1.3. 1999.

[OMG01a] Object Management Group. Action semantics for the UML. http://www.uml-actionsemantics.org, 2001.

[OMG01b] Object Management Group. OMG Unified Modeling Language Specification 1.4. 2001.

[OMG02a] Object Management Group. UML Profile for Schedulability, Performance and Time. 2002.

[OMG02b] Object Management Group. UML 2.0 Superstructure Specification. Version 2.0, 2002.

[OMG03a] Object Management Group. OMG Unified Modeling Language Specification. Version 1.5, 2003.

[OMG03b] Object Management Group. UML 2.0 Infrastructure Specification. Version 2.0, 2003.

[OMG03c] Object Management Group. UML 2.0 Diagram Interchange. Version 1.0, 2003.

[OSEK01] OSEK/VDX. Binding Specification. Version 1.3, 2001.

[Otto98] Christian Otto. Ist die UML bereit für eingebettete Echtzeitsysteme? In: OBJEKTspektrum 4/98, S. 16 - 22, 1998.

[Page88] M. Page-Jones. The Practical Guide to Structured System Design. Prentice Hall, Englewood Cliffs, New York, 1988.

[PBLF+00] J. Petersen, T. Bertram, A. Lapp, P. T. Flores, J. Schirmer. Strukturelle und verhaltensorientierte Modellierung von Regelkreisen in der Unified Modeling Language. In 8. Workshop des Arbeitskreises GROOM der GI Fachgruppe 2.1.9 Objektorientierte Software-Entwicklung, Universität Münster, 2000.

[Peuk97] R. Peukert. Einsatzmöglichkeiten von Model-Checking-Systemen beim Entwurf komplexer digitaler Steuerungen. In 42. Internationales Wissenschaftliches Kolloquium, Ilmenau, 1997.

[Piet03] S. Pietsch. Automatische Überprüfung von UML-Statecharts anhand definierbarer Design-Regeln. Diplomarbeit, TU Braunschweig, 2003.

[PMHV03] A. Pataricza, I. Majzik, G. Huszerl, G. Várnai. UML-Based Design and Formal Analysis of a Safety-Critical Railway Control Software Module. In: Symposium FORMS 2003, 2003.

[PMP01] Z. Pap, I. Majzik, A. Pataricza. Checking General Safety Criteria on UML Statecharts. In: 20th International Conference on Computer Safety, Reliabil-

ity and Security 2001, Springer Verlag, Lecture Notes in Computer Science 2187, S. 46 - 55, 2001.

[PMPS01] Z. Pap, I. Majzik, A. Pataricza, A. Szegi. Completeness and Consistency Analysis of UML Statechart Specifications. In: Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop, Győr, Ungarn, S. 83 - 90, ISBN 963-7175-16-4, 2001.

[Potr05] L. Potratz: Entwurf und Implementierung eines OCL Interpreters für den Regel Checker. Studienarbeit, unveröffentlicht, TU Braunschweig, 2005.

[Proc03] S. Prochnow. Modelltransformation von Statecharts - Formale Regeln zur Übersetzung verschiedener Semantiken. Diplomarbeit, TU Braunschweig, 2003.

[PS91] A. Pnueli, M. Shalev. What is in a Step: On the Semantics of Statecharts. In: Theoretical Aspects of Computer Science, S. 244 - 264, 1991.

[Quib99] K. Quibeldey-Cirkel. Entwurfsmuster - Design Patterns in der objectorientierten Softwaretechnik. Springer Verlag, ISBN 3-540-65825-4, 1999.

[Rapp04] M. Rappl. Entwurfsorientierte Modellierung eingebetteter Systeme. Dissertationsschrift, TU München, 2004.

[Rau01a] A. Rau. Decomposition and Interfaces Revisited. In: Gesellschaft für Informatik, FG 2.1.1: Softwaretechnik Trends, ISBN 0720-8928, 2001.

[Rau01b] A. Rau. Einsatz von Metriken bei der modellbasierten Software-Entwicklung, 15. Symposium in Paderborn, ISBN 1-56555-229-6, 2001.

[Rau02] A. Rau. Model-Based Development of Embedded Automotive Control Systems. Dissertationsschrift, Tübingen, ISBN 3-89825-599-9, 2002.

[Raus01] A. Rausch. Componentenware: Methodik des evolutionären Architekturentwurfs. Dissertationsschrift, TU München, 2001.

[RBPE+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. Object-Oriented Modeling and Design. Englewood Cliffs, Prentice Hall, 1991.

[Rein97] M. Reinhold. Die UML und das standardisierte Prozessmodell V-Model ’97. In: OBJEKTspektrum 5/97, S. 70 - 76, 1997.

[Reis86] W. Reisig. Petri-Netze. Springer Verlag, ISBN 3-54016-622-X, Berlin , 1986.

[RG99] J. Ryser, M. Glinz. A Scenario-Based Approach to Validating and Testing Software Systems Using Statecharts. In: 12th International Conference Software & Systems Engineering and Their Applications, Paris, France, 1999.

[RG03] M. Richters, M. Gogolla. Validating UML Models and OCL Constraints. In: Lecture Notes in Computer Science 1939, 2003.

[RHV04] A. Rausch, U. Hammerschall, S. Vogel. Das V-Modell 200x – ein modulares Vorgehensmodell. In: Lecture Notes in Informatics, Tutorial, Modellierung

2004, S.135-148, Marburg, 2004.

[Rich01] M. Richters. A precise approach to validating UML models and OCL constraints. Dissertationsschrift, Bremen, 2001.

[Roma85] G.-C. Roman. A Taxonomy of Current Issues in Requirements Engineering. IEEE Computer, 18 (4), S. 14 - 23, 1985.

[Romb00] J. Romberg. Objektorientierte Modellbildung von eingebetteten Echtzeit-Systemen in mechatronischen Anwendungen. Diplomarbeit, Universität Karlsruhe, 2000.

[RR99] S. Robertson, J. Robertson. Mastering the Requirements Process. Addison-Wesley (Harlow), 1999.

[Rupp00] C. Rupp. Requirements-Engineering - der Einsatz einer natürlichsprachlichen Methode bei der Ermittlung und Qualitätsprüfung von Anforderungen. In: OBJEKTspektrum 2/00, 2000.

[Rupp01] C. Rupp: Requirements-Engineering und –Management. Hanser Verlag, 2001.

[RW99] G. Reggio, R. J. Wieringa. Thirty one problems in the semantics of the UML 1.3 dynamics. In: OOPSLA'99 workshop Rigorous Modelling and Analysis of the UML, Denver, USA, 1999.

[SB05] T. Saul, G. Bikker. GraForSys - Graphische Formalisierung von Systemarchitekturen mit der UML 2.0. 4. ARTiSAN Benutzerforum, Lindau, 2005.

[Scha05] T. Scharnhorst. Management of the E/E Complexity by Introducing a Software Development Process and the Open System Architecture. In: 6. Braunschweig Conference Automation, Assistance and Embedded Real Time Platforms for Transportation, Braunschweig, 2005.

[Schl02] W. Schleuter. Herausforderungen der Automobil-Elektronik. IKB Unternehmerforum, Köln, 2002.

[Schr01] C. Schröder. Requirements Engineering for Embedded Systems – Anforderungen an die automobile Systementwicklung, 2001.

[Schu04] H.-M. Schulz. ExITE Handbuch. Version 1.4, Extessy AG, Braunschweig, 2005.

[Seem00] J. Seemann. Wiederverwendbare Komponenten für Software-Systeme im Automobil. Praxis Profiline, IN-CAR COMPUTING, 1. Auflage, Vogel Verlag, ISBN 3-8259-1909-9, 2000.

[Seke98] E. Sekerinski. Graphical Design of Reactive Systems. In: D. Bert (Ed.), B'98: Recent Advances in the Development and Use of the B Method, Second International B Conference, Montpellier, France, April 1998, S. 182 - 197, Lecture Notes in Computer Science 1393, Springer Verlag, 1998.

[SG99] A. Simons, I. Graham. 30 things that go wrong in object modelling with UML 1.3. In: H. Kilov, B. Rumpe, I. Simmonds. Precise Behavioral Specification of Businesses and Systems. Kluwer Academic Publishers, 1999.

[SGW94] B. Selic, G. Gullekson, P. T. Ward. Real-Time Object-Oriented Modeling. J. Wiley & Sons., Inc., New York, 1994.

[SHPR+03] B. Schätz, T. Hain, W. Prenninger, M. Rappl, et. al. CASE Tools for Embedded Systems. Technischer Bericht TU MI-0309, Fakultät für Informatik, TU München, 2003.

[SP00] C. Schröder, U. Pansa. UML@Automotive - Ein durchgängiges und adaptives Vorgehensmodell für den Softwareentwicklungsprozess in der Automobilindustrie, Praxis Profiline. IN-CAR COMPUTING, 1. Auflage, Vogel Verlag, ISBN 3-8259-1909-9, 2000.

[Span00] M. Spaniol. XML & UML. Seminar Internet-Informationssysteme mit XML, Rheinisch-Westfälische Technische Hochschule Aachen, 2000.

[SPHP02] B. Schätz, A. Pretschner, F. Huber, J. Philipps. In: J.-M. Bruel, Z. Bellahsene (Eds.). Model-Based Development of Embedded Systems. Advances in Object-Oriented Information Systems OOIS 2002 Workshops, Montpellier, France, Springer Verlag, 2002.

[Spre96] M. Spreng. Rapid Prototyping elektronischer Steuerungssysteme in der Automobilentwicklung. Dissertationsschrift, Universität Karlsruhe, 1996.

[SS97] I. Sommerville, P. Sawyer. Requirements Engineering: A Good Practice Guide. Wiley, New York, 1997.

[SSDC+04] H. Schlingloff, C. Stühl, H. Dörr, M. Conrad, J. Stroop, et al. IMMOS – eine integrierte Methodik zur modellbasierten Steuergeräteentwicklung. Proceedings, BMBF-Statusseminar „Software-Engineering 2006“, Berlin, 2004.

[SSR90] R. Saracco, J. Smith, R. Reed. Telecommunications Systems Engineering Using SDL. North-Holland, ISBN 0-4448-8084-4, 1990.

[SSW00] M. Sachenbacher, P. Struss, R. Weber. Advances in Design and Implementation of OBD Functions for Diesel Injection Based on a Qualitative Approach to Diagnosis. SAE 2000, Detroit, USA, 2000.

[Stan95] Standish Group. The Chaos Report, www.standishgroup.com, 1995.

[Star94] G. Starke. Sprachen zur Software-Prozessmodellierung. Dissertation, Universität Linz, Lehrstuhl für Systemwissenschaften und Automation, Shaker Verlag, Aachen, 1994.

[Stei94] W. Stein. Objektorientierte Analysemethoden - Vergleich, Bewertung, Auswahl, BI-Wissenschaftsverlag, 1994.

[Stei02] J. Steiner. Entwurf und Implementierung von UML-Strukturmodellen für

Komfortfunktionen im Kfz mit Artisan. Studienarbeit, TU Braunschweig, 2002.

[SW97] G. P. Supe, A. Weissbecker. Standardisierungsbestrebungen bei objektorientierten Modellierungsmethoden. Technischer Bericht des Frauenhofer Instituts Arbeitswirtschaft und Organisation, 1997.

[SW00] J. Seemann, J. Wolff von Gudenberg. Softwareentwurf mit UML. Springer Verlag, 2000.

[SZ03] J. Schäuffele, T. Zurawka. Automotive Software Engineering. Zeitschrift ATZ-MTZ-Fachbuch, Vieweg Verlag, ISBN 3-528-01040-1, 2003.

[Tich85] W. Tichy: RCS - A System for Version Control. In: Software - Practice & Experience 15, S. 637 - 654, 1985.

[TITUS00] ETAS, FZI, Vector. International TITUS Symposium. Copthorne Hotel, Stuttgart, 2000.

[Unbe02] H. Unbehauen. Regelungstechnik I - Klassische Verfahren zur Analyse und Synthese linearer kontinuierlicher Regelsysteme. Fuzzy-Regelsysteme, Vieweg Verlag, ISBN 3528113324, 12. Auflage, 2002.

[Vect02] VECTOR. Networking the way to new systems. International Automobile Management, 1/2002, S. 10 - 13, 2002.

[Vect04] VECTOR. Virtuelles Design von Kfz-Elektronik-Netzwerken. Automotive Electronics 1/2004, Sonderausgabe von ATZ, MTZ und Automotive Engineering Partners, Vieweg Verlag, Wiesbaden, S. 8 - 13. 2004

[Vers00a] G. Versteegen. Das V-Modell in der Praxis. dpunkt Verlag, ISBN 3-932588-39-8, 2000.

[Vers00b] C. Versteegen. Die Unified Modeling Language und das V-Modell 97. In: G. Versteegen. Das V-Modell in der Praxis. dpunkt Verlag, ISBN 3-932588-39-8, 2000.

[WBP98] M. Wolf, R. Burkhardt, I. Philippow. Software Engineering Process with the UML. In: The Unified Modeling Language, Technical Aspects and Applications, Physica-Verlag, 1998.

[Wern03] T. Werner: Evaluierung von Artisan RtS anhand der STEP-X Methodik. Studienarbeit, TU Braunschweig, 2003.

[West91] B. Westfechtel: Revisions- und Konsistenzkontrolle in einer integrierten Software-Entwicklungsumgebung. Informatik Fachberichte 280, Springer Verlag, Berlin, 1991.

[Wien97] H.-P. Wiendahl. Betriebsorganisation für Ingenieure. Hanser Verlag, 4. Auflage, ISBN 3-446-18776-6, 1997.

[WK98] J. Warmer, A. Kleppe. The Object Constraint Language: Precise Modeling

with UML. Addison-Wesley, 1998.

[WM99] A. Wohnhaas, R. Moser. Modellaustausch zwischen Steuergeräte-Entwicklungstools auf Basis einheitlicher graphischer Modellbibliotheken. 3. Stuttgarter Symposium Kraftfahrwesen und Verbrennungsmotoren, ISBN 3-8169-1751-8, Expert-Verlag, 1999.

[Wolf00] M. Wolf. Überprüfung objektorientierter Modelle, Konzepte und deren Integration in ein UML-Vorgehensmodell. Universität Ilmenau Dissertationsschrift, 2000.

[WWW90] R. Wirfs-Brock, B. Wilkerson, L. Wiener. Designing Object-Oriented Software. Prentice-Hall, Englewood Cliffs, New York, 1990.

Lebenslauf

DATEN ZUR PERSON

Name:	Martin Mutz
Adresse:	Ahmstorfer Str. 4, 38368 Rennau
Geburtsdatum:	19.06.1973
Geburtsort:	Görlitz (Polen)
Staatsangehörigkeit:	deutsch
Familienstand:	verheiratet, 2 Kinder

SCHULBILDUNG UND BERUFSERFAHRUNG

seit 01.01.2005	Carmeq GmbH, Berlin
01.05.01 – 31.12.04	Promotionsstudent am Fachbereich Mathematik und Informatik, Institut für Programmierung und Reaktive Systeme der TU Braunschweig
2000 – 2001	Fernstudium „Master of Science“ an der Universität Posen, Polen
1999 – 2001	Wissenschaftlicher Mitarbeiter an der Fachhochschule Braunschweig/Wolfenbüttel
1995 – 1999	Studium „Industrieinformatik im Praxisverbund“ an der Fachhochschule Braunschweig/Wolfenbüttel, einschließlich halbjährigem Praxissemester bei der Siemens Transportation Systems Inc., Kalifornien, USA
1994 – 1995	Wehrdienst im Rechenzentrum Flensburg
1993 – 1994	Fachoberschule Technik Hannover
1991 – 1993	Ausbildung „Technischer Assistent für Informatik“
1987 – 1991	Realschule Hannover