Verlag Dr. Hut

Dirk Ahrens

Vorgehensmodell zur Modellierung, Strukturierung und objektiven Bewertung von Software-Architekturen in der Fahrerassistenz

INGENIEURWISSENSCHAFTEN

Vorgehensmodell zur Modellierung, Strukturierung und objektiven Bewertung von Software-Architekturen in der Fahrerassistenz

DISSERTATION

zur Erlangung des Grades

Doktor-Ingenieur

der

Fakultät für Elektrotechnik und Informationstechnik
der Technischen Universität Dortmund

von

Dipl.-Ing. Dirk Steffen Ahrens
Essen, NRW, Deutschland

Tag der Einreichung: 29. Januar 2016
Tag der mündlichen Prüfung: 4. Juli 2016

Erstgutachter: Univ.-Prof. Dr.-Ing. Prof. h.c. Dr. h.c. Torsten Bertram
Zweitgutachter: Univ.-Prof. Dr. rer. nat. Bernhard Rumpe

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über
http://dnb.d-nb.de abrufbar.

ISBN 978-3-8439-2818-2

© Verlag Dr. Hut, München 2016
Sternstr. 18, 80538 München
Tel.: 089/66060798
www.dr.hut-verlag.de

Die Informationen in diesem Buch wurden mit großer Sorgfalt erarbeitet. Dennoch können Fehler nicht vollständig ausgeschlossen werden. Verlag, Autoren und ggf. Übersetzer übernehmen keine juristische Verantwortung oder irgendeine Haftung für eventuell verbliebene fehlerhafte Angaben und deren Folgen.

Alle Rechte, auch die des auszugsweisen Nachdrucks, der Vervielfältigung und Verbreitung in besonderen Verfahren wie fotomechanischer Nachdruck, Fotokopie, Mikrokopie, elektronische Datenaufzeichnung einschließlich Speicherung und Übertragung auf weitere Datenträger sowie Übersetzung in andere Sprachen, behält sich der Autor vor.

1. Auflage 2016

Vorwort und Danksagung

Die vorliegende Arbeit ist in einer Kooperation der Technischen Universität Dortmund mit der BMW Group in München entstanden. Trotz des ungeplant langen Zeitraums der Bearbeitung, den auch ich im Vorfeld nicht für möglich gehalten hätte, konnte die Arbeit am Ende noch zu einem erfolgreichen Abschluss gebracht werden. Dies hätte ich ohne die Mithilfe einer Vielzahl an Personen nicht bewerkstelligen können. Ihnen gilt mein Dank, wobei ich im Folgenden einige besonders hervorheben möchte.

Zunächst möchte ich Univ.-Prof. Dr.-Ing. Prof. h.c. Dr. h.c. Torsten Bertram für die engagierte und geduldige Betreuung als Erstgutachter danken. Die regelmäßigen Betreuungstermine, auch über die planmäßige Doktorandenzeit hinaus, haben mir enorm weiter geholfen. Des Weiteren gilt mein Dank Univ.-Prof. Dr. rer. nat. Bernhard Rumpe für die Übernahme des Zweitgutachtens.

Bei BMW möchte ich in erster Linie meinen beiden Betreuern Dr.-Ing. Andreas Pfeiffer und Prof. Dr.-Ing. Andreas Frey danken. Danke für die vielen, guten fachlichen Ratschläge, Euer Durchhaltevermögen und die mir immer wieder entgegen gebrachten Motivationsspritzen.

Trotz der anspruchsvollen Aufgaben neben der Promotion hatte ich in meinen Fachabteilungen und durch die verantwortlichen Hierarchien stets einen großen Rückhalt und entsprechenden Freiraum die Arbeit voranzutreiben. Namentlich bedanken möchte ich mich bei Reiner Friedrich, Dr. Detlef König, Dr. Renate Vachenauer, Dr. Heiko Sprenger, Jochen Otzelberger, Gerhard Fischer, Thomas Toelge und Dr. Tilo Meister.

Des Weiteren möchte ich mich bei meinen (damaligen) Studenten Sebastian Grunow, Jörg Heinichen, Erik Resvoll und Christoph Vodermair für Ihre Mitarbeit und Unterstützung im Rahmen von Praktika, Werkstudententätigkeiten und Abschlussarbeiten bedanken. Ich hoffe ihr konntet für Euren weiteren Werdegang auch etwas mitnehmen.

Während der Bearbeitungszeit hat ein reger Austausch mit vielen Doktoranden stattgefunden. Besonders wertvoll war jedoch der fachliche und nicht-fachliche Austausch sowie die Unterstützung durch intensives Korrekturlesen durch Dr.-Ing. Tobias Heier und Dr. Heiko Stullich.

Als letztes erwähnt aber ausdrücklich hervorheben und bedanken möchte ich mich bei meinen Eltern. Ohne die Unterstützung und den unerschütterlichen Glauben an den erfolgreichen Abschluss hätte ich die Arbeit vermutlich nicht beendet.

Dirk Ahrens, München im August 2016

Kurzfassung

Die Komplexität von E/E-Systemen und damit auch deren Anteil an der Wertschöpfung im Automobil nimmt beständig zu. Dies gilt in besonderem Maße für Fahrerassistenzsysteme mit ihrer hohen Vernetzung und langen, zum Teil sicherheitskritischen Wirkketten. Die vorliegende Arbeit fokussiert auf die Entwicklung der Software-Anteile von Fahrerassistenzsystemen und berücksichtigt dabei ausdrücklich auch nicht-funktionale Einflussgrößen und Anforderungen. Als wichtigster Stellhebel wird dabei die explizite Modellierung von Software-Architektur identifiziert.

Auf Basis eines vorhandenen in der Serienentwicklung etablierten Vorgehensmodells werden signifikante Änderungen und Erweiterungen vorgeschlagen, die eine formale und durchgängige Modellierung von Software-Architektur ermöglichen. Das Herzstück bildet die Abstrakte Automotive Software-Architektur (ABSOFA), welche eine eigenständige Modellierungssprache definiert, um Software-Architekturen in der Automobil-Domäne frühzeitig im Entwicklungsprozess und vollständig realisierungsunabhängig zu beschreiben. Über ein zentrales Datenmodell können automatisierte (Modell-)Transformationen in unterschiedliche Werkzeuge und zwischen unterschiedlichen Abstraktionsebenen beziehungsweise Prozessschritten durchgeführt werden. Ergänzt wird dieses zielgerichtete und effiziente Vorgehen um ein Verfahren zur objektiven Software-Architekturbewertung mit Hilfe von quantifizierten Metriken.

Darüber hinaus wird ausführlich diskutiert wie Software-Architekturen in der Fahrerassistenz konkret zu strukturieren sind, um den besonderen Anforderungen dieses Fachbereichs gerecht zu werden. Dies erfolgt am realen Beispiel von heutigen Fahrerassistenzsystemen der Längsführung. Die bestehende Software-Architektur wird analysiert und neu strukturiert. Das dient gleichzeitig als Evaluierung der zuvor genannten Ansätze unter Bedingungen der Serienentwicklung. An einem konkreten Fallbeispiel der Implementierung eines Fahrerassistenzsystems werden zum Abschluss die Vorteile der neuen Struktur eindeutig aufgezeigt.

Abstract

The complexity and therefore their stake in the overall value of E/E-systems in the automotive branch is steadily increasing. This is especially true for driver assistance systems due to their high level of interconnectedness and overall complexity including safety critical components. This thesis focuses on the software development of driver assistance systems explicitly also taking non-functional influences and requirements under consideration. Explicit modelling of software architectures is identified as most important aspect with this regard.

Basing on an existing process model from series development, several significant changes and extensions are proposed that allow a formal and consistent modelling of software architectures. The core is formed by the Abstract Automotive Software Architecture (ABSOFA) which defines a modelling language for the description of software architectures in the automotive domain early in the process and totally independent (i.e. abstracting) from realisation details. By using a central data model fully automated (model) transformations can be executed between several tools as well as several abstraction layers and process steps. This efficient and purposeful procedure is complemented with a method of an objective software architecture evaluation using quantified metrics.

Furthermore this work discusses extensively how software architectures of driver assistance systems need to be structured specifically to fulfil the particular requirements of this sector. This is accomplished by the practical example of driver assistance systems for longitudinal dynamics. The existing software architecture is analysed and restructured thus evaluating the aforesaid approaches under series development conditions. At last the advantages of the new structure are clearly proven by a case study of the implementation of a concrete driver assistance system.

Inhaltsverzeichnis

Abkürzungsverzeichnis **xiii**

1 Einführung **1**

1.1 Motivation und Problemstellung 2

1.2 Stand der Technik 4

1.3 Ziele der Arbeit 18

2 Strukturierung von Funktions- und Software-Architekturen im Automobil **23**

2.1 Begriffsdefinitionen und Ziele der Modellierung 23

2.2 Allgemeine Kriterien zur Strukturierung 26

2.2.1 Funktions-Architektur 26

2.2.2 Software-Architektur 29

3 Vorgehensmodell zur Entwicklung mechatronischer Systeme im Automobil **33**

3.1 Aktuelles Vorgehensmodell in der Serienentwicklung 33

3.1.1 Phasenmodell zur Entwicklung von eingebetteter Software 34

3.1.2 Analyse der Schwachstellen 36

3.2 Konkrete Ziele zur Anpassung des Vorgehensmodells 38

3.3 An die Zielsetzung angepasstes Vorgehensmodell 39

3.3.1 Prozessrahmen und Sichten auf das Gesamtsystem Fahrzeug . . . 40

3.3.2 Das angepasste Vorgehensmodell im Detail 43

4 Einführung der Abstrakten Automotive Software-Architektur – ABSOFA **45**

4.1 Grundlegende Ziele und Eigenschaften des Ansatzes 45

4.2 Metamodell und UML-Profil 48
4.2.1 Standpunkte, Sichten und Diagramme 50
4.2.2 Strukturelemente 51
4.2.3 Schnittstellen 53
4.3 Entwicklung eines objektiven Bewertungsverfahrens 54
4.3.1 Qualitätsmodell und Metriken 55
4.3.2 Umsetzbarkeit und Anwendbarkeit des Verfahrens 59
4.4 Variantenmodellierung und Variantenmanagement 60
4.5 Datenmodell und Austauschformat für Modelltransformationen 64
4.5.1 Ziele und Eigenschaften des Datenmodells 64
4.5.2 Modellierung und Umsetzung des Datenmodells 67
4.5.3 Anwendung innerhalb der Prozessschritte 68

5 Strukturierung von Software-Architekturen für Fahrerassistenzsysteme 73
5.1 Spezifische Zielsetzung und Randbedingungen 73
5.2 Vorbereitung der Neustrukturierung der FAS-Längsführungs-Architektur 76
5.2.1 Analyse der aktuellen Architektur 77
5.2.2 Festlegung des methodischen Vorgehens 79
5.3 Durchführung der Neustrukturierung der FAS-Längsführungs-Architektur 85
5.3.1 Grobkonzept der Software-Architektur 86
5.3.2 Iterative Entwurfsarbeit 89
5.3.3 Feinkonzept der Software-Architektur und Fazit 98
5.4 Diskussion der universellen Eignung der Struktur für Fahrerassistenzsysteme 101

6 Fallbeispiel zum Nachweis des praktischen Nutzens der Struktur 105
6.1 Ziele des Fallbeispiels 105
6.2 Durchführung und Ergebnisse 106

7 Zusammenfassung und Ausblick 111
7.1 Zusammenfassung der Ergebnisse 111
7.2 Ausblick 114

Glossar **117**

A Wesentliche Anforderungen an die Ansätze **131**

B Das UML-Profil der ABSOFA **135**

C Qualitätskriterien und Qualitätsattribute **145**

D Software-Architekturmetriken **149**

E Neustrukturierung einer Software-Architektur **159**

Abbildungsverzeichnis **171**

Tabellenverzeichnis **173**

Literaturverzeichnis **175**

Abkürzungsverzeichnis

AADL	Architecture Analysis & Design Language
ABS	Antiblockiersystem
ABSOFA	Abstrakte Automotive Software-Architektur
ACC	Adaptive Cruise Control
ACC S&G	Adaptive Cruise Control mit Stop&Go-Funktion
ADAS	Advanced Driver Assistance System
ADL	Architecture Description Language
AML	Automotive Modeling Language
ASIL	Automotive Safety Integrity Level
AutoMoDe	Automotive Model Based Development
AUTOSAR	Automotive Open System Architecture
BASt	Bundesanstalt für Straßenwesen
BMW	Bayerische Motoren Werke
CASE	Computer-Aided Software Engineering
CDS	Comfort Dynamic System
COLA	Component Language
DCC	Dynamic Cruise Control
E/E	Elektrik/Elektronik
EAST-ADL	Electronics Architecture and Software Technology – Architecture Description Language
EAST-EEA	Electronics Architecture and Software Technology – Embedded Electronic Architecture
ECBA	Electronically Controlled Brake Actuator
ESP	Elektronisches Stabilitätsprogramm
FAS	Fahrerassistenzsystem
FGR	Fahrgeschwindigkeits-Regelung
GUI	Graphical User Interface
HDC	Hill Descent Control

iBrake	Intelligent Brake (auch Aktive Gefahrenbremsung)
IEEE	Institute of Electrical and Electronics Engineers
JAXB	Java Architecture for XML Binding
KIFA	Karosserie, Innenraum, Fahrdynamik, Antrieb
LDM	Längsdynamikmanagement
MDA	Model Driven Architecture
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
PMA	Parkmanöver-Assistent
PMA-LQ	Parkmanöver-Assistent mit Längs- und Querführung
RAM	Random Access Memory
ROM	Read Only Memory
RTE	Run-Time Environment
RUP	Rational Unified Process
SLD	Speed Limitation Device
STA	Stau-Assistent
SVW	Spurverlassens-Warnung
SWC	(AUTOSAR) Software Component
SWW	Spurwechsel-Warnung
SysML	Systems Modeling Language
TLC	Time to Lane Crossing
UML	Unified Modeling Language
VFB	Virtual Functional Bus
VZA	Verzögerungs-Assistent
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations

1 Einführung

„Nichts steigt so schnell wie Ansprüche". Diese Aussage von Braess, 1993 [24] trifft seit Anbeginn für die Entwicklung des Automobils und dessen Kunden zu. Aufgrund der rasanten Durchdringung von eingebetteten, das heißt mechatronischen und softwaredominanten, Systemen und der mit diesen Systemen einhergehenden Möglichkeiten, gilt dies aktuell und für die Zukunft in besonderem Maße. Für das Ende des vergangenen Jahrzehnts wurde bereits 2005 von Seiffert et al. [106] vorhergesagt, dass der anteilige Wert von Software und Elektrik/Elektronik (E/E) im Automobil auf ca. 35% ansteigen wird. Dieser Trend setzt sich weiter fort. Dabei ist dieses Umfeld nach Schnelle, 2005 [104] einerseits durch einen steigenden Funktionsumfang der Einzelfunktionen, andererseits durch eine stetig steigende Vernetzung der Funktionen untereinander beziehungsweise nach Draeger, 2011 [35] mit Außenwelt und Fahrer charakterisiert. Zur Realisierung von Kundenfunktionen sind in einem Fahrzeug eine Vielzahl an Sensoren, Aktoren, Bedien- und Anzeigeelementen sowie Plattformen zur Ausführung von Software vonnöten. Die Beherrschung der Komplexität wird zur Herausforderung. Für Fahrerassistenzsysteme (FAS) und sicherheitskritische Systeme gelten diese Aussagen in besonderem Maße.

Aus dieser Komplexität resultieren hohe Entwicklungsaufwände, welche unter den bekannten Randbedingungen und Herausforderungen der Automobil-Domäne aufzubringen sind. Hierbei sind insbesondere zu nennen:

- Zunahme der Anzahl an Baureihen, Modellen und Varianten,
- Verkürzte Entwicklungszeiten bis zur Serienreife und
- Angespannte Kostenziele.

Entwicklungsmethoden und -prozesse konnten dabei in der Vergangenheit mit dem Tempo der Wandlung der oben beschriebenen Herausforderungen nicht immer Schritt halten. Es existiert somit ein Bedarf nach Methoden, die die Entwicklung mechatronischer Systeme im Automobil ganzheitlich als vollständiges, durchgängiges Vorgehensmodell – oder bezüglich bestimmter Aspekte innerhalb eines solchen – ermöglichen oder verbessern.

Die vorliegende Dissertation beschäftigt sich im Schwerpunkt mit der Entwicklung der Software-Anteile der oben genannten Systeme am Beispiel von Fahrerassistenzsystemen. Die folgenden Abschnitte dieses ersten Kapitels sollen eine Einführung in das Thema geben. Dabei wird zunächst auf die Motivation und Problemstellung eingegangen. Im Anschluss erfolgt ein Überblick über den aktuellen Stand der Technik. Aufgrund der darin vorliegenden Mängel werden konkrete Ziele der Arbeit abgeleitet und vorgestellt.

Wichtige Begriffe, die in diesem und den weiteren Kapiteln dieser Arbeit verwendet werden, sind zum eindeutigen Verständnis der Arbeit im Kapitel „Glossar" erläutert.

1.1 Motivation und Problemstellung

Diese Arbeit ist in Kooperation mit der BMW Group München, Bereich Fahrerassistenzsysteme, entstanden. Somit entstammen die im Folgenden geschilderten Aspekte dem realen Entwicklungsalltag eines konkreten Automobilherstellers, sie werden jedoch im Rahmen der Arbeit als stellvertretend für die Automobil-Branche in Gänze betrachtet.

Aktive Fahrerassistenzsysteme sind gemäß Winner et al., 2009 [128] (5.1 Begriffsklärung „Fahrerassistenzsysteme“) Systeme, die aktiv in die Fahrzeugführung eingreifen und weisen im Bereich der eingebetteten Systeme im Automobilbereich mit die höchste Komplexität auf, da für ihre Realisierung immer ein stark verteiltes Netzwerk aus einer Vielzahl von elektrischen und elektronischen Komponenten notwendig ist. Dazu gehören typischerweise Umfeldsensorik (z.B. Radar, Ultraschall, Kamera), Bedienelemente (z.B. Taster, Hebel), Anzeigeelemente (z.B. Kombi-Instrument), Ablaufumgebungen für Software (z.B. elektronische Steuergeräte) sowie Aktorik (z.B. Antrieb, Bremse). Bild 1.1 zeigt Bestandteile einer typischen Wirkkette eines solchen Systems.

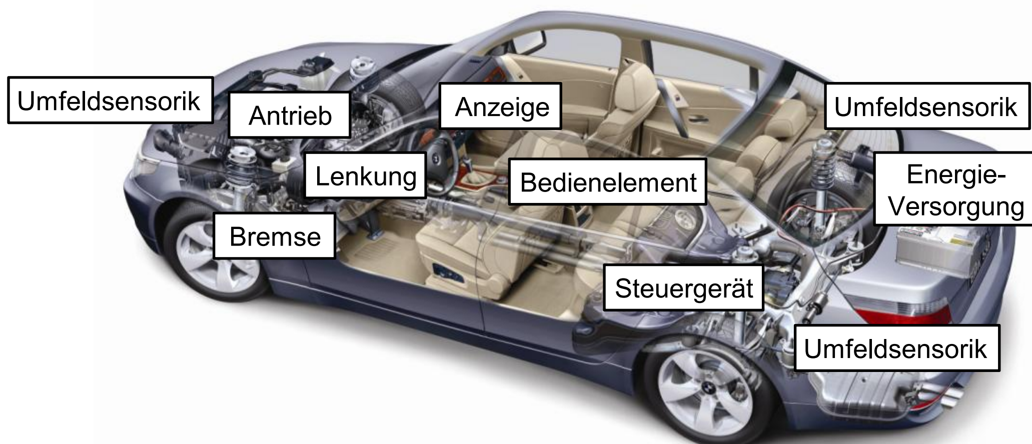

Bild 1.1: Typische Bestandteile der Wirkkette eines Fahrerassistenzsystems

Neben der sich für jedes Fahrerassistenzsystem explizit ergebenden Komplexität aufgrund des zuvor beschriebenen Aufbaus und des Umfangs der zugehörigen Kundenfunktion an sich, ergibt sich eine zusätzliche, implizite Komplexität aufgrund der Vernetzung mehrerer, unterschiedlicher Systeme. Verschiedene Fahrerassistenzsysteme müssen beispielsweise bezüglich Aktivierung und Deaktivierung – inklusive eines für den Fahrer angenehmen Übergangsverhaltens – aufeinander abgestimmt werden. Außerdem existiert in der Regel nur ein Aktor für eine bestimmte Art von Stelleingriff im Fahrzeug. Potenziell widersprüchliche Anforderungen an einen Aktor sind zu priorisieren und zu koordinieren. Fahrerassistenzsysteme sind darüber hinaus häufig den sicherheitsrelevanten Systemen zuzuordnen. Dies bedeutet, dass zumindest auf Teile der Wirkkette Sicherheitsziele alloziert werden. So erhöhen sich die Komplexität und die Anforderungen an

die Entwicklung der Systeme zusätzlich.

Viele Eigenschaften von Fahrerassistenzsystemen ergeben sich aus den sogenannten funktionalen Anforderungen. Zu diesen zählen, vereinfacht ausgedrückt, alle Anforderungen, die in einem für den Kunden unmittelbar erlebbaren Verhalten des Fahrerassistenzsystems – in diesem Kontext auch als Kundenfunktion bezeichnet – resultieren. Die funktionalen Anforderungen werden im Anforderungsprozess mit verschiedenen Stakeholdern intensiv abgestimmt, bis das für den Kunden erwünschte Verhalten ausreichend detailliert und unter Zustimmung aller Stakeholder spezifiziert ist. Da die Umsetzung der funktionalen Anforderungen in einem direkt erlebbaren Verhalten resultiert, sind diese im Rahmen der Verifikation und Validation sehr gut testbar.

Darüber hinaus existieren jedoch auch eine Vielzahl nicht-funktionaler Anforderungen an ein Fahrerassistenzsystem. Diese spiegeln sich im Umkehrschluss nicht in einem für den Kunden unmittelbar erlebbaren Verhalten wider. Sie beschreiben viel mehr Eigenschaften des Systems, die sich für an der Entwicklung beteiligte Personen über den gesamten Produktlebenszyklus auswirken. Beispielhaft seien hier genannt: Funktionsentwickler, Modultester, Mitarbeiter in der Handelsorganisation. Für den Kunden ist die Erfüllung der nicht-funktionalen Anforderungen bestenfalls mittelbar erlebbar, da diese unter anderem den Verkaufspreis, Instandsetzungskosten und Zeitpunkt der Verfügbarkeit am Markt beeinflussen. Da im weiteren Verlauf der Arbeit die nicht-funktionalen Anforderungen noch ausführlich behandelt werden, seien an diese Stelle nur einige wichtige Aspekte ohne Anspruch auf Vollständigkeit genannt, um einen Überblick über Art und Umfang nicht-funktionaler Anforderungen im Bereich von eingebetteten Systemen in der Automobil-Domäne zu erhalten.

- Hohe Wiederverwendbarkeit und Portierbarkeit auf unterschiedliche Ablaufumgebungen,
- Hohe Skalierbarkeit und Erweiterbarkeit,
- Hohe Ressourceneffizienz bezüglich Speicher- und Laufzeitbedarf,
- Hohe Verfügbarkeit, Robustheit und schnelle Instandsetzung im Fehlerfall und
- Konfigurierbarkeit von Länder- und Marktspezifika bei Bandende und Händler.

Nicht-funktionale Anforderungen sind oft nicht oder nur unzureichend objektivier- und quantifizierbar und daher entsprechend schwierig zu überprüfen. Gleichzeitig hat die Erfahrung der letzten Jahre gezeigt, dass nicht-funktionale Anforderungen oft überhaupt nicht explizit betrachtet wurden. Die entsprechenden Eigenschaften stellten sich somit automatisch nach Umsetzung der funktionalen Anforderungen in einer nicht weiter vorgegebenen Art und Weise von selbst ein. Dies liegt einerseits daran, dass der Fokus erst in der näheren Vergangenheit auf diese Thematik gerichtet wurde. Andererseits sind diese Aspekte bei starker zeitlicher oder finanzieller Anspannung eines Entwicklungsprojektes häufig zuerst betroffen von Entfeinerung – dies bedeutet Vereinfachung oder Entfall von Anforderungen. Die Priorität liegt zunächst auf den kundenerlebbaren Aspekten.

Genau diese Herangehensweise rächt sich jedoch zu einem späteren Zeitpunkt. Wenn sich beispielsweise erst nach Abschluss eines Projektes in einem Folgeprojekt offenbart,

dass große Teile der entwickelten Software nicht wiederverwendbar oder portierbar sind, muss ein unnötiger Personal- und Zeitaufwand betrieben werden. Es setzt sich mehr und mehr die Erkenntnis durch, dass zur Erreichung aller Ziele unter den zu Beginn der Einführung genannten äußeren Randbedingungen, die explizite Betrachtung von nicht-funktionalen Anforderungen unerlässlich und methodisch verbindlich im Entwicklungsprozess zu verankern ist.

Einen sehr großen Stellhebel nimmt dabei die Funktions- und Software-Architektur ein. Die Entscheidungen, die bei diesen Arbeitsergebnissen zu frühen Zeitpunkten in der Entwicklung getroffen werden, bestimmen am Ende maßgeblich die Eigenschaften des Produktes bezogen auf die nicht-funktionalen Anforderungen. Die Funktions- und Software-Architektur der existierenden Fahrerassistenzsysteme hat nach mehrjähriger evolutionärer Weiterentwicklung um immer neue Teilfunktionalitäten bestehender Kundenfunktionen einerseits, sowie Hinzufügungen von neuen Kundenfunktionen andererseits, eine Struktur erreicht, die eine weitere Bearbeitung unter Anwendung des bisherigen Vorgehens unmöglich macht. Ein Hinzufügen neuer Aspekte sowie ein effizientes Herauslösen von Teilfunktionalität entsprechend der durch den Kunden bestellten Sonderausstattungen in einem Fahrzeug ist nicht mehr unter vertretbarem Aufwand möglich. Anders formuliert: Skalierbarkeit ist nicht gegeben. Dies liegt daran, dass die Architekturen im Sinne von Berücksichtigung nicht-funktionaler Anforderungen verglichen mit der rasanten Weiterentwicklung der funktionalen Aspekte vernachlässigt wurden.

Die vorliegende Arbeit leistet einen Beitrag zur Entwicklung von aktuellen und zukünftigen Fahrerassistenzsystemen, fokussiert wird dabei auf die Software-Anteile. Nicht-funktionale Anforderungen an die Software werden erarbeitet und explizit berücksichtigt. Entscheidenden Einfluss auf die Erreichung haben die Arbeitsergebnisse der Funktions- und Software-Architektur. Es werden konkrete Vorgaben zur Strukturierung geliefert sowie Strukturen objektiv messbar gemacht. Die erarbeiteten Methoden werden in ein formales und durchgängiges Vorgehensmodell integriert. Die detaillierten Ziele der Arbeit finden sich in Abschnitt 1.3. Da Fahrerassistenzsysteme verglichen mit anderen Fachbereichen in der Automobil-Entwicklung eine sehr hohe Komplexität aufweisen, können Methoden, die alle oben aufgeführten Anforderungen und Randbedingungen der Entwicklung von komplexen Fahrerassistenzsystemen erfüllen, auch für andere Bereiche eingesetzt werden. Dazu sind unter Umständen spezifische Anpassungen und Evaluierungen der Anwendbarkeit notwendig, die nicht mehr Gegenstand dieser Arbeit sind.

1.2 Stand der Technik

Im vorliegenden Abschnitt wird der für die Bearbeitung der zuvor beschriebenen Problemstellung relevante Stand der Technik dargelegt. Vorhandene Ansätze und Methoden werden verglichen und bezüglich ihrer Eignung untersucht. Mängel im Stand der Technik sollen identifiziert werden, um daraus im darauffolgenden Kapitel 1.3 den Handlungsbedarf in Form der konkreten Zielsetzung der Arbeit ableiten zu können.

Fahrerassistenzsysteme

Fahrerassistenzsysteme können generell sehr weit gefasst werden und es existiert eine Vielzahl an Definitionen. Die Schnittmenge dieser beschreibt Fahrerassistenzsysteme als Systeme, die den Fahrer während einer Fahrt entlasten, unterstützen oder beliebige Aufgaben vor, während und nach der Fahraufgabe übernehmen. Dies umspannt somit einen Bereich an Assistenz, der von heute als selbstverständlich vorausgesetzten Funktionalitäten wie einer automatischen Blinkerrückstellung bis hin zu Funktionen reicht, welche ohne Eingreifen des Fahrers die Fahrzeugführung vollständig in bestimmten Anwendungsbereichen übernehmen (Spannheimer et al., 2012 [107]).

Unter Fahrerassistenzsystemen werden im Rahmen dieser Arbeit die nach Schwarz, 2006 [105] so genannten weiterführenden Fahrerassistenzsysteme (Englisch: Advanced Driver Assistance Systems (ADAS)) verstanden. Diese zeichnen sich durch die Verwendung von Umfeldsensoren zur Erfassung der Fahrsituation und das Durchführen von aktiven Reaktionen auf diese aus. Aus diesem Grund ist auch der Begriff Aktive Fahrerassistenzsysteme üblich. Nach Donges, 1982 [34] können Fahrerassistenzsysteme in einem so genannten 3-Ebenen-Modell kategorisiert werden. Maßgeblich zur Kategorisierung ist die Ebene der Fahraufgabe – Stabilisierung, Bahnführung, Navigation – in welcher das Fahrerassistenzsystem einen Beitrag leistet. Mit Stabilisierung ist die Fahrzeugführung „im Kleinen“ gemeint, das bedeutet das Fahrzeug unter Kontrolle und damit in einem stabilen Fahrzustand zu halten. Hierbei unterstützen bekannte Systeme wie Elektronisches Stabilitätsprogramm (ESP) und Antiblockiersystem (ABS). Unter Bahnführung wird der Teil der Fahraufgabe verstanden, der das Fahrzeug auf einer konkreten Trajektorie in einer bestimmten Geschwindigkeit bewegt, beispielsweise das Folgen einer Spur in einer Kurve. Bei der Bahnführung unterstützen Systeme wie Adaptive Cruise Control (ACC) und Spurverlassens-Warnung (SVW). Navigation ist die Fahrzeugführung „im Großen“. Hierzu gehört die Routenplanung, welche durch ein Navigationssystem unterstützt werden kann.

Darüber hinaus existiert eine von der Bundesanstalt für Straßenwesen (BASt) (Gasser, 2012 [41]) vorgestellte und inzwischen weit verbreitete Klassifikation, die Fahrerassistenzsysteme bezüglich ihres Grades der Unterstützung unterteilt. Dabei werden die folgenden Kategorien vorgeschlagen:

- Driver Only,
- Assistiert,
- Teilautomatisiert,
- Hochautomatisiert und
- Vollautomatisiert.

Tabelle 1.1 führt diese Kategorien aus und gibt jeweils ein Anwendungsbeispiel. Von besonderer Bedeutung sind im weiteren Verlauf der Arbeit die Fahrerassistenzsysteme der Längsführung. Diese zeichnen sich dadurch aus, dass sie die Fahrzeugbewegung durch Eingriffe in Antrieb und/oder Bremse beeinflussen. Die typischen Fahraufgaben sind

Tabelle 1.1: Benennung und Klassifizierung automatisierter Fahrfunktionen [41]

Bezeichnung	Beschreibung und Beispiel
Driver Only	Der Fahrer führt dauerhaft Längs- und Querführung aus. Es ist kein in die Längs- oder Querführung eingreifendes (Fahrerassistenz-)System aktiv.
Assistiert	Der Fahrer führt dauerhaft entweder die Längs- oder die Querführung aus. Die jeweils andere Fahraufgabe wird in gewissen Grenzen vom System ausgeführt. Der Fahrer muss das System dauerhaft überwachen und jederzeit zur Übernahme bereit sein. Beispiel: Adaptive Cruise Control.
Teilautomatisiert	Das System übernimmt Längs- und Querführung (für einen gewissen Zeitraum und/oder in spezifischen Situationen). Der Fahrer muss das System dauerhaft überwachen und jederzeit zur Übernahme bereit sein. Beispiel: Stau-Assistent.
Hochautomatisiert	Das System übernimmt Längs- und Querführung für einen gewissen Zeitraum und/oder in spezifischen Situationen. Der Fahrer muss das System nicht dauerhaft überwachen. Alle Systemgrenzen werden vom System erkannt und der Fahrer rechtzeitig zur Übernahme der Fahraufgabe aufgefordert. Das System ist nicht in allen Situationen in der Lage, in den risikominimalen Systemzustand zurückzuführen. Beispiel: Autobahn-Chauffeur.
Vollautomatisiert	Das System übernimmt Längs- und Querführung in einem definierten Anwendungsfall. Der Fahrer muss das System nicht dauerhaft überwachen. Vor dem Verlassen des Anwendungsfalls fordert das System den Fahrer mit ausreichender Zeitreserve zur Übernahme der Fahraufgabe auf. Alle Systemgrenzen werden vom System erkannt. Das System ist in allen Situationen in der Lage, in den risikominimalen Systemzustand zurückzuführen. Beispiel: Autobahn-Pilot.

dabei das Regeln (oder Limitieren) von Geschwindigkeit und Abstand im Bereich der Komfortsysteme beziehungsweise das Reduzieren der Geschwindigkeit zur Vermeidung von Kollisionen im Bereich der Sicherheitssysteme. Fahrerassistenzsysteme der Längsführung können nach den soeben vorgestellten Definitionen und Klassifikationen wie folgt zusammengefasst werden:

- Sie sind Teil der weiterführenden beziehungsweise aktiven Fahrerassistenzsysteme (ADAS) nach Schwarz.
- Sie unterstützen im Bereich der (Ebene der) Bahnführung nach Donges.
- Ihr Automatisierungsgrad entspricht der Definition „assistiert“ nach der BASt.

Einen Überblick über Fahrerassistenzsysteme am Markt geben Winner et al., 2009 [128]. Assistierende Systeme sind inzwischen flächendeckend in allen Fahrzeugklassen in Serienreife verfügbar. Teilautomatisierte Systeme befinden sich nach Ahrens, 2012 [3] und Schaller, 2009 [99] aktuell kurz vor der Markteinführung oder sind bereits verfügbar. Hoch- und vollautomatisierte Systeme sind aktuell Gegenstand von Forschung und Vorentwicklung. Mit der Markteinführung kann erst in einigen Jahren gerechnet werden.

Entwicklungsprozesse und -methoden

In der Literatur existieren eine Reihe von Prozessen und Methoden, die bei der Entwicklung von Systemen eingesetzt werden können. Sie sind zum einen Teil von recht allgemeinem Charakter, zum anderen Teil bereits auf spezifische Domänen, Arten von Systemen oder nur auf bestimmte Teilumfänge eines Entwicklungsprozesses fokussiert.

Ein Beispiel für die Erstgenannten ist der Bereich des Systems Engineering, zu finden unter anderem in Züst, 2004 [131]. Hierbei wird eine allgemeine Arbeitsweise für die Entwicklung von beliebigen Systemen vorgeschlagen, welche jedoch für das spezifische Problem bewusst noch viel Spielraum für Entwickler lässt. Die VDI-Richtlinie 2206, 2004 [120] beschreibt eine auf die Entwicklung von mechatronischen Systemen spezialisierte Methodik. Somit werden die spezifischen Eigenschaften und Randbedingungen dieser Systeme grundsätzlich adressiert, der Ansatz ist jedoch trotz allem allgemeingültig gehalten, um für jede Art mechatronischer Systeme anwendbar zu sein. Basis der Richtlinie ist das weit verbreitete V-Modell, 1997 [27], welches notwendige Aktivitäten und Ergebnisse während der gesamten Phase einer Entwicklung vorschlägt. Dieses wurde ursprünglich für die Entwicklung von informationstechnischen Systemen entwickelt, hat aber inzwischen in nahezu allen Bereichen der Systementwicklung eine hohe Durchdringung erreicht. Die aktuelle Version ist das V-Modell-XT, 2012 [14]. Neben diesen umfassenden Ansätzen existieren auch Varianten, die sich nur auf Teilumfänge einer Gesamtentwicklung beschränken, diese dafür sehr detailliert betrachten. Beispielhaft sei hier der Rational Unified Process (RUP) nach Kruchten, 2003 [67] genannt, welcher ein Vorgehensmodell zur Software-Entwicklung darstellt.

In der Automobil-Domäne verhält es sich grundsätzlich ähnlich. Es existieren sowohl Ansätze, die auf den gesamten Entwicklungsprozess abzielen wie bei Kleinod, 2006 [64], als auch Ansätze, die bestimmte Teilbereiche, beispielsweise die Software-Entwicklung, betrachten. Insbesondere Letztgenannte ist derzeit Gegenstand von umfangreichen Aktivitäten in Forschung und Industrie. Ansätze zur Software-Entwicklung im Automobil werden gesondert im Abschnitt „Software-Entwicklung in der Automobil-Domäne" behandelt.

Neben den zuvor vorgestellten Ansätzen, welche frei zugänglich sind, existieren in Industrieunternehmen häufig interne Dokumente, die in der Produktentwicklung des jeweiligen Unternehmens verbindlich anzuwenden sind. Ein Beispiel ist das in Kapitel 3 dargestellte Vorgehensmodell, welches derzeit in der Praxis für die Entwicklung von Fahrerassistenzsystemen eingesetzt wird.

Ein weiterer wichtiger Aspekt in der Entwicklung von Systemen ist das Anforderungsmanagement. Anforderungen müssen zu verschiedenen Zeitpunkten während der Entwicklung in unterschiedlicher Art (Zweck, Granularität, Empfänger der Anforderungen) erstellt werden. Anforderungsmanagement ist zwar nicht direkter Fokus dieser Arbeit, die grundlegenden Konzepte nach Rupp, 2009 [97] und Schienmann, 2002 [102] werden jedoch angewendet.

Darüber hinaus sind die in dieser Arbeit vorausgesetzten Grundlagen zu eingebetteten

Systemen unter anderem in Marwedel, 2007 [76] und Wietzke et al., 2005 [126] sowie zu Automobil-Elekronik und -Bordnetzen (insbesondere elektronische Steuergeräte und Bussysteme) in Braess et al., 2013 [25], Reif, 2006 [95] und Zimmermann et al., 2006 [130] verfügbar.

Funktions-Entwicklung und Funktions-Architektur in der Automobil-Domäne

Unter Funktions-Entwicklung versteht man in der Automobil-Domäne die Entwicklung der Logik von E/E-Funktionen. Dazu gehört das Verständnis, die Modellierung und die Manipulation von Wirkketten im Fahrzeug. Es wird typischerweise grafisch modelliert unter Einsatz von Werkzeugen wie Simulink [77]. Die dort entwickelten Modelle werden häufig über automatische Code-Generierung in Software umgesetzt und anschließend entweder in Rapid-Prototyping-Umgebungen oder in Serien-Steuergeräten ausgeführt. Eine Einführung in die Thematik geben Schäuffele et al., 2006 [101]. Zu diesem Zweck weisen sie eine zeitliche Diskretisierung auf und haben Aspekte von Blockschaltbildern (sequentieller Signalfluss und Abarbeitung von Bausteinen) und Zustandsautomaten nach Hopcroft et al., 1990 [49].

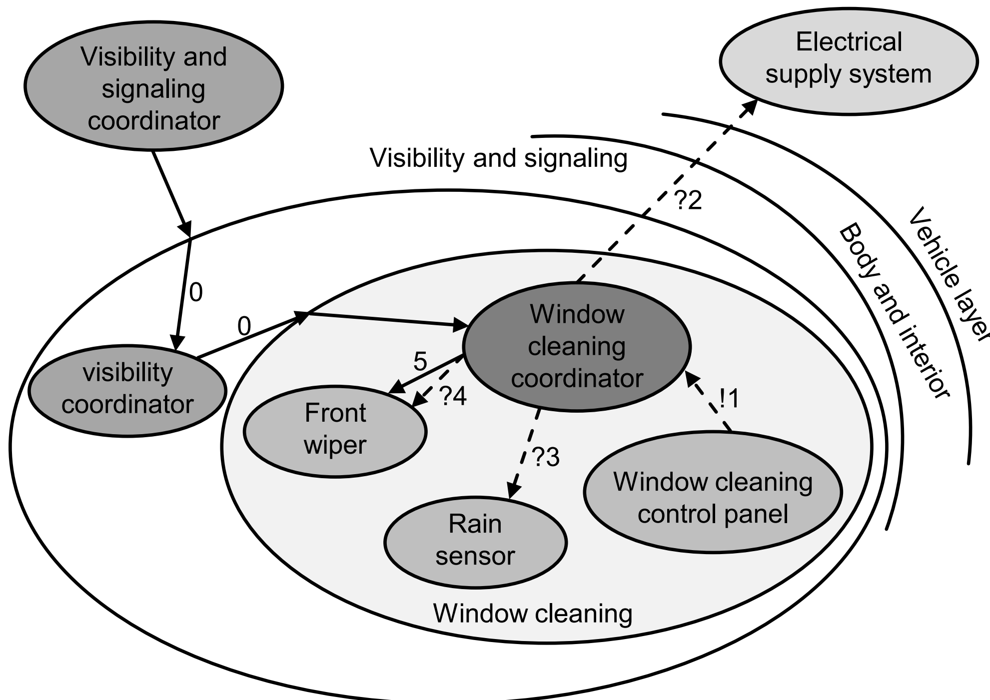

Bild 1.2: Beispielhafte Modellierung mit der CARTRONIC-Notation [73]

Ein wichtiger Aspekt ist hierbei die Funktions-Architektur. Dieses Thema wird in Kapitel 2 noch ausführlich behandelt inklusive einer genauen Definition. Zur Modellierung

von Funktions-Architekturen gibt es in der Literatur verschiedene Ansätze. Das Ordnungskonzept CARTRONIC nach Bertram et al., 1997 [19], Lapp et al., 2001 [73] und Walther et al., 2002 [124] ermöglicht mit einer vorgegebenen Notation die Modellierung von logischen Funktionen auf Gesamtfahrzeug-Ebene mit Hilfe von logischen Systemkomponenten und Kommunikationsbeziehungen. Der Begriff „logisch“ kann hier mit „abstrahierend von Realisierungsdetails“ gleichgesetzt werden. Bild 1.2 zeigt eine beispielhafte Modellierung einer Komponente „Visibility and signaling“.

CARTRONIC bietet zur Anwendung der Notation ein Profil für die Unified Modeling Language (UML) [89] an, Grundlagen dazu finden sich in Kecher, 2005 [62] sowie Rupp et al., 2007 [98]. Dieser Trend auf einen mächtigen und offenen Standard wie die UML zurückzugreifen, lässt sich bei etlichen Ansätzen wie zum Beispiel nach von der Beeck et al., 2003 [17] beobachten. Im Gegensatz dazu gibt es jedoch auch proprietäre Werkzeuge wie PREEvision [121] sowie frei verfügbare Ansätze, die nicht auf die UML sondern auf eine komplett eigenständige Modellierung und Metamodell aufbauen. Beispiele hierfür sind Kugele et al., 2007 [70] und Wild et al., 2006 [127].

Nicht alle vergleichbaren Ansätze verwenden den Begriff „Funktions-Architektur“. In der Literatur ebenfalls weit verbreitet sind die Begriffe „Logische Architektur“ und „Funktionsnetz“, geprägt unter anderem von Grönniger et al., 2008 [42]. Es zeigt sich jedoch, dass der Grundgedanke einer abstrahierten, realisierungsunabhängigen Sicht auf das technische System sich in nahezu allen Ansätzen, welche sich mit einer durchgängigen Modellierung im Rahmen der Software-Entwicklung beschäftigen, etabliert hat. Eine ausführliche tabellarische Gegenüberstellung dieser Ansätze findet sich im folgenden Abschnitt.

Software-Entwicklung in der Automobil-Domäne

Unter „Software-Entwicklung“ (auch: „Software-Engineering“, „Software-Technik“) werden Aspekte eines methodischen, systematischen Vorgehens verstanden, um Softwareprodukte zu erhalten. Es handelt sich hierbei um eine vergleichsweise junge Disziplin, welche erst in den vergangenen Jahrzehnten aufgrund der Zunahme von Software-Anteilen und deren steigender Komplexität in den unterschiedlichsten Domänen in den Fokus der Wissenschaft gerückt ist. Grundlagen können unter anderem Balzert, 1998 [11] und 2001 [12] entnommen werden. Objektorientierung (beschrieben in Balzert, 1999 [10], Oestereich, 2006 [83] und Oestereich et al., 2001 [84]) wird seit einiger Zeit eingesetzt, um einerseits komplexe (Software- und andere) Systeme zu modellieren und damit ihrer Herr zu werden. Andererseits beschreibt diese auch ein Programmierparadigma zur Erzeugung von Software. Bekannte objektorientierte Programmiersprachen sind C++ von Stroustrup, 2013 [111] und Java von SUN Microsystems [112]. Grundlagen zur Anwendung von Java finden sich in Krüger et al., 2012 [68].

In der Literatur werden einige Vorgehensmodelle definiert, um der stetig steigenden Komplexität von Software zu begegnen. Exemplarisch zu nennen seien agile Methoden nach Beck, 2003 [15], Cockburn, 2002 [31] und Hruschka et al., 2002 [50] sowie der

oben zitierte Rational Unified Process (RUP). Man kann grundsätzlich feststellen, dass sich modellbasierte und modellgetriebene Ansätze domänenübergreifend immer mehr durchsetzen. Zur Unterscheidung dieser wird auf Stahl et al., 2007 [108] sowie das Glossar verwiesen. Diese Entwicklung wurde unterstützt durch frei verfügbare und ausgereifte Ansätze wie die UML und die Systems Modeling Language (SysML) von der Object Management Group (OMG) [88], beschrieben unter anderem bei Weilkiens, 2006 [125].

Aufgrund der Vielzahl an existierenden Ansätzen und Methoden wird im weiteren Verlauf dieser Arbeit auf modellgetriebene Vorgehensmodelle fokussiert. Die Kombination aus Veranschaulichung von Systemen durch Einsatz grafischer Modelle und Nutzung von Formalisierung zur Automatisierung von Übergängen zwischen Modellierungsebenen und von Code-Generierung wird für die Lösung der gegebenen Problemstellung als zielführend erachtet.

Es soll zunächst ein Überblick über die Besonderheiten von Software – und damit der Software-Entwicklung – in der Automobil-Domäne gegeben werden. Detaillierte Informationen können Broy, 2006 [26], Kugele, 2012 [69] und Schäuffele et al., 2006 [101] entnommen werden. Aus diesen geht hervor, dass im Wesentlichen die folgenden Aspekte anzuführen sind:

- Hohe Anforderungen an die funktionale Sicherheit,
- Hohe Anforderungen an Zuverlässigkeit und Verfügbarkeit,
- Einhaltung allgemeiner und spezifischer Normen, zum Beispiel ISO 9001, 2009 [55], ISO 15622, 2010 [56], ISO 26262, 2011 [57], MISRA C, 2012 [82],
- Große Stückzahlen,
- Vergleichsweise lange Produktlebenszyklen inklusive Wartung nach Vertrieb,
- Hohe Innovativität in Verbindung mit kurzen Entwicklungszeiten,
- Strenge Kostenziele,
- Bedarf nach effizienten Umsetzungen aufgrund knapper Hardware-Ressourcen und
- Unterschiedliche Geschäftsmodelle mit Anteilen in Eigen- und Fremdentwicklung.

Software-Entwicklung in der Automobil-Domäne beschreibt folgerichtig alle Aktivitäten im Rahmen der Fahrzeug-Entwicklung, die auf Erstellung von Software abzielen. Das Endergebnis stellt Software dar, welche auf Hardware-Plattformen im Fahrzeug lauffähig ist. In der Automobil-Domäne wird derzeit objektorientierter Quellcode für eingebettete Systemen nur selten eingesetzt. Sehr häufig anzutreffen ist weiterhin Code der Programmiersprache C, beschrieben in Kernighan, 1988 [63]. Dieser kann entweder händisch oder mit Hilfe von automatischer Code-Generierung aus so genannten CASE-Werkzeugen (Computer-Aided Software Engineering) erstellt werden. Verbreitet sind das bereits erwähnte Simulink sowie ASCET [36]. Diese Werkzeuge stellen eine Vorstufe zur modellgetriebenen Software-Entwicklung dar.

Software-Architektur und Architektur-Beschreibung

Die steigende Komplexität von Software-Systemen wurde zuvor bereits mehrfach erwähnt. Im Rahmen der Entwicklung von Methoden und Vorgehensmodellen für Software-Entwicklung hat sich über die Jahre ein eigenes Betätigungsfeld heraus kristallisiert. Hierbei handelt es sich um Software-Architektur. Diese kann nach Hofmeister et al., 2001 [48] als Brücke zwischen Anforderungsanalyse und Implementierung verstanden werden. Es existiert zwar keine feste, allgemein akzeptierte Definition, man kann jedoch in der verfügbaren Literatur wie zum Beispiel in Bosch, 2000 [22], Hofmeister et al., 2001 [48], Posch et al., 2007 [93], Reussner et al., 2009 [96] und Starke, 2005 [109] einen grundsätzlichen Konsens darüber feststellen, was Software-Architektur ist und was zu deren Betrachtungsumfang gehört. Die daraus abgeleitete und im Rahmen dieser Arbeit gültige Definition lautet:

> *Software-Architektur beschreibt die Struktur eines Software-Systems, welche die Architekturbausteine, deren extern sichtbare Eigenschaften sowie die Schnittstellen zwischen diesen umfasst. Software-Architekturen abstrahieren von Details und sind hierarchisch aufgebaut.*

Durch explizite Betrachtung der Struktur – dazu gehören Arbeitsschritte wie Analyse, Entwurf und Bewertung – des Software-Systems vor der Implementierung soll sichergestellt werden, dass dieses alle daran gestellten Anforderungen erfüllen kann. Zu nennen sind hier insbesondere die eingangs erwähnten nicht-funktionalen Anforderungen. Bei komplexen Software-Systemen ist die Entwicklung häufig auf mehrere Personen und Teams, teilweise über verschiedene Standorte und Unternehmen, verteilt. Ohne Vorgabe einer übergeordneten Struktur, die all diese Anforderungen adressiert, ist die Gefahr groß, dass das Endprodukt den Anforderungen nicht gerecht wird. Änderungen sind zu diesem Zeitpunkt jedoch nicht mehr oder nur unter extrem großen Aufwand zu bewerkstelligen. Die vorgelagerte und entwicklungsbegleitende Tätigkeit der Entwicklung einer Software-Architektur wird somit zu einem entscheidenden Erfolgsfaktor eines Projekts.

Um die von der Anwendung von Software-Architektur erhofften Ziele erreichen zu können, muss diese als Tätigkeit und Arbeitsprodukt verpflichtend in ein Vorgehensmodell integriert werden. Ein wichtiger Teilaspekt davon ist die Art der Dokumentation und der Modellierung. Eine Software-Architektur wird typischerweise über unterschiedliche Sichten und Standpunkte dargestellt. Bezüglich Anzahl, Inhalt und Zusammenspiel dieser gibt es viele verschiedene Ansätze, beispielsweise in Clements et al., 2005 [29] und Kruchten, 1995 [66]. Das Institute of Electrical and Electronics Engineers (IEEE) hat mit dem Dokument IEEE 1471, 2000 [53] einen Rahmen veröffentlicht, in welchem sich Beschreibungen von Software-Architektur bewegen sollen. Es wird bewusst ausreichend Spielraum zur Anpassung an die spezifischen Anforderungen der jeweiligen Domäne und des Projekts gelassen. Weitere wichtige Aspekte sind die eingesetzte Notation und das dazu passende Modellierungs-Werkzeug. Existierende Ansätze variieren hierbei von komplett eigenständigen Notationen und Werkzeugen bis hin zur Verwendung offener

Standards wie der UML. Da Letztgenannte über so genannte Profile mannigfaltige, jedoch vergleichsweise einfach umsetzbare, Möglichkeiten der Anpassung des Standards an spezifische Bedarfe bieten, werden diese Ansätze immer zahlreicher. Bild 1.3 zeigt eine Darstellung einer Software-Architektur unter Einsatz der Standard-UML.

Vorgaben zu einer Notation von Software-Architekturen werden auch als Architektur-Beschreibungssprache (Englisch: Architecture Description Language (ADL)) bezeichnet. In Medvidovic et al., 2000 [79] ist eine Übersicht über existierende ADLs zusammengestellt und ausführlich in einem Framework verglichen.

In der folgenden Tabelle 1.2 werden relevante Ansätze zu Architektur-Beschreibungen und Vorgehensmodellen bezüglich Software-Entwicklung in der Automobil-Domäne vorgestellt und tabellarisch verglichen. Weitere Details können den jeweils aufgeführten Quellen und die ausführlichen Bezeichnungen der Ansätze dem Abkürzungsverzeichnis entnommen werden.

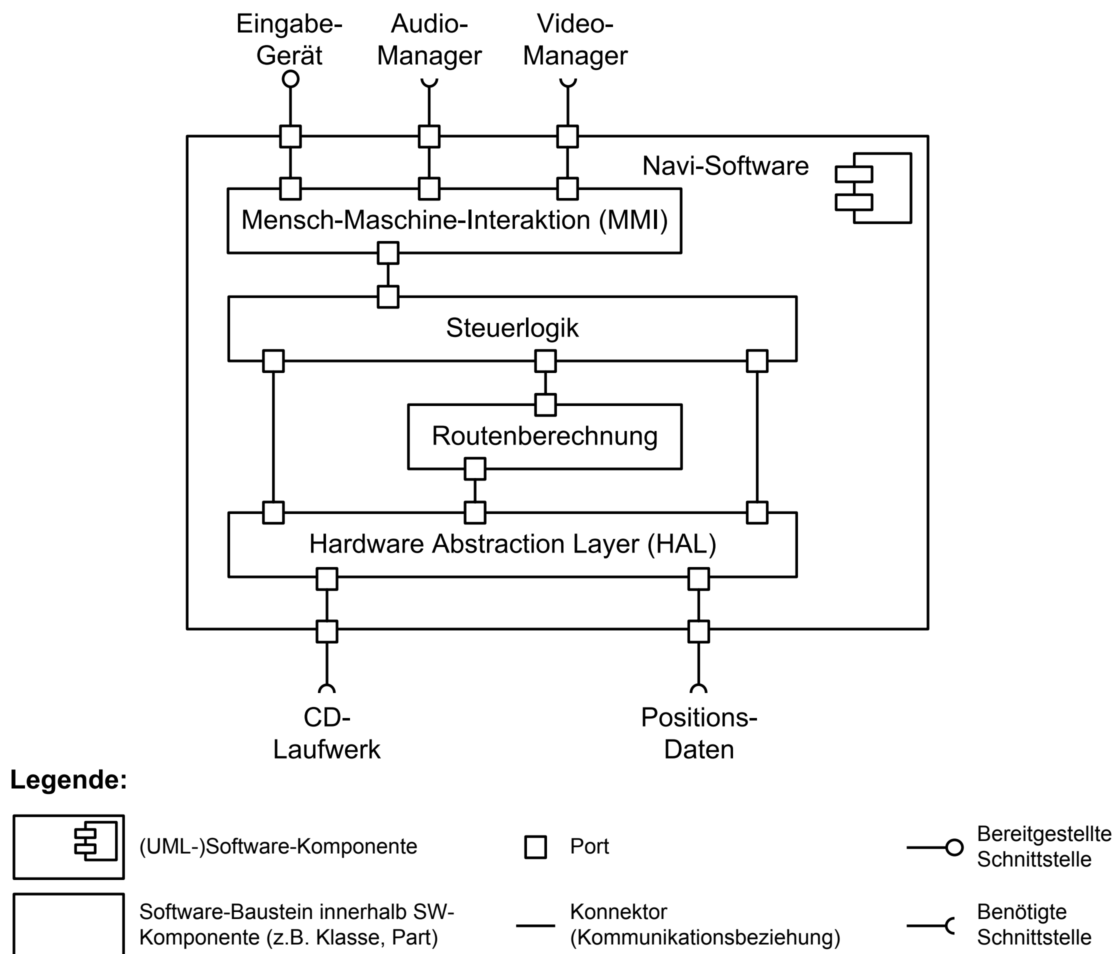

Bild 1.3: Software-Architektur eines Navigations-Systems [93]

Tabelle 1.2: Gegenüberstellung von ausgewählten Architektur-Beschreibungen und Vorgehensmodellen in der Automobil-Domäne

Bezeichnung	Quelle	Beschreibung des Ansatzes
AADL	[37][59]	Entstammt der Luftfahrt. Beschreibung von Hardware- und Software-Komponenten sowie deren Schnittstellen. Keine Definition von expliziten Abstraktions-Ebenen. Software immer konkreter Hardware zugeordnet. Modellierung von Verhalten, insbesondere Timing, und sicherheitskritischen Aspekten. Notation als UML-Profil verfügbar.
AUTO-FOCUS	[51][100]	Werkzeug zum Einsatz der FOCUS-Notation. Verbindung von formalen, mathematischen Beschreibungen mit einer intuitiven, grafischen Modellierung für verteilte Systeme. Proprietäres Metamodell. Fokussierung auf die Modellierung von Verhalten.
AutoMoDe	[13][129]	Grafische Beschreibungstechniken im Rahmen eines Vorgehensmodells. Umsetzung in einem Werkzeug erfolgt. Definition von 4 Prozess-Ebenen (Funktionsarchitektur, Komponentenarchitektur, Prozessarchitektur, Implementierung). Fokus auf die Prozessschritte: Reengineering, Refactoring, Refinement. Teilweise automatisierte Übergänge, z.B. in ASCET, möglich.
Automotive UML & AML	[17]	Die AML legt Modellierungselemente zur Beschreibung von Automotive Software-Architekturen fest. Die Automotive UML ist ein UML-Profil zur Anwendung dieser. Unterschiedliche Abstraktions-Ebenen: Logische und Technische Architektur. Logische Architektur identisch zu Funktions-Architektur. Anwendbarkeit nur an kleinen Beispielen erprobt. Keine automatisierten Übergänge.
CAR-DL	[127]	Vorgehensmodell zur Entwicklung von Software in der Automobil-Domäne. Definiert 4 Abstraktions-Ebenen (Service Level, Functional Level, Logical Cluster Level, Platform Level). Keine automatisierten Übergänge, jedoch formale Beschreibung der Modellierungselemente und deren erlaubter Wechselwirkungen. Sehr ähnlich der EAST-ADL/EAST-EEA, jedoch ohne deren Kompatibilität zu AUTOSAR.
COLA	[69][70]	Gleichzeitige Modellierung von hierarchischen Funktions-Architekturen und Verhalten von deren Bausteinen. Abstraktion von konkreten Realisierungsdetails für beide Aspekte so lange wie möglich. Proprietäres, umfangreiches Metamodell und Werkzeug vorhanden. Verhaltensmodellierung auf Granularität von kommerziellen Werkzeugen wie ASCET und Simulink. Ausgewiesener Vorteil ist die Durchgängigkeit über Architektur, realisierungsunabhängigem Verhalten und realisierungsabhängiger Implementierung in einem Ansatz mit (teil- und voll-)automatisierten Übergängen.
EAST-ADL & EAST-EEA	[20][59]	Vorgehensmodell zur Entwicklung von E/E-Systemen in der Automobil-Domäne. Definiert 4 Abstraktions-Ebenen (Vehicle Level, Analysis Level, Design Level, Implementation). Darüber hinaus Anteile zur Anforderungs- und Variantenmodellierung sowie Funktions-Sicherheit. Vorgehensmodell in Gänze oder in Teilen anwendbar. Implementierung erfolgt vollständig über AUTOSAR-Ansatz. Daher vollständige Kompatibilität bezüglich Notation. UML-Profil verfügbar. Keine automatisierten Übergänge.

AUTOSAR

Automotive Open System Architecture (AUTOSAR) ist eine weltweite Partnerschaft zwischen Automobilherstellern, Zulieferern und Werkzeug-Herstellern. Primäres Ziel ist die Schaffung eines offenen, frei verfügbaren Industriestandards für E/E-Architekturen in der Automobil-Domäne, um den hohen Herausforderungen zu begegnen. AUTOSAR adressiert hierbei grundsätzlich den gesamten Entwicklungsprozess von E/E-Komponenten und -Architekturen. Bezüglich konkret ausgearbeiteter Vorschläge und Standards sind jedoch die Entwicklung und Standardisierung von Software-Anteilen mit großem Abstand am weitesten fortgeschritten. Die für die vorliegende Arbeit besonders relevanten Aspekte werden im Folgenden vorgestellt. Gegenstand der Betrachtung ist das Release 4.0 aus 2012 [1].

Die allgemeine Zielsetzung von AUTOSAR im Rahmen der Software-Entwicklung kann am besten mit dem folgenden, offiziellen Motto der Initiative verstanden werden:

> *„Cooperate on Standards, Compete on Implementation.“*

Es soll eine Basis geschaffen werden, die es allen an der Entwicklung beteiligten Partnern ermöglicht, effektiv und effizient zusammenzuarbeiten. Die großen wiederkehrenden Aufwände an Software-Anteilen, die keine für den Kunden erlebbare Funktionalität repräsentieren, sollen auf ein Minimum reduziert werden. Hierbei sind beispielsweise zu erwähnen: Betriebssysteme, Treiber, Adapter, Diagnosedienste. Diese Software-Anteile sind hochgradig abhängig von der konkret eingesetzten Hardware und erfordern daher aufgrund der Heterogenität der verwendeten Plattformen sowohl innerhalb eines Automobilherstellers als auch Hersteller übergreifend einen hohen Aufwand. Je stärker dieser Aufwand reduziert werden kann, desto mehr kann der Automobilhersteller seine Ressourcen in die Entwicklung der Kundenfunktionen verlagern, welche für ihn im Rahmen der Differenzierung vom Wettbewerb entscheidend ist. AUTOSAR stellt dabei die folgenden Aspekte und deren Definitionen besonders hervor:

- **Modularität:** Modularität von Elementen ermöglicht das Tailoring von Software entsprechend der individuellen Anforderungen von Plattformen.
- **Skalierbarkeit:** Skalierbarkeit von Funktionen ermöglicht die Anpassung von allgemeingültigen Software-Modulen an unterschiedliche Plattformen zur Vermeidung der Verbreitung von unterschiedlicher Software mit ähnlicher Funktionalität.
- **Portabilität:** Die Ausnutzung der vorhandenen Ressourcen in einer Fahrzeug-Architektur kann optimiert werden, indem eine Funktionalität einfach auf unterschiedlichen Plattformen ausgeführt werden kann.
- **Wiederverwendbarkeit:** Wiederverwendbarkeit von Funktionen hilft die Produktqualität und -zuverlässigkeit zu verbessern und eine markentypische Ausprägung über unterschiedliche Produktlinien sicherzustellen.

Zur Lösung der adressierten Aspekte schlägt AUTOSAR eine Schichten-Architektur für Steuergeräte vor, wie sie in Bild 1.4 dargestellt ist. Entscheidend sind hierbei die Abstraktion von Hardware sowie die Standardisierung von Software und Schnittstellen. Das Bild

ist eine vereinfachte Darstellung der Schichten. So ist beispielsweise die Basis-Software noch in definierte Komponenten wie Betriebssystem, Services und Communication unterteilt, welche wiederum definierte Schnittstellen zu den umgebenden Schichten ausweisen. Wichtig zum Verständnis ist, dass eine Schicht immer von der darunter liegenden abstrahiert. Das heißt, alle in dieser Schicht befindlichen Software-Komponenten – und natürlich alle in darüber liegenden Schichten – funktionieren unabhängig von den Besonderheiten der darunter liegenden Schicht(en). So kann immer dasselbe Betriebssystem verwendet werden, da über das „Microcontroller Abstraction Layer“ von der konkreten Hardware abstrahiert wurde.

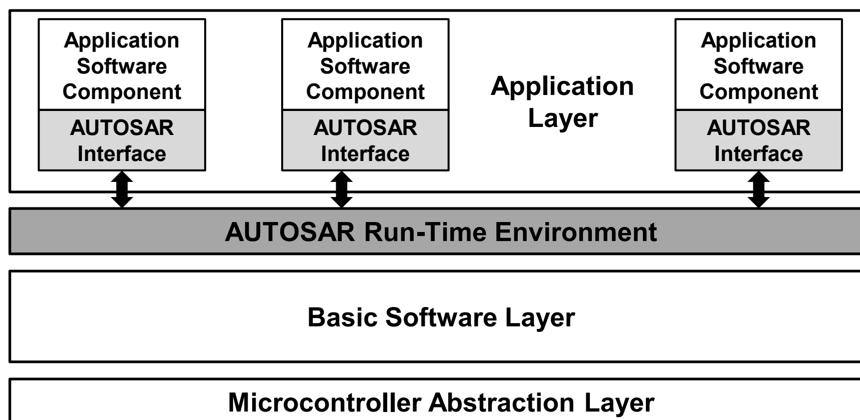

Bild 1.4: Vereinfachte Darstellung der AUTOSAR Schichten-Architektur [1]

Besonders hervorzuheben ist die Run-Time Environment (RTE). Sie kapselt die Komponenten der Anwendungs-Software (so genannte AUTOSAR Software Components (SWC)) von den übrigen Schichten und stellt zu diesem Zweck standardisierte Schnittstellen, die so genannten AUTOSAR-Interfaces, bereit. Dadurch wird ermöglicht, dass die Komponenten auf jedem AUTOSAR-konformen Steuergerät im Fahrzeug verwendbar sind. Diese können zuvor in einer übergeordneten Entwicklungsumgebung erstellt werden. Solange noch keine konkreten Steuergeräte vorliegen, sind sie durch den Virtual Functional Bus (VFB) verbunden. Dieser repräsentiert einen abstrakten Bus als Kommunikationskanal. Bei der Verteilung der Software-Komponenten auf konkrete Plattformen wird der VFB durch eine oder mehrere RTE implementiert und konkrete Telegramme auf realen System-Bussen definiert. Die Beschreibung von AUTOSAR-SWC und der zugehörigen AUTOSAR-Interfaces ist durch die Spezifikation eng vorgegeben. Es existieren standardisierte Templates und Austauschformate. Die innere Struktur und Implementierung von Komponenten ist nicht Gegenstand der Standardisierung. Die von AUTOSAR definierten Modellierungselemente und deren erlaubte Interaktionen werden in einem Metamodell festgelegt, das in Form eines UML-Profils frei verfügbar ist.

Gegenwärtig ist festzustellen, dass die vorgestellten Ansätze der Initiative eine überwältigende Anwendung erfahren. Immer mehr Steuergeräte werden so entwickelt, dass

die Schichten-Architektur darauf verfügbar ist. Die darauf ausgeführte Anwendungs-Software ist zu einem großen Teil als AUTOSAR-SWC realisiert, welche durch den Automobilhersteller selbst oder einen Zulieferer entwickelt wird. Dieser Zulieferer kann, muss aber nicht, der Zulieferer des Steuergeräts sein. Die versprochenen Vorteile der Standardisierung werden somit bereits in einem hohen Maße in der Praxis bestätigt.

Modelltransformation

Im Rahmen der modellgetriebenen Software-Entwicklung sowie für die Realisierung von durchgängigen Vorgehensmodellen mit automatisierten Übergängen zwischen Prozessschritten sind Modelltransformationen ein notwendiger Bestandteil. Das allgemeine Ziel ist unterschiedliche Modelle mit unterschiedlichem Informationsgehalt oder Betrachtungsgegenständen ineinander zu überführen. In der Literatur gibt es dazu eine Vielzahl an Ansätzen, die man unter anderem nach folgenden Kriterien unterscheiden kann: formale oder semi-formale Transformations-Regeln; automatische, semi-automatische oder manuelle Durchführung. Kühl et al., 2006 [72] und Reichmann, 2005 [94] fassen dies zusammen. Für diese Arbeit kommen nur Ansätze mit formalen Regeln und mindestens semi-automatischer Durchführung in Frage. Die Model Driven Architecture (MDA), welche von der Object Management Group (OMG) veröffentlicht wurde [85], definiert einen offenen Standard, welcher unabhängig von Domänen und Werkzeugen, Ideen und Vorgehensweisen zu Modelltransformationen beschreibt. Eine Einführung liefern Petrasch et al., 2006 [92].

Modelltransformationen benötigen formale Beschreibungen für die Transformations-Regeln und definierte Formate für die Speicherung und Modifikation der Modelle. In diesem Kontext hat sich die Extensible Markup Language (XML) [122] als universell einsetzbares Datenformat und die XML Metadata Interchange (XMI) [90] als Metamodell basiertes Austauschformat inzwischen weit verbreitet. Das Metamodell der XMI baut dabei ebenso auf der Meta Object Facility (MOF) [86] auf wie jenes der UML. Für die Manipulation und damit auch die Transformation von XML-Dateien gibt es verschiedene Ansätze. Von diesen seien exemplarisch die Java Architecture for XML Binding (JAXB) [113] und Extensible Stylesheet Language Transformations (XSLT) [123] erwähnt.

Für eine durchgängige Software-Entwicklung wird zur Modellierung von Software-Architekturen häufig auf die UML zurückgegriffen. Um diese Modelle für die Anreicherung mit Verhalten in etablierten CASE-Werkzeugen weiterverarbeiten zu können, existieren Ansätze zur Überführung wie von Hachmeister, 2007 [44].

Bewertung von Software-Architektur

Der Nutzen von Software-Architekturen ist offensichtlich unbestritten. Wie kann jedoch die Eignung einer Software-Architektur für einen spezifischen Anwendungsfall überprüft werden? Dazu müssen Bewertungsmethoden existieren. Während es für Software solche Ansätze schon seit längerer Zeit gibt (Boehm et al., 1978 [21], ISO9126-1, 2001 [54],

McCall, 1994 [78]), befindet sich die Entwicklung entsprechender Methoden für Software-Architekturen noch in einem frühen Stadium. Es existieren vorherrschend qualitative, szenario-basierte Ansätze wie die von Bengtsson et al., 1998 [18] sowie Kazman et al., 1994 [60] und 1998 [61]. Quantitative Ansätze existieren nur in einem unzureichenden Maße und richten ihren Fokus bisher lediglich auf bestimmte Aspekte oder Qualitätskriterien. Bosch et al., 2001 [23] fokussieren beispielsweise auf Wartbarkeit, Lung et al., 2000 [75] auf Robustheit. Umfassende, objektive und quantifizierte Metriken sind nicht anzutreffen. Den Versuch einer automatischen, quantifizierten Bewertung über Bayes'sche Netze unternehmen Gurp et al., 2000 [43]. Darüber hinaus existieren keine Ansätze, welche speziell für die Bewertung von Software-Architekturen in der Automobil-Domäne angepasst sind. Quantifizierte Verfahren innerhalb der Domäne findet man beispielsweise nur für vergleichbare, jedoch eigenständige und nicht direkt übertragbare Fragestellungen wie die Bewertung von Bordnetz-Architekturen in Florentz, 2007 [38] sowie Florentz et al., 2006 [39].

Ein Gesamtüberblick über bestehende Ansätze kann Abowd et al., 1996 [2], Babar et al., 2004 [9] und Clements et al., 2002 [30], eine Diskussion und Einordnung in den Kontext dieser Arbeit kann Ahrens et al., 2013 [7] (1.2 Related Work) entnommen werden. Vielversprechend ist das in Losavio et al., 2003 [74] dargestellte Qualitätsmodell, das an die ISO9126-1 angelehnt ist. Für Erweiterungen, spezifische Definitionen und die Erarbeitung von quantitativen Metriken bildet das Qualitätsmodell eine gute Ausgangsbasis.

Strukturierung von Funktionen und Software im Automobil

Die zuvor beschriebenen Aspekte können als das „Handwerkszeug" für die an der Entwicklung beteiligten Personen verstanden werden. Sie sind der notwendige Rahmen, der die Erreichung der (funktionalen und nicht-funktionalen) Eigenschaften unter den genannten Randbedingungen prinzipiell möglich macht. Damit diese produktiv zum Einsatz kommen können, sind konkrete Richtlinien zur Modellierung auf den unterschiedlichen Abstraktions-Ebenen notwendig. An dieser Stelle sollen Strukturierungsvorgaben für Funktions- und Software-Architekturen diskutiert werden, die in der Literatur anzutreffen sind.

Bezüglich Funktions-Architektur wird sich im Rahmen dieser Diskussion auf Ansätze zu Fahrerassistenzsystemen (FAS) beschränkt. Hier existieren bisher nur wenige veröffentlichte Beiträge mit konkreten Empfehlungen zur Strukturierung. Handmann et al., 1999 [45] beschreiben eine lösungsneutrale FAS-Architektur auf einer sehr abstrakten Ebene. Andere Ansätze wie Miller et al., 2003 [80] beschreiben Architekturen für Prototypen-Aufbauten, welche auf Serienfahrzeuge nicht direkt anwendbar sind. In Steinle et al., 2005 [110] wird mit dem Längsdynamikmanagement (LDM) ein Ansatz vorgestellt, wie von konkreten Antriebsvarianten abstrahiert werden kann, um diese effizient zu beherrschen.

Bezüglich Software-Architektur kann auf ein größeres Repertoire bestehender Ansätze zurückgegriffen werden. Die bereits vorgestellte Basis-Literatur nennt dabei lediglich

allgemeine Entwurfsprinzipien, Heuristiken und Einflussfaktoren. Es fehlen Vorgaben, wie diese für konkrete Problemstellungen im Detail zu gewichten und anzuwenden sind, um einen konkreten Architektur-Entwurf zu erhalten. Diese Lücke sollen so genannte Architektur-Muster schließen. Es gibt inzwischen eine Fülle an bewährten, gut dokumentierten und damit wiederverwendbaren Lösungen auf den unterschiedlichen Granularitätsebenen einer Software-Architektur: Beck et al., 1994 [16], Buschmann et al. 1996 [28], Gamma et al., 2008 [40].

Für Software-Architekturen in der Automobil-Domäne im Allgemeinen beziehungsweise in der Fahrerassistenz im Speziellen finden sich keine vergleichbaren Kataloge bewährter Muster. Es gibt jedoch einige Ansätze, die allgemein gehaltene Strukturierungsvorgaben machen, um die speziellen Herausforderungen der Automobil-Domäne zu adressieren. Diese Ansätze finden sich in Damm et al., 2005 [32], Hardung et al., 2004 [46], Heinecke et al., 2008 [47] und Schmidt et al., 2003 [103]. Sie beschreiben, wie eine Software-Architektur als Endprodukt aussehen soll. Der Prozess zum Erreichen dieser Struktur wird jedoch nicht explizit beschrieben, beziehungsweise es wird implizit vorausgesetzt, dass der Anwender von Grund auf mit der Modellierung beginnt. Ein wichtiger Aspekt in der Software-Entwicklung – und auch für die vorliegende Arbeit – ist jedoch die Wartung, Pflege und Weiterentwicklung von bestehenden Software-Systemen und -Architekturen. Insbesondere bei komplexen Systemen werden spezielle Techniken benötigt, um eine bestehende Software-Architektur in eine neue Struktur zu überführen. Der Prozess wird auch als Evolution oder Refactoring von Software-Architekturen bezeichnet und bei Krahn et al., 2009 [65], Thomson et al., 1994 [116] und Tichelaar et al., 2000 [117] beschrieben.

1.3 Ziele der Arbeit

In diesem Abschnitt sollen die konkreten Ziele der Arbeit dargelegt werden. Sie ergeben sich mit Blick auf die gegebene Problemstellung aus jenen Aspekten des Stands der Technik, die noch deutliches Verbesserungspotenzial aufweisen. Über alle im Rahmen des vorherigen Abschnitts 1.2 diskutierten Themenbereiche lassen sich insbesondere die folgenden Punkte zusammenfassen, die als nicht ausreichend im betrachteten Kontext angesehen werden können:

- **Keine explizite Modellierung von (realisierungsunabhängigen) Software-Architekturen:** Software-Architektur wird in vielen Ansätzen von Vorgehensmodellen, welche für die Automobil-Domäne entwickelt worden sind, nur implizit durch Gruppieren von Bausteinen aus Funktions-Architekturen gebildet. Kriterien zur Strukturierung dieser beiden Architekturarten sind jedoch unterschiedlich, um die jeweiligen Ziele zu erreichen. Ansätze, die explizit vorgeben Software-Architekturen zu modellieren, beziehen diese jedoch bereits auf konkrete technische Plattformen. Eine abstrakte, realisierungsunabhängige Betrachtung fehlt.

- **Geringer Grad an Formalisierung und Durchgängigkeit:** Die Ansätze weisen zu großen Teilen nur modellbasierten statt modellgetriebenen Charakter auf. Für eine effektive und effiziente Entwicklung, insbesondere von komplexen Software-Systemen, sind letztgenannte Aspekte deutlich zu verstärken. Hierzu zählen insbesondere automatisierte Übergänge zwischen Modellierungs- und Prozess-Ebenen.
- **Fehlender Nachweis von Praxistauglichkeit:** Die Ansätze werden nur mit Hilfe kleiner, akademischer Beispiele erprobt. Es fehlt der Nachweis, dass diese im Rahmen der Serienentwicklung von komplexen Software-Systemen einsetzbar sind. Einige dieser Ansätze sind zu allgemein beschrieben, als dass sie in realen Prozessen anwendbar sind. Andere Ansätze beschreiben umfangreiche, in sich vollständig abgeschlossene Vorgehensmodelle und Prozesse. Ohne einen Nachweis der Praxistauglichkeit werden diese von Automobilherstellern jedoch nicht übernommen, um damit Bestehende zu ersetzen. Bis auf AUTOSAR hat daher keiner der vorgestellten Ansätze Einzug in die Industrie gehalten.
- **Keine konkreten Vorgaben zur Strukturierung von Funktions- und Software-Architekturen:** Die Ansätze beschreiben nur das allgemeine Vorgehen beziehungsweise die grundsätzlich im Rahmen der Modellierung erlaubten Elemente. Es fehlen konkrete Vorgaben, Erfahrungen und Muster. Dies gilt insbesondere für Software-Architektur. AUTOSAR legt bezüglich dieser Fragestellung nur den äußeren methodischen Rahmen fest, wie Software-Komponenten erhalten und nach außen beschrieben werden können. Wie die Software innerhalb dieser strukturiert ist, wird nicht betrachtet.
- **Fehlende Möglichkeit der objektiven Bewertung von Software-Architekturen:** Es fehlen sowohl allgemeine als auch spezifische Ansätze, um Software-Architekturen objektiv und quantifiziert auf ihre Eignung für das jeweilige Problem bewerten zu können.
- **Keine Ansätze speziell für Fahrerassistenz:** Ansätze, die auf die besonderen Herausforderungen der Entwicklung von Fahrerassistenzsystemen abzielen, existieren generell nicht. Dies gilt für alle zuvor genannten Teilaspekte.

Die konkreten Ziele der Arbeit, die sich aus der bereits ausgeführten Problemstellung ergeben, sollen im Folgenden detailliert vorgestellt werden. Allgemein gilt, dass die methodische Betrachtung immer am Beispiel von Fahrerassistenzsystemen der Längsführung erfolgt. Wichtig ist hierbei anzumerken, dass sich alle Ziele und deren methodische Bearbeitung im Bereich der nicht-funktionalen Anforderungen bewegen. Das kundenerlebbare Verhalten bestehender und künftiger Kundenfunktionen muss zwingend unverändert erhalten bleiben und der gültigen Funktions-Spezifikation entsprechen. Die Erfüllung der funktionalen Anforderungen ist also immer gegeben, auch wenn an späterer Stelle auf diese nicht explizit eingegangen wird.

Im bisher gelebten Entwicklungsalltag werden Funktions-Architekturen nur teilweise und Software-Architekturen gar nicht explizit modelliert. Der erste Schritt ist somit zu-

nächst das Bearbeiten und Etablieren dieser zwei Arbeitsergebnisse. Dabei ist ausdrücklich gefordert, beide separat zu verfolgen. Es gilt festzulegen, nach welchen Kriterien die jeweiligen Architekturarten zu strukturieren und zu modellieren sind. Dies soll zunächst für die Automobil-Domäne übergreifend geschehen.

Ziel (1): Ausarbeitung von Kriterien für die Modellierung und Strukturierung von Funktions- und Software-Architekturen in der Automobil-Domäne.

Diese Überlegungen finden sich in Kapitel 2. Im weiteren Verlauf steht Software-Architektur im Fokus, da hier das größte Verbesserungs-Potenzial im Hinblick auf die Erfüllung der nicht-funktionalen Anforderungen erwartet wird. Es ist eine Notation beziehungsweise Methode zur Modellierung von Software-Architekturen zu entwickeln, die den besonderen Ansprüchen von Fahrerassistenzsystemen gerecht wird. Die Software-Architektur soll dabei lange von Realisierungsdetails abstrahieren und erst so spät wie möglich in eine konkrete Realisierung übergehen. Dieser Übergang soll formal und automatisierbar sein. Um die Güte der Software-Architektur kontinuierlich überprüfen und die Architektur gezielt weiter entwickeln zu können, muss eine Methode zur objektivierten Bewertung von verschiedenen Ansätzen und Entwürfen erarbeitet werden. Sie ist mit quantifizierten Software-Architekturmetriken zu realisieren. Zusammengefasst stehen für diese Ansprüche die Ziele (2) und (3):

Ziel (2): Entwicklung einer Methode zur Modellierung und automatisierten Weiterverarbeitung von komplexen Software-Architekturen in der Automobil-Domäne am Beispiel von Fahrerassistenzsystemen.

Ziel (3): Entwicklung eines automatisiert anwendbaren Bewertungsverfahrens für Software-Architekturen in der Automobil-Domäne unter Verwendung von objektiven, das heißt quantifizierten, Software-Architekturmetriken.

Die Bearbeitung dieser beiden Ziele kann Kapitel 4 entnommen werden. Darüber hinaus ist es nicht Ziel der Arbeit, den aktuellen Entwicklungsprozess in Gänze neu zu gestalten. Dies ist einerseits vom Umfang her im Rahmen der Bearbeitung nicht möglich, andererseits ist die Praxis-Akzeptanz in der Automobil-Domäne gegenüber einem völlig neuen Prozess erfahrungsgemäß zunächst gering. An bewährten Prozessen – auch wenn sie bekanntermaßen Schwächen aufweisen, dafür aber schon in vielen Projekten schlussendlich zum Erfolg geführt haben – wird festgehalten. Für einen gänzlich neuen Prozess müssten zahlreiche erfolgreiche Evaluierungen an Praxisbeispielen durchgeführt werden, bis dieser im Alltag der Serienentwicklung Einzug findet. Der Ansatz der vorliegenden Arbeit ist daher innerhalb des bewährten Entwicklungsprozesses durch punktuelle Änderungen deutliche Verbesserungen zu erzielen. Die Integrierbarkeit der Änderungen in den bestehenden Prozess ist nachzuweisen. Im Vordergrund stehen dabei die Verbesserung von Durchgängigkeit und Formalismus mit möglichst automatisierten Übergängen zwischen Prozessschritten und Arbeitsergebnissen. Für eine zielgerichtete Erarbeitung von Verbesserungen ist zunächst eine Analyse des Ist-Prozesses durchzuführen. Im wei-

teren Verlauf der Arbeit wird synonym für den Begriff „Entwicklungsprozess“ auch die Bezeichnung „Vorgehensmodell zur Entwicklung“ eingesetzt.

Ziel (4): Analyse und Dokumentation des heutigen Vorgehensmodells zur Entwicklung sowie der Funktions- und Software-Architektur der Fahrerassistenzsysteme der Längsführung und Herausarbeiten der Schwachstellen.

Ziel (5): Sicherstellung der Integrierbarkeit aller neu erarbeiteten Methoden und Prozessschritte in das Vorgehensmodell.

Kapitel 3 adressiert die Ziele (4) und (5). Über die zuvor genannten Ziele (1) und (3) wird erarbeitet, wie Software-Architekturen in der Automobil-Domäne zu strukturieren sind und wie durch eine objektive Bewertungsmethode ein gezieltes, iteratives Vorgehen erreicht wird, um gute Software-Architekturen im Hinblick auf die Erfüllung der nicht-funktionalen Anforderungen zu erhalten. Darüber hinausgehend soll detailliert untersucht werden, welche Kriterien, Richtlinien und Muster zur Strukturierung von Software-Architekturen von Fahrerassistenzsystemen im Speziellen gelten. Dies geschieht am Beispiel der Fahrerassistenzsysteme der Längsführung.

Ziel (6): Erarbeitung von konkreten Kriterien zur Strukturierung von Software-Architekturen von Fahrerassistenzsystemen der Längsführung.

Alle bis hierhin erarbeiten Methoden, Prozessschritte und Erkenntnisse sollen in der Praxis anhand von Fahrerassistenzsystemen der Längsführung evaluiert werden. Dabei ist zunächst die real vorhandene, gemeinsame Software-Architektur dieser Systeme nach den entwickelten Kriterien neu zu strukturieren und damit gemäß der identifizierten (nicht-funktionalen) Eigenschaften zu optimieren. Im Rahmen der Neustrukturierung kommen alle relevanten Beiträge der zuvor erwähnten Ziele zum Einsatz. Im Anschluss daran soll nachgewiesen werden, dass die strukturellen Änderungen tatsächlich einen messbaren Nutzen für eine konkrete Implementierung haben. Da die nicht zufriedenstellende Skalierbarkeit der Ausgangs-Struktur die ausgemachte, dominierende Schwachstelle darstellt, soll an einem geeigneten Beispiel die verbesserte Skalierbarkeit der überarbeiteten Struktur in Bezug auf benötigte Speicher- und Laufzeit-Ressourcen der lauffähigen Implementierung nachgewiesen werden.

Ziel (7): Durchführung einer Neustrukturierung der Software-Architektur von Fahrerassistenzsystemen der Längsführung unter Anwendung der zuvor entwickelten Methoden.

Ziel (8): Nachweis der Wirksamkeit und des Nutzens der neuen Struktur anhand der Gegenüberstellung der Implementierungen eines geeigneten Fahrerassistenzsystems aus der alten und der neuen Struktur.

Die Ziele (6) und (7) werden gemeinsam in Kapitel 5 betrachtet, während Kapitel 6 das abschließende Ziel (8) erfüllt.

2 Strukturierung von Funktions- und Software-Architekturen im Automobil

Funktions- und Software-Architekturen wurden im vorangegangenen Kapitel 1 bereits erwähnt, sie sind im weiteren Verlauf der Arbeit von besonderer Bedeutung. Daher erfolgt im vorliegenden Kapitel 2 eine Vertiefung der beiden Architekturarten im Kontext der Automobil-Domäne. Das Kapitel soll eine Einführung geben, welche Ziele jeweils verfolgt werden und wie sie sich bei der Modellierung voneinander abgrenzen sollten, so dass beide Architekturarten einen eigenständigen, wichtigen Beitrag bei der Entwicklung mechatronischer Systeme in dieser Domäne leisten können. Ein Schwerpunkt liegt dabei auf der Angabe von Kriterien zur Strukturierung. Alle beschriebenen Aspekte beruhen, sofern nicht ausdrücklich mit einer Quelle gekennzeichnet, auf selbst gemachten Erfahrungen, Beobachtungen und Erkenntnissen aus der Entwicklungspraxis.

2.1 Begriffsdefinitionen und Ziele der Modellierung

Der Begriff Funktions-Architektur wird im Rahmen dieser Arbeit wie folgt definiert:

> *Funktions-Architektur beschreibt die Struktur von Funktionen, welche die Architekturbausteine, deren extern sichtbare Eigenschaften sowie die Schnittstellen zwischen diesen umfasst. Funktions-Architekturen abstrahieren von Details und sind hierarchisch aufgebaut.*

Die Definition wurde auf dieser Granularität bewusst sehr ähnlich zur Software-Architektur gehalten. In beiden stehen die Abstraktion von Details sowie ein hierarchischer Aufbau unter Verwendung von bestimmten (Architektur-)Bausteinen und definierten Schnittstellen zwischen diesen im Fokus. Im Falle der Funktions-Architektur ist der Betrachtungsgegenstand eine Funktion als eine von einer konkreten Realisierung abstrahierende, jedoch vollständige, Wirkkette. Diese wird typischerweise durch das Zusammenspiel von Hard- und Softwarebausteinen gebildet, die in Summe die oben genannten Architekturbausteine umfassen. Eine einzige Funktions-Architektur kann darüber hinaus auch mehrere Funktionen und Wirkketten abbilden. Im Gegensatz dazu ist der Betrachtungsgegenstand einer Software-Architektur auf die Software-Anteile solcher Wirkketten beschränkt. Architekturbausteine sind dort somit ausschließlich Softwarebausteine.

Da der Betrachtungsgegenstand jeweils ein anderer ist, differieren die mit der Modellierung der beiden Architekturarten verfolgten Ziele ebenfalls. Durch die Funktions-Architektur soll im Wesentlichen mindestens eins der beiden folgenden Ziele erreicht werden:

- Einordnen einer Kundenfunktion als Blackbox im logischen Zusammenspiel mit anderen Kundenfunktionen in einem übergeordneten Kontext (zum Beispiel „Fahrerassistenzsysteme im Gesamtfahrzeug“) oder
- Aufzeigen der inneren logischen Struktur einer Kundenfunktion als Whitebox.

Wichtig für die eingesetzte Methode der Modellierung ist somit die Durchgängigkeit über unterschiedliche Hierarchie-Ebenen. Sowohl in der Blackbox- als auch in der Whitebox-Darstellung wird von einer konkreten Realisierung vollständig abstrahiert. Dies ist mit „logischem Zusammenspiel“ beziehungsweise „logischer Struktur“ gemeint.

Kundenfunktionen können unterschiedliche Ausprägungen – und damit funktionale Skalierungen – aus Kundensicht aufweisen. So gibt es von der ACC-Funktion die seit den Anfangstagen bekannte Funktionsausprägung, welche typischerweise oberhalb circa 30 km/h verwendet werden kann (ACC 30+). Seit einigen Jahren gibt es darüber hinaus eine weitere, die eine zusätzliche Stop&Go-Funktion aufweist (ACC Stop&Go) und bis in den Stillstand nutzbar ist. In einer Funktions-Architektur würden sich diese Ausprägungen durch das Hinzufügen von logischen Systemkomponenten (Architekturbausteinen) wie „Standzielregler“ oder „Stillstandsmanagement“ im Falle ACC Stop&Go unterscheiden. Neben den bereits erwähnten, übergeordneten Zielen stehen bei der Modellierung von Funktions-Architekturen darüber hinaus die Folgenden im Fokus:

- Überführung der Kundenfunktion in eine logische Wirkkette,
- Festlegung von Systemgrenzen: Abgrenzung der logischen Wirkkette von seiner Umwelt und damit die Festlegung, welche Architekturbausteine innerhalb und welche außerhalb dieser Grenze liegen,
- Strukturierung der Wirkkette nach logischen, physikalischen Zusammenhängen: Dazu zählen das Finden von funktional zusammengehörigen Einheiten (Architekturbausteinen) und den logischen Schnittstellen zwischen diesen,
- Zuordnen der Architekturbausteine zu Hardware und Software,
- Erarbeiten einer Unterstruktur der Architekturbausteine, wo als nötig erachtet. Ein Architekturbaustein kann somit Blackbox oder Whitebox sein und
- Überblick schaffen über Wirkungsweise und Komplexität einer (Kunden-)Funktion bei verschiedenen Stakeholdern.

Unter „logischen Schnittstellen“ werden von der konkreten Umsetzung abstrahierende Interaktionen wie Verwendung von angebotenen Operationen oder Diensten, Informationsaustausch sowie Wechselwirkungen wie Verwendungsbeziehungen oder Abhängigkeiten verstanden. Zur Hardware zählen in diesem Kontext primär Aktorik und Sensorik. Ablaufumgebungen für die Software-Anteile werden erst in späteren, realisierungsnäheren Darstellungen betrachtet. Software ist somit in der Funktions-Architektur als ideal, das bedeutet immateriell, angenommen. Die Struktur und Granularität der Funktions-Architektur muss so gewählt werden, dass Bausteine eindeutig und ausschließlich Hardware oder Software zugeordnet werden können.

Mit Hilfe der Software-Architektur erfolgt die Überführung der im Rahmen der Funktions-Architektur identifizierten softwaredominanten Anteile in eine geeignete Software-Struktur. Die Software-Anteile sind vereinfacht gesagt die Anteile einer Wirkkette, die eine Funktions-Logik ohne unmittelbare Erfassung oder Manipulation der Umwelt repräsentieren. Auch in den Architekturbausteinen, die als dominierend per Hardware realisiert eingestuft wurden, ist nahezu immer auch Software in gewissem Maße anzutreffen. Diese kann im Rahmen der Software-Architektur mit betrachtet werden, wenn es im Rahmen der spezifischen Problemstellung als notwendig erachtet wird.

Da auf Software-Systeme und damit auch auf Software-Architekturen unterschiedliche Sichten und Abstraktionsgrade existieren, sollen in diesem Abschnitt die übergeordnet gültigen Ziele vorgestellt werden. Im späteren Verlauf der Arbeit (Kapitel 4) wird ein Ansatz vorgestellt, der die Software-Architektur als vollständig abstrahierend von Realisierungsdetails vorschlägt. Die weiterführenden, konkreten Ziele dieses Ansatzes werden dort diskutiert. In der Literatur werden Software-Architekturen zumeist als realisierungsabhängig innerhalb von konkreten Plattformen verstanden. Zusammengefasst ist die Motivation für den Einsatz von Software-Architektur die Folgende:

> *„Although having a good software architecture does not guarantee that a product meets its requirements, having a poorly designed or ill-defined architecture makes it nearly impossible to meet the product requirements.“* [48]

Die Software-Architektur adressiert die Vielzahl der nicht-funktionalen Anforderungen an ein Software-System. Durch geschickte Definition der Architekturbausteine und der Schnittstellen dazwischen, werden sie mehr oder weniger gut erfüllt. Typischerweise stellen die nicht-funktionalen Anforderungen an vielen Stellen widersprüchliche Ansprüche an die Architektur. Den Kompromiss zu finden, der die Gesamtheit dieser Anforderungen für das spezifische Problem bestmöglich erfüllt, ist die besondere Herausforderung des Software-Architekten. Neben diesem Kernziel werden die folgenden Nebenziele verfolgt:

- Verständnis schaffen und Ergebnisse sowie Kernwissen für unterschiedliche Stakeholder dokumentieren,
- Durchgängige Modellierung auf unterschiedlichen Detaillierungs-Ebenen,
- Durchgängige Modellierung auf unterschiedlichen Abstraktions-Ebenen,
- Entwurfs-Entscheidungen transparent machen und
- Komplexität beherrschbar machen.

Wie aus den vorangegangenen Ausführungen ersichtlich ist, verfolgen beide Architekturarten grundsätzlich andere Zielsetzungen. Sie leisten einen wichtigen eigenständigen Beitrag in der Entwicklung komplexer mechatronischer Systeme im Automobil. Aus diesem Grund wird im Rahmen dieser Arbeit der explizite und voneinander unabhängige Entwurf von Funktions- und Software-Architekturen verbindlich im Rahmen des eingesetzten Vorgehensmodells für die Entwicklung von Fahrerassistenzsystemen gefordert.

Es ist naheliegend, dass die Architekturarten aufgrund der diskutierten Unterschiede bei Betrachtungsgegenstand und Zielsetzung ebenfalls Unterschiede bezüglich anzuwendender Kriterien zur Strukturierung aufweisen. Diese werden im folgenden Abschnitt 2.2 im Detail vorgestellt. Damit wird die oben genannte Forderung zur expliziten Modellierung von beiden Architekturarten zusätzlich untermauert.

2.2 Allgemeine Kriterien zur Strukturierung

Die Diskussion der Kriterien zur Strukturierung von Funktions- und Software-Architekturen soll am Beispiel des bereits erwähnten Adaptive Cruise Control mit Stop&Go-Funktion erfolgen. Die kommenden beiden Unterabschnitte zeigen dabei für die Architekturarten einzeln auf, wie diese strukturiert werden sollten, um die zuvor dargelegten Ziele zu erreichen. Dies erfolgt auf eine allgemeine Art und Weise und ist daher für die Automobil-Domäne in Gänze gültig. Insbesondere für Software-Architekturen werden im weiteren Verlauf der Arbeit noch deutlich konkretere Vorgaben zur Strukturierung innerhalb des Fachbereichs Fahrerassistenz gemacht.

2.2.1 Funktions-Architektur

Der wichtigste Aspekt bezüglich der Strukturierung von Funktions-Architekturen ist, dass ausschließlich funktionale Anforderungen zu berücksichtigen sind. Zu diesen zählen neben den Anforderungen, die das Verhalten der Kernfunktionalität beschreiben, auch solche der Gebrauchs- und Funktionssicherheit, welche ebenfalls unmittelbar durch den Kunden erlebbar sind. Die Funktions-Architektur stellt die von technischen Realisierungsdetails abstrahierte Wirkkette dar, die zur Erreichung der Funktionalität notwendig ist. Somit legt die beabsichtige Funktionalität im Kontext des betrachteten Gesamtsystems (hier: Automobil) die Funktions-Architektur bereits in groben Zügen fest. Die Funktions-Architektur ist damit primär als Visualisierung einer logischen Wirkkette zu verstehen.

Zwei erfahrene Funktions-Architekten, welche ein identisches Werk an Regeln zur Strukturierung verwenden, sollten daher bei einer identischen Problemstellung zu sehr ähnlichen Architekturen kommen. Die Unterschiede können beispielsweise in der Namensgebung von Architekturbausteinen, unterschiedlicher Gruppierung von Teilfunktionen zu übergeordneten Bausteinen oder der Wahl der Systemgrenze an bestimmten Stellen liegen. Der Grundcharakter der Architektur ist jedoch gleich, der Spielraum des Funktions-Architekten ist begrenzt. Anders gesagt: Die Funktions-Architektur ergibt sich in den wesentlichen Grundzügen von selbst, da diese durch die gewünschte Funktionalität und die zur Verfügung stehende Wirkkette im Kontext des Gesamtsystems festlegt wird. Dies ist ein wesentlicher Unterschied zu den im folgenden Unterabschnitt betrachteten Kriterien zur Strukturierung von Software-Architekturen.

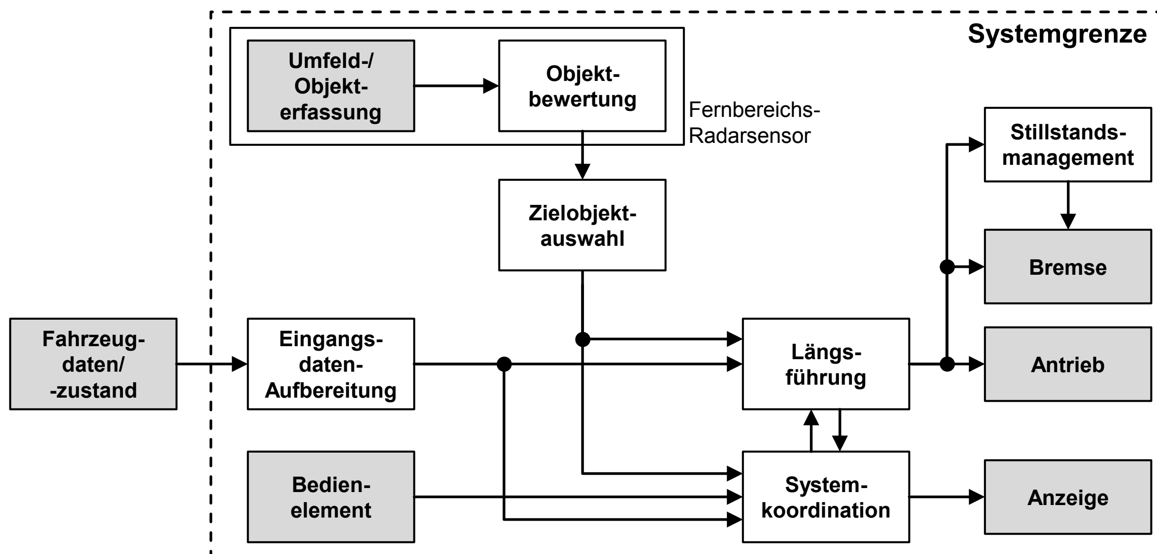

Bild 2.1: Funktions-Architektur ACC mit Stop&Go-Funktion – oberste Ebene

Die Leistung des Funktions-Architekten liegt im Durchdringen des Problems, des Erkennens und Strukturierens der Wirkkette und der sinnvollen Gruppierung der identifizierten Architekturbausteine. Der vorliegende Unterabschnitt ist als Leitfaden zu dieser Strukturierung zu verstehen. Die Arbeit legt bewusst keine konkrete Notation oder Modellierungssprache fest. Es können bestehende Ansätze verwendet werden, da die Regeln universell anwendbar sind.

Bild 2.1 zeigt eine Funktions-Architektur des ACC Stop&Go auf oberster Ebene. Auf dieser Granularität sind die folgenden Informationen ersichtlich:

- Abgrenzung des Systems von der Umwelt (Wahl der Systemgrenze),
- Identifizierung der grundsätzlich beteiligten, logischen Architekturbausteine,
- Identifizierung der grundsätzlichen Interaktionen der Bausteine und
- Zuordnung der Bausteine zu Hardware (hier: grau ausgefüllt) und Software.

Systemgrenzen werden so gewählt, dass alle Bausteine, an die explizit Anforderungen gestellt werden, innerhalb des Systems liegen. Dazu zählen unter anderem das Bedienelement oder die Aktoren. Bausteine oder Informationen, welche ohne direkte Einflussnahme auf deren Gestaltung und Ausprägung eingesetzt werden, liegen außerhalb. Hierzu zählen beispielsweise Informationen wie Fahrzeuggeschwindigkeit oder Informationen zur GPS-Position aus dem Baustein „Fahrzeugdaten/-zustand“. Die Zuordnung Hardware/Software erfolgt durch Experteneinschätzung. Bausteine, die später im Wesentlichen nur mit Hilfe physikalischer Sensoren realisiert werden können, sind klar der Hardware zuzuordnen. Beispiele sind das „Bedienelement“ sowie die „Umfeld-/Objekterfassung“. Aktoren sind zumeist ebenso klar der Hardware zuzuordnen. Bausteine, die eine Nachverarbeitung oder Interpretation dieser Informationen vornehmen, sind mit Software zu

realisieren. Ein Beispiel ist die „Objektbewertung“.

In Bild 2.1 ist noch eine Besonderheit zu sehen: Wie bereits erwähnt wurde, soll eine Funktions-Architektur von der konkreten Realisierung abstrahieren. Daher wurde die Sensorik mit dem allgemeingültigen Begriff „Umfeld-/Objekterfassung“ bezeichnet. In manchen Fällen gibt es jedoch zu Projektstart bereits Vorgaben zur Realisierung, weil diese zum Beispiel dem allgemeinen Stand der Technik oder dem Wettbewerb entsprechen. Angenommen es liegt die Verwendung eines Fernbereichsradars inkl. der Zuordnung einer Objektbewertung zu diesem Sensor als feste Vorgabe zu Beginn der Entwicklung bereits vor, so sollte dies in der Funktions-Architektur berücksichtigt werden. Eine „künstliche Abstraktion“ bringt keinen Mehrwert.

Interaktionen repräsentieren in Funktions-Architekturen im Automobil zum größten Teil Signalflüsse. Auf der obersten Betrachtungsebene werden Signalflüsse qualitativ bezüglich Vorhandensein und Richtung eingetragen. Realisierungsdetails wie Datentyp, Größe und Bitbelegung werden noch nicht angegeben. Wichtige Randbedingungen wie beispielsweise Zykluszeiten, Berechnungsreihenfolgen oder maximal zulässiges Alter von Informationen, müssen bei Bedarf modelliert werden. Neben dem Signalfluss können Interaktionen auch Kontrollfluss sowie Aufrufe von Operationen oder Diensten repräsentieren. Je nach Bedarf können die übertragenen Informationen im Detail benannt werden. Im Beispiel erfolgt dies erst in tieferen Hierarchie-Ebenen (Bild 2.2).

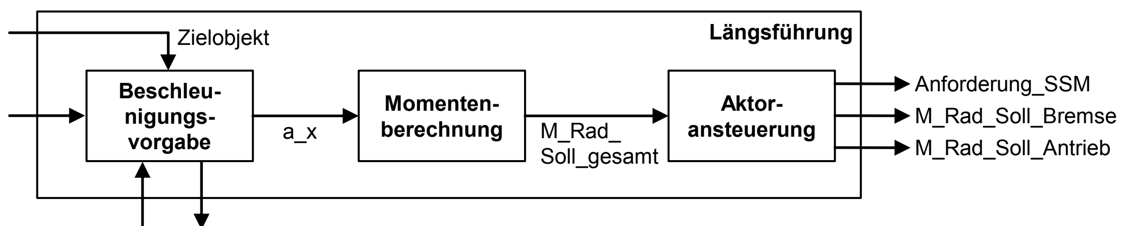

Bild 2.2: Funktions-Architektur ACC mit Stop&Go-Funktion – Längsführung

Hierarchisierung ist ein wichtiger Aspekt der Modellierung von Funktions-Architekturen. Eine beliebige (das heißt so viele Ebenen wie notwendig) und flexible (das heißt eine Hierarchisierung nur an Stellen, an denen dies vom Modellierer benötigt wird) wird vorausgesetzt. Eine weitere Empfehlung ist, dass ein definierter Typ Baustein verwendet werden sollte, der bestimmte Eigenschaften zusichert und eine Sonderstellung im Modell einnimmt. Im Rahmen der Betrachtungen dieser Arbeit ist das der so genannte Funktions-Baustein. Funktions-Bausteine haben folgende Eigenschaften:

- Sie kapseln eine gewisse innerhalb der Architektur einzigartige Funktionalität.
- Sie werden entweder vollständig durch Hardware oder durch Software realisiert.
- Sie haben definierte Schnittstellen nach außen.
- Sie sind bezüglich Umfang untereinander in etwa gleich.

Eine allgemein gültige Vorgabe, wie Funktions-Bausteine zu erhalten sind, kann nicht getroffen werden. Dies hängt stark von der Problemstellung und der Intuition beziehungsweise Erfahrung des Modellierers ab. Das Kapseln von Funktionalität zu sinnvollen Einheiten steht im Vordergrund. Die in Bild 2.1 dargestellten Architekturbausteine entsprechen Funktions-Bausteinen. Je nach Bedarf können Funktions-Bausteine detailliert werden (Bild 2.2) oder auch zu übergeordneten Einheiten (z.B. Subsystemen) gruppiert werden. Typische Sub-Funktionalitäten unterhalb von Funktions-Bausteinen sind: Signalaufbereitung, Zustandsautomat, Ablauflogik, Regler, Koordinator, Schalter. Im Rahmen dieser Arbeit werden aufgrund der Fokussierung auf die Softwareanteile insbesondere Funktions-Bausteine, welche durch Software repräsentiert sind, hierarchisch zerlegt.

2.2.2 Software-Architektur

Software-Architektur wird im Wesentlichen durch nicht-funktionale Einflussgrößen geprägt. Funktionale Anforderungen sind zwar nicht völlig irrelevant, sie sind jedoch für die Strukturierung nur von untergeordneter Bedeutung. Ganz im Gegensatz zur Funktions-Architektur gilt somit die Aussage:

> *„Würden die funktionalen Anforderungen die Architektur entscheidend beeinflussen, so müssten zwei Architekten unabhängig voneinander zur gleichen Architektur kommen. Dies ist aber nicht der Fall, da das Architekturdesign entscheidend von anderen Faktoren beeinflusst wird.“* [93]

Während die Erstellung einer Funktions-Architektur primär dem Verständnis der Projektbeteiligten dient und grundsätzlich ein optionales Arbeitsergebnis ist, liegt eine Software-Architektur spätestens mit der Implementierung der Software-Anteile immer vor. Die Software-Architektur kann sich also entweder implizit und automatisch mit fortschreitender Implementierung von selbst einstellen oder sie wird – dies ist die im Rahmen dieser Arbeit vorgeschlagene Herangehensweise – bereits im Vorfeld der Implementierung explizit entworfen.

Der Software-Architekt hat somit gegenüber dem Funktions-Architekten einen unvergleichlich größeren Spielraum bezüglich der Strukturierung. Anders gesagt: Es gibt unendlich viele Software-Architekturen, die die funktionalen Anforderungen des Systems erfüllen. Jede davon erfüllt jedoch äußerst unterschiedlich gut die nicht-funktionalen Anforderungen. Dies ist wie eingangs erwähnt die besondere Herausforderung des Software-Architekten und aufgrund der immensen Auswirkungen auf das spätere Produkt zugleich die besondere Bedeutung und Verantwortung dieser Aufgabe. Im späteren Verlauf der Arbeit wird auf die Gesamtheit der funktionalen und nicht-funktionalen Einflussgrößen auf Software-Architektur am Beispiel Fahrerassistenz noch detailliert eingegangen.

Literatur bezüglich Software-Architektur wurde bereits in Abschnitt 1.2 vorgestellt. In dieser sind Richtlinien zur Strukturierung so allgemein gehalten, dass sie zwar in jeder

Domäne anwendbar sind, gleichzeitig dem Entwickler damit nur eine grobe Hilfestellung geben. Laut Posch et al., 2007 [93] zeichnet sich gute Software-Architektur durch folgende Eigenschaften aus:

- Sie besitzt wohldefinierte Abstraktionsschichten.
- Jede dieser Schichten ist in sich abgeschlossen mit wohldefinierten Schnittstellen.
- Die Abstraktionsschichten bauen aufeinander auf.
- Zwischen Schichten beziehungsweise Bausteinen gibt es eine klare Aufgabentrennung zwischen Schnittstelle und Implementierung.
- Die Architektur ist einheitlich und einfach.

Die folgenden, ebenfalls recht allgemein gehaltenen Entwurfsprinzipien nach Reussner et al., 2009 [96] haben sich beim Entwurf von Software-Architekturen etabliert:

- **Abstraktion:** Unwichtiges weglassen, Gemeinsames zusammenfassen.
- **Modularisierung:** Logisch zusammengehörende Abstraktionen werden zu Einheiten zusammengefasst. Anwendung der Prinzipien „Teile und Herrsche“ sowie „Hohe Kohäsion, schwache Kopplung“.
- **Kapselung:** Implizite Abhängigkeiten reduzieren. Abgrenzung des Verantwortungsbereichs von Bausteinen und Schaffen von wiederverwendbarer Funktionalität. Verstecken von Informationen in Bausteinen.
- **Hierarchische Dekomposition:** Rangfolgen von Abstraktionen einführen. Gleicher Rang führt zu Abstraktions-Ebenen. Innerhalb einer Ebene sind weitere Elemente nach den Prinzipien „Teil-von“ oder „ist-ein“ enthalten.
- **Trennung von Verantwortlichkeiten:** Gliederung von Abstraktionsebenen. Für die Lösung einer Aufgabe ist genau ein Element zuständig.
- **Einheitlichkeit:** Durchgängige Anwendung von Strukturen, Schemata, Mustern, Vorgehensweisen und Entwurfsentscheidungen.

Aufgrund des allgemeinen Charakters der obigen Empfehlungen dienen sie auch im Rahmen dieser Arbeit zunächst als Einstieg. In der Praxis der Entwicklung mechatronischer Systeme in der Automobil-Domäne existieren keine etablierten und detaillierten Empfehlungen zur Strukturierung von Software-Architekturen. Dies gilt sowohl im Allgemeinen als auch auf spezifische Fachbereiche bezogen. Es existiert jedoch ein Bedarf an deutlich konkreteren Vorgaben, die Entwicklern in der Praxis des Software-Architekturentwurfs in der Automobil-Domäne und speziell bei komplexen Systemen, wie sie in der Fahrerassistenz anzutreffen sind, helfen. Dies wird im weiteren Verlauf dieser Arbeit ab Kapitel 5 erarbeitet. Im Folgenden wird noch auf weitere grundsätzliche Unterschiede in der Strukturierung von Software-Architekturen gegenüber Funktions-Architekturen eingegangen.

Gegenstand der Betrachtungen der Software-Architektur sind die über die Funktions-Architektur identifizierten Software-Anteile (Funktions-Bausteine). Dies bedeutet jedoch

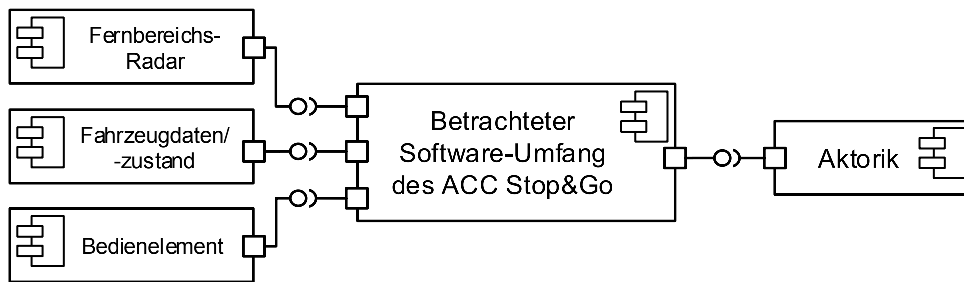

Bild 2.3: Software-Architektur ACC mit Stop&Go-Funktion – Blackbox-Sicht

nicht automatisch, dass alle Software-Bausteine gleich bedeutend sind. Anders gesagt: Die Systemgrenze kann unterschiedlich gewählt werden. Bild 2.3 verdeutlicht das am bereits bekannten Beispiel. Im (fiktiven) Geschäftsmodell wurde bereits entschieden, dass der Anteil „Objektbewertung“ durch den Lieferanten des Fernbereichsradars umzusetzen ist und auf dieser Plattform ausgeführt wird. Aus Sicht des Automobilherstellers besteht daher kein Interesse, diesen Software-Anteil im Rahmen der Software-Architektur detaillierter zu betrachten. Der Umfang wird aus der Systemgrenze entfernt, obwohl dies aus Sicht Gesamtsystem und damit der Funktions-Architektur zur unmittelbar beeinflussten Wirkkette gehört. In dieser Blackbox-Ansicht werden alle außen stehenden Bestandteile als Komponente abstrahiert, es wird nicht zwischen Software oder Hardware unterschieden, da dies aus Sicht des Software-Systems irrelevant ist. Entscheidend ist die Einhaltung der Schnittstellenverträge an den Systemgrenzen.

Als Startpunkt für den ersten Entwurf zur Detaillierung der über die Betrachtung der Systemgrenze identifizierten relevanten Anteile der Software-Architektur bietet sich die Übernahme der Funktions-Bausteine an. Es liegt nun eine Software-Architektur vor, die rein nach funktionalen Kriterien erstellt worden ist. Der weitere Entwurf ist grundsätzlich iterativ und inkrementell. So wird nach und nach die Vielzahl der nicht-funktionalen Kriterien beziehungsweise Anforderungen eingearbeitet. Dadurch wandelt sich die Architektur, Teilfunktionalitäten werden anders geschnitten und zu übergeordneten Einheiten komponiert.

Auch in der Software-Architektur muss eine beliebige Hierarchisierung ermöglicht werden und es existiert analog zum Funktions-Baustein ein Bausteintyp, der bestimmte Eigenschaften zusichert. Im Rahmen dieser Arbeit ist dies der Typ „Module“, Details zu diesem und allen anderen definierten Modellierungselementen findet sich in Kapitel 4. Wichtig ist an dieser Stelle zu verstehen, dass in Funktions- und Software-Architekturen zwar ähnliche Grundsätze sowie Struktur- und Schnittstellenelemente angewendet werden, jedoch die Struktur aufgrund der unterschiedlichen Einflussgrößen und Ziele der Architekturarten bewusst unterschiedlich ist. Dies ist der Hauptgrund, weshalb im Rahmen dieser Arbeit die explizite Modellierung beider Arten ausdrücklich empfohlen wird.

Bild 2.4 verdeutlicht diesen Sachverhalt. Während der Baustein „Eingangsdaten-Aufbereitung“ in identischer Abgrenzung und Umfang von der Funktions- auch in die

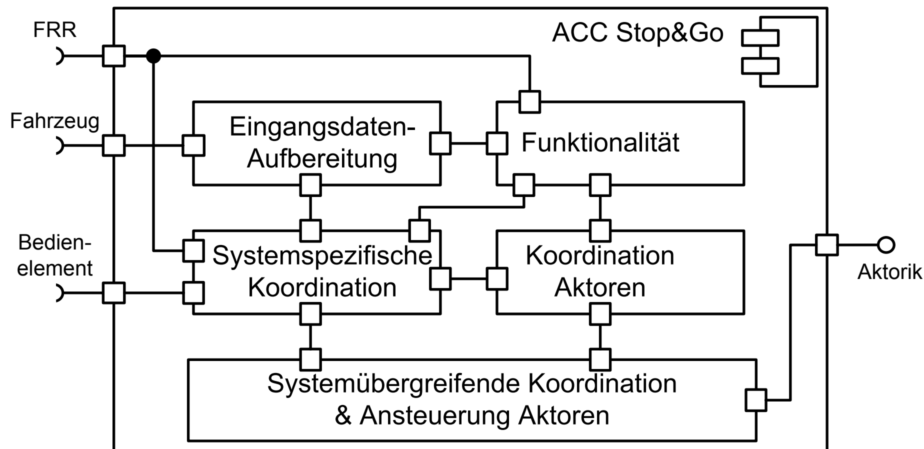

Bild 2.4: Betrachteter Software-Umfang des ACC mit Stop&Go-Funktion – Whitebox-Sicht

Software-Architektur übernommen wurde (Vergleich zu Bild 2.1), sind die übrigen Anteile der Kundenfunktion anders zugeordnet. Hauptunterschied ist das Subsystem „Systemübergreifende Koordination & Ansteuerung Aktoren“. Da typischerweise mehrere Fahrerassistenzsysteme bezüglich ihres Aktivierungszustands zu koordinieren sind und nur eines von ihnen zur selben Zeit auf gemeinsame Aktoren wie beispielsweise die Bremse zugreifen kann, werden diese Anteile notwendig. Treiber dieser Aufteilung ist somit die nicht-funktionale Randbedingung, dass nur eine einzige und von mehreren Kundenfunktionen gemeinsam genutzte Aktor-Schnittstelle zur Verfügung steht.

Eine weitere große Herausforderung des Software-Architekten speziell in der Fahrerassistenz ist demnach in großen Software-Systemen über eine Vielzahl an Kundenfunktionen den Überblick zu behalten. Identische und ausdrücklich unterschiedliche Software-Anteile sind zu identifizieren. Die Komposition der Bausteine und die definierten Schnittstellen hängen wesentlich von diesen Zusammenhängen ab.

Zusammenfassend seien an dieser Stelle bezüglich der Strukturierung und Verwendung der beiden Architekturarten die folgenden Punkte noch einmal genannt:

- Funktions- und Software-Architektur verfolgen unterschiedliche Ziele.
- Die Strukturierung erfolgt aufgrund unterschiedlicher Einflüsse.
- Beide sind innerhalb des Vorgehensmodells zur Entwicklung mechatronischer Systeme wichtig und verbindlich anzuwenden.
- Bei der Software-Architektur ist der Spielraum des Architekten insgesamt größer. Die Software-Architektur ermöglicht oder verhindert wesentlich die Erreichung nicht-funktionaler Anforderungen.
- Im weiteren Verlauf der Arbeit wird der Fokus auf Software-Architektur am Beispiel Fahrerassistenz gerichtet.

3 Vorgehensmodell zur Entwicklung mechatronischer Systeme im Automobil

In den vorherigen Kapiteln wurde auf Basis der sich aus dem Entwicklungsalltag ergebenden Probleme unter Berücksichtigung des Stands der Technik eine allgemeine Zielsetzung erarbeitet. Im Anschluss daran wurde dargelegt, dass der Schwerpunkt der Ziele sowie der zu entwickelnden Ansätze und Methoden im Themenfeld Software-Architektur anzusiedeln sind. Das vorliegende Kapitel beschäftigt sich mit Vorgehensmodellen zur Entwicklung mechatronischer, insbesondere softwaredominanter, Systeme im Automobil. Im ersten Schritt erfolgt eine Analyse des aktuell in der Serienentwicklung von Fahrerassistenzsystemen eingesetzten Vorgehensmodells, um derzeitige Schwachstellen auszumachen. So können die zuvor noch recht allgemein beschriebenen Probleme besser verstanden und konkreten Aktivitäten im Vorgehensmodell zugeordnet werden. Erst dann ist die Bearbeitung von weiterführenden Ansätzen und Methoden zur Anpassung des Vorgehensmodells sinnvoll möglich, die wirksam die Schwachstellen ausmerzen und eine Erreichung der gesetzten Ziele erlauben.

3.1 Aktuelles Vorgehensmodell in der Serienentwicklung

Der vorliegende Abschnitt ist durch die Beobachtung der Abläufe in der Serienentwicklung von Fahrerassistenzsystemen bei der BMW Group entstanden. Diese Beobachtungen und die gezogenen Schlussfolgerungen werden als typisch für die Entwicklung von Fahrerassistenzsystemen beziehungsweise von softwaredominanten Systemen in der Automobil-Domäne im Allgemeinen angesehen.

Die Entwicklung der oben genannten Systeme stellt nur einen Teilumfang der Entwicklung des Gesamtprodukts „Fahrzeug“ eines Automobilherstellers dar. Es existiert daher ein übergeordneter „Produktentstehungsprozess“, der die inhaltliche und organisatorische Koordination zwischen Fachbereichen und -abteilungen innerhalb des Unternehmens sowie zwischen Zulieferern und weiteren externen Partnern übernimmt. Zu festgelegten Zeitpunkten auf dem Weg zum Start der Serienproduktion müssen alle Teilumfänge des Fahrzeugs ihren Beitrag mit einem festgelegten Reifegrad leisten. Sowohl das Gesamtprodukt als auch die Teilumfänge werden zu diesen Meilensteinen Test und Absicherung unterzogen. Der Produktentstehungsprozess ist somit ein für alle Beteiligten verbindlicher Projekt- und Integrationsplan.

Einzelne Fachbereiche müssen gegebenenfalls weitere, darüber hinausgehende unternehmensinterne oder gesetzliche Vorgaben (Normen, Prozessanweisungen) verbindlich

berücksichtigen. Für das hier betrachtete Beispiel Fahrerassistenz gilt das im folgenden Unterabschnitt diskutierte Vorgehensmodell zur Entwicklung von eingebetteter Software. Darüber hinaus zu berücksichtigende, allgemein bekannte sowie interne Normen zur Entwicklung von Elektrik/Elektronik werden nicht im Detail vorgestellt.

3.1.1 Phasenmodell zur Entwicklung von eingebetteter Software

Das für die Software-Entwicklung eingesetzte Vorgehensmodell wird auch als „Phasenmodell“ bezeichnet und entspricht einem an die Bedürfnisse der Automobilindustrie angepassten V-Modell (Bild 3.1). Es ist speziell für die Entwicklung von eingebetteter Software vorgesehen, die in der Automobil-Domäne nahezu ausschließlich im Kontext von Software anzutreffen ist. Neben Software wird ab dem Funktions-Architekturentwurf parallel auch Hardware nach diesem Phasenmodell entwickelt. Da die Entwicklung von Hardware nicht Gegenstand der Arbeit ist, wird dies in Bild 3.1 nur zur Vollständigkeit angedeutet. Das Phasenmodell ist zu Beginn per Tailoring durch den Anwender an die spezifische Aufgabenstellung (Fachbereich, Projektumfang) anzupassen.

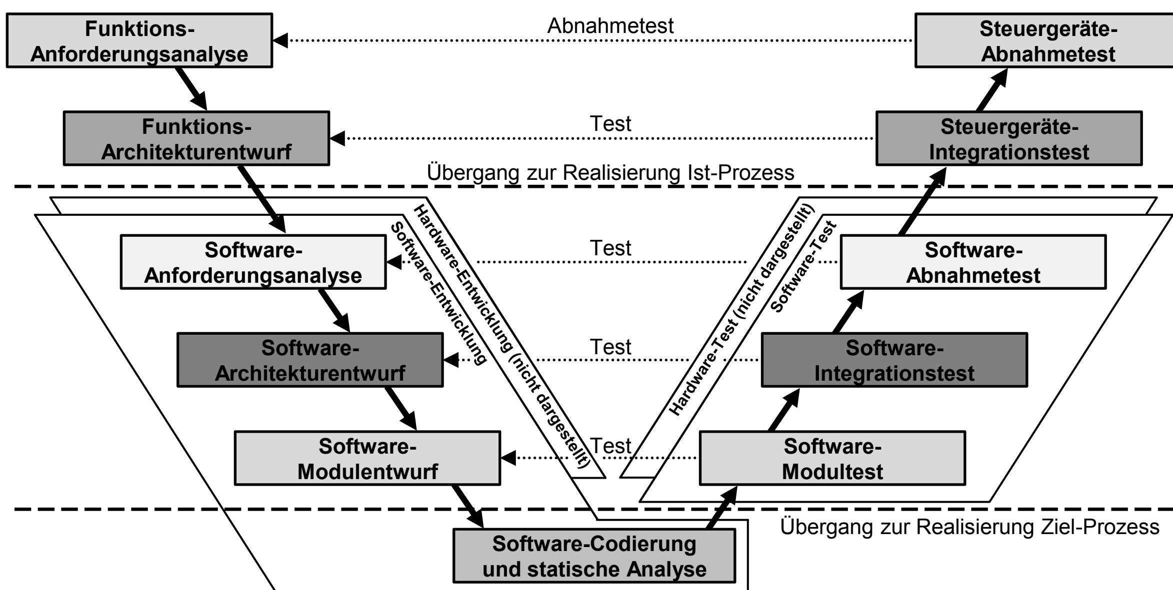

Bild 3.1: Phasenmodell zur Entwicklung von eingebetteter Software im Automobil

Das Phasenmodell unterscheidet für die Software-Entwicklung elf Phasen, die prinzipiell in der hier dargestellten Reihenfolge durchlaufen werden. Ein iteratives Vorgehen wird jedoch ausdrücklich unterstützt, wenn aufgrund von Erkenntnissen in einem tiefer gelegenen Prozessschritt eine Rückführung nach oben notwendig wird. Jeder Prozessschritt im linken Ast wird durch ein entsprechendes Pendant im rechten Ast getestet. Im Fokus der Betrachtung der vorliegenden Arbeit sollen Aspekte der Entwicklung und somit des linken Astes liegen. Daher wird im Folgenden nur auf diese Prozessschritte detailliert

eingegangen und in späteren Abbildungen lediglich der linke Ast dargestellt. Für die erarbeiteten Anpassungen und Erweiterungen im Vorgehensmodell werden keine expliziten Gegenstücke zur Absicherung entwickelt, das obliegt eventuellen Folgearbeiten. Da die bisherigen Prozessschritte jedoch erhalten bleiben, bleiben die vorhandenen Prozessschritte zu deren Absicherung ebenfalls bestehen, so dass kein Qualitätsverlust aufgrund geringerer Testtiefe zu erwarten ist. Dies ist ein weiterer Vorteil von punktuellen Verbesserungen eines bestehenden Vorgehensmodells gegenüber einer Neuentwicklung.

Funktions-Anforderungsanalyse: Hier werden die funktionalen und nicht-funktionalen Anforderungen auf lösungsunabhängige Art und Weise gesammelt. Das Ergebnis ist eine Spezifikation.

Funktions-Architekturentwurf: Auf Basis der zuvor erstellten Spezifikation erfolgt eine Zerlegung und Gruppierung der Gesamtfunktion in Funktions-Bausteine aufgrund logischer beziehungsweise physikalischer Zusammenhänge. Systemgrenzen sowie beteiligte Hard- und Softwarebestandteile werden identifiziert. Anschließend erfolgt eine Verteilung (Partitionierung) der aus Software bestehenden Systembestandteile auf reale Rechenplattformen (Steuergeräte). Je nach Problemstellung kann eine Detaillierung des Inneren von Funktions-Bausteinen (Hierarchie-Ebenen mit unterschiedlichen Architekturbausteinen) erfolgen.

Software-Anforderungsanalyse: Für die identifizierten Software-Bestandteile erfolgt eine Analyse der spezifischen Anforderungen. Betrachtet werden unter anderem Anforderungen an Rechenleistung, funktionale Sicherheit, Datensicherheit und Softwaremodifikation. Die Ziel-System-Architektur muss zu diesem Zeitpunkt als Eingangsdokument bereits vorliegen.

Software-Architekturentwurf: Die Phase des Software-Architekturentwurfs beschreibt die eindeutige Zuordnung der Software-Anforderungen zu einzelnen Software-Komponenten. Dabei gilt es eine innere Architektur der zuvor identifizierten – und bereits auf Plattformen verteilte – Bausteine zu entwerfen. Alle externen und internen Schnittstellen der Bausteine sind zu beschreiben.

Software-Modulentwurf: Aus den zuvor erstellten Anforderungen an die Software-Bausteine wird eine Spezifikation zu deren Implementierung abgeleitet. Hierzu müssen das Verhalten sowie Schnittstellen und Datenstrukturen auf Implementierungsniveau beschrieben werden.

Software-Codierung und statische Analyse: Es erfolgt eine Umsetzung des Software-Modulentwurfs in Quelltext. Dies kann durch händischen Code oder durch Einsatz von CASE-Werkzeugen mit automatischer Codegenerierung erfolgen. Zu beachten sind jeweils Richtlinien zur Codierung beziehungsweise Modellierung, deren Einhaltung im Rahmen der statischen Analyse geprüft wird. Im Rahmen dieser erfolgt ebenfalls die Überprüfung des Daten- und Kontrollflusses.

3.1.2 Analyse der Schwachstellen

Im Folgenden werden die Schwachstellen in Bezug auf die bekannten Probleme und zu erreichenden Ziele systematisch analysiert. Dabei werden sowohl die Vorgaben des Prozesses als solchem als auch die tatsächliche praktische Anwendung (so genannter „gelebter Prozess“) in der Entwicklung einbezogen.

Zu früher Übergang auf technische Realisierung

Im derzeitigen Phasenmodell erfolgt der Übergang von einer lösungs- beziehungsweise technologieunabhängigen zu einer von einer konkreten Realisierung abhängigen Darstellung zu früh. Unter einer konkreten Realisierung ist hier die Software für ein bestimmtes Fahrzeug mit einer vorgegebenen System-Architektur, die unter anderem Steuergeräte (Plattformen) und Bussysteme festlegt, zu verstehen.

Per Prozessdefinition erfolgt im Anschluss an den Funktions-Architekturentwurf bereits eine Verteilung der gefundenen funktionalen Bausteine auf konkrete Plattformen. Das ist gleichbedeutend mit einem unumkehrbaren Übergang zur technischen Realisierung. Damit sind mehrere Nachteile verbunden. Die System-Architektur muss vor Beginn der Software-Entwicklung bereits vorliegen. Das verhindert eine parallele Entwicklung und verlängert in der Folge die Entwicklungsdauer. Zudem sind Software-Architektur und -Entwicklung abhängig von der eingesetzten Plattform beziehungsweise Hardware. Alle nachfolgenden Schritte sind für jede einzelne, konkrete Realisierung immer wieder zu durchlaufen. Der Aufwand ist somit proportional zur Anzahl der Realisierungen. Die Vielzahl an Derivaten und Varianten ist bekanntlich ein zunehmendes Problem in der Automobilindustrie. Wiederverwendung ist aus diesem Grunde kaum gegeben.

Darüber hinaus ist bekannt, dass die fachliche Logik (komplett abstrahiert von jeglichen Realisierungsdetails) von Systemen weitaus stabiler ist und sich weniger häufig wandelt als Technologien und Plattformen zu ihrer technischen Umsetzung. Im betrachteten Vorgehen liegt eine Vermischung dieser beiden vor, so dass von diesem Umstand nicht profitiert werden kann. Der frühe Übergang auf die konkrete technische Realisierung ist in Bild 3.1 durch die obere gestrichelte Linie dargestellt.

Unzureichende Berücksichtigung der nicht-funktionalen Einflussgrößen

Die nicht-funktionalen Einflussgrößen, insbesondere die nicht-funktionalen Anforderungen, werden nicht in ausreichendem Maße berücksichtigt. Das Vorgehensmodell ermöglicht dies grundsätzlich in den Schritten „Funktions-Anforderungsanalyse“ sowie „Software-Anforderungsanalyse“ durchzuführen. In der gelebten Praxis werden diese jedoch weder systematisch betrachtet noch verbindlich spezifiziert. Dies führt zwangsläufig dazu, dass im späteren Verlauf der Entwicklung keine expliziten Entwurfsentscheidungen durchgeführt werden, die positiv in dieser Richtung wirken. Somit werden Eigenschaften wie Skalierbarkeit und Änderbarkeit, die durchaus implizit erwartet werden, sehr wahrscheinlich nicht erreicht.

Keine explizite Modellierung der Software-Architektur

Die frühe Zuordnung der funktionalen Bausteine zu konkreten Plattformen hat einen weiteren Nachteil. Damit wird eine explizite und gesamthafte Gestaltung, Modellierung und Optimierung einer Software-Architektur verhindert. Zwar sieht das Vorgehensmodell den Schritt des „Software-Architekturentwurfs“ vor, der Architekt ist aufgrund der Vorarbeit aber bereits zu stark eingeschränkt.

Die Software-Architektur ist somit im Groben bereits fest vorgegeben. Die grundsätzliche Komposition von Bausteinen muss der Struktur der Funktions-Architektur folgen. In Kapitel 2 wurde diskutiert, dass eine eigenständige Strukturierung der Software-Architektur jedoch zwingend notwendig ist. Der Software-Architekt hat nunmehr lediglich innerhalb der designierten Bausteine auf den Plattformen einen Spielraum zur Gestaltung der Architektur im Kleinen. So ist weder eine Veranschaulichung des Software-Systems unter den Stakeholdern zur Schaffung von Verständnis und Dokumentation von Entwurfsentscheidungen noch eine umfassende Betrachtung nicht-funktionaler Einflussgrößen möglich. Die Tatsache, dass die Letztgenannten derzeit nicht hinreichend betrachtet werden, ist primär ein Problem der gelebten Praxis. Bedeutender ist das zuvor genannte, systematisch im Vorgehensmodell selbst begründete Problem, dass diese auch bei Vorliegen nicht umfänglich im Entwurf berücksichtigt werden können.

Diese Einschränkungen führen aktuell dazu, dass auch im Kleinen der Spielraum für einen Software-Architekturentwurf nicht ausgenutzt wird. Die Software-Architektur wird gar nicht explizit modelliert. Sie ergibt sich vollständig implizit über die Modellierung der Realisierung im CASE-Werkzeug ASCET. Dies erschwert die Kommunikation der Software-Architektur noch zusätzlich, da die Visualisierung der Bausteine und Abhängigkeiten dort nicht sehr übersichtlich ist. Des Weiteren ist die Software-Architektur so über das proprietäre Datenformat fest an dieses Experten-Werkzeug, das nicht allen Projekt-Mitarbeitern zur Verfügung steht, gekoppelt.

Der Vollständigkeit halber sei noch erwähnt, dass aufgrund der oben beschriebenen Einschränkungen in der Behandlung der Software-Architektur eine objektive Bewertung dieser ebenfalls nicht durchgeführt werden kann und damit ein zielgerichtetes, iterativ inkrementelles Vorgehen zum Entwurf nicht ermöglicht wird.

Weitere Aspekte

Innerhalb des Vorgehensmodells ist eine Modellierung von Varianten-Informationen sowie ein Varianten-Management auf fachlicher Ebene (zum Beispiel für funktionale Ausprägungen) nicht vorgesehen. Dies ist aufgrund des erwähnten frühen Übergangs auf die konkrete technische Realisierung auch nicht zwingend notwendig.

Außerdem sind keine automatisierten Übergänge zwischen Prozessschritten realisiert. Der Ablauf ist somit aufgrund der Vielzahl an manuellen Tätigkeiten sehr aufwändig und potenziell fehleranfällig. Lediglich innerhalb des CASE-Werkzeugs ASCET findet sich ein formaler Automatismus im Rahmen der Code-Generierung.

3.2 Konkrete Ziele zur Anpassung des Vorgehensmodells

Nachdem die Schwachstellen im Detail bekannt sind, können die konkreten Zielsetzungen zur Anpassung des Vorgehensmodells formuliert werden. Dies erfolgt mit Blick auf die bereits in Kapitel 1.3 formulierten globalen Ziele der Arbeit. Aufgrund der Vorarbeit sind diese nun konkretisierbar.

Die Auflistung erfolgt an dieser Stelle nur stichpunktartig. Alle Ansätze werden im weiteren Verlauf der Arbeit jeweils noch im Detail vorgestellt.

Berücksichtigung nicht-funktionaler Einflüsse und Anforderungen

- Nicht-funktionale Einflussgrößen und Anforderungen sind explizit zu ermitteln und verbindlich zu berücksichtigen.
- Dazu sind die vorhandenen Prozessschritte Funktions-Anforderungsanalyse (allgemeine) und Software-Anforderungsanalyse (software-spezifische) zu verwenden.
- Die Anforderungserfassung im Detail (zum Beispiel methodisches Vorgehen und Werkzeug) wird an dieser Stelle nicht näher vorgeschrieben.
- Die Gesamtheit funktionaler und nicht-funktionaler Einflussgrößen zeigt Bild 3.2.

Explizite Modellierung einer gesamthaften Software-Architektur

- Software-Architektur ist explizit und für alle Software-Anteile übergreifend zu modellieren. Dazu ist der Prozessschritt Software-Architekturentwurf zu verwenden.
- Die Strukturierungsvorgaben und Beschreibung der Abgrenzung zur Funktions-Architektur gemäß Kapitel 2 sind zu berücksichtigen.

Abstraktion und möglichst lange Trennung fachliche Logik und Realisierung

- Konsequente Abstraktion von Details der technischen Realisierung sowie
- Konsequente und möglichst lange Trennung der fachlichen Logik von der technischen Realisierung. Erst unmittelbar vor der Software-Codierung als spätest möglicher Zeitpunkt darf dieser Übergang erfolgen (Bild 3.1, untere gestrichelten Linie).
- Die Verteilung von Bausteinen auf Plattformen darf nicht auf Basis der Funktions-Architektur sondern erst im Anschluss an die Software-Architektur erfolgen.
- Die Software-Architektur muss vollständig abstrakt gehalten sein sowie ein Varianten-Management ermöglichen.

Objektives Bewertungsverfahren für Software-Architekturen

- Ein Entwurf der Abstrakten Software-Architektur muss objektiv bewertbar sein.
- Dazu ist ein automatisiertes Verfahren mit objektiven, das heißt quantifizierten, Metriken anzuwenden.
- Der damit unterstützte Software-Architekturentwurf erfolgt iterativ inkrementell.

Automatisierte Übergänge durch Formalismen und Standards

- Durch Einsatz von Formalismen sollen die folgenden Schritte automatisiert durchgeführt werden:
  - Konfiguration von Varianten aus der Abstrakten Software-Architektur,
  - Konfiguration von plattformspezifischen Architekturen aus der Abstrakten Software-Architektur und
  - Übergabe der Software-Architektur an die bisher verwendeten CASE-Werkzeuge zur Software-Umsetzung.
- Einsatz von offenen Standards sowie OpenSource-Werkzeugen und -datenformaten.

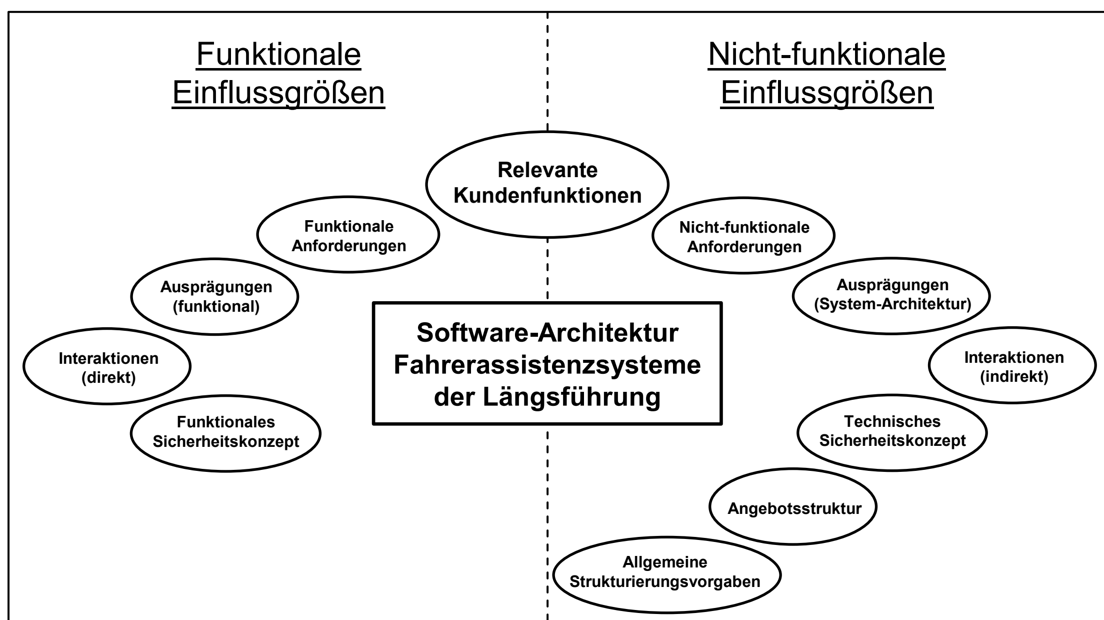

Bild 3.2: Einflussgrößen auf Software-Architektur am Beispiel Fahrerassistenzsysteme

3.3 An die Zielsetzung angepasstes Vorgehensmodell

In diesem Abschnitt wird beschrieben, wo und wie die zuvor diskutierten Anpassungen und Erweiterungen im Vorgehensmodell integriert werden sollen. Als Ergebnis wird das neue Vorgehensmodell in Gänze beschrieben, wie es für die Entwicklung softwaredominanter Systeme im Automobil – speziell für Fahrerassistenzsysteme – vorgeschlagen und in dieser Arbeit weiter verfolgt wird. Fokus der Ausführungen sind die vorgenommenen Änderungen. Dabei wird deutlich gemacht, dass die Basis dieses neuen Vorgehensmodells weiterhin das in Abschnitt 3.1.1 dargestellte Phasenmodell ist. Die Zielsetzung der punktuellen Verbesserung unter Beibehaltung der Basis wurde demnach erfüllt.

3.3.1 Prozessrahmen und Sichten auf das Gesamtsystem Fahrzeug

Bevor das Vorgehensmodell im Detail diskutiert wird, sollen noch die folgenden Überlegungen vorgestellt werden. Dabei geht es um die Fragestellung, welche Betrachtungsebenen – oder abstrakt gefasste Architekturen – auf das Gesamtsystem Fahrzeug grundsätzlich möglich sind. Das Ergebnis der Überlegungen kann als „Prozessrahmen" bezeichnet werden. Es werden nicht wie in einem Vorgehensmodell beziehungsweise Prozess konkrete Aktivitäten, Abfolgen dieser und definierte Arbeitsergebnisse festgelegt, vielmehr wird die Fahrzeug-Entwicklung systematisch eingeteilt. Dies soll das Verständnis der Beteiligten erhöhen, indem Zusammenhänge klarer werden.

Dabei ist vor allen Dingen interessant, dass dies kein Rahmen ist, der für eine Entwicklung zunächst vorgegeben und explizit angewendet werden muss. Viel mehr ist dies eine abstrahierte, strukturierte Sicht auf die Entwicklung beziehungsweise das Fahrzeug selbst. Arbeitsergebnisse aus realen Vorgehensmodellen können den identifizierten Ebenen des Prozessrahmens zugeordnet werden, unabhängig davon, ob den Entwicklern dieser bereits bekannt ist oder nicht. Wenn sich jedoch Projektbeteiligte die Ebenen und Zielsetzungen der Ebenen bewusst gemacht haben, so kann gezielt auf die Entwicklung im Detail Einfluss genommen werden. Dies kann beispielsweise bedeuten, im Vorgehensmodell Arbeitsergebnisse einer bestimmten Art und Güte festzulegen.

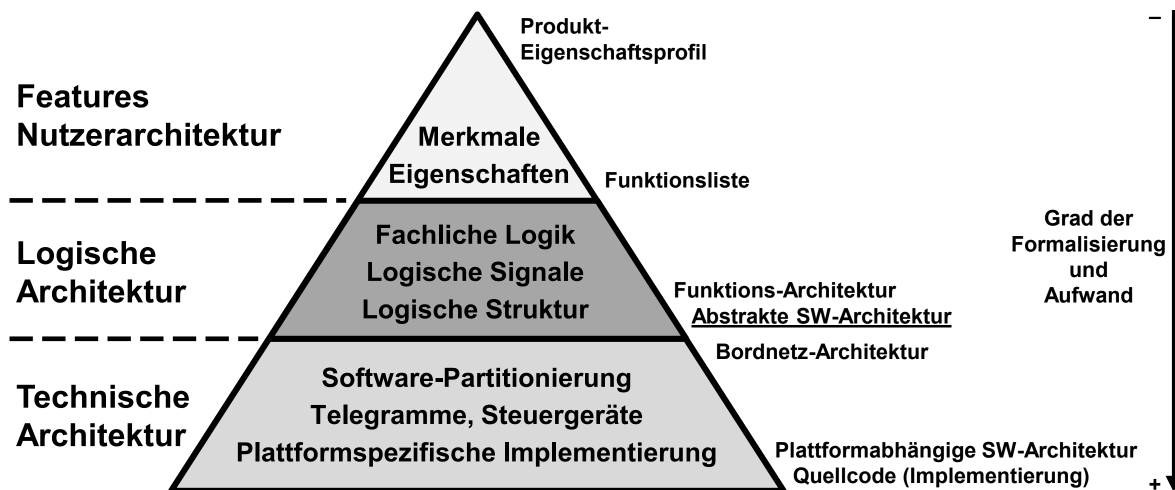

Bild 3.3: Betrachtungsebenen auf das Fahrzeug in der (Software-)Entwicklung

Bild 3.3 zeigt diesen Prozessrahmen. Links der Pyramide sind die drei Bezeichnungen der eingeführten Ebenen ersichtlich, rechts daneben befinden sich die wesentlichen Arbeitsergebnisse, welche typischerweise in der jeweiligen Ebene produziert (z.B. „Funktionsliste" für Ebene 1) oder als Eingangsdokument (z.B. „Produkt-Eigenschaftsprofil" für Ebene 1) benötigt werden. Im Inneren stehen die Betrachtungsgegenstände und Arten von Informationen, die auf der jeweiligen Ebene verarbeitet werden. Zum besseren Verständnis werden alle drei Ebenen im Folgenden detailliert vorgestellt.

Nutzerarchitektur

In der Nutzerarchitektur werden die aus Kundensicht wesentlichen und erlebbaren Merkmale und Eigenschaften des Fahrzeugs verwaltet und strukturiert. Als Eingangsdokument dient das Produkt-Eigenschaftsprofil, welches alle die oben genannten Eigenschaften auf unformale oder semiformale Art und Weise bereitstellt. Die unterschiedlichsten Anforderungen, die an ein Fahrzeug gestellt werden, liegen gleichberechtigt nebeneinander vor. Besonders wichtig ist die völlig abstrahierte, das heißt lösungs- und umsetzungsneutrale, Formulierung der Eigenschaften genau so, wie sie sich dem Kunden darstellen sollen. Auf dieser Ebene geschieht im Anschluss die Zuordnung dieser Merkmale zu den Kundenfunktionen. Das Ergebnis ist die Funktionsliste. Als Beispiel für die Identifikation einer Kundenfunktion sei hier das Folgende betrachtet. Aus einem oder mehreren Merkmalen wie beispielsweise „Das Fahrzeug soll eine vom Fahrer vorgegebene Wunschgeschwindigkeit selbstständig halten können“ kann eine Kundenfunktion „Aktive Geschwindigkeitsregelung“ abgeleitet werden. Üblicherweise wird im Rahmen der Generierung der Funktionsliste auch die Zuordnung der einzelnen Kundenfunktionen zu den Hauptbereichen der Fahrzeugentwicklung vorgenommen: Karosserie, Innenraum, Fahrdynamik, Antrieb (KIFA). Die Nutzerarchitektur dient somit auch organisatorisch der Klärung von Verantwortlichkeiten und Durchführung von Zielvereinbarungen.

Logische Architektur

In der Logischen Architektur erfolgt die Zuordnung und Verfeinerung der Kundenfunktionen zu eigenständigen funktionalen Ausprägungen. Bei Fortführung des oben genannten Beispiels ist eine solche konkrete Ausprägung die Funktion „ACC mit Stop&Go-Funktion“. Zur Verfeinerung gehört ebenfalls die Spezifikation der funktionalen und nicht-funktionalen Anforderungen. Diese ist auf dieser Ebene jedoch weiterhin noch völlig umsetzungsneutral. In der Praxis ist diese Trennung nicht immer vollständig durchführbar, da teilweise unterschiedliche funktionale Ausprägungen von Funktionen beispielsweise unmittelbar bestimmten Sensorbauformen (z.B. Radar, Kamera) zugeordnet werden, eine vollständige Abstraktion ist dann nicht immer zweckmäßig.

Für die konkreten Ausprägungen erfolgt des Weiteren eine logisch physikalische Strukturierung mit dem bekannten Ergebnis der Funktions-Architektur. Als weiteres Arbeitsergebnis der logischen Architektur sei insbesondere noch die Abstrakte Software-Architektur erwähnt. Diese erfasst und strukturiert die in der Funktions-Architektur identifizierten Softwareanteile des Systems gesamthaft auf umsetzungsneutrale Art und Weise. Die Abstrakte Software-Architektur, im weiteren Verlauf der Arbeit auch als Abstrakte Automotive Software-Architektur (ABSOFA) bezeichnet, stellt eines der wesentlichen Konzepte dieser Arbeit dar. Auf sie wird in Kapitel 4 ausführlich eingegangen.

Die logische Architektur ist das Bindeglied und damit die Vorstufe der technischen Realisierung. In der heutigen Entwicklungspraxis kommen nach wie vor zu großem Anteil informale Arbeitsergebnisse zum Einsatz, beispielhaft seien Darstellungen von

Funktions-Architekturen in rein visualisierenden Werkzeugen wie Powerpoint genannt. Semi-formale und formale Ansätze sind eine Möglichkeit die Ebene zu stärken und Vorgehensmodelle effizienter zu gestalten.

Technische Architektur

Die Technische Architektur bildet die zuvor lösungsunabhängigen Konzepte auf eine konkrete Realisierung ab. Darunter ist ein konkretes Fahrzeug (Derivat, Variante) einer Baureihe mit den dort gültigen, technischen Randbedingungen wie Bordnetz (Busse, Steuergeräte), Sensor- und Aktorbauformen zu verstehen. Software-Anteile werden auf die Steuergeräte verteilt (partitioniert), aus logischen Signalen werden technische Signale, die je nach Partitionierung zu Steuergeräte-internen oder -externen Informationen (Telegramme am Bus) werden.

In der Technischen Architektur sind Arbeitsergebnisse in der Regel formal. Spätestens das Endergebnis der Software-Implementierung muss formal und eindeutig sein, da diese als Binärdatei auf einer Hardware ausführbar ist. Auch bei anderen Dokumenten wie Beschreibungen von Bordnetz-Architekturen werden mehr und mehr formale und maschinenlesbare Beschreibungen eingesetzt, beispielsweise in standardisiertem XML-Format. Dies wird insbesondere durch die AUTOSAR-Initiative gefördert. In der Pyramide kann also ein Ansteigen des Grads der Formalisierung der Arbeitsergebnisse von oben nach unten attestiert werden. Außerdem spiegelt die Pyramide aufgrund ihrer Form auch die Verteilung der Aufwände zu den Ebenen näherungsweise wider.

Fazit

Die Darstellung und die darin beschriebenen Ebenen sind für die Entwicklung aller Arten von Umfängen des Fahrzeugs verwendbar. Je nach Fachbereich können relevante Arbeitsergebnisse unterschiedlich ausgeprägt sein, nicht benötigt oder zusätzlich benötigt werden. Die angetragenen Informationen fokussieren sich auf die Entwicklung softwaredominanter Systeme. Bei der Entwicklung dieser Systeme ist die Automobilindustrie insbesondere auf der obersten und der untersten Ebene sehr gut aufgestellt. Die Kundenbedürfnisse und Ziele an das Fahrzeug auf sehr abstrakter Ebene sind auf der einen Seite sehr gut verstanden und werden für die Entwicklung als verbindlich vorgegeben. Auf der anderen Seite erfolgt auch die Umsetzung erfolgreich entsprechend dieser Vorgaben. Das bedeutet, die in den Markt gebrachten Systeme entsprechen der Erwartung der Kunden. Mit dem letztgenannten Aspekt ist dieses Vorgehen vermeintlich bereits optimal. Das Hauptproblem ist jedoch die fehlende Effizienz. Aufgrund der angesprochenen Variantenvielfalt ist es wegen des Aufwandes mit den zur Verfügung stehenden Ressourcen in der geforderten Zeit nicht mehr möglich, alle notwendigen Schritte bis zur finalen Umsetzung immer wieder zu durchlaufen.

Aus Sicht dieser Arbeit ist der Schlüssel zur Lösung des Problems in der mittleren Ebene zu finden. Mit der konsequenten Trennung von Logik und Realisierung soll diese

Ebene gestärkt werden, es wird so genanntes „Frontloading“ betrieben. Das heißt es wird verstärkter Aufwand an dieser Stelle investiert, dafür werden spezifische Aufwände für die Realisierung der Varianten (n-fach) reduziert, so dass der Gesamtaufwand sinkt. Automatische Übergänge innerhalb und zwischen den Ebenen reduzieren den Aufwand zusätzlich und sorgen aufgrund der guten Anwendbarkeit für eine höhere Akzeptanz bei den Anwendern. Die zuvor beschriebenen Detail-Ziele der Arbeit und der Anpassung des Vorgehensmodell fußen auf dieser Aussage.

3.3.2 Das angepasste Vorgehensmodell im Detail

Zum Abschluss wird das angepasste Vorgehensmodell im Detail vorgestellt. Bild 3.4 zeigt den gesamten Ablauf mit allen Prozessschritten. Die wesentlichen über diese Arbeit geänderten oder erweiterten Aspekte sind grau markiert. Zur inhaltlichen Einordnung ist jedem Prozessschritt die Ebene des Prozessrahmens (der geschichteten Pyramide) aus Unterabschnitt 3.3.1 zugeordnet. Die folgenden Informationen dienen als zusätzliche Erläuterung zur Abbildung.

- **Nutzerarchitektur:** Die oberste Ebene ist in der Arbeit nicht im Fokus. Daher sind hier die Schritte von abstrakten Produktmerkmalen bis zu einer definierten Kundenfunktion vereinfacht in einem Prozessschritt dargestellt.
- **Funktions-Anforderungsanalyse:** Sie dient der systematischen Ermittlung aller Anforderungen, die sich aus funktionalen und nicht-funktionalen Einflussgrößen (Bild 3.2) ableiten. Ansätze mit Erfassung und Verlinkung von Anforderungen in Modellen wie bei SysML werden empfohlen aber nicht vorgeschrieben. Da textuelle Anforderungen jedoch nie formal und eindeutig sein können, ist im Prozessschritt und im Übergang aus diesem heraus immer ein manueller Anteil notwendig.
- **Funktions-Architekturentwurf:** In diesem Prozessschritt werden im Rahmen dieser Arbeit ebenfalls keine expliziten Vorgaben gemacht. Empfohlen werden formale Ansätze, die auf UML oder SysML basieren. So kann ein weiteres modellgetriebenes Verarbeiten bis zur Software-Architektur vorbereitet werden. Dies ermöglicht beispielsweise einen automatisierten Transfer der Funktions-Architektur in die Abstrakte Software-Architektur als ersten Entwurf.
- **Software-Architekturentwurf:** Der alte Prozessschritt ist auf zwei aufgeteilt. Der größte Modellierungsaufwand wird in die Abstrakte Software-Architektur gesteckt. Diese wird in einem iterativ inkrementellen Vorgehen unterstützt durch das objektive Bewertungsverfahren erstellt. Wesentlich ist hier des Weiteren ein zentrales Datenmodell. In diesem wird die Software-Architektur verwaltet sowie Metriken und Modelltransformationen darauf angewendet. Zum Übergang in die konkrete Realisierung kann nach Vorgabe der Variante und Plattform die Plattformabhängige Software-Architektur automatisiert erzeugt werden. Die „Software-Anforderungsanalyse“ taucht bewusst nicht mehr separat auf sondern ist wesentlicher Bestandteil des Entwurfs der Abstrakten Software-Architektur.

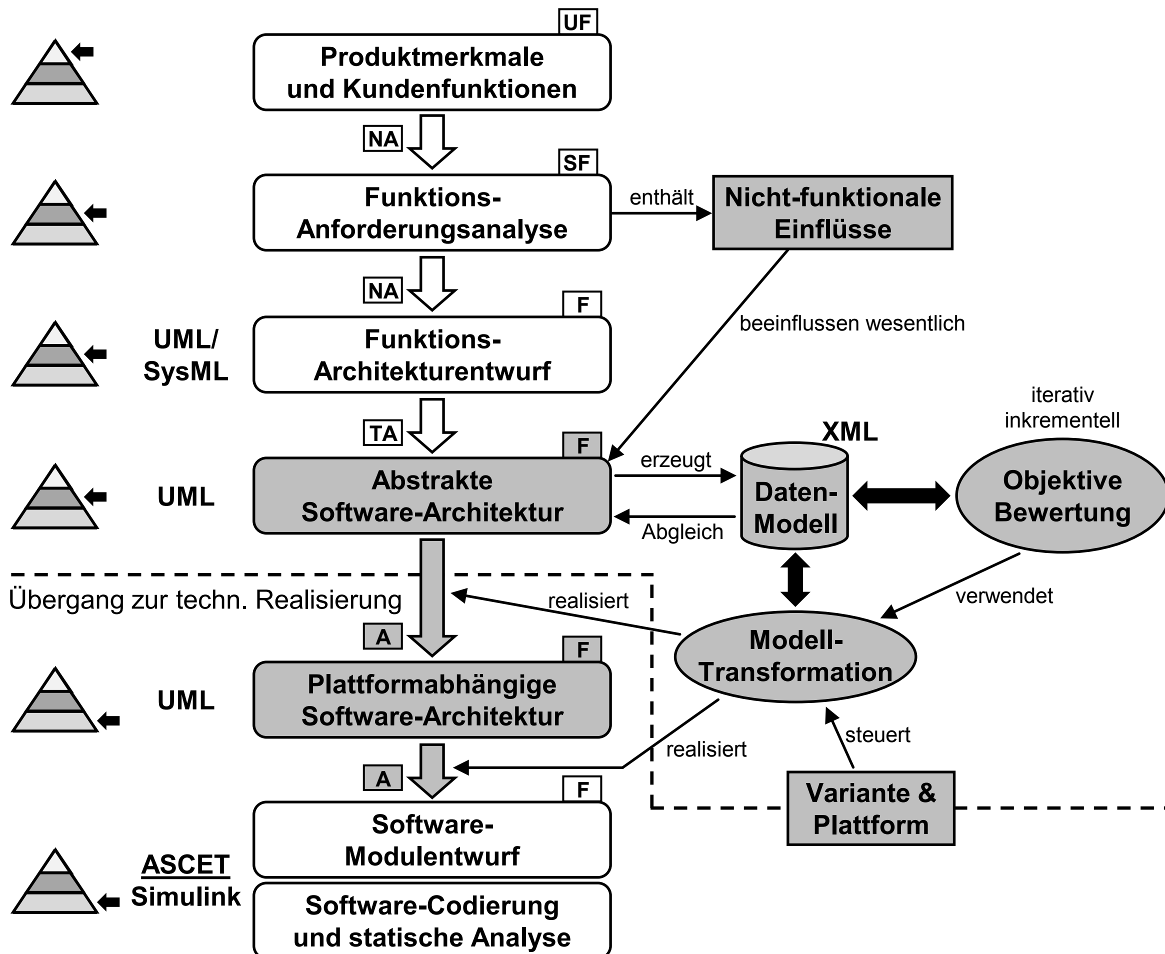

Bild 3.4: Angepasstes Vorgehensmodell zur Entwicklung softwaredominanter Systeme

4 Einführung der Abstrakten Automotive Software-Architektur – ABSOFA

Wie erörtert, soll eine abstrakte Software-Architektur im Rahmen des erweiterten Vorgehensmodells eine zentrale Rolle einnehmen. Im vorliegenden Kapitel wird diese sehr detailliert bezüglich der genauen Zielsetzung und erarbeiteten Umsetzung vorgestellt. Der Ansatz wird konkret ab sofort als „Abstrakte Automotive Software-Architektur“ (ABSOFA) bezeichnet. Darüber hinaus werden ebenfalls dessen Integration in das Vorgehensmodell und das Zusammenspiel mit den übrigen vorgeschlagenen Neuerungen im Detail erklärt.

4.1 Grundlegende Ziele und Eigenschaften des Ansatzes

Zunächst soll noch einmal auf die genaue Zielsetzung und damit die angestrebten Eigenschaften des Modellierungsansatzes eingegangen werden. Die vier Hauptziele – dies ist eine Wiederholung aus den vorangegangenen beiden Kapiteln – sind die Folgenden:

- Möglichst lange Trennung von fachlicher Logik und technischer Realisierung,
- Explizite und eigenständige Modellierung der Software-Architektur,
- Vollständige Abstraktion von Realisierungsdetails der Software-Architektur sowie Modellierung aller relevanter Software-Anteile gesamthaft (in einem Modell) und
- Formalismus und automatisierte Übergänge für eine durchgängige Entwicklung.

Darauf aufbauend wurde mit dem in Bild 4.1 dargestellten Vorgehen, das im Anschluss erläutert wird, der Ansatz der ABSOFA im Detail entwickelt.

Anforderungen erfassen

Der erste Schritt ist dabei die formale und möglichst vollständige Erfassung der Anforderungen an den Ansatz. Um dies zu gewährleisten, wurden in insgesamt fünf für die Ausgestaltung der Lösung wichtigen Kategorien Anforderungen gesammelt. Hinter den Kategorien verbergen sich im Einzelnen die folgenden Aspekte:

- **Zielsetzung und Umfang:** Hierzu gehören alle Anforderungen, die sich nicht in die danach folgenden Kategorien einsortieren lassen. Es wird insbesondere der Einsatzbereich (Software-Architektur in der Automobil-Domäne), der Betrachtungsumfang (statische Software-Struktur), das Zusammenspiel mit anderen Prozessschritten (zum Beispiel Bewertungsverfahren und Datenmodell) und Werkzeugen

der Entwicklung (zum Beispiel ASCET) sowie grundsätzliche Eigenschaften (zum Beispiel konsequente Abstraktion, Modellierung von Varianten) festgelegt.

- **Darstellung:** In diesem Teilumfang wird festgelegt, welche Vorgaben für die Darstellung existieren. Dazu zählen Aspekte bezüglich Hierarchisierung, Übersichtlichkeit sowie welche Standpunkte und Sichten mit welchen Informationen für welche Stakeholder existieren müssen.
- **Modellierungselemente:** Hierbei werden Anforderungen an das Vorhandensein, die Ausprägung und die Eigenschaften von Elementen wie Strukturelementen, Schnittstellenelementen und Kommunikationsbeziehungen erfasst.
- **Einschränkungen/Randbedingungen zur Verwendung der Elemente:** Es werden Vorgaben gemacht, welche Elemente in welchem Kontext (zum Beispiel in welcher Sicht) verwendet werden dürfen und mit welchen anderen Elementen in welchem Kontext wie interagiert werden darf.
- **Notation und Werkzeug:** Welche Anforderungen werden an die Notation beziehungsweise Modellierungssprache und auch das Werkzeug als solches gestellt? Hierzu gehören Anforderungen wie Verwendung offener und kostenfreier Standards, Kompatibilität mit anderen Werkzeugen oder Austauschformaten und der Verwaltung des Modells im Werkzeug (zum Beispiel Visualisierung von Modellelementen in unterschiedlichen Diagrammen und Kontexten bei einmaliger Ablage aller relevanter Informationen im Modell).

Die wichtigsten der so ermittelten Anforderungen finden sich in Tabelle A.2 im Anhang A.2, Einschränkungen zur Verwendung der Elemente sind in den Tabellen B.1-B.3 in Anhang B enthalten.

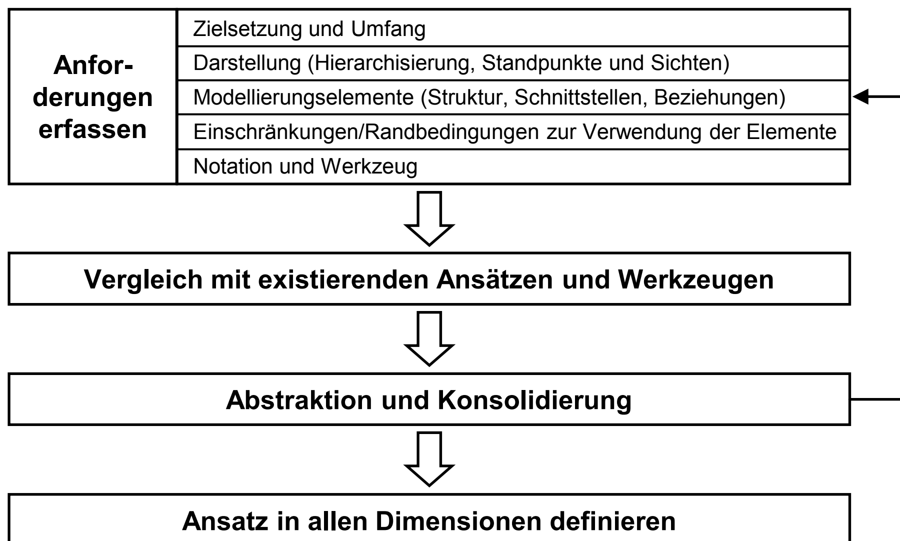

Bild 4.1: Vorgehen zur Entwicklung der Abstrakten Automotive Software-Architektur

Vergleich mit existierenden Ansätzen und Werkzeugen & Abstraktion und Konsolidierung

Im nächsten Schritt wird geprüft, ob Teilaspekte bereits in existierenden Ansätzen (siehe Kapitel 1.2) gelöst wurden und damit verfügbar sind. So kann auch ein Bild darüber gemacht werden, welche Zielsetzungen, Modellierungs-Elemente und Modellierungs-Techniken in verschiedenen Ansätzen vergleichbar sind. Durch bewusste Abstraktion werden diese Gemeinsamkeiten, die aufgrund unterschiedlicher Namensgebung oder Verwendung im Modell nicht immer auf der Hand liegen, identifiziert und für den eigenen Ansatz nutzbar gemacht. Damit können einerseits existierende Ideen – beispielsweise zu sinnvollen Attributen von Strukturelementen – wiederverwendet werden, andererseits wird damit die an bestimmten Stellen geforderte Kompatibilität zu anderen Ansätzen und Werkzeugen erreicht. Beispielhaft sei hier der Bausteintyp „Module“ der ABSOFA genannt, welcher als Obermenge der „AUTOSAR-Software-Component“, dem „ASCET-Module“ und dem „Simulink-Subsystem“ entstanden ist.

Die im Rahmen dieser Aktivitäten erkannten Aspekte werden wiederum für die weitere Verfeinerung der Anforderungen aus Schritt 1 verwendet. Der Prozess der Erstellung der Anforderungen ist somit iterativ. In Bild 4.1 ist dies durch den Rückkopplungspfad visualisiert. Das Endergebnis ist ein konsolidierter, umfassender Stand an Anforderungen an den zu entwickelnden Ansatz.

Ansatz in allen Dimensionen definieren

Mit den oben genannten Aktivitäten sind die Voraussetzungen geschaffen, den Ansatz bezüglich aller relevanter Aspekte in allen Dimensionen ausarbeiten zu können. Im weiteren Verlauf des vorliegenden Kapitels werden diese Ergebnisse vorgestellt.

Aufgrund der speziellen Anforderungen, die an den Ansatz gestellt werden, in Verbindung mit der gefordert hohen Durchgängigkeit und Effizienz in der Anwendung im Vorgehensmodell ist ein neuer, in sich stimmiger und umfassender Modellierungsansatz sinnvoll. Die Wahl fiel unter anderem aufgrund ihrer freien Verfügbarkeit, hohen Verbreitung bei gleichzeitig sehr guter Anpassbarkeit auf die UML 2.0. Die für die ABSOFA notwendigen Erweiterungen und Einschränkungen werden mit einem UML-Profil und damit über eine so genannte leichtgewichtige Änderung des Sprachumfangs realisiert. UML-Profile werden im Gegensatz zu Metamodell-Änderungen so bezeichnet, da sie mit den vorhandenen Elementen auskommen und nur bezüglich deren Verwendung (beispielsweise neue abgeleitete Subtypen oder Interaktionen) geänderte Vorgaben machen.

Im Folgenden soll dargestellt werden, warum der Fokus der ABSOFA auf der statischen Struktur der Software liegen soll. Dynamische Änderungen der Struktur in Form von Konstruktion oder Destruktion von Bausteinen während der Laufzeit spielt in den betrachteten Anwendungen der Automobil-Domäne, insbesondere der Fahrerassistenz, keine Rolle. Benötigte Bausteine werden somit immer zur Entwicklungszeit für ein Software-System festgelegt. Darüber hinaus gilt, dass auch Interaktionen zwischen Bau-

steinen nicht dynamisch während der Laufzeit entstehen können. Auch diese werden bezüglich Vorhandensein und Ausprägung während der Entwicklungszeit festgelegt. Die wichtigste Form der Interaktion von Bausteinen ist dabei der Fluss von Informationen (Signalen). Interaktion und Muster der Kommunikation sind somit als statisch zu verstehen und werden voll vom Ansatz unterstützt. Einzig relevante „Dynamik“ aus Sicht der Software ist somit die Modellierung des Verhaltens der Bausteine. Dies geht über den typischen und sinnvollen Bereich der Betrachtung von Software-Architektur hinaus und obliegt den nachgeschalteten (CASE-)Werkzeugen der Software-Umsetzung. Sollten aufgrund des Verhaltens jedoch bestimmte Randbedingungen existieren, die sich zwingend auf die (statische) Struktur auswirken, so können diese in der ABSOFA angegeben werden. Ein Beispiel dafür ist eine zeitkritische Folge der Abarbeitung von Bausteinen, welche für eine enge Kopplung und gemeinsame Betrachtung dieser Bausteine sorgt. Die Folge ist die Berücksichtigung dieser Kopplung beim Komponieren von übergeordneten Bausteinen und späteren Verteilen dieser auf Plattformen. Das Verhalten kann somit die Struktur beeinflussen. Umgekehrt gilt die Prämisse, dass durch die Gestaltung der Software-Architektur den Software-Entwicklern möglichst viel Freiheit zur Umsetzung des Innenlebens der Bausteine belassen werden soll.

Abschließend soll die ABSOFA noch einmal ausdrücklich im Kontext der AUTOSAR-Initiative eingeordnet werden. AUTOSAR verfolgt ebenfalls in starkem Maße nichtfunktionale Anforderungen. Diese sind jedoch auf einer anderen, höheren Betrachtungsebene angesiedelt. Standardisierung, Wiederverwendung und Portierbarkeit von Software wird herstellerübergreifend angestrebt und soll dort möglichst einfach gemacht werden. Die Hauptleistung ist hierbei die Standardisierung der Basis-Software. Dadurch wird die Möglichkeit geschaffen, dass Software-Komponenten unabhängig von Plattformen entwickelt werden können. Hier findet sich somit der Grundgedanke der Abstraktion und Trennung der fachlichen Logik von der Realisierung wieder. AUTOSAR unterscheidet jedoch Funktions- und Software-Architekturen nicht explizit. Funktionalität wird nur in einem sehr groben Rahmen zu Komponenten zusammengefasst. Ein Fahrerassistenzsystem entspricht zum Beispiel einer einzigen Komponente. Der innere Aufbau dieser wird weder näher betrachtet noch vorgegeben. Die vorliegende Arbeit setzt genau hier an und liefert Antworten darauf, wie Software-Architekturen, speziell der Fahrerassistenz, im Detail aufzubauen sind, damit ein Hersteller seine spezifischen nicht-funktionalen Anforderungen und Ziele erreichen kann. Der Rahmen und die grundsätzliche Zielrichtung, die AUTOSAR bietet, werden somit ausgenutzt und konsequent weiter detailliert.

4.2 Metamodell und UML-Profil

Das UML-Profil, welches die Möglichkeiten zur Modellierung der ABSOFA verbindlich vorgibt und damit als Metamodell zu verstehen ist, wird in zwei Stufen erarbeitet. Zunächst wurden alle verwendbaren Elemente festgelegt. Dazu zählen in erster Linie Struktur- und Schnittstellenelemente sowie die zwischen den Letztgenannten möglichen

Beziehungen. Diese drei Kategorien bilden den Kern der Sprache und werden im vorliegenden Abschnitt näher erläutert. Des Weiteren werden Elemente zur Modellierung von Variation erarbeitet. Dieser Sprachumfang ist als nicht immer benötigte Erweiterung des Kerns zu verstehen und wird daher an späterer Stelle in diesem Kapitel in Abschnitt 4.4 vorgestellt. In der zweiten Stufe wird durch formale Regeln die erlaubte kontextabhängige Verwendung der Elemente beschrieben. Dazu kann die Object Constraint Language (OCL) [87] eingesetzt werden, die von der Object Management Group (OMG) als Standard zur Angabe verbindlicher Randbedingungen (Invarianten) in der UML herausgegeben wurde. OCL-Ausdrücke können von einem UML-Modellierungswerkzeug eindeutig interpretiert und so automatisch die korrekte Anwendung des Profils überprüft werden.

Der gesamte Umfang der ABSOFA ist in gebündelter Form als einfaches Ordnungsschema ohne semantische Informationen in Bild 4.2 dargestellt. Weitere Abbildungen und Erläuterungen zu dem UML-Profil, welche die verwendeten Meta-Elemente, deren semantische Beziehungen und OCL-Konstrukte im Detail beschreiben, sind in Anhang B enthalten.

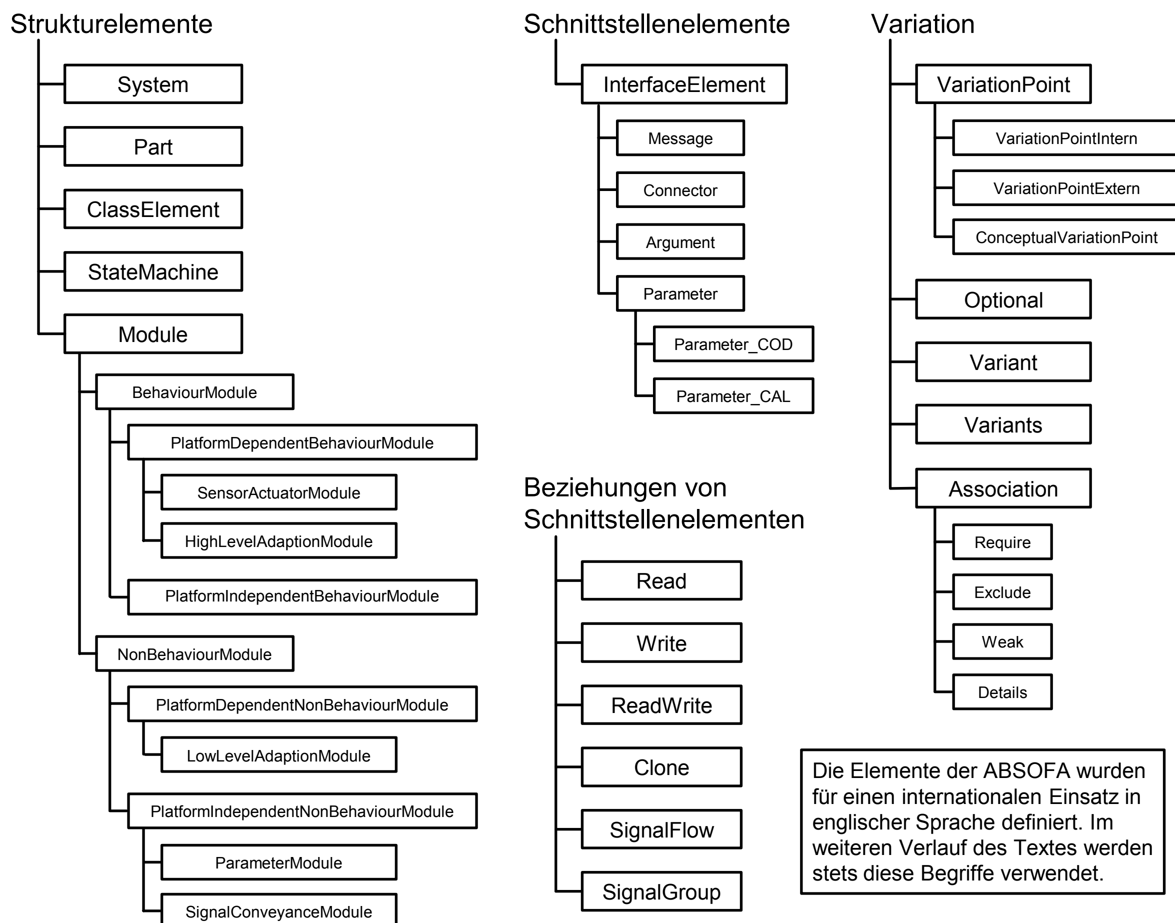

Bild 4.2: Gesamtüberblick über den Umfang der ABSOFA-Modellierung

Über die mit der OCL formal erfassbaren Regeln hinaus wird noch ein Satz an informalen Modellierungsregeln beziehungsweise -richtlinien erarbeitet. Es existiert aktuell keine Möglichkeit oder Werkzeug diese zu formalisieren und automatisch bei der Anwendung zu erzwingen. Es obliegt daher dem Anwender diese einzuhalten.

4.2.1 Standpunkte, Sichten und Diagramme

Zur Modellierung mit der ABSOFA werden zwei Standpunkte vorgeschlagen. Wie bereits erwähnt, sind beide Standpunkte auf die statischen Aspekte der Software-Architektur fokussiert. Der „Abstrakte Standpunkt" repräsentiert die im Rahmen dieser Arbeit wesentlichen, vorgeschlagenen Ideen und Ansätze der konsequenten Abstraktion und somit Trennung von Logik und technischer Realisierung. Der Großteil der gesamten Entwurfstätigkeit für die Software-Architektur geschieht in diesem Standpunkt. Der „Realisierungs-Standpunkt" als Bindeglied und Vorbereitung in die konkrete Umsetzung wird jedoch ebenso ermöglicht und ist im Rahmen des Vorgehensmodells unerlässlich.

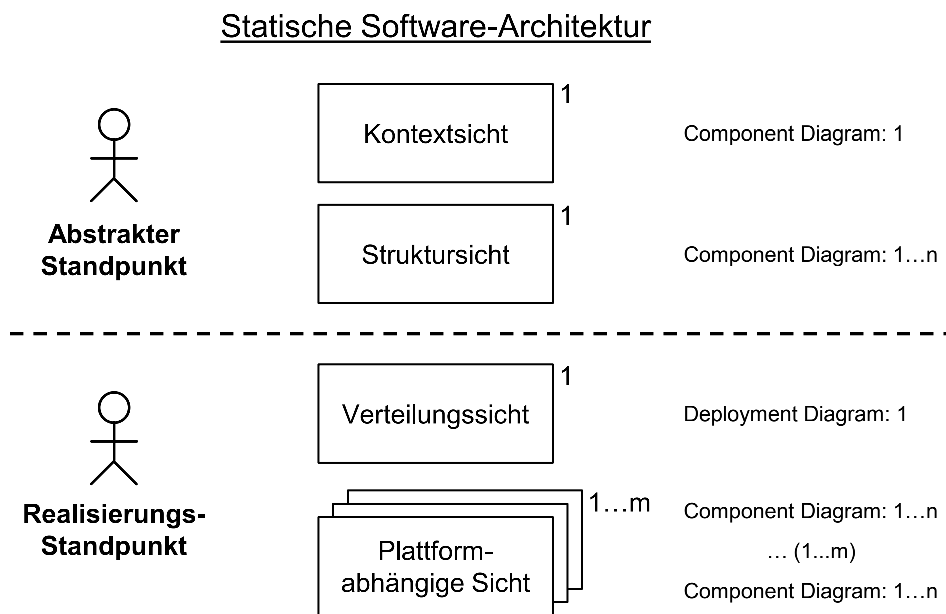

Bild 4.3: Standpunkte, Sichten und Diagramme für die ABSOFA-Modellierung

Die hier beschriebenen beiden Standpunkte genügen dabei ausdrücklich der Obermenge der Ansprüche aller Stakeholder in einem. Es wird als nicht zielführend erachtet, dass für jeden Stakeholder ein eigener Standpunkt definiert wird. Damit würde die Anzahl an Standpunkten erhöht und die Übersichtlichkeit vermindert werden. Jeder Standpunkt besteht aus mehreren Sichten, die wiederum durch mehrere und unterschiedliche Diagramme gebildet werden. Es stehen grundsätzlich alle Informationen, die beteiligte Stakeholder benötigen, zur Verfügung. Die Entscheidung, welche davon genutzt werden

soll, obliegt dem Stakeholder selbst. Ein weiterer Vorteil ist der, dass so alle beteiligten Personen ein Grundverständnis über den Aufbau der Modellierung und Dokumentation aufbauen müssen und so ein Wissen, das über den engsten, jeweils eigenen Bedarf hinaus geht, entsteht. Damit wird die Kommunikation, gegenseitiges Verständnis und Akzeptanz der Beteiligten im gesamten Projekt verbessert. Bild 4.3 zeigt die Standpunkte, Sichten und Diagramme. Die Ziele der vier Sichten sind im Einzelnen die Folgenden:

- **Kontextsicht:** Die Kontextsicht beschreibt die Einbettung des zu entwickelnden Software-Systems in die Umgebung. Das System wird dabei von außen im Sinne einer Blackbox betrachtet. In dieser Sicht werden die Systemgrenzen identifiziert und die nach außen sichtbaren Schnittstellen des Systems festgelegt. Verwendet wird ein einziges (UML-)Component-Diagram.
- **Struktursicht:** In der Struktursicht wird der innere Aufbau des Software-Systems im Sinne einer Whitebox betrachtet. Es sollen alle Architekturbausteine sowie deren Schnittstellen und Beziehungen zueinander identifiziert werden. Die Zerlegung des Systems erfolgt hierarchisch auf beliebig vielen Ebenen. Dazu werden (UML-) Component-Diagrams in unbestimmter Anzahl verwendet. Es ist ebenfalls das Ziel darzustellen, wie die in der Kontextsicht sichtbaren Schnittstellen nach außen im inneren des Systems verwendet und bereitgestellt werden. Die Konsistenz und Übergänge zwischen den Diagrammen muss eindeutig modelliert werden. Variation wird ebenfalls hier modelliert und aufgelöst.
- **Verteilungssicht:** Die Verteilungssicht zeigt die Verteilung der aus dem Abstrakten Standpunkt – insbesondere der Struktursicht – bekannten Architekturbausteine auf physische Rechenplattformen einer konkreten Technischen Architektur. Verteilt werden können nur Bausteine eines bestimmten, festgelegten Typs. Verwendet wird ein einziges (UML-)Deployment-Diagram.
- **Plattformabhängige Sicht:** Die Plattformabhängige Sicht repräsentiert einen für eine konkrete Plattform konfigurierten Teilumfang der Struktursicht. Im Übergang werden nötige Anpassungen vorgenommen, damit die benötigten Bausteine auf der Plattform lauffähig sind. Ein Beispiel ist ein Software-Adapter zur Anpassung eines abstrakten Bausteins an spezifische Basis-Software. Es entsteht dabei eine eigene Sicht pro eingesetzter Plattform. Jede Sicht kann je nach Bedarf zur hierarchischen Darstellung beliebig viele (UML-)Component-Diagrams aufweisen.

4.2.2 Strukturelemente

Die so genannten Strukturelemente der ABSOFA bilden die Übermenge von Bausteintypen, welche die Architekturbausteine repräsentieren. In Bild 4.2 wurden diese bereits kurz vorgestellt, Bild B.1 in Anhang B zeigt sie innerhalb des UML-Profils. Das Hauptziel bei den Strukturelementen ist mit einer möglichst kleinen Anzahl an unterschiedlichen Typen auszukommen. Dadurch soll Komplexität vermindert, das Einarbeiten in die und Anwendung der Modellierung erleichtert sowie ein hoher Wiedererkennungswert in den

unterschiedlichen Sichten erreicht werden. Insgesamt sind fünf unterschiedliche Typen notwendig, welche an dieser Stelle kurz vorgestellt werden:

- **Module:** Das Module nimmt eine besondere Rolle in der Architektur ein. Modules kennzeichnen einen in sich geschlossenen Umfang, der einzeln auf Plattformen verteilbar ist – in anderen Ansätzen auch als „atomic“ bezeichnet. Sie kapseln Funktionalität vollständig nach außen, ihre Kommunikation mit anderen Modules wird über ganz bestimmte, eindeutig beschriebene Schnittstellen, die so genannten „Messages“, bewerkstelligt. Andere Schnittstellen sind ausdrücklich nicht erlaubt. Für die Anwendung der Architekturmetriken ist die Ebene der Modules ebenfalls von besonderer Bedeutung. Unter anderem wird hierfür die innere Komplexität als Attribut angegeben. Modules kann als weiteres Attribut eine Sicherheitsintegrität nach Automotive Safety Integrity Level (ASIL) zugewiesen werden. Als einziger Bausteintyp der ABSOFA verfügt das Module über weitere Subtypen. Modules sind nicht instanziierbar und in der gesamten Software-Architektur einmalig.
- **System:** Das System ist ein Element, welches ermöglicht oberhalb der Module-Ebene Bausteine zu gruppieren. Diese Gruppierung hat primär einen visuellen Charakter und dient der Erhöhung der Übersichtlichkeit und beliebigen Hierarchisierung „nach oben“. Systems haben keine eigenen Schnittstellen sondern propagieren die der Modules. Sie sind nicht instanziierbar.
- **Part:** Das Part ermöglicht in ähnlicher Art und Weise wie das System das Gruppieren von Bausteinen unterhalb der Module-Ebene. Im Gegensatz zum System ist das Part jedoch als stärker gekapselt zu verstehen. Es kann einerseits Schnittstellen lediglich propagieren, andererseits jedoch auch eigene Schnittstellen aufweisen. Das Part weist daher auch Charakter der folgenden beiden Strukturelemente auf. Es ist somit eine Mischung aus einem rein visualisierenden System und einem vollständig kapselnden ClassElement oder StateMachine. Dies ermöglicht die notwendige Flexibilität für eine beliebige Hierarchisierung „nach unten“. Das Part als solches ist nicht instanziierbar, enthaltene Elemente können dies jedoch sein. Daher ist die Enthaltensbeziehung in diesem Fall als Referenz auf diesen Baustein zu verstehen.
- **ClassElement:** Das ClassElement repräsentiert Funktionalität auf einer tieferen Hierarchie-Ebene unterhalb der Modules, welche einerseits durch seine Schnittstellen vollständig gekapselt ist und andererseits potenziell als Schablone für eine mehrfache Verwendung (Instanziierung) dient. Die Funktionalität innerhalb des Elements wird als so eng zusammengehörig verstanden, dass sie immer in Gänze abgearbeitet wird. Durch diese Eigenschaften unterscheidet es sich vom Part. Der Name ist bewusst so gewählt, weil diese Eigenschaften von Klassen in der Objektorientierung bekannt sind. Typischer Inhalt eines ClassElements ist ein in einem Software-System mehrfach verwendeter Regler oder Filter.
- **StateMachine:** Die StateMachine ist identisch zum ClassElement zu verwenden. Der Inhalt repräsentiert jedoch einen Zustandsautomaten. Da diese sehr häufig verwendet werden, bietet sich ein eigener Bausteintyp an.

4.2.3 Schnittstellen

Schnittstellen sind neben den Strukturelementen der andere Teil des Kerns der ABSOFA-Modellierung. Dazu gehören Schnittstellenelemente und Beziehungen zwischen diesen.

Schnittstellenelemente

Alle Schnittstellenelemente sind von einem einzigen Basistyp „InterfaceElement“ abgeleitet, der naheliegenderweise im UML-Profil das Meta-Element „Interface“ erweitert (Bild B.3 in Anhang B). Der Vorteil hierbei ist, dass alle später abgeleiteten Subtypen auf identische Art und Weise durch die vorhandenen Attribute wie (abstrakter) Datentyp, Dimension und Einheit beschrieben werden können. Die Wahl des Subtyps hat dann nur noch Auswirkungen auf die erlaubte Anwendung innerhalb des Modells, nicht auf den Aufbau und die übertragbaren Informationen des Elements an sich. So gehen auch bei beliebig tiefer Hierarchisierung und Wechselwirkung unterschiedlicher Elementtypen keine Informationen verloren. An dieser Stelle soll noch einmal hervorgehoben werden, dass die Beschreibung der Schnittstellenelemente vollständig unabhängig von Informationen zur technischen Realisierung (beispielsweise konkreter Datentyp einer Programmiersprache wie „int8“) erfolgt. Die vier Subtypen werden im Folgenden knapp erläutert, weiterführende Informationen sind Anhang B zu entnehmen.

- **Message:** Die Message ist das zentrale Schnittstellenelement zur Kapselung von Modules und den Informationsaustausch zwischen diesen. Externe Kommunikation zwischen Modules (und den darüber visuell gruppierenden Systems) findet ausschließlich über diesen Mechanismus statt.
- **Argument:** Arguments dienen ebenso der Kapselung und sind daher verwandt zur Message. Sie werden in den Strukturelementen ClassElement, StateMachine und Part verwendet. Eine Verwendung in einem Module ist nicht zulässig.
- **Parameter:** Parameter dienen dazu Software an spezifische Umgebungen anpassbar zu machen und leisten einen großen Beitrag zur Wiederverwendung. Typischerweise ändern sich die Werte von Parametern zur Laufzeit nicht. Es werden Applikations- und Codier-Parameter unterschieden. Parameter dürfen nur auf Module-Ebene verwendet werden.
- **Connector:** Der Connector nimmt eine Sonderstellung innerhalb der Schnittstellenelemente ein. Er stellt keine eigenständige Schnittstelle dar, sondern ist eine Repräsentation („Kopie/Klon“) einer bereits vorhandenen Schnittstelle. Dieser Klon (siehe Beziehungen) wird verwendet, um Schnittstellen durch „weiche“ Bausteingrenzen wie Part und System zu propagieren.

Beziehungen zwischen Schnittstellenelementen

Ebenso wichtig wie die Schnittstellenelemente selbst, ist es die Beziehungen dazwischen anzugeben. Nur so können die Kommunikation und die Wechselwirkungen der Struktur-

elemente vollständig im Sinne der Relevanz für eine Software-Architekturmodellierung beschrieben werden. Grundsätzlich gilt für die gesamte Kommunikation die Annahme, dass sie gemäß dem bekannten „Sender-Receiver-Pattern“ als asynchroner Mechanismus abläuft. Die zulässigen Beziehungen sind in Bild B.4 in Anhang B ersichtlich. Dabei sind die für die Modellierung wesentlichsten und am häufigsten genutzten Beziehungen der SignalFlow, der Clone und die SignalGroup.

- **SignalFlow:** Der SignalFlow verdeutlicht einen Informationsfluss eines Schnittstellenelements zu einem anderen. Er wird immer dann eingesetzt, wenn eine 1-zu-1-Beziehung zwischen Schnittstellenelementen unterschiedlicher Strukturelemente modelliert werden soll. Ziel ist somit zu visualisieren, wie der Signalfluss einer Information durch die unterschiedlichen Hierarchie-Ebenen verläuft. Ein Beispiel hierfür ist die Verbindung einer empfangenen Message an einer Module-Grenze mit einem Eingangs-Argument eines in diesem Module enthaltenen ClassElements.
- **Clone:** Ein Clone verdeutlicht keine Verbindung von zwei Schnittstellenelementen sondern visualisiert das Propagieren von einem einzigen Schnittstellenelement von außen nach innen oder innen nach außen durch „weiche“ Bausteingrenzen hindurch. Ein Ende der Beziehung ist immer ein beliebiges Schnittstellenelement der drei bekannten Typen, das andere Ende ist immer ein Connector.
- **SignalGroup:** Die SignalGroup stellt eine Bündelung einer Menge von Schnittstellenelementen dar, die zwischen zwei Strukturelementen A und B verlaufen. Dadurch kann eine komplexe Kommunikation zwischen zwei Strukturelementen vereinfacht und übersichtlich dargestellt werden. Dies ist insbesondere in der Kontextsicht hilfreich. Sie kann gerichtet oder ungerichtet modelliert werden.

4.3 Entwicklung eines objektiven Bewertungsverfahrens

Wie bereits erläutert wurde, hat die Software-Architektur immensen Einfluss auf die nicht-funktionalen Eigenschaften eines Software-Systems. Die Software-Architektur im Rahmen einer expliziten Entwurfstätigkeit zu betrachten, ist somit richtig und wichtig. Es ist jedoch eine systematische Entwurfstätigkeit notwendig, die gezielt und iterativ die gewünschten Eigenschaften einstellt und dabei auch permanent den aktuellen Entwurf hinsichtlich der Zielerreichung bewertet. Das ist bisher in der Entwicklung nicht der Fall.

Die Entwicklung basiert heute in einem sehr großen Maße auf Erfahrungen und der Intuition der Entwickler beziehungsweise Architekten. Das Wissen ist in wenigen Köpfen gebunden und kann nur schwer an neue Mitarbeiter weitergegeben oder überhaupt nachhaltig für das Unternehmen dokumentiert werden. Auch bei erfahrenen Architekten gibt es keine Garantie, dass am Ende eine gute Software-Architektur erreicht wird. Erst nach Abschluss der Implementierung kann festgestellt werden, wie gut die Software-Architektur in Bezug auf die nicht-funktionalen Eigenschaften geworden ist. Änderungen sind nun jedoch nicht mehr – oder nur unter sehr großem Aufwand – einzuarbeiten.

Diese Unzulänglichkeit soll aufgelöst werden. Dazu ist ein objektives Bewertungsverfahren zu entwickeln, welches auf die Software-Architektur im Abstrakten Standpunkt angewendet wird. Die grundsätzlichen Ziele an den Ansatz wurden bereits in Abschnitt 3.2 erarbeitet, zusammenfassend seien diese noch einmal genannt: Das Verfahren soll eine objektive Bewertung durch quantifizierte Metriken ermöglichen. Es soll darüber hinaus automatisiert arbeiten, um auch für große Systeme effizient und gleichzeitig stets korrekt sowie reproduzierbar anwendbar zu sein. Ein iterativer Entwurf im Bottom-Up- (Neustrukturierung bestehender Systeme) und Top-Down-Vorgehen (Entwurf neuer Systeme) muss unterstützt werden.

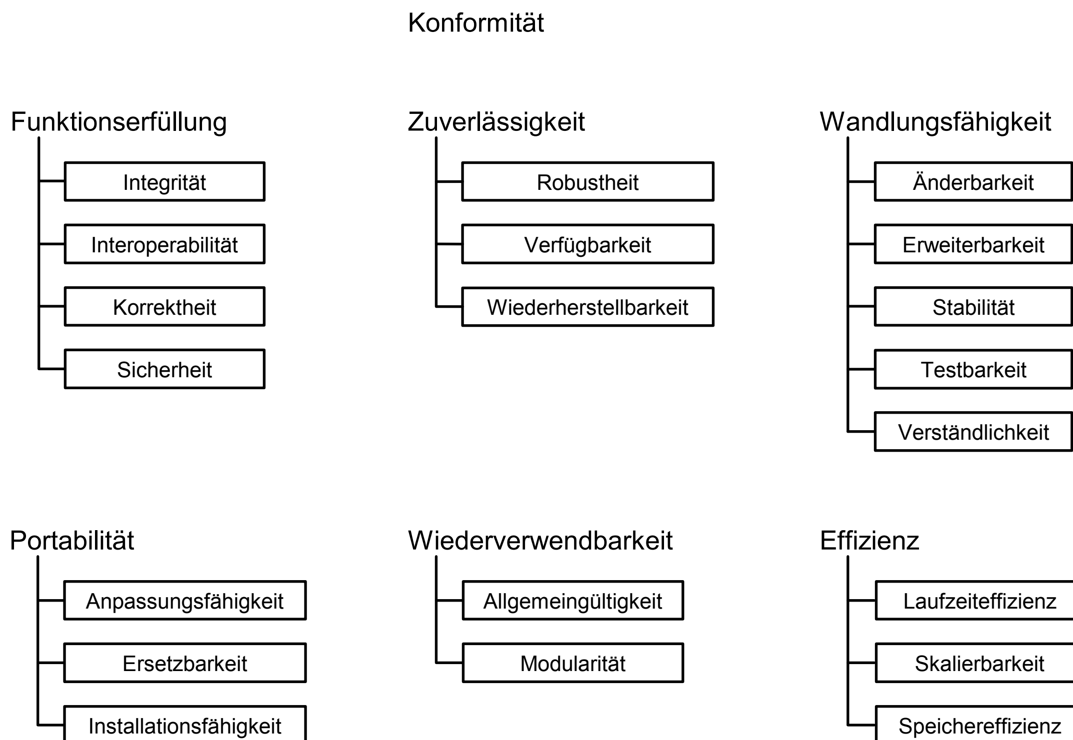

Bild 4.4: Qualitätsmodell zur Klassifizierung und Bewertung von Software-Architektur

Der aktuelle Abschnitt beschreibt das Vorgehen zur Erarbeitung des so genannten Qualitätsmodells, der Metriken und deren Verifikation sowie diskutiert die Anwendbarkeit und Grenzen des Verfahrens. Weitere Details zur Motivation, dem Vorgehen und den Ergebnissen können Ahrens et al., 2010 [5] und 2013 [7] entnommen werden.

4.3.1 Qualitätsmodell und Metriken

Aus der Definition des Begriffs Software-Architektur von Seite 11 ist klar ersichtlich, welche Elemente und Informationen für eine Bewertung einer Software-Architektur überhaupt zur Verfügung stehen. Somit sind die Grenzen der objektiven Bewertung klar ab-

gesteckt. Gleichwohl besteht im Rahmen der Untersuchungen ebenfalls die Möglichkeit bestimmte Informationen, die für die Bewertung zusätzlich notwendig sind, als Erweiterungen des UML-Profils der ABSOFA zu etablieren. Ein Beispiel hierfür ist das Attribut „complexity“ für den Bausteintyp Module. Aufgrund dieser Wechselwirkungen wird das Bewertungsverfahren thematisch der ABSOFA zugeordnet.

Der erste Schritt ist eine möglichst vollständige Übersicht aller grundsätzlich möglichen Eigenschaften einer Software-Architektur aufzustellen. Darin sind zunächst sowohl funktionale als auch nicht-funktionale Kriterien enthalten. Bei der Sammlung der Kriterien kann auf bestehende Ansätze zur Bewertung von Software beziehungsweise Software-Qualität zurückgegriffen werden. Ergebnis ist das in Bild 4.4 dargestellte, so genannte Qualitätsmodell, welches sieben Oberkriterien mit jeweils mehreren – mit Ausnahme der Konformität – Unterkriterien enthält. Oberkriterien, von jetzt an auch als Qualitätskriterien bezeichnet, sind nicht direkt messbar sondern fassen vielmehr ein oder mehrere direkt messbare Kriterien, von jetzt an auch als Qualitätsattribute bezeichnet, zu einer Obergattung zusammen.

Das hier vorgestellte Qualitätsmodell kombiniert, erweitert und ergänzt die aus den unterschiedlichen Ansätzen entnehmbaren Kriterien und passt sie mit Hilfe von Definitionen im Anschluss daran an die speziellen Bedürfnisse des vorliegenden Anwendungsfalls von Software-Architekturen in der Automobil-Domäne an. Die Spezialisierung ermöglicht im Anschluss die Findung von ebenso spezialisierten Metriken. Das Qualitätsmodell als solches ist jedoch für alle Arten von Software-Architektur beziehungsweise auch die Bewertung von Software geeignet. Die Definitionen aller Qualitätskriterien und -attribute, die im Rahmen dieser Arbeit und insbesondere für die Bewertung der Software-Architektur verwendet werden, befinden sich in Anhang C. Damit werden die im bisherigen Verlauf der Arbeit lediglich recht allgemein erwähnten, möglichen funktionalen und nicht-funktionalen Eigenschaften von Software und Software-Architekturen für das bessere Verständnis konkretisiert.

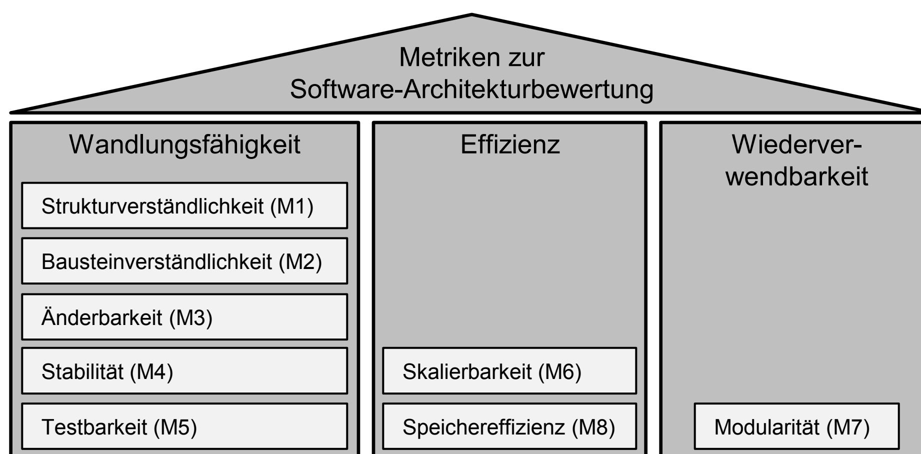

Bild 4.5: Übersicht über die entwickelten Software-Architekturmetriken

Für die Entwicklung konkreter Metriken sollen ausschließlich nicht-funktionale Qualitätskriterien betrachtet werden. Es wurden dabei die Kriterien Wandlungsfähigkeit, Effizienz und Wiederverwendbarkeit als am viel versprechendsten und am dringendsten benötigt identifiziert. Innerhalb dieser wurden insgesamt acht quantifizierbare Metriken identifiziert und erarbeitet (Bild 4.5).

Ziel aller Architekturmetriken ist es, Designentscheidungen bezüglich ihrer Auswirkungen auf ausgewählte Qualitätsattribute hin objektiv messbar zu machen. Dazu ist es notwendig, subjektive Einschätzungen, implizite Architekturmuster und Best Practices erfahrener Mitarbeiter der Domäne zugänglich zu machen und in eine objektive Form zu überführen. Dies wird durch Befragungen mit standardisierten Fragebögen sowie die Analyse von Fallbeispielen erreicht. Darüber hinaus werden diese mit aus der Literatur bekannten, allgemeingültigen Best Practices abgeglichen.

Es zeigt sich, dass grundsätzlich für nahezu alle der relevanten Attribute der drei Kriterien eine Möglichkeit besteht, mit Hilfe der in der Software-Architektur zur Verfügung stehenden Informationen eine quantitative Metrik herzuleiten. Lediglich für die Kriterien Erweiterbarkeit und Allgemeingültigkeit lässt sich zum jetzigen Zeitpunkt kein geeignetes Verfahren finden. Dies liegt in beiden Fällen daran, dass reines Strukturwissen nicht ausreichend ist, um eine zuverlässige Aussage zu treffen. Jede Architektur ist grundsätzlich erweiterbar, wie „gut“ oder einfach allerdings eine solche Erweiterung vorgenommen werden kann, liegt ganz konkret an der Funktionalität, die hinzugefügt werden soll. Somit ist Kontextinformation, mit anderen Worten Detailwissen über mögliche künftige Erweiterungen, zwingend erforderlich. Ähnlich verhält es sich mit der Allgemeingültigkeit, die ebenso nicht ausschließlich auf Strukturinformationen fußend bewertet werden kann. Als Beispiel für eine Metrik wird im Folgenden die Metrik „Modularität – M7“ vorgestellt. Alle weiteren befinden sich detailliert in Anhang D.

Metrik zur Modularität – M7

Idee

Eine gute Modularität zeichnet sich durch eine starke Kohäsion und schwache Kopplung aus. Logisch zusammenhängende Bausteine, die viele Informationen austauschen, sollten demnach zu einer Komponente zusammengefasst werden. Eine starke Zusammengehörigkeit und schwache äußere Abhängigkeit kann in der modellgetriebenen Entwicklung vor allem über das Verhältnis interner zu externer Kommunikation überprüft werden. Das gilt für alle Strukturelemente auf beliebigen Hierarchie-Ebenen.

Erläuterungen

Für diese Metrik wird kein zusätzlicher Faktor benötigt, maßgeblich ist für jeden Baustein allein das Verhältnis zwischen Informationen, die auf derselben Hierarchie-Ebene ausgetauscht werden, zu der Gesamtzahl aller (internen und externen) Informationen, die an diesem Baustein verwendet werden. Je nachdem wie detailliert die Software-Architektur dokumentiert ist, kann eine Bewertung nicht nur bis zur Module-Ebene,

sondern auch unterhalb dieser stattfinden. Zunächst ermittelt die Formel das Verhältnis zwischen interner und externer Kommunikation eines Bausteins. Mit einem Gewicht von 1:1 fließen rekursiv die Modularitätswerte der gegebenenfalls vorhandenen intern enthaltenen Bausteine in die Berechnung ein. Der Wertebereich liegt zwischen 0 und 1.

Metrik

$$M7_i = \frac{\frac{\sum Mess_{ext,i}}{\sum Mess_{ext,i} + \sum Mess_{int,i}} + \frac{\sum_{j=1}^{n} M7_j}{n}}{2} \tag{4.1}$$

$$1 \leq j \leq n$$
$$i \mathrel{\hat{=}} \textit{Aktuelle Komponente } i$$
$$n \mathrel{\hat{=}} \textit{Anzahl aller Subkomponenten von } i$$
$$Mess_{ext,i} \mathrel{\hat{=}} \textit{Ein- oder ausgehende Schnittstelle von } i \textit{ zu einer anderen Komponente}$$
$$Mess_{int,i} \mathrel{\hat{=}} \textit{Interne Schnittstelle zwischen Subkomponenten der Komponente } i$$
$$M7_i \mathrel{\hat{=}} \textit{Modularität der aktuell untersuchten Komponente } i$$
$$M7_j \mathrel{\hat{=}} \textit{Modularität von Subkomponente } j$$

Anwendungsbeispiel

Das folgende, einfache Beispiel soll die Berechnung der Metrik noch einmal verdeutlichen. Eine exemplarische Software-Architektur (Bild 4.6) besteht aus vier Systems. Auf der obersten Ebene werden diese zu einem fiktiven Wurzelelement zusammengefasst, das ist für die spätere Berechnung notwendig. Die Software-Architektur ist bis zur Module-Ebene detailliert. Das heißt, die Berechnung kann nur bis zu dieser Ebene erfolgen. Von Seite der Metrik ist auch eine Berechnung auf (beliebig) tieferer Ebene möglich.

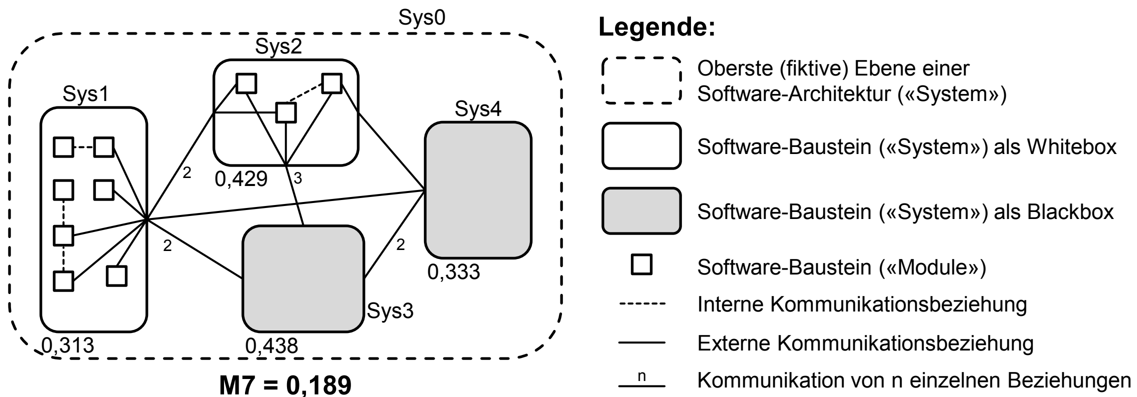

Bild 4.6: Akademisches Beispiel zur Berechnung der Metrik M7 (Modularität)

Zur Vereinfachung ist der innere Aufbau von zwei der vier Modules (Sys3 und Sys4) nicht dargestellt. Die Werte der Modularität für diese Bausteine sind bereits ermittelt und angetragen (0,438 und 0,333). Da das Verfahren rekursiv arbeitet, müssen die Werte je Baustein immer von innen nach außen ermittelt werden. Sys1 hat auf der Module-Ebene insgesamt 5 externe und 3 interne Kommunikationsbeziehungen. Der linke Term des Zählers der Metrik wird somit zu $\frac{5}{5+3} = 0,625$. Da auf der jeweils niedrigsten Ebene keine Modularitätswerte von Unterbausteinen existieren, wird der rechte Term des Zählers nicht ausgewertet und zu null gesetzt (kein Dividieren durch $n = 0$ möglich). Der Metrikwert des Bausteins ergibt sich somit zu $M7_{Sys1} = \frac{\frac{5}{5+3}+0}{2} = 0,313$. Auf identische Art und Weise wird $M7_{Sys2} = \frac{\frac{6}{6+1}+0}{2} = 0,429$ ermittelt.

Eine Ebene höher wird nun vom Prinzip her identisch verfahren. Zwischen den Systems Sys1…Sys4 gibt es 11 (interne) Kommunikationsbeziehungen. Da die höchste Ebene bereits erreicht ist und die Bausteine nur fiktiv gruppiert sind, gibt es per Definition keine externe Kommunikation. Somit wird der linke Term der Berechnung zu null, es fließt nur der Mittelwert der Bausteine ein: $M7 = \frac{\frac{0}{0+10}+\frac{0,313+0,429+0,438+0,333}{4}}{2} = 0,189$.

4.3.2 Umsetzbarkeit und Anwendbarkeit des Verfahrens

Die Metriken wurden mit Hilfe der Entwicklungsumgebung Eclipse für Java [114] in einem Software-Prototypen realisiert. Details zu dieser Umsetzung und Anwendung des Prototypen finden sich in Abschnitt 4.5.3.

Zur Verifikation der Metriken, das bedeutet zur Überprüfung, ob diese richtig gemäß Spezifikation umgesetzt wurden, sind in einer überschaubaren, exemplarischen Software-Architektur Testfälle erarbeitet und angewendet worden. Diese Testfälle haben isolierte Änderungen an der Software-Architektur vorgegeben, die sich gezielt auf bestimmte Metrikergebnisse mit möglichst wenig Wechselwirkungen auswirken. Die Verifikation wurde durch manuelle Berechnung der Metriken (Ergebnis sind die korrekten Werte) und Gegenüberstellung der Ergebnisse mit den durch den Software-Prototypen ermittelten Werten realisiert. Im vorliegenden Abschnitt werden die generelle Anwendbarkeit und die Grenzen des Ansatzes diskutiert.

Alle Metriken können grundsätzlich unabhängig voneinander eingesetzt werden. Dies ermöglicht einen flexiblen und individuellen Einsatz je nach den Bedürfnissen des bewertenden Entwicklers. Für eine gesamthafte Architektur-Bewertung werden alle Metriken einzeln berechnet und in einer bestimmten Gewichtung ins Verhältnis gesetzt. Die Gewichtung der konkreten Metriken (für Qualitätsattribute) bestimmt den daraus ermittelten Wert für das übergeordnete Qualitätskriterium. Die Qualitätskriterien werden ebenfalls zueinander gewichtet und so ein einziger Gesamtwert für die Software-Architektur ermittelt. Tabelle 4.1 zeigt die initiale und gleichzeitig aktuell im Prototypen umgesetzte Gewichtung.

Auch wenn eine absolute Bewertung einer einzigen Architektur in diesem Vorgehen möglich ist, hat sich gezeigt, dass ein relativer Vergleich zweckmäßiger ist. Dies liegt zum

Tabelle 4.1: Initiale Gewichtung aller Qualitätskriterien und -attribute

Kriterium	Gewicht	Metrik	Gewicht
Wandlungsfähigkeit	2	Strukturverständlichkeit	1
		Bausteinverständlichkeit	1
		Änderbarkeit	3
		Stabilität	1
		Testbarkeit	2
Effizienz	2	Skalierbarkeit	2
		Speichereffizienz	1
Wiederverwendbarkeit	1	Modularität	1

einen daran, dass die Metriken prinzipbedingt unterschiedliche Wertebereiche haben, was ein Kombinieren zu einem einzelnen absoluten Gesamtwert erschwert. Zum anderen ist in der aktuellen Phase der Entwicklung des Verfahrens eine Einordnung der quantitativen Metrikwerte nicht möglich. Erst mit steigender Erfahrung durch zahlreiche Bewertungen von unterschiedlichen Projektarten und -umfängen kann für künftige Projekte auch über den absoluten Zahlenwert eine Einschätzung der Architekturgüte vorgenommen werden.

Bis diese Erfahrung vorliegt, wird stets eine relativ vergleichende Architekturbewertung vorgeschlagen, bei der eine so genannte Referenz- mit einer Vergleichsarchitektur bewertet und direkt verglichen werden. Entscheidendes Kriterium für eine Verbesserung der Architektur ist die relative Verbesserung von einem oder mehreren Metrikwerten. Über die steigende Erfahrung im Umgang mit dem Verfahren können Wertebereiche, innerhalb der Metriken verwendete Faktoren sowie die Gewichtung der Metriken in der Gesamtbewertung noch feinparametriert werden. Bis dahin kann der absolute Zahlenwert einer relativen Änderung nur bedingt als Gütemaß der Verbesserung oder Verschlechterung herangezogen werden. Entscheidend ist das Vorzeichen der Veränderung.

4.4 Variantenmodellierung und Variantenmanagement

Die Beherrschung der Variantenvielfalt stellt eine besondere Herausforderung in der Automobil-Domäne dar. Dies gilt in gleichem Maße für softwaredominante Systeme sowie Fahrerassistenzsysteme und damit auch für deren Software-Architekturen. Damit das Potenzial der ABSOFA zur Erreichung der bereits vorgestellten Ziele voll ausgeschöpft werden kann, muss Variabilität im Modell abgebildet werden können. So kann lange in einem einzigen, gesamthaften fachlichen Modell gearbeitet werden, welches erst zum letztmöglichen Zeitpunkt in die konkrete Realisierung übergeht.

Die notwendige Variabilität einer Software-Architektur wird durch eine Vielzahl an unterschiedlichen Aspekten beziehungsweise Einflussfaktoren bestimmt. Diese werden

auch als Variantentreiber bezeichnet. Aufgrund der Komplexität und Vernetzung von Fahrerassistenzsystemen ist ein Großteil davon in diesem Anwendungsbereich relevant. Es lässt sich eine Unterteilung in interne und externe Variantentreiber vornehmen.

Unter internen Variantentreibern werden alle Ursachen von Variation zusammengefasst, die ihren Ursprung innerhalb der Organisation haben und somit durch den Fahrzeughersteller direkt beeinflusst werden können. Hierzu zählen beispielsweise:

- Marken, Produktlinien, Fahrzeuge, Derivate,
- Bordnetz-Topologien und Steuergeräte-Varianten,
- Ausprägungen von Sensoren, Bedien- und Anzeigeelementen,
- Vielfalt der Antriebsarten (Verbrenner, Hybrid, elektrisch), Motoren- und Getriebeprogramm sowie Bremsen- und Lenkungsvarianten und
- Lieferantenstrategie und -vielfalt, Paketierung von Sonderausstattungen.

Externe Variantentreiber sind dem gegenüber Ursachen von Variation, die ihren Ursprung nicht innerhalb der Organisation haben und folgerichtig nicht durch den Fahrzeughersteller direkt beeinflusst werden können. Beispiele hierfür sind:

- Märkte und Kunden,
- Wettbewerb,
- Gesetzgebung und Normen.

Bei dieser Unterteilung ist anzumerken, dass externe und interne Variantentreiber nicht rückwirkungsfrei sind. Beispielsweise wird das Motorenprogramm maßgeblich von Vorgaben des Gesetzgebers zu Flottenverbrauch oder Emissionen beeinflusst. Im Rahmen der ABSOFA erfolgt die Modellierung von Variabilität ebenfalls abstrakt. Was dies genau bedeutet und welche Modellierungskonstrukte zu diesem Zweck als zusätzlicher Umfang eingeführt wurden, wird im weiteren Verlauf dieses Abschnitts beschrieben.

Bild 4.7 verdeutlicht, wie die Modellierung von Variabilität über die unterschiedlichen (Meta-)Ebenen der Modellierung mit der ABSOFA verteilt ist. ME0 legt dabei die zulässigen Konstrukte und Regeln zu deren Verwendung fest. Dies ist somit die Ebene eines Metamodells und entspricht in diesem Fall dem UML-Profil der ABSOFA. ME0 ist das Rüstzeug, um in Modellen beliebiger Anwendungsfälle beliebige Variabilität beschreiben zu können. Dies wird durch das Symbol für Unendlichkeit angedeutet. ME1 entspricht einem konkreten Modell eines solchen Anwendungsfalls. Ein Beispiel ist die Software-Architektur der Fahrerassistenzsysteme der Längsführung, welche im weiteren Verlauf der Arbeit noch wichtig werden wird. Hierbei wird unterstellt, dass diese Architektur als ein Gesamtmodell mit allen grundsätzlich vorhandenen Kundenfunktionen für alle denkbaren Kombinatoriken und Ausprägungen modelliert werden soll. Die Architektur ist durch diese Variabilitätsinformationen „überbestimmt“. Daher ist in der Literatur auch die Bezeichnung 150%-Modell üblich. Das Modell auf Ebene ME1 entspricht der vorgestellten Struktursicht im Abstrakten Standpunkt. Wenn eine konkrete Variante weiter betrachtet werden soll, muss eine Ausleitung aus dem 150%-Modell in Form einer

Konfiguration erfolgen. Bei dieser werden nach formalen Vorgaben alle Variationspunkte aufgelöst und daraus ein neues, in sich stimmiges, Modell erzeugt. Die Konfiguration in das 100%-Modell auf Ebene ME2 ist ebenfalls noch Teil der Struktursicht im Abstrakten Standpunkt. Wenn in einem Anwendungsfall keinerlei Variabilität vorliegt, gibt es die explizite Unterscheidung zwischen ME1 und ME2 nicht.

ME0	Metamodell (UML-Profil)	∞
ME1	Gesamtmodell Anwendungsfall	150%
ME2	Instanz (Konfigurierte Variante)	100%

Bild 4.7: Meta-Ebenen der Variantenmodellierung in der ABSOFA

Im Folgenden werden die konkreten Erweiterungen des Kernumfangs der ABSOFA um die Modellierung von Variabilität beschrieben. Wie bereits in den Abschnitten zuvor wird an dieser Stelle ein kompakter Überblick gegeben. Abbildungen und weiterführende Informationen finden sich in Anhang B. Die folgenden drei Fragen adressieren alle für diesen Modellierungs-Ansatz relevanten Aspekte:

1. Was variiert?
2. Wie variiert es?
3. Welche Variationsbeziehungen existieren?

Was variiert?

Um diese Frage zu beantworten wird das Konzept des „VariationPoint“ (siehe Bild B.5) eingeführt. Dieser lokalisiert die variablen Stellen in der Architektur und hebt sie hervor. Das kann von einzelnen Elementen bis hin zu komplexen Gebilden reichen und ist auf jeder beliebigen Ebene (Hierarchie) möglich. Ein VariationPoint kann dabei jedes der in den vorhergehenden Kapiteln beschriebenen Struktur- sowie Schnittstellenelemente oder eine Beziehung zwischen Schnittstellenelementen sein. Es handelt sich hierbei nicht um ein eigenständiges, zusätzliches Modellierungselement. Vielmehr wird der Stereotyp den vorhandenen Elementen bei Bedarf zusätzlich zugewiesen. Dies wird von der UML unterstützt, indem mehrere Stereotypen durch Kommata getrennt werden. Der VariationPoint ist das Standard-Konstrukt zur Kennzeichnung variabler Stellen. Für besondere Anwendungsfälle kann auf seine drei Subtypen zurückgegriffen werden. Dabei handelt es sich um den ConceptualVariationPoint zur Modellierung komplexer Variationsgebilde mit Wechselwirkungen zwischen verschiedenen Elementen, dem VariationPointIntern zur Kennzeichnung von SignalGroups, bei denen Signale im Inneren variieren sowie den VariationPointExtern. Letzterer ermöglicht die Kennzeichnung und den Verweis auf externe Elemente, die im aktuellen Diagramm nicht eingezeichnet sind.

Wie variiert es?

Zur Beschreibung, wie ein Element variiert, gibt es die folgenden zwei Typen (Bild B.6):

- **Optional:** Optionalität drückt aus, dass das betroffene Element je nach Kontext enthalten oder nicht enthalten ist. Die zwei wesentlichen Informationen sind erstens, dass das Element optional ist und zweitens unter welchen Bedingungen es enthalten ist. „Optional“ kann ebenso wie VariationPoint Schnittstellenelementen als zusätzlicher Stereotyp zugewiesen werden. Für optionale Strukturelemente wird ein eigenes Element vom Typ „Optional“ modelliert, welches per Abhängigkeitsbeziehung auf das Strukturelement verweist.
- **Variant(s):** Varianten werden immer dann benötigt, wenn ausgedrückt werden soll, dass von einem Element mehrere Ausprägungen existieren. Die wesentlichen Informationen sind hier, dass das Element Varianten aufweist und unter welchen Bedingungen, welche Variante ausgewählt wird. Der Stereotyp im Plural „Variants“ wird dabei dem betroffenen Element zugewiesen. Jede der möglichen Ausprägungen davon ist eine einzelne „Variant“ (Singular).

Welche Variationsbeziehungen existieren?

Unterschiedliche VariationPoints stehen sehr häufig in Beziehung zueinander. Dies bedeutet, dass die Auflösung eines solchen, die Auflösung eines weiteren beeinflusst. Der Lösungsraum zur Auflösung des zweiten wird hierbei eingeschränkt oder bereits in Gänze mit festgelegt. Folgende Beziehungen sind möglich (Bild B.6):

- **Require:** Diese Beziehung entspricht einer Zwangskopplung. Wenn eine bestimmte Variante an der einen Stelle gewählt wird, muss eine bestimmte Variante an einer anderen Stelle ausgewählt werden.
- **Exclude:** Diese Beziehung entspricht einem Ausschluss. Wenn eine bestimmte Variante an der einen Stelle gewählt wird, kann eine bestimmte Variante an einer anderen Stelle nicht mehr ausgewählt werden. Require und Exclude sind auf einer höheren Granularitätsebene Neuwagenkäufern aus den Fahrzeug-Konfiguratoren der Hersteller bezüglich Auswahl von Sonderausstattungen bekannt.
- **Weak:** Die Weak-Beziehung erlaubt die Zusammengehörigkeit zweier Variabilitäten anzuzeigen, wobei der genaue Zusammenhang (noch) nicht spezifiziert ist. Insbesondere in einer frühen Phase des Entwurfs ist dies hilfreich, um wahrscheinliche Wechselwirkungen frühzeitig kenntlich machen zu können, die erst zu einem späteren Zeitpunkt konkretisierbar sind.
- **Details:** Die Details-Beziehung drückt aus, dass die Variabilität eines Elementes vollständig durch die Variabilität eines oder mehrerer darin enthaltener Subelemente aufgelöst beziehungsweise näher spezifiziert wird. Details ist nur bei Strukturelementen anwendbar.

4.5 Datenmodell und Austauschformat für Modelltransformationen

Der vorliegende Abschnitt beschäftigt sich mit der Frage, wie die im Rahmen dieser Arbeit vorgeschlagenen Verbesserungen und Erweiterungen des Vorgehensmodells rund um die Abstrakte Software-Architektur konkret miteinander in Beziehung zu setzen sind. Es werden neue Prozessschritte eingeführt, welche sowohl untereinander als auch mit den bereits Vorhandenen wechselwirken und dabei Übergänge erzeugen. Für eine hohe Akzeptanz und Anwendbarkeit der angestrebten Neuerungen müssen die verschiedenen Übergänge durch einen einzigen, durchgängigen und flexibel erweiterbaren Ansatz realisiert werden. Vielfältige Ansätze und Werkzeuge sind schwieriger zu erlernen und erschweren damit ihre Anwendbarkeit und Durchdringung. Gleichzeitig würde bei den designierten Nutzern der Fokus auf die isolierten Prozessschritte gelenkt anstelle ein umfassendes Gesamtverständnis und den Blick aufs große Ganze zu fördern.

Die Lösung für diese Fragestellung ist ein zentrales Datenmodell für die Ablage und Weiterverarbeitung der Software-Architektur und ein darauf aufbauender, universeller Ansatz zur Modelltransformation und -analyse. Die Einbettung des Datenmodells in das Vorgehensmodell wurde in Bild 3.4 auf Seite 44 bereits dargestellt. Es ist sowohl Bindeglied zu externen Werkzeugen der Implementierung (ASCET, Simulink) als auch die Basis zur Anwendung der Algorithmen zur Bewertung, Varianten-Konfiguration und Überführung der Abstrakten in eine Plattformabhängige Software-Architektur.

Die folgenden Unterabschnitte beschreiben die Ziele und Eigenschaften des Datenmodells, das Vorgehen zu seiner Entwicklung sowie die konkrete Umsetzung und Anwendung im Rahmen der vorgeschlagenen Prozessschritte im Detail.

4.5.1 Ziele und Eigenschaften des Datenmodells

Das Vorgehen um das Datenmodell zu erarbeiten, entspricht grundsätzlich ebenfalls dem in Bild 4.1 auf Seite 46 dargestellten Vorgehen zur Entwicklung der ABSOFA: Anforderungen erfassen, Vergleich mit existierenden Ansätzen, Abstraktion und Konsolidierung sowie konkrete Ausarbeitung beziehungsweise Umsetzung. Die wesentlichen Ziele an das Datenmodell und die daraus abgeleiteten Anforderungen wurden in den folgenden Kategorien ermittelt. Die Anforderungen im Detail finden sich in Tabelle A.1 in Anhang A.1. Einige der Anforderungen sind nicht ausschließlich an das Datenmodell gerichtet sondern werden nur in Kombination mit dem damit eng verbundenen Ansatz zur Modelltransformation erfüllt.

- **Umfang:** Hiermit wird festgelegt, was alles im Datenmodell abzulegen ist.
- **Handhabung:** Unter Handhabung wird verstanden, welche Anwendungsfälle und Schritte zur Weiterverarbeitung mit dem Datenmodell ermöglicht werden müssen.
- **Format:** Alle Anforderungen, die an das Datenformat an sich gestellt werden, beispielsweise die Verwendung frei verfügbarer Standards.

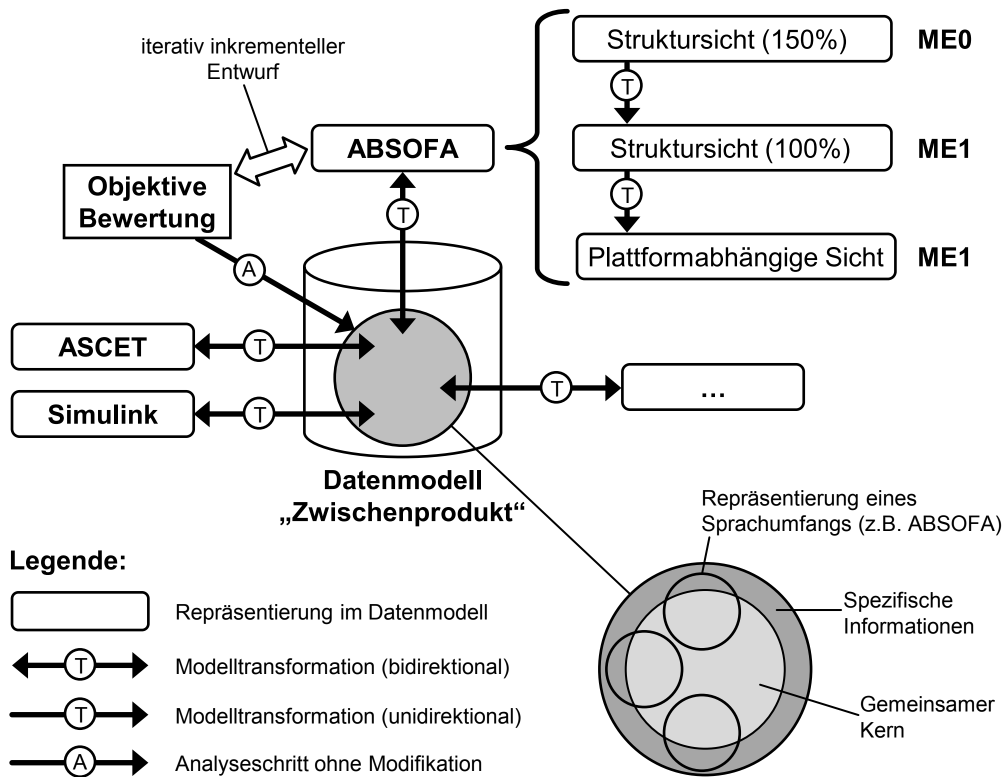

Bild 4.8: Das zentrale Datenmodell im Kontext der Werkzeuge und Sichten

Bild 4.8 zeigt das Datenmodell und seine Bedeutung im Kontext der verwendeten Werkzeuge und Sichten auf die Software-Architektur. Es existiert ein gemeinsamer Kern an Informationen, der in allen Werkzeugen – die ABSOFA wird in dieser Darstellung vereinfacht ebenfalls als Werkzeug angesehen – verwendet wird. Dieser Kern wird analog zur Definition der ABSOFA durch eine Abstraktion im Sinne einer Generalisierung von Informationen erhalten. Damit ist gemeint, dass die Informationen aller Werkzeuge gegenübergestellt werden und Informationseinheiten, die einen identischen oder sehr verwandten Sachverhalt beschreiben, zu einem abstrahierten Datenelement zugeordnet werden. Die Zuordnung der spezifischen Elemente in den jeweiligen Werkzeugen zu dem abstrahierten Element ist gleichzeitig Basis für die später erstellten Transformations-Regeln. Da ein ähnliches Vorgehen bereits bei der Entwicklung der ABSOFA durchgeführt wurde und die ABSOFA somit zu gewissen Teilen bereits eine Abstraktion und Vereinheitlichung von spezifischen Ansätzen der übrigen Werkzeuge darstellt, ist eine große Nähe zwischen der Repräsentation von Informationen in der ABSOFA und dem Datenmodell vorhanden. Die ABSOFA hat somit eine Sonderstellung innerhalb der Werkzeuge.

Der gemeinsame Kern ist demnach der Teil des Datenmodells, der zwischen allen Werkzeugen ohne Informationsverlust ausgetauscht werden kann. Es hat sich jedoch gezeigt, dass es ebenfalls Informationen gibt, die nicht in allen Werkzeugen beziehungsweise nur in einem der Werkzeuge benötigt werden. Damit keine Informationen verloren

gehen und somit ein echtes Roundtrip-Verfahren möglich ist, können diese ebenfalls im Datenmodell abgelegt werden. In Bild 4.8 sind die unterschiedlichen Anteile des Datenmodells im Detailausschnitt unten rechts als Prinzipdarstellung visualisiert. Anzumerken ist des Weiteren, dass die Werkzeuge ASCET und Simulink primär für die Software-Umsetzung verwendet werden und keine originären Werkzeuge der Modellierung von Software-Architekturen darstellen. Es wurden aus allen verfügbaren Informationen dieser Werkzeuge nur solche in das Datenmodell aufgenommen, die relevant für die Modellierung von Software-Architektur sind. Einige weitere Details zum Vorgehen bezüglich des Vergleichs der Werkzeuge mit dem Ziel eine abstrahierte Sicht zu gelangen, finden sich in Ahrens et al., 2008 [8]. In Bild 4.8 ist darüber hinaus zu sehen, dass das Datenmodell ebenfalls alle unterschiedlichen Sichten und Konfigurationen innerhalb der ABSOFA darzustellen vermag. Für alle darin vorkommenden Modellierungselemente und Beziehungen gibt es eine entsprechende Repräsentation. Zu den einzelnen Sichten werden korrespondierende Dateien im Format des Datenmodells erzeugt. Das objektive Bewertungsverfahren wird an einer konkreten Instanz des Datenmodells angewendet. Dies ist als spezieller Zugriff in Form einer „Analyse" visualisiert, da im Gegensatz zu den Transformationen keine Änderungen am Datensatz vorgenommen werden. Für diesen Zugriff und alle dargestellten Transformationen wird derselbe Ansatz verwendet. Dies wird in Abschnitt 4.5.3 im Detail diskutiert.

Bild 4.9: Anzahl notwendiger Regelsätze für Modelltransformationen ohne zentrales Datenmodell am Beispiel von drei und vier Werkzeugen

Durch alle genannten Maßnahmen wird das Datenmodell zum zentralen Bestandteil im angepassten Vorgehensmodell und deckt alle gestellten Anforderungen, die eingangs erwähnt wurden, ab. Ein weiterer Vorteil des zentralen Datenmodells liegt in der Effizienz in Bezug auf die Modelltransformationen zwischen den Werkzeugen. Für eine bidirektionale Modelltransformation sind immer zwei Regelsätze notwendig. Für den Fall, dass die Transformationen direkt zwischen den Werkzeugen ablaufen, sind für n Werkzeuge immer $(n^2 - n)$ Regelsätze vonnöten. Bild 4.9 verdeutlicht dies am Beispiel von drei beziehungsweise vier Werkzeugen. Wird demgegenüber jedoch immer ein zentrales Datenmodell dazwischen geschaltet, wie es in dem hier vorgestellten Ansatz der Fall ist, reduzieren sich die notwendigen Regelsätze auf $(2n)$. Im Hinblick auf mögliche zukünftige Erweiterungen um weitere Werkzeuge ist dieser Ansatz vielversprechend. In der aktuellen

Konstellation mit drei Werkzeugen hält sich der Aufwand mit sechs Regelsätzen genau die Waage. Da das Datenmodell immer zwischen den beteiligten Werkzeugen angesiedelt ist, wird es im weiteren Verlauf der Arbeit auch als „Zwischenprodukt“ bezeichnet.

4.5.2 Modellierung und Umsetzung des Datenmodells

Im Folgenden soll die konkrete Umsetzung des Zwischenprodukts in einem Datenformat beschrieben werden. Ausgewählt wird die Extensible Markup Language (XML). Diese ist inzwischen sehr weit verbreitet und als offener Standard kostenfrei verfügbar. XML ist mit einer definierten Semantik und Syntax ausreichend formal für maschinelle Verarbeitung und gleichzeitig für den Menschen lesbar, da es als wohl strukturiertes Textdokument aufgebaut ist. Bekannte UML-Werkzeuge bieten ausnahmslos den Ex- und Import über XML-Dateien an. Gleiches gilt für ASCET. Simulink sieht zwar keinen direkten Austausch mit XML-Dateien vor, es sind jedoch eigenständige Erweiterungen wie SimEx von der Firma ITPower Solutions [58] verfügbar, die eine Transformation von Simulink-Dateien im .mdl-Format nach XML ermöglichen. Auch AUTOSAR nutzt zur Beschreibung von Software-Komponenten (SWC) das XML-Format in Form einer eigenen Definition namens AUTOSAR XML (ARXML). Es ist zu erwarten, dass die Unterstützung dieses Formats künftig noch weiter zunimmt, so dass auch für die Erweiterung um weitere Werkzeuge die XML eine gute Wahl darstellt.

Zur Festlegung des Aufbaus von XML-Dateien werden so genannte Schemata verwendet. Diese kann man als Metamodell verstehen. Sie legen die zulässigen Informationselemente sowie Regeln zu deren Verwendung innerhalb der Datenstruktur fest. XML-Dateien mit konkretem Inhalt sind dann gleichbedeutend mit Instanzen zu dem Metamodell. Es kann eine automatisierte Validierung einer XML-Datei zu dem dazugehörigen Schema durchgeführt werden.

```
<xsd:element name="Zwischenprodukt">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:choice>
                <xsd:element name="system" type="systemType"/>
                <xsd:element name="module" type="moduleType"/>
                <xsd:element name="classElement" type="classElementType"/>
                <xsd:element name="part" type="partType"/>
                <xsd:element name="statemachine" type="statemachineType"/>
            </xsd:choice>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Bild 4.10: Die Grundstruktur des Zwischenproduktes als XML-Schema

Werkzeuge, die einen Austausch per XML unterstützen, stellen die zugehörigen Schemata typischerweise zur Verfügung. So ist es auch bei den für den vorliegenden Ansatz Relevanten. Für das Zwischenprodukt ist entsprechend ein neues Schema zu erstellen.

Die oberste Ebene dieses Schemas zeigt Bild 4.10. Es ist ersichtlich, dass als Wurzelelement des Zwischenprodukts alle bekannten Strukturelemente der ABSOFA zulässig sind. Auf die hohe Ähnlichkeit der ABSOFA zum Zwischenprodukt wurde bereits hingewiesen. Der Regelfall ist, dass eine Software-Architektur ein „system“-Element als Wurzel erhält. Dies ermöglicht einen beliebigen weiteren Aufbau nach unten mit weiteren systems oder anderen Bausteinen. Jedes andere Strukturelement als Wurzel hat zur Folge, dass die Architektur nur aus diesem einen Element bestehen kann. Dies kann in Sonderfällen für isolierte Betrachtungen einzelner Elemente dienen, zu erwarten ist jedoch die Betrachtung von Software-Architekturen, die aus mehreren Bausteinen bestehen.

4.5.3 Anwendung innerhalb der Prozessschritte

Im vorliegenden Abschnitt soll dargestellt werden, wie das zentrale Datenmodell (Zwischenprodukt) als entscheidendes Hilfsmittel konkret innerhalb des Vorgehensmodells verwendet wird, um die unterschiedlichen vorgeschlagenen beziehungsweise angepassten Prozessschritte zu realisieren. Da das Vorgehensmodell zur Entwicklung flexibel einsetzbar sein soll – in Bild 3.4 auf Seite 44 wurde das im Rahmen der Entwicklung neuer Systeme wichtige Top-Down-Vorgehen visualisiert, je nach Zielsetzung sind jedoch auch andere Ausprägungen denkbar – werden eine Vielzahl an Interaktionen mit dem Zwischenprodukt notwendig. Anpassungen im Vorgehensmodell für den spezifischen Projektbedarf werden auch als „Tailoring“ bezeichnet. In Bild 4.8 auf Seite 65 sind die unterschiedlichen Anwendungsfälle des Zwischenprodukts bereits in Gänze aufgezeigt. Im Vorfeld wurden über das in Bild A.1 in Anhang A.1 dargestellte Schema eine systematische Erfassung, Vervollständigung und Kategorisierung dieser Anwendungsfälle erreicht. Es existieren somit vier Hauptkategorien, die alle für das Vorgehensmodell relevanten Interaktionen mit dem Zwischenprodukt abdecken. Auf der jeweils untersten Ebene der Nummerierung in den jeweiligen Hauptkategorien finden sich konkret umsetzbare Anwendungsfälle. Auf die dort aufgeführte, eindeutige Nummerierung der einzelnen Anwendungsfälle wird im weiteren Verlauf der Arbeit Bezug genommen.

Wie eingangs von Abschnitt 4.5 bereits erwähnt wurde, sollen alle Anwendungsfälle des Zwischenprodukts in einem „universellen Ansatz zur Modelltransformation und -analyse“ realisiert werden. Das zuvor erwähnte Schema der Anwendungsfälle legt in Kombination mit den grundsätzlichen Anforderungen (Tabelle A.1) den Rahmen und die Zielsetzung für diesen Ansatz in Gänze fest. Zur Findung eines geeigneten Ansatzes wurden die folgenden Technologien und Werkzeuge auf Ihre Eignung verglichen:

- M2M (Firma Aquintos GmbH, kommerzielles Werkzeug) [71],
- Fujaba (Universität Paderborn, Open Source) [118],
- XSLT (W3C – World Wide Web Consortium, Open Source) [123],
- MTRANS (Universität Nantes, Open Source) [91] und
- JAXB (Firma Sun Microsystems, Open Source) [113].

Ausgewählt wurde – ohne den Auswahlprozess an dieser Stelle näher darzustellen – der

JAXB-Ansatz. Im Rahmen der Arbeit stellt dieser Teilaspekt ein Hilfsmittel dar und ist nicht Gegenstand methodischer Detailuntersuchung oder -forschung. Es war vorrangig wichtig, einen Ansatz zu identifizieren, der geeignet ist, im Kontext des methodischen Rahmens von Vorgehensmodell und Prozessschritten eingegliedert und durch konkrete Implementierung von Regeln und Algorithmen als Werkzeug nutzbar gemacht zu werden. Das Grundprinzip von JAXB und wie es für die konkrete Aufgabenstellung angewendet wurde, wird im Folgenden sowie in Bild 4.11 beschrieben.

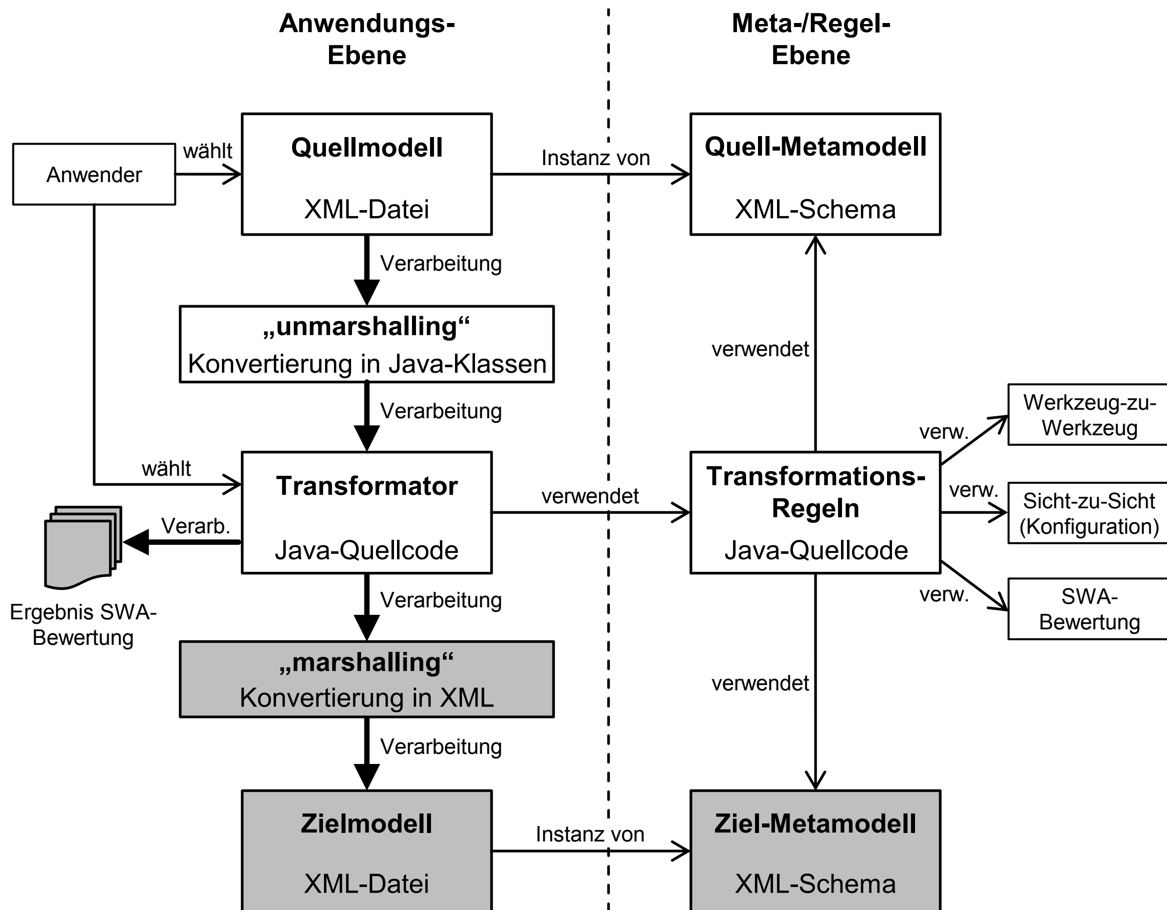

Bild 4.11: Der Ablauf des JAXB-Transformations- und Analyseansatzes gesamthaft

Man kann grundsätzlich die Meta-/Regel-Ebene (Bild 4.11 rechte Seite) und die Anwendungsebene (Bild 4.11 linke Seite) unterscheiden. In der Erstgenannten werden über XML-Schemata die zulässige Struktur von Quell- und Zieldokumenten formal und eindeutig vorgegeben. Darüber hinaus werden Regeln zur Analyse beziehungsweise Manipulation in der späteren Anwendung erfasst. In der Abbildung ist ersichtlich, dass je nach Anwendungsfall über den Block „Transformations-Regeln“ das richtige Set an Regeln ausgewählt werden muss. Das Grundprinzip ist für alle identisch.

Auf der Anwendungs-Ebene wird mit konkreten Artefakten gearbeitet. Der Anwender wählt ein Quellmodell in Form einer XML-Datei aus, welche konform zum definierten Schema (=Metamodell) sein muss und daher eine Instanz dessen darstellt. Durch JAXB wird der nun folgende, so genannte „unmarshalling“-Vorgang bereitgestellt. Dabei wird das Quellartefakt in eine Java-spezifische Struktur gemäß Schema überführt, es werden Java-Beans sowie Operationen zum Zugriff (get, set) auf diese Elemente erzeugt. Der Transformator kann die konkreten Regeln anwenden und führt dabei entweder eine Analyse (Hauptkategorie 3 in Bild A.1, entspricht der Software-Architekturbewertung) oder eine Manipulation (Hauptkategorien 1,2,4 in Bild A.1) der Struktur durch. Nur nach einer Manipulation werden die weiteren, grau ausgefüllten Schritte notwendig. Über den „marshalling“-Vorgang wird die neue Struktur wieder in ein XML-Dokument gemäß Ziel-XML-Schema zurückgeschrieben. Im Falle der Überführung in eine andere Sicht sind Quell- und Ziel-Metamodell identisch, es findet lediglich ein Ändern der Datenelemente statt. In allen anderen Fällen wird in eine durch ein anderes XML-Schema (zum Beispiel konform zu Ziel-Werkzeug) vorgegebene Datenstruktur transformiert. Das Ergebnis einer Analyse wird in ein separates Ergebnis-Artefakt geschrieben. Da dieses nur in diesem Anwendungsfall erzeugt wird, ist es ebenfalls als optional und damit grau ausgefüllt dargestellt.

Der implementierte Software-Prototyp

Um den Ansatz zur „Modelltransformation und -analyse“ zu evaluieren, wurde ein Software-Prototyp entwickelt, in welchem ausgewählte Anwendungsfälle aus der diskutierten Kategorisierung implementiert wurden. Es wurde dabei, wie in Abschnitt 4.3.2 bereits erwähnt, die Entwicklungsumgebung Eclipse for Java eingesetzt. Zur Bedienung durch den Anwender wurden insgesamt drei grafische Oberflächen (Graphical User Interfaces (GUIs)) entworfen, jede repräsentiert eine der folgenden Anwendungsbereiche:

- Objektive Software-Architekturbewertung (Kategorie 3.1 in Bild A.1),
- Überführung zwischen Sichten (Kategorie 2.2 in Bild A.1) und
- Modelltransformation zwischen Werkzeugen (Kategorien 1.2, 2.3, 4.1 in Bild A.1).

Der richtige Satz an Regeln wird somit einerseits aufgrund der durch den Anwender gewählten GUI, andererseits aufgrund der darin vorgenommenen Detail-Einstellung ausgewählt. So wird beispielsweise in der GUI für die Modelltransformation zwischen Werkzeugen die Transformationsrichtung (zum Beispiel „ASCET → Zwischenprodukt“) vom Anwender ausgewählt. Sobald alle Optionen der Konfiguration vom Anwender getroffen worden sind, können Quell- und gegebenenfalls Zieldatei ausgewählt werden. Im Falle der Software-Architekturbewertung werden als Sonderfall bis zu zwei Quelldateien ausgewählt, da eine vergleichende Bewertung durchgeführt werden kann. Die drei GUIs sind optisch bewusst ähnlich aufgebaut, um dem Anwender zu helfen, sich schnell zurecht zu finden. Stellvertretend für alle Oberflächen kann die GUI für den letztgenannten Fall

der Software-Architekturbewertung aus Ahrens et al., 2010[4] (im dortigen Anhang A, Seite 17) entnommen werden.

Zum gegenwärtigen Zeitpunkt wurden im Prototypen die konkreten Anwendungsfälle vollständig implementiert, die im bereits bekannten Bild A.1 durch graues Ausfüllen kenntlich gemacht sind. Zur Bestätigung der Wirksamkeit und Anwendbarkeit des gesamtheitlichen Ansatzes war es notwendig, aus jeder Hauptkategorie einen konkreten Anwendungsfall zu implementieren und an Beispielen zu erproben. Für die Umsetzung der Software-Architekturmetriken wurde dies in Abschnitt 4.3.2 stellvertretend bereits dargestellt. Die übrigen Anwendungsfälle werden mit vergleichbarem Vorgehen individuell auf ihre Wirksamkeit und Korrektheit überprüft. Darüber hinaus ist für den weiteren Verlauf der Arbeit ab Kapitel 5 insbesondere die Umsetzung der Anwendungsfälle 1.2.2, 3.1 und 4.1.2 aus Bild A.1 essentiell notwendig.

Beispielhafte Ausprägungen des Vorgehensmodells und die damit verbundene Anwendung des Zwischenprodukts

Abschließend für das vorliegende Kapitel sollen zwei mögliche Ausprägungen und damit Abläufe des Vorgehensmodells mit Blick auf die Anwendungsfälle des Zwischenprodukts diskutiert werden. Dies zeigt die Vielseitigkeit des Ansatzes und soll das Verständnis bezüglich des Zusammenspiels von Zwischenprodukt und den Prozessschritten vertiefen.

Exemplarisch gezeigt ist ein so genanntes Top-Down-Vorgehen, welches für die Entwicklung neuer (Software-)Systeme eingesetzt wird (Bild 4.12 links). Als Gegenpol ist ein Bottom-Up-Vorgehen dargestellt, bei welchem ein vorhandenes System (oft auch als „Legacy-System" bezeichnet) weiterentwickelt wird. Wichtig ist hierbei zu verinnerlichen, dass das dargestellte Vorgehen jeweils nur einer möglichen Variante des Vorgehensmodells für die beiden Grundtypen der Entwicklung entspricht. Je nach konkreter Problemstellung und Vorlieben des Anwenders sind weitere Abwandlungen möglich, so auch Hybrid-Vorgehen mit Top-Down- und Bottom-Up-Anteilen.

In einem Top-Down-Vorgehen ist als zentraler Bestandteil der iterative Entwurf der Abstrakten Software-Architektur über die Modellierung mit ABSOFA fest verankert, der durch die objektive Software-Architekturbewertung unterstützt wird. Über das automatisierte Erstellen des Zwischenprodukts (Transformation nach Kategorie 1.2.1) werden die nächsten Schritte eingeleitet. Diese können je nach Problemstellung variieren. Das Hinzufügen von Varianteninformationen im Architektur-Entwurf und damit auch das Auflösen dieser (Transformation nach Kategorie 2.2.1) sind optional, ebenso wie der Zwischenschritt über die Plattformabhängige Sicht auf die Software-Architektur (Transformation nach Kategorie 2.2.2). Abschließend liegt jedoch immer eine Transformation in ein Werkzeug der Software-Umsetzung, hier ASCET (Transformation nach Kategorie 4.1.2), vor.

Im Bottom-Up-Vorgehen kann einerseits eine explizite Software-Architektur in der ABSOFA vorliegen. In diesem Fall ähnelt der Ablauf stark dem zuvor dargestellten Top-Down-Vorgehen. In diesem Beispiel wird vom anderen Fall ausgegangen, dass kei-

ne explizite Software-Architektur des bestehenden Systems vorliegt, sondern über eine Transformation aus dem Werkzeug der Umsetzung erzeugt werden muss. Dieses Werkzeug ist im vorliegenden Beispiel ebenfalls ASCET, so dass eine Transformation nach Kategorie 1.2.2 durchzuführen ist. Genauso wichtig wie im Top-Down-Vorgehen ist auch im Bottom-Up-Vorgehen die Verwendung der Architekturmetriken. Ob der sich dadurch ergebende iterative Entwurf weiterhin in ASCET erfolgt wie dargestellt, oder ob zunächst eine Transformation vom Zwischenprodukt in die ABSOFA gewünscht wird, obliegt der Entscheidung des Anwenders. In diesem Fall soll nur ein einfaches Refactoring durchgeführt werden, so dass keine der weiterführenden Schritte notwendig sind. Nach Überarbeitung der Software-Architektur kann direkt zur Software-Umsetzung übergegangen werden (Transformation nach Kategorie 4.1.2).

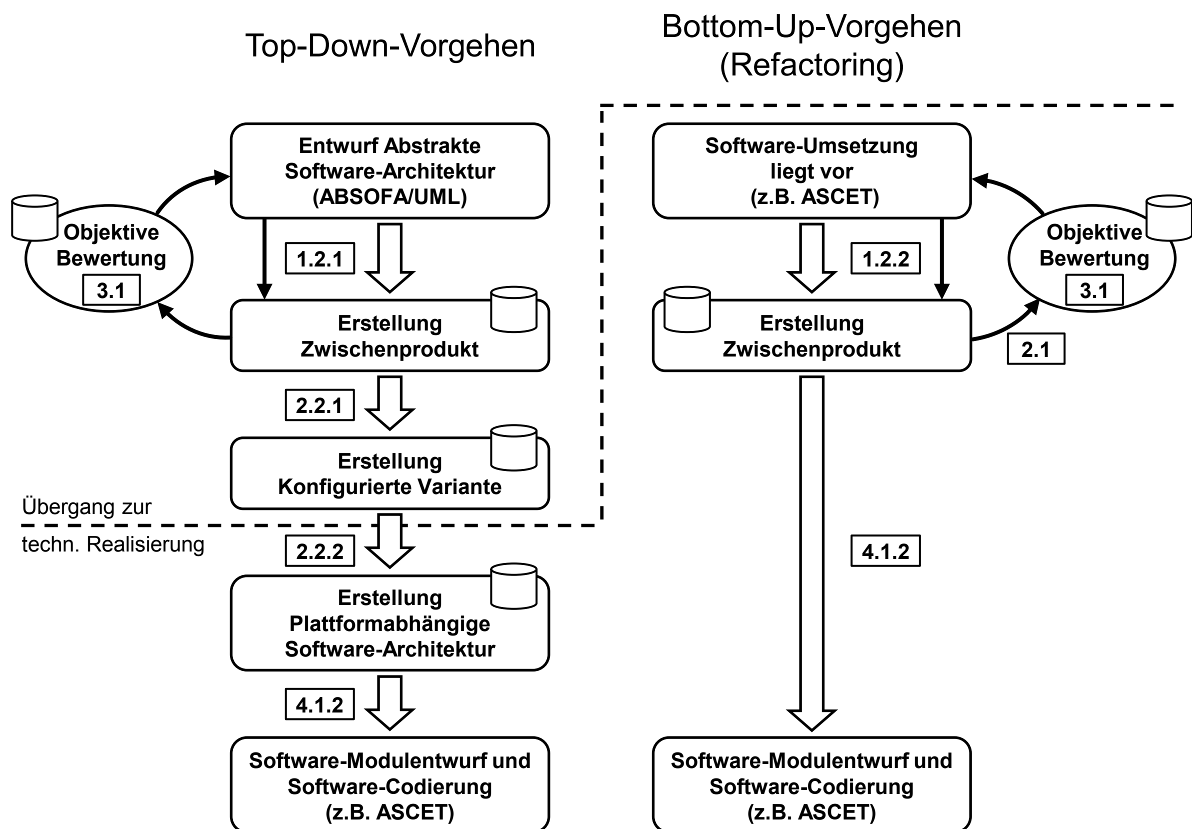

Bild 4.12: Anwendung des Zwischenprodukts in verschiedenen Entwicklungs-Vorgehen

5 Strukturierung von Software-Architekturen für Fahrerassistenzsysteme

Die bisherigen Überlegungen zur Strukturierung von Funktions- und Software-Architekturen, dem Vorgehensmodell zur Entwicklung sowie insbesondere der Modellierung und Weiterverarbeitung von Software-Architekturen wurden bewusst allgemein anwendbar für die gesamte Automobil-Domäne angestellt. Am konkreten Beispiel von Fahrerassistenzsystemen sollen diese nun in der Praxis angewendet werden. Ein Schwerpunkt ist dabei zu erarbeiten, wie Software-Architekturen in diesem Bereich zu gestalten sind, um die speziell geltenden Ziele, Schwierigkeiten und Rahmenbedingungen zu adressieren. Für die Entwicklung heutiger und künftiger Fahrerassistenzsysteme wird dabei eine wichtige Arbeitsgrundlage geschaffen. Der andere Schwerpunkt ist die Evaluation der Wirksamkeit und Praxis-Anwendbarkeit der Ansätze.

5.1 Spezifische Zielsetzung und Randbedingungen

Im vorliegenden Kapitel sollen zunächst die Besonderheiten von Fahrerassistenzsystemen – primär bezogen auf Software, Software-Entwicklung und Software-Architektur – herausgestellt werden. Damit wird verdeutlicht, warum es für diese Art von Systemen notwendig und lohnenswert ist, spezifische Überlegungen anzustrengen.

Im Anschluss daran wird sich intensiv mit der Software-Architektur von aktuellen in der Serienentwicklung und -produktion befindlichen Fahrerassistenzsystemen der Längsführung auseinandergesetzt. Wie bereits eingangs in Abschnitt 1.1 erwähnt wurde, hat diese nach evolutionärer Weiterentwicklung über Jahre hinweg inzwischen eine Struktur erreicht, mit der nicht länger effizient weitergearbeitet werden kann. Es liegt somit ein realer Bedarf der Industrie nach einer optimierten Struktur vor. Die bisherige Ist-Struktur wird ausführlich untersucht und dabei deren Schwachstellen identifiziert, die gleichzeitig als Negativ-Beispiele für Aspekte, die zu vermeiden sind, dienen können. Nach einer Aufstellung der zu verfolgenden Ziele der angestrebten Neustrukturierung, kann mit dieser begonnen werden. Im selben Zuge soll ausführlich und konkret auf die funktionalen und insbesondere die nicht-funktionalen Einflüsse auf Software-Architektur am Beispiel von Fahrerassistenzsystemen eingegangen werden, da diese im bisherigen Verlauf der Arbeit zwar mehrfach im Hinblick auf ihre Bedeutung erwähnt aber noch nicht ausreichend im Detail betrachtet wurden. Über die Neustrukturierung am konkreten Beispiel hinaus sollen im selben Vorgang zusätzlich auch allgemeingültige Richtlinien und Muster für die Strukturierung von Software-Architekturen von Fahrerassistenzsys-

temen abgeleitet werden. Die Ergebnisse werden im Kontext aktueller und künftiger Fahrerassistenzsysteme diskutiert und dabei aufgezeigt, weswegen sie dem Anspruch der Allgemeingültigkeit innerhalb des Fachbereichs genügen. Das Ergebnis dieser Überlegungen kann und soll somit in künftigen Entwicklungen von Fahrerassistenzsystemen immer wieder als generische Arbeitsgrundlage dienen.

Durch das oben genannte Vorgehen werden gleichzeitig die wesentlichen Ansätze der vorliegenden Arbeit am realen Beispiel evaluiert und damit deren Praxistauglichkeit bestätigt. Konkret werden die folgenden Aspekte beziehungsweise Ansätze im Rahmen dieser Evaluierung betrachtet:

- Das angepasste Vorgehensmodell zur Entwicklung,
- Unterschiedliche Anwendungen des Vorgehensmodells nach Vorgabe des Anwenders (Tailoring). Hier: Anwendung im Bottom-Up-Vorgehen,
- Umfang und Anwendung der ABSOFA sowie des Datenmodells,
- Automatisierte Transformationen: Werkzeug → Datenmodell und
- Automatisierte Anwendung der objektiven Software-Architekturmetriken in einem iterativ inkrementellen Vorgehen.

Fahrerassistenzsysteme nehmen aufgrund ihrer spezifischen Eigenschaften und den im Rahmen ihrer Entwicklung zu betrachtenden Randbedingungen eine Sonderstellung innerhalb der mechatronischen Systeme im Automobil ein. Das soll im folgenden Abschnitt verdeutlicht werden, da so das Verständnis für die später folgende Neustrukturierung und die Ableitung allgemeingültiger Richtlinien und Muster für Software-Architekturen von Fahrerassistenzsystemen erhöht wird.

Auf einige eher allgemein wissenswerte Aspekte zu Fahrerassistenzsystemen wurde in den Abschnitten 1.1 und 1.2 bereits eingegangen. Diese bleiben weiterhin gültig, werden an dieser Stelle jedoch nicht wiederholt. Viel mehr wird, wie weiter oben bereits erwähnt, auf Besonderheiten dieser Systeme in Bezug auf Software, Software-Entwicklung und Software-Architektur fokussiert.

Technische Eigenarten

In diesem Abschnitt wird auf die besonderen Eigenschaften bezogen auf Technik und Auslegung der Systeme eingegangen.

- **Komplexität/Umfang:** Fahrerassistenzsysteme unterhalten äußerst viele Kommunikationsbeziehungen (Signale, Informationsfluss). Dies gilt sowohl für jedes einzelne System für sich als auch bezüglich der Kommunikation zwischen Systemen untereinander. Darüber hinaus sind die Modelle (oder der Quell-Code) für Software in der Regel groß, da die Logik sehr umfangreich ist. Beides führt dazu dass die Software(-Architektur) schnell unübersichtlich wird.
- **Arbeiten mit unterschiedlichen physikalischen Größen:** Es sind häufig Signalaufbereitungen auf Eingangs- und Ausgangs-Seite notwendig, welche die Logik

des Fahrerassistenzsystems mit dem Rest des Fahrzeugs verbinden. Beispiele sind die Umrechnung zwischen unterschiedlichen Einheiten (z.B. Nicht-SI-Einheit in SI-Einheit) oder zwischen unterschiedlichen physikalischen Größen (z.B. Beschleunigung in Radmoment). Da aktive Fahrerassistenzsysteme in die Fahrzeugführung eingreifen und somit der Fahrer eine direkte Rückkopplung hat, bestehen hohe Anforderungen an das Verwenden von richtigen Größen und die korrekte und komfortable Errechnung, Umrechnung und Rundung von Ist- und Sollwerten.

- **Sicherheitsrelevanz:** Fast immer sind bestimmte Umfänge der Wirkkette – und damit auch der Software – mit Sicherheitszielen belegt. Es bestehen dann bestimmte Anforderungen an Dokumentation, Erstellung und Ausführung der Software. So müssen zum Beispiel explizite Sicherheitsfunktionen zur Überwachung hinzugefügt oder bestimmte Umfänge redundant ausgeführt werden.
- **Variantenvielfalt:** Aus einem Gesamtansatz, das bedeutet auch einer einzigen Software-Architektur, ist eine Vielzahl an unterschiedlichen Systemausprägungen (z.B. ACC mit Kamera, ACC mit Radar), Funktionsausprägungen (z.B. ACC 30+, ACC Stop&Go) sowie Kombinationen und Ausbaustufen (welche Funktionen können durch den Fahrer einzeln oder in bestimmten Paketierungen bestellt werden) zu bedienen. Diese Varianz kann über unterschiedliche Baureihen oder Derivate ebenfalls differieren. Auch dies muss der Gesamtansatz abdecken. Niedrige Ausbaustufen, insbesondere in unteren Fahrzeugklassen, haben dabei besonders hohe Anforderungen an eine ressourceneffiziente Umsetzung und damit an die Skalierbarkeit des Ansatzes.

Verteilte und wechselnde Geschäftsmodelle

Aufgrund der hohen Komplexität der Wirkkette und damit der Vielzahl an Komponenten und Teilfunktionen, herrschen zur Realisierung eines Fahrerassistenzsystems fast immer ein verteiltes, gemischtes Geschäftsmodell und verteilte Verantwortlichkeiten vor. Bestimmte Umfänge werden von Lieferanten auf Basis einer Spezifikation des Fahrzeugherstellers umgesetzt, andere Umfänge werden vom Fahrzeughersteller selbst entwickelt. Diese Geschäftsmodelle können sich zwischen Derivaten unterscheiden oder sich während der Laufzeit beziehungsweise zwischen Folgeprojekten ändern. Es ist ein hohes Maß an Flexibilität vonnöten. Je nach (vom Fahrzeughersteller für sich definierter) Bedeutung einzelner Software-Umfänge werden konkrete Vorgaben für die Software-Architektur an den Lieferanten oder den Fachbereich im Hause gemacht.

Dynamisches Umfeld

Der Fachbereich Fahrerassistenz ist eine sehr junge Disziplin innerhalb der Automobil-Domäne. Das bedeutet, dass zum gegenwärtigen Zeitpunkt eine sehr hohe Dynamik im Umfeld (Wettbewerb der Fahrzeughersteller und Zulieferer sowie Einfluss durch die Wissenschaft) herrscht. Änderungen und Erweiterungen an bestehenden Produkten sind

daher auf Sicht von mehreren Jahren kaum im Detail vorhersagbar, notwendige (zum Teil umfangreiche) Änderungen sind aber in jedem Projektumfang sehr wahrscheinlich. Diese müssen durch eine änder- und erweiterbare Software-Architektur vorbereitet werden.

Fazit

Als Fazit kann festgehalten werden, dass bei Fahrerassistenzsystemen etliche in sich bereits komplexe Rand- und Rahmenbedingungen gleichzeitig und konkurrierend zueinander auftreten. Die sich daraus ergebende Gesamtkomplexität macht diesen Fachbereich derzeit zum möglicherweise anspruchsvollsten Themengebiet überhaupt in der Automobil-Domäne. Da auch in Zukunft mit einer weiteren Zunahme an Aktivitäten und Innovationen in diesem Bereich gerechnet werden kann, ist eine ausdrückliche und spezifische Betrachtung im Rahmen der Entwicklung wie in dieser Arbeit zielführend.

5.2 Vorbereitung der Neustrukturierung der FAS-Längsführungs-Architektur

Die aktuelle Software-Architektur von Fahrerassistenzsystemen der Längsführung (kurz: FAS-Längsführung) versucht über einen gesamthaften Ansatz alle notwendigen Konfigurationen abzudecken. Das ist naheliegend, da aufgrund der hohen Synergien (identische Regelungsansätze, Verwendung identischer oder sehr ähnlicher Schnittstellen zu Sensoren und Aktoren) und notwendiger Koordination und Wechselwirkungen zwischen unterschiedlichen Systemen (Aktivierung, Deaktivierung und Übergänge sowie Zugriff auf Aktoren) eine Trennung auf eigenständige Wirkketten und Softwaresysteme kaum möglich und auch nicht zielführend ist. Der Ansatz funktioniert grundsätzlich, das bedeutet es muss nur diese eine Gesamt-Architektur gepflegt werden, die Skalierbarkeit in beide Richtungen ist jedoch unzureichend. Das bedeutet, dass weder Erweiterungen um neue Teilfunktionalität bestehender Kundenfunktionen oder gänzlich neue Kundenfunktionen („Skalierbarkeit nach oben“) noch das Herauslösen von Teilumfängen für geringe Ausstattungen („Skalierbarkeit nach unten“) effizient möglich sind. Unter effizient ist dabei sowohl der einzusetzende Arbeitsaufwand als auch die benötigen Hardware-Ressourcen bezogen auf die gewünschte Funktionalität zu verstehen.

Ein konkretes Beispiel dafür ist die Darstellung der Geschwindigkeitsregelung mit Bremseingriff, welche bei BMW auch als Dynamic Cruise Control (DCC) bezeichnet wird, als stand-alone-Lösung. Sie kann aktuell nur mit einem – an der Funktionalität gemessenen – großen Ressourcenaufwand als Teil des Gesamtsystems für den Kunden realisiert werden. Da jedoch der Anwendungsfall inzwischen eine hohe Ausstattungsrate besitzt, ist die zuvor genannte Tatsache aus Sicht des Unternehmens besonders bedeutsam. Die Verbesserung der Software-Architektur im Hinblick auf diesen Sachverhalt ist daher eines der Hauptziele der Neustrukturierung.

Der Grund der suboptimalen Software-Architektur ist die sukzessive Erweiterung des Gesamtsystems und damit stetige Zunahme der Komplexität über Jahre hinweg. Hierbei war die Entwicklung stark getrieben durch die funktionalen Ziele in Verbindung mit Zeit- und Kostendruck der Umsetzung. Nicht-funktionale Eigenschaften wurden nicht in gleichem Maße berücksichtigt. Das Resultat ist eine Struktur, die unnötige Abhängigkeiten und falsche Zuordnungen von Funktionalitäten zu Bausteinen sowie keine übergreifende Leitidee in Form eines Architekturstils aufweist. Nicht benötigte Software-Umfänge können nicht unter angemessenem Aufwand herausgelöst werden und sind lediglich über Parametrierung unwirksam gemacht. Je mehr mit dieser Struktur künftig weiter gearbeitet würde, desto aufwändiger werden zusätzliche Erweiterungen oder Änderungen. Es ist offensichtlich, dass mit diesem Zustand die aktuellen und besonders die künftigen Herausforderungen in der Fahrerassistenz nicht mehr zu bewältigen sind.

Bevor noch näher auf die Struktur eingegangen wird, soll kurz erläutert werden, welche Fahrerassistenzsysteme zum hier betrachteten Umfang der FAS-Längsführung gehören. Die folgenden Systeme sind der Kernumfang, der vollständig in Eigenentwicklung erfolgt:

- Adaptive Cruise Control (ACC 30+),
- Adaptive Cruise Control mit Stop&Go-Funktion (ACC Stop&Go),
- Geschwindigkeitsregelung mit Bremseingriff (DCC),
- Intelligent Brake mit Warn- und Anbremsfunktion (iBrake) und
- Speed Limitation Device (SLD)

Zusätzlich werden die folgenden, an anderer Stelle durch Lieferanten-Software realisierten, Kundenfunktionen bezüglich Koordination im Gesamtsystem mitberücksichtigt:

- Hill Descent Control (HDC) und
- Parkmanöver-Assistent (PMA) mit Querführung.

In Bild E.1 in Anhang E.1 wird ein Ordnungsschema für Fahrerassistenzsysteme vorgeschlagen. Dieses enthält die soeben genannten Systeme, welche zur schnellen Identifizierung dicker umrandet sind, sowie zeigt bereits die nächste Generation an Fahrerassistenzsystemen mit Längs- und Querführung auf. Es wird unterschieden bezüglich Art der Fahrzeugführung (keine, Längsführung, Querführung, Längs- und Querführung) sowie nach generalisierten Oberklassen von Funktionen und konkreten Kundenfunktionen als Spezialisierung einer Oberklasse analog dem Prinzip der Vererbung. Der Automatisierungsgrad spielt hier bewusst keine Rolle. Somit können auch künftige Fahrerassistenzsysteme bis hin zum hoch- und vollautomatisierten Fahren hier einsortiert werden.

5.2.1 Analyse der aktuellen Architektur

Die zuvor genannte unzureichende Skalierbarkeit der Software-Architektur ist der Hauptgrund für die folgenden Aktivitäten. Trotzdem wird eine umfassende Analyse der Schwächen der Ist-Struktur methodisch strukturiert durchgeführt. An dieser Stelle soll das methodische Vorgehen, nicht aber die Arbeitsergebnisse im Detail vorgestellt werden.

Hauptanliegen der Analyse ist es alle oben genannten, in der Gesamt-Architektur integrierten, Fahrerassistenzsysteme einzeln und im Zusammenspiel vollständig funktional zu verstehen. Nur mit dieser Basis ist es möglich, im Anschluss auch die Gesamtheit aller nicht-funktionalen Anforderungen so zu berücksichtigen, dass die Funktion im Sinne des Kundenerlebnisses in Gänze unverändert erhalten bleibt. Zur Ausgangs-Situation liegt lediglich eine Sammlung der funktionalen Anforderungen als Spezifikation vor.

Wie bereits mehrfach erwähnt, wurden die nicht-funktionalen Einflussgrößen bisher noch nicht explizit betrachtet. Genau das soll unter anderem durch die anschließende Neustrukturierung erfolgen. Die oben erwähnte Spezifikation wurde funktional strukturiert („Funktions-Architektur“) und in der exakt selben Struktur in Software überführt, ein expliziter Software-Architekturentwurf fand nicht statt. Dies wurde bereits in Kapitel 3 als Schwachstelle des Vorgehensmodells identifiziert und hier in der praktischen Anwendung bestätigt. Auch dies soll durch die Neustrukturierung nachgeholt und etabliert werden. Neben der textbasierten Spezifikation, welche im Werkzeug DOORS [52] vorliegt, wurde ebenfalls die Umsetzung in ASCET – und damit die dort implizit geschaffene Software-Architektur – als Quelle zur Analyse betrachtet. Die folgenden Arbeitsergebnisse werden während der Analyse produziert:

- **System-Steckbriefe:** Nach einer festen, selbst definierten Vorlage werden die einzelnen relevanten Fahrerassistenzsysteme in eigenen Worten beschrieben und dabei abstrahiert wiedergegeben. So kann einerseits das korrekte Verständnis überprüft und sich andererseits von den bereits existierenden Vorgaben der Umsetzung gelöst werden. Diese können für die Projektbeteiligten zu „Fesseln“ werden, wenn sich diese als gegeben und damit unabänderlich im Kopf festsetzen. Ein bewusster Schritt zurück kann helfen, die Struktur ergebnisoffen zu hinterfragen.
- **Detaillierter Signalflussplan:** Da kein expliziter Software-Architekturentwurf vorliegt und das Werkzeug ASCET auch für die implizit entstandene Architektur keine geeignete Visualisierung anbietet, wird diese manuell in einem grafischen Werkzeug aus den angegebenen Quellen erzeugt. Alle Module und deren Wechselwirkungen in Form von Signalflüssen sind dargestellt. So können alle Abhängigkeiten schnell überblickt und mit denen vom funktionalen Mindestbedarf verglichen werden. Dies erleichtert die Suche nach unnötigen Abhängigkeiten.
- **Anwendung der objektiven Software-Architekturmetriken:** Die Software-Architekturmetriken werden vollautomatisiert am durch eine Modelltransformation aus ASCET erzeugten Zwischenprodukt ausgeführt. Da die Metrikergebnisse als Absolutwerte wenig Aussagekraft besitzen, ist dieses Teilergebnis vor allem als Absprungbasis und Vergleichswert für die Iterationen im Rahmen der Neustrukturierung nützlich.
- **Liste struktureller und werkzeugbedingter Mängel:** Die mit Hilfe der Analyse identifizierten Mängel werden für alle Bausteine der Software-(Architektur) einzeln sowie für übergreifend geltende Aspekte gesamthaft textuell erfasst.

Durch das so aufgebaute, umfassende Verständnis von Soll-Funktion und Umsetzung in

Software können die eigentlichen funktionalen Anforderungen mit der Ist-Struktur beziehungsweise Ist-Umsetzung verglichen werden. Die Anwendung von allgemein bekannten Richtlinien und Mustern aus dem Stand der Technik kann auf diese Art qualitativ, die Anwendung der Vorgaben der entwickelten Software-Architekturmetriken sogar quantitativ ermittelt werden. Die dabei identifizierten, wichtigsten Erkenntnisse zu den Schwachstellen sind in kompakter Form in der Tabelle E.1 in Anhang E.1 aufgeführt.

Die Ziele der Neustrukturierung

Nachdem zuvor die Schwachstellen und globale Hauptzielsetzung, die mit der Neustrukturierung verfolgt werden sollen, dargelegt worden sind, werden hier die erweiterten Ziele in gebündelter Form genannt:

- Alle funktionalen Anforderungen werden weiterhin erfüllt, das für den Kunden erlebbare Verhalten ändert sich nicht.
- Es muss ein expliziter Software-Architekturentwurf unabhängig von der und zusätzlich zur Funktions-Architektur erfolgen und so im gelebten Vorgehensmodell etabliert werden.
- Die nicht-funktionalen Einflüsse und Anforderungen auf das konkrete Beispiel müssen umfassend ermittelt und im Entwurfsprozess berücksichtigt werden. Die neue Struktur wird somit auf diese Vielzahl an notwendigen Eigenschaften optimiert und nicht nur auf eine singuläre Eigenschaft wie die schon erwähnte fehlende Skalierbarkeit.
- Es muss ein klarer Architekturstil herausgearbeitet werden, der nicht nur für die heutigen Fahrerassistenzsysteme der Längsführung sondern auch für zukünftige Generationen, insbesondere mit Längs- und Querführung (siehe Bild E.1), als generisches Muster für aktive Fahrerassistenzsysteme verwendet werden kann. Das Konzept des Architekturstils wird in Abschnitt 5.3.1 vertieft.

5.2.2 Festlegung des methodischen Vorgehens

Für den iterativen Entwurf von Software-Architekturen als Teilumfang des bereits vorgestellten Vorgehensmodells wird ein universeller Ansatz nach Bild 5.1 vorgeschlagen. Die Universalität begründet sich darin, dass der Ansatz für beliebige Projektumfänge sowohl im Top-Down- (entspricht einer Neuentwicklung) als auch im Bottom-Up-Vorgehen (entspricht der Neustrukturierung einer bestehenden Architektur) eingesetzt werden kann. Durch Tailoring kann das Vorgehen an den spezifischen Bedarf angepasst werden.

Vorbereitende Schritte

Vor Beginn der eigentlichen Entwurfstätigkeit sind einige Aktivitäten zur Vorbereitung durchzuführen. Dazu gehört die vollständige Erfassung aller funktionalen und nicht-funktionalen Einflüsse, die zu berücksichtigen sind. Diese werden später in diesem Ab-

schnitt noch ausführlich betrachtet. Des Weiteren sollte ein Erstentwurf der Software-Architektur vorliegen. Dieser kann im Falle des Bottom-Up-Vorgehens direkt von der bis dato vorliegenden (Legacy-)Software-Architektur übernommen oder im Falle Top-Down aus einer (manuellen oder automatischen) Überführung der Funktions-Architektur gewonnen werden. Der Zweck dieses Erstentwurfs ist es eine Ausgangsbasis für die zielgerichtete Entwurfsarbeit zu haben, die analysiert (siehe Abschnitt 5.2.1) und im Anschluss systematisch weiter entwickelt werden kann.

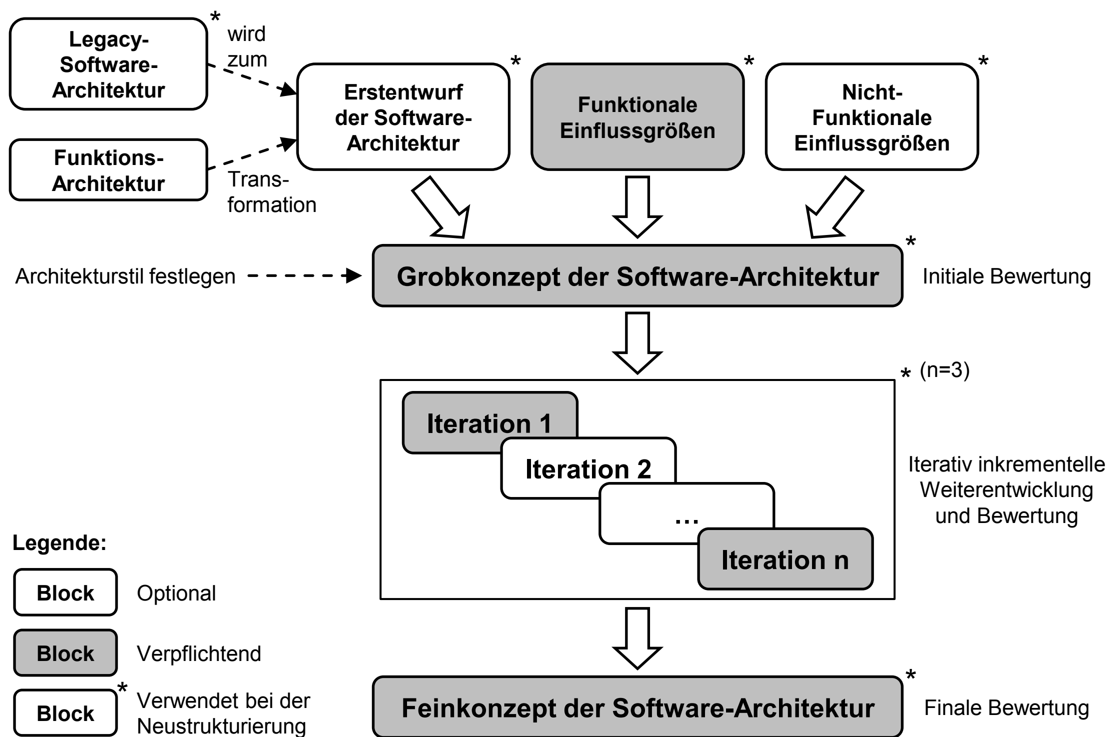

Bild 5.1: Vorgehen zum iterativen Entwurf von Software-Architekturen

Grobkonzept

Für das so genannte Grobkonzept wird ein Architekturstil benötigt, der für die gesamte Architektur eine übergeordnete Struktur vorgibt. Diese richtet sich stark nach der betrachteten Domäne (bisherige Erfahrungen, Muster und Best Practices) sowie der vorangegangenen Analyse des Erstentwurfs und den identifizierten Einflüssen. Wenn in der Domäne noch keine bewährten Architekturstile vorliegen oder das Problem in Art und Umfang neuartig ist, kann auch der Erstentwurf fortgeschrieben oder ein „best guess“ basierend auf Intuition als Grobkonzept erstellt werden. Das Grobkonzept ist als Start weiterer Iterationen zu verstehen, es kann nichts unwiderruflich falsch gemacht werden.

Iterativ inkrementeller Entwurf

Über mehrere Iterationen, das bedeutet eigenständige und in sich vollständige Entwürfe, wird das zu bewältigende Problem nach und nach immer besser in Bezug auf die Randbedingungen und Ziele gelöst. Die Anzahl n sowie die inhaltlichen Änderungen der Iterationen liegen im Ermessen des Software-Architekten und hängen primär von Art und Umfang der Problemstellung ab, auch $n = 1$ wäre zulässig, aber als Sonderfall für kleine Aufgaben zu verstehen. Jeder der Entwürfe wird einer Bewertung unterzogen, um den richtigen Zeitpunkt zum Beenden des iterativ inkrementellen Vorgehens zu identifizieren. Hierauf wird noch gesondert eingegangen.

Feinkonzept

Das Feinkonzept kann entweder identisch zur Iteration n sein oder noch einmal als letzter Schritt der Verbesserung einige Änderungen gegenüber n aufweisen. Diese letzten Änderungen sollten dabei von geringem Umfang und Tragweite für die gesamte Architektur sein und sich nur im Kleinen abspielen. Eine vollständige Änderung des Architekturstils ist an dieser Stelle nicht mehr zulässig. Ein solcher Bedarf würde dazu führen, dass der iterative Entwurfsprozess erneut gestartet werden muss.

Weiteres

Bild 5.1 zeigt neben den zuvor genannten Aspekten noch eine Unterscheidung in verbindliche und optionale Anteile. Wie bereits erwähnt, soll durch Tailoring dem Software-Architekten vollständig selbst überlassen bleiben, welche Schritte als notwendig erachtet werden. Daher sind wirklich nur die Schritte als verpflichtend markiert, die essentiell für das gesamte Vorgehen sind. Für einen Minimalansatz muss es mindestens funktionale Anforderungen geben, ohne sie kann kein Software-System mit einem Kundennutzen entwickelt werden. In diesem Szenario könnte durch die unmittelbare Software-Umsetzung implizit eine Software-Architektur geschaffen werden. Grobkonzept, Iterationen und Feinkonzept würden mit dieser zusammen fallen.

Dies ist jedoch lediglich ein extremes Beispiel, um die Universalität des Ansatzes zu bestätigen. Wie bereits zuvor mehrfach vertreten, wird bei Software-Systemen mit mittlerer und großer Komplexität dringend empfohlen, alle vorbereitenden Schritte aus Bild 5.1 durchzuführen. Die Prozessschritte sind somit größtenteils nicht verpflichtend in dem Sinne, dass ansonsten die vorgeschlagene Methode des Vorgehens zum Entwurf von Software-Architekturen verletzt wird. Für das erfolgreiche Entwickeln von komplexen Software-Systemen, auch und speziell in der Fahrerassistenz, sind sie dennoch obligatorisch. Mit der Markierung „*“ ist darüber hinaus noch explizit gekennzeichnet, welche Prozessschritte und damit relevante Eingangsgrößen für die anstehende Neustrukturierung berücksichtigt werden.

Bezüglich Werkzeugkette und operativem Vorgehen wird die Neustrukturierung entsprechend Bild 4.12 rechte Seite, oberer Bereich, durchgeführt.

Bewertung der Architektur-Entwürfe

Eine Bewertung der Entwürfe wird immer nach Bedarf und Ermessen des Software-Architekten durchgeführt, mindestens aber für das Grobkonzept und das Feinkonzept. Nur so kann die Verbesserung über die Entwurfstätigkeit dokumentiert und für andere nachvollziehbar der richtige Zeitpunkt zum Beenden gefunden werden. Es wird jedoch empfohlen, nach jeder Iteration eine Bewertung durchzuführen. Eine solche Bewertung besteht aus den folgenden beiden Anteilen:

1. Objektive, automatisierte Bewertung mit den Software-Architekturmetriken und
2. Subjektive, manuelle Bewertung durch Experten.

Menschliche Intuition und Expertenwissen kann zum jetzigen Zeitpunkt noch nicht vollumfänglich durch die automatisierten Metriken ersetzt werden. Gleichzeitig sollen die Metriken auch kein Selbstzweck werden mit dem alleinigen Ziel deren Ausgabewerte zu optimieren, viel mehr sollen sie ein Hilfsmittel des Software-Architekten sein, um am Ende der Entwurfstätigkeit ein möglichst gutes Ergebnis zu erhalten. Es hat sich bewährt, beide Aspekte der Bewertung parallel einzusetzen. Es gibt hierbei kein absolut definierbares Kriterium, wann die Entwurfstätigkeit abzubrechen ist. Ebenso gibt es kein Kriterium für eine optimale, das heißt objektiv beweisbar nicht weiter zu verbessernde, Software-Architektur. Im Rahmen der Neustrukturierung wird noch diskutiert, wie sich beide Bewertungsarten für dieses Beispiel aus der praktischen Anwendung ergänzt haben und wie hier das Endkriterium für die iterative Entwurfsarbeit gewählt wurde.

Für die subjektive, manuelle Bewertung wurde das in Tabelle E.2 in Anhang E.1 dargestellte Schema eingeführt, um zu jedem Entwurfsschritt einheitlich bewerten zu können.

Detaillierung der funktionalen und nicht-funktionalen Einflüsse

In Bild 3.2 auf Seite 39 wurde bereits ein Vorgriff auf alle funktionalen und nicht-funktionalen Einflussgrößen auf Software-Architekturen gegeben, um die Komplexität und Bedeutung im Rahmen des Vorgehensmodells zu verdeutlichen. Diese Einflussgrößen wurden konkret im Kontext der Beobachtungen und Analysen im Bereich Fahrerassistenzsysteme ermittelt, die Kategorien und nachfolgenden Erläuterungen gelten jedoch auch in anderen Fachbereichen. Ziel des vorliegenden Abschnitts ist es diese Kategorien und wie sie Software-Architekturen im Entwurf beeinflussen (sollen), so darzustellen, dass die Kategorien in vergleichbaren Projekten wiederverwendet werden können. Die sich für ein Projekt inhaltlich konkret ergebenden Anforderungen sind natürlich individuell unterschiedlich und müssen jedes Mal neu ermittelt – oder bei Vorhandensein von ähnlichen Vorgängerprojekten mindestens kontrolliert und angepasst – werden. Dies wurde im Rahmen der Neustrukturierung der FAS-Längsführung im Detail durchgeführt, aufgrund des Umfangs der ermittelten Anforderungen konnte jedoch nur ein repräsentativer Ausschnitt davon im Anhang E.1 aufgenommen werden, um Art und Detaillierung widerzuspiegeln. Daran kann sich künftig für ähnliche Aufgaben orientiert werden.

Über beiden Kategorien in Bild 3.2 stehen die relevanten Kundenfunktionen. Sie lassen sich nicht klar einer der beiden Hälften zuordnen, sondern beeinflussen als übergeordnete Randbedingung in beiden Themenfeldern die Anforderungen der übrigen Kategorien. Aus diesem Grund wurden sie bereits zu Beginn in Abschnitt 5.2 aufgeführt. Im Folgenden erfolgt eine detaillierte Beschreibung aller funktionalen und nicht-funktionalen Kategorien an Einflussgrößen, wobei die Letztgenannten den Schwerpunkt ausmachen, da diese die Software-Architektur deutlich maßgeblicher beeinflussen.

Funktionale Einflussgrößen

- **Funktionale Anforderungen:** Zu den funktionalen Anforderungen zählt vor allem die Spezifikation. Sie beschreibt das Verhalten auf unterschiedlichen Detaillierungs-Ebenen, von der grobgranularen Kundenfunktionsebene bis zu einer Feinspezifikation, die die Funktion in allen relevanten Details, jedoch noch völlig lösungsneutral, beschreibt. Die Funktions-Architektur ist primär ein Hilfsmittel die Struktur besser zu verstehen, wird aber in diesem Kontext zu den funktionalen Anforderungen gezählt. Beide Umfänge lagen in der gegebenen Problemstellung bereits vor und sind daher in der Arbeit nicht abgebildet. Insbesondere die Spezifikation bleibt während der gesamten Neustrukturierung uneingeschränkt gültig.
- **Ausprägungen (funktional):** Unter diesem Punkt sind unterschiedliche Varianten einer Kundenfunktion zu verstehen, die dem Kunden bewusst einzeln je nach Kontext angeboten werden. Dabei sind zwei verschiedene Konstellationen anzutreffen:
  1. Unterschiede der Funktionalität bei gleicher System-Architektur. Ein Beispiel wäre das Angebot einer Geschwindigkeitsregelung mit und ohne Bremsfunktion.
  2. Unterschiede der Funktionalität bei unterschiedlichen System-Architekturen, diese sind meist getrieben durch unterschiedlichen Sensor-/Aktor-Bauformen. Ein Beispiel ist ein videobasiertes ACC Stop&Go im Vergleich zu einem radarbasierten. Der Fokus liegt hier auf den kundenerlebbaren Unterschieden aufgrund dieser Änderung, beispielsweise das Annäherungsverhalten in einer Folgefahrt.
- **Interaktionen (direkt):** Direkte Interaktionen zwischen Funktionen sind solche, die der Kunde unmittelbar erleben kann, zum Beispiel:
  - Welche Funktionen können parallel aktiv sein, welche nicht?
  - Ändert sich die erlebbare Funktion, wenn eine andere parallel aktiv ist?
  - Gibt es Übergänge und wie laufen diese ab (z.B. Rampen, ...)?
- **Funktionales Sicherheitskonzept:** Das funktionale Sicherheitskonzept legt lösungsneutral fest, wie die Sicherheitsziele durch Teilfunktionen (z.B. Überwachungen, Abschaltungen) zu erreichen sind. Dieses Anforderungen werden gleichberech-

tigt zu den weiter oben genannten funktionalen Anforderungen weiterhin berücksichtigt ohne separat an dieser Stelle aufgeführt zu werden.

Nicht-funktionale Einflussgrößen

- **Nicht-funktionale Anforderungen:** Diese Kategorie macht den Großteil (58 eigenständige Anforderungen und 30 zugehörige Informationen) der im Rahmen der Neustrukturierung explizit erarbeiteten Anforderungen an die neue Struktur aus. Sie sind in die folgenden Unterkategorien aufgespalten:
  - Vorgegebene Schnittstellen auf technischer Ebene nach außen (zum Beispiel Radmomentenschnittstelle zur Bremse),
  - Geschäftsmodelle und Vorgaben zur organisatorischen (Firma, Fachabteilung) und technischen Verteilung (Plattformen) von Software-Umfängen,
  - Entwicklung, Verwendung, Wartung, Inbetriebnahme und Außerbetriebnahme der Software(-Architektur) sowie
  - Produktfaktoren/Qualitätsmerkmale, die im Wesentlichen den über das Qualitätsmodell der Software-Architekturmetriken definierten Kriterien und Attributen entsprechen und an dieser Stelle bezüglich konkreter Erwartungen ausformuliert werden.
- **Ausprägungen (System-Architektur):** Unter diesen Ausprägungen von Kundenfunktionen sind solche zu verstehen, die dem Kunden nicht bewusst angeboten werden, sondern die sich automatisch durch bestimmte Änderungen an der System-Architektur ergeben. Diese sind nicht immer eindeutig zu trennen von den weiter oben unter Subpunkt 1. genannten funktionalen Ausprägungen. Der Fokus soll hier auf Änderungen an der System-Architektur liegen, die der Kunde nicht bewusst wahrnimmt. Treiber dieser Varianten sind zum Beispiel Unterschiede bezüglich verwendeten Bussystemen, Plattformen oder Bedienelementen.
- **Interaktionen (indirekt):** Indirekte Interaktionen sind Interaktionen zwischen den einzelnen Systemen, die vom Kunden nicht unmittelbar erlebbar sind. Sie entstehen auf Umsetzungsebene durch bestimmte Vorgaben. Ein Beispiel ist, dass alle Längsführungsregler der unterschiedlichen Systeme auf Beschleunigungsebene arbeiten sollen und daher zu koordinieren sind. Eine Abgrenzung zu den Schnittstellen nach außen aus dem Bereich der nicht-funktionalen Anforderungen ist nicht immer eindeutig möglich. Viele der letztgenannten Anforderungen werden unter den indirekten Interaktionen wieder aufgegriffen und verfeinert.
- **Technisches Sicherheitskonzept:** Das technische Sicherheitskonzept fordert auf Umsetzungsebene ganz konkrete Sicherheits-Eigenschaften bestimmter Software-Umfänge. Beispiele sind Anforderungen an Entwicklungsprozesse und Logistik von Software, an die Sicherheitsintegrität von Hardware-Partitionen in den Plattformen, die diese Software-Umfänge ausführen sollen oder dass bestimmte Anteile redundant vorhanden sein und ausgeführt werden müssen.

- **Angebotsstruktur:** Die Angebotsstruktur legt fest, welche Funktionen dem Kunden in welchem Derivat angeboten werden. Zur Definition dieses Angebots gehört auch die Festlegung, welche Ausprägung einer Funktion jeweils angeboten wird und ob Funktionen einzeln oder nur in bestimmten Ausstattungspaketen bestellbar sind. Es wird somit implizit festgelegt, welche Konfigurationen und Ausbaustufen von einem Gesamtsystem mindestens ermöglicht werden müssen. Eine beispielhafte grafische Übersicht einer solchen Angebotsstruktur zeigt Bild 5.2. Es sind der Serien- und Sonderausstattungsumfang sowie Zwangskopplungen in Form von Paketen und Verbindungen ersichtlich. ACC Stop&Go ist in diesem Beispiel immer nur zusammen mit iBrake und nur in Fahrzeugen mit Automatikgetriebe erhältlich.
- **Allgemeine Strukturierungsvorgaben:** Unter diesen Punkt fallen alle über die bisherigen Kategorien hinausgehenden Aspekte, die ebenfalls zu berücksichtigen sind. Dazu gehören insbesondere:
  - Berücksichtigung des im Rahmen dieser Arbeit vorgeschlagenen Vorgehens zum iterativ inkrementellen Entwurf von Software-Architekturen,
  - Berücksichtigung der den Software-Architekturmetriken zugrundeliegenden Ideen und Kriterien,
  - Berücksichtigung von allgemeinen Richtlinien, Mustern und Best Practices zum Entwurf von Software-Architekturen und
  - Berücksichtigung von speziellen Richtlinien, Mustern und Best Practices zum Entwurf von Software-Architekturen für die Domäne der Problemstellung.

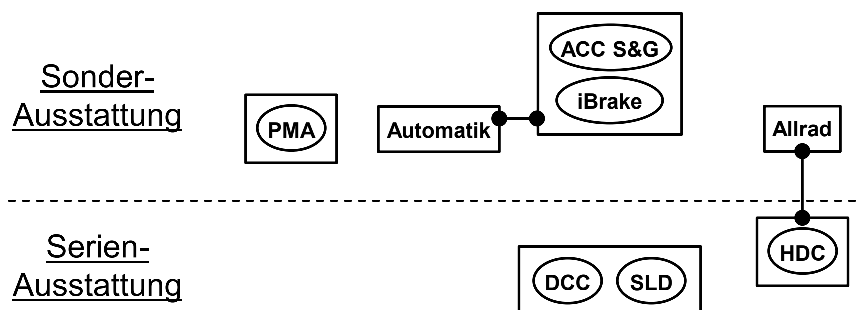

Bild 5.2: Angebotsstruktur von Fahrerassistenzsystemen im BMW 5er (2010-2013)

Weitere Details zum methodischen Vorgehen finden sich in [6].

5.3 Durchführung der Neustrukturierung der FAS-Längsführungs-Architektur

Im folgenden Abschnitt wird auf Basis der Vorüberlegungen die konkrete Neustrukturierung am realen Beispiel der Fahrerassistenzsysteme der Längsführung bezüglich

Ablauf und Ergebnissen beschrieben. Der Aufbau des Abschnitts richtet sich nach den wesentlichen Schritten des Vorgehens nach Bild 5.1. Aufgrund der hohen Komplexität dieses Beispiels kann nicht jeder Detailschritt ausführlich aufgeführt werden, es werden vielmehr alle entscheidenden Überlegungen und grundsätzlichen Maßnahmen zur Änderung der Struktur dargestellt. Es werden jedoch ausgewählte Detailschritte beispielhaft diskutiert sowie alle eigenständigen Entwürfe der Software-Architektur zur besseren Anschaulichkeit der iterativ inkrementellen Entwurfstätigkeit visualisiert.

5.3.1 Grobkonzept der Software-Architektur

Das Grobkonzept der Software-Architektur wird in Form eines Architekturstils vorgegeben. Da bisher in der Literatur kein solcher Strukturierungs-Rahmen für FAS Längsführung verfügbar ist, werden als Basis für die ersten Überlegungen etablierte Muster aus der allgemeinen Software-Architekturentwicklung verwendet und an die Spezifika der Domäne angepasst. Aufgrund dieser Unsicherheit darf der Architekturstil nicht von Beginn an als fest vorgegeben und unveränderlich angesehen werden, er muss für die im Anschluss folgende iterative Entwurfsarbeit ausreichend Freiraum geben und die Möglichkeit einräumen, auch nachträglich noch angepasst werden zu können. Dominierendes Ziel der Neustrukturierung ist eine möglichst gute Gesamtstruktur am Ende der Aktivitäten zu erreichen und nicht einen starren Prozess abzuarbeiten.

Für den Architekturstil existiert keine formale Methode, um aus den zuvor in Abschnitt 5.2.2 ermittelten Anforderungen eine geeignete übergeordnete Struktur zu erhalten. Die Summe der Einzellösungen, um jeden dieser Aspekte für sich ideal zu lösen, führt nicht zwangsläufig zu einem idealen Architekturstil. Dieser muss vielmehr aus der Intuition des Software-Architekten aus einer Gesamtsicht heraus gebildet werden. Die objektiven Software-Architekturmetriken können erst auf einer tieferen Detaillierungs-Ebene eingesetzt werden. Zur Ermittlung eines geeigneten Architekturstils werden einige grundsätzliche Alternativen aufgestellt und die beste im Anschluss weiterverfolgt.

Bild 5.3 zeigt den über alle Alternativen identisch eingesetzten Strukturrahmen. Er ist ein abstraktes Muster und gilt damit übergeordnet, das bedeutet er ist nicht an konkrete Umsetzungen, insbesondere Partitionierungen auf Plattformen, gebunden. Der Vollständigkeit halber ist die Unterscheidung zwischen HighLevel- und LowLevel-Software ebenfalls dargestellt. Die HighLevel-Software repräsentiert gesamthaft die vom Kunden erlebbare Funktionalität, die dort identifizierten Umfänge existieren im Fahrzeug genau einmal, können jedoch über unterschiedliche Plattformen verteilt sein. Die LowLevel-Software ist vom Kunden nicht erlebbar und stellt die Befähigung dar, die HighLevel-Software – oder Teile davon – auf einer konkreten Plattform ausführen zu können und wird somit im Zuge der Integration der HighLevel-Software mehrfach (genau einmal pro verwendeter Plattform) benötigt. Aus diesem Grund ist die im Bild dargestellte Instanz der LowLevel-Software eher symbolisch zu verstehen. Im Rahmen der Neustrukturierung wird sich ausschließlich auf die HighLevel-Software beschränkt. Die HighLevel-Software ist bereits unterteilt dargestellt in eine Signalbereitstellung, die für komplexe Software-

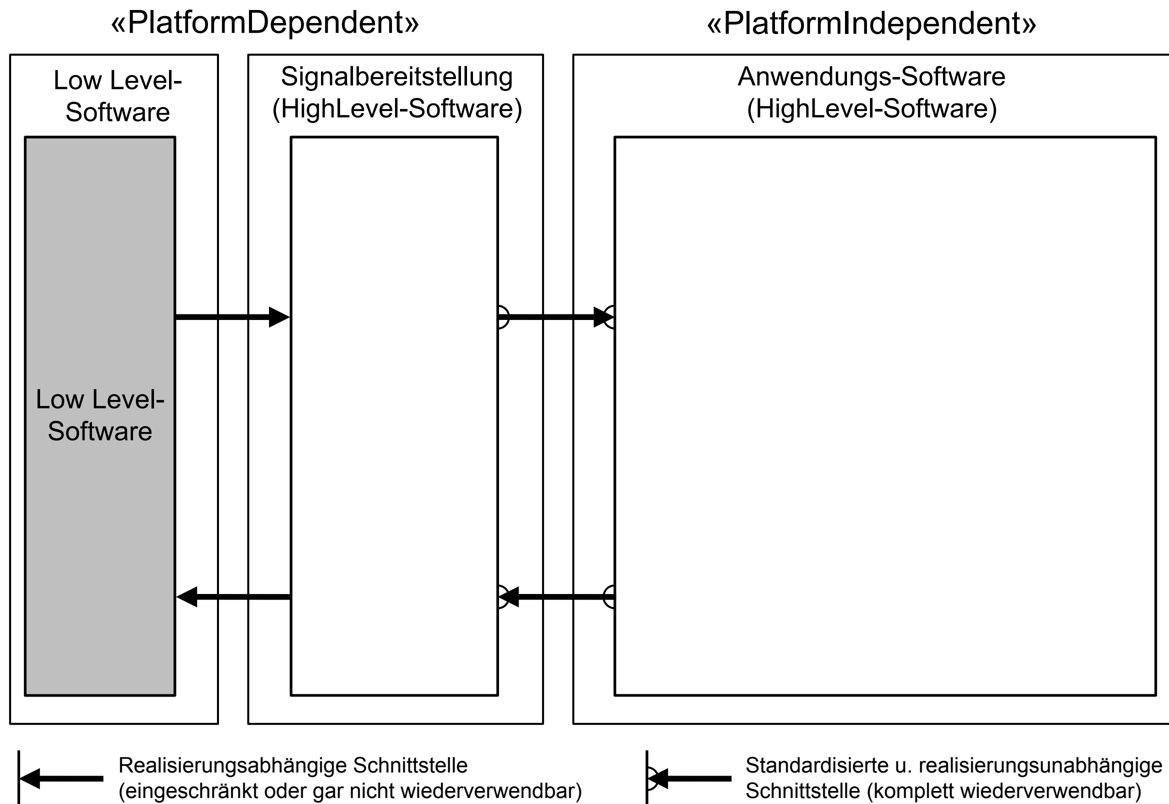

Bild 5.3: Grundsätzlicher Strukturrahmen des Architekturstils

Systeme in allen Domänen notwendig ist, und den Rest der Anwendungs-Software, die den domänenspezifischen Anteil repräsentiert. Der zu entwerfende Architekturstil wird im Folgenden als Detaillierung des letztgenannten Anteils entworfen. Für alle in diesem Kapitel noch folgenden Abbildungen gilt, dass optisch zwischen Schnittstellen unterschieden wird, die realisierungsabhängig beziehungsweise realisierungsunabhängig (Pfeile mit herkömmlichen Linienenden beziehungsweise mit Halbkreiselementen daran) sind.

Für den Architekturstil wurden zwei grundsätzliche, völlig unterschiedliche Arten der Schneidung des Systems identifiziert, die im Anschluss bezüglich ihrer grundsätzlichen Eigenschaften verglichen werden:

1. Die Schneidung nach wiederkehrenden Teilaufgaben und
2. Die Schneidung nach in sich abgeschlossenen Kundenfunktionen.

Für beide im Folgenden dargestellten Alternativen gilt, dass die eingezeichneten Kommunikationspfade eine Erwartungshaltung darstellen, welcher Austausch von Informationen mindestens für die Erfüllung der Aufgaben zwischen den Subsystemen notwendig ist. Diese ist im weiteren Verlauf der Neustrukturierung noch zu bestätigen. Grundsätzliches Ziel ist immer die Wechselwirkungen und damit die Abhängigkeiten so gering wie möglich zu halten.

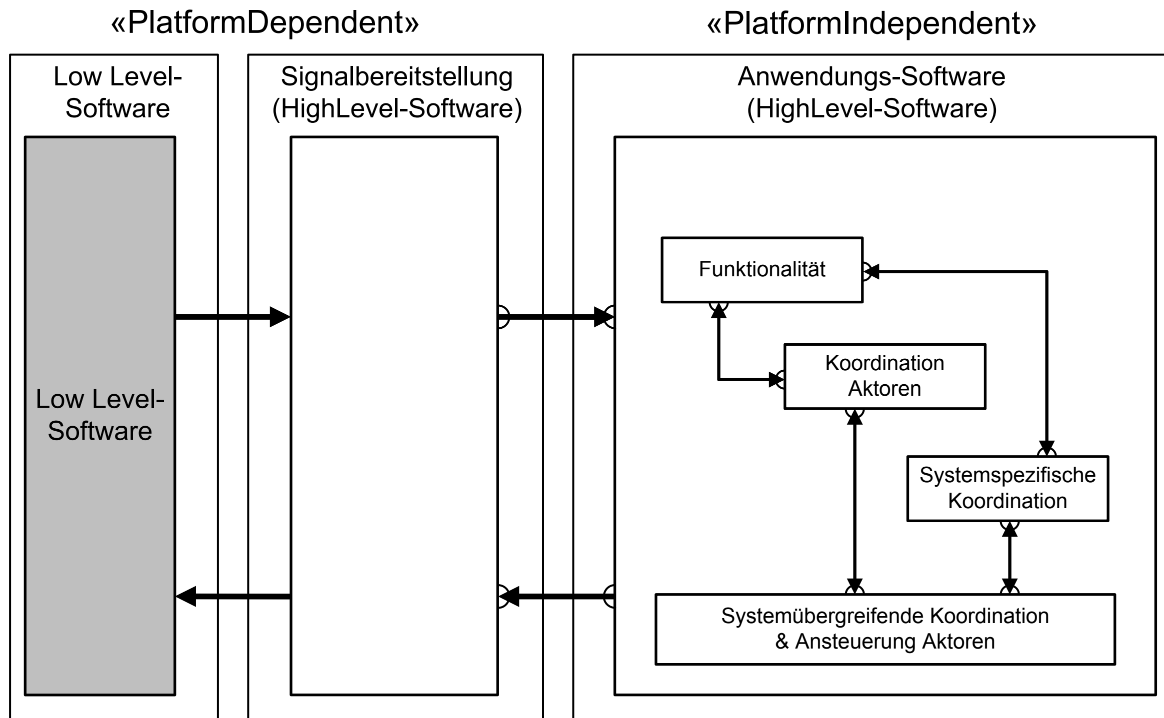

Bild 5.4: Architekturstil – Variante 1a

Die Schneidung nach wiederkehrenden Teilaufgaben (Variante 1a)

Für die Variante 1 wird grundsätzlich die Trennung von Verantwortlichkeiten umgesetzt. In Bild 5.4 ist die Variante 1a dargestellt, in Anhang E.2 findet sich die eng verwandte Variante 1b. Die Variante 1a besteht im Kern aus vier Subsystemen:

- **Funktionalität:** Das Subsystem enthält die „Kernfunktionalität" aller Kundenfunktionen. Dazu gehören funktionsspezifische Signalinterpretationen (Vorverarbeitung), die über den allgemeinen Umfang (siehe Signalbereitstellung, Bild 5.3) hinausgehen. Darüber hinaus werden hier Sollwerte für die Längsführung (als Beschleunigungsvorgabe) und die Anzeigesteuerung generiert.
- **Systemspezifische Koordination:** Hier erfolgt die Steuerung der jeweiligen Systemzustände aller Kundenfunktionen.
- **Koordination Aktoren:** Unter Koordination Aktoren wird die Priorisierung der Beschleunigungsvorgaben, die Umrechnung auf Radmomente sowie die Aufteilung auf Antrieb und Bremse verstanden.
- **Systemübergreifende Koordination & Ansteuerung Aktoren:** Hier erfolgt einerseits die Zustandssteuerung des Gesamtsystems und damit die Koordinierung der Aktivierung aller Kundenfunktionen. Andererseits werden die priorisierten und koordinierten Sollwerte der aktuell aktiven Funktion an die Aktoren (Antrieb, Bremse, Anzeigen) ausgegeben.

Hinzufügen neuer Kundenfunktionen zum Gesamtsystem erhöht nicht die Anzahl der Subsysteme beziehungsweise ändert die Oberstruktur nicht. In diesem Falle sind lediglich spezifische, zusätzliche Anteile in die vorhandenen Subsysteme zu integrieren. Die Subsysteme sind unabhängig voneinander und müssen nicht viele Informationen austauschen (schwache Kopplung). In einem Subsystem sind dafür mehrere, sehr ähnliche Anteile vorhanden, wie beispielsweise die Generierung einer Beschleunigungsvorgabe (hohe Kohäsion). Für eine Skalierbarkeit dieses Gesamtsystems müssen alle Subsysteme in leicht zu konfigurierenden, unterschiedlichen Ausbaustufen vorliegen, da die Anzahl der Subsysteme immer identisch ist. Es ändert sich der Inhalt jedes Subsystems von Konfiguration zu Konfiguration.

Die Schneidung nach in sich abgeschlossenen Kundenfunktionen (Variante 2)

Variante 2 (Bild 5.5) weist einen komplett unterschiedlichen Ansatz der Strukturierung auf. Zwar wird ein Subsystem der vorgestellten Variante 1a ebenfalls verwendet (Systemübergreifende Koordination & Ansteuerung Aktoren), darüber hinaus ist die Struktur jedoch grundlegend anders. Die Idee ist, dass jede Kundenfunktion mit einem eigenen Subsystem an diesen kommunalen Baustein andockt und somit eine beliebig erweiterbare Grundstruktur von „parallel geschalteten“ Kundenfunktionen entsteht. Der Vorteil ist ein übersichtlicher und wiederkehrender Aufbau mit einer guten, intuitiven Verständlichkeit sowie eine auf den ersten Blick gute Skalierbarkeit. Die Skalierbarkeit in Form des einfachen Weglassens einzelner Säulen je Kundenfunktion wird jedoch sehr teuer in Bezug auf Ressourcen erkauft. Um jede einzelne Säule derart unabhängig von den anderen zu machen, muss in jeder Säule ein sehr ähnlicher Aufbau (Signalverarbeitung, Sollwertgenerierung, Zustandskoordination) angelegt werden. Somit werden sehr ähnliche Software-Anteile mehrfach ausgeführt, außerdem hat jede Säule eine enorm hohe Anzahl an (identischen) Ein- und Ausgangssignalen. Das bedeutet, dass lediglich die Konfiguration mit einer Säule aus Ressourcensicht sinnvoll ist, ab zwei Säulen hat diese Struktur immer Nachteile gegenüber Variante 1. Mit steigender Anzahl an Kundenfunktionen steigt ebenso die Anzahl an Säulen, so dass die Architektur immer unübersichtlicher wird.

5.3.2 Iterative Entwurfsarbeit

Die Struktur von Variante 2 ist ein gutes Beispiel für ein Schneiden nach funktionalen Gesichtspunkten und damit dafür, wie eine Software-Architektur nicht aufgebaut sein sollte. Wie bereits über die Arbeit hinweg schon mehrfach erwähnt wurde, ist die Strukturierung nach nicht-funktionalen Aspekten der zielführende Weg. Unter Einbeziehung und Gewichtung der im vorherigen Verlauf gesammelten Anforderungen wird für die folgende iterative Entwurfsarbeit Variante 1a ausgewählt. Unter gänzlich anderen Rahmenbedingungen wie einem verteilen Geschäftsmodell mit hohem Kauf-Anteil und unterschiedlichen Lieferanten für die einzelnen Kundenfunktionen könnte die Variante 2

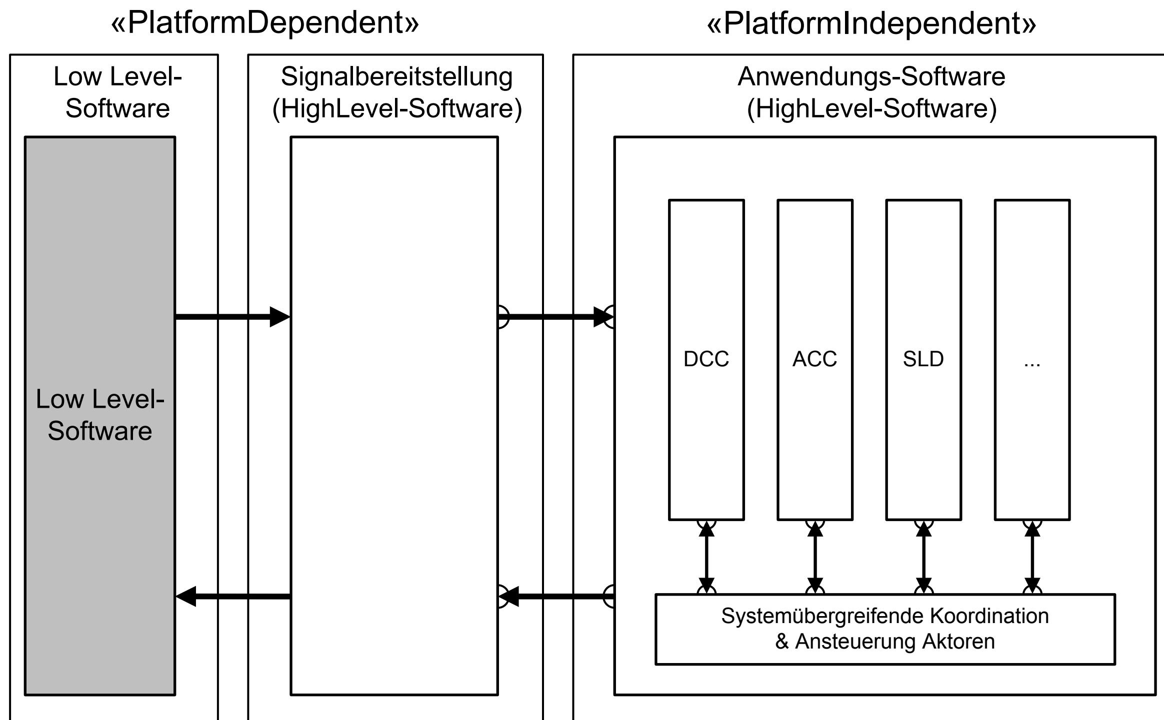

Bild 5.5: Architekturstil – Variante 2

besser geeignet sein.

Der Architekturstil wurde durch übergreifende Überlegungen auf Basis der Kenntnisse in der Domäne Fahrerassistenz und den konkret erarbeiteten Randbedingungen erreicht. Im vorliegenden Abschnitt wird die Architektur genauer – das heißt über alle Hierarchie-Ebenen hinweg – betrachtet und dabei auf die bereits konkret vorhandene Software-Architektur der FAS-Längsführung zurückgegriffen. Die existierenden Konstrukte werden den Subsystemen aus dem Architekturstil zugeordnet (denn diesen gab es bisher noch nicht) und dann innerhalb der Subsysteme die eigentliche Neustrukturierung durchgeführt. Dies soll iterativ inkrementell geschehen, die Anzahl der Iterationen muss jedoch beherrschbar bleiben. Mit Blick auf den Architekturstil wurden drei Iterationen mit jeweils vorab festgelegtem Betrachtungsumfang definiert. Alternativ wären auch fünf Iterationen denkbar gewesen, so dass jedem Subsystem der obersten Ebene genau eine Iteration hätte zugeordnet werden können.

- Stufe1: Signalbereitstellung (außerhalb der eigentlichen Anwendungs-Software),
- Stufe2: Funktionalität und Systemspezifische Koordination und
- Stufe3: Koordination Aktoren und Systemübergreifende Koordination & Ansteuerung Aktoren.

Als Namenskonvention wird ab hier eingeführt, dass Subsysteme auf Ebene des Architekturstils grundsätzlich mit zwei Buchstaben und Modules als wichtigster Typ Archi-

tekturbaustein unterhalb dieser Ebene mit drei Buchstaben abzukürzen sind. Ausgangsbasis der iterativen Durchführung der Neustrukturierung ist ein Serien-Softwarestand des BMW 5er, der die Kundenfunktionen aus Bild 5.2 enthält. Dieser wird mit dem in Kapitel 4 beschriebenen Vorgehen automatisch von ASCET ins Zwischenprodukt überführt. Da ASCET keine Subsysteme unterstützt, werden die aus dem Architekturstil ersichtlichen «systems» manuell in die XML-Datei des Zwischenprodukts eingetragen. Alle darüber hinausgehenden strukturellen Umbauten, die im folgenden Abschnitt beschrieben sind, erfolgen in ASCET. Die Bewertung des Zwischenprodukts mit den Software-Architekturmetriken wird zu allen Punkten der Bearbeitung (Ausgangsstand, je Iteration (1–3) und für den Endstand) automatisiert durchgeführt. Die Ergebnisse der Metriken werden in Tabelle E.11 in Anhang E.2 gebündelt dargestellt.

Iteration 1 – Signalbereitstellung (SB)

Der erste Schritt ist eine sinnvolle Unterstruktur dieses Subsystems zu definieren, die sich über die im weiteren Verlauf dieser Iteration folgenden Schritte bewähren muss (Bild 5.6). Ganz links ist das Module „Signale_HR“ zu sehen, welches die Brücke zur LowLevel-Software des Fahrzeuges bildet. Es wird hier nicht näher betrachtet, kann jedoch über den Typ «LowLevelAdaptationModule» als Module ohne Verhalten in der ABSOFA beschrieben werden. Es wird davon ausgegangen, dass sich die Ausgänge dieses Adapters von Variante (Konfiguration) zu Variante je nach Fahrzeug ändern und damit realisierungsabhängig sind. Damit sind die Eingänge der Bausteine des Subsystems per Definition ebenfalls realisierungsabhängig. Wichtig ist jedoch, dass über das Verhalten der Bausteine erreicht wird, dass die Ausgänge vollständig realisierungsunabhängig sind, so dass der übrige Teil der Anwendungs-Software maximal wiederverwendbar ist. Dies wird für die Bausteine in „SB“ durch Umsetzung in mehreren Varianten erreicht.

Die zwei Hauptbausteine sind das „Input-“ (INP) und das „OutputModul“ (OUT). Diese sind dafür zuständig, die von der Anwendungs-Software benötigten und bereitgestellten Signale so aufzubereiten, dass diese mit der Fahrzeugseite zusammenpassen. Dabei ist jedoch anzumerken, dass nur solche Signale durch diese Modules gehen, die funktional bearbeitet werden müssen. Somit sind sie ganz klar den Modules mit Verhalten zuzuordnen. Gleichzeitig ist es zulässig, dass Signale in beiden Richtungen zwischen Signale_HR und der Anwendungs-Software ausgetauscht werden dürfen, wenn diese schon dem richtigen Format entsprechen. Die Modules INP und OUT dürfen nicht zu einem Selbstzweck verkommen mit dem Anspruch jegliche Kommunikation zu steuern und explizit umzuleiten, dies wäre aus Ressourcensicht hochgradig ineffizient.

Das Module „InterpretationBedienelement“ (IBE) stellt einen Sonderfall der Eingangsdaten-Aufbereitung dar. Dieser Umfang könnte auch dem INP zugeordnet werden. Bediensignale sind jedoch von ihrem Charakter und Bedeutung für die Kundenfunktionen so speziell, dass im Sinne der Trennung von Verantwortlichkeiten das zusätzliche Module vorgehalten wird. Ein IBE ist nur dann zulässig und strukturell richtig, wenn keine Eingangs-Signale sowohl an IBE als auch an INP anliegen und auch keine Kommunika-

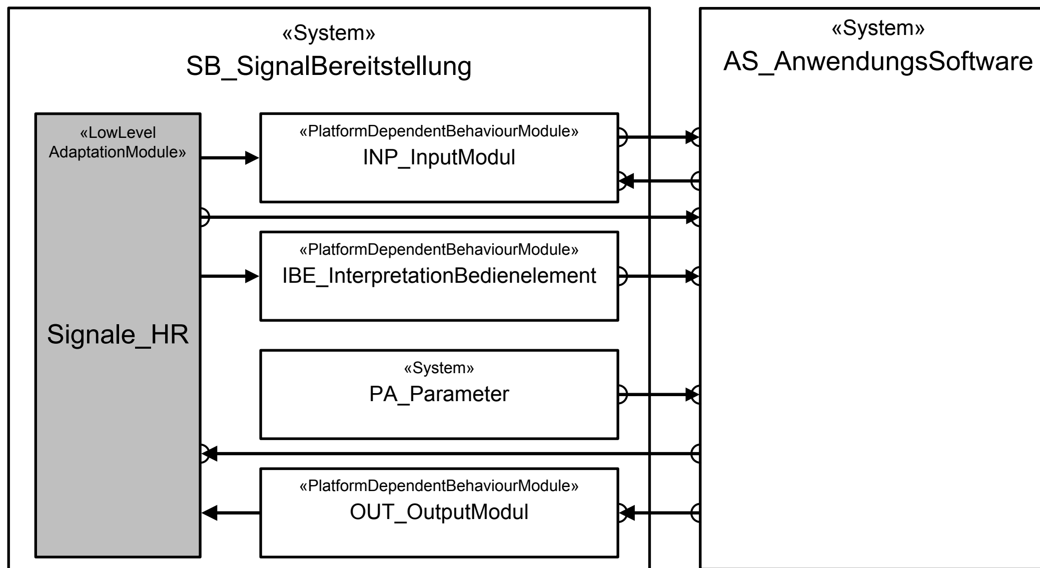

Bild 5.6: Iteration 1 – innere Struktur von SB_Signalbereitstellung

tion zwischen diesen Modules vorliegt.

Parameter sind für Automotive-Software enorm wichtig und können als eigenständige Quelle verstanden werden, da diese ohne Abhängigkeit von anderen Informationen direkt bereitgestellt werden. Aus diesem Grund sind die «ParameterModules» ein Sondertyp der (realisierungsunabhängigen) Modules ohne Verhalten. Da die Parameter sehr leicht in unabhängige Ausbaustufen zerlegt werden können, sind diese in einem weiteren «system» „PA“ gruppiert. Ausbaustufen werden im Anschluss noch näher erläutert.

Neben dem vorgestellten Grundmuster werden noch die folgenden Ziele als Teil der Neustrukturierung im Kleinen verfolgt:

- Signale, die von keinem Baustein mehr benötigt werden, entfernen,
- ASCET-spezifische Schwachstellen auflösen (globale Variablen, mehrfaches Schreiben von Messages, Verwendung von Send-Receive-Messages),
- Unnötige Abhängigkeiten intern und mit der Anwendungs-Software auflösen und
- Die Bausteine innerhalb SB mit Hilfe von Ausbaustufen modularisieren.

Ein wichtiges Hilfsmittel ist die Erstellung von Ausbaustufen oder anders ausgedrückt Konfigurationsvarianten der Bausteine. Das Konzept wird im Folgenden vorgestellt und wird für alle Bausteine in allen Iterationen immer wieder auf ähnliche Art angewendet. Da der Architekturstil bewusst nicht nach Funktionen (wie Variante 2) sondern nach Verantwortlichkeiten schneidet, ist es notwendig alle Bausteine für unterschiedliche Kontexte und Funktionsumfänge vorzubereiten. Die meisten Bausteine kommen in allen Architekturvarianten vor, weil immer eine Teilfunktionalität darin umgesetzt sein muss. Das Ziel ist hierbei, dass genau die benötigte Funktionalität – und nicht mehr

– enthalten ist, die richtige Konfiguration eindeutig bekannt ist und ohne großen Aufwand ausgewählt werden kann. Dies wird über eine bestimmte Anzahl an so genannten Ausbaustufen erreicht. Diese orientieren sich stark an der Angebotsstruktur. Neben der Identifikation von Basis-Umfängen, die mehreren Funktionen gemein sind, gibt es auch spezifische Anteile, die nur bestimmte Funktionen benötigen. Basis-Umfänge müssen jedoch mindestens in zwei Funktionen benötigt werden, die in geringer Ausstattung standalone vorkommen können wie beispielsweise DCC und SLD. Spezifische Anteile werden immer so gestaltet, dass diese als unabhängige Erweiterung zu einfacheren Funktionen verwendbar sind. Dies gilt zum Beispiel für ACC, welches immer als modulare Erweiterung einer DCC-Funktion, die wiederum zusätzlich zu den Basis-Umfängen vorhanden ist, angesehen wird. Bild 5.7 zeigt die Struktur der Ausbaustufen.

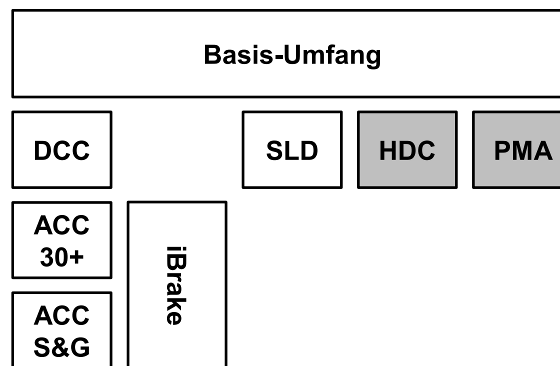

Bild 5.7: Relevante Ausbaustufen für die Architekturbausteine

Iteration 2 – Systemspezifische Koordination (SK) und Funktionalität (FU)

In dieser Iteration steht die Strukturierung der Subsysteme SystemspezifischeKoordination (SK) und Funktionalität (FU) im Fokus. Sie repräsentieren jeweils einen größeren Umfang als die zuvor betrachtete Signalbereitstellung, so dass die gesamte Iteration deutlich umfangreicher ist als die erste. Der grundsätzliche Ziel-Inhalt beider Subsysteme wurde bereits in Abschnitt 5.3.1 vorgestellt und bleibt gültig. Sehr ähnlich wie in der Iteration zuvor werden die folgenden Hauptverbesserungsmaßnahmen mit einigen spezifischen Ergänzungen als Ziele verfolgt:

- Sinnvolle Anzahl und Verantwortlichkeiten der Modules schaffen,
- Unnötige Abhängigkeiten intern und mit dem Rest der Anwendungs-Software auflösen,
- Übergeordnet koordinierende Anteile aus der spezifischen Zustandssteuerung des ACC entfernen und
- Die Bausteine mit Hilfe von Ausbaustufen modularisieren.

Systemspezifische Koordination (SK)

In der alten Struktur war das Module „SystemCoordination“ (SCO), das historisch aus der ersten Generation des ACC entstanden ist und eigentlich die Zustandssteuerung von ACC und DCC übernehmen sollte, sehr umfangreich und strukturell nicht stringent aufgebaut. Wie auch schon der übergreifend klingende Name andeutet, hat der SCO nach und nach auch Teile der übergreifenden Zustandssteuerung übernommen. Dies führte zu einem unnötig komplexen Aufbau des Modules und einer unklaren Aufgabenteilung mit der systemübergreifenden Koordination sowie etlichen unnötigen Abhängigkeiten innerhalb der Architektur und einer insgesamt kaum nachvollziehbaren Gesamtstruktur. Das Umgestalten dieses Teilumfangs wurde demnach zum Schwerpunkt der Tätigkeit innerhalb SK, dabei wurde der SCO zerlegt und nur die für die Geschwindigkeitsregelsysteme ACC und DCC spezifischen Anteile in einem neuen Module „ZustandssteuerungGeschwindigkeitsregelung“ (ZGR) übrig behalten. Stellvertretend für etliche vergleichbare Maßnahmen innerhalb der Neustrukturierung von SK wird das folgende Beispiel genannt. Bisher existierte die Funktionalität zur Einstellung einer Wunschgeschwindigkeit mehrfach im Gesamtsystem, einmal im SCO (neu: ZGR) und einmal in „ZustandssteuerungSLD“ (ZSL). Diese konnte in einem einzigen Module „KoordinationGeschwindigkeitsVerstellung“ (KGV) zentralisiert werden, spezifische Einstellungen wie die Wunsch-Abstandsstufe des ACC verbleiben in den jeweiligen spezifischen Bausteinen.

Funktionalität (FU)

Unter der Funktionalität (FU) werden alle einen Sollwert generierenden Umfänge verstanden. Das sind zum einen die verschiedenen Regelungen auf Beschleunigungsebene für Freifahrt (DCC, ACC), Folgefahrt (ACC), Standzielregelung (ACC S&G), für die Begrenzung bei aktiver Geschwindigkeitsbegrenzung (SLD) sowie durch direkte Fahrerbetätigung im so genannten Comfort Dynamic System (CDS). Zum anderen gehörten nach bisheriger Sichtweise auch die Umfänge dazu, die Sollwerte für die Anzeigen generieren. Für die erst genannten Umfänge wurden einige Verbesserungsmaßnahmen im Kleinen durchgeführt auf die hier nicht näher eingegangen werden soll. Die grundsätzliche Struktur mit eigenständigen, parallelen Bausteinen für die Beschleunigungsvorgabe der unterschiedlichen Kundenfunktionen soll beibehalten werden. Was bei der näheren Betrachtung jedoch auffiel ist, dass die Vorgabe der Anzeige-Sollwerte viel stärker an die jeweiligen Funktionszustände als an die Vorgabe der Beschleunigungs-Sollwerte gekoppelt ist. Somit liegt ein hoher Bedarf an Austausch von Informationen zwischen SK und FU vor, der aus struktureller Sicht auf Ebene der Subsysteme des Architekturstils nicht erwartet wurde.

Diese Erkenntnis hat zu einer Anpassung des Architekturstils zur neuen Variante 1c geführt, die in Bild 5.8 dargestellt ist. Der Betrachtungsumfang des Subsystems SK wird erhöht. Seine Bezeichnung wird erweitert, um dem Rechnung zu tragen und lautet ab sofort „Systemspezifische Koordination & (Sollwertgenerierung) Anzeigen“, wobei

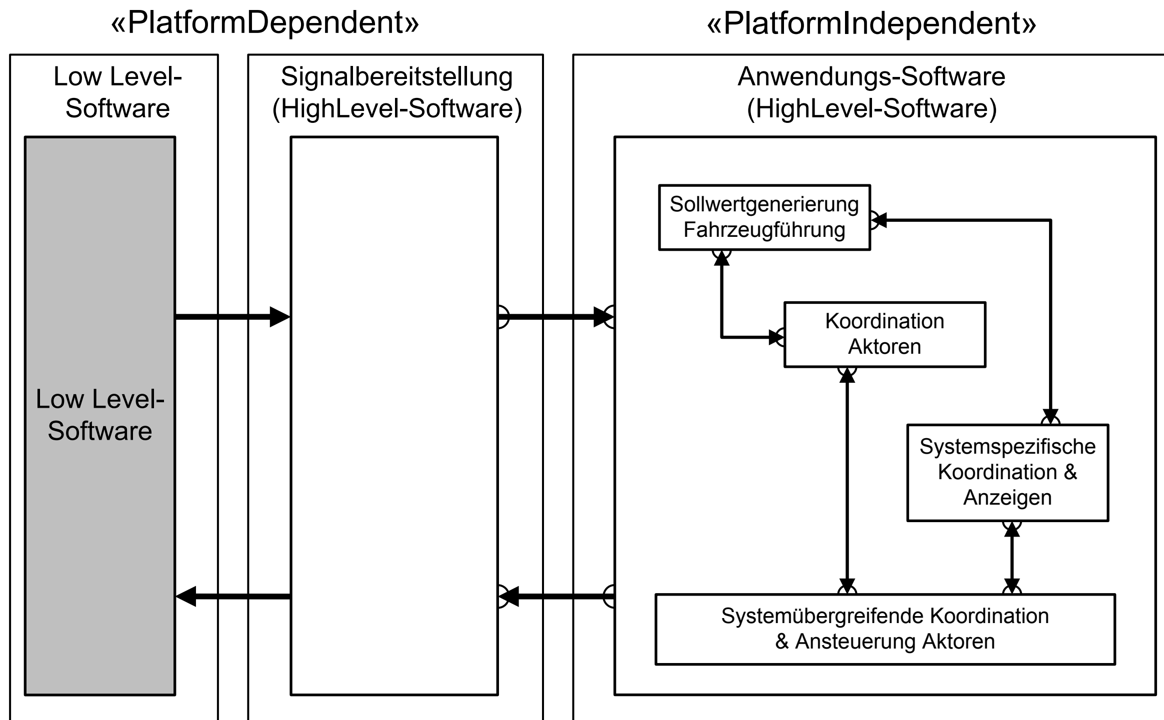

Bild 5.8: Architekturstil – Variante 1c

das Wort in Klammern zur Vereinfachung weggelassen wird. Konkret bedeutet dies, dass die folgenden Modules vom alten Umfang FU in den neuen Umfang SK verlagert werden: „AnsteuerungAnzeigeGeschwindigkeitsregelung“ (AAG), „Fahrerübernahmeaufforderung“ (TOR: aus dem Englischen für „Take Over Request“) und „AnsteuerungAnzeigeSLD“ (AAL). Aufgrund des hohen Kommunikationsbedarfs zwischen den Zustandssteuerungen und den dazugehörigen die Anzeigen steuernden Bausteinen, werden diese für die Geschwindigkeitsregelsysteme und das Speed Limitation Device noch jeweils in einem eigenen Subsystem zusammengelegt. Wegen der zuvor genannten Änderungen ist der Name Funktionalität für die übrigen Umfänge nicht mehr passend. Die neue Bezeichnung lautet „SollwertgenerierungFahrzeugführung“ (SF). In SF verbleiben somit lediglich die eingangs erwähnten Regelungen für die Sollwert-Vorgabe auf Beschleunigungsebene. Die Detail-Unterstrukturen sowie die Kommunikationsbeziehungen der beiden Subsysteme SK und SF können Bild E.3 und E.4 in Anhang E.2 entnommen werden. Dort ist die jeweils letzte Version entsprechend der zuvor genannten Änderung des Architekturstils und der Zuständigkeiten der Subsysteme dargestellt.

Iteration 3 – Koordination Aktoren (KA) und Systemübergreifende Koordination & Ansteuerung Aktoren (AA)

Die vermutlich größte Herausforderung im Rahmen dieser Neustrukturierung liegt in der dritten Iteration. Während das Ziel der Strukturierung bisher war die Anteile der einzelnen Kundenfunktionen möglichst wechselwirkungsfrei zu gestalten und sauber voneinander zu trennen, müssen die Anteile nun zusammengeführt werden, damit am Ende das Gesamtsystem mit der Vielzahl an Kundenfunktionen korrekt und funktionsfähig ist. Im Fokus stehen dabei zum einen die Koordinierung der Systemzustände der Einzelfunktionen zu einem Gesamtzustand und die anschließende Ansteuerung der Aktoren wie Antrieb, Bremse und Anzeigen. Dies wird in dem Subsystem „SysuebergreifendeKoordAnsteuerungAktoren“ (AA) erzielt. Zum anderen muss zuvor aus der Vielzahl der potenziellen Sollwert-Vorgaben auf Beschleunigungsebene eine Priorisierung sowie Umrechnung auf Radmomente und anschließende Verteilung dieser auf Antrieb und Bremse erfolgen. Letztgenanntes erfolgt im Subsystem „KoordinationAktoren“ (KA). Da die beiden Anteile so eng zusammen gehören und stark wechselwirken, werden die Überlegungen und Maßnahmen der Neustrukturierung dieser Umfänge gesamthaft beschrieben.

Anders als beispielsweise für Iteration 2 wird für Iteration 3 im Falle KA vorab bereits eine Struktur, die durch Vorüberlegungen gewonnen wurde, als klares Ziel festgelegt. Dabei handelt es sich um das Konzept der so genannten zwei Pfade für jeweils eine Maximum- und eine Minimumauswahl aus den ermittelten Soll-Radmomenten des Fahrerassistenzsystems verglichen mit dem aktuellen Fahrerwunsch.

- **Pfad 1 – Maximumauswahl:** Aus Sicht des Gesamt-Fahrzeugs wird das Maximum aus der System- und der Fahrervorgabe für die Längsführung ausgewählt. Dies entspricht dem Standardfall bei Geschwindigkeitsregelsystemen sowie Notbremsfunktionen. Der Fahrer kann jederzeit per Fahr- oder Bremspedal das System übersteuern (übertreten). Im Falle von untertreten durch den Fahrer wird weiterhin die Systemvorgabe umgesetzt, ein Bremseingriff führt allerdings häufig zum Abschalten von FAS-Längsführungsfunktionen.
- **Pfad 2 – Minimumauswahl:** Aus Sicht des Gesamt-Fahrzeugs wird das Minimum aus der System- und der Fahrervorgabe für die Längsführung ausgewählt. Dieses Prinzip wird für Funktionen angewendet, welche die Geschwindigkeit (konkret: Speed Limitation Device) oder die Leistung des Fahrzeuges beschränken sollen. Auch bei höherer Momenten-Anforderung des Fahrers per Fahrpedal als die aktuelle Systemvorgabe, wird weiterhin diese umgesetzt. Solange der Fahrer das Fahrzeug unterhalb der Grenze bewegt und damit weniger anfordert, wird sein Wunsch umgesetzt. Die Systemvorgabe ist hier daher eher als Begrenzung des Sollwertes zu verstehen und nicht wie bei einem der oben genannten Systeme ein kontinuierlicher Sollwert einer Regelung. Der Pfad der Minimumauswahl für die genannten Systeme bezieht sich allerdings nur auf den Antrieb, da einerseits kein aktiver Bremseingriff des Systems erfolgt, der mit dem Fahrer verglichen werden

könnte. Andererseits ist es aus Sicht der Funktionssicherheit – und auch konstruktiv bedingt bei Bremsanlagen des heutigen Stands der Technik – nicht zulässig, einen (höheren) Bremswunsch des Fahrers nicht umzusetzen.

Die zwei getrennten Pfade sind deswegen sinnvoll, da bereits jetzt abzusehen ist, dass künftig Fahrerassistenzsysteme aus beiden Pfaden gleichzeitig aktiv sein können. Ein Beispiel ist eine Fahrt mit aktivem ACC während eine Leistungsbegrenzung des Fahrzeugs im Hintergrund aktiv ist. Aus Sicht Gesamt-Fahrzeug wird das Maximum aus Fahrer- und ACC-Vorgabe in eine Minimumauswahl mit der Leistungsbegrenzung geführt. Voraussetzung für eine erfolgreiche Umsetzung im Fahrzeug ist eine entsprechende Schnittstelle von den Fahrerassistenzsystemen zum Antrieb, die es in der bisherigen Struktur noch nicht gibt. In beiden Pfaden werden Beschleunigungsvorgaben generiert, wobei im Minimumpfad diese wie erwähnt einer Grenze einer noch zulässigen Beschleunigung entspricht. Die Sollwerte werden in ihrem jeweiligen Pfad gemäß der Vorgabe der systemübergreifenden Koordination über die Aktivierungszustände der einzelnen Systeme priorisiert und anschließend auf ein Gesamt-Radmoment umgerechnet. Dieses wird im Anschluss auf Antrieb und Bremse (nur im Maximum-Pfad relevant) verteilt und an die systemübergreifende Koordination weitergegeben, da der Zugriff auf die Aktoren Antrieb und Bremse nur zentral von dort aus zulässig ist. Die Schnittstelle nach außen über die Soll-Radmomente sowohl zum Antrieb als auch zur Bremse existiert bereits und hat sich bewährt, daher wird an dieser festgehalten. Aufgrund des zweiten Pfades müssen jedoch künftig zumindest an den Antrieb zwei Radmomente übertragen werden. In der bisherigen Struktur wird nur ein Moment übertragen mit der Information, welches Fahrerassistenzsystem aktiv ist, so kann der Antrieb die Vorgabe interpretieren.

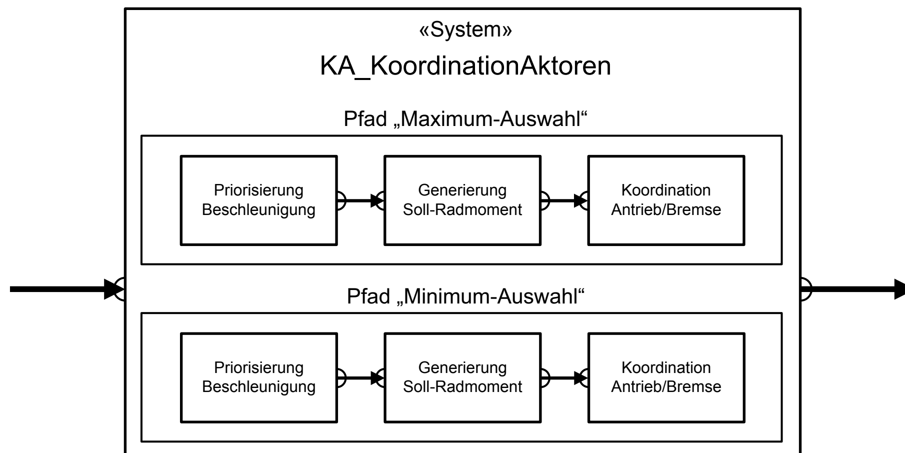

Bild 5.9: Iteration 3 – Grundidee von KA_KoordinationAktoren

Die neue Struktur etabliert die getrennten Pfade und setzt die Priorisierung, Umrechnung und Aufteilung auf Antrieb und Bremse noch konsequenter um. Ein wesentlicher

Anteil ist die so genannte Situationserkennung, die für die aus Fahrersicht komfortable Aufteilung auf Antrieb und Bremse benötigt wird. Eine Situation ist beispielsweise das „Anfahren“. Je nach Situation läuft die Aufteilung und Berechnung der Radmomente unterschiedlich ab, insbesondere Rampen und Übergänge stehen im Fokus. In der bisherigen Struktur ist dieses Konstrukt inzwischen sehr unübersichtlich und dezentral mehrfach für die unterschiedlichen Kundenfunktionen vorhanden. Durch einen generischen Situationskatalog soll dies an einer zentralen Stelle gebündelt werden. Bild 5.9 visualisiert die Grundidee der zwei Pfade im Subsystem KoordinationAktoren (KA).

Das letzte verbleibende Subsystem der Software-Architektur ist SysuebergreifendeKoordAnsteuerungAktoren (AA). In diesem finden sich die Module „FASKoodination“ (FAK) für die Steuerung der Aktivitäten der einzelnen Kundenfunktionen und der Priorisierung in KA, „FASMomentenKoordination“ (FMK) für die finale Nachbearbeitung der Soll-Momente der beiden Pfade (zum Beispiel Sicherheitsabschaltung) und Ansteuerung der Aktoren Antrieb und Bremse sowie „AnzeigenKoordinatorFAS“ (AKF) für die Aktoren der Anzeigen analog zu FMK. Für die Subsysteme der Iteration 3 gelten grundsätzlich die gleichen Hauptziele für die Neustrukturierung im Kleinen (primär: Generierung von Ausbaustufen und Auflösen unnötiger Abhängigkeiten) wie in den zuvor beschriebenen Iterationen, an dieser Stelle sollen noch einige spezielle Aspekte aufgeführt werden, die ebenfalls berücksichtigt wurden:

- Nur der FAK verwaltet die Aktivitätszustände der Kundenfunktionen. Die spezifischen Zustandsautomaten dürfen nicht selbst entscheiden, aktiv zu gehen, sondern dürfen nur im Aktivzustand ihre spezifischen Unterzustände selbst bestimmen.
- Die Entscheidung, welcher der Sollwerte innerhalb KA priorisiert wird, darf ebenfalls nur über FAK und nicht über eigene Logiken in KA getroffen werden.
- Übergänge (zum Beispiel Momenten-Rampen) von Systemen eines Pfades müssen in KA erfolgen. Im FMK dürfen nur Übergänge abgefangen werden, die pfadübergreifend ein komfortables Regelverhalten sicherstellen müssen.
- Alle Ansteuerungen von Anzeigen müssen zentral über den AKF geführt werden. Dies gilt auch für Anzeigeelemente, die nur in einer einzigen Kundenfunktion (zum Beispiel Anzeige der Soll-Abstandsstufe ACC) benötigt werden. So wird Konsistenz zum FMK geschaffen und gleichzeitig die Erweiterbarkeit verbessert für den Fall, dass andere Kundenfunktionen ebenfalls auf dieses Element zugreifen wollen.

Bild E.5 in Anhang E.2 stellt die innere Struktur von KA und AA dar. Dabei ist die Kommunikation über Subsystem-Grenzen hinaus vereinfacht modelliert.

5.3.3 Feinkonzept der Software-Architektur und Fazit

In den zuvor vorgestellten Iterationen 1-3 konnten alle geplanten Änderungen der Neustrukturierung umgesetzt werden, sowohl was die Änderungen im Großen (Architekturstil) als auch im Kleinen (Unterstruktur der Subsysteme) angeht. Es besteht somit

kein Bedarf nach einem weiteren Entwurf bezüglich struktureller Änderungen für das Feinkonzept. Dies zeigt auch, dass sich der Architekturstil in der Variante 1c schlussendlich für das gegebene Problem bewährt hat. Das Feinkonzept kann im Rahmen einer Neustrukturierung optional noch genutzt werden, um das Software-System über die Software-Architektur hinausgehend zu optimieren. Darunter fällt zum Beispiel das Analysieren auf ungenutzte Code-Anteile oder ungeschickte Programmierkonstrukte. Dies wurde jedoch während der oben genannten Schritte für offensichtliche Fehler bereits simultan durchgeführt. Außerdem ist dies nicht mehr unter Optimierung der Software-Architektur einzuordnen und wird daher im Kontext der Arbeit nicht näher erläutert.

An dieser Stelle soll daher ein Fazit über die Neustrukturierung gezogen werden. Dazu gehört die Zusammenfassung der erzielten Ergebnisse sowie eine Diskussion über die Software-Architekturbewertung, die begleitend und unterstützend zu den beschriebenen Aktivitäten durchgeführt wurde. Neben der automatisierten und objektiven Bewertung jedes der Entwürfe, die in Anhang E.2 über die Tabelle E.11 dokumentiert ist, wurden die Entwürfe des Ausgangsstands und des Endstands noch zusätzlich einer subjektiven Bewertung unterzogen. Dies soll helfen, die (objektiven) quantifizierten Ergebnisse der Metriken der (subjektiven) Einschätzung von Experten gegenüberstellen zu können, da menschliche Intuition und Wissen zum jetzigen Zeitpunkt noch nicht vollständig durch die Metriken ersetzt werden können. Die Ergebnisse dieser Bewertung sind in der folgenden Tabelle 5.1 dargestellt. Die aufgezeigten Werte ergeben sich als Mittelwert der Einschätzungen von erfahrenen Software-Architekten und -Entwicklern, die mit dem konkret betrachteten Software-System inhaltlich seit Jahren sehr gut vertraut sind.

Tabelle 5.1: Subjektive Bewertung von Ausgangsstand und Endstand

Nr.	Kriterium	Ausgangs-Stand	Endstand
1	Indirekte Interaktionen	o	+
2	Angebotsstruktur	–	+
3	Technisches Sicherheitskonzept	o	o
4	Ausprägungen (System-Architektur)	–	+ +
5	Vorgegebene Schnittstellen	o	o
6	Geschäftsmodelle	–	+ +
7	Entwicklung, Verwendung, Wartung, Inbetriebnahme, Außerbetriebnahme	–	+
8	Qualitätsmerkmale: Effizienz & Skalierbarkeit	– –	+ +
9	Qualitätsmerkmale: Wiederverwendbarkeit & Portabilität	– –	+
10	Qualitätsmerkmale: Wandlungsfähigkeit/Änderbarkeit	– –	+ +
11	Qualitätsmerkmale: Konformität zu verbindlichen Vorgaben	o	o
12	(optional: Umsetzung der allgemeinen Strukturierungsvorgaben)	(– –)	(o)
Mittelwert der Bewertung		–	+

Die manuelle Bewertung kann sinnvoll nur für vollständige Architektur-Entwürfe durchgeführt werden, da die vorgegebenen Kriterien der Tabelle nur als „Big Picture“ bewertbar sind. Da während der Iterationen der durchgeführten Neustrukturierung immer Umfänge – die bis zur Iteration 3 immer kleiner werden – im Platzhalter „Anwendungs-Software“ maskiert waren, wurde dort auf die subjektive Kontrollbewertung verzichtet und lediglich das Endergebnis wieder betrachtet. Die subjektive Einschätzung deckt sich

grundsätzlich mit der objektiven und sieht bei fast allen Kriterien eine Verbesserung. Speziell bei den wesentlichen Schwachstellen wie „Effizienz & Skalierbarkeit“ und „Wandlungsfähigkeit/Änderbarkeit“ kann eine deutliche Verbesserung bescheinigt werden. Für die subjektive Bewertung gibt es bewusst nur wenige zulässige Werte zur Einstufung, weil eine feinere Skala mit einer subjektiven Einschätzung nicht aufgelöst werden kann. Hier haben die Metriken grundsätzlich einen Vorteil. Es ist deutlich zu erkennen, dass die subjektive Bewertung im Mittel noch immer deutliches Verbesserungspotenzial aufzeigt. Im Mittel wäre für die Architektur noch ein weiterer, ganzer Punkt mehr erreichbar. Eine Neustrukturierung, speziell von einem solch komplexen Gesamtsystem, hat dabei strukturell immer Nachteile gegenüber einem kompletten Neudesign, da mit vertretbarem Aufwand nicht bis ins allerletzte Detail vorgedrungen werden kann. So gibt es auch im finalen Entwurf noch immer Abhängigkeiten an vereinzelten Stellen, die aus Sicht der Struktur nicht zwingend notwendig erscheinen, aber von der Funktionalität vermeintlich benötigt werden. Dies ließe sich nur durch Hinterfragen und mögliches Entfeinern der funktionalen Anforderungen final auflösen, was bei einem System dieser Größe und der Vielzahl der gewachsenen Substrukturen nicht praktikabel ist. Aufwand und Nutzen des Erschließens dieser letzten kleinen Unzulänglichkeiten stehen in keinem günstigen Verhältnis zueinander. Die Bewertung zeigt jedoch, dass Verbesserungen an den entscheidenden Stellen in der erhofften Güte erzielt wurden.

Die objektiven Software-Architekturmetriken wurden demgegenüber zu jeder Iteration angewendet. Die entwickelte Methode und Umsetzung im Werkzeug ist damit als praktikabel für Problemstellungen dieser Art bestätigt. Im Folgenden wird auf die Erfahrungen in der Anwendung der Metriken eingegangen. Sie eignen sich nicht nur, um vollständige Entwürfe zu bewerten, sondern auch zwischenzeitliche Entwurfsalternativen. Das liegt daran, dass ein quantitativ besserer Zahlenwert immer auch eine bessere Architektur anzeigt. Selbst das Maskieren von Teilen der Architektur in Platzhaltern, wie oben beschrieben, ändert nichts an dieser grundsätzlichen Aussage. So wurde der Entscheidungsprozess den Architekturstil in die Variante 1c zu ändern, wesentlich durch die Bewertung von verschiedenen Alternativen beeinflusst. Die Metriken haben trotzdem eine begrenzte Aussagekraft. Wie bereits in Abschnitt 4.3.2 ausgeführt, wurde auch während dieser Neustrukturierung bestätigt, dass mit dem aktuellen Stand nur relative Vergleiche von Entwurfsalternativen aufschlussreich sind. Die absoluten Zahlenwerte der Metriken und auch der relativen Änderungen kann nicht als Maß der Verbesserung interpretiert werden. Dazu kommt, dass die Metriken (bewusst) allgemeingültig formuliert sind und somit allgemeingültige Strukturierungsprinzipien widerspiegeln und belohnen. Domänenspezifische Muster für die Fahrerassistenz (Längsführung), die im Rahmen des Fortschritts der Arbeit erst noch entwickelt wurden, konnten naturgemäß während der Anwendung noch nicht in den Metriken berücksichtigt werden. Somit kommt es durch die spezifischen Rahmenbedingungen vereinzelt zu Konflikten mit den Metriken, die schlussendlich nur durch den Software-Architekten, und damit wieder basierend auf seiner Intuition, aufzulösen sind. Die Metriken helfen jedoch während des gesamten iterativen Entwurfsprozess sehr gut dabei, innerhalb des vorhandenen Freiraums die richtigen

Entwurfsentscheidungen zu treffen und leisten so ihren Beitrag beim iterativen Entwurf.

5.4 Diskussion der universellen Eignung der Struktur für Fahrerassistenzsysteme

Im vorliegenden Abschnitt soll abschließend noch diskutiert werden, inwieweit die für das konkrete Beispiel der FAS-Längsführung erarbeitete, neue Software-Architektur für die Domäne Fahrerassistenz in Gänze anwendbar ist. Erörtert wird dies anhand der aktuell immer stärker aufkommenden Systeme mit Teilautomatisierung, das bedeutet mit gleichzeitiger Übernahme von Längs- und Querführung durch das System bei weiterhin voller Verantwortung und Aufmerksamkeit des Fahrers. Derartige Systeme weisen gegenüber den bisher vorgestellten die folgenden Hauptunterschiede und zusätzlichen Anforderungen auf, anhand derer die Struktur-Diskussion erfolgen soll:

- Es muss ein Zugriff auf die gleichen Anzeigeelemente ermöglicht werden, so soll beispielsweise ein Stau-Assistent wegen der großen Nähe zu ACC auch auf die Anzeigen der Abstandsstufe zugreifen können.
- Es muss ein Zugriff auf Antrieb und Bremse ermöglicht werden, das heißt der Eingriff des Längsführungs-Anteils erfolgt über dieselben Aktoren wie bisher und ist damit im Gesamtsystem zu koordinieren.
- Zusätzlich muss eine Sollwert-Vorgabe für die Querführung generiert werden. Dies kann in teilautomatisierten Fahrerassistenzsystemen der ersten Generation, so wie sie bereits im Markt erhältlich sind, noch unabhängig zur Längsführung erfolgen. In späteren Generationen ist von einer globalen Längs-/Querplanung auszugehen, welche die beiden Anteile der Fahrzeugführung permanent und dynamisch aufeinander abstimmt.
- Die Funktionen müssen gleichzeitig mit einer Limiter-Funktion aktiv sein können.
- Es existieren Funktionsübergänge, die während der Fahrt ein komfortables Umschalten von Längs- und Querführung in eine gänzliche andere Aufgabe erfordern. Ein Beispiel ist der nahtlose Wechsel aus dem Stau-Assistenten in eine Parkfunktion, die das Fahrzeug in eine Parklücke am Straßenrand bewegt.

Die Diskussion der Eignung der vorgeschlagenen Struktur bezüglich der oben genannten Fragestellungen erfolgt von innen nach außen, das bedeutet über den inneren Aufbau der betroffenen Subsysteme bis zum übergreifenden Architekturstil in der Variante 1c.

Der konkurrierende Zugriff auf Anzeigeelemente kann über den vorhandenen, übergreifenden Anzeigenkoordinator sichergestellt werden. Hier wurde bewusst mit Weitblick vorgeschrieben, dass die Anforderungen an alle Anzeigeelemente – auch auf Elemente, die zum jetzigen Stand nur von einer Kundenfunktion verwendet werden – durch diesen Koordinator laufen müssen. Somit kann auch nachträglich leicht eine Logik eingebaut werden, welche (je nach Aktivierungszustand gemäß FAS-Koordinator) die eine oder die

andere Kundenfunktion die Anzeigen ansteuern lässt. Im FAS-Koordinator müssen die zusätzlichen Funktionen bezüglich der Verwaltung der Aktivierung hinzugefügt werden, gleiches gilt für funktionsspezifische Anteile im zuständigen Subsystem.

Entscheidender Prüfstein für die Architektur ist die Frage nach der Integrierbarkeit und Koordinierbarkeit der Eingriffe der Querführung im Zusammenspiel mit der Längsführung. Für die erste Generation an teilautomatisierten Systemen ohne Kopplung von Quer- und Längsführung kann die Erweiterung sehr leicht umgesetzt werden. Die Generierung der Sollwert-Vorgabe der Querführung kann in der Struktur logisch gut passend im Subsystem „Sollwertgenerierung Fahrzeugführung“ hinzugefügt werden. Ob dies in einem einzigen Module pro teilautomatisierter Kundenfunktion für Längs- und Querführung passiert oder je ein Module für Längs- und eines für Querführung angelegt wird, ist dabei eine Detailfragestellung, die noch zu entscheiden ist. Da sich in diesem einfachen Szenario die Längs- und Querführung nicht beeinflussen, können die bereits vorgestellten Anteile der Längsführung ab „Koordination Aktoren“ in jedem Fall uneingeschränkt wiederverwendet werden. Auf Ebene der übergreifenden Koordination ist eine Funktionalität notwendig, die analog zur Längsführung auch für die Querführung den Zugriff auf den gemeinsamen Aktor koordiniert. Welche physikalische Schnittstelle hier verwendet wird – im Fahrzeug heute etabliert sind eine Radlenkwinkel- und eine Lenkradmomentenschnittstelle – ist aus Sicht der Struktur irrelevant. Die geforderte gleichzeitige Aktivierbarkeit mit einer Limiter-Funktion ist auch hier problemlos möglich aufgrund der zwei unabhängigen Pfade. Funktionsübergänge durch komfortable Momentenrampen werden innerhalb eines Pfades bereits dort sichergestellt und koordiniert. Ausschließlich im Falle, dass sich bei gleichzeitiger Aktivität einer Funktion des Maximum- und des Minimumpfads die Sollwerte annähern und somit ein merklicher Übergang droht, übernimmt in der Soll-Struktur der übergreifende Momentenkoordinator die Verschleifung der Momente, so dass der Fahrer keinen harten Wechsel spürt. Dieser Mechanismus ist unabhängig vom Vorhandensein einer gleichzeitigen Querführung.

Noch anspruchsvoller werden die Fragestellungen mit Blick auf die möglichen Weiterentwicklungen mit einer Kopplung der Längs- und Querführung. Es wird von der folgenden Annahme ausgegangen. Die Kopplung der Anteile der Fahrzeugführung erfolgt ausschließlich auf der Planungs-Ebene. Das bedeutet, dass zu jedem Zeitpunkt eine vorausgeplante Trajektorie mit einem Sollkrümmungsverlauf oder x-/y-Punktmenge existiert, der ein Profil der Längsgeschwindigkeit zugeordnet ist. Eine derartig gekoppelte Längs-/Querplanung ist gut unter „Sollwertgenerierung Fahrzeugführung“ zu integrieren. Die dort generierten Führungsgrößen entsprechen in diesem Fall denen, die heute für die Längsführung alleine beziehungsweise für unabhängige Längs- und Querführungen (siehe vorangestellte Ausführungen) existieren. Wenn somit die oben genannte Annahme erfüllt wird, ist die vorgeschlagene Struktur auch für diese gravierenden Erweiterungen uneingeschränkt geeignet. Für den Übergang von teilautomatisierten Systemen während der Fahrt müssen zwei Dinge sichergestellt werden. Das ist zum einen, dass der globale Längs-/Querplaner dieses Verfahren unterstützt, was jedoch Teil seiner Umsetzung und nicht der Software-Architektur ist. Zum anderen muss in den nachgeschalteten Bau-

steinen ein kontrollierter und komfortabler Übergang der Sollwerte erfolgen. Für die Längsführung ist dies in der vorgestellten Struktur möglich. Systeme für die ein Wechsel nach der oben beschriebenen Art in Frage kommen, führen kontinuierlich und sind somit dem Maximumpfad zuzuordnen. Innerhalb des Pfades ist ein solches Umschalten und Übergangsverhalten bereits etabliert. Für den hier nicht im Detail betrachteten Pfad der Querführung wäre ein ähnlicher Mechanismus umzusetzen, dies wird von der Struktur grundsätzlich unterstützt.

Die hier erörtere Erweiterbarkeit ist ein direktes Ergebnis der gewählten Schneidung des Architekturstils. Entscheidend ist, dass die übergeordnete Struktur mit der vorgeschriebenen Trennung der Verantwortlichkeiten und den zulässigen Kommunikations-Richtungen strikt eingehalten wird, um eine Unterwanderung und damit unkontrollierte Wildwuchs zu vermeiden. Es ist denkbar, dass durch aktuell noch nicht bekannte Funktionalitäten künftig ganz neue Verantwortlichkeiten auf niedriger oder hoher Ebene notwendig sind. Um solche gravierenden Änderungen am etablierten Muster zu verabschieden, ist in jedem Fall eine eindringliche Prüfung und Vergleich von Alternativen notwendig. Erst wenn aus Sicht mehrere Experten diese Änderung befürwortet wird, sollte ihr zugestimmt werden. Bis dahin gilt eine unnachgiebige Einhaltung der Vorgaben der hier vorgestellten und bewährten Referenzstruktur.

6 Fallbeispiel zum Nachweis des praktischen Nutzens der Struktur

Im voran gegangenen Kapitel 5 wurde anhand der Software-Architektur der FAS-Längsführung eine konkrete aus der industriellen Praxis stammende Problemstellung bearbeitet und eine neue Referenzstruktur für die gesamte Fahrerassistenz-Domäne geschaffen. Diese Struktur wurde einerseits subjektiv von Experten (Software-Architekten und Software-Entwicklern) bewertet und dabei eine deutliche Verbesserung attestiert, andererseits wurde mit Hilfe der selbst entwickelten Software-Architekturmetriken auch eine objektive Bewertung vorgenommen. Aufgrund der diskutierten Grenzen dieser Metriken kann jedoch auch dieses objektive Verfahren letztlich nur eine Verbesserung an sich, nicht jedoch eine eindeutig messbare Höhe – und damit einen für die Praxis relevanten Nutzen – der Verbesserung nachweisen.

Diese Lücke soll im vorliegenden Kapitel geschlossen werden und somit den letzten noch fehlenden Nachweis der Höhe der Verbesserung aufgrund der beschriebenen Maßnahmen der Neustrukturierung erbringen. Dazu wird ein ausgewähltes, für die Praxis relevantes Fallbeispiel der Umsetzung eines Serien-Fahrerassistenzsystems durchgeführt. Die genauen Ziele, die Durchführung und die erzielten Ergebnisse werden über die folgenden beiden Abschnitte beschrieben.

6.1 Ziele des Fallbeispiels

Eine häufig verwendete Basis-Konfiguration ist die Darstellung der Geschwindigkeitsregelung mit Bremsfunktion (auch Dynamic Cruise Control (DCC)), da diese bei vielen Fahrzeugen inzwischen zur Serienausstattung gehört (bekanntes Bild 5.2 auf Seite 85). Sobald eine Kundenfunktion zur Serienausstattung zählt und der Kunde demnach kein zusätzliches Geld bezahlt, steigt der Druck auf den Fahrzeughersteller immens an, die Umsetzung dieser Kundenfunktion so kostengünstig wie möglich zu realisieren. Idealerweise können die benötigten Software-Anteile auf einem bereits im Fahrzeug vorhandenen Steuergerät zusätzlich integriert werden. Falls dies nicht möglich ist und eine zusätzliche Plattform benötigt wird, so sollte diese eine möglichst einfache, und damit kostengünstige, Hardware-Ausstattung aufweisen. Der Schlüssel für die kostengünstige Umsetzung ist in jedem Fall so wenig Ressourcen wie möglich zu belegen. Unter Ressourcen sind dabei statischer (Read Only Memory (ROM)) und dynamischer (Random Access Memory (RAM)) Speicherbedarf sowie Laufzeitbedarf zur Ausführung gemeint.

Aufgrund der hohen Relevanz genau dieser Fragestellung wird im Fallbeispiel gegen-

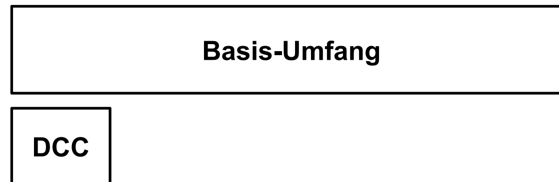

Bild 6.1: Für das Fallbeispiel ausgewählte Ausbaustufe der Software-Architektur

übergestellt, mit welchem Ressourcenaufwand das DCC als eigenständige Kundenfunktion aus dem ursprünglichen Software-System mit der Ausgangs-Software-Architektur, sowie aus dem neuen Software-System mit der neuen Software-Architektur herausgelöst und umgesetzt werden kann. Für das ursprüngliche System existiert dazu bereits eine spezifische Konfiguration, die unter den ungünstigen Randbedingungen dieser Struktur eine in Bezug auf Ressourcenbedarf bestmögliche Umsetzung ermöglicht. Für die neue Struktur ist eine Konfiguration mit den Ausbaustufen der kommunalen Bausteine gemäß Bild 6.1 zu erstellen.

Durch die Gegenüberstellung der zwei oben genannten Umsetzungen soll gezeigt werden, dass die neu entwickelte Software-Architektur von Fahrerassistenzsystemen für relevante Problemstellungen in der Praxis signifikante Vorteile hat. Dies soll die Höhe der Verbesserung und damit den Wert für die Serienentwicklung explizit bestätigen. Mit dem Ergebnis kann und soll ausdrücklich nicht nachgewiesen werden, dass die Architektur „ideal“ beziehungsweise „optimal“ für die Domäne Fahrerassistenz ist.

6.2 Durchführung und Ergebnisse

Das Fallbeispiel soll die Frage beantworten, wieviel Speicherplatz und Laufzeit für beide Varianten der Umsetzung auf einer Rechen-Plattform benötigt werden. Als Plattform wurde in diesem Fall eine gewöhnliche PC-Umgebung eingesetzt und somit kein Serien-Steuergerät oder eingebettetes System. Das liegt einerseits daran, dass zum Zeitpunkt der Durchführung sowohl die Serien-Werkzeugkette zum Erstellen eines Serien-Software-Standes als auch die Infrastruktur zum Einbinden eines Serien-Steuergeräts in eine Testumgebung nicht verfügbar waren. Andererseits existieren am PC etablierte Methoden und Werkzeuge für einfache und zuverlässige Ressourcenmessungen auf die zurückgegriffen werden kann. Da die oberste Prämisse während der gesamten Tätigkeit der Neustrukturierung war, dass Funktionalität nicht verändert werden darf und somit nur strukturelle Änderungen umgesetzt wurden, ist ein Test der DCC-Umsetzung im Fahrzeug nicht zwingend notwendig. Die Einhaltung dieser Prämisse wurde als Experteneinschätzung mehrerer beteiligter Personen bestätigt.

Die Analyse der Umsetzungen erfolgt in den folgenden vier Schritten, wobei die Schritte 1-3 der Ermittlung des Speicherbedarfs dienen und Schritt 4 für die Ermittlung des Laufzeitbedarfs zuständig ist:

1. Analyse des generierten Quell-Codes, dazu zählen:
   - Anzahl der so genannte „Logical Lines of Code“,
   - Anzahl der logischen Operationen (z.B. &&, ==, >=),
   - Anzahl der arithmetischen Operationen (z.B. +, –, *),
   - Anzahl von Programmverzweigungen (z.B. if…else) und
   - Anzahl von Schleifen (z.B. while…do).
2. Analyse der verwendeten Schnittstellentypen bezüglich Verteilung und Anzahl,
3. Größe der generierten Binärdatei und
4. Analyse des Laufzeitbedarfs der Ausführung des Codes.

In der tabellarischen Übersicht der Ergebnisse in Tabelle 6.1 werden für alle Kategorien lediglich die relativen Verbesserungen ausgewiesen. Dies ist für den Nachweis des praktischen Nutzens der erarbeiteten Maßnahmen der entscheidende Zahlenwert. Aus Wettbewerbsgründen können die absoluten Zahlenwerte der einzelnen Größen eines konkreten Automobilherstellers nicht veröffentlicht werden.

Zur Bewertung des Quell-Codes (Schritt 1) wird auf ein frei erhältliches Werkzeug der University of Southern California mit der Bezeichnung „Universal Code Count (UCC)“ [119] für die Programmiersprache C zurückgegriffen. Die zu untersuchenden Dateien, welche durch die Auto-Codegenerierung von ASCET erzeugt wurden, werden zur Analyse in einem frei definierbaren Ordner abgelegt, das Ergebnis der Analyse wird nach Abarbeitung umfangreich in mehreren Text-Dateien ausgegeben. Die für die Analyse im Fallbeispiel relevanten Informationen sind in die Tabelle 6.1 übertragen. UCC wertet die unterschiedlichen Programmierkonstrukte für Variablen in C ebenfalls aus. Da jedoch ASCET eigene Typen definiert und im Code ausschließlich diese benutzt, können die Schnittstellenelemente (Schritt 2) des Fallbeispiels über diese Methode nicht erfasst werden. Daher wurde das selbst entwickelte Werkzeug zur Anwendung der Software-Architekturmetriken eingesetzt. Neben den Kernmetriken (M1-M7) werden – unter anderem aus diesem Grund – auch alle relevanten Schnittstellenelemente in der Software nach Typ unterschieden und gezählt sowie in der Ausgabe-Oberfläche angegeben (M8). Die Ergebnisse des Vergleichs finden sich in Tabelle 6.1. In Schritt 3 soll die Größe der Binärdatei nach dem Compilieren und Linken ausgewertet werden. Dazu wird der Quellcode mit Eclipse für C/C++ [114] in Verbindung mit der GNU Compiler Collection (GCC) [115] in eine ausführbare Datei (.exe) überführt.

Zum Abschluss der Analyse wird die Betrachtung des Laufzeitbedarfs von Schritt 4 durchgeführt. Einer Übersicht in ASCET können die so genannten Prozesse entnommen werden, die in der zyklisch abgearbeiteten Task des Betriebssystems eines möglichen Ziel-Steuergeräts abgearbeitet werden. Die Prozesse entsprechen im Quell-Code C-Funktionen gleichen Namens. Somit kann leicht nachgebildet werden, welche Umfänge zur Simulation der Laufzeit in welcher Reihenfolge abgearbeitet werden müssen. Für die eigentliche Laufzeitmessung wird prinzipiell wie folgt vorgegangen. Es wird ein so genannter Counter verwendet, der bei C-Programmen ermöglicht, eine Systemzeit zu Beginn und Ende der Abarbeitung von Programmteilen auszuwerten und somit die

Tabelle 6.1: Ergebnisse der Analyse des Ressourcenbedarfs beim Fallbeispiel

Bewertungsgegenstand	Relative Änderung
Schritt 1: Analyse des generierten Quell-Codes	
Logical Lines of Code	-47,1 %
Logische Operationen (gesamt)	-51,3 %
&&	-57,9 %
||	-50,2 %
==	-62,3 %
!	-42,4 %
>	-29,8 %
<	-37,9 %
>=	-35,9 %
Arithmetische Operationen / Berechnungen (gesamt)	-26,1 %
+	-33,8 %
–	-15,4 %
*	-27,4 %
/	-20,0 %
»	± 0,0 %
«	-25,0 %
Zuweisungen (=) (gesamt)	-51,0 %
Programmverzweigungen / Bedingungen (gesamt)	-43,2 %
if	-43,9 %
else	-29,8 %
else if	+66,7 %
case	-46,4 %
switch	-37,9 %
?	-41,9 %
Schleifen (gesamt)	-30,8 %
while	-39,1 %
for	+33,3 %
Schritt 2: Schnittstellenelemente (gesamt)	-50,2 %
Messages	-62,5 %
Übergabeparameter	-45,6 %
Applikationsparameter	-61,9 %
Codierparameter	-52,6 %
Globale Variablen	-95,0 %
Lokale Variablen	-33,7 %
Schritt 3: Größe der generierten Binärdatei	-45,2 %
Schritt 4: Laufzeitbedarf	-64,8 %

Laufzeit des Programmteils zu bestimmen. Da die Laufzeit einer einzelnen Abarbeitung der Liste der Funktionen für jeden der verfügbaren Counter zu kurz für eine Messung in der benötigten Auflösung ist, wird über ein Programmierkonstrukt mit einer inneren Schleife eine feste, große Anzahl an Durchläufen vorgegeben und dann über eine äußere Schleife mehrmals durchgeführt. So ist eine zuverlässige Mittelwertbildung und Berechnung der Laufzeit eines einzelnen Durchlaufs möglich. Um einen systematischen Fehler durch einen konkreten und möglicherweise im Testsystem fehlerhaften Counter zu vermeiden, wurden zwei unterschiedliche Counter eingesetzt:

- clock() mit $3 \cdot 10^6$ inneren und 10 äußeren Durchläufen und
- QueryPerformanceCounter() mit $5 \cdot 10^4$ inneren und 10 äußeren Durchläufen.

Bei beiden Countern wurden fünf Messungen mit den oben genannte Parametern durchgeführt. Das Ergebnis in Tabelle 6.1 stammt vom letztgenannten. Es ist anzumerken, dass die Ergebnisse der Laufzeitmessung an einem PC nicht unmittelbar auf ein eingebettetes System übertragbar sind. Dies liegt an einer unterschiedlichen Prozessor-Architektur, Betriebssystem und den verwendeten Compilern. Das Betriebssystem am PC ist grundsätzlich immer in der Lage, laufende Tasks zu unterbrechen. Diese Unterbrechungen sind im Detail nicht vorhersagbar, können durch die hohe Anzahl an Berechnungen über die Schleifen aber gut kompensiert werden. Da die Streuung sowohl zwischen den einzelnen Messungen als auch zwischen den Mittelwerten der verschiedenen Counter in einer Größenordnung von ± 1 % liegen, ist dieser Fehler vernachlässigbar gegenüber der ermittelten relativen Änderung des Laufzeitbedarfs von 64,8 %.

Über die Schritte 1 und 2 lässt sich anhand Tabelle 6.1 feststellen, dass eine Reduktion in einem Bereich 40…50 % erzielt wurde. Dieser Wert wird durch Schritt 3 in guter Näherung bestätigt. In der eingesetzten Methode lässt sich keine scharfe Zuordnung zum statischen und dynamischen Speicher vornehmen. Da jedoch beide Speicherarten grundsätzlich ähnlich wertvoll sind, ist die Angabe der Größenordnung der Gesamtersparnis ausreichend, um eine signifikante Verbesserung zu bescheinigen. Einsparungen am Code führen zwangsläufig auch zu Einsparungen in der Laufzeit, es gibt jedoch in der Regel keine exakte quantitative Korrelation. Die durchgeführte Messung bestätigt das, die Verbesserung des Laufzeitbedarfs liegt in der Größenordnung der Verbesserung des Speicherbedarfs, zeigt jedoch noch einmal eine deutlich höhere Veränderung. Die umgesetzten Maßnahmen wirken sich somit überproportional gut auf die Laufzeit aus.

Das Fazit des Fallbeispiels lautet, dass die vorgeschlagene Software-Architektur für Fahrerassistenzsysteme signifikante und für die Praxis relevante, objektiv messbare Verbesserungen aufweist. Die Empfehlung lautet somit, die Software-Architektur künftig in der Serienentwicklung von Fahrerassistenzsystemen flächendeckend einzusetzen.

7 Zusammenfassung und Ausblick

In diesem abschließenden Kapitel werden die Ergebnisse der Arbeit noch einmal zusammengefasst und ein Ausblick auf weitere lohnenswerte Forschungsaktivitäten im dargelegten Themenfeld gegeben.

7.1 Zusammenfassung der Ergebnisse

Die vorliegende Arbeit beschäftigt sich mit der Entwicklung (der Software-Anteile) von Fahrerassistenzsystemen als konkrete Anwendungsbeispiele von mechatronischen Systemen im Automobil. Entwicklungsmethoden und -prozesse haben zuletzt mit der rasanten Zunahme der Komplexität dieses Themenfelds in einem gleichzeitig anspruchsvollen Entwicklungsumfeld nicht mehr schritthalten können. Einen Schlüssel stellen Software-Architekturen und die explizite Berücksichtigung von nicht-funktionalen Einflüssen dar. In der industriellen Praxis wurden, wie am konkreten Beispiel der Fahrerassistenz Längsführung zu sehen ist, diese beiden Aspekte über Jahre vernachlässigt, so dass für eine effiziente und wettbewerbsfähige Entwicklung unerlässliche Eigenschaften wie Skalierbarkeit und Erweiterbarkeit nicht mehr gegeben sind. Heute bekannte Ansätze im Stand der Forschung weisen dabei Mängel auf, die im Folgenden zusammengefasst werden:

- Keine explizite Modellierung von (realisierungsunabhängigen) Software-Architekturen,
- Geringer Grad an Formalisierung und Durchgängigkeit,
- Fehlender Nachweis von Praxistauglichkeit,
- Keine konkreten Vorgaben zur Strukturierung von Funktions- und Software-Architekturen,
- Fehlende Möglichkeit der objektiven Bewertung von Software-Architekturen und
- Keine auf die Bedürfnisse der Fahrerassistenz angepasste Ansätze.

Für die Arbeit werden aus den obigen Überlegungen sieben Hauptziele abgeleitet, die im weiteren Verlauf des Abschnitts bezüglich der erzielten Ergebnisse diskutiert werden.

> **Ziel (1):** Ausarbeitung von Kriterien für die Modellierung und Strukturierung von Funktions- und Software-Architekturen in der Automobil-Domäne.

In Kapitel 2 werden die unterschiedlichen Ziele der Modellierung und Strukturierung von beiden Architekturarten im Kontext der Automobil-Domäne diskutiert. Das dient

einerseits dem Leser als detaillierte Einführung in die Thematik, andererseits wird damit die These untermauert, dass beide bei der Entwicklung von mechatronischen und softwaredominanten Systemen im Automobil wichtig und explizit einzusetzen sind.

> **Ziel (2):** Entwicklung einer Methode zur Modellierung und automatisierten Weiterverarbeitung von komplexen Software-Architekturen in der Automobil-Domäne am Beispiel von Fahrerassistenzsystemen.

Als eines der wesentlichen Ergebnisse dieser Arbeit wird in Kapitel 4 die Abstrakte Automotive Software-Architektur (ABSOFA) eingeführt. Sie legt als UML-Erweiterung per Profil eine eigenständige Methode zur Modellierung von Software-Architekturen fest. Kerngedanke ist eine möglichst lange Trennung von fachlicher Logik und technischer Realisierung und damit das Erreichen hoher Wiederverwendung. Um die ABSOFA herum werden weitere Methoden, Prozessschritte und Arbeitsergebnisse definiert, die eine automatisierte und formale Weiterverarbeitung der Software-Architektur im Rahmen des Vorgehensmodells ermöglichen. Hierbei sind insbesondere das zentrale Datenmodell, die objektive Software-Architekturbewertung und Modelltransformationen zwischen unterschiedlichen Abstraktions-Ebenen sowie Artefakten zu nennen.

> **Ziel (3):** Entwicklung eines automatisiert anwendbaren Bewertungsverfahrens für Software-Architekturen in der Automobil-Domäne unter Verwendung von objektiven, das heißt quantifizierten, Software-Architekturmetriken.

Die Methode zur Software-Architekturbewertung wird wie erwähnt im Kontext der ABSOFA in Abschnitt 4.3 entwickelt. Sie definiert ein universell gültiges Qualitätsmodell mit Qualitätskriterien und Qualitätsattributen. Für die Anwendung in der Automobil-Domäne werden spezifisch gültige Definitionen vorgenommen. Für insgesamt acht dieser Qualitätsattribute können quantifizierte und damit objektive Metriken definiert werden, die automatisiert auf Software-Architekturen im zentralen Datenmodell angewendet werden. Es liegt eine vollständige Umsetzung der Metriken in einem Werkzeug vor.

> **Ziel (4):** Analyse und Dokumentation des heutigen Vorgehensmodells zur Entwicklung sowie der Funktions- und Software-Architektur der Fahrerassistenzsysteme der Längsführung und Herausarbeiten der Schwachstellen.

Die Ergebnisse der Analyse des heutigen Vorgehensmodells zur Entwicklung befinden sich in Kapitel 3. Neben den dabei ermittelten Schwachstellen wie des zu frühen Übergangs auf die technische Realisierung, der unzureichenden Berücksichtigung nicht-funktionaler Einflussgrößen und der generell fehlenden, expliziten Modellierung der Software-Architektur werden auch Ziele zur Anpassung und damit Verbesserung des Vorgehensmodells definiert, die anschließend durch die ABSOFA konkret gelöst werden. Der zweite Teil des Ziels (4) wird ausführlich in Kapitel 5.2 behandelt und bereitet so die systematische Neustrukturierung vor.

Ziel (5): Sicherstellung der Integrierbarkeit aller neu erarbeiteten Methoden und Prozessschritte in das Vorgehensmodell.

Da ausdrücklich kein gänzlich neues Vorgehensmodell entworfen werden soll, müssen die Verbesserungen als optionale Änderungen und Zusätze in das bestehende Vorgehensmodell integriert werden. Diese Vorgabe wird in den Abschnitten 3.2 und 3.3 bei der Erarbeitung der Zielsetzung sowie bei Ausarbeitung der konkreten Lösungen im Rahmen der ABSOFA berücksichtigt.

Ziel (6): Erarbeitung von konkreten Kriterien zur Strukturierung von Software-Architekturen von Fahrerassistenzsystemen der Längsführung.

Ziel (7): Durchführung einer Neustrukturierung der Software-Architektur von Fahrerassistenzsystemen der Längsführung unter Anwendung der zuvor entwickelten Methoden.

Bei der Bearbeitung dieser beiden Ziele wird in umgekehrter Reihenfolge vorgegangen. Über die Erkenntnisse im Rahmen der durchgeführten Neustrukturierung der Software-Architektur von realen Fahrerassistenzsystemen der Längsführung in Kapitel 5 können allgemeingültige Kriterien zur Strukturierung eben dieser Systeme ausgewiesen werden. Über die getroffene Zielsetzung (6) hinausgehend werden die strukturellen Vorgaben ebenfalls im Kontext Fahrerassistenz im Allgemeinen diskutiert. Das Ergebnis ist, dass die Kriterien und Muster übergreifend für die gesamte Domäne Fahrerassistenz anwendbar sind. Sie werden als Referenzstruktur vorgeschlagen. Für die Neustrukturierung werden alle Methoden und eingeführten Erweiterungen beziehungsweise Neuerungen des Vorgehensmodells eingesetzt und somit evaluiert. Die so erfolgte Bestätigung ihrer Anwendbarkeit leistet darüber hinaus den noch fehlenden Beitrag zu Ziel (5) bezüglich des Nachweises der Sicherstellung der Integrierbarkeit in das bestehende Vorgehensmodell.

Ziel (8): Nachweis der Wirksamkeit und des Nutzens der neuen Struktur anhand der Gegenüberstellung der Implementierungen eines geeigneten Fahrerassistenzsystems aus der alten und der neuen Struktur.

Abschließend wird in Kapitel 6 ein objektiver Nachweis erbracht, dass die neu entwickelte Software-Architektur von Fahrerassistenzsystemen der Längsführung und damit auch der generischen Referenzstruktur der Domäne Fahrerassistenz einen Praxisnutzen aufweist. Dazu wird ein reales Fallbeispiel in Form der Geschwindigkeitsregelung mit Bremsfunktion betrachtet und die Umsetzungen dieser Kundenfunktion aus der alten und der neuen Struktur bezüglich Ressourcenverbrauch verglichen. Das Ergebnis zeigt eine signifikante Reduktion von Speicher- und Laufzeitbedarf zwischen 40 und 50 %.

Als Gesamtfazit der Arbeit lassen sich zwei Kernbotschaften festhalten. Das ist zum einen die Feststellung, dass die entwickelten Methoden praktikabel in der Anwendung sind und in der industriellen Praxis an realen Beispielen wirksam helfen, effiziente Software-Strukturen zu erarbeiten. Zum anderen kann konstatiert werden, dass die vorgeschlagene Software-Architektur für Fahrerassistenzsysteme signifikante und für die

Praxis relevante, objektiv messbare Verbesserungen aufweist.

Die Arbeit leistet über die adressierten, oben genannten Hauptziele hinaus einen wichtigen Beitrag die Entwicklung von softwaredominanten Systemen formal, durchgängig und effizient zu gestalten.

7.2 Ausblick

Im vorliegenden letzten Abschnitt soll ein Ausblick auf mögliche weitere Aktivitäten im betrachteten Themenumfeld im Allgemeinen sowie zur Fortführung der Ansätze in dieser Arbeit im Speziellen gegeben werden. Dabei wird auf die folgenden vier Aspekte eingegangen.

Vorgehensmodell zur Entwicklung

Das existierende Vorgehensmodell zur Entwicklung, das auf dem bekannten V-Modell basiert, wurde bewusst nur stellenweise angepasst und erweitert. Alle Änderungen beziehen sich auf den linken „Ast“ und damit die Entwicklung. Zum jetzigen Zeitpunkt wurden keine expliziten Überlegungen angestellt, ob sich im rechten Ast bezüglich Aspekten zu Test und Absicherung besondere Herausforderungen stellen und wie genau die neuen Prozessschritte und Arbeitsergebnisse effektiv und effizient abgesichert werden können. Erstrebenswert wäre es hier ebenfalls Formalismen und Automatismen analog zum neuen linken Ast einzusetzen.

Software-Architekturbewertung

Die entwickelte Methode zur Software-Architekturbewertung sowie die konkreten Metriken helfen bereits viel beim iterativen Entwurf und der iterativen Optimierung von Software-Architekturen. Folgende Aktivitäten bieten sich an, um die für die Entwurfstätigkeit so wichtige Architekturbewertung weiter zu verbessern. Das Qualitätsmodell ist bereits universell einsetzbar, die darunter liegenden Qualitätskriterien und -attribute wurden jedoch für die Automobil-Domäne zum Teil spezifisch definiert. Je nach Einsatzzweck ist zu prüfen, ob geänderte, spezifische Definitionen notwendig sind. Damit ist zum einen das Anpassen an ganz andere Domänen, zum anderen das Spezialisieren an Fachbereiche innerhalb der Automobil-Domäne gemeint. Gleiches gilt für die Ausprägung der Metriken selbst. Bisher orientieren sich die Metriken an allgemeinen Richtlinien zur Strukturierung von Software-Architektur in der Automobil-Domäne. Durch ihren Einsatz konnten im Rahmen der Neustrukturierung der Software-Architektur der Fahrerassistenzsysteme der Längsführung spezifische Richtlinien und Muster für diese Teildomäne gefunden werden. Für Folgeprojekte könnten die Metriken an diese Erkenntnisse angepasst werden. In anderen Teildomänen ist ein ähnliches Vorgehen und Spezialisieren denkbar. Wie bereits erläutert, eignen sich die Metriken vor allem für einen relativen Vergleich zweier konkurrierender Entwürfe. Durch häufigen Einsatz der Metriken für ähnli-

che Problemstellungen können die absoluten Werte der einzelnen Metriken besser eingeschätzt werden. So können die Metrikergebnisse auch ohne Vergleichs-Architektur direkt als Bewertung der Güte verwendet werden. Gleichzeitig kann ein Fein-Parametrieren der Faktoren erfolgen, um die Sensitivitäten der Metriken anzugleichen.

Modelltransformationen

Modelltransformationen unterstützen beim Ziel eines formalen, durchgängigen und effizienten Vorgehensmodells. Im Rahmen dieser Arbeit wurde das universelle Grundgerüst dazu als Methode etabliert und ausgewählte, konkrete Transformationen in einem Werkzeug umgesetzt. Dabei wurde sich vor allem an den für die Durchführung der Neustrukturierung notwendigen Teilschritten orientiert. Um alle (Tailoring-)Varianten des Vorgehensmodells abzubilden, ist die Umsetzung weiterer, konkreter Transformations-Regelsätze notwendig. Die Gesamtheit aller Transformationen sowie die Information, welche bereits umgesetzt sind, kann Bild A.1 auf Seite 133 entnommen werden. Besonders wichtig sind zunächst die Transformationen zwischen ABSOFA und dem Zwischenprodukt sowie die Überführung von der Abstrakten in die Plattformabhängige Sicht. Neben den im Bild dargestellten Schritten kann es ebenfalls nützlich sein, eine automatisierte Partitionierung der Architekturbausteine auf Steuergeräte eines konkreten Bordnetzes einer Fahrzeugvariante mit Hilfe einer solche Transformation zu unterstützen. Bisher wurde diese Information als gegeben betrachtet. Der methodische Rahmen unterstützt ein solches Vorgehen, zusätzlich müssen automatisierbare Kriterien und eine standardisierte Ablage der Informationen der Technischen Architektur an den Transformations-Ansatz übergeben werden.

Zukunftsfähigkeit der Software-Struktur für hochautomatisiertes Fahren

Die erarbeitete Software-Struktur wurde in Kapitel 5.4 bereits im Kontext der gesamten Domäne Fahrerassistenz unter Einbeziehung der heutigen Generationen an Systemen mit assistierendem und teilautomatisiertem Charakter diskutiert und für gut befunden. Für die zu erwartenden Ausprägungen der nächsten Jahre wurde somit eine stabile Architektur erarbeitet. In einem weiter gefassten Fokus mit Ersteinsätzen in Serienproduktion zwischen 2020 und 2025 sind hochautomatisierte Fahrerassistenzsysteme zu erwarten. Dabei sind große Auswirkungen auf Logische und Technische Architektur abzusehen, da aufgrund der geänderten Prämisse bezüglich Fahreraufmerksamkeit ein Paradigmenwechsel bevorsteht. Beispiele sind redundante Ausführungen von Sensoren, Plattformen und Aktoren sowie neuen Methoden zur Entwicklung und Absicherung. Darüber hinaus wird auch der Gesetzgeber auf aktuell noch nicht absehbare Art und Weise Einfluss nehmen. Die Fahrerassistenz-(Software-)Architektur als Ergebnis dieser Arbeit muss kontinuierlich an den neuen Herausforderungen gemessen werden. Aufgrund der Schwere der Änderungen der Rahmenbedingungen ist es möglich, dass eine evolutionäre Weiterentwicklung nicht ausreicht sondern revolutionäre Ansätze notwendig werden.

Glossar

Abstrakte Software-Architektur

Eine abstrakte Software-Architektur führt den Grundgedanken der Abstraktion, der allen Software-Architekturen gemein ist, konsequent weiter. Sie abstrahiert völlig von Technologien und Werkzeugen, Plattformen, Software-Verteilung und Varianten. Eine hierzu konkret vorgeschlagene Architektur-Beschreibungssprache ist die Abstrakte Automotive Software-Architektur (ABSOFA).

Anwendungs-Software

Synonym zu High-Level-Software.

Applikations-Parameter

Applikations-Parameter werden verwendet, um eine Software derivatspezifisch konfigurieren zu können. Die Software selbst ist vom Aufbau und Inhalt her allgemeingültig. An den notwendigen Stellen mit parametrierter Varianz sind alle potentiell benötigten Parameter persistent abgelegt. In jedem Fahrzeug wird per Konfigurationsschalter (Codier-Parameter) genau der dem Fahrzeug zugeordnete Parameter verwendet, die übrigen bleiben ungenutzt. Applikations-Parameter sind 0- (Skalar), 1- (Feld; auch: Kennlinie) oder 2-dimensional (Matrix; auch: Kennfeld), die hinterlegten Werte sind unabänderlich in die Software integriert.

Architektur

Unter Architektur wird die innere Struktur eines (informations-)technischen Systems verstanden. In der Betrachtung stehen insbesondere die zur Ziel-Erfüllung des Systems beteiligten Systembausteine und deren Wechselwirkungen untereinander.

Architektur-Beschreibung

Eine Architektur-Beschreibung ist eine Menge von Modellen und Sichten, die eine Architektur für unterschiedliche Zwecke und Stakeholder dokumentieren.

Architektur-Beschreibungssprache

Eine Architektur-Beschreibungssprache ist eine Notation für die formale und eindeutige Beschreibung von Software-Architekturen.

Architektur-Bewertung

Im Rahmen einer Architektur-Bewertung wird eine Architektur bezüglich ihrer Eignung für einen vorgesehenen Bestimmungszeck bewertet. Dies kann grundsätzlich anhand subjektiver (qualitativer) oder objektiver (quantitativer) Kriterien durchgeführt werden. Architektur-Bewertungen dienen typischerweise der Entscheidung zwischen zwei oder mehreren Entwürfen oder der gezielten Weiterentwicklung eines vorhandenen Entwurfs. Im Rahmen dieser Arbeit stellt die objektive Software-Architekturbewertung einen Sonderfall der Architektur-Bewertung dar.

Architektur-Muster

Ein Architektur-Muster ist eine bewährte, generische Lösung für ein wiederkehrendes Problem im Architektur-Entwurf. Die Lösung enthält alle benötigten Architekturbausteine sowie deren Verantwortlichkeiten und Wechselwirkungen untereinander.

Architekturbaustein

Der Begriff Architekturbaustein wird als allgemeiner Platzhalter für eine Vielzahl von Bausteintypen verwendet, welche je nach Hierarchie-Ebene in Funktions- oder Software-Architekturen Anwendung finden.

Architekturstil

Architekturstile sind globale, übergeordnete Strukturierungsprinzipien, die auf die gesamte Software-Architektur oder einen wesentlichen Teil davon angewendet werden.

Artefakt

Unter einem Artefakt wird ein Produkt im Sinne eines eigenständigen Arbeitsergebnisses verstanden, welches als Zwischen- oder Endergebnis aus einem (Arbeits-) Prozess oder (Arbeits-)Prozessschritt resultiert.

Basis-Software

Synonym zu Low-Level-Software.

Baukasten

Baukästen haben das Ziel, zwischen mehreren verwandten Produkten Synergien in Form von Wiederverwendung zu schaffen. Dies bedeutet, dass identische Gleichteile in diesen Produkten verwendet werden. Gleichteile können hierbei von unterschiedlichster Beschaffenheit sein: z.B. mechanische Komponenten wie Karosserieteile, Fahrwerksteile, Antriebskomponenten; elektrisch/elektronische Komponenten wie Bordnetze, Steuergeräte, Schalter, Software. Die Wiederverwendung kann dabei sowohl durch identischen Einsatz eines komplexen Gesamtsystems als auch durch

das individuelle Zusammenstellen von Teil-Systemen bzw. Komponenten zu einem übergeordneten Ganzen aus dem Baukasten heraus geschaffen werden.

Blackbox

Eine Blackbox ist ein System, das eigenständig oder Teil eines Gesamtsystems sein kann. Die Blackbox zeichnet sich dadurch aus, dass der innere Aufbau dem Verwender nicht bekannt ist. Bekannt sind lediglich alle Ein- und Ausgangsschnittstellen sowie das angestrebte (Übertragungs-)Verhalten des Systems. Verwandt zum Begriff der Blackbox sind die Begriffe Whitebox und Greybox.

Bordnetz

Das Bordnetz eines Kraftfahrzeugs besteht aus der Gesamtheit aller verbauter Elektrik-/Elektronik-Komponenten. Im engeren Sinne dieser Arbeit gefasst sind alle Verarbeitungseinheiten (Steuergeräte, Sensoren, Aktoren) und deren Kommunikationsverbindungen Teil des Bordnetzes. Dabei bilden Steuergeräte, welche über Bus-Verbindungen kommunizieren, den Schwerpunkt.

Bus

Ein Bus ermöglicht Kommunikation zwischen zwei oder mehreren Partnern über eine gemeinsame Datenleitung. Er ersetzt somit proprietäre Punkt-zu-Punkt-Verbindungen durch eine standardisierte Übertragung und senkt die Komplexität des Verdrahtungsaufwands. Neben der elektrischen Verbindung wird ein Bus ebenfalls in Hinblick auf das Format der Datenübertragung (Protokolle, Schichten) standardisiert. Die gängigsten Bus-Systeme in Kraftfahrzeuge sind LIN, CAN und FlexRay. Einen Überblick liefern Zimmermann et al., 2006 [130].

Bus-Signal

Unter einem Bus-Signal wird eine Schnittstelle verstanden, die bezogen auf ein Steuergerät als externe Kommunikation anzusehen ist und über einen Bus versendet oder empfangen wird. Dem gegenüber stehen Schnittstellen, welche nur innerhalb eines Steuergeräts ausgetauscht werden und nach außen nicht sichtbar und notwendig zu senden bzw. zu empfangen sind.

Codier-Parameter

Ähnlich wie Applikations-Parameter dienen auch Codier-Parameter der derivatspezifischen Konfiguration von Software. Codier-Parameter fungieren einerseits als Schalter zur Auswahl der richtigen Applikations-Parameter, andererseits kann der Codier-Parameter selbst ein spezifischer Wert einer bestimmten Konfiguration sein, den die Software direkt verwendet. Codier-Parameter sind in den meisten Fällen Skalare. Sie können am Bandende oder bei der Handelsorganisation ohne Änderung

der hinterlegten Software einzeln verstellt werden, d.h. sie sind nicht unabänderlich sondern für den autorisierten Zugriff jederzeit änderbar im nicht-flüchtigen Speicher abgelegt.

Daten-Bordnetz

Synonym zu Bordnetz.

Datenbus

Synonym zu Bus.

Derivat

Derivate (von lateinisch derivare: ableiten) stellen ähnliche/verwandte Untertypen eines gemeinsamen Fahrzeugtyps dar. Für den Kunden sichtbar unterscheiden sich Derivate oft in bestimmten Aspekten der äußeren Geometrie der Karosserie. Seitens der Fahrzeughersteller werden für Derivate viele Gleichteile bzw. Teile aus gemeinsamen Baukästen verwendet. Bordnetze von Derivaten sind typischerweise identisch.

Domänen-Architektur

Domänen-Architekturen stellen Architekturen speziell angepasst an (zumeist wiederkehrende) Problemstellungen konkreter Domänen, d.h. Fachgebiete, dar. Für die Beschreibung und Dokumentation derartiger Architekturen finden oft so genannte Domain Specific Languages (DSL) Anwendung.

Eingebettetes System

Eingebettete Systeme sind informationsverarbeitende Systeme, die in ein größeres Produkt wie beispielsweise Kraftfahrzeuge, Unterhaltungs- oder Telekommunikationsgeräte integriert sind, und die normalerweise nicht direkt vom Benutzer wahrgenommen werden. Häufig sind eingebettete Systeme mit der physikalischen Umwelt verbunden. Sensoren sammeln Informationen über die Umgebung und Aktoren nehmen Einfluss auf die Umwelt. Über eine oder mehrere Verarbeitungseinheiten, in Kraftfahrzeugen meist als Microcontroller in Steuergeräten ausgeführt, erfolgt die Verarbeitung der Informationen.

Entwicklungsprozess

Ein Entwicklungsprozess beschreibt alle Rollen, die durchzuführenden Arbeitsschritte, die Wechselwirkungen und Übergänge zwischen diesen sowie die zu jedem Schritt zu erbringenden Arbeitsergebnisse in einem bewusst abgegrenzten Betrachtungsumfang einer Entwicklungsaufgabe.

Fahrerassistenzsystem

Fahrerassistenzsysteme unterstützen den Fahrer bei seiner Ausübung der Fahraufgabe. Die Unterstützung kann dabei rein informativer Art (Anzeigen, Töne) bis hin zu einer teilweisen oder vollständigen Übernahme von Fahrzeugführungsaufgaben sein. Im Rahmen dieser Arbeit stehen aktive Fahrerassistenzsysteme aus dem Bereich Komfort im Fokus, die als mechatronische Systeme ausgeführt sind und Eingriffe in die Längsführung zur Unterstützung und Entlastung des Fahrers durchführen können. Der Fahrer behält dabei immer die Verantwortung der Fahraufgabe und hat jederzeit die Möglichkeit zur Übersteuerung des Systems.

Feature

Synonym zu Kundenfunktion.

Frontloading

Unter Frontloading wird die bewusste Stärkung von frühen Aktivitäten innerhalb eines (Arbeits-)Prozesses verstanden. Dies kann einerseits durch das Verschieben von Arbeitsschritten auf eine frühere Phase als es aus der Abfolge der Prozessschritte mindestens gefordert wird, erfolgen. Häufiger ist mit Frontloading gemeint, dass bestimmte Arbeitsschritte zu einem frühen Zeitpunkt des Prozesses freiwillig zusätzlich durchgeführt werden, um damit im späteren Verlauf notwendige Arbeitsleistung in größerer Menge als die zuvor eingesetzte einzusparen. Die frühe Phase wird somit „aufgeladen“.

Funktion

Funktionen dienen der Darstellung eines gewünschten, kundenerlebbaren Verhaltens im Fahrzeug. Es gibt einen definierten Ursache-Wirkungs-Zusammenhang zwischen Ein- und Ausgangsgrößen. Eine einzelne Funktion kann gleichzeitig eine Kundenfunktion sein. Eine Kundenfunktion kann jedoch auch durch das Zusammenwirken einer Vielzahl von Funktionen gebildet werden.

Funktionale Anforderung

Eine funktionale Anforderung beschreibt eine geforderte Eigenschaft des betrachteten Gegenstandes, die durch den Kunden in der bestimmungsgemäßen Verwendung unmittelbar erlebbar ist. Dazu zählen unter anderem Ausprägungen einer Teilfunktion (z.B. konkretes Annäherungsverhalten der Folgefahrtregelung ACC) sowie das Anzeige und Bedienkonzept (z.B. Anzeige „Vorderfahrzeug erkannt“ bei aktiver Folgefahrtregelung ACC; Einstufige Aktivierung ACC).

Funktions-Architektur

Funktions-Architektur beschreibt die Struktur von Funktionen, welche die Architekturbausteine, deren extern sichtbare Eigenschaften sowie die Schnittstellen zwi-

schen diesen umfasst. Funktions-Architekturen abstrahieren von Details und sind hierarchisch aufgebaut.

Funktionsnetz

Synonym zu Funktions-Architektur.

Greybox

Eine Greybox ist in Abgrenzung zur Blackbox bzw. Whitebox ein System oder Teil eines Gesamtsystems, das Eigenschaften beider zuvor angesprochener Begriffe aufweist. Typischerweise ist dem Anwender anders als bei einer Blackbox der innere Aufbau des Systems zumindest teilweise bekannt. Dieser Anteil kann je nach spezifischen Anwendungsfall klein oder groß im Verhältnis zur Gesamtfunktionaliät sein und wird durch die Begrifflichkeit Greybox nicht genauer vorgegeben.

High-Level-Software

Die High-Level-Software ist der Anteil der insgesamt eingesetzten Software, welche die für den Kunden erlebbare Funktionalität abbildet. High-Level-Software zur Realisierung einer bestimmten Kundenfunktion kann auf ein oder mehrere Steuergeräte verteilt sein, während sich innerhalb eines Steuergeräts High-Level-Software befinden kann, die teilweise oder gesamthaft eine oder mehrere Kundenfunktionen realisiert. Der High-Level-Software gegenüber steht die Low-Level-Software.

Kundenfunktion

Eine Kundenfunktion ist eine für den Kunden im Fahrzeug einzeln erlebbare Funktionalität. Bei Fahrerassistenzsystemen ist eine Kundenfunktion häufig als einzeln bestellbare Sonder-Austattung ausgeprägt. Dies ist jedoch kein notwendiges Kriterium, da die Bündelung zu Ausstattungspaketen aufgrund der zunehmenden Anzahl an einzelnen Kundenfunktionen an Bedeutung gewinnt. Eine Kundenfunktion zeichnet sich durch ein reproduzierbares Verhalten in wiederkehrenden Fahr-Situationen und eine dieser Funktion eindeutig zuordenbare Interaktion zwischen Mensch und Maschine aus.

Leitidee

Synonym zu Architekturstil.

Logische Architektur

Die Logische Architektur schlägt die Brücke zwischen der Nutzer-Architektur und der Technischen Architektur. Sie enthält einen detaillierten Entwurf über alle logisch – d.h. inhaltlich und eigenständig – notwendigen Bausteine sowie deren Verhalten und Wechselwirkungen (Schnittstellen), um das gewünschte Verhalten des

Gesamtsystems zu erreichen. Die logische Architektur abstrahiert dabei jedoch völlig von konkreten, technischen Realisierungen wie beispielsweise physikalischen Wirkprinzipien, Sensor- und Aktorbauformen, Bordnetzen und Plattformen. Daher ist sie universell und übergreifend für alle Produktlinien und Fahrzeugtypen gültig.

Logisches Signal

Logische Signale werden in Funktions- und abstrakten Software-Architekturen verwendet. Sie zeichnet aus, dass sie einen Signalfluss zwischen Architekturbausteinen repräsentieren und somit Wechselwirkungen kenntlich machen können, dabei aber bewusst von Realisierungsdetails (z.B. Bitlänge, Datentyp) abstrahieren.

Low-Level-Software

Die Low-Level-Software ist der Anteil der insgesamt eingesetzten Software, welche als notwendige Infrastruktur verwendet wird, um eine High-Level-Software zu befähigen. Die Low-Level-Software selbst enthält keine für den Kunden erlebbare Funktionalität. Häufige Bestandteile von Low-Level-Software sind: Betriebssystem, Treiber, Adapter. Mit dem Begriff Low-Level-Software wird typischerweise die einem konkreten Steuergerät zuordenbare Low-Level-Software assoziiert. Zur Realisierung einer Kundenfunktion werden häufig mehrere Steuergeräte mit jeweils eigenen Anteilen an Low-Level-Software benötigt.

Längsführung (Teilbereich der Fahrzeugführung)

Unter (Fahrzeug-)Längsführung wird jeder durch den Fahrer- oder ein technisches System vorgenommene Eingriff verstanden, der die Bewegungsrichtung in x-Richtung (definiert durch das Fahrzeugkoordinatensystem nach DIN 70000, 1994 [33]) beeinflusst. Typischerweise erfolgt ein solcher Eingriff über den Antriebsstrang oder die Bremsanlage.

Mechatronisches System

Mechatronische Systeme (Kunstwort „Mechatronik“ zusammengesetzt aus „Mechanik“ und „Elektronik“) vereinigen mehrere klassische Teilsysteme interdisziplinär zu einem komplexen Gesamtsystem. Sie zeichnen sich durch das Zusammenwirken von Sensoren, Aktoren, informationsverarbeitender Einheit und einem Grundsystem aus. Die Vereinigung kann dabei funktional und/oder räumlich sein.

Meta-Modell

Ein Meta-Modell ist ein Modell, das eine Menge anderer Modelle definiert, die als Instanzen des Meta-Modells bezeichnet werden. Das Meta-Modell steht über einen höheren Abstraktionsgrad in Beziehung zu den konkreten Modellen. Es macht Aussagen über deren Aufbau nicht jedoch über deren Gegenstandsbereich.

Metrik

Siehe Software-Architekturmetrik.

Modell

Ein Modell bildet einen Gegenstandsbereich der Wirklichkeit unter einem oder mehreren Aspekten ab. Dabei werden nicht alle Informationen und Attribute des Originals erfasst sondern nur diejenigen, die für das Modell im Kontext der Anwendung relevant sind, das Modell abstrahiert.

Modellbasierte Software-Entwicklung

In der modellbasierten Software-Entwicklung werden grafische Modelle eingesetzt. Diese haben in der Regel nur dokumentierenden Charakter im Sinne einer gedanklichen Verbindung zwischen Modell und Implementierung. Es existiert keine formale Beziehung zur Sicherstellung der Konsistenz oder zur Automatisierung von Übergängen zwischen den Arbeitsprodukten. Sinngemäß aus Stahl et al., 2007 [108].

Modellgetriebene Software-Entwicklung

In der modellgetriebenen Software-Entwicklung werden grafische Modelle eingesetzt. Diese sind gleichzusetzen mit der Implementierung, da diese automatisch aus dem Modell erzeugt wird. Die Konsistenz zwischen den Arbeitsprodukten ist zu jedem Zeitpunkt sichergestellt. Es können darüber hinaus mehrere Modellierungsebenen (Abstraktionen) existieren, die über formale Beschreibungen zueinander in Beziehung stehen. Sinngemäß aus Stahl et al., 2007 [108].

Modelltransformation

Überführung eines Modells in ein anderes nach festgelegten Transformations-Regeln. Die Beschreibung der beteiligten Modelle und der Regeln kann formal oder semi-formal erfolgen. Zu unterscheiden sind automatische, semi-automatische und manuelle Modelltransformationen.

Nicht-funktionale Anforderung

Eine nicht-funktionale Anforderung beschreibt eine geforderte Eigenschaft des betrachteten Gegenstandes, die durch den Kunden in der bestimmungsgemäßen Verwendung nicht unmittelbar oder gar nicht erlebbar ist. Dazu zählen unter anderem geforderte Eigenschaften in der Entwicklung (z.B. Kompatibilität mit dem AUTOSAR-Standard), Inbetriebnahme (z.B. keine Sensorkalibrierung am Bandende notwendig), Wartung (z.B. Diagnostizierbarkeit von Fehlern) und Außerbetriebnahme (z.B. Recyclingfähigkeit von Bauteilen in einem Steuergerät).

Nutzer-Architektur

Unter der Nutzer-Architektur wird die Sichtweise auf das Gesamtsystem Fahrzeug verstanden, wie es der Nutzer, das heißt der Fahrer beziehungsweise Kunde, wahrnimmt. Gegenstand der Nutzer-Architektur sind alle direkt mit dem Nutzer interagierenden technischen Elemente sowie alle für den Nutzer direkt erlebbaren Merkmale und Eigenschaften des Gesamtsystems.

Perspektive

Synonym zu Standpunkt.

Plattform

Unter einer Plattform wird in diesem Kontext eine konkrete Hardware als Ablaufumgebung einer Software verstanden.

Plattformabhängige Software-Architektur

Eine plattformabhängige Software-Architektur repräsentiert eine Software, die in diesem Aufbau und Konfiguration für eine konkrete Implementierung in einer konkreten Plattform, auf die sich die Architektur bezieht, gültig ist.

Produktlinie

Eine Produktlinie fasst mehrere konkrete Fahrzeugtypen mit ähnlichen Eigenschaften (z.B. Bauart, Produktionsstandorte, Bordnetz-Topologie) sowie alle von diesen abgeleiteten Derivate zu einer gemeinsamen Organisationseinheit zusammen. Innerhalb einer Produktlinie werden häufig Gleichteile, identische Bordnetz-Konzepte oder gemeinsame Baukästen verwendet. Ziel ist die Schaffung von Synergien sowie die vereinfachte Organisation von Fahrzeugtypen innerhalb der gesamten Modellpalette.

Querführung (Teilbereich der Fahrzeugführung)

Unter (Fahrzeug-)Querführung wird jeder durch den Fahrer- oder ein technisches System vorgenommene Eingriff verstanden, der die Bewegungsrichtung in y-Richtung (definiert durch das Fahrzeugkoordinatensystem nach DIN 70000, 1994 [33]) beeinflusst. Typischerweise erfolgt ein solcher Eingriff über die Lenkung und/oder seitenindividuelle Bremseingriffe.

Refactoring

Refactoring ist der Prozess, ein Software-System so zu verändern, dass das externe Verhalten nicht geändert wird, sondern eine bessere innere Struktur (Software-Architektur bzw. implementierter Code) erzielt wird.

Reverse Engineering

Reverse Engineering bezeichnet den Prozess der Analyse eines Systems wegen zweier Ziele: 1. der Identifizierung der Komponenten, aus denen das System besteht, und der Beziehungen zwischen diesen Komponenten; 2. der Herstellung von Repräsentationen des Systems in einer anderen Form oder auf einer höheren Abstraktions-Ebene.

Schnittstelle

Eine Schnittstelle ist ein Vertrag zwischen zwei Architekturbausteinen: einem Baustein, der eine bestimmte Funktionalität benötigt und einem Baustein, der diese Funktionalität bereitstellt.

Sender-Receiver-Pattern

„The sender-receiver pattern gives solution to the asynchronous distribution of information, where a sender distributes information to one or several receivers. The sender is not blocked (asynchronous communication) and neither expects nor gets a response from the receivers (data or control flow), i.e. the sender just provides the information and the receivers decide autonomously when and how to use this information. It is the responsibility of the communication infrastructure to distribute the information. The sender component does not know the identity or the number of receivers to support transferability and exchange of AUTOSAR Software Components.“ [1]

Sicht

Komplexe Architekturen können oft nicht in Gänze in einer einzigen Darstellung erfasst werden. Aus diesem Grund wird eine Architektur in verschiedenen Sichten dargestellt, welche eine Einschränkung der Informationen vornehmen und von bestimmten Details und Aspekten abstrahieren. Jeder Standpunkt verwendet eine bestimmte Untermenge aller vorhandener Sichten.

Signal

Signale sind in Funktions- und Software-Architekturen in der Automobil-Domäne die am häufigsten anzutreffende Form der Schnittstelle. Dabei handelt es sich um die Bereitstellung von Zahlenwerten, die gemäß des Schnittstellenvertrages befüllt bzw. interpretiert werden und unterschiedlichsten Zwecken dienen können. Signale sind 0- (Skalar), 1- (Feld) oder 2-dimensional (Matrix).

Software-Architektur

Software-Architektur beschreibt die Struktur eines Software-Systems, welche die Architekturbausteine, deren extern sichtbare Eigenschaften sowie die Schnittstellen zwischen diesen umfasst. Software-Architekturen abstrahieren von Details und sind hierarchisch aufgebaut.

Software-Architekturbewertung

Siehe Architektur-Bewertung.

Software-Architekturmetrik

Eine Metrik im Allgemeinen ist ein zahlenmäßiges Gütemaß zur Bewertung bestimmter Eigenschaften oder Kriterien unterschiedlichster Anwendungsbereiche. Eine Software-Architekturmetrik im Speziellen ist somit eine objektivierte Kennzahl zur Bewertung der Qualitätsattribute von Software-Architekturen.

Software-Architekturmuster

Siehe Architektur-Muster.

Software-Komponente

Eine Software-Komponente ist ein Baustein zur Komposition von Software-Systemen mit Qualitätsattributen und explizit definierten Schnittstellen, die bereitgestellt, benötigt oder zur Konfiguration verwendet werden.

Software-Partitionierung

Die Software-Partitionierung legt fest, welche der im Rahmen einer Software-Architekturerstellung definierten Architekturbausteine auf welche Plattformen, die in der technischen Architektur zur Verfügung stehen, verteilt werden.

Software-Verteilung

Synonym zu Software-Partitionierung.

Softwaredominantes System

Ein softwaredominantes System ist ein Sonderfall eines mechatronischen Systems, in welchem die darin verwendete Software den Großteil der Wertschöpfung (Aufwände für Entwicklung, Test, Absicherung) des Gesamtsystems ausmacht.

Stakeholder

Ein Stakeholder ist eine Person, die ein Interesse an einem System oder Projekt hat. Interessen von Stakeholdern sind in der Regel äußerst heterogen oder widersprüchlich. Typische Stakeholder in dem Kontext dieser Arbeit sind: Kunde, Software-Architekt, Software-Entwickler, Funktions-Architekt, Funktions-Entwickler, Tester, Vertrieb, Management.

Standpunkt

Standpunkte (engl. viewpoints) werden von Stakeholdern eingenommen, um die für sie wichtigen Informationen zu erhalten. Jedem Standpunkt ist ein Set von Sichten zugeordnet.

Steuergerät (auch Electronic Control Unit)

Steuergeräte sind zentraler Bestandteil von eingebetteten Systemen unterschiedlichster Anwendungen im Kraftfahrzeugbereich. Sie stellen eigenständige, elektronische Module mit einem geschlossenen Gehäuse und nach außen geführtem Stecker dar. Der innere Aufbau variiert je nach Anwendungsfall, typische Bauelemente sind: Platine, Microcontroller, Bus-Transceiver, Endstufe, ASIC, Analog-Digital-Wandler.

System

Ein (technisches) System ist eine Ansammlung von (System-)Bausteinen, die gemeinsam ein Ziel – die Realisierung einer Funktion – verfolgen, das von den Einzelelementen nicht erreicht werden kann. Ein Baustein kann aus Software, Hardware, Personen oder beliebigen anderen Einheiten bestehen.

System-Architektur

Synonym zu Technische Architektur.

Tailoring

Tailoring ist die Anpassung eines Prozesses oder Artefaktes an einen spezifischen Anwendungsbereich bzw. -fall. Die Anpassung kann durch Auslassen, Hinzufügen oder durch geänderte Anwendung von Teilaspekten erfolgen.

Technische Architektur

Die Technische Architektur wird folgend auf die Logische Architektur erstellt. Es werden Entscheidungen zur Realisierung und Implementierung getroffen. Die technische Architektur repräsentiert den Lösungsraum für eine konkrete technische Realisierung in einem bestimmten Fahrzeugtyp bzw. Derivat. Dazu zählen insbesondere konkrete Steuergeräte beziehungsweise Plattformen, konkrete Bus-Verbindungen zwischen diesen sowie konkrete Sensor- und Aktorbauformen.

Telegramm

Synonym zu Bus-Signal.

Validation

Unter Validation wird eine Überprüfung verstanden, ob ein Produkt für seinen bestimmten Einsatzzweck geeignet ist.

Variante

Eine Variante ist eine abweichende Ausprägung eines Teilaspekts zwischen unterschiedlichen Fahrzeugtypen oder Derivaten. Im Rahmen dieser Arbeit wird der Begriff Variante für unterschiedliche Ausprägungen der High-Level-Software verwendet.

Varianten-Management

Varianten-Management ist ein Ansatz zur Beschreibung, Verwaltung und Optimierung von Varianten.

Verifikation

Unter Verifikation wird eine Überprüfung verstanden, ob ein Produkt korrekt im Sinne seiner Spezifikation umgesetzt wurde.

Vorgehensmodell zur Entwicklung

Synonym zu Entwicklungsprozess.

Whitebox

Eine Whitebox ist ein System, das eigenständig oder Teil eines Gesamtsystems sein kann. Die Whitebox zeichnet sich dadurch aus, dass dem Verwender im Gegensatz zur Blackbox nicht nur das Schnittstellen- und (Übetragungs-) Verhalten des Systems sondern auch dessen innerer Aufbau bekannt sind. Verwandt zu den Begriffen Blackbox und Whitebox ist darüber hinaus der Begriff der Greybox.

Wirkkette

Eine Wirkkette ist die gesamte Abfolge von logischen Bausteinen beziehungsweise Teilfunktionen, die durchlaufen werden müssen, um auf Gesamtfahrzeug-Ebene eine bestimmte Wirkung (Funktion) zu realisieren. Je nach Betrachtungsumfang können statt logischen Bausteinen auch konkrete Bausteine und Komponenten (z.B. Sensor, Aktor, elektronisches Steuergerät) gemeint sein.

Zwischenprodukt

Das so genannte Zwischenprodukt ist ein zentrales Artefakt zur Dokumentation und Weiterverarbeitung von Software-Architekturen im XML-Format [122], beschrieben durch ein erweiterbares XML-Schema. Das Zwischenprodukt wird bei allen im Rahmen dieser Arbeit vorgeschlagenen Prozessschritten eingesetzt.

A Wesentliche Anforderungen an die Ansätze

A.1 Wesentliche Anforderungen an das zentrale Datenmodell

Tabelle A.1: Wesentliche Anforderungen an das zentrale Datenmodell

Kategorie	Anforderungstext
Umfang	Das Datenmodell muss den gesamten Sprachumfang (Kern & optionales) der ABSOFA abdecken.
	Das Datenmodell muss den gesamten architektur-relevanten Sprachumfang von ASCET abdecken.
	Das Datenmodell muss den gesamten architektur-relevanten Sprachumfang von Simulink abdecken.
	Das Datenmodell muss alle Informationen, die zur Bewertung von Software-Architekturen notwendig sind, abdecken.
Handhabung	Auf eine im Datenmodell abgespeicherte Software-Architektur müssen automatisiert die entwickelten Software-Architekturmetriken angewendet werden können.
	Eine im Datenmodell abgespeicherte Software-Architektur muss bidirektional in ein Format, das für die Werkzeuge ABSOFA, ASCET und Simulink lesbar ist, transformiert werden können.
	Das Datenmodell muss per Roundtrip-Verfahren mit den genannten Werkzeugen abgeglichen werden können. Das bedeutet, dass Informationen, die nicht in allen Werkzeugen relevant sind, als spezifische Informationen im Datenmodell gepflegt werden können.
	Eine Transformation zwischen dem Datenmodell mit ASCET und Simulink muss sowohl zur Übernahme und Weiterverarbeitung von bestehenden (Bottom-Up-Vorgehen) als auch im Rahmen der Entwicklung neuer Software-Architekturen mit Übergabe zur Implementierung am Ende der Entwicklung (Top-Down-Vorgehen) möglich sein.
	Eine mit Varianten-Informationen versehene Software-Architektur (150%-Modell) muss unter Angabe der konkreten Variante automatisiert in eine konkrete Konfiguration überführt werden können.
Format	Das Datenformat muss mit den Austauschformaten der oben genannten Werkzeuge kompatibel sein.
	Das Datenmodell muss um den Sprachumfang anderer, aktuell noch nicht bekannter, Werkzeuge grundsätzlich erweiterbar sein.
	Das Datenmodell muss um zusätzliche Modellierungselemente und Attribute grundsätzlich erweiterbar sein.
	Das Datenmodell muss um ganz neue Teilumfänge für automatisierte Weiterverarbeitung grundsätzlich erweiterbar sein. Konkret muss zukünftig eine kriteriengestützte, automatisierte Partitionierung von Bausteinen aus dem Ansatz heraus möglich sein.
	Das Datenformat muss frei verfügbar (das heißt ein offener Standard) sein.
	Die Informationen müssen formal und eindeutig im Datenmodell hinterlegt werden, und so eine automatisierte Verarbeitbarkeit ermöglichen.
	Die Informationen müssen für einen Menschen lesbar abgelegt werden oder leicht in eine solche Darstellung überführt werden können, um ein Editieren der Informationen zu ermöglichen.

Anmerkung zur Plattformabhängigen Sicht einer Software-Architektur

Zum Anwendungsfall 2.2.2 „Überführung Abstrakte → Plattformabhängige Sicht" aus Bild A.1 soll an dieser Stelle noch das Folgende angemerkt werden. Das Gesamtkonzept dieser Arbeit ermöglicht dem Modellierer diese Sicht explizit zu verwenden. Eine Verwendung ist jedoch nicht immer notwendig. Das Auflösen von Varianteninformationen und damit das Konfigurieren einer konkreten Variante gehört per Definition noch zum Abstrakten Standpunkt. Bei dieser Konfiguration werden jedoch einige spezifische Aspekte der technischen Realisierung durch die Variante bereits vorgegeben. Der „Plattformabhängigen Sicht" bleiben somit nur noch Aspekte vorbehalten, die sich aufgrund der Partitionierung in ein ganz konkretes Steuergerät ergeben. Dazu gehören beispielsweise die Umwandlung in konkrete Implementierungs-Datentypen wie „int 16", die Zuordnung zu zyklischen Tasks (z.B. 10ms-Raster) oder das Hinzufügen von Software-Adaptern, um die Anwendungs-Software mit der spezifischen LowLevel-Software zu verbinden. Für letztgenanntes können bei Bedarf bereits Bausteine der ABSOFA im 150%-Modell vorgehalten werden (zum Beispiel das „LowLevelAdaptionModule"), alternativ könnte ein solcher Adapter auch automatisiert in einer Sichten-Transformation nach Kategorie 2.2.2 oder vom Werkzeug zur Software-Umsetzung automatisch erzeugt werden.

Im Regelfall sollten möglichst viele Informationen über das 150%-Modell erfasst werden, so dass eine Variantenkonfiguration zum 100%-Modell bereits etliche Spezifika der konkreten Zielvariante enthält. Das Hinzufügen von Informationen, die sich nur aufgrund konkreter Hardware der Rechenplattform ergeben, erfolgt im Werkzeug zur Softwareumsetzung im Rahmen der Code-Generierung. An dieses wird per automatischer Transformation nach Kategorie 4.1.X direkt aus dem 100%-Modell übergeben.

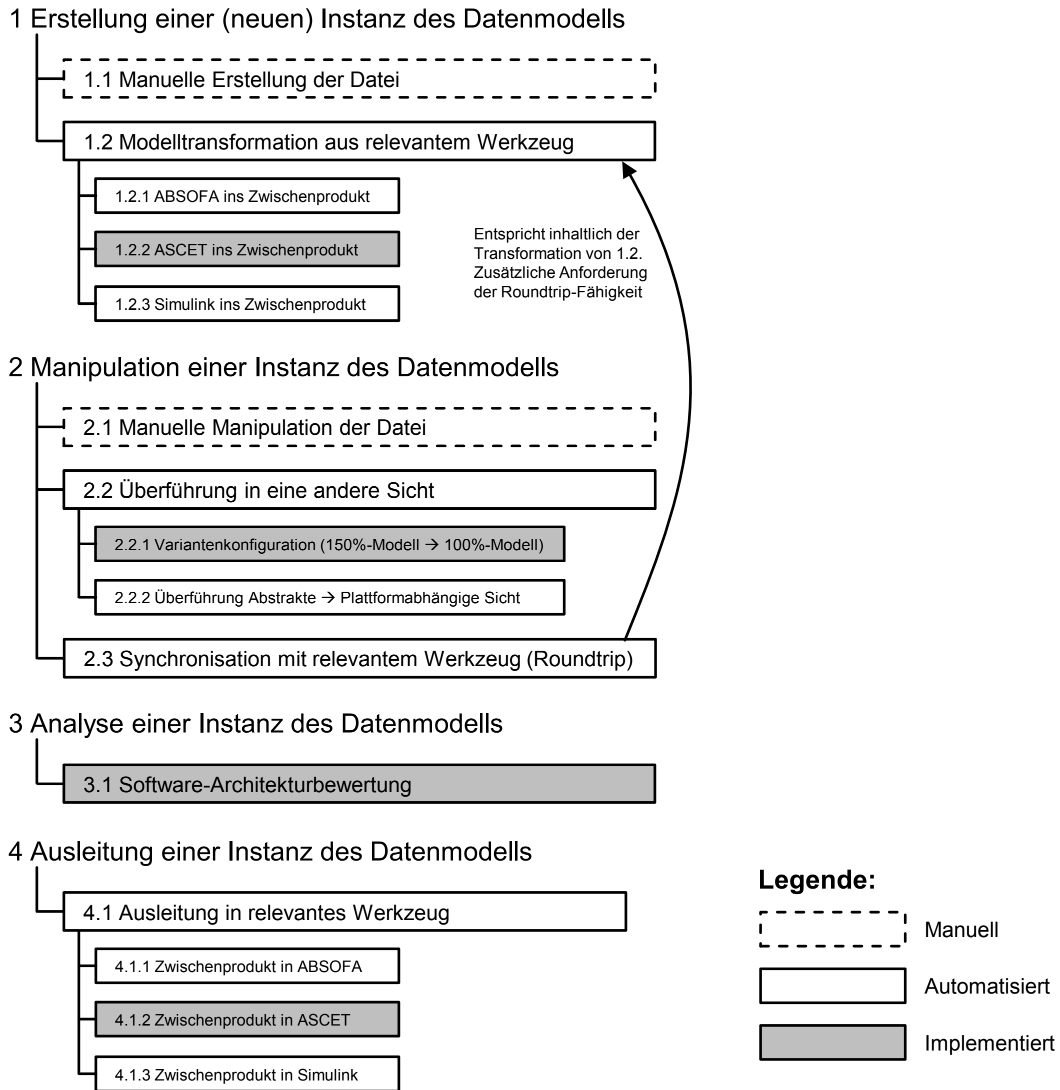

Bild A.1: Die unterschiedlichen Kategorien der Verwendung des Datenmodells

A.2 Wesentliche Anforderungen an die ABSOFA

Tabelle A.2: Wesentliche Anforderungen an den Ansatz der ABSOFA

Kategorie	Anforderungstext
Zielsetzung und Umfang	Die Software-Architektur soll vollständig von Details zur technischen Umsetzung abstrahieren.
	Die Darstellung der Software-Architektur muss unabhängig von Zielplattformen sein.
	Die Darstellung der Software-Architektur muss unabhängig von konkreten Werkzeugen der Software-Umsetzung sein.
	Die Software-Architektur muss die Software-Umfänge eines Systems gesamtheitlich darstellen ohne Abhängigkeit von konkreten Konfigurationen.
	Die statische Software-Struktur mit allen relevanten Aspekten muss dargestellt werden können.
	Die Modellierung von variantenspezifischen Architektur-Informationen muss möglich sein.
	Es muss eine automatische Konfiguration im Sinne einer Auswahl („Binding“) einer 150%-Architektur unter Angabe der zu wählenden Variante und Plattform möglich sein.
	Die Software-Architektur muss auch direkt im Kontext einer konkreten Plattform modelliert werden können.
	Die ABSOFA muss mit dem Datenmodell („Zwischenprodukt“) kompatibel sein.
	Alle Informationen der ABSOFA müssen im Zwischenprodukt eindeutig abgelegt werden können.
	Eine automatische Modelltransformation zwischen ABSOFA und ASCET muss möglich sein.
	Es muss ein Roundtrip-Verfahrens zwischen der ABSOFA, ASCET und Simulink möglich sein. Der Roundtrip soll sich auf architekturrelevante Anteile beschränken.
Darstellung	Die Modellierung muss eine beliebige Hierarchisierung ermöglichen.
	Auf derselben Hierarchie-Ebene müssen (hierarchisch) gleichgestellte Bausteine unterschiedlich detailliert werden können.
	Es muss ein modularer und skalierbarer Aufbau einer Software-Architektur ermöglicht werden.
	Alle Informationen sollen über alle Darstellungen und Hierarchie-Ebenen hinaus einheitlich und eindeutig dargestellt werden.
	Eine Interaktion zwischen Schnittstellen muss grafisch dargestellt werden.
	Die Bezeichnungen der Sichten und der einzelnen Diagramme muss durch eine einheitliche Namensgebung (-konvention) sichergestellt werden.
	Es müssen unterschiedliche Standpunkte und Sichten unterstützt werden.
Modellierungs-Elemente	Die Einordnung in die Stufen der Sicherheitsintegrität (ASIL-Level) von Software-Bausteinen muss unterstützt werden.
	Die Modellierung soll mit möglichst wenigen unterschiedlichen Elementen auskommen und auf unterschiedlichen Hierarchie-Ebenen möglichst dieselben Elemente verwenden
	Alle Informationen, die für die objektive Bewertung der Software-Architektur notwendig sind, müssen modelliert und dargestellt werden können.
	Informationen über untrennbare Bausteine müssen modelliert werden können.
	Es muss ein Bausteintyp verfügbar sein, der ausdrücklich mit AUTOSAR-Software-Components kompatibel ist und in den eine 1-zu-1-Überführung möglich ist.
Notation und Werkzeug	Das Werkzeug muss alle oben genannten Anforderungen der übrigen Kategorien erfüllen.
	Die Modellierung muss formal und in der Folge durch Automatismen verarbeitbar sein.
	Die selbe Information darf innerhalb eines Modells nur einmal modelliert werden.
	Der Modellierungsansatz muss um weitere Informationen erweiterbar sein.
	Das Werkzeug soll einem Open Standard genügen.
	Das Werkzeug muss eine XML-Schnittstelle zum Export und Import besitzen.
	Das Werkzeug muss eine Meta-Modellierung beziehungsweise Möglichkeiten zur Einschränkung der Verwendung der Modellierungs-Elemente erlauben.
	Das Werkzeug muss die Einhaltung der Modellierungs-Vorschriften beziehungsweise des Metamodells gesamthaft und automatisch überprüfen können.
	Das Werkzeug muss die Darstellung desselben Sachverhaltes in unterschiedlichen Diagrammen erlauben, dabei jedoch auf nur eine einzige Instanz der Datenablage zurückgreifen.
	Die Navigation innerhalb der Hierarchie-Ebenen soll möglichst einfach sein.

B Das UML-Profil der ABSOFA

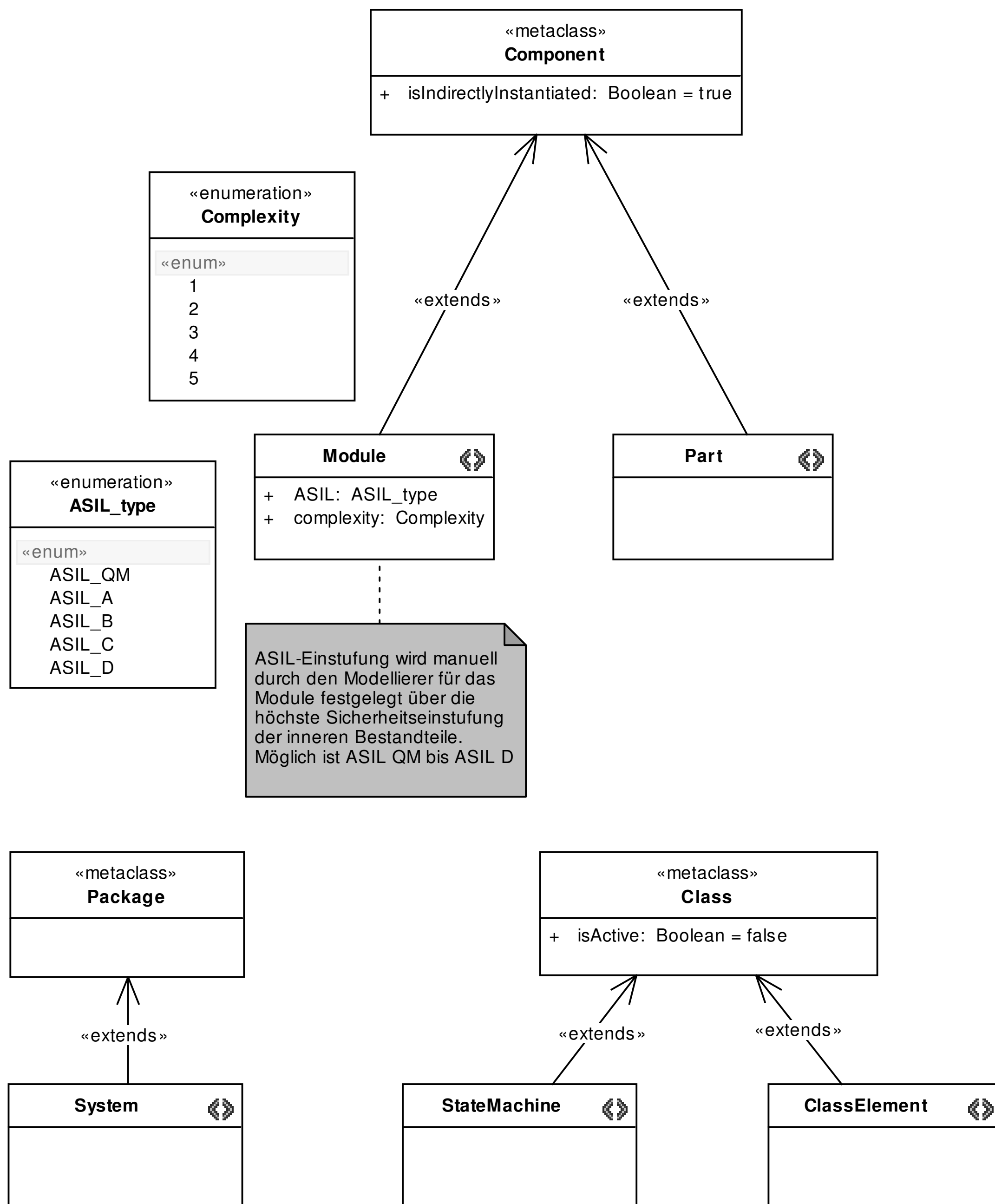

Bild B.1: Gesamtüberblick über die Strukturelemente der ABSOFA

Die Subtypen des Module

Aufgrund der besonderen Bedeutung des Module-Typs ist eine Spezialisierung in verschiedene Subtypen notwendig (Bild B.2). Die Vorgabe an die Modellierer ist immer den hierarchisch tiefst möglichen (und damit am spezialisiertesten) Typ einzusetzen. Die folgenden Subtypen sind verfügbar:

BehaviourModule: Unter die Kategorie „Modules mit Verhalten“ fällt in der Regel die Mehrzahl der modellierten Bausteine vom Typ Module. Gemeint sind alle Modules, die ein Verhalten im Sinne einer erlebbaren Funktionalität (Teil einer Kundenfunktion) aufweisen. „Erlebbar“ wird dabei bewusst weit gefasst: Jede Form von Funktionalität, die einen Beitrag zur Kundenfunktion leistet, ist darin enthalten, egal von welcher Art oder Umfang. Im Zweifel gilt: Alle Modules, die nicht eindeutig einer der übrigen Spezialisierungen des NonBehaviourModule-Typs zugeordnet werden, fallen im Umkehrschluss in die Kategorie der BehaviourModules.

PlatformIndependentBehaviourModule: Ein Hauptziel der Verwendung der ABSOFA ist bekanntlich eine möglichst konsequente realisierungsunabhängige Modellierung der Software-Architektur. Daher stellt die Verwendung von Bausteinen dieses Typs den Regelfall innerhalb der BehaviourModules und damit gleichzeitig in der gesamten Architektur dar. Alle Modules, die „Logik“ einer Kundenfunktion – beziehungsweise Teile davon – modellieren, fallen in diese Kategorie. Dieser Typ verfügt über keine weiteren Subtypen. Ein zusätzliches Attribut („functionality“) ist die Angabe des Hauptcharakters des Modules: Zustandsautomat, Regler/Steuerer oder Ansteuerung Aktorik.

PlatformDependentBehaviourModule: Wird für alle Modules mit Verhalten verwendet, die nicht realisierungsunabhängigen Charakter aufweisen (können). Dies kann in Ausnahmefällen innerhalb des Abstrakten Standpunkts (Struktursicht) der Fall sein, in der Regel jedoch innerhalb der Plattformabhängigen Sicht. Es wurden bei der Analyse der bestehenden Fahrerassistenzsysteme folgende notwendige Subtypen identifiziert:

- **SensorActuatorModule:** Wird für alle Modules verwendet, die als Adapter fungieren, um den realisierungsunabhängigen Anteil der Anwendungs-Software (der maximiert werden soll) mit konkreten Sensoren – zum Beispiel Full-Range-Radar bestimmter Bauform eines bestimmten Zulieferers – beziehungsweise Aktoren – zum Beispiel 4-Zylinder-Benzinmotor – zu verbinden. Anmerkung: In der Kontextsicht kann dieser Typ auch als Repräsentant des Sensors/Aktors an sich verwendet werden.
- **HighLevelAdaptionModule:** Wird für alle Modules verwendet, die als Adapter fungieren, um den realisierungsunabhängigen Anteil der Anwendungs-Software mit anderen Anteilen zu verbinden, die diesem Anspruch noch nicht in gleichem Maße genügen (zum Beispiel so genannte „Legacy-Software“).

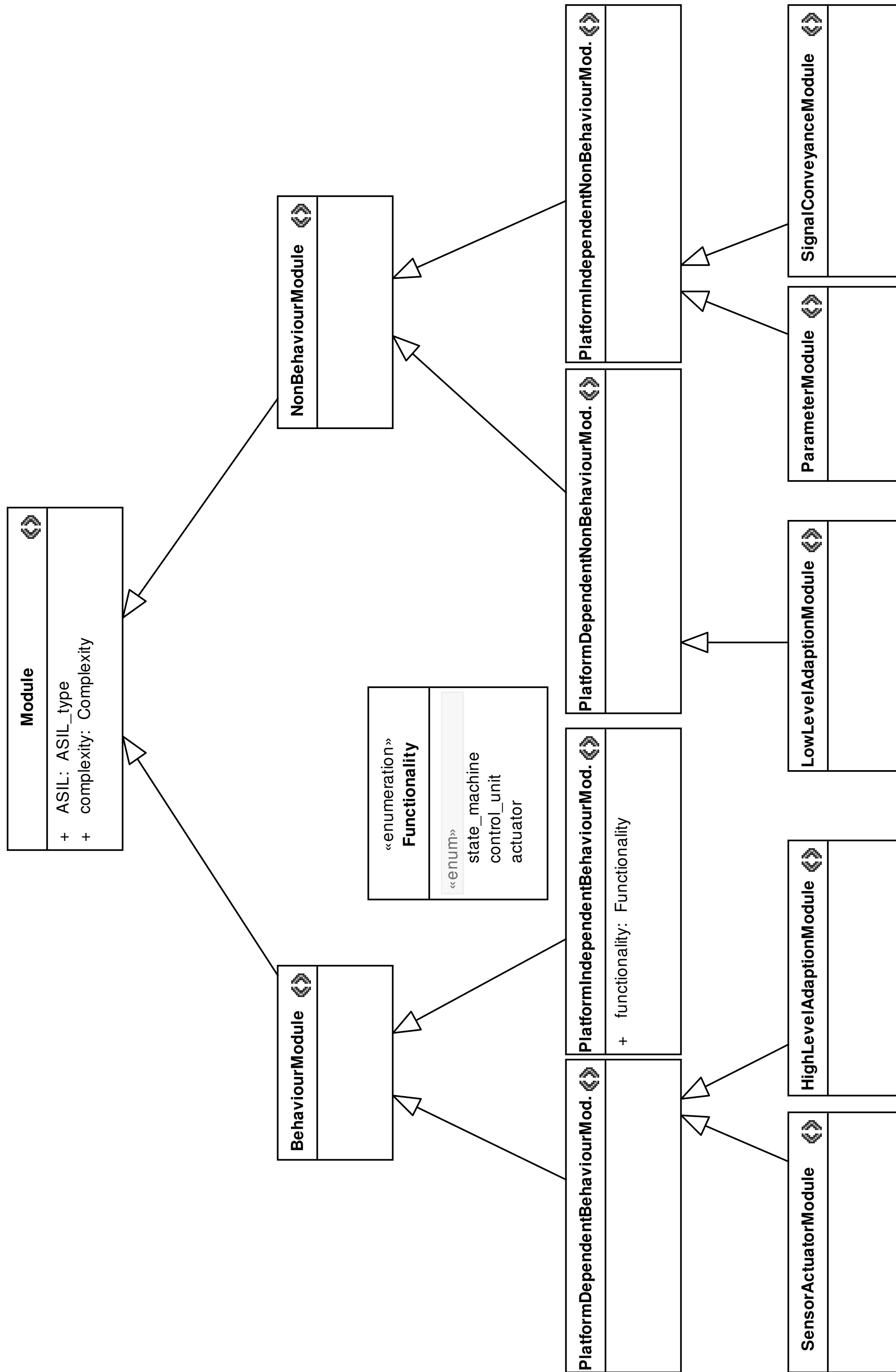

Bild B.2: Der Module-Typ und alle seine Subtypen

Anmerkung: Ist eine Software-Architektur nach diesem Grundsatz bereits in Gänze aufgebaut, ist ein solcher Adapter nicht notwendig, er ist lediglich Hilfskonstrukt in Misch-Architekturen.

NonBehaviourModule: Dieser Bausteintyp stellt einen Sonderfall zur Modellierung von Modules dar, die kein Verhalten im Sinne der Kundenfunktion enthalten. Diese sind, wie sich im Anschluss noch zeigen wird, immer Adapter beziehungsweise Hüllen. Da sie per Definition kein Verhalten aufweisen, kann in diesen Bausteinen keine Unterstruktur bestehend aus weiteren Strukturelementen modelliert werden. Es können nur Schnittstellenelemente für die Ein- und Ausgänge des Bausteins angegeben werden.

PlatformIndependentNonBehaviourModule: Es gelten grundsätzlich dieselben Überlegungen zur Einteilung in realisierungsabhängige und -unabhängige Subtypen wie weiter oben bereits erläutert. Das PlatformIndependentNonBehaviourModule benötigt die folgenden beiden Spezialisierungen:

- **ParameterModule:** In der Software-Entwicklung in der Automobil-Domäne nehmen Parameter in Form von Applikations- (auch Kalibrierungs- genannt) und Codierparametern eine wichtige Rolle ein. Es kann für eine Kundenfunktion leicht eine Vielzahl (bis zu mehreren Hundert) solcher Parameter geben. Wenn der Modellierer diese in einem oder mehreren Modulen zentral verwalten möchte, so wird dieser Module-Typ gewählt.
- **SignalConveyanceModule:** Dieser Bausteintyp stellt einen Sonderfall für einen Adapter auf Anwendungs-Software-Ebene dar. Wie der Name andeutet, werden Signale lediglich unverändert durchgeschleust, das bedeutet von einem Eingang auf einen Ausgang mit anderer Bezeichnung gelegt ohne eine Manipulation der Werte vorzunehmen (so genanntes „Mapping“). Daher wird dieser Typ bei den Modules ohne Verhalten einsortiert und bewusst nicht als weitere Spezialisierung des HighLevelAdaptionModules innerhalb der Modules mit Verhalten.

PlatformDependentNonBehaviourModule: Es gelten grundsätzlich dieselben Überlegungen zur Einteilung in realisierungsabhängige und -unabhängige Subtypen wie weiter oben bereits erläutert. Das PlatformDependentNonBehaviourModule benötigt nur noch eine weitere Spezialisierung:

- **LowLevelAdaptionModule:** Dieser Bausteintyp stellt einen Adapter zwischen der realisierungsspezifischen LowLevel-Software und der realisierungsunabhängigen Anwendungs-Software dar. Da somit eine der beiden Seiten des Adapters realisierungsabhängig ist, ist das gesamte Module der Kategorie PlatformDependent zuzuschlagen. Anmerkung: Es wird dabei davon ausgegangen, dass es hierbei primär um das Mapping von Schnittstellen, ähnlich wie im SignalConveyanceModule, geht. Ist eine Manipulation der Werte der

Signale nötig, kann dies über ein nachgeschaltetes HighLevelAdaptionModule erfolgen oder auch eine Logik im inneren des LowLevelAdaptionModules. Hier werden keine weiteren Vorgaben gemacht.

Einschränkungen und Regeln zur Verwendung der Strukturelemente

Die folgende Tabelle B.1 gibt einen Überblick über die formalen Regeln, welche die Anwendung der Strukturelemente in der ABSOFA einschränken. Die korrespondierenden OCL-Ausdrücke wurden dabei aus Platzgründen ebenso weggelassen wie alle nichtformalen Regeln und Richtlinien zur Modellierung, welche für alle Anwender jedoch nichtsdestotrotz verbindlich gelten.

Tabelle B.1: Formal definierte Einschränkungen und Regeln zur Verwendung der Strukturelemente der ABSOFA

Bausteintyp	Regel zur Verwendung
Module	Ein Module kann nur über Interfaceelemente des Typs Message und Parameter verfügen.
	Ein Module darf nur Parts, ClassElements und StateMachines enthalten.
	Ein Module muss in einem übergeordneten Element enthalten sein.
	Eine aus einem Module herausgehende Coupling muss auf ein Module, ein ClassElement oder eine StateMachine verweisen. Anmerkung: Couplings (ohne Abb.) zeigen Zwangskopplungen mit anderen Elementen an.
NBModule	NonBehaviourModules haben immer die Komplexität (Attribut „complexity“) = 0.
	NonBehaviourModules enthalten nur Schnittstellenelemente.
ParModule	Ein ParameterModule kann nur über Interfaceelemente des Typs Parameter verfügen.
SCModule	Ein SignalConveyanceModule kann nur über Interfaceelemente des Typs Message verfügen.
System	Ein System hat selbst keine Schnittstellen.
	Ein System darf nur Systems und Modules enthalten.
	Ein System kann alleine stehen oder in einem übergeordneten Element enthalten sein.
Part	Ein Part kann nur über Interfaceelemente des Typs Connector und Argument verfügen.
	Ein Part darf nur Parts, ClassElements und Statemachines enthalten.
	Ein Part muss immer in einem übergeordneten Element enthalten sein.
	Parts dürfen erst in Modules und unterhalb genutzt werden und nicht bereits direkt in beziehungsweise unterhalb Systems.
ClassElement	Ein ClassElement kann nur über Interfaceelemente des Typs Argument verfügen.
	Ein ClassElement darf nur Parts, ClassElements und StateMachines enthalten.
	Ein ClassElement muss immer in einem übergeordneten Element enthalten sein.
	ClassElements dürfen erst in Modules und unterhalb genutzt werden und nicht bereits direkt in beziehungsweise unterhalb Systems.
	Eine aus einem ClassElement ausgehende Coupling muss auf ein Module, ein ClassElement oder eine StateMachine verweisen
StateMachine	Eine StateMachine kann nur über Interfaceelemente des Typs Argument verfügen.
	Eine StateMachine darf nur Parts, ClassElements und StateMachines enthalten.
	Eine StateMachine muss immer in einem übergeordneten Element enthalten sein.
	StateMachines dürfen erst in Modules und unterhalb genutzt werden und nicht bereits direkt in beziehungsweise unterhalb Systems.
	Eine aus einer StateMachine ausgehende Coupling muss auf ein Module, ein ClassElement oder eine StateMachine verweisen.

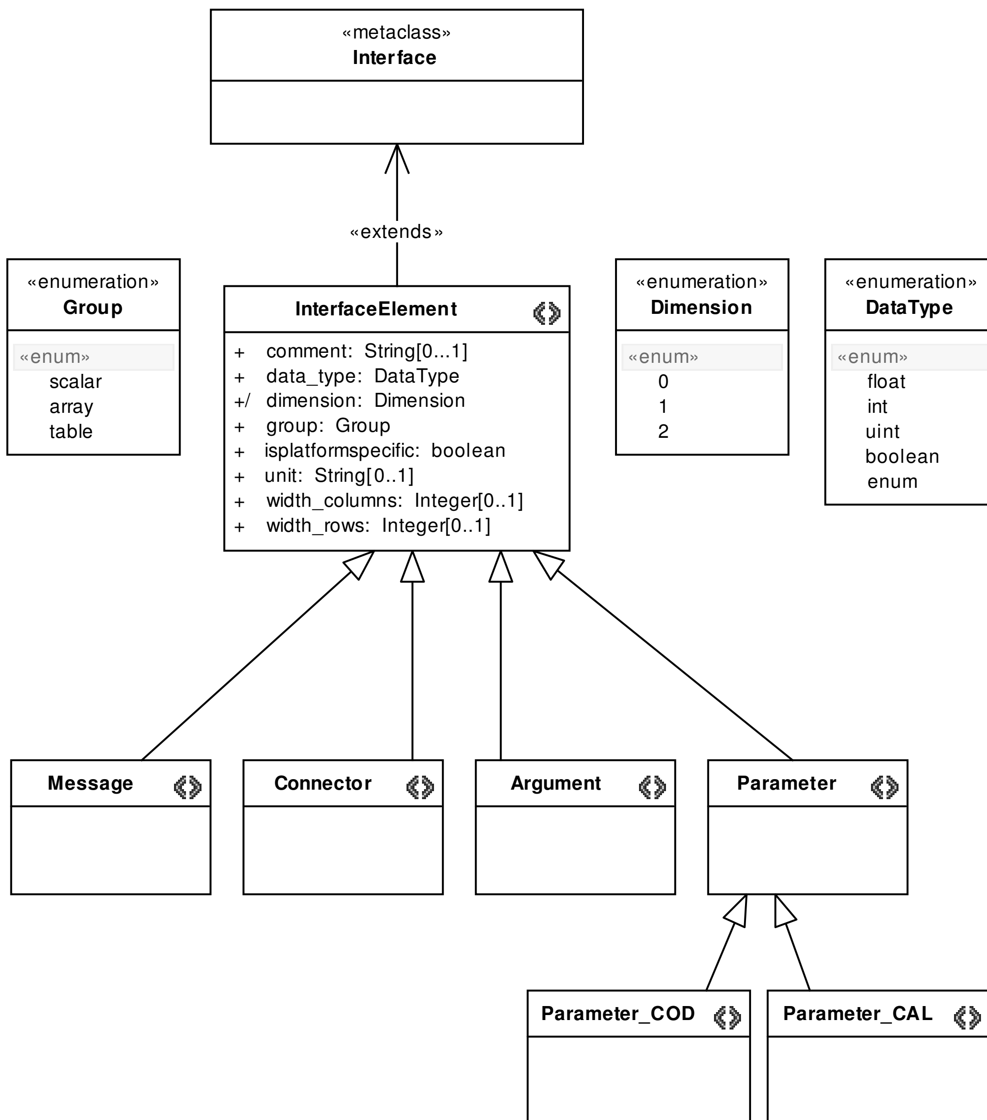

Bild B.3: Der InterfaceElement-Typ

Die Beziehungen Read, Write und ReadWrite

Optional können die Beziehungen Read, Write und ReadWrite eingesetzt werden. Damit wird keine Beziehung zwischen zwei Schnittstellenelementen angezeigt, sondern die Art der Verwendung eines Schnittstellenelements durch ein Strukturelement (Zugriff lesend, schreibend oder im Sonderfall lesend und schreibend). Innerhalb der Modellierung mit UML ist dies durch den anzugebenden Charakter eines InterfaceElements als „required“ oder „provided“ eindeutig. Die Beziehung wurde aus Gründen der Kompatibilität mit CASE-Werkzeugen, insbesondere ASCET, eingeführt (Bild B.4).

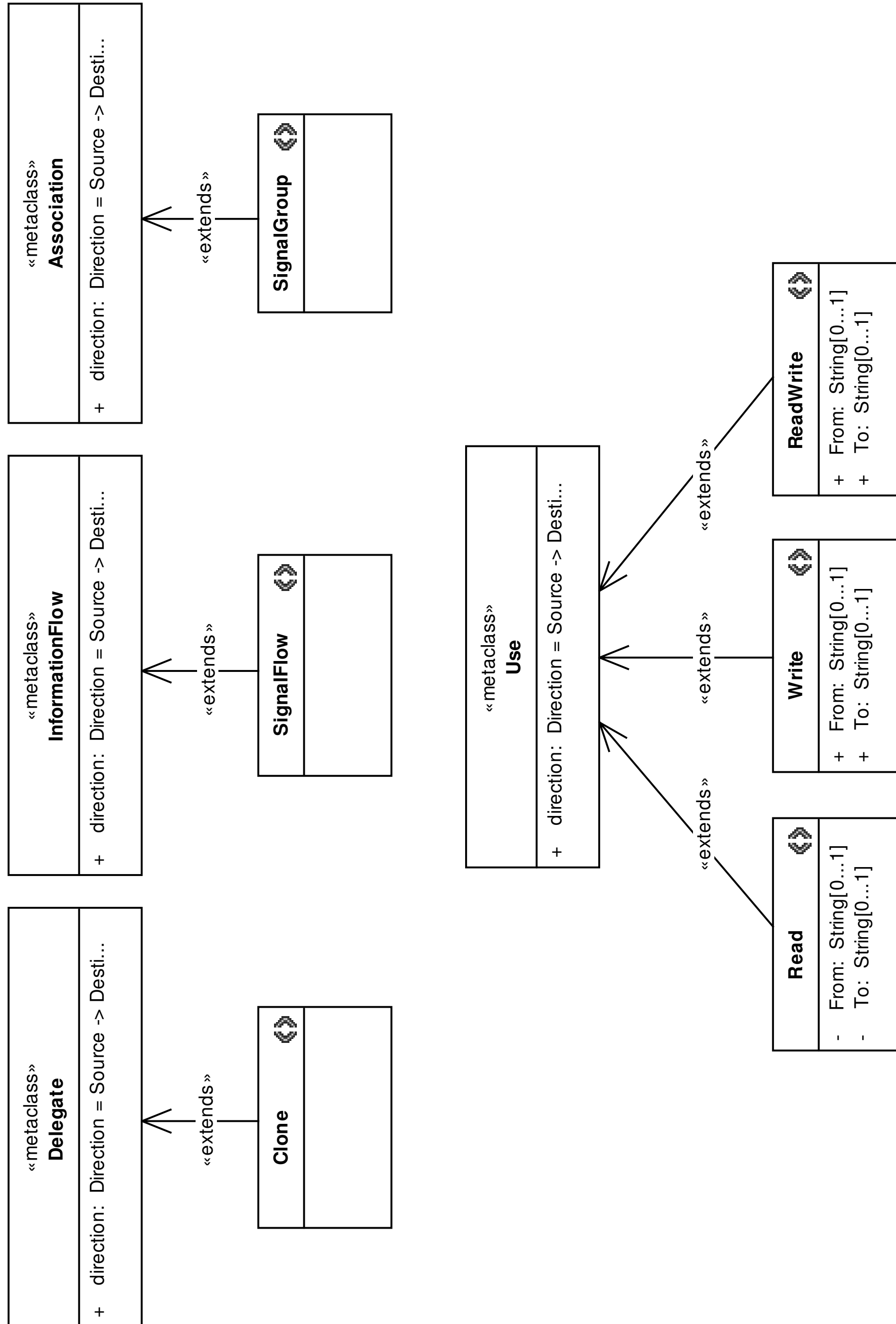

Bild B.4: Beziehungen zwischen Schnittstellenelementen

Tabelle B.2: Formal definierte Einschränkungen und Regeln zur Verwendung der Schnittstellen der ABSOFA

Element	Regel zur Verwendung
Interface Element	Ist das InterfaceElement skalar, so impliziert dies, dass die Tagged Values dimension, width_columns und width_rows = 0 sein müssen.
	Ist das InterfaceElement ein Array, so impliziert dies, dass die Tagged Values dimension = 1 und width_columns größer null und width_rows = 0 sein müssen.
	Ist das InterfaceElement eine Table, so impliziert dies, dass die Tagged Values dimension = 2 und width_columns größer null und width_rows größer null sein müssen.
Connector	Jeder Connector muss über genau eine Clone-Beziehung verfügen.
SignalFlow	Die Multiplizitäten einer SignalFlow-Beziehung müssen an beiden Enden = 1 sein.
	Ein SignalFlow muss immer zwischen zwei unterschiedlichen Interfaceelementen sein.
	Die Enden einer SignalFlow-Beziehung müssen entweder vom Typ Message oder vom Typ Argument oder vom Typ Parameter sein.
Clone	Die Multiplizitäten einer Clone-Beziehung müssen an beiden Enden = 1 sein.
	Die Clone-Beziehung muss immer zwischen zwei unterschiedlichen Interfaceelementen sein.
	Der Ursprung der Clone Beziehung muss ein Interfaceelement vom Typ Connector sein.
SignalGroup	Die Multiplizitäten einer SignalGroup-Beziehung müssen an beiden Enden = 1 sein.
	Die SignalGroup-Beziehung muss immer zwischen zwei unterschiedlichen Ports sein.

Modellierung von Variabilität

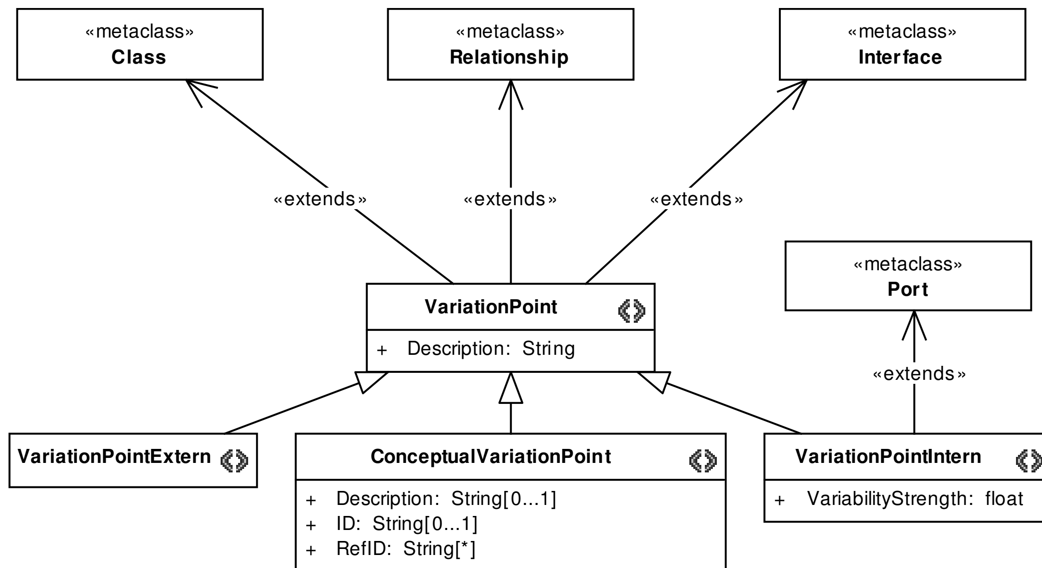

Bild B.5: Modellierung von Variabilität – Der VariationPoint

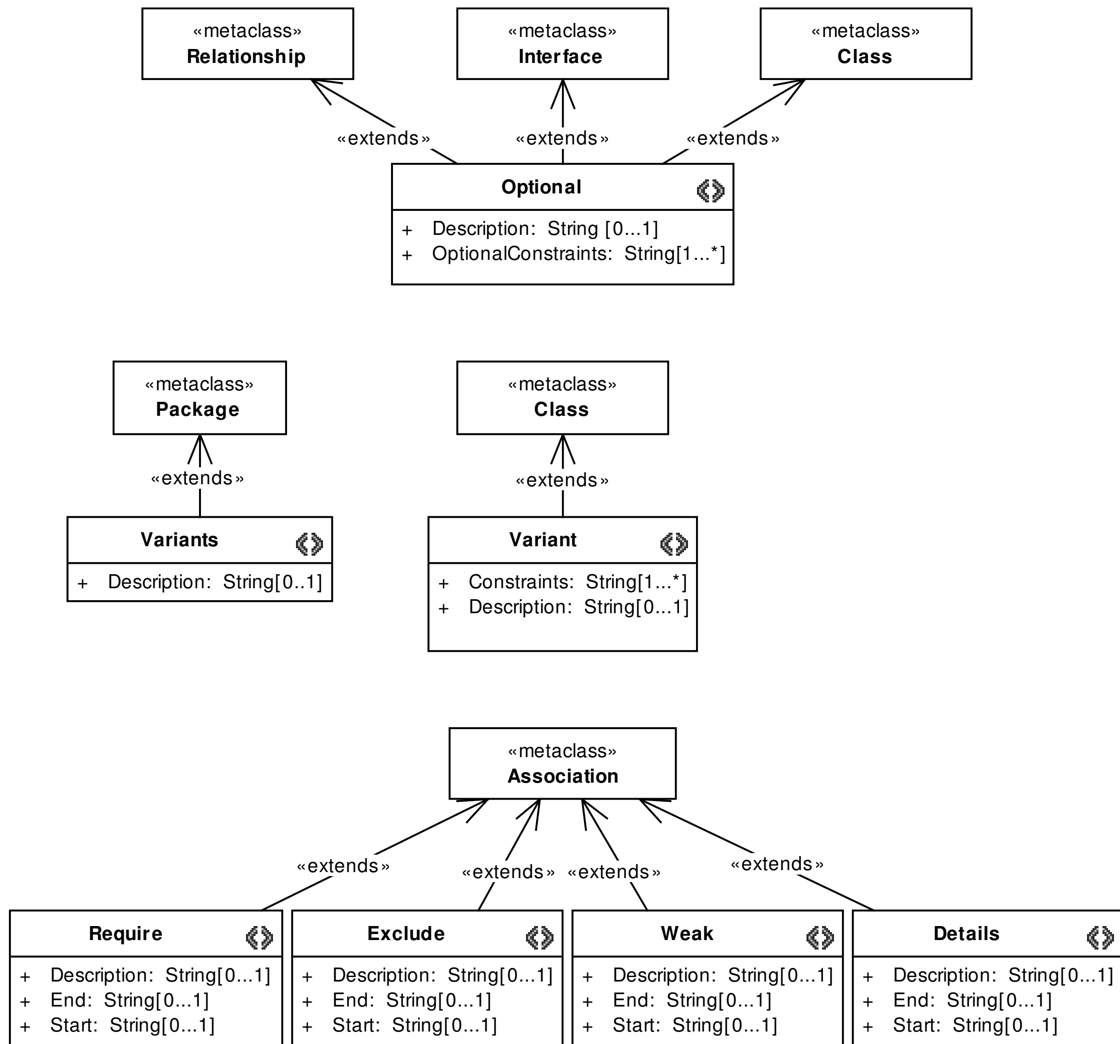

Bild B.6: Modellierung von Variabilität – Optionen, Varianten und Beziehungen

Tabelle B.3: Formal definierte Einschränkungen und Regeln zur Verwendung der Elemente der ABSOFA zur Modellierung von Variabilität

Element	Regel zur Verwendung
VariationPoint	Der Stereotyp VariationPoint darf nur Sprachelementen der ABSOFA zugewiesen werden.
Optional	Wird Optional einer Assoziation zugewiesen, so darf es sich dabei nur um eine SignalGroup-Beziehung handeln.
Variants	Ein Variantenpaket (Variants) darf nur Elemente vom Stereotyp Variant enthalten.
	Jedes Variantenpaket (Variants) ist über genau eine Dependency-Beziehung mit einem Variationspunkt verbunden.

C Qualitätskriterien und Qualitätsattribute

Im Folgenden werden alle Qualitätskriterien und -attribute des in Abschnitt 4.3.1 vorgestellten Qualitätsmodells im Detail definiert. Es ist ersichtlich, dass einige davon auf eher allgemeine, andere auf eine ausführlichere und an die Automotive-Domäne angepasste Art und Weise definiert werden. Insbesondere die in sich nicht messbaren Qualitätskriterien, deren primäre Aufgabe es ist, verschiedene Qualitätsattribute zu gruppieren, haben sich in Verbindung mit allgemeingültigen Definitionen bewährt. Dadurch kann das Qualitätsmodell in Gänze auf beliebige andere Domänen übertragen werden. Für Qualitätsattribute kann eine spezifische Definition grundsätzlich Sinn machen. In der vorliegenden Arbeit trifft dies beispielsweise auf die Attribute Änderbarkeit, Skalierbarkeit und Wiederherstellbarkeit zu. Eine Spezialisierung ist jedoch auch bei Qualitätsattributen nicht pauschal vonnöten, wie es beispielsweise am Attribut Verständlichkeit ersichtlich ist. Verständlichkeit wird im Rahmen dieser Arbeit durch die beiden konkreten Metriken M1 und M2 (Details finden sich in Anhang D) repräsentiert. Aufgrund der hier vorgenommenen, allgemeingültigen Definition könnten diese Metriken daher grundsätzlich auch in anderen Domänen verwendet werden. Die Eignung der Metriken ist jedoch immer in der konkreten Domäne innerhalb der dort vorliegenden, spezifischen Randbedingungen und Intentionen für den konkreten Anwendungsfall zu prüfen.

Effizienz

Effizienz ist die Fähigkeit einer Software, ihre festgelegte Funktionalität mit einem Minimum an Verbrauch von Ressourcen bereitzustellen. Ressourcen können dabei sowohl in Einheiten von eingesetzten Personen (beispielsweise zur Entwicklung, Inbetriebnahme oder Wartung) als auch von Rechen-Kapazitäten (beispielsweise Laufzeit, Speicher) angegeben werden.

Laufzeiteffizienz Die Laufzeiteffizienz gibt die Fähigkeit der Software an, möglichst geringe Antwort- und Verarbeitungszeiten sowie gleichzeitig hohe Durchsatzraten zu erzielen, wenn sie unter festgelegten Bedingungen ausgeführt wird. Laufzeiteffizienz kann in der Regel nur in Bezug auf konkrete Plattformen angegeben und verglichen werden.

Skalierbarkeit Skalierbarkeit wird durch die Eigenschaften einer Software bezüglich Änderung ihres statischen und dynamischen Ressourcenverbrauchs ausgedrückt, wenn Teilfunktionalitäten hinzugefügt oder entfernt werden.

Speichereffizienz Speichereffizienz ist die Fähigkeit einer Software, eine möglichst geringe Menge von statischem (ROM) und dynamischem (RAM) Speicherplatz in Anspruch zu nehmen, wenn sie auf einer Plattform installiert und ausgeführt wird.

Funktionserfüllung

Funktionserfüllung ist die Fähigkeit einer Software, alle explizit festgelegten und implizit erwarteten Anforderungen zu erfüllen.

Integrität Integrität ist die Unempfindlichkeit einer Software gegenüber unbefugter Benutzung sowie Zerstörung oder Manipulation von Daten. Dies umfasst externe (Bus-Kommunikation) und interne Schnittstellen sowie alle Daten in ROM- oder RAM-Speicherbereichen von Plattformen.

Interoperabilität Interoperabilität ist die Fähigkeit einer Software, mit allen externen Komponenten und Software-Systemen gemäß Spezifikation interagieren zu können.

Korrektheit Korrektheit ist die Fähigkeit einer Software, gestellte Anforderungen zu erfüllen.

Sicherheit Sicherheit stellt die Fähigkeit einer Software dar, Risiken und Gefahren für den Benutzer zu reduzieren oder zu verhindern.

Konformität

Konformität ist die Fähigkeit einer Software Standards, Vereinbarungen oder Regeln einhalten zu können.

Portabilität

Portabilität ist die Fähigkeit einer Software, von einer Umgebung in eine andere übertragen und dort genutzt werden zu können.

Anpassungsfähigkeit Ein Software-System ist anpassungsfähig, wenn es mit minimalem Aufwand in eine andere Umgebung übertragen und dort genutzt werden kann. Eine andere Umgebung kann dabei unter anderem durch eine unterschiedliche Plattform, ein unterschiedliches Fahrzeug oder Derivat sowie Benutzer-Schnittstellen charakterisiert sein.

Ersetzbarkeit Ersetzbarkeit ist die Fähigkeit einer Software, eine andere Software oder Teile davon, die für gleiche Zwecke eingesetzt wurde, in der entsprechenden Umgebung ersetzen zu können.

Installationsfähigkeit Installationsfähigkeit bezeichnet die Fähigkeit einer Software, in unterschiedlichen Umgebungen installiert werden zu können. Eine Installation umfasst den Transfer der Software auf die Plattform (zum Beispiel über flashen) sowie das Aktivieren und Konfigurieren (zum Beispiel über codieren) im Rahmen der Umgebung.

Wandlungsfähigkeit

Wandlungsfähigkeit ist die Fähigkeit eines Software-Systems, während seines Lebenszyklus verändert werden zu können. Änderungen können Korrekturen, Verbesserungen oder Anpassungen einer Software umfassen.

Änderbarkeit Änderbarkeit bedeutet die Fähigkeit, spezifische Modifikationen an einer Software vornehmen zu können. Modifikationen können aufgrund von geänderten funktionalen oder nicht-funktionalen Anforderungen notwendig werden.

Erweiterbarkeit Erweiterbarkeit ist die Fähigkeit, eine Software auch nach Fertigstellung um zusätzliche Anforderungen erweitern zu können. Zusätzliche Anforderungen können beispielsweise das Hinzufügen neuer Funktionalität, neue Wechselwirkungen zu externen Systemen oder neue funktionale sowie plattformspezifische Varianten beschreiben.

Stabilität Stabilität einer Software stellt die Fähigkeit dar, unbeabsichtigte Auswirkungen durch Änderungen an der Software zu verhindern. Eine Software kann als stabil angesehen werden, wenn beabsichtige Änderungen an bestimmten Stellen nicht unausweichlich zu weiteren, jedoch unbeabsichtigten Änderungen an anderen Stellen der Software resultieren, um diese an die erste Änderung anzupassen.

Testbarkeit Testbarkeit ist die Fähigkeit, eine Software gemäß einer Spezifikation verifizieren zu können. Testen findet auf allen Ebenen des Entwicklungsprozesses statt von der untersten Ebene (über Komponenten- oder Modultest) bis zur obersten Ebene (Gesamtfahrzeugintegration) und kann über unterschiedliche Verfahren erfolgen (z.B. model-in-the-loop, software-in-the-loop, hardware-in-the-loop).

Verständlichkeit Verständlichkeit ist die Fähigkeit den Aufbau, das Konzept und die Anwendung einer Software oder eines Software-Elements einem Stakeholder deutlich zu machen.

Wiederverwendbarkeit

Wiederverwendbarkeit ist die Fähigkeit einer Software oder Teilen davon, in mehr als einem Anwendungsfall Verwendung zu finden.

Allgemeingültigkeit Allgemeingültigkeit ist die Eigenschaft eines Software-Elements, so wenige funktionale Spezialisierungen wie möglich aufzuweisen.

Modularität Die Modularität eines Software-Systems besagt, ob und wie es in eine Menge jeweils in sich geschlossener Bausteine zerlegt wurde.

Zuverlässigkeit

Zuverlässigkeit ist die Fähigkeit einer Software, ihr Leistungsniveau unter festgelegten Bedingungen über einen festgelegten Zeitraum zu bewahren.

Robustheit Robustheit stellt die Fähigkeit einer Software dar, auch dann noch korrekt zu funktionieren, wenn Randbedingungen nicht der zulässigen Spezifikation entsprechen. Dies umfasst Eingangssignale und Parameter außerhalb von spezifizierten Wertebereichen sowie Fehlbedienung und Missbrauch durch den Benutzer.

Verfügbarkeit Verfügbarkeit ist die Wahrscheinlichkeit, ein System zu einem vorgegebenen Zeitpunkt in einem funktionsfähigen Zustand anzutreffen.

Wiederherstellbarkeit Wiederherstellbarkeit ist die Fähigkeit von Software, einen funktionsfähigen Zustand zu erreichen, nachdem ein fehlerhafter Zustand eingetreten ist. Der funktionsfähige Zustand kann durch Selbstheilung oder externe Maßnahmen wieder eingenommen werden. Wiederherstellbarkeit umfasst zusätzlich alle Maßnahmen zur Kommunikation und Speichern von Fehler-Modi und -Codes an den Benutzer sowie an das Wartungspersonal.

D Software-Architekturmetriken

Im folgenden Abschnitt finden sich detaillierte Informationen zu den übrigen sieben Software-Architekturmetriken M1-M6 und M8. M7 wurde bereits in Abschnitt 4.3.1 vorgestellt. M8 ist die Metrik zur Speichereffizienz. Da sie als einzige Metrik nicht nur Architektur- sondern auch Implementierungs-Informationen berücksichtigt, nimmt sie eine Sonderstellung innerhalb der Metriken ein. Aus diesem Grund fließt M8 nicht in die Berechnung des Architektur-Gesamtwertes ein, sondern liegt als Zusatzinformation vor. Der ideale (optimale) Wert aller Metriken liegt bei 0 und es gilt grundsätzlich, dass ein niedrigerer Wert immer besser ist als ein höherer.

Metrik der Strukturverständlichkeit – M1

Idee
Die innere Struktur einer Software-Architektur hat einen enormen Einfluss auf deren Verständlichkeit gegenüber allen Stakeholdern. Durch den Einsatz von Abstraktion können Bausteine auf derselben Ebene zu übergeordneten Strukturen zusammengefasst werden. Umgekehrt erlaubt eine Verfeinerung die Dekomposition einzelner Elemente in Sub-Komponenten. So entstehen unterschiedliche Sichten und Ebenen der Detaillierung, das heißt eine hierarchische Struktur. Nach Miller, 1956 [81] sind Systeme, die aus 7±2 Sub-Komponenten bestehen, am besten verständlich. Diese erste Metrik bewertet somit die Erfüllung dieser Regel in jedem Strukturelement auf jeder Hierarchie-Ebene über die gesamte Software-Architektur hinweg.

Metrik
Es ist ein Klassifikations-Schema vonnöten, welches die Zuordnung der Menge an Sub-Komponenten innerhalb eines Strukturelements zu einem Faktor ermöglicht. Mit diesem Faktor wird der Wertebereich der Metrik und die Gewichtung der Abweichungen von der oben genannten Regel quantifiziert. Der Faktor wird im weiteren Verlauf als *Component Complexity Factor (CCF)* bezeichnet und wird über die folgende Tabelle D.1 ermittelt. Der CCF wird im Algorithmus zur Berechnung der Metrik unmittelbar verwendet.

$$M1_i = \frac{CCF_i + \frac{\sum_{j=1}^{n} M1_j}{n}}{2} \tag{D.1}$$

$$1 \leq j \leq n$$

$$i \mathrel{\widehat{=}} Aktuelle\ Komponente\ i$$
$$n \mathrel{\widehat{=}} Anzahl\ der\ Sub\text{-}Komponenten\ in\ der\ Komponente\ i$$
$$CCF_i \mathrel{\widehat{=}} CCF\ der\ Komponente\ i$$
$$M1_j \mathrel{\widehat{=}} Strukturverständlichkeit\ der\ Sub\text{-}Komponente\ j$$

Tabelle D.1: Ermittlung des Component Complexity Factors (CCF)

# Sub-Komponenten	CCF
0	0
1-4	2
5-9	0
10-11	3
12-14	5
>14	10

Erläuterungen
Die Metrik arbeitet rekursiv über alle Hierarchie-Ebenen. In der aktuell betrachteten Ebene wird der Wert zur Strukturverständlichkeit einer darin untersuchten Sub-Komponente ermittelt und eine Ebene höher gereicht. Dort wird der Wert zusammen mit den Werten der übrigen Sub-Komponenten auf identischer Ebene verwendet, um die Strukturverständlichkeit der übergeordneten Komponente zu ermitteln. Der Wertebereich liegt zwischen 0 und 10.

Metrik der Bausteinverständlichkeit – M2

Idee
Die Verständlichkeit eines Software-Bausteins hängt typischerweise von zwei messbaren Eigenschaften ab. Die erste ist die Anzahl der in dem Baustein verwendeten (das heißt die Summe der ein- und ausgehenden) Schnittstellen, die zweite ist die innere Komplexität des Bausteins. Als Schnittstellen gelten in diesem Kontext alle Messages sowie Applikations- und Codierparameter. Parameter werden häufig durch designierte und nur dafür eingesetzte Bausteine bereitgestellt und können dann in unterschiedlichen Bausteinen verwendet werden. Dies ist ein für die Automobil-Domäne spezifischer Aspekt und muss berücksichtigt werden. Parameter und Messages haben einen unterschiedlichen Einfluss, welcher bei der Berechnung der Metrik einzukalkulieren ist.

Metrik
Zur Berechnung des Zahlenwerts der Metrik werden drei Faktoren benötigt. Alle ein- und ausgehenden Messages werden aufsummiert und die Gesamtzahl zur Ermittlung des Message Quantity Factor (MQF) herangezogen. Analog wird mit Parametern verfahren, Ergebnis ist der Parameter Quantity Factor (PQF). Tabelle D.2 zeigt die Ermittlung

der beiden genannten Faktoren. Es ist sofort ersichtlich, dass unterschiedliche Anzahlen an Messages beziehungsweise Parametern notwendig sind, um denselben Zahlenwert des zugehörigen Faktors zu erhalten. Der Einfluss der Parameter auf die Komplexität und damit auch die Verständlichkeit eines Bausteins ist deutlich geringer als die von Messages. Die Gewichtung basiert auf den Erfahrungen von Software-Architekten und -Entwicklern. Die innere Komplexität des Bausteins wird durch den Module Complexity Factor (MCF) repräsentiert. Dieser wird aufgrund (subjektiver) Einschätzung des für einen Baustein verantwortlichen Software-Architekten oder -Entwicklers auf Basis des erwarteten Umfangs und Komplexität vergeben.

$$M2 = \frac{\sum_{i=1}^{n}((MQF_i + PQF_i) \cdot MCF_i)}{n} \tag{D.2}$$

$$n \mathrel{\hat{=}} \textit{Gesamtzahl aller Modules}$$
$$MQF_i \mathrel{\hat{=}} \textit{Message Quantity Factor von Module i}$$
$$PQF_i \mathrel{\hat{=}} \textit{Parameter Quantity Factor von Module i}$$
$$MCF_i \mathrel{\hat{=}} \textit{Module Complexity Factor von Module i}$$

Tabelle D.2: Faktoren zur Ermittlung der Bausteinverständlichkeit

# Messages	MQF	# Parameter	PQF	Innere Komplexität	MCF
0-3	0	0-10	0	keine	0
4-10	1	11-25	1	gering	1
11-20	2	26-50	2	mittel	2
21-40	5	51-100	5	hoch	3
>40	10	>100	10		

Erläuterungen
Der Metrikwert für die gesamte Architektur ergibt sich aus dem Durchschnitt der Metrikwerte zur Bausteinverständlichkeit aller Modules. Die Metrik arbeitet konkret auf der Ebene der Modules als spezieller Typ Strukturelement. Diese Bausteine werden hier als Blackbox betrachtet und die mögliche innere Struktur nicht weiter rekursiv betrachtet. Dies wäre möglich aber für die Aufgabe an dieser Stelle nicht angemessen. Der Wertebereich liegt zwischen 0 und 60.

Metrik der Änderbarkeit – M3

Idee

Grundsätzlich gilt, dass an einem Software-System respektive einer Software-Architektur immer Änderungen vorgenommen werden können. Die entscheidende Frage ist, wie leicht (das bedeutet mit wie viel Aufwand) diese Änderungen eingebracht werden können und welche Auswirkung diese Änderungen auf den Rest der Architektur haben. Aus diesem Grund sind „Änderbarkeit" und „Stabilität" sehr eng miteinander verknüpft. Eine Aussage, wie die innere Änderbarkeit eines Bausteins beschaffen ist, kann ausschließlich mit Architektur-Informationen nicht getroffen werden. Im Kontext der Gesamt-Architektur kann jedoch eine Einschätzung zur Änderbarkeit eines Bausteins über den Grad der Verwendung von ausgehenden Schnittstellen in anderen Bausteinen getroffen werden. Dies erfolgt auch bei M3 speziell auf Ebene der Modules, die betrachteten Schnittstellen auf dieser Ebene sind Messages und Parameter. Neben der Verwendung der Schnittstelle an sich, wirken sich ebenfalls die innere Komplexität der empfangenden Modules und die Wahrscheinlichkeit der Änderung der betrachteten Schnittstellen aus.

Metrik

Ähnlich wie bei M2 werden insgesamt drei Faktoren zur Berechnung der Änderbarkeit benötigt. Der erste ist der bereits erwähnte MCF, die anderen beiden Faktoren sind neu. Wie häufig ein Schnittstellenelement über die Architektur hinweg verwendet wird, kann durch den Interface Use Factor (IUF) angegeben werden. Die Wahrscheinlichkeit einer Änderung einer Schnittstelle ist schwierig zu ermitteln. Im Einzelnen kann dies nie zuverlässig vorhergesagt werden, da dies am konkreten Software-System und dem konkreten, nicht vorhersehbaren Änderungsbedarf abhängt. Der hier verwendete, verallgemeinerte Ansatz basiert auf Erfahrungen von Software-Architekten und beruft sich auf den Typ der Schnittstelle. Schnittstellen vom Typ „enum" ändern sich erfahrungsgemäß häufiger als andere Typen. Dies kann nur als statistischer Ansatz verstanden werden und spiegelt eine grundsätzliche Änderungswahrscheinlichkeit wider. In Form eines Zahlenwerts wird dieser Sachverhalt über den Data Type Factor (DTF) ausgedrückt. Tabelle D.3 zeigt die Ermittlung von IUF und DTF.

Tabelle D.3: Faktoren zur Ermittlung der Änderbarkeit

# Schnittstellenverwendungen	IUF	Datentyp	DTF
1	1	nicht enum	1
2	2	enum	4
3-4	5		
5-10	10		
>10	20		

$$M3 = \frac{\sum_{i=1}^{m}(DTF_i \cdot IUF_i \cdot \oslash MCF_i)}{m} \tag{D.3}$$

$$\oslash MCF_i = \frac{\sum_{j=1}^{l_i} MCF_j}{l_i} \tag{D.4}$$

$m \mathrel{\hat{=}}$ *Gesamtzahl aller Schnittstellen*
$l_i \mathrel{\hat{=}}$ *Anzahl aller Modules verbunden mit Schnittstelle i*
$DTF_i \mathrel{\hat{=}}$ *Data Type Factor der ausgehenden Schnittstelle i*
$IUF_i \mathrel{\hat{=}}$ *Interface Use Factor der ausgehenden Schnittstelle i*
$MCF_i \mathrel{\hat{=}}$ *Module Complexity Factor von Module i*
$\oslash MCF_i \mathrel{\hat{=}}$ *Durchschnitt aller MCF_j, die die ausgehende Schnittstelle i verwenden*

Erläuterungen
Die Änderbarkeit wird auf der Module-Ebene ermittelt. Das Produkt aus IUF, DTF und dem Durchschnitt der MCFs aller die Schnittstelle verwendenden Modules wird für jedes Schnittstellenelement der Architektur gebildet. Anschließend wird der Durchschnitt über alle Schnittstellen ermittelt. Kerngedanken sind die Folgenden: Eine zahlreiche Verwendung von Schnittstellen wird durch den IUF bestraft, da bei Änderung des Schnittstellenverhaltens etliche Bausteine geändert werden müssen. Zur Abschätzung der Tragweite einer solchen (möglichen) Änderung an einem Module wird dessen innere Komplexität durch den MCF berücksichtigt. Die Wahrscheinlichkeit dieser Änderung fließt über den DTF ein. Der Wertebereich liegt zwischen 0 und 240.

Metrik der Stabilität – M4

Idee
Die Stabilität eines Modules wird ermittelt durch die Frage, wie häufig Änderungen an diesem Module aufgrund äußerer Einflüsse nötig sind. Dies kann nur durch Änderungen an Eingangs-Schnittstellen provoziert werden. Aus diesem Grund ist ein Module mit vielen Eingangs-Schnittstellen anfälliger gegenüber äußeren Einflüssen als ein Module mit wenigen. Die Idee der Metrik ist sehr ähnlich zur Änderbarkeit, sie wird jedoch aus Sicht eines Modules ermittelt, während Letztgenannte über die Schnittstellen auf Gesamt-Architekturebene errechnet wird.

Metrik
Zur Ermittlung des Zahlenwerts von M4 werden alle ein- und ausgehenden Schnittstellen eines Modules berücksichtigt. Zu den relevanten Schnittstellen auf Module-Ebene zählen auch hier Messages sowie Applikations- und Codierparameter. Die Angabe der

Wahrscheinlichkeit der Änderung einer einzelnen Schnittstelle erfolgt anhand des bereits von M3 bekannten Data Type Factors (DTF).

$$M4 = \frac{\sum_{i=1}^{n} \left(\frac{\sum DTF_{in,i}}{\sum DTF_{in,i} + \sum DTF_{out,i}} \right)}{n} \tag{D.5}$$

$n \mathrel{\hat{=}}$ *Gesamtzahl aller Modules*
$\sum DTF_{in,i} \mathrel{\hat{=}}$ *Summe der Data Type Factors aller eingehenden Schnittstellen von Module i*
$\sum DTF_{out,i} \mathrel{\hat{=}}$ *Summe der Data Type Factors aller ausgehenden Schnittstellen von Module i*

Erläuterungen
Die Metrik berechnet das Verhältnis der eingehenden zur Summe der ein- und ausgehenden Schnittstellen eines Modules. Das Ergebnis ist ein prozentualer Werte für die (In-)Stabilität dieses Modules. Der Wertebereich liegt zwischen 0 und 1, je kleiner der Wert desto stabiler ist das Module. Der optimale Wert 0 kann nur für Modules ohne eingehende Schnittstellen erreicht werden.

Metrik der Testbarkeit – M5

Idee
Das Testen stellt ein weites Feld mit einer Reihe unterschiedlicher und unabhängiger Test-Aktivitäten innerhalb des Vorgehensmodells dar. Die Metrik der Testbarkeit fokussiert auf Software-Tests und dabei speziell auf Tests auf der wichtigen Module-Ebene. Gemäß Aussagen von Experten hängt die Testbarkeit eines Bausteins vorrangig von Anzahl und Typ der eingehenden Schnittstellen sowie dem innerem Aufbau (und damit der Komplexität) des Bausteins ab.

Metrik
Als relevante Eingangs-Schnittstellen wurden Messages und Codier-Parameter identifiziert. Applikations-Parameter spielen in Bezug auf Testbarkeit nur eine untergeordnete Rolle und werden nicht berücksichtigt. Gemäß Tabelle D.4 wird der so genannte Input Quantity Factor (IQF) gebildet, der einen Zahlenwert anhand der Anzahl der eingehenden Schnittstellen ermittelt. Des Weiteren werden die bekannten Faktoren MCF und DTF in die Berechnung einbezogen.

$$M5 = \frac{\sum_{i=1}^{n} (MCF_i \cdot IQF_i \cdot \oslash DTF_i)}{n} \tag{D.6}$$

$$\oslash DTF_i = \frac{\sum_{j=1}^{m_i} DTF_j}{m_i} \tag{D.7}$$

$$n \mathrel{\widehat{=}} \textit{Gesamtzahl aller Modules}$$
$$m_i \mathrel{\widehat{=}} \textit{Anzahl aller eingehenden Schnittstellen von Module } i$$
$$MCF_i \mathrel{\widehat{=}} \textit{Module Complexity Factor von Module } i$$
$$IQF_i \mathrel{\widehat{=}} \textit{Input Quantity Factor von Module } i$$
$$DTF_j \mathrel{\widehat{=}} \textit{Data Type Factor der eingehenden Schnittstelle } j$$
$$\oslash\ DTF_i \mathrel{\widehat{=}} \textit{Durchschnitt der Data Type Factors aller eingehenden Schnittstellen von Module } i$$

Tabelle D.4: Ermittlung des Input Quantity Factors (IQF)

# Input (Messages & Codier-Parameter)	IQF
0-10	1
11-20	3
21-40	10
>40	20

Erläuterungen
Der Testaufwand ist proportional zur Anzahl eigenständiger Testfälle. Diese leiten sich aus der Anzahl der eingehenden Schnittstellen und deren möglichen Werten ab. Schnittstellen vom Typ „enum“ haben ein größeres Gewicht, da diese typischerweise Verzweigungen im Programmablauf steuern und somit separat zu testende Pfade der Logik repräsentieren. Jeder enum steht somit für mehrere Testfälle. Der Durchschnitt aller DTFs der eingehenden Schnittstellen eines Modules wird gebildet und mit dem MCF und IQF multipliziert. Der Gesamtwert zur Testbarkeit der Architektur ergibt sich als Durchschnitt aller Werte der einzelnen Modules. Der Wertebereich liegt zwischen 0 und 240.

Metrik der Skalierbarkeit – M6

Idee
Variantenmanagement ist ein wichtiger Aspekt für Software-Systeme in der Automobil-Domäne. Fahrzeuge in einer Basis- oder reduzierten Ausstattung sollten nur genau die Software-Anteile an Bord haben und Hardware-Ressourcen verbrauchen, die für exakt die ausgelieferten Kundenfunktionen notwendig ist. Dies wird durch eine gute Skalierbarkeit erreicht. Die Metrik der Skalierbarkeit soll ermitteln, wie gut beziehungsweise einfach

Teile der Software(-Architektur) aus dem Gesamtsystem entfernt werden können. Dies wird primär durch Abhängigkeiten von Modules untereinander bestimmt. Modules, die von vielen anderen Modules verwendet werden, können deutlich schwieriger entfernt werden als solche mit weniger Abhängigkeiten.

Metrik
Um Abhängigkeiten zu erfassen, wird der Module Use Factor (MUF) eingeführt. Tabelle D.5 zeigt, wie dieser gebildet wird. Einer Anzahl an Abhängigkeiten wird ein Wert des Faktors zugeordnet. Als Abhängigkeiten aus Sicht des Modules gelten Verwendungen von ausgehenden Messages und Parametern in anderen Modules.

$$M6 = \frac{\sum_{i=1}^{n} MUF_i}{n} \tag{D.8}$$

$$n \mathrel{\hat{=}} \textit{Gesamtzahl aller Modules}$$
$$MUF_i \mathrel{\hat{=}} \textit{Module Use Factor von Module } i$$

Tabelle D.5: Ermittlung des Module Use Factors (MUF)

# Abhängigkeiten	MUF
0	0
1	1
2	3
3-4	7
5-10	20
>10	50

Erläuterungen
Die Metrik der Skalierbarkeit ermittelt den MUF für alle Modules der Architektur und bildet dann den Durchschnittswert daraus. Die Metrik auf Module-Ebene anzuwenden ist zweckmäßig, da diese Bausteine per Definition entfernbar oder partitionierbar auf andere Hardware-Plattformen sein sollen. Es ist bemerkenswert, dass gegenüber den anderen bisher definierten Faktoren der MUF eine deutlich größere Steigung des Ausgabewertes über der Bezugsgröße aufweist. Der Grund ist, dass Modules, die ein bis zwei Abhängigkeiten erzeugen deutlich einfacher entfernt werden können als solche mit beispielsweise fünf oder mehr. Der Aufwand und die Wechselwirkungen nehmen nicht linear zu. Daher werden alle weiteren Abhängigkeiten bewusst stark bestraft. Der Wertebereich liegt zwischen 0 und 50.

Metrik der Speichereffizienz – M8

Idee
Ressourceneffizienz ist grundsätzlich erst dann vollständig ermittelbar und unterschiedliche Ansätze untereinander eindeutig vergleichbar, wenn der reale Ressourcenverbrauch des vollumfänglich implementierten Software-Systems auf seiner Ziel-Hardware vorliegt. Auf der Ebene der Software-Architektur, welche primär eine Struktur und (leere) Hüllen für die spätere Umsetzung darstellt, ist eine Einschätzung schwierig. Der Großteil der Software liegt zu diesem Zeitpunkt noch nicht vor. Viele Entscheidungen, die die Ressourceneffizienz beeinflussen, wurden noch nicht getroffen. Für eine grundsätzliche Abschätzung des Speicherbedarfs, insbesondere zum direkten Vergleich von Architektur-Alternativen, sollen je nach Reife des Entwurfes alle verfügbaren Informationen verwendet werden.

Metrik
Grundsätzlich werden mit der Metrik der Speichereffizienz alle zum gegenwärtigen Entwurfszeitpunkt bekannten, das heißt modellierten, Schnittstellenelemente mit einbezogen. Je nach Reife des Entwurfs können dies nur Messages und Parameter (grober Entwurf) bis hin zu Schnittstellen innerhalb Modules wie Übergabeparameter und lokale Variablen sein (feiner Entwurf oder Refactoring bestehendes Software-System). Die Metrik geht dabei von außen nach innen vor und zählt dabei alle unterschiedlichen Schnittstellenelemente jeweils genau einmal. Anders als bei den übrigen Metriken wird somit der Ausgabewert nicht relativ zu Bezugsgrößen angegeben sondern absolut ermittelt. Dies erklärt die eingangs angesprochene Sonderstellung dieser Metrik.

$$M8 = \sum_{i=1}^{n}(\sum_{j=1}^{a} S_{Mess_j} + \sum_{j=1}^{b} S_{Para_j} + \sum_{j=1}^{c} S_{ParaVal_j} + \sum_{j=1}^{d} S_{LocVar_j}) \quad \text{(D.9)}$$

$n \mathrel{\hat{=}}$ *Gesamtzahl aller Modules*
$a \mathrel{\hat{=}}$ *Anzahl der Messages im aktuellen Module*
$b \mathrel{\hat{=}}$ *Anzahl der Parameter im aktuellen Module*
$c \mathrel{\hat{=}}$ *Anzahl der Übergabe-Parameter im aktuellen Module*
$d \mathrel{\hat{=}}$ *Anzahl der lokalen Variablen im aktuellen Module*
$S_{Mess_j} \mathrel{\hat{=}}$ *Ressourcen-Äquivalent der aktuellen Message*
$S_{Para_j} \mathrel{\hat{=}}$ *Ressourcen-Äquivalent des aktuellen Parameters*
$S_{ParaVal_j} \mathrel{\hat{=}}$ *Ressourcen-Äquivalent des aktuellen Übergabe-Parameters*
$S_{LocVar_j} \mathrel{\hat{=}}$ *Ressourcen-Äquivalent der aktuellen lokalen Variablen*

Erläuterungen Im gewöhnlichen Architektur-Entwurf werden Schnittstellen nur bis zur Ebene der Übergabe-Parameter als Wechselwirkungen zwischen inneren Bausteinen

definiert. Lokale Variablen liegen nur in implementierten Software-Systemen vor und gehören nicht mehr zu typischen Informationen einer Software-Architektur. Nichtsdestotrotz soll M8 alle Schnittstellen und Variablen, die bekannt sind, sammeln und den Ressourcenbedarf bezüglich Speicherplatz aufaddieren. Es obliegt dem Anwender daraus Schlüsse zu ziehen. Zur Addition des Speicherbedarfs wird für jede Schnittstelle ein Ressourcen-Äquivalent verwendet, welches sich aus Typ der Schnittstelle und dem dazu hinterlegten Datentyp ermittelt.

E Neustrukturierung einer Software-Architektur

E.1 Vorbereitung der Neustrukturierung

Im Folgenden sind die wesentlichsten Schwachstellen der Ist-Struktur FAS-Längsführung tabellarisch aufgelistet. Einige der Schwachstellen sind grundsätzlicher Art, diese sind nicht weiter gekennzeichnet. Andere sind konkret auf das eingesetzte Werkzeug ASCET zurückzuführen und damit mit (W) gekennzeichnet.

Tabelle E.1: Identifizierte Schwachstellen der Ist-Struktur FAS-Längsführung

Kategorie	Schwachstelle
Struktur-Elemente	Leichen (Ausgangsschnittstellen des Bausteins nicht verwendet) von unterschiedlichen Strukturelementtypen in hoher Zahl vorhanden.
	OUTput-Modul wird nicht analog zum INPut-Modul verwendet. → Inkonsistenz
	Bausteine abhängig von Realisierungsdetails, wo es nicht nötig wäre.
	Keine unterschiedlichen Ausbaustufen/Konfigurationsmöglichkeiten von Bausteinen, die in unterschiedlichem Kontext benötigt werden.
	Kaum modularer Aufbau und keine saubere Trennung von Verantwortlichkeiten.
	Keine Bildung von Hierarchie-Ebenen zur Erhöhung der Verständlichkeit.
	Extrem unterschiedliche Größe von Bausteinen gleichen Typs auf gleicher Ebene.
	Anteile zur Anpassung an Varianten dauerhaft in Software enthalten, die sich zur Fahrzeug-Lebenszeit niemals ändern können, z.B. Typen von Bedienelementen.
	Identische oder sehr ähnliche Teilfunktionalitäten mehrfach und unterschiedlich umgesetzt.
	Zielstruktur LDM [110] nicht konsequent umgesetzt sondern über mehrere dezentrale Längsregler verwässert.
Schnitt-stellen	Bausteine haben keine Flexibilität/Robustheit bzgl. Vorhanden-/Nichtvorhandensein von Eingangsschnittstellen → Keine Ausbaustufen/Skaliervarianten vorhanden.
	Keine Trennung und Kennzeichnung von Bausteinen mit komplett realisierungsunabhängigen Schnittstellen von solchen mit (teilweise oder ausschließlich) realisierungsabhängigen.
	Kopplung/Abhängigkeiten zwischen Bausteinen sehr hoch, davon viele funktional unnötig.
	Mehrfaches Schreiben von Messages macht Kommunikation intransparent.
	(W): Parameter werden direkt in Sub-Bausteinen verwendet (importiert). → unterwandert das Schnittstellenkonzept und macht Kommunikation intransparent.
	(W): Messages dürfen nur skalare und keine 1- oder 2-dimensionalen Datenstrukturen sein.
	(W): Häufige Verwendung von globalen Variablen statt Messages. Somit können weitere Abhängigkeiten nicht überblickt werden, macht Kommunikation intransparent.
	(W): Keine grafische Visualisierung des Signalflusses und damit der Abhängigkeiten von Bausteinen möglich. → Nur durch aufwändige (manuelle) Analyse ermittelbar.
Imple-men-tierung	Unnötige lokale Variablen für Speicherung von Zwischenwerten werden verwendet.
	Konstrukte, deren Ergebnis von Berechnungen nirgends verwendet werden, sind vorhanden.
	Umständliche, ressourcenineffiziente Berechnungskonstrukte werden verwendet.

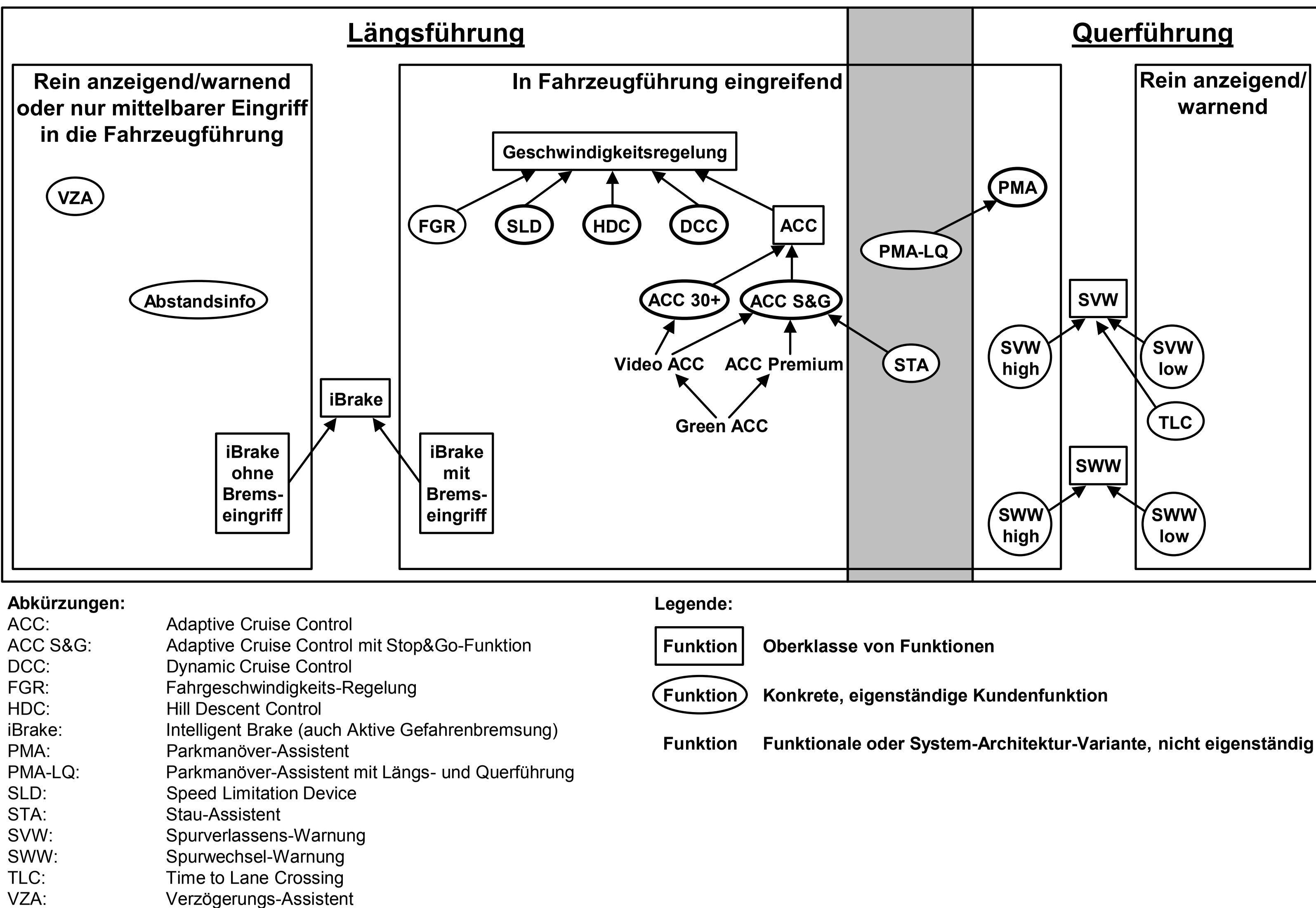

Bild E.1: Ordnungsschema für Fahrerassistenzsysteme

Tabelle E.2: Schema zur manuellen Bewertung von Software-Architekturentwürfen

Bezeichnung des Entwurfs		
Nr.	Kriterium	Bewertung
1	Indirekte Interaktionen	
2	Angebotsstruktur	
3	Technisches Sicherheitskonzept	
4	Ausprägungen (System-Architektur)	
5	Vorgegebene Schnittstellen	
6	Geschäftsmodelle	
7	Entwicklung, Verwendung, Wartung, Inbetriebnahme, Außerbetriebnahme	
8	Qualitätsmerkmale: Effizienz und Skalierbarkeit	
9	Qualitätsmerkmale: Wiederverwendbarkeit und Portabilität	
10	Qualitätsmerkmale: Wandlungsfähigkeit/Änderbarkeit	
11	Qualitätsmerkmale: Konformität zu verbindlichen Vorgaben	
12	(optional: Umsetzung der allgemeinen Strukturierungsvorgaben)	

Tabelle E.3: Zulässige Werte zur Befüllung der Bewertungstabelle

sehr gut (optimal)	++	2
gut	+	1
mittel	o	0
schlecht	−	-1
sehr schlecht (unzureichend, Veto für den Entwurf)	−−	-2

Die Befüllung mit Symbolen wie „+“ ist dabei für die von Menschen lesbare Dokumentation gedacht, die numerischen Werte für eine rechnerische Mittelwertbildung und Verarbeitung von Maschinen.

Ausschnitt der konkreten Anforderungen aus den funktionalen und nicht-funktionalen Einflussgrößen

Im Folgenden werden die konkreten Anforderungen und Randbedingungen gemäß der in Abschnitt 5.2.2 vorgestellte Kategorien aufgelistet. Aus jeder dieser Kategorien ist nur ein Teilumfang aufgelistet. Die Anforderungen sind mit einer Priorisierung versehen, wobei 1 die wichtigste Einstufung darstellt. Dies ist sinnvoll, da Anforderungen im Widerspruch stehen können und somit ein Hinweis zur Auflösung dieser Konflikte gegeben wird.

Tabelle E.4: Relevanz von Fahrerassistenzsystemen für die Neustrukturierung

Funktion	Prio	Begründung
VZA	n.r.	Greift nicht aktiv in die Längsführung ein, war und ist nicht Teil des Gesamtsystems.
DCC	1	Teil des heutigen Gesamtsystems.
iBrake	1	Teil des heutigen Gesamtsystems.
SLD	1	Teil des heutigen Gesamtsystems.
PMA	1	Teil des heutigen Gesamtsystems, greift jedoch nicht aktiv in die Längsführung ein, im Rahmen der Aktivierungskoordination zu berücksichtigen.
STA	2	Nicht Teil des heutigen Gesamtsystems, jedoch als nächste Generation bereits jetzt zu berücksichtigen.

Tabelle E.5: Funktionale Ausprägungen, die von der neuen Struktur abzubilden sind

ID	Anforderungstext	Prio
Fu_1.0	Die Software-Architektur muss die Umsetzung von ACC30+ ermöglichen.	1
Fu_2.0	Die Software-Architektur muss die Umsetzung von ACCSnG ermöglichen.	1

Tabelle E.6: Direkte Interaktionen, die von der neuen Struktur abzubilden sind

ID	Anforderungstext	Prio
Dir_Int_1.0	iBrake greift in die Fahrzeugführung nur über die Betriebsbremse ein.	1
Dir_Int_2.0	SLD greift in die Fahrzeugführung nur über Steuerung (im Sinne von Begrenzung) des Antriebsmomentes ein.	1
Dir_Int_10.0	iBrake ist immer aktivierbar und deaktivierbar.	1
Dir_Int_14.0	Wenn ein Anbremsen über iBrake erfolgen soll, so überstimmt dies eine aktive Anforderung von ACC, DCC, SLD, HDC.	1

Tabelle E.7: Nicht-funktionale Anforderungen, die von der neuen Struktur zu berücksichtigen sind

ID	Anforderungstext	Prio
Vorgegebene Schnittstellen auf technischer Ebene nach außen		
Ni_Fu_1.0	Zur Bremse darf nur die ECBA-Schnittstelle verwendet werden.	1
Ni_Fu_3.0	Zur Anzeige der Setzgeschwindigkeit müssen sich ACC, DCC, SLD und HDC den Scheibenzeiger sowie die Kammerleuchte und das Feld im HeadUp-Display teilen.	1
Ni_Fu_5.0	Zur akustischen „Anzeige“ des Take-Over-Request müssen sich ACC und iBrake dieselbe Schnittstelle zum Kombiinstrument teilen.	1
Geschäftsmodelle und Vorgaben zur organisatorischen und technischen Verteilung		
Ni_Fu_10.0	PMA muss im vereinbarten Geschäftsmodell „buy“ in die Software-Architektur integriert werden können.	1
Ni_Fu_10.1	PMA-LQ muss im vereinbarten Geschäftsmodell „make-und-buy“ in die Software-Architektur integriert werden können.	2
Entwicklung, Verwendung, Wartung, Inbetriebnahme und Außerbetriebnahme der Software(-Architektur)		
Ni_Fu_31.0	Die Software-Architektur muss mit AUTOSAR kompatibel sein. Dazu gehören die folgenden Aspekte:	2
Ni_Fu_31.1	Es muss eine eindeutige Zuordnung eines Bausteintyps der Software-Architektur zu „AUTOSAR Atomic Software Components“ geben.	2
Ni_Fu_31.2	Dieser Typ muss grundsätzlich in eine AUTOSAR-SWC-Description überführbar sein.	2
Ni_Fu_33.0	Das Software-System muss eine gute Fähigkeit zur Reglerabstimmung im Fahrzeug („Applikation“ über Parameter) aufweisen.	1
Produktfaktoren / Qualitätsmerkmale		
Ni_Fu_40.0	Die Software-Architektur soll eine gute Effizienz und Skalierbarkeit aufweisen.	1
Ni_Fu_40.1	Skalierbarkeit in beiden Richtungen verbessern: –einfaches und effizientes Entfernen von Teilfunktionalität. –einfaches und effizientes Erweitern bestehender Funktionalität.	1
Ni_Fu_40.2	Effizienter Code: –Speicherbedarf statisch & dynamisch dem Funktionsumfang angemessen. –Laufzeitbedarf dem Funktionsumfang angemessen.	1
Ni_Fu_40.3	Effizientes Herauslösen des DCC aus der Gesamtstruktur muss möglich sein: –keine unnötigen Abhängigkeiten in der Software. –keine unnötigen Fragmente von eigentlich unnötigen Bausteinen.	1
Ni_Fu_44.0	Die Software-Architektur soll eine gute Verständlichkeit aufweisen.	2
Ni_Fu_44.1	Die Software-Architektur soll auf allen Hierarchie-Ebenen möglichst verständlich sein.	2
Ni_Fu_44.2	Die Software-Architektur soll für alle relevanten Stakeholder möglichst verständlich sein.	2

Tabelle E.8: Ausprägungen aufgrund geänderter System-Architektur, die von der neuen Struktur abzubilden sind

ID	Anforderungstext	Prio
Var_Ang_5.3	Die Software-Architektur muss mit dem Multifunktionslenkrad und dem FAS-Bedienfeld zusammen spielen können.	1
Var_Ang_6.0	Die Software-Architektur muss mit dem Lenkstockhebel zusammenspielen können.	2

Tabelle E.9: Indirekte Interaktionen, die von der neuen Struktur abzubilden sind

ID	Anforderungstext	Prio
Ind_Int_2.0	ACC, DCC, HDC, iBrake und PMA-LQ müssen bzgl. Zugriff auf die ECBA-Schnittstelle koordiniert werden, da nur ein Sollwert umgesetzt werden kann.	1
Info	Abgeleitet von Ni_Fu_1.0	–
Ind_Int_4.0	In bestimmten Situationen fordern ACC Stop&Go und PMA-LQ das Stillstandsmanagement auf, den weiteren Stillstand zuverlässig sicher zu stellen.	1
Ind_Int_5.0	Der Zugriff auf das Stillstandsmanagement seitens des Systems muss koordiniert werden, da nur ein Sollwert (Kommunikationszustand) umgesetzt werden kann.	2

Tabelle E.10: Anforderungen von der Angebotsstruktur, die von der neuen Struktur abzubilden sind

ID	Anforderungstext	Prio
Var_Ang_1.0	Die Software-Architektur muss die Paketierungen, das heißt Angebote der Funktionen entweder einzeln oder in Kombination mit anderen, gemäß Bild 5.2 erfüllen.	1
Var_Ang_1.1	Die Software-Architektur muss die Umsetzung der DCC-Funktion in Verbindung mit SLD ermöglichen.	1
Var_Ang_1.2	Die Software-Architektur muss die Umsetzung der DCC-Funktion in Verbindung mit SLD und HDC ermöglichen.	1
Var_Ang_1.3	Die Software-Architektur muss die Umsetzung der DCC-Funktion in Verbindung mit SLD und PMA ermöglichen.	1
Var_Ang_1.4	Die Software-Architektur muss die Umsetzung der DCC-Funktion in Verbindung mit SLD, HDC und PMA ermöglichen.	1

E.2 Durchführung der Neustrukturierung

Die Schneidung nach wiederkehrenden Teilaufgaben (Variante 1b)

Variante 1b ist sehr ähnlich zu Variante 1a. Der Hauptunterschied ist ein Verschmelzen der Subsysteme Koordination Aktoren und Systemübergreifende Koordination & Ansteuerung Aktoren zu einem einzigen. Die grundsätzlichen Eigenschaften von Variante 1a bleiben, wie im Hauptteil beschrieben, erhalten. Durch die Fusion wird weitere externe Kommunikation (Kopplung) reduziert, gleichzeitig wird das neue Subsystem jedoch deutlich umfangreicher und komplexer.

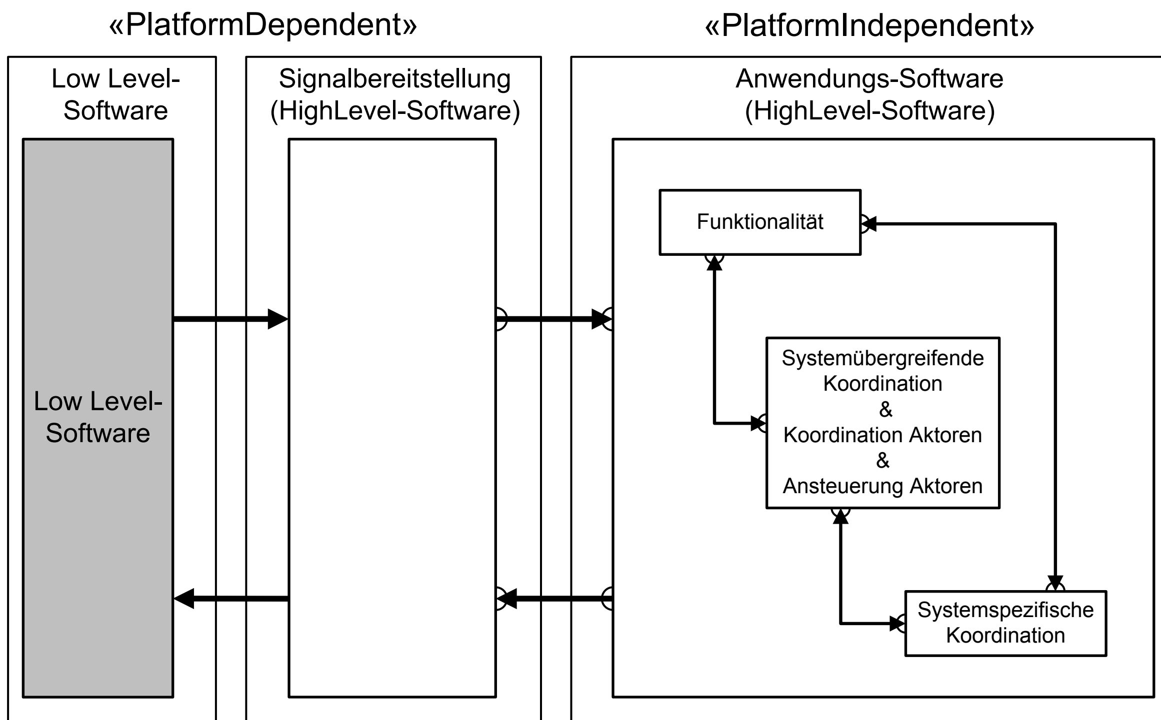

Bild E.2: Architekturstil – Variante 1b

Eine mögliche Variante 3 könnte durch eine Mischung aus Variante 1(a) und Variante 2 gebildet werden, ist hier aber nicht näher ausgeführt. Würde in die Struktur von Variante 2 ein Subsystem Koordination Aktoren hinzugefügt werden, scheint der Gedanke der Skalierbarkeit und Abstraktion von konkreten Aktoren möglich. Eine erste Bewertung hat jedoch ergeben, dass diese Variante 3 die Nachteile – und nicht die Vorteile – der Varianten 1a und 2 kombiniert und daher nicht erstrebenswert ist.

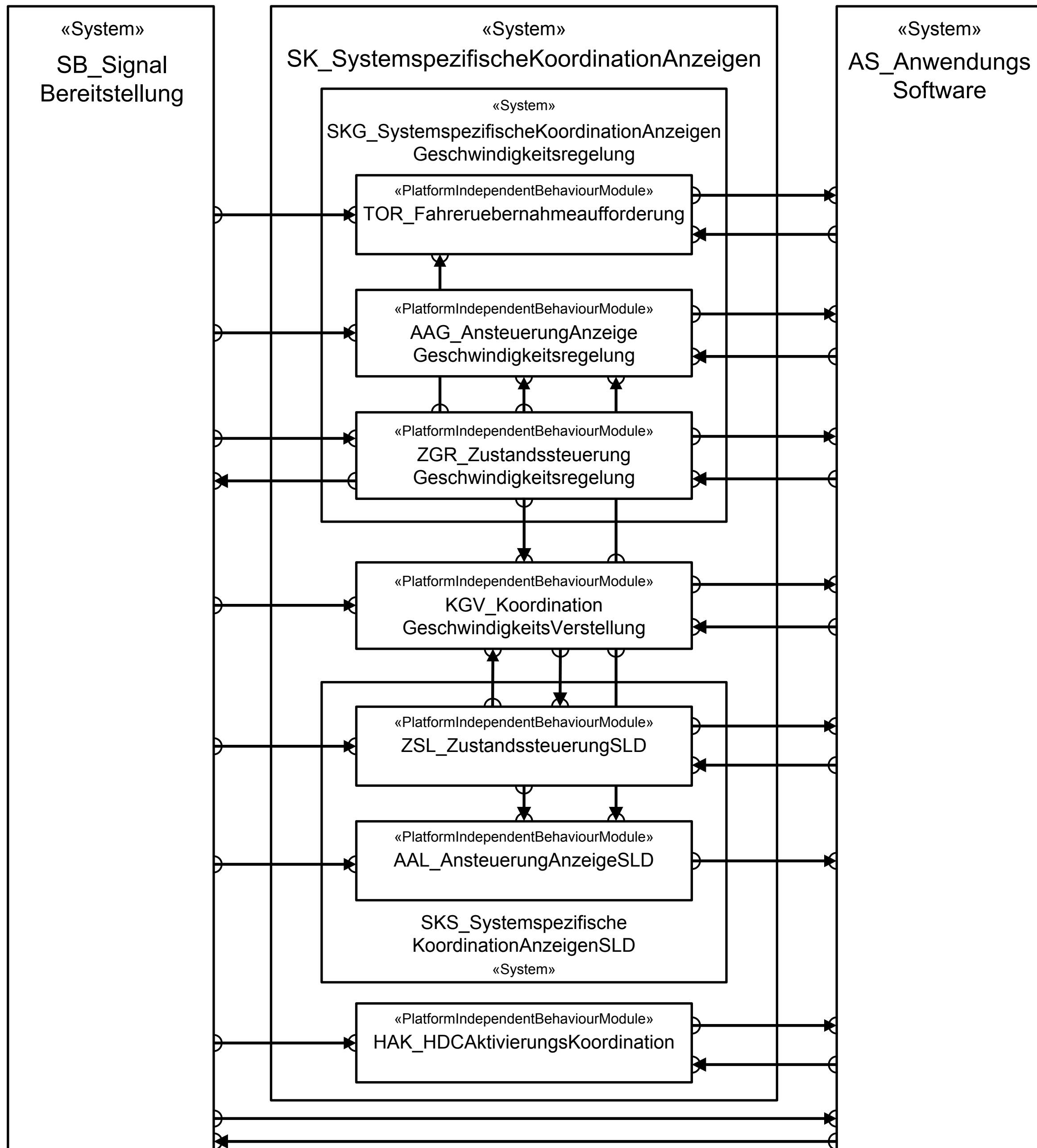

Bild E.3: Iteration 2 – innere Struktur von SystemspezifischeKoordinationAnzeigen

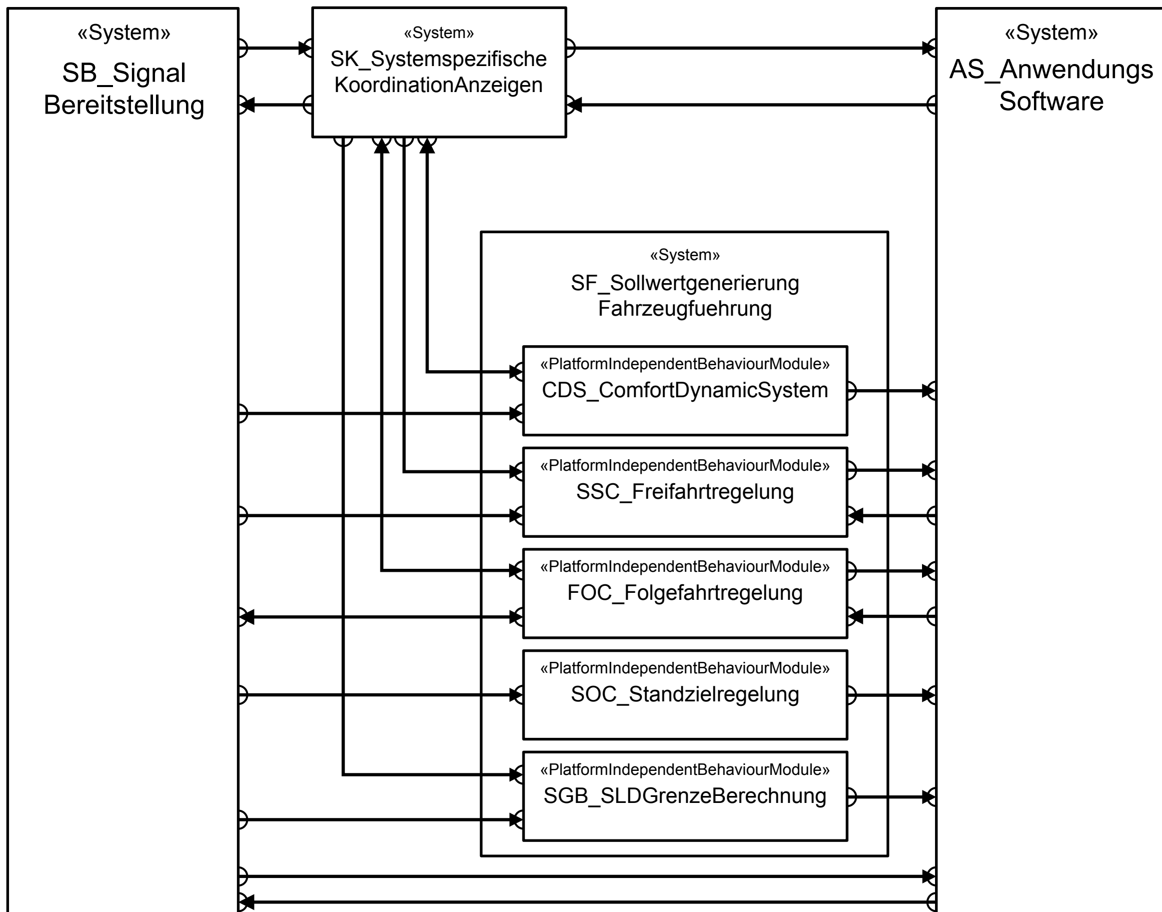

Bild E.4: Iteration 2 – innere Struktur von SF_SollwertgenerierungFahrzeugführung

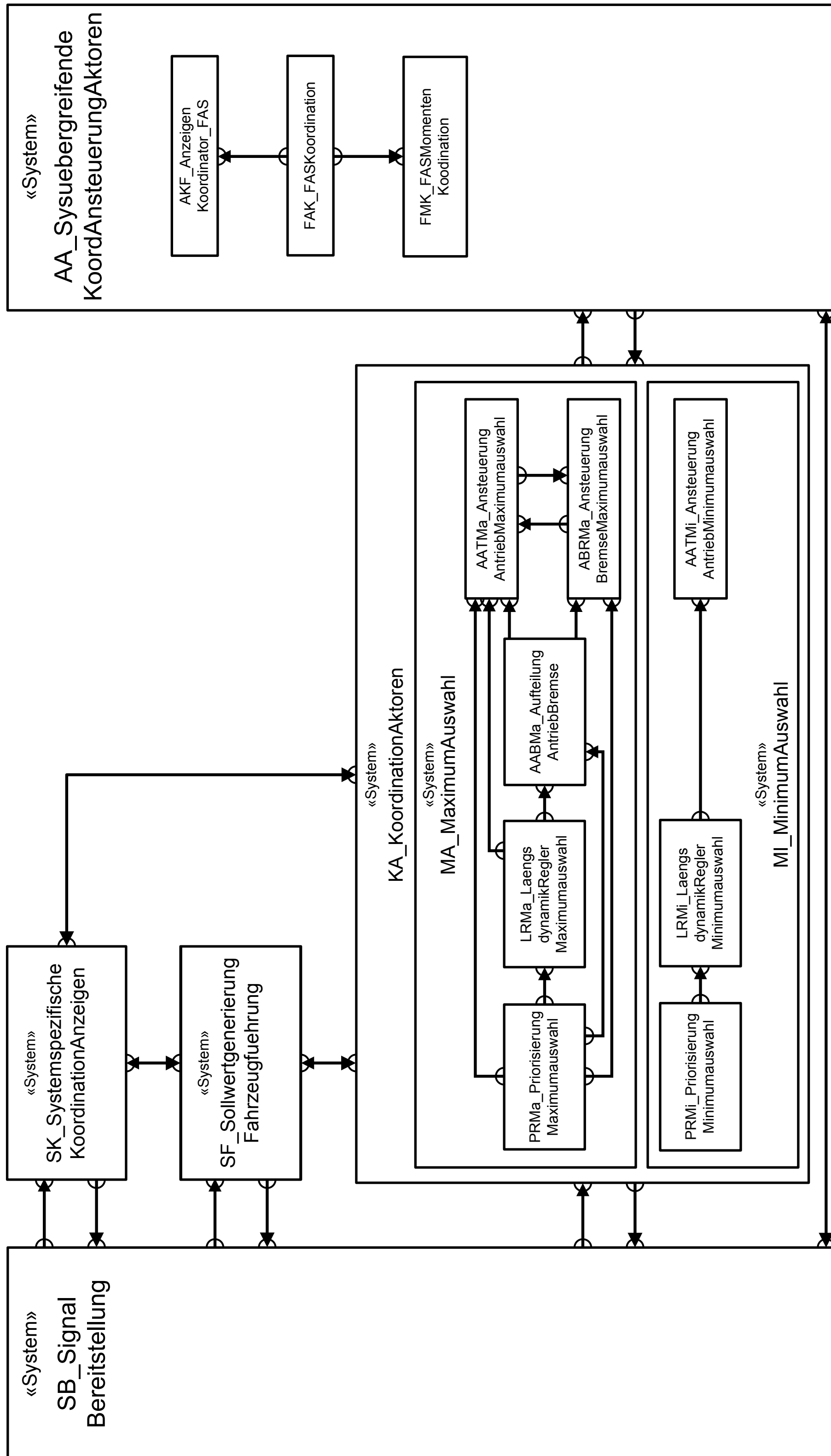

Bild E.5: Iteration 3 – innere Struktur von KoordinationAktoren und SysuebergreifendeKoordAnsteuerungAktoren

Tabelle E.11: Ergebnisse der Software-Architekturmetriken über die gesamte Neustrukturierung

				Ausgangs-Stand	Iteration 1		Iteration 2		Iteration 3		Endstand	
Metrik		Werte-Bereich	Gewichtung	Abs. Wert	Abs. Wert	Rel. Änd.(%)	Abs. Wert	Rel. Änd.(%)	Abs. Wert	Rel. Änd.(%)	Abs. Wert	Rel. Änd.(%)
Strukturverständlichkeit	M1	0-10	1	6,18	2,84	-54,04	2,25	-63,63	1,19	-80,78	0,54	-91,33
Bausteinverständlichkeit	M2	0-60	1	13,68	11,71	-14,40	10,88	-20,50	10,91	-20,25	10,64	-22,24
Änderbarkeit	M3	0-240	3	7,23	6,39	-11,66	5,70	-21,12	5,58	-22,82	6,04	-16,42
Stabilität	M4	0-1	1	0,50	0,57	14,55	0,57	13,44	0,55	10,49	0,56	12,18
Testbarkeit	M5	0-240	2	28,34	30,00	5,85	27,54	-2,80	27,75	-2,07	27,70	-2,25
Skalierbarkeit	M6	0-50	1	23,23	16,02	-31,03	16,25	-30,04	16,18	-30,33	16,18	-30,33
Modularität	M7	0-1	1	0,00	0,19	∞	0,23	∞	0,24	∞	0,26	∞
Qualitätskriterien												
Wandlungsfähigkeit	M1-M5		2			-9,65		-17,46		-20,39		-19,39
Effizienz	M6		2			-31,03		-30,04		-30,33		-30,33
Wiederverwendbarkeit	M7		1			∞		∞		∞		∞
Schnittstellenelemente (nur relative Änderungen dargestellt)												
Messages						-16,35		-14,76		-15,77		-26,63
Übergabeparameter						-4,62		-0,43		-1,65		-1,72
Applikationsparameter						-0,60		-0,95		-1,07		-1,19
Codierparameter						-29,90		-30,93		-30,93		-34,02
Globale Variablen						-95,37		-95,37		-95,37		-95,37
Lokale Variablen						13,53		12,39		13,08		15,63

Im Rahmen dieser Arbeit sind die folgenden Veröffentlichungen entstanden:

[V1] Ahrens, D. ; Pfeiffer, A. ; Bertram, T. : Comparison of ASCET and UML – Preparations for an Abstract Software Architecture. In: *Forum on specification and Design Languages, FDL 2008,* Electronic Chips & Systems Design Initiative (ECSI), Stuttgart, 2008

[V2] Ahrens, D. ; Frey, A. ; Pfeiffer, A. ; Bertram, T. : Entwicklung einer leistungsfähigen Darstellung für komplexe Funktions- und Softwarearchitekturen im Bereich Fahrerassistenz. In: *VDI Mechatronik 2009, Wiesloch,* VDI-Verlag, 2009

[V3] Ahrens, D. ; Frey, A. ; Pfeiffer, A. ; Bertram, T. : Designing Reusable and Scalable Software Architectures for Automotive Embedded Systems in Driver Assistance. In: *SAE 2010 World Congress & Exhibition,* Society of Automotive Engineers (SAE), Detroit, USA, 2010

[V4] Ahrens, D. ; Frey, A. ; Pfeiffer, A. ; Bertram, T. : Entwicklung eines objektiven Bewertungsverfahrens für Softwarearchitekturen im Bereich Fahrerassistenz. In: Engels, G. (Hrsg.) ; Luckey, M. (Hrsg.) ; Schäfer, W. (Hrsg.): *Software Engineering 2010,* GI-Edition – Lecture Notes in Informatics (LNI), Bd. P-159, Bonner Köllen Verlag, 2010

[V5] Ahrens, D. ; Frey, A. ; Pfeiffer, A. ; Bertram, T. : Restructuring and Optimization of Software Architectures for Longitudinal Driver Assistance Systems. In: *FISITA 2010 World Automotive Congress,* The International Federation of Automotive Engineering Societies (FISITA), 2010

[V6] Ahrens, D. ; Frey, A. ; Pfeiffer, A. ; Bertram, T. : Objective evaluation of software architectures in driver assistance systems; Methods − quality model – metrics. In: *Computer Science – Research and Development* 28 (Special Issue) (2013), Nr. 1, S. 23–43

Des Weiteren wurden die folgenden Studienabschluss-Arbeiten betreut:

[S1] Heinichen, J. : *Entwicklung eines objektiven Bewertungsverfahrens für Softwarearchitekturen im Bereich Fahrerassistenz,* Technische Universität Chemnitz, 2008

[S2] Resvoll, E. : *Modelltransformationen für Embedded Software – Konzeptvergleich und Ansätze,* Universität Karlsruhe (TH), 2008

[S3] Grunow, S. : *Modellierung von Variabilitätsinformationen von Automotive Embedded Software-Architekturen,* Technische Universität München, 2009

Abbildungsverzeichnis

1.1 Typische Bestandteile der Wirkkette eines Fahrerassistenzsystems 2
1.2 Beispielhafte Modellierung mit der CARTRONIC-Notation 8
1.3 Software-Architektur eines Navigations-Systems 12
1.4 Vereinfachte Darstellung der AUTOSAR Schichten-Architektur 15

2.1 Funktions-Architektur ACC mit Stop&Go-Funktion – oberste Ebene . . . 27
2.2 Funktions-Architektur ACC mit Stop&Go-Funktion – Längsführung . . . 28
2.3 Software-Architektur ACC mit Stop&Go-Funktion – Blackbox-Sicht . . . 31
2.4 Betrachteter Software-Umfang des ACC Stop&Go – Whitebox-Sicht . . . 32

3.1 Phasenmodell zur Entwicklung von eingebetteter Software im Automobil 34
3.2 Einflussgrößen auf Software-Architektur am Beispiel FAS 39
3.3 Betrachtungsebenen auf das Fahrzeug in der (Software-)Entwicklung . . . 40
3.4 Angepasstes Vorgehensmodell zur Entwicklung 44

4.1 Vorgehen zur Entwicklung der ABSOFA 46
4.2 Gesamtüberblick über den Umfang der ABSOFA-Modellierung 49
4.3 Standpunkte, Sichten und Diagramme für die ABSOFA-Modellierung . . 50
4.4 Q-Modell zur Klassifizierung und Bewertung von Software-Architektur . 55
4.5 Übersicht über die entwickelten Software-Architekturmetriken 56
4.6 Akademisches Beispiel zur Berechnung der Metrik M7 (Modularität) . . . 58
4.7 Meta-Ebenen der Variantenmodellierung in der ABSOFA 62
4.8 Das zentrale Datenmodell im Kontext der Werkzeuge und Sichten 65
4.9 Anzahl notwendiger Regelsätze für Modelltransformationen 66
4.10 Die Grundstruktur des Zwischenproduktes als XML-Schema 67
4.11 Der Ablauf des JAXB-Transformations- und Analyseansatzes 69
4.12 Anwendung des Zwischenprodukts 72

5.1 Vorgehen zum iterativen Entwurf von Software-Architekturen 80
5.2 Angebotsstruktur von Fahrerassistenzsystemen im BMW 5er 85
5.3 Grundsätzlicher Strukturrahmen des Architekturstils 87
5.4 Architekturstil – Variante 1a 88
5.5 Architekturstil – Variante 2 90
5.6 Iteration 1 – innere Struktur von SB_Signalbereitstellung 92
5.7 Relevante Ausbaustufen für die Architekturbausteine 93
5.8 Architekturstil – Variante 1c 95

5.9 Iteration 3 – Grundidee von KA_KoordinationAktoren 97

6.1 Für das Fallbeispiel ausgewählte Ausbaustufe der Software-Architektur . 106

A.1 Die unterschiedlichen Kategorien der Verwendung des Datenmodells . . . 133

B.1 Gesamtüberblick über die Strukturelemente der ABSOFA 135
B.2 Der Module-Typ und alle seine Subtypen 137
B.3 Der InterfaceElement-Typ 140
B.4 Beziehungen zwischen Schnittstellenelementen 141
B.5 Modellierung von Variabilität – Der VariationPoint 142
B.6 Modellierung von Variabilität – Optionen, Varianten und Beziehungen . . 143

E.1 Ordnungsschema für Fahrerassistenzsysteme 160
E.2 Architekturstil – Variante 1b 164
E.3 Iteration 2 – innere Struktur von SK 165
E.4 Iteration 2 – innere Struktur von SF 166
E.5 Iteration 3 – innere Struktur von KA und AA 167

Tabellenverzeichnis

1.1 Benennung und Klassifizierung automatisierter Fahrfunktionen 6
1.2 Gegenüberstellung Architektur-Beschreibungen und Vorgehensmodelle . . 13

4.1 Initiale Gewichtung aller Qualitätskriterien und -attribute 60

5.1 Subjektive Bewertung von Ausgangsstand und Endstand 99

6.1 Ergebnisse der Analyse des Ressourcenbedarfs beim Fallbeispiel 108

A.1 Wesentliche Anforderungen an das zentrale Datenmodell 131
A.2 Wesentliche Anforderungen an den Ansatz der ABSOFA 134

B.1 Einschränkungen und Regeln der Strukturelemente der ABSOFA 139
B.2 Einschränkungen und Regeln der Schnittstellen der ABSOFA 142
B.3 Einschränkungen und Regeln zur Modellierung von Variabilität 143

D.1 Ermittlung des Component Complexity Factors (CCF) 150
D.2 Faktoren zur Ermittlung der Bausteinverständlichkeit 151
D.3 Faktoren zur Ermittlung der Änderbarkeit 152
D.4 Ermittlung des Input Quantity Factors (IQF) 155
D.5 Ermittlung des Module Use Factors (MUF) 156

E.1 Identifizierte Schwachstellen der Ist-Struktur FAS-Längsführung 159
E.2 Schema zur manuellen Bewertung von Software-Architekturentwürfen . . 161
E.3 Zulässige Werte zur Befüllung der Bewertungstabelle 161
E.4 Relevanz von Fahrerassistenzsystemen für die Neustrukturierung 161
E.5 Funktionale Ausprägungen, die von der neuen Struktur abzubilden sind . 162
E.6 Direkte Interaktionen, die von der neuen Struktur abzubilden sind 162
E.7 Nicht-funktionale Anforderungen an die neue Struktur 162
E.8 Ausprägungen aufgrund System-Architektur der neuen Struktur 163
E.9 Indirekte Interaktionen, die von der neuen Struktur abzubilden sind . . . 163
E.10 Anforderungen von der Angebotsstruktur an die neue Struktur 163
E.11 Ergebnisse der Software-Architekturmetriken für die Neustrukturierung . 168

Literaturverzeichnis

[1] *AUTOSAR – AUTomotive Open System ARchitecture.* Release 4.0 – V1.2.1, 19.01.2012. http://www.autosar.org, Abruf: 26.07.2015

[2] Abowd, G. ; Bass, L. ; Clements, P. ; Kazman, R. ; Northrop, L. ; Zaremski, A. : Recommended Best Industrial Practice for Software Architecture Evaluation / Software Engineering Institute, Carnegie Mellon University, USA. 1996 (CMU/SEI-96-TR-025). – Forschungsbericht

[3] Ahrens, D. : Parkassistent mit Längs- und Querführung. In: *5. Tagung Fahrerassistenz.* München, Mai 2012

[4] Ahrens, D. ; Frey, A. ; Pfeiffer, A. ; Bertram, T. : Designing Reusable and Scalable Software Architectures for Automotive Embedded Systems in Driver Assistance. In: *SAE 2010 World Congress & Exhibition*, Society of Automotive Engineers (SAE), Detroit, USA, 2010

[5] Ahrens, D. ; Frey, A. ; Pfeiffer, A. ; Bertram, T. : Entwicklung eines objektiven Bewertungsverfahrens für Softwarearchitekturen im Bereich Fahrerassistenz. In: Engels, G. (Hrsg.) ; Luckey, M. (Hrsg.) ; Schäfer, W. (Hrsg.): *Software Engineering 2010*, GI-Edition – Lecture Notes in Informatics (LNI), Bd. P-159, Bonner Köllen Verlag, 2010

[6] Ahrens, D. ; Frey, A. ; Pfeiffer, A. ; Bertram, T. : Restructuring and Optimization of Software Architectures for Longitudinal Driver Assistance Systems. In: *FISITA 2010 World Automotive Congress*, The International Federation of Automotive Engineering Societies (FISITA), 2010

[7] Ahrens, D. ; Frey, A. ; Pfeiffer, A. ; Bertram, T. : Objective evaluation of software architectures in driver assistance systems; Methods – quality model – metrics. In: *Computer Science – Research and Development* 28 (Special Issue) (2013), Nr. 1, S. 23–43

[8] Ahrens, D. ; Pfeiffer, A. ; Bertram, T. : Comparison of ASCET and UML – Preparations for an Abstract Software Architecture. In: *Forum on specification and Design Languages (FDL), Stuttgart*, Electronic Chips & Systems Design Initiative (ECSI), 2008

[9] BABAR, M. A. ; ZHU, L. ; JEFFERY, R. : A Framework for Classifying and Comparing Software Architecture Evaluation. In: *ASWEC '04 Proceedings of the 2004 Australian Software Engineering Conference*, IEEE Computer Society, Washington D.C., USA, 2004, S. 309–318

[10] BALZERT, H. : *Lehrbuch der Objektmodellierung. Analyse und Entwurf.* 1. Auflage. Spektrum Akademischer Verlag, Heidelberg, 1999

[11] BALZERT, H. : *Lehrbuch der Software-Technik 1.* 1. Auflage. Spektrum Akademischer Verlag, Heidelberg, 1998

[12] BALZERT, H. : *Lehrbuch der Software-Technik 2.* 1. Auflage. Spektrum Akademischer Verlag, Heidelberg, 2001

[13] BAUER, A. ; FREUND, U. ; MATA, N. ; PHILIPPS, J. ; ROMBERG, J. ; SCHÄTZ, B. ; SLOTOSCH, O. : AutoMoDe – Automotive Model Based Development. In: *Eröffnungskonferenz Forschungsoffensive Software Engineering 2006*, Bundesministerium für Bildung und Forschung, Berlin, 2004

[14] BEAUFTRAGTER DER BUNDESREGIERUNG FÜR INFORMATIONSTECHNIK: *Das V-Modell XT – Version 1.4*. `http://www.v-modell-xt.de/`, Abruf: 26.07.2015

[15] BECK, K. : *Extreme Programming – die revolutionäre Methode für Softwareentwicklung in kleinen Teams.* 2. Auflage. Addison-Wesley, Boston, USA, 2003

[16] BECK, K. ; JOHNSON, R. : Patterns Generate Architectures. In: *Eigth European Conference on Object-Oriented Programming – ECOOP '94, Bologna, Spanien*, Springer-Verlag, Berlin, 1994, S. 139–149

[17] BEECK, M. ; BRAUN, P. ; RAPPL, M. ; SCHRÖDER, C. : Automotive UML: A (Meta) Model-Based Approach for Systems Development. In: SELIC, B. (Hrsg.) ; MARTIN, G. (Hrsg.) ; LAVAGNO, L. (Hrsg.): *UML for Real: Design of Embedded Real-Time Systems.* Kluwer Academic Publishers, Boston, USA, 2003, Kapitel 13, S. 271–299

[18] BENGTSSON, P.-O. ; BOSCH, J. : Scenario-Based Software Architecture Reengineering. In: *Proceedings of the 5th International Conference on Software Reuse*, IEEE Computer Society, Washington D.C., USA, 1998, S. 308–317

[19] BERTRAM, T. ; SCHRÖDER, W. ; DOMINKE, P. ; VOLKART, A. : CARTRONIC – ein Ordnungskonzept für die Steuerungs- und Regelungssysteme in Kraftfahrzeugen. In: *Systemengineering in der KFZ-Entwicklung*, VDI-Berichte Bd. 1374, VDI-Verlag, Düsseldorf, 1997, S. 369–397

[20] BLOM, H. ; LÖNN, H. ; HAGL, F. ; PAPADOPOULOS, Y. ; REISER, M.-O. ; SJÖSTEDT, C.-J. ; CHEN, D.-J. ; KOLAGARI, R. T.: *EAST-ADL – An Architecture Description Language for Automotive Software-Intensive Systems. White Paper, Version M2.1.10.* 2012

[21] BOEHM, B. W. ; BROWN, J. R. ; KASPAR, H. ; LIPOW, M. ; MACLEOD, G. J. ; MERRIT, M. J.: *Characteristics of Software Quality.* 1. Auflage. Elsevier Science Ltd., Amsterdam, Niederlande, 1978

[22] BOSCH, J. : *Design and Use of Software Architectures.* Addison-Wesley, Harlow, Großbritannien, 2000

[23] BOSCH, J. ; BENGTSSON, P.-O. : Assessing Optimal Software Architecture Maintainability. In: *Fifth European Conference on Software Maintenance and Reengineering, Lissabon, Portugal*, IEEE Computer Society, Washington D.C., USA, 2001, S. 168–175

[24] BRAESS, H.-H. : Nichts steigt so schnell wie Ansprüche – Gedanken zur weiteren Entwicklung des Personenwagens. In: *Automobiltechnische Zeitschrift – ATZ* (1993), Nr. 9, S. 452–458

[25] BRAESS, H.-H. (Hrsg.) ; SEIFFERT, U. (Hrsg.): *Vieweg Handbuch Kraftfahrzeugtechnik.* 7. Auflage. Springer Vieweg Verlag, Wiesbaden, 2013

[26] BROY, M. : Challenges in Automotive Software Engineering. In: *Proceedings of the 28th International Conference on Software Engineering, ICSE '06, New York, USA*, ACM – Association for Computing Machinery, New York, USA, 2006

[27] BUNDESMINISTERIUM DES INNEREN, KOORDINIERUNGS- UND BERATUNGSSTELLE DER BUNDESREGIERUNG FÜR INFORMATIONSTECHNIK IN DER BUNDESVERWALTUNG: *V-Modell – Entwicklungsstandard für IT-Systeme des Bundes.* 1997

[28] BUSCHMANN, F. ; MEUNIER, R. ; ROHNERT, H. ; SOMMERLAD, P. ; STAL, M. : *Pattern-Oriented Software Architecture – A System of Patterns.* Bd. 1. John Wiley & Sons Ltd, Chichester, Großbritannien, 1996

[29] CLEMENTS, P. ; BACHMANN, F. ; BASS, L. ; GARLAN, D. ; IVERS, J. ; LITTLE, R. ; NORD, R. ; STAFFORD, J. : *Documenting Software Architectures: Views and Beyond.* 7. Auflage. Addison-Wesley, Boston, USA, 2005

[30] CLEMENTS, P. ; KAZMAN, R. ; KLEIN, M. : *Evaluating Software Architectures.* Addison-Wesley, Boston, USA, 2002

[31] COCKBURN, A. : *Agile Software Development.* 1. Auflage. Addison-Wesley, Boston, USA, 2002

[32] Damm, W. ; Votintseva, A. ; Metzner, A. ; Josko, B. ; Peikenkamp, T. ; Böde, E. : Boosting Re-use of Embedded Automotive Applications Through Rich Components. In: *Proceedings FIT 2005 – Foundations of Interface Technologies, San Francisco, USA*, 2005

[33] DIN – Deutsches Institut für Normung: *DIN70000:1994-01 – Straßenfahrzeuge; Fahrzeugdynamik und Fahrverhalten; Begriffe.* 1994

[34] Donges, E. : Aspekte der aktiven Sicherheit bei der Führung von Personenkraftwagen. In: *Automobil-Industrie* 27 (1982), Nr. 2, S. 183–190

[35] Draeger, K. : Das Automobil in einer vernetzten Welt. In: *ATZ extra – 125 Jahre Automobil* (2011), S. 22–26

[36] ETAS GmbH: *ASCET MD/SE, Version 6.1.0.* http://www.etas.com/de/, Abruf: 26.07.2015

[37] Feiler, P. H. ; Lewis, B. ; Vestal, S. : The SAE Avionics Architecture Description Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering. In: *RTAS 2003 Workshop on Model-Driven Embedded Systems, Washington D.C., USA*, 2003

[38] Florentz, B. : Inside Architecture Evaluation: Analysis and Representation of Optimization Potential. In: *Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA'07), Mumbai, Indien*, IEEE Computer Society, Washington D.C., USA, 2007, S. 3

[39] Florentz, B. ; Huhn, M. : Embedded Systems Architecture: Evaluation and Analysis. In: *Lecture Notes in Computer Science, Quality of Software Architectures* Bd. 4214. Springer-Verlag, Berlin, 2006, S. 145–162

[40] Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J. : *Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley, München, 2008

[41] Gasser, T. M.: Ergebnisse der Projektgruppe Automatisierung: Rechtsfolgen zunehmender Fahrzeugautomatisierung. In: *5. Tagung Fahrerassistenz.* München, Mai 2012

[42] Grönniger, H. ; Hartmann, J. ; Krahn, H. ; Kriebel, S. ; Rothhardt, L. ; Rumpe, B. : Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In: *4th European Congress ERTS – Embedded Real Time Software, Toulouse, Frankreich*, 2008

[43] Gurp, J. van ; Bosch, J. : *Automating Software Architecture Assessment.* 2000

[44] Hachmeister, M. : Informationstransfer von UML nach ASCET. In: *ETAS RealTimes* (2007), Nr. 1, S. 24–27

[45] Handmann, U. ; Leefken, I. ; Tzomakas, C. : Eine flexible Architektur für Fahrerassistenzsysteme. In: *Mustererkennung 1999, 21. DAGM-Symposium, Bonn*, Springer-Verlag, Berlin, 1999, S. 36–43

[46] Hardung, B. ; Kölzow, T. ; Krüger, A. : Reuse of Software in Distributed Embedded Automotive Systems. In: *4th International Conference on Embedded Software – EMSOFT '04, Pisa, Italien*, ACM – Association for Computing Machinery, New York, USA, 2004, S. 203–210

[47] Heinecke, H. ; Damm, W. ; Josko, B. ; Metzner, A. ; Kopetz, H. : Software Components for Reliable Automotive Systems. In: *Proceedings of the 2008 Conference on Design, Automation and Test in Europe (DATE '08), München*, IEEE Computer Society, Washington D.C., USA, 2008, S. 549–554

[48] Hofmeister, C. ; Nord, R. ; Soni, D. : *Applied Software Architecture*. 3. Auflage. Addison-Wesley, Reading, USA, 2001

[49] Hopcroft, J. E. ; Ullman, J. : *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. 2. Auflage. Addison-Wesley, Bonn, 1990

[50] Hruschka, P. ; Rupp, C. : *Agile Softwareentwicklung für Embedded Real-Time Systems mit der UML*. 1. Auflage. Hanser Verlag, München, 2002

[51] Huber, F. ; Schätz, B. ; Einert, G. : Consistent Graphical Specification of Distributed Systems. In: Fitzgerald, J. (Hrsg.) ; Jones, C. B. (Hrsg.) ; Lucas, P. (Hrsg.): *FME '97 Industrial Applications and Strengthened Foundations of Formal Methods: 4th International Symposium of Formal Methods Europe, Graz, Österreich*, Springer-Verlag, Berlin, 1997, S. 122–141

[52] IBM: *Rational DOORS*. `http://www-03.ibm.com/software/products/de/ratidoor`, Abruf: 26.07.2015

[53] IEEE – Institute of Electrical and Electronics Engineers: *IEEE 1471 – Recommended Practice for Architectural Description of Software-Intensive Systems*. Sept. 2000

[54] ISO – International Organization for Standardization: *ISO/IEC 9126-1: Software Engineering – Product Quality*. 2001

[55] ISO – International Organization for Standardization: *DIN EN ISO 9001: Qualitätsmanagement*. 2009

[56] ISO – INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *ISO 15622:2010: Intelligent transport systems – Adaptive Cruise Control systems – Performance requirements and test procedures.* 2010

[57] ISO – INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *ISO 26262: Road vehicles – Functional safety.* 2011

[58] ITPOWER SOLUTIONS GMBH: *SimEx – Konvertierung von Simulink/Stateflow Modelldaten in das XML-Format.* `http://www.itpower.de/102-0-SimEx-Konvertierung-von-MATLABSimulink-Modelldaten-nach-XML.html`, Abruf: 26.07.2015

[59] JOHNSEN, A. ; LUNDQVIST, K. : Developing Dependable Software-Intensive Systems: AADL vs. EAST-ADL. In: *16th Ada-Europe International Conference on Reliable Software Technologies, Edinburgh, Großbritannien*, Springer-Verlag, Berlin, Juni 2011, S. 103–117

[60] KAZMAN, R. ; BASS, L. ; ABOWD, G. ; WEBB, M. : SAAM: A Method for Analyzing the Properties of Software Architectures. In: *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italien*, IEEE Computer Society Press, Los Alamitos, USA, 1994, S. 81–90

[61] KAZMAN, R. ; KLEIN, M. ; BARBACCI, M. ; LONGSTAFF, T. ; LIPSON, H. ; CARRIERE, J. : The Architecture Tradeoff Analysis Method. In: *Fourth IEEE International Conference on Engineering of Complex Computer Systems – ICECCS '98, Monterey, USA*, IEEE Computer Society, Washington D.C., USA, 1998, S. 68–78

[62] KECHER, C. : *UML 2.0 – Das umfassende Handbuch.* 1. Auflage. Galileo Press GmbH, Bonn, 2005

[63] KERNIGHAN, B. W. ; RITCHIE, D. (Hrsg.): *The C programming language.* 2. Auflage. Prentice Hall, Upper Saddle River, USA, 1988

[64] KLEINOD, E. : Modellbasierte Systementwicklung in der Automobilindustrie – Das MOSES-Projekt / Fraunhofer-Institut für Software und Systemtechnik. 2006. – Forschungsbericht

[65] KRAHN, H. ; RUMPE, B. : Grundlagen der Evolution von Software-Architekturen. In: REUSSNER, R. (Hrsg.) ; HASSELBRING, W. (Hrsg.): *Handbuch der Software-Architektur.* 2. Auflage. dpunkt.verlag GmbH, Heidelberg, 2009, Kapitel 8, S. 181–197

[66] KRUCHTEN, P. : The 4+1 View Model of Architecture. In: *IEEE Software* 12 (1995), Nr. 6, S. 42–50

[67] Kruchten, P. : *The Rational Unified Process: An Introduction.* 3. Auflage. Addison-Wesley Professional, Boston, USA, 2003

[68] Krüger, G. ; Hansen, H. : *Handbuch der Java-Programmierung.* 7. Auflage. Addison-Wesley, München, 2012

[69] Kugele, S. : *Model-Based Development of Software-intensive Automotive Systems*, Technische Universität München, Diss., 2012

[70] Kugele, S. ; Tautschnig, M. ; Bauer, A. ; Schallhart, C. ; Merenda, S. ; Haberl, W. ; Kühnel, C. ; Müller, F. ; Wang, Z. ; Wild, D. ; Rittmann, S. ; Wechs, M. : COLA – The component language / Institut für Informatik, Technische Universität München. 2007 (TUM-I0714). – Forschungsbericht

[71] Kühl, M. ; Reichmann, C. : Entwurfsbegleitende Modelltransformation. In: *ETAS RealTimes* (2006), Nr. 1, S. 20–22

[72] Kühl, M. ; Reichmann, C. ; Wolff, H. : Entwurfsbegleitende Modelltransformation. Automatisierung von Werkzeugübergängen durch den Einsatz von regelbasierten Modelltransformationen. In: *Steuerung und Regelung von Fahrzeugen und Motoren – AUTOREG 2006, Wiesloch*, VDI-Berichte Bd. 1931, VDI-Verlag, Düsseldorf, 2006, S. 751–758

[73] Lapp, A. ; Torre Flores, P. ; Schirmer, J. ; Kraft, D. ; Hermsen, W. ; Bertram, T. ; Petersen, J. : Softwareentwicklung für Steuergeräte im Systemverbund – Von der CARTRONIC-Domänenstruktur zum Steuergerätecode. In: *10. Internationaler Kongress Elektronik im Kraftfahrzeug, Baden-Baden*, VDI-Berichte Bd. 1646, VDI-Verlag, Düsseldorf, 2001, S. 249–276

[74] Losavio, F. ; Chirinos, L. ; Lévy, N. ; Ramdane-Cherif, A. : Quality Characteristics for Software Architecture. In: *Journal of Object Technology* 2 (2003), S. 133–150

[75] Lung, C.-H. ; Kalaichelvan, K. : An Approach to Quantitative Software Architecture Sensitivity Analysis. In: *International Journal of Software Engineering and Knowledge Engineering* 10 (2000), Nr. 1, S. 97–114

[76] Marwedel, P. : *Eingebettete Systeme.* 1. Auflage. Springer-Verlag, Berlin, 2007

[77] MathWorks Inc.: *The MathWorks Simulink.* `http://www.mathworks.de/products/simulink/`, Abruf: 26.07.2015

[78] McCall, J. A.: Quality Factors. In: *Encyclopedia of Software Engineering* Bd. 2 O-Z. John Wiley & Sons Ltd., 1994, S. 958–969

[79] Medvidovic, N. ; Taylor, R. N.: A Classification and Comparison Framework for Software Architecture Description Languages. In: *IEEE Transactions on Software Engineering* 26 (2000), Nr. 1, S. 70–93

[80] Miller, B. W. ; Hwang, C. H. ; Torkkola, K. ; Massey, N. : An Architecture for an Intelligent Driver Assistance System. In: *IEEE Intelligent Vehicles Symposium*, 2003, S. 639–644

[81] Miller, G. A.: The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. In: *The Psychological Review* 63 (1956), S. 81–97

[82] MISRA – Motor Industry Software Reliability Association: *MISRA C:2012 (MISRA C3) – Guidelines for the Use of the C Language in Critical Systems.* 2012

[83] Oestereich, B. : *Objektorientierte Softwareentwicklung – Analyse und Design mit UML 2.1.* 8. Auflage. Oldenbourg Verlag, München, 2006

[84] Oestereich, B. ; Hruschka, P. : *Erfolgreich mit Objektorientierung.* 2. Auflage. Oldenbourg Verlag, München, 2001

[85] OMG – Object Management Group: *MDA – Model Driven Architecture.* `http://www.omg.org/mda/`, Abruf: 26.07.2015

[86] OMG – Object Management Group ; OMG (Hrsg.): *MOF – Meta Object Facility.* `http://www.omg.org/mof/`, Abruf: 26.07.2015

[87] OMG – Object Management Group: *OCL – Object Constraint Language.* `http://www.omg.org/spec/OCL/`, Abruf: 26.07.2015

[88] OMG – Object Management Group: *SysML – Systems Modeling Language.* `http://www.omgsysml.org/`, Abruf: 26.07.2015

[89] OMG – Object Management Group: *UML – Unified Modeling Language.* `http://www.uml.org`, Abruf: 26.07.2015

[90] OMG – Object Management Group ; OMG (Hrsg.): *XMI – XML Metadata Interchange.* `http://www.omg.org/spec/XMI/`, Abruf: 26.07.2015

[91] Peltier, M. ; Bézivin, J. ; Guillaume, G. : MTRANS: A general framework, based on XSLT, for model transformations. In: *WTUML'01, Proceedings of the Workshop on Transformations in UML, Genua, Italien*, 2001

[92] Petrasch, R. ; Meimberg, O. : *Model-Driven Architecture: Eine praxisorientierte Einführung in die MDA.* 1. Auflage. dpunkt.verlag GmbH, Heidelberg, 2006

[93] Posch, T. ; Birken, K. ; Gerdom, M. : *Basiswissen Softwarearchitektur – Verstehen, entwerfen, wiederverwenden.* 2. Auflage. dpunkt.verlag GmbH, Heidelberg, 2007

[94] Reichmann, C. : *Grafisch notierte Modell-zu-Modell-Transformationen für den Entwurf eingebetteter elektronischer Systeme*, Universität Karlsruhe, Diss., 2005

[95] Reif, K. : *Automobilelektronik. Eine Einführung für Ingenieure.* 2. Auflage. Vieweg+Teubner Verlag, Wiesbaden, 2006

[96] Reussner, R. (Hrsg.) ; Hasselbring, W. (Hrsg.): *Handbuch der Software-Architektur.* 2. Auflage. dpunkt.verlag GmbH, Heidelberg, 2009

[97] Rupp, C. : *Requirements-Engineering und -Management.* 5. Auflage. Hanser Verlag, München, 2009

[98] Rupp, C. ; Queins, S. ; Zengler, B. : *UML 2 glasklar.* 3. Auflage. Hanser Verlag, München, 2007

[99] Schaller, T. : *Stauassistenz – Längs- und Querführung im Bereich niedriger Geschwindigkeit*, Technische Universität München, Diss., 2009

[100] Schätz, B. ; Spies, K. : Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik / Technische Universität München. 1995 (TUM-I9529). – Forschungsbericht

[101] Schäuffele, J. ; Zurawka, T. : *Automotive Software Engineering.* 3. Auflage. Vieweg+Teubner Verlag, Wiesbaden, 2006

[102] Schienmann, B. : *Kontinuierliches Anforderungsmanagement – Prozesse, Techniken, Werkzeuge.* Addison-Wesley, Boston, USA, 2002

[103] Schmidt, A. ; Görzig, S. ; Levi, P. : ANTSRT – Eine Software-Architektur für Fahrerassistenzsysteme. In: *Autonome Mobile Systeme 2003, 18. Fachgespräch Karlsruhe*, 2003, S. 264–271

[104] Schnelle, K.-P. : Vernetzung im Automobil. Systeme wachsen zusammen. In: *Sonderausgabe ATZ und MTZ und Automotive Engineering Partners: Automotive Electronics I* (2005), S. 32–35

[105] Schwarz, J. : Response 3 – Code of Practice für die Entwicklung, Validierung und Markteinfuehrung von weiterführenden Fahrerassistenzsystemen. In: *Integrierte Sicherheit und Fahrerassistenzsysteme, Wolfsburg*, VDI-Verlag, Düsseldorf, 2006, S. 465–472

[106] Seiffert, U. ; Varchmin, J.-U. : Der Nutzen von mehr Fahrzeugelektronik. In: *Sonderausgabe ATZ und MTZ und Automotive Engineering Partners: Automotive Electronics I* (2005), S. 46–49

[107] Spannheimer, H. ; Heimrath, M. ; Wisselmann, D. : Hochautomatisiertes Fahren – Technologie und Herausforderungen. In: *28. VDI-VW-Gemeinschaftstagung Fahrerassistenz und Integrierte Sicherheit, Wolfsburg*, VDI-Verlag, Düsseldorf, 2012, S. 275–286

[108] Stahl, T. ; Völter, M. ; Efftinge, S. ; Haase, A. : *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management.* 2. Auflage. dpunkt.verlag GmbH, Heidelberg, 2007

[109] Starke, G. : *Effektive Software-Architekturen. Ein praktischer Leitfaden.* 1. Auflage. Hanser Verlag, München, 2005

[110] Steinle, J. ; Toelge, T. ; Thissen, S. ; Pfeiffer, A. ; Brandstäter, M. : Kultivierte Dynamik – Geschwindigkeitsregelung im neuen BMW 3er. In: *ATZ/MTZ extra* (2005), S. 122–131

[111] Stroustrup, B. : *The C++ programming language.* 4. Auflage. Addison-Wesley, Upper Saddle River, USA, 2013

[112] SUN Microsystems: *Java Programming Language.* http://www.java.com, Abruf: 26.07.2015

[113] SUN Microsystems: *JAXB – Java Architecture for XML Binding.* https://jaxb.java.net/, Abruf: 26.07.2015

[114] The Eclipse Foundation: *Eclipse IDE for Java Developers, Eclipse IDE for C/C++ Developers. Helios SR 1.* http://www.eclipse.org, Abruf: 26.07.2015

[115] The GNU-Project: *GCC, the GNU Compiler Collection, Version 4.8.2.* https://gcc.gnu.org/, Abruf: 26.07.2015

[116] Thomson, R. ; Huff, K. E. ; Gish, J. W.: Maximizing Reuse during Reengineering. In: *Third International Conference on Software Reuse, Rio de Janeiro, Brasilien*, IEEE Computer Society, Washington D.C., USA, 1994, S. 16–23

[117] Tichelaar, S. ; Demeyer, S. D. S. ; Nierstrasz, O. : A Meta-model for Language-Independent Refactoring. In: *International Symposium on Principles of Software Evolution, Kanazawa, Japan*, IEEE Computer Society, Washington D.C., USA, 2000, S. 154–164

[118] Travkin, D. : *Bewertung automatisch erkannter Instanzen von Software-Mustern*, Universität Paderborn, Diplomarbeit, 2006

[119] University of Southern California – Center for Systems and Software Engineering: *UCC – Unified Code Count, Version v.2010.07*. http://sunset.usc.edu/ucc_wp/, Abruf: 26.07.2015

[120] VDI – Verein Deutscher Ingenieure: *VDI 2206 – Entwicklungsmethodik für mechatronische Systeme*. 2004

[121] Vector Informatik GmbH: *PREEvision 6.5*. http://vector.com/vi_preevision_de.html, Abruf: 26.07.2015

[122] W3C – World Wide Web Consortium: *XML – Extensible Markup Language*. http://www.w3.org/XML/, Abruf: 26.07.2015

[123] W3C - World Wide Web Consortium: *XSL Transformations (XSLT)*. http://www.w3.org/TR/1999/REC-xslt-19991116, Abruf: 26.07.2015

[124] Walther, M. ; Torre Flores, P. ; Bertram, T. : CARTRONIC als Ordnungskonzept für den Systemverbund – Analyse mechatronischer Systeme im Kraftfahrzeug. In: *Elektronik im Kraftfahrzeugwesen*. 3. Auflage. expert Verlag, Renningen, 2002, Kapitel 6.2, S. 376–397

[125] Weilkiens, T. : *Systems Engineering mit SysML/UML. Modellierung, Analyse, Design*. 2. Auflage. dpunkt.verlag GmbH, Heidelberg, 2006

[126] Wietzke, J. ; Tien, T. M.: *Automotive Embedded Systeme*. 1. Auflage. Springer-Verlag, Heidelberg, 2005

[127] Wild, D. ; Fleischmann, A. ; Hartmann, J. ; Pfaller, C. ; Rappl, M. ; Rittmann, S. : An Architecture-Centric Approach Towards the Construction of Dependable Automotive Software. In: *SAE World Congress*, Society of Automotive Engineers (SAE), Detroit, USA, 2006

[128] Winner, H. (Hrsg.) ; Hakuli, S. (Hrsg.) ; Wolf, G. (Hrsg.): *Handbuch Fahrerassistenzsysteme*. 1. Auflage. Vieweg+Teubner Verlag, Wiesbaden, 2009

[129] Ziegenbein, D. ; Braun, P. ; Freund, U. ; Bauer, A. ; Romberg, J. ; Schätz, B. : AutoMoDe – Model-Based Development of Automotive Software. In: *Proceedings of the 2005 Conference on Design, Automation and Test in Europe (DATE '05), München*, IEEE Computer Society, Washington D.C., USA, 2005, S. 171–176

[130] Zimmermann, W. ; Schmidgall, R. : *Bussysteme in der Fahrzeugtechnik. Protokolle und Standards*. 2. Auflage. Vieweg+Teubner Verlag, Wiesbaden, 2006

[131] Züst, R. : *Einstieg ins Systems Engineering*. 3. Auflage. Orell Füssli Verlag, Zürich, Schweiz, 2004