Lehrstuhl für Realzeit–Computersysteme

Systematisierung des funktionalen Tests eingebetteter Software

Robert Olejniczak

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.–Prof. Dr. sc. techn. (ETH) Andreas Herkersdorf
Prüfer der Dissertation:	1. Univ.–Prof. Dr.–Ing. Georg Färber
	2. Univ.–Prof. Dr. rer. nat., Dr. rer. nat. habil. Peter O. A. Struss

Die Dissertation wurde am 25.09.2007 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 14.01.2008 angenommen.

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Realzeit–Computersysteme (RCS) der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München in Kooperation mit den Ingolstadt Instituten der Technischen Universität München (INI.TUM) und der AUDI AG.

An erster Stelle bedanke ich mich bei meinem Doktorvater Prof. Färber für sein entgegengebrachtes Vertrauen, für die stetige Förderung der Arbeit sowie die zahlreichen sehr angenehmen Diskussionen und wertvollen Anregungen. Gleichzeitig danke ich Prof. Struss für sein Interesse an dieser Arbeit, für die vielen Anmerkungen sowie für die Übernahme des Zweitgutachters. Ebenso danke ich Dr. Rudolph für das Initiieren der Aufgabenstellung sowie für die Betreuung der Arbeit von Industrieseite.

Weiterhin richtet sich mein Dank an alle Kolleginnen und Kollegen am Lehrstuhl, bei INI.TUM und bei der AUDI AG für die kollegiale Zusammenarbeit und sehr angenehme Arbeitsatmosphäre sowie für die stets kreativen und motivierenden Impulse. Die regelmäßigen Diskussionsrunden haben wesentlich zum Gelingen dieser Arbeit beigetragen und die zahlreichen außerberuflichen Aktivitäten ließen mich nie vergessen, dass neben der Arbeit auch noch das Leben stattfindet.

Außerdem danke ich den zahlreichen Korrekturlesern für ihre inhaltlichen und formalen Kommentare.

Zu guter Letzt gilt ein besonderer Dank meinen Eltern sowie meiner Freundin Katja für ihre unermüdliche Unterstützung, ihre Geduld und ihr Verständnis sowie die immer passenden und motivierenden Worte während der Entstehung dieser Arbeit.

München, im April 2008

Kurzfassung

Die Innovationen im Automobilbereich sind geprägt von Elektrik/Elektronik und in besonderem Maße von der darin eingebetteten Software. Heutzutage kommunizieren zahlreiche in die Fahrzeugumgebung eingebettete Steuergeräte über verschiedene Bussysteme miteinander, um die enthaltenen Funktionen zu realisieren. Diese Systemkomplexität spiegelt sich auch im Entwicklungsprozess wider. Neben dem Fahrzeughersteller sind zahlreiche Zulieferer an der Entwicklung neuer Fahrzeugsysteme beteiligt.

Trotz dieses Anstiegs der Gesamtkomplexität muss neben der Sicherheit auch die Qualität der Fahrzeugelektronik gewährleistet werden. Begriffe wie Hardware−in−the−Loop und automatische Testausführung sind aus dem Testumfeld der Fahrzeugelektronik nicht mehr wegzudenken und tragen wesentlich zur Qualitätssicherung heutiger Elektroniksysteme bei. Um auch in Zukunft die Qualität der Fahrzeugelektronik sicherzustellen, forciert die Automobilindustrie den Einsatz systematischer Testverfahren. Qualität wird immer mehr zum wettbewerbsentscheidenden Faktor. Dem Testen, als wichtigste Maßnahme der Qualitätssicherung, kommt immer mehr Bedeutung im Entwicklungsprozess zu.

Große Herausforderungen stellen in diesem Zusammenhang das systematische Ermitteln geeigneter Testfälle sowie die Bewertung des durch die ermittelten Testfälle erreichten Testumfangs dar. Grundlage für die Testfallermittlung ist derzeit die funktionale Spezifikation der Systeme. Testersteller leiten daraus, hauptsächlich aufgrund ihrer Erfahrung, Testfälle ab und halten die ermittelten Testfälle in textuellen Testspezifikationen fest. Aussagen über den darin enthaltenen Testumfang und über die entsprechende Qualität der Tests sind aufgrund des unsystematischen Vorgehens bei der Ermittlung von Tests sowie des geringen Formalisierungsgrades der Testspezifikationen sehr schwierig.

Vor diesem Hintergrund wird in der vorliegenden Arbeit der Einsatz zweier existierender systematischer Testverfahren (Classification−Tree Method (CTM) und Classification−Tree Method for Embedded Systems (CTM/ES)) im Umfeld der Funktionserprobung mittels Hardware−in−the−Loop Testsystemen bei der AUDI AG untersucht. Die CTM unterstützt die Testfallermittlung durch eine systematische Vorgehensweise und bietet Ansatzpunkte zur Bewertung des erreichten Testumfangs. Die CTM/ES berücksichtigt zusätzlich Charakteristika eingebetteter Systeme und erlaubt z.B. die Spezifikation von Testsequenzen, welche sich wiederum aus einzelnen Testschritten zusammensetzen. Beide Methoden verfügen über eine grafische Werkzeugunterstützung in Form des Classification−Tree Editor eXtended Logics (CTE XL).

Aufbauend auf einem neuartigen Konzept zur Kopplung der beiden ausgewählten, systematischen Testverfahren werden mögliche Integrationsszenarien der Testverfahren sowie der Werkzeugunterstützung im vorhandenen Testprozess aufgezeigt. Die mit dem Integrationskonzept verbundene Vorgehensweise wird anschließend durch Testersteller verschiedener Bereiche anhand realer Beispiele in einem Praxistest auf Praxistauglichkeit erprobt.
Nach Aussage der befragten Testersteller liefern die erstellten Klassifikationsbäume einen guten Überblick über den Eingabedatenraum des zu testenden Systems. Einige bisher bei existierenden Tests nicht berücksichtigte Inhalte, wie z.B. Randbedingungen, konnten durch den Einsatz der Methoden identifiziert werden.
Aufgrund der Unübersichtlichkeit der Testsequenzen, in Kombination mit der nicht möglichen Aussage über unberücksichtigte Übergänge zwischen den einzelnen Testschritten der Testsequenzen, lehnen die Testersteller jedoch das im vorgestellten Integrationskonzept enthaltene Szenario zum Einsatz der CTM/ES ab.
Der Einsatz der CTM schneidet im durchgeführten Praxistest dagegen überwiegend positiv ab. Vor allem der gute Überblick über ähnliche Testinhalte und die damit verbundene Möglichkeit zur Strukturierung der Testfälle, zur Diskussion der Testinhalte mit anderen Personen sowie das Vorhandensein von Überdeckungskriterien wird von den Testerstellern positiv bewertet.

Die angestrebte Integration der beiden systematischen Testverfahren in den existierenden Testprozess erfolgt aufgrund der Ergebnisse des durchgeführten Praxistests in abgewandelter Form. Auf den Einsatz der CTM/ES wird gänzlich verzichtet. Bei der finalen Umsetzung des Integrationskonzeptes wird allein die CTM in den existierenden Testprozess integriert. Eine speziell entwickelte Schnittstelle (ctm2exam), zwischen dem CTE XL und der bei der AUDI AG eingesetzten Testautomatisierung EXtended Automation Method (EXAM), stellt dabei die im CTE XL spezifizierte Teststruktur dem geschilderten Testprozess zur Verfügung. Durch diesen Einsatz der CTM sowie des CTE XL wird die Systematik bei der Ermittlung von Tests gefördert. Damit verbunden verbessert sich die Strukturierung von ähnlichen Testinhalten, wodurch Redundanzen in den Testspezifikationen vermieden werden. Außerdem ist eine Aussage über die erreichte Testabdeckung ebenso wie die Angabe von messbaren Testendekriterien möglich. Allein die Testqualität ist nach wie vor vom Qualifikationsniveau des jeweiligen Testerstellers abhängig. Sie lässt sich jedoch aufgrund der grafischen Repräsentation besser mit anderen Personen diskutieren und damit auch einfacher bewerten.

Mit dieser Arbeit wird somit ein Beitrag zur Systematisierung des funktionalen Tests von in Fahrzeugsysteme eingebetteter Software geleistet.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

μC	Mikrocontroller
ABS	Anti Blockier System
ACC	Adaptive Cruise Control
ADAC	Allgemeiner Deutscher Automobil Club e.V.
CAN	Control Area Network
CANoe	CAN Open Environment
CART	Classification And Regression Trees
CASE	Computer−Aided Software/Systems Engineering
CTC	Classification−Tree Coverage
CTE	Classification−Tree Editor
CTE XL	Classification−Tree Editor eXtended Logics
CTM	Classification−Tree Method
CTM/ES	Classification−Tree Method for Embedded Systems
DB	Datenbank
DIN	Deutsches Institut für Normung e.V.
DOORS	Dynamic Object Oriented Requirements System
DP	Dunkelphase
E/E	Elektrik/Elektronik
ECU	Electronic Control Unit
EMV	Elektromagnetische Verträglichkeit
EN	Europäische Norm
ESP	Elektronisches Stabilitätsprogramm
EXAM	EXtended Automation Method
GW	Gateway
HIL	Hardware−in−the−Loop
HP	Hellphase
ICTM	Integrated Classification−Tree Method
ID	Identifier

IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
INCA	Integrated Calibration and Application Tools
ISO	International Organization for Standardization
Kfz	Kraftfahrzeug
LDW	Lane Departure Warning
MB^3T	Model−based black−box testing
MDA	Model Driven Architecture
MIL	Model−in−the−Loop
MOST	Media Oriented System Transport
NBWB	Notbremswarnblinken
NLZ	Nachlaufzeit
OEM	Original Equipment Manufacturer
OLE	Object Linking and Embedding
OMG	Object Management Group
PC	Personal Computer
PIL	Processor−in−the−Loop
RB	Richtungsblinken
SE	Simultaneous Engineering
SG	Steuergerät
SIL	Software−in−the−Loop
SMLS	Schaltermodul−Lenksäule
SOP	Start of Production
SUT	System Under Test
UML	Unified Modeling Language
VIL	Vehicle−in−the−Loop
WBT	Warnblinktaster
XML	Extensible Markup Language

1 Einleitung

1.1 Motivation

Eine Vielzahl von elektronischen Geräten unterstützt uns heute bei der Erledigung zahlreicher alltäglicher Aufgaben. Mobiltelefone, Waschmaschinen sowie Toaster, um nur einige Beispiele zu nennen, kommen heutzutage nicht mehr ohne Elektronik aus. Es werden immer häufiger programmierbare Mikroprozessoren zum Steuern dieser Geräte eingesetzt, wodurch sich die Komplexität der Elektrik in die Software dieser Prozessoren verlagert. Solche Steuerungs– und Regelsysteme werden als eingebettete Systeme und die enthaltene Software entsprechend als eingebettete Software bezeichnet. Der wesentliche Unterschied zu klassischen Rechnersystemen besteht darin, dass eingebettete Systeme Teil eines größeren Gesamtsystems sind und nur darin ihre volle Funktionsfähigkeit entfalten können.

Ein weiteres großes Anwendungsgebiet eingebetteter Systeme stellt der Automobilbereich dar. Die Begriffe Komponente, Steuergerät bzw. Electronic Control Unit (ECU) werden hierbei synonym für ein eingebettetes System verwendet. Es ist in die Fahrzeugumgebung eingebunden und ohne diese nicht funktionsfähig. Bekannte Vertreter von eingebetteten Systemen stellen hier z.B. das Anti Blockier System (ABS), das Elektronische Stabilitätsprogramm (ESP) oder die elektronische Motorsteuerung dar. Ohne diese Systeme könnte die Zahl der im Straßenverkehr umgekommenen Menschen, trotz des stetig steigenden Verkehrsaufkommens, kaum gemindert werden (*Fastenmeier*, 2005). Auch Abgasnormen könnten ohne intelligente Motorsteuerungen nicht eingehalten werden. Weiterhin sind so genannte Fahrerassistenzsysteme derzeit auf dem Vormarsch. Sie unterstützen den Fahrer bei seiner Fahraufgabe, indem sie z.B. Spurwechsel überwachen, passende Parklücken signalisieren oder aber einen konstanten Abstand zu einem vorausfahrenden Fahrzeug halten.

Für die Umsetzung dieser neuen Systeme werden im Lauf der Zeit immer mehr neue Sensoren (z.B. Radar, Kameras, ...) sowie die entsprechenden Systemschnittstellen im Automobil benötigt. Die eingesetzten Sensoren und Aktoren werden meist von mehreren Systemen verwendet und Funktionen sind in der Regel über mehrere Komponenten verteilt. Dies spiegelt sich in einem exponentiellen Anstieg der Anzahl der Funktionen wider (*Schmid*, 2003). Dementsprechend rasant steigt auch die Gesamtkomplexität der Automobilelektronik (s. Abb. 1.1). Diese exponentielle Zunahme der Gesamtkomplexität ist jedoch nicht allein auf neue Funktionalitäten zurückzuführen, sondern in einem gewissen Maße auch auf die erforderlichen Diagnoseanteile der Funktionen (*Wohnhaas*, 1997). Die Entwicklungs– und

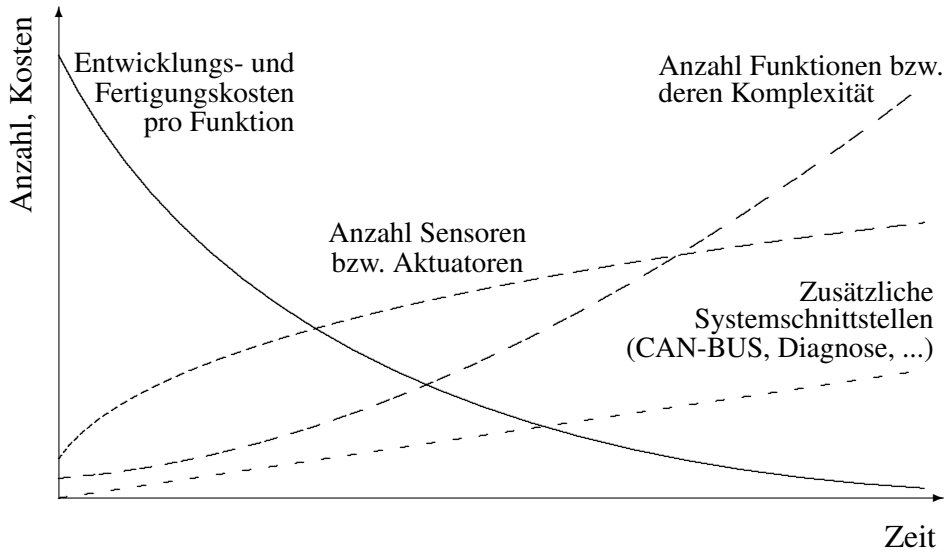


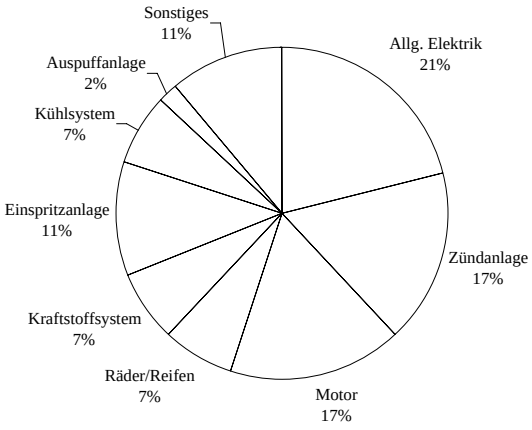
Abbildung 1.1: Qualitativer Vergleich von Funktionen, Kosten und Peripherie (*Schmid*, 2003)

Fertigungskosten pro Funktion fallen dagegen exponentiell ab. Abb. 1.1 visualisiert den soeben beschriebenen Sachverhalt in qualitativer Weise.

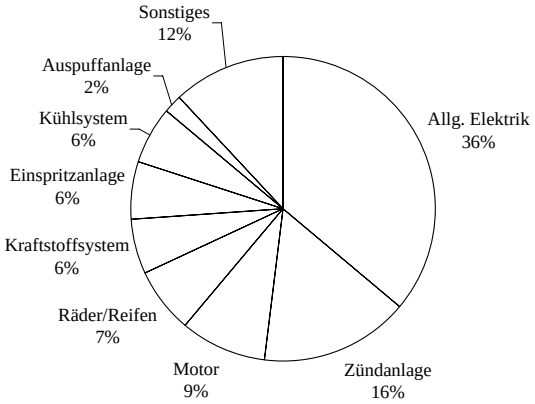
Durch diese starke Vernetzung der Komponenten können sich die Entwickler von Fahrzeugsystemen nicht mehr auf ein Einzelsystem beschränken, sondern müssen funktionale Abhängigkeiten im Gesamtsystem geeignet berücksichtigen. *Conrad* (2004) sieht dabei mehr und mehr den Entwicklungs– und Testprozess der Fahrzeugsysteme und der darin eingebetteten Software als limitierenden Faktor bei deren Entwicklung.

Die eingebettete Software wird damit immer mehr zum Wettbewerbsfaktor. Neben innovativen Funktionen bzw. Systemen werden in der Öffentlichkeit jedoch immer häufiger die Qualitätsprobleme der Systeme diskutiert. Abb. 1.2 zeigt die Gegenüberstellung der ADAC–Pannenstatistiken aus den Jahren 1985 und 2004. Dabei wird die prozentuale Verteilung der im jeweiligen Jahr registrierten Pannen auf die verschiedenen Baugruppen eines Fahrzeugs betrachtet. Der Anteil der allgemeinen Elektrikpannen ist dabei von 21% auf 36% angestiegen, und ein weiterer Anstieg ist aufgrund der immer höher werdenden Komplexität der Elektrik/Elekronik–Systeme sehr wahrscheinlich.

Die Zuverlässigkeit der Systeme ist daher ein entscheidender Wettbewerbsvorteil und muss dementsprechend gefördert werden. Die damit verbundenen hohen Qualitätsanforderungen an die Systeme können neben einem strukturierten Entwicklungsprozess nur durch ausgiebige und systematische Tests gewährleistet werden.



(a) Pannenstatistik von 1985 – Verteilung der Baugruppen



(b) Pannenstatistik von 2004 – Verteilung der Baugruppen

Abbildung 1.2: Gegenüberstellung der ADAC–Pannenstatistiken von 1985 und 2004 (*David & Rempfer*, 2005)

1.2 Zielsetzung

Die Komplexität der soeben vorgestellten Systeme spiegelt sich im dazugehörigen Testprozess wider. Immer aufwändigere Tests sind notwendig, um das geforderte korrekte und zuverlässige Funktionieren der Systeme sicherzustellen.

Die Ermittlung geeigneter Tests stellt zwar eine anspruchsvolle Aufgabe dar, in der Praxis wird sie jedoch derzeit zum großen Teil ohne methodische Unterstützung durchgeführt (*Armbrust et al.*, 2004b). Tests werden hauptsächlich Ad–hoc bzw. aufgrund der Erfahrung der Testersteller ermittelt und in textuellen Testspezifikationen festgehalten. Damit ist die Qualität jedes einzelnen Tests von der Erfahrung bzw. dem Qualifikationsniveau des jeweiligen Testerstellers abhängig. Die resultierenden Tests erlauben größtenteils keine ausreichende Prüfung der immer komplexeren Systeme. Aussagen über die Testabdeckung, die eine Grundlage für die Bewertung des Testumfangs sowie für die Definition von Testendekriterien bildet, sind aufgrund der nicht vorhandenen Systematik und des geringen Formalisierungsgrades der textuellen Testspezifikationen kaum möglich. Um die Qualität von Automobilelektronik auch weiterhin sicherzustellen, ist der Einsatz systematischer Testverfahren erforderlich. Geeignete Verfahren inkl. Werkzeugunterstützung existieren. In der Praxis konnten sie sich bisher jedoch nicht durchsetzen.

Ziel dieser Arbeit ist daher die Integration von existierenden, systematischen Testverfahren in den bei der AUDI AG existierenden Testprozess von in Fahrzeugsystemen eingebetteter Software mit der Absicht, die soeben erwähnten Defizite zu beheben. Dafür müssen zunächst für den vorhandenen Testprozess relevante Testverfahren miteinander verglichen und die für eine Integration vielversprechendsten ausgewählt werden. Darauf aufbauend muss ein Integrationskonzept des/der Testverfahren erstellt und von Testerstellern verschiedener Berei-

che in einem Praxistest anhand realer Beispiele auf Praxistauglichkeit erprobt werden. Nach eventueller Anpassung ist die Umsetzung des Integrationskonzeptes durchzuführen und eine geeignete Werkzeuganbindung bzw. –unterstützung für den existierenden Testprozess sowie für die existierende Testautomatisierung EXtended Automation Method (EXAM) bereitzustellen.

1.3 Gliederung

Zunächst wird der Leser im zweiten Kapitel mit dem Software–Test vertraut gemacht. Dabei wird auf die Qualität von Software sowie auf das Testen als ein Teil der analytischen Qualitätssicherung eingegangen. Dem folgt die Klärung verwendeter Begriffe rund um das Thema Testen. Abschließend werden Testaktivitäten des Testprozesses vorgestellt und außerdem die Notwendigkeit von Testautomatisierung erläutert.

Gegenstand des dritten Kapitels ist die Automobilelektronik. Nach einer kurzen Betrachtung des Aufbaus wird sowohl der Entwicklungs– als auch der Testprozess von Automobilelektronik inkl. der eingesetzten Testsysteme vorgestellt. Besonderer Fokus liegt dabei auf dem Testprozess mittels Hardware–in–the–Loop Testsystemen. Abschließend wird die vorliegende Problemstellung sowie ein daraus abgeleitetes Fazit präsentiert.

Im vierten Kapitel werden existierende dynamische Testverfahren kurz vorgestellt und im Hinblick auf den Einsatz im zuvor beschriebenen Testprozess bewertet.

Kapitel fünf stellt die zwei ausgewählten, systematischen Testverfahren, die Classification–Tree Method (CTM) und die Classification–Tree Method for Embedded Systems (CTM/ES), detailliert vor. Weiterhin werden überblicksartig weitere existierende Erweiterungen der beiden Testverfahren sowie die vorhandene, grafische Werkzeugunterstützung präsentiert.

Aufbauend auf den beiden ausgewählten und vorgestellten Testverfahren enthält Kapitel sechs ein neuartiges Konzept zur Kopplung der beiden systematischen Testverfahren. Zudem wird in einem damit verbundenen Integrationskonzept die Einbindung der Testverfahren inkl. Werkzeugunterstützung in den bei der AUDI AG existierenden Testprozess sowie in die vorhandene Testautomatisierung EXAM präsentiert. Abschließend erfolgt an einem praxisnahen Beispiel eine ausführliche Darlegung der mit dem Integrationskonzept verbundenen Vorgehensweise.

Kapitel sieben beinhaltet die Validierung des zuvor präsentierten Integrationskonzeptes durch einen Praxistest. Anhand realer Beispiele aktueller Testprojekte werden in dem Praxistest Testersteller, als potenzielle Anwender der vorgeschlagenen Vorgehensweise, mit dem neuartigen Integrationskonzept konfrontiert. Der Einsatz der CTM wird dabei überwiegend positiv bewertet, während der Einsatz der CTM/ES von den am Praxistest beteiligten Testerstellern größtenteils negativ beurteilt wird. Basierend auf diesen Ergebnissen des Praxistests

zeigt der letzte Abschnitt dieses Kapitels schließlich die Konsequenzen für das präsentierte Integrationskonzept auf.

Die finale Umsetzung des modifizierten Integrationskonzeptes (Integration der CTM in den vorgestellten Testprozess) ist Bestandteil des achten Kapitels. Die entwickelte Schnittstelle (ctm2exam) zwischen dem CTE XL und der eingesetzten Testautomatisierung EXAM wird dabei ausführlich vorgestellt. Anhand eines weiteren Praxisbeispiels wird abschließend der erfolgreiche Einsatz der angepassten Vorgehensweise in einem aktuellen Entwicklungsprojekt der AUDI AG in Kurzform dargestellt.

Die Arbeit endet mit einer Zusammenfassung in Kapitel neun.

Im Anhang befindet sich außerdem eine ausführliche Klassifikation existierender Testverfahren, die Vorstellung der bei der AUDI AG eingesetzten Testautomatisierung EXAM (EXtended Automation Method) sowie die Beschreibung der Schnittstelle (ctmes2exam), die im Lauf der Arbeit zwar prototypisch entwickelt, aufgrund des negativen Feedbacks im durchgeführten Praxistest zum Einsatz der CTM/ES jedoch nicht weiter verfolgt wurde.

2 Theoretischer Hintergrund

Die Theorie, die sich hinter dem Testen verbirgt, ist sehr umfangreich. Dieses Kapitel vermittelt daher nur einen für diese Arbeit relevanten Einblick in dieses Thema. Zunächst werden zwei verschiedene Herangehensweisen beim Test aufgezeigt, gefolgt von der Definition von Software−Qualität und der Einordnung des Tests als Bestandteil der analytischen Qualitätssicherung. Eine Begriffsklärung stellt die wichtigsten Begriffe, die im Zusammenhang mit dem Testen stehen, vor. Abschließend werden die wichtigsten Testaktivitäten des Testprozesses aufgelistet und außerdem die Notwendigkeit von Testautomatisierung erläutert.

2.1 Einordnung des Software−Testens

Grundsätzlich lassen sich bzgl. der Definition des Testens im Allgemeinen und des Software−Testens im Besonderen zwei Herangehensweisen unterscheiden (*Vigenschow*, 2005):

- demonstrative Herangehensweise
- destruktive Herangehensweise

Ziel der demonstrativen Herangehensweise ist nach *Riedemann* (1997) die Auswahl von Testdaten um die „Fehlerfreiheit des Programms nachzuweisen. Bei der destruktiven Vorgehensweise ist dagegen das Auffinden von Fehlern das primäre Ziel.“ Diese destruktive Herangehensweise geht bereits auf *Myers* zurück, der 1979 das Testen definiert hat als: „the process of executing a program with the intent of finding errors.“

Im Zusammenhang mit den soeben vorgestellten Vorgehensweisen spricht man auch von Validierung und Verifikation. Die Validierung entspricht der demonstrativen Vorgehensweise, bei der das Softwareprodukt erprobt wird, um seine Funktionsfähigkeit gegenüber dem Auftraggeber im so genannten Abnahmetest zu demonstrieren. Der Nutzen für den Kunden bzw. die Entwicklung des richtigen Produkts steht hierbei im Vordergrund. Verifikation entspricht dagegen der destruktiven Vorgehensweise. Hierbei geht es darum, möglichst viele Fehler frühzeitig im Testobjekt aufzudecken und so den Reifegrad des Produkts möglichst kostengünstig zu erhöhen. Die richtige Entwicklung des Produkts steht dabei im Vordergrund. Der direkte Nutzen für den Kunden ist dabei zunächst zweitrangig.

Um ein Produkt zu entwickeln, das sowohl den Erwartungen des Kunden als auch seiner funktionalen Spezifikation entspricht, müssen beide Vorgehensweisen beim Test berücksichtigt werden. Denn es ist z.B. denkbar, dass ein Softwareprodukt zwar seine funktionale Spezifikation erfüllt, für den Kunden jedoch nur von geringem Nutzen ist, weil z.B. die funktionale Spezifikation selbst fehlerbehaftet ist.

Für die vorliegende Arbeit spielen daher beide Vorgehensweisen eine Rolle, wobei der Fokus deutlich auf der Seite der Verifikation liegt. Deshalb wird für den weiteren Verlauf der Arbeit eine Definition des Testens gewählt, die beide Vorgehensweisen beinhaltet: „Unter dem Testen von Software wird jede (im Allgemeinen stichprobenartige) Ausführung eines Testobjekts verstanden, die der Überprüfung des Testobjekts dient. Die Randbedingungen für die Ausführung des Tests müssen festgelegt sein. Ein Vergleich zwischen Soll– und Istverhalten des Testobjekts dient zur Bestimmung, ob das Testobjekt die geforderten Eigenschaften erfüllt“ (*Spillner & Linz*, 2005).

Durch das Auffinden und Beheben von Fehlern wird die Qualität des Testobjekts verbessert, sofern bei der Fehlerbehebung keine neuen Fehler hinzugefügt wurden. Falls trotz intensiver Tests keine Fehler in der Software entdeckt wurden, handelt es sich entweder um ein qualitativ hochwertiges Software–Produkt oder aber die durchgeführten Tests sind nicht repräsentativ bzw. fehlersensitiv genug. Sind die durchgeführten Tests repräsentativ und decken keine Fehler auf, so steigt in der Regel das Vertrauen in die Software, dass sie auch in der späteren Einsatzumgebung fehlerfrei funktioniert.

Um zu beurteilen, ob eine Software von hoher Qualität ist, werden im nächsten Abschnitt zunächst der Begriff der Qualität betrachtet und existierende Qualitätsmerkmale von Software aufgeführt. Weiterhin findet darin die Vorstellung des Tests als wichtigste Maßnahme der analytischen Qualitätssicherung statt. Weitere mit dem Testen verbundene Begriffe betrachtet Kap. 2.2.

2.1.1 Qualität von Software

Zentraler Punkt dieser Arbeit ist die Unterstützung des Testprozesses von in Fahrzeugsysteme eingebetteter Software (s. Kap. 3.5), mit dem Ziel, die Qualität der Fahrzeugsysteme zu verbessern. Aus diesem Grund wird der Begriff der Qualität an dieser Stelle näher betrachtet.

Es existieren zahlreiche Definitionen dieses zentralen Begriffs. In *Abran et al.* (2004) findet sich beispielsweise eine kurze Betrachtung der verschiedensten Definitionen. Eine aktuelle Definition von Qualität lautet: „Grad, in dem ein Satz inhärenter Merkmale Anforderungen erfüllt“ (*DIN EN ISO 9000*).

Diese Definition fordert die Erfüllung von festgelegten Anforderungen durch Merkmale, die dem Produkt (z.B. Software) innewohnen. Doch welche Merkmale existieren für Software? Auf diese Frage gibt die Norm *ISO/IEC 9126-1* eine Antwort. Darin sind folgende Qualitätsmerkmale für Software definiert (s. Abb. 2.1):

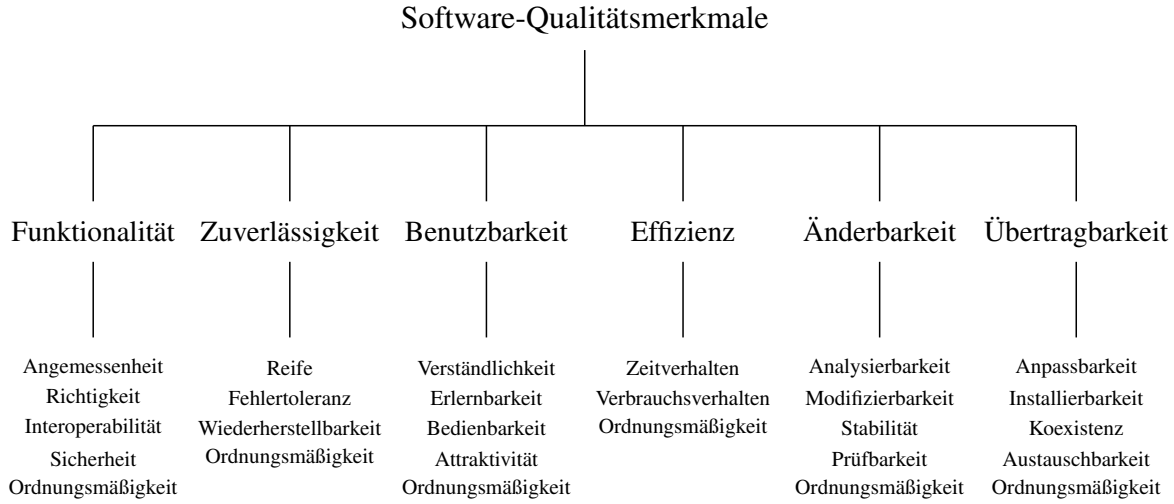


Abbildung 2.1: Klassifikation der Software–Qualitätsmerkmale nach *ISO/IEC 9126-1*

Funktionalität beinhaltet Merkmale zur Beurteilung des Vorhandenseins und der Eignung von Funktionen, spezifizierte Aufgaben zu erfüllen. Die Lieferung der richtigen bzw. spezifizierten Ergebnisse, die Zusammenarbeit mit anderen Systemen sowie Datensicherheit sind ebenso dem Merkmal der Funktionalität zugeordnet. Ordnungsmäßigkeit beschreibt die Erfüllung von geforderten Normen oder Standards zur Funktionalität.

Zuverlässigkeit beschreibt u.a. die Versagenshäufigkeit der Software, die Fehlertoleranz gegenüber internen und externen Fehlerquellen sowie die Fähigkeit zur Wiederherstellung nach einem Fehlerfall.

Benutzbarkeit adressiert direkt die Anwender der Software. Die definierten Merkmale beziehen sich auf die individuelle Benutzung der Software.

Effizienz beinhaltet Merkmale zur Bewertung des Zeit– und Verbrauchsverhaltens.

Änderbarkeit beinhaltet Merkmale zur Beurteilung des Aufwands, der notwendig ist, um z.B. Korrekturen an der Software vorzunehmen.

Übertragbarkeit bezieht sich auf Eignung der Software, in andere Umgebungen übertragen zu werden.

„Die einzigen Merkmale, welche durch Testen systematisch und objektiv unterstützt werden können, sind die Funktionalität und die Zuverlässigkeit“ (*Riedemann*, 1997). Die Effizienz kann bei hinreichend genauer Spezifikation ebenfalls überprüft werden. Die Merkmale Änderbarkeit und Übertragbarkeit lassen sich durch Testen jedoch nicht überprüfen. Ebenso schwierig ist die Überprüfung der Benutzbarkeit.

Qualitativ hochwertige Software zeichnet sich dadurch aus, dass sie die durch die Anforderungen definierten Software−Qualitätsmerkmale zu einem hohen Grad erfüllt. Aufgrund der Existenz von Wechselwirkungen (positive und negative) zwischen den einzelnen Merkmalen, ist die Optimierung jedes einzelnen Merkmals nicht immer erreichbar. So steht beispielsweise die Effizienz im natürlichen Widerspruch zu Verständlichkeit, Prüfbarkeit, Änderbarkeit, Analysierbarkeit und Austauschbarkeit (*Guddat*, 2003). Daher ist es bei der Entwicklung von Software wichtig, die verschiedenen Qualitätsmerkmale entsprechend zu priorisieren und den angestrebten Erfüllungsgrad in der funktionalen Spezifikation festzuhalten.

Diese Übereinstimmung zwischen der funktionalen Spezifikation und der realisierten Softwarefunktionalität wird als Korrektheit der Software bezeichnet. Dabei handelt es sich um einen relativen Begriff, denn Software kann an sich niemals korrekt sein sondern nur im direkten Vergleich mit ihrer funktionalen Spezifikation.

Korrekte Software ist jedoch nicht unbedingt gleichzeitig zuverlässig, falls z.B. die funktionale Spezifikation fehlerhaft ist, und die Anforderungen entsprechend in der Software umgesetzt wurden. Für die vorliegende Arbeit ist daher neben der Überprüfung der Korrektheit von Software auch die Prüfung der Funktionalität, der Zuverlässigkeit sowie des Zeitverhaltens von Bedeutung.

Im weiteren Verlauf der Arbeit wird in diesem Zusammenhang immer von Korrektheit und Zuverlässigkeit gesprochen. Funktionalität und Zeitverhalten werden nicht immer explizit genannt, werden aber dennoch immer mitberücksichtigt.

2.1.2 Testen als Teil der analytischen Qualitätssicherung

Qualität kann nicht im Nachhinein in ein Produkt „hineingeprüft“ werden. Vielmehr bedarf es geeigneter Maßnahmen, welche die geforderte Qualität in das entsprechende Produkt „hineinkonstruieren“. Zur Erreichung einer hohen Produktqualität sind daher zahlreiche Tätigkeiten entlang des gesamten Entwicklungsprozesses notwendig.

Dies ist Aufgabe des Qualitätsmanagement, das nach der Norm *DIN EN ISO 9000* definiert ist als: „Aufeinander abgestimmte Tätigkeiten zum Leiten und Lenken einer Organisation bezüglich Qualität.“ Die Qualitätssicherung ist dabei ein „Teil des Qualitätsmanagements, der auf das Erzeugen von Vertrauen darauf gerichtet ist, dass Qualitätsanforderungen erfüllt werden“. Qualitätssicherung unterteilt sich in folgende drei Teilgebiete:

Organisatorische Maßnahmen beinhalten Aktionen wie Organisation, Planung und Kontrolle der Sicherungsmaßnahmen. Dabei helfen standardisierte Richtlinien wie z.B. die DIN EN ISO900x, die in allen Industrie− und Wirtschaftszweigen zum Einsatz kommen. Diese planenden Aktivitäten bereiten die eigentliche Überprüfung des Produktes vor. Sie stellen somit den allgemeinen Teil einer umfassenden Qualitätsprüfung dar.

Konstruktive Maßnahmen geben in der Regel ein methodisches Vorgehensmodell vor, das den Rahmen für praktische Maßnahmen darstellt. Neben der Festlegung der verwendeten Sprachen werden die verwendeten Hilfsmittel, Methoden und Werkzeuge ausgewählt. Die konstruktive Qualitätssicherung stellt somit die planende Phase für den eigentlichen Bewertungsprozess dar.

Analytische Maßnahmen stellen den für diese Arbeit relevanten Teil dar, denn ihr wesentliches Element ist das Testen. Existierende Testverfahren der analytischen Qualitätssicherung werden in Kap. 4.1 kurz vorgestellt und im Hinblick auf den in Kap. 3.5 vorgestellten Testprozess von in Fahrzeugsysteme eingebetteter Software in Kap. 4.2 bewertet.

2.2 Begriffsklärung

Dieses Kapitel erläutert die wichtigsten Begriffe, die im Zusammenhang mit dem Testen stehen.

2.2.1 Testobjekt

In Kap. 2.1 ist bei der Einordnung bzw. bei der Definition des Testens von der Ausführung eines Testobjekts die Rede. Weitere Synonyme für Testobjekt sind z.B. Prüfgegenstand, Prüfling sowie zu testendes System bzw. System Under Test (SUT).

Im weiteren Verlauf der Arbeit wird hauptsächlich der Begriff des SUT verwendet. Ein SUT kann sowohl ein ganzes System bestehend aus mehreren Subsystemen darstellen, ein einzelnes Subsystem, ein einzelnes Modul, jedoch auch eine einzelne Funktion oder Subfunktion.

2.2.2 Fehler

Der Standard *IEEE Std 610.12–1990* unterscheidet zwischen:

- Irrtum (error)
- Fehler bzw. Defekt (fault, defect)
- Fehlverhalten bzw. Ausfall (failure)

Irrtümer werden vom Entwickler gemacht und führen während der Entwicklung zu falschen Entscheidungen, Maßnahmen und Aktivitäten. Als Folge dieser Irrtümer ergeben sich während der Entwicklung Fehler bzw. Defekte, die statisch im SUT[1] vorhanden sind. Dabei kann

[1] Inkonsistenzen in den Anforderungsdokumenten werden in diesem Zusammenhang ebenfalls als Fehler bzw. Defekte betrachtet.

sich ein Irrtum durchaus in mehreren Fehlern äußern. Erst beim dynamischen Testen können Fehlverhalten bzw. Ausfälle des SUT als Wirkungen von Fehlern beobachtet werden.

Je später ein Fehler im SUT während des Entwicklungsprozesses entdeckt wird, umso mehr Kosten verursacht seine Behebung. Einer genaueren Betrachtung der Fehlerentdeckung und −behebung ist deshalb Kap. 2.3 gewidmet. Darin findet ebenfalls eine Klassifikation von Fehlern statt (s. Kap. 2.3.1).

2.2.3 Testziel

Entsprechend den zwei verschiedenen Herangehensweisen des Testens (vgl. Kap. 2.1) existieren zwei verschiedene allgemeine Zielsetzungen von Tests:

Fehler finden Durch das Entdecken und Beheben von Fehlern steigt die Korrektheit des SUT, sofern keine neuen Fehler bei der Fehlerkorrektur eingefügt wurden. Dabei müssen Fehler so früh wie möglich identifiziert und behoben werden, da die Fehlerbehebungskosten exponentiell mit ihrer Verweildauer ansteigen (s. Kap. 2.3).

Vertrauen schaffen Tests, die keine Fehler aufdecken, erhöhen das Vertrauen in das SUT, auch im späteren Einsatz fehlerfrei zu funktionieren. Die Korrektheit des SUT kann jedoch aufgrund des Stichprobencharakters durch Tests nicht bewiesen werden (s. Anh. A.1).

Bender et al. (2001) nennen neben den soeben erwähnten Kategorien eine dritte allgemeine Kategorie, die Vorbeugung. Zielsetzung dieser Kategorie ist einerseits die Analyse und Detaillierung der Systemspezifikation und −performanz sowie die Bereitstellung von Informationen zur Fehlerreduzierung. Weiteres Ziel ist die Identifikation von Risiken, Problemen und Wegen sowie deren künftige Vermeidung.

Bei der Definition von Testzielen sollte darauf geachtet werden, dass neben diesen allgemeinen Testzielen auch messbare Testziele, wie z.B. Testendekriterien (s. Kap. 2.2.5), definiert werden, deren Erfüllung nachgewiesen werden kann. Denn nur anhand messbarer Ziele lassen sich Aussagen bzgl. des entsprechenden Erfüllungsgrades treffen sowie evtl. Konsequenzen für das aktuelle und für künftige Projekte ableiten. Bereits *DeMarco* (1982) formulierte diese Forderung nach Messbarkeit mit folgendem allgemeinen Satz: „You can't control what you can't measure“ in seinem Buch über die Steuerung von Software Projekten. Neben dem Kriterium der Messbarkeit sollten Testziele zudem auch das Kriterium der Realisierbarkeit erfüllen.

2.2.4 Testorakel

Grundlage zur Beurteilung eines Tests stellt eine Referenz dar, die das korrekte Verhalten des SUT kennt. Für diese Referenz existiert der Begriff des Testorakels. „An oracle is any

(human or mechanical) agent which decides whether a program behaved correctly in a given test, and accordingly produces a verdict of „pass“ or „fail“. There exist many different kinds of oracles, and oracle automation can be very difficult and expensive“ *Abran et al.* (2004).

Dieses Zitat verdeutlicht die Problematik bei der Erstellung des Orakels. *Riedemann* (1997) nennt folgende drei Möglichkeiten zur Bestimmung des Testorakels:

- Sollwerte werden durch Menschen aus der Spezifikation abgeleitet
- mehrfache, unabhängige Implementierung des SUT
- bei Vorhandensein einer formalen Spezifikation kann evtl. automatisch ein Modell generiert werden, gegen das getestet werden kann

Falls Menschen das Orakel aus der Spezifikation ableiten, besteht die Gefahr der Fehlinterpretation der Spezifikation. Nicht nur das SUT kann somit Fehler enthalten, sondern auch das Orakel kann fehlerhaft sein. In der Praxis stellt dies die häufigste Form der Orakelerstellung dar (s. Kap. 2.5).

Die unabhängige Implementierung mehrerer verschiedener Versionen (n−version programming (s. auch diversifizierender Test in Anh. A)) des SUT ist mit höheren Entwicklungskosten verbunden. Dafür sinken der Aufwand und damit die Kosten für den Testprozess. Allerdings können auf diese Art des Testens nur Fehler gefunden werden, die lediglich in einer Version enthalten sind. In jeder Version enthaltene Fehler bleiben unentdeckt[2].

Formale Spezifikationen (s. Anh. A) sind ebenso wie automatisch generierte Modelle zur Überprüfung des Verhaltens des SUT in der Praxis nicht weit verbreitet.

2.2.5 Testendekriterium

Die Fragen „Wann wurde genug getestet?“ bzw. „Wann kann die Testphase beendet werden?“ lassen sich nur sehr schwer beantworten. Allein aus wirtschaftlichen Gründen muss die Testphase bereits begrenzt werden. Abb. 2.2 zeigt die Qualitätskosten[3] über dem Qualitätsgrad des SUT aufgetragen. Sie setzen sich aus den exponentiell steigenden Prüf− und Fehlerverhütungskosten sowie den exponentiell fallenden Fehlerkosten zusammen. Im Schnittpunkt beider Kurven herrscht das optimale Qualitätsniveau, obwohl noch weitere Fehler im SUT vorhanden sind. Dieses Qualitätskostenminimum stellt den ökonomischen Grund für die Begrenzung der Testphase dar. Eine weitere Erhöhung des Qualitätsgrades, also die Reduzierung weiterer vorhandener Fehler, ist mit einem enormen Prüfaufwand verbunden und führt zu einem Anstieg der Qualitätskosten.

[2] Nach *Liggesmeyer* (2005a) entstehen beim n−version programming ca. 5% gleiche Fehler in den verschiedenen Programmversionen.

[3] Laut *DIN 55350-11* versteht man unter Qualitätskosten alle Kosten, die durch Tätigkeiten der Fehlerverhütung, der planmäßigen Qualitätsprüfung sowie durch intern oder extern festgestellte Fehler verursacht werden.

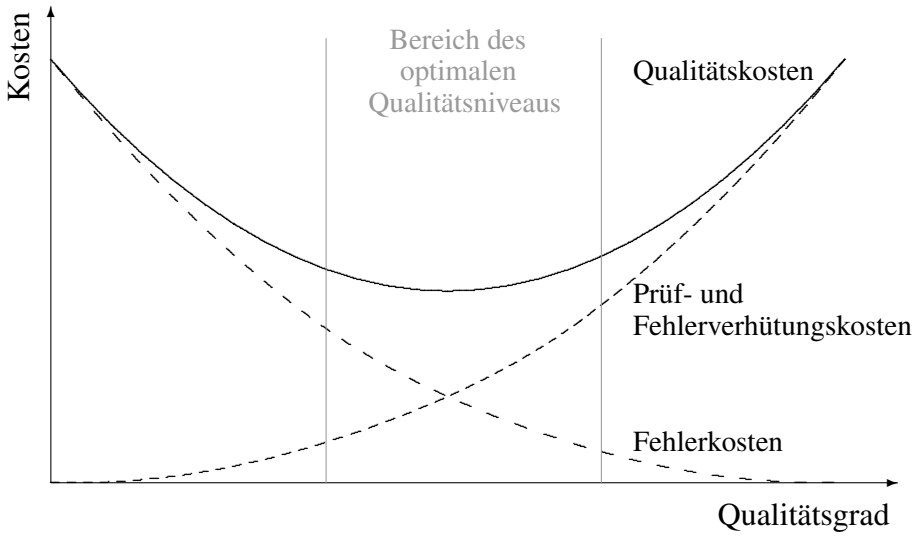


Abbildung 2.2: Qualitätskosten (*Vitrián*, 2004)

Weitere Gründe für die Begrenzung der Testphase stellen so genannte Testendekriterien dar. Folgende vier Testendekriterien gibt *Schmid* (2003) an:

Fertigstellungstermin Stellt der Fertigstellungstermin das Testendekriterium dar, so kann davon ausgegangen werden, dass die Testphase entsprechend gekürzt wurde. Grundsätzlich ist jedoch anzumerken, dass jeder Entwicklungsprozess zeitlich begrenzt ist, und die Testphase daher nicht beliebig verlängert werden kann.

Anzahl gefundener Fehler Ziel des Tests ist u.a. das Auffinden von Fehlern. Der Abbruch der Testphase durch die Vorgabe einer bestimmten Anzahl von entdeckten Fehlern kann daher als Testendekriterium dienen.

Erreichter Überdeckungsgrad nach Methode Falls die Testfälle anhand einer Methode ermittelt wurden, kann das Testen beendet werden, wenn alle Testfälle keine Fehler mehr finden und ein vorher festgelegter Überdeckungsgrad erreicht wurde. Beispiele stellen z.B. die Anweisungsüberdeckung und Zweigüberdeckung (s. Anh. A.2) oder die Anforderungsüberdeckung (s. Kap. 3.4.1) dar. In Kap. 5.1.3 werden weitere auf der Classification−Tree Method basierende Überdeckungsmaße vorgestellt, welche ebenso als Testendekriterien herangezogen werden können.

Durchschnittliche Kosten pro entdecktem Fehler Durch die Begrenzung der durchschnittlichen Kosten pro entdecktem Fehler existiert ein weiteres Testendekriterium. Als Grundlage für die Festlegung des Grenzwertes können z.B. Erfahrungswerte aus vergangenen Projekten dienen.

2.2.6 Teststrategie

Eine geeignete Teststrategie ist notwendig, da aufgrund der hohen Systemkomplexität ein vollständiger Test des Systems in der Regel nicht durchführbar ist. Idealerweise sollten die wichtigsten Fehler so früh und kostengünstig wie möglich gefunden werden (*Pol et al.*, 2002). Ziel einer Teststrategie ist es daher, auf der Grundlage von Risikobewertungen und evtl. vorhandenem Wissen um Schwachstellen des Systems, Prioritäten bzgl. der zu testenden Systemteile festzulegen. Kritischen Systemteilen sollte dabei mehr Aufmerksamkeit gewidmet werden, während weniger kritische Systemteile entsprechend ihrer Einstufung weniger intensiv getestet werden können. Bei sicherheitsrelevanten Systemen müssen zudem teilweise existierende Normen und Standards entsprechend berücksichtigt werden.

Außerdem kann eine Teststrategie so genannte Integrationsstrategien beinhalten, welche die Reihenfolge der Tests beschreiben. Folgende zwei Integrationsstrategien sind in der Praxis am häufigsten anzutreffen:

top−down Bei der top−down Integrationsstrategie steht das Gesamtsystem im Vordergrund. Details der einzelnen Module und Funktionen werden dabei vorerst vernachlässigt. Erst im Laufe des Tests werden immer mehr Details betrachtet, bis schließlich das gesamte System getestet wurde. Vorteil dieses Vorgehens ist der frühzeitige Test der Systemschnittstellen. Das Gesamtverhalten des SUT kann dadurch frühzeitig beurteilt werden.

bottom−up Die bottom−up Strategie testet dagegen zuerst die einzelnen Bestandteile des SUT, bevor der Test des Gesamtsystems durchgeführt wird. Dadurch liegen Aussagen bzgl. der Gesamtfunktionalität erst in späten Phasen des Tests vor.

Des Weiteren existieren einige weitere Integrationsstrategien wie z.B. sandwich oder vertical− und horizontal−slicing (*Hennell et al.*, 1987).

Eine Teststrategie stellt damit eine wichtige Grundlage für die Testaktivität der Testplanung (s. Kap. 2.4) dar.

2.3 Fehlerentdeckung und Fehlerbehebung

Nur entdeckte Fehler können behoben werden. Für die Identifikation von Fehlern müssen nach *Morell* (1990) drei Voraussetzungen erfüllt sein:

Defektausführung Der Teil des SUT, in dem der Fehler enthalten ist, muss bei dem durchgeführten Test beansprucht bzw. ausgeführt werden.

Fehlerfortpflanzung Bei der Ausführung des fehlerhaften SUT−Teils muss sich der Fehler auswirken und im weiteren Ablauf im Systemzustand halten.

Fehlerbeobachtung Der fehlerhafte Systemzustand muss im weiteren Ablauf als Fehlverhalten beobachtet werden.

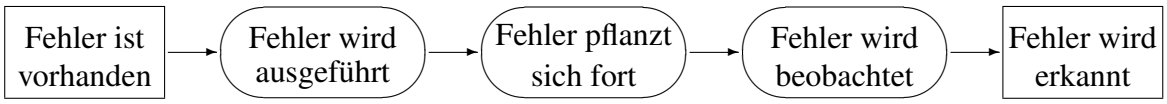


Abbildung 2.3: Wirkkette bei der Fehlererkennung (*Bergmann*, 1998)

Abb. 2.3 veranschaulicht die Wirkkette bei der Fehlererkennung. Demnach kann ein Fehler nur beobachtet werden, wenn die komplette Wirkkette der Fehlererkennung durchlaufen wird. Der fehlerhafte Teil des SUT muss zur Ausführung kommen, der Fehler muss sich fortpflanzen und schließlich beobachtet werden. Wird die Wirkkette dagegen nicht vollständig durchlaufen, d.h. wird entweder der fehlerhafte Teil des SUT nicht ausgeführt, pflanzt sich der Fehler nicht fort oder wird kein Fehlverhalten beobachtet, so bleibt der Fehler unerkannt.

Grundlage zur Fehlerentdeckung ist die Fähigkeit, zwischen korrekter und fehlerhafter Realisierung des Systems unterscheiden zu können. Diese Differenzierung erfolgt anhand der Auswertung der vom SUT eingenommenen und nach außen sichtbaren Ausgaben bzw. Zustände durch das Testorakel (vgl. Kap. 2.2.4), welches die korrekte Realisierung des SUT kennt.

Eine korrekte Realisierung des Systems R überführt das System vom Zustand S_0 bei jeder erlaubten Eingabe E in den Zustand S innerhalb des Zeitintervalls T, wie es die funktionale Spezifikation $Spec.$ festlegt (*Bergmann*, 1998).

$$\forall E: \quad S_0 \xrightarrow[T]{E} S\,|^{R}_{korrekt} \quad \equiv \quad S_0 \xrightarrow[T]{E} S\,|^{Spec.} \tag{2.1}$$

Eine fehlerhafte Realisierung des Systems R überführt das System vom Zustand S_0 bei mindestens einer erlaubten Eingabe E nicht in den Zustand S innerhalb des Zeitintervalls T.

$$\exists E: \quad S_0 \xrightarrow[T]{E} S\,|^{R}_{fehlerhaft} \quad \not\equiv \quad S_0 \xrightarrow[T]{E} S\,|^{Spec.} \tag{2.2}$$

Eine Besonderheit der Fahrzeugelektronik ist, dass der Systemzustand S permanent neu berechnet wird (s. Kap. 3.3). Die darin eingebettete Software „kann aufgrund der ständigen Interaktion mit dem technischen Umfeld während der Ausführung zu unterschiedlichen Zeitpunkten Eingaben aufnehmen und zu unterschiedlichen Zeitpunkten Ausgaben produzieren“ (*Bergmann*, 1998).

Ein SUT ist nach Gl. 2.2 fehlerhaft, wenn das SUT ausgehend vom Zustand S_0 nicht innerhalb des Zeitintervalls T in den Zustand S gemäß der Spezifikation $Spec.$ übergeht. Demzufolge ist jeder Fehler entweder ein

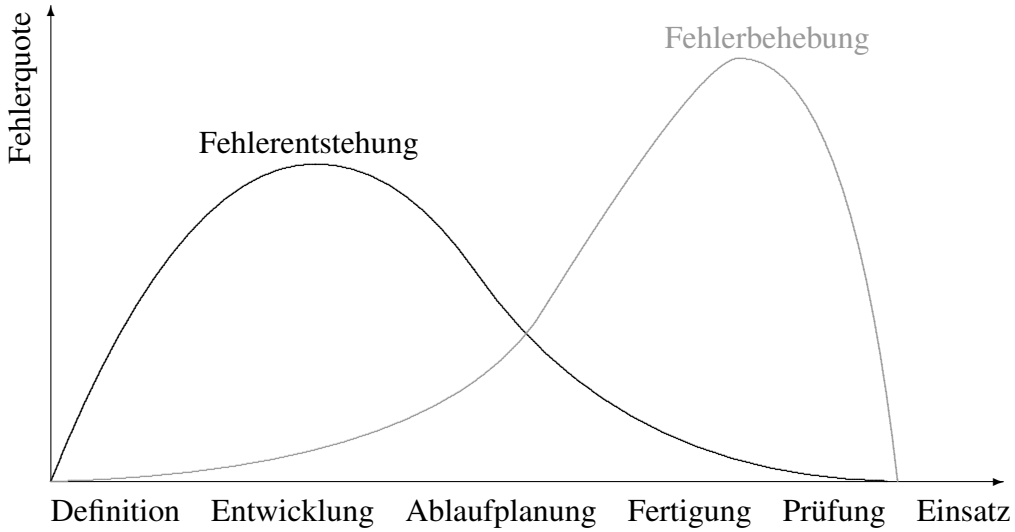


Abbildung 2.4: Fehlerentstehung und –behebung (*Vitrián*, 2004)

- Operationsfehler oder ein
- Zeitfehler.

In Kap. 2.3.1 werden diese zwei Fehlerklassen weiter unterteilt.

Zur Behebung erkannter Fehler wird nach Durchführung eines erfolgreichen[4] Tests ein Fehlerbehebungsprozess initiiert. Dieser Prozess besteht nach *Riedemann* (1997) aus folgenden drei Schritten:

1. Erkennen der Fehlersymptome durch einen Soll–/Istvergleich von spezifiziertem (bzw. erwartetem) und realisiertem (bzw. tatsächlichem) Verhalten.
2. Auffinden der Fehlerursache, indem die Funktion und die Struktur des SUT, insbesondere die SUT–Ausführung und –Steuerung, betrachtet werden.
3. Korrektur des SUT, wobei die Abänderung wieder zu testen ist, um sicherzustellen, dass der Fehler korrigiert wurde und keine neuen Fehler eingefügt wurden.

Fehlerentdeckung (Schritt 1) und Fehlerursachenfindung (Schritt 2) machen 95% des Aufwands der Fehlerbehebung aus, wohingegen die Fehlerkorrektur (Schritt 3) lediglich 5% des Aufwands verursacht (*Riedemann*, 1997). Für eine effektive Fehlerbehebung ist die Reproduzierbarkeit der durchgeführten Tests ein wesentlicher Faktor (s. Kap. 2.5).

[4] Ein Test gilt bei der destruktiven Herangehensweise (vgl. Kap. 2.1) als erfolgreich, wenn Hinweise auf die Existenz von Fehlern vorliegen.

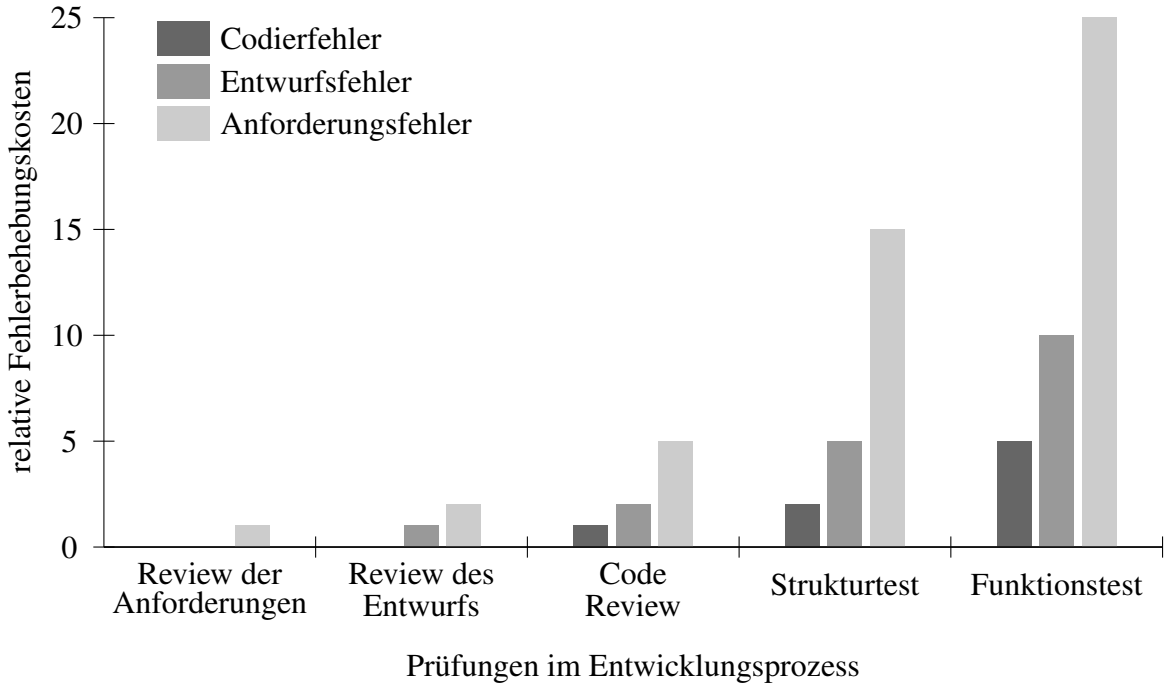


Abbildung 2.5: Relative Fehlerbehebungskosten in Abhängigkeit von der Verweilzeit (*Frühauf et al.*, 2004)

Frühauf et al. (2004) beschreiben die Erfahrung, dass Fehler auf derselben Abstraktionsebene entdeckt werden, auf der sie auch begangen wurden. Dementsprechend werden früh im Entwicklungsprozess begangene Fehler spät entdeckt. Abb. 2.4 visualisiert diese Lücke. Da 55% aller Fehler nach *Bergsmann & Achtert* (2003) in der Anforderungs– und Entwurfsphase entstehen, werden entsprechend viele erst in späten Phasen des Entwicklungsprozesses entdeckt. Betrachtet man zusätzlich die Fehlerbehebungskosten (s. Abb. 2.5), die mit der Verweilzeit eines Fehlers exponentiell ansteigen, so folgt, dass die frühe Fehlerentdeckung mit hohem Aufwand vorangetrieben werden sollte. In diesem Zusammenhang wird oftmals von Frontloading gesprochen. Dabei wird versucht, Entwicklungstätigkeiten (z.B. durch Simultaneous Engineering (SE)) zu parallelisieren und in frühere Phasen des Entwicklungsprozesses zu verlagern. Tests sind ein unverzichtbarer Bestandteil des Absicherungsprozesses. Sie können jedoch erst in späten Entwicklungsphasen beginnen, sobald erste Prototypen existieren. Zur Früherkennung von Irrtümern und Fehlern sind daher in besonderem Maße statische Testverfahren wie z.B. Reviews (s. Anh. A) geeignet.

2.3.1 Fehlerklassifikation

Die Klassifikation von Fehlern ist wichtig, um z.B. Fehlerstatistiken zu erheben, damit häufige Fehler zu identifizieren und so die Grundlage für eine Fehlervermeidung zu legen. *Ligges-*

meyer (1990) nennt folgende Kriterien zur Bildung von entsprechenden Fehlerkategorien:

- betroffene Qualitätseigenschaft
- Phase der Fehlerentstehung
- Phase der Fehlererkennung und –behebung
- Aufwand zur Fehlerbehebung
- Art der Inkonsistenz zur Spezifikation
- Fehlerursache
- Fehlerwirkung
- Art des strukturellen Merkmals im Programmtext

Die Definition der Fehlerkategorien hängt dabei sehr stark von der jeweiligen Ausprägung des SUT sowie dem Verwendungszweck der Fehlerstatistik ab. Dementsprechend existiert eine Vielzahl von Klassifikationsmöglichkeiten. Handelt es sich dabei z.B. um eine Funktion, deren Quellcode bekannt ist, so können strukturelle Merkmale des Programmtextes nützliche Informationen liefern. Liegt der Quellcode dagegen nicht vor, ist dieses Kriterium völlig ungeeignet. Dient die Statistik lediglich der Zuordnung von Fehlern zu der jeweiligen Entwicklungsphase, so sind die strukturellen Merkmale, falls sie zugänglich sind, ebenfalls irrelevant.

Grundlage für die Überprüfung der Korrektheit und Zuverlässigkeit der betrachteten eingebetteten Software bildet die funktionale Spezifikation[5]. Aus diesem Grund ist für die Bildung von für diese Arbeit relevanten Fehlerkategorien hauptsächlich das Kriterium der Art der Inkonsistenz zur Spezifikation von Bedeutung. Im übergeordneten Abschnitt wurden bereits Operationsfehler und Zeitfehler als zwei Inkonsistenzen zwischen Spezifikation und Realisierung vorgestellt. Diese zwei Kategorien werden im Folgenden weiter klassifiziert.

Operationsfehler lassen sich in folgende drei Unterkategorien klassifizieren (*Schmid*, 2003):

- fehlende Operation
- unnötige Operation
- fehlerhafte Operation

Zeitfehlern sind folgende zwei Unterkategorien zugeordnet:

[5]Der Zulieferer liefert dem Automobilhersteller in der Regel ein System als Black–Box. Der Quellcode ist dabei nicht einsehbar.

- Operation erfolgt zu früh bzw. zu kurz
- Operation erfolgt zu spät bzw. zu lang

In der Regel kann jedoch nicht immer zwischen einem Operationsfehler und einem Zeitfehler unterschieden werden, da sich die Fehlereffekte bis zu ihrer Entdeckung mehrfach wandeln können. Operationsfehler können sich als Zeitfehler auswirken und umgekehrt (*Bergmann*, 1998). So kann z.B. eine Operation als fehlend klassifiziert werden, obwohl sie vorhanden ist und aufgrund eines Zeitfehlers nicht mehr im Beobachtungszeitraum ausgeführt wurde. Zur genauen Eingrenzung des Fehlers müssen daher entweder weitere Tests durchgeführt oder aber der Quellcode muss mit in den Test einbezogen werden.

Neben der soeben vorgestellten Fehlerklassifizierung existiert der Standard *IEEE Std 1044−1993* zur Klassifikation von Software−Anomalien, der jedoch in der Praxis noch nicht weit verbreitet ist.

2.4 Testaktivitäten

Grundlage für einen systematischen Test bildet die Aufteilung des Tests in einzelne Testaktivitäten. „Dadurch wird eine strukturierte Vorgehensweise und die Definition von aufeinander aufbauenden Zwischenergebnissen gefördert“ (*Wegener*, 2001) und eine bessere Dokumentation und somit Nachvollziehbarkeit des Tests erreicht.

Riedemann (1997) teilt den Testablauf in drei grobe Phasen ein:

- Testvorbereitung
- Testausführung
- Testauswertung

Jede dieser Phasen setzt sich wiederum aus einzelnen Testaktivitäten (*Grimm*, 1995) zusammen. Diese sind in Abb. 2.6 inkl. der einzelnen groben Phasen dargestellt.

Die wichtigste Testaktivität ist dabei die Ermittlung von Testfällen, mit denen der Test durchgeführt werden soll. Sie beruht idealerweise auf den später vorgestellten Testverfahren (Anh. A.1) und baut auf der funktionalen Spezifikation oder dem SUT selbst auf. Die Testfallermittlung entscheidet über die Qualität des Tests und damit über die Möglichkeit, Fehler aufzudecken. Leider wird diese wichtige Testaktivität in der Praxis häufig unsystematisch durchgeführt, wie u.a. die Online−Umfrage von *Armbrust et al.* (2004b) verdeutlicht (s. Kap. 4.3).

Ein Testfall legt dabei eine Menge von Eingabewerten fest, die für die Prüfung des SUT unter einem bestimmten Gesichtspunkt herangezogen werden kann. Für die spätere Testdurchführung müssen für die ermittelten Testfälle konkrete Testdaten generiert werden. Dies

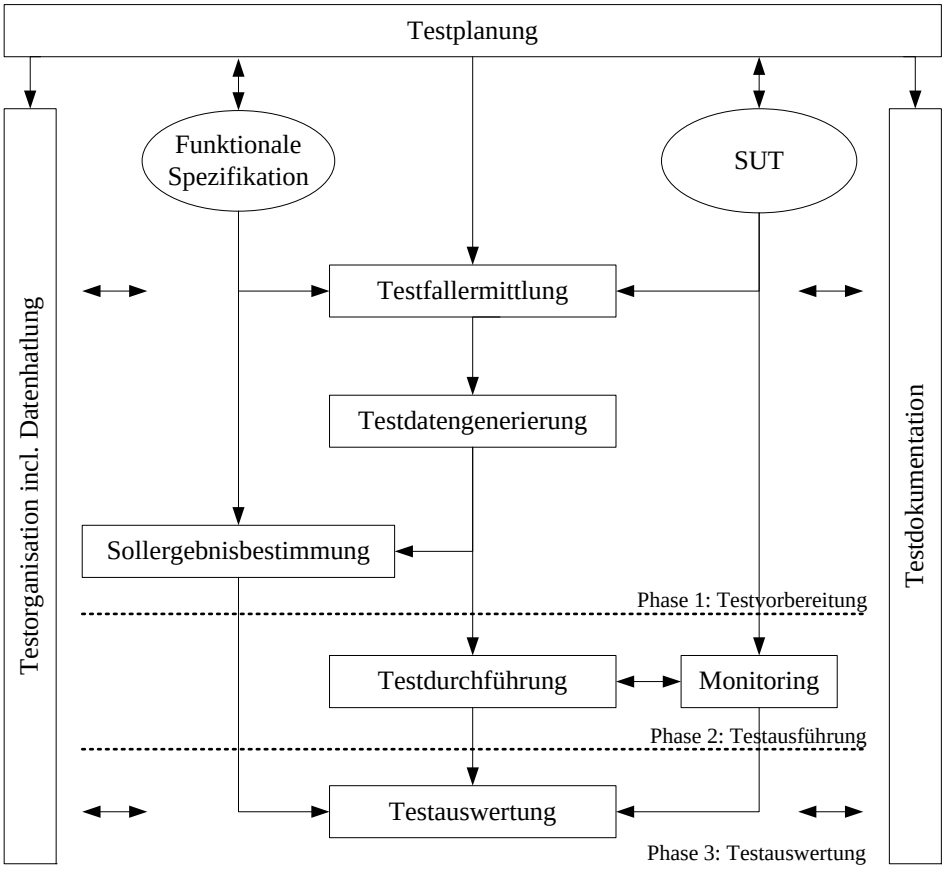


Abbildung 2.6: Die Testaktivitäten und ihr Zusammenwirken (*Schmid*, 2003)

ist Inhalt der Testdatengenerierung. Zur Beurteilung der Tests müssen für die gewählten Testdaten Sollergebnisse bestimmt werden. Hierfür kommen so genannte Testorakel zum Einsatz (vgl. Kap. 2.2.4).

Während der Testdurchführung wird das SUT mit den ausgewählten Testdaten ausgeführt. Monitoring hat den Zweck, Informationen über das Programm zur Laufzeit zu liefern.

Die abschließende Testauswertung beurteilt mithilfe des erstellten Testorakels das Verhalten des SUT bzw. die Istergebnisse unter Berücksichtigung definierter Akzeptanzkriterien sowie der festgelegten Sollergebnisse. Für die Testauswertung gibt es nach *Schmid* (2003) zwei Möglichkeiten:

- Auswertung zur Laufzeit
- Auswertung nach der Testdurchführung

Großer Vorteil bei der Auswertung zur Laufzeit ist die bestehende Möglichkeit der Steuerung des Tests anhand eines vorliegenden Teilergebnisses. Nachteil ist dagegen die eventuelle Beeinflussung des SUT, wodurch das Testergebnis verfälscht werden kann.

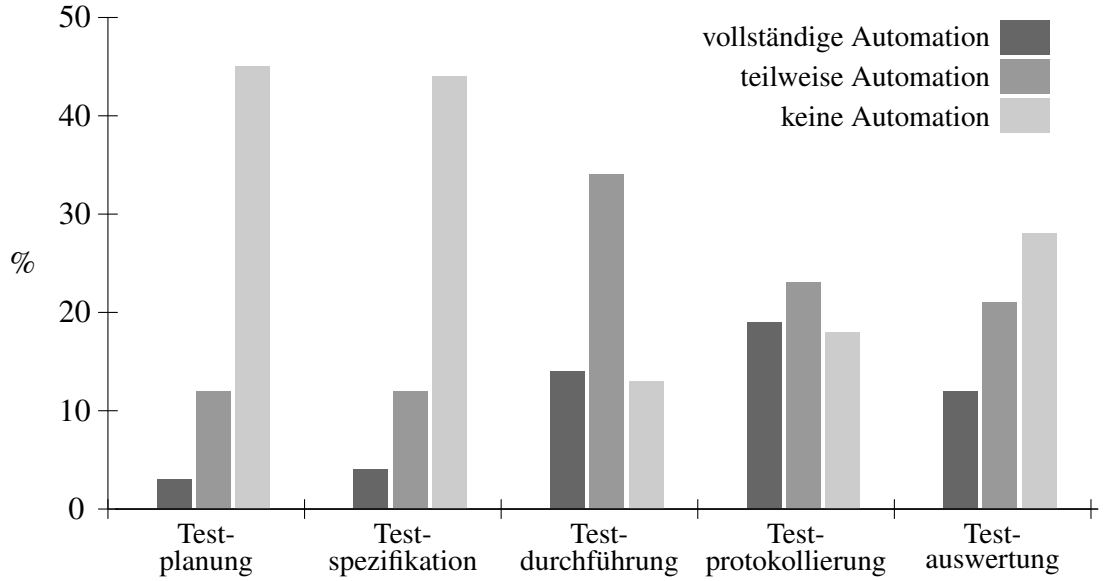


Abbildung 2.7: Automatisierungsgrad von Qualitätssicherung (*Armbrust et al.*, 2004b)

Die Testaktivitäten Testplanung, Testorganisation und Testdokumentation sind testübergreifend. Im Rahmen der Testplanung wird die Vorgehensweise für den Test festgelegt, ebenso Testziele (vgl. Kap. 2.2.3) und Testende– bzw. Testabbruchkriterien (vgl. Kap. 2.2.5). Die Testorganisation stellt die Verwaltung der SUT, sowie der dazugehörigen Testfälle sicher. Damit ist die Testorganisation entscheidend für die Regressionsfähigkeit der Testdaten. Weiterhin fasst die Testorganisation alle Tätigkeiten zusammen, die für die technische Durchführbarkeit des Tests notwendig sind. Die Testdokumentation dient der Nachvollziehbarkeit der Tests. Bestandteile der Dokumentation sind eine ausführliche Erläuterung der Teststrategie und –ziele sowie die detaillierte Beschreibung aller weiteren in der Testplanung und Testorganisation getroffenen Festlegungen. Die Testergebnisse sollten so aufbereitet werden, dass Abweichungen zwischen Ist– und Sollwerten deutlich aufgezeigt werden, die Erfüllung der Testziele einfach bewertet werden kann und die entdeckten Fehler in einer Fehlerstatistik dargestellt werden können.

2.5 Testautomatisierung

„Manual Testing doesn't work. It never did work very well, it doesn't work now, and it won't work in the future. Manual testing is ill–contrived self–deception. It confuses sweat with accomplishment. And worst of all, it leads to false confidence“ (*Beizer*, 1995).

Viele andere Autoren formulieren diesen Sachverhalt nicht ganz so dramatisch, sind sich jedoch bzgl. der Notwendigkeit des Einsatzes von Testautomatisierung einig. Testautomatisierung bedeutet in diesem Zusammenhang, dass eine oder mehrere der in Kap. 2.4 vor-

gestellten Testaktivitäten nicht von Menschen sondern von Maschinen durchgeführt werden. *Armbrust et al.* (2004b) untersuchten in einer Online−Umfrage den „Stand der Praxis von Software−Tests und deren Automatisierung“. Der dabei ermittelte Automatisierungsgrad der einzelnen Testaktivitäten ist in Abb. 2.7 dargestellt. Es zeigt sich, dass die Aktivitäten Testplanung sowie Testspezifikation größtenteils manuell durchgeführt werden und alle nachfolgenden Aktivitäten über einen deutlich höheren Automatisierungsgrad verfügen. Testprotokollierung gefolgt von der Testdurchführung weist dabei den höchsten Automatisierungsgrad auf. Die Aktivität der Testauswertung findet wieder zum großen Teil manuell statt. Dies deutet darauf hin, dass das Testorakel (s. Kap. 2.4) größtenteils von Menschen anhand der funktionalen Spezifikation erstellt wird bzw. Menschen das Testorakel darstellen.

Die Forderung nach Testautomatisierung wird von *Guddat* (2003) mit folgenden Erkenntnissen gestützt:

Komplexität, Kosten Aufgrund der hohen Komplexität der Systeme sind der Aufwand beim manuellen Test der Systeme sowie die damit verbundenen Kosten sehr hoch.

Fehlerträchtigkeit Manuelles Testen ist fehlerträchtiger als automatisiertes Testen. Zwei mögliche Fehlerquellen stellen dabei z.B. falsches Ablesen oder Fehleingaben von Werten dar. Zurückzuführen ist diese Tatsache auf den Menschen als ausführendes Organ, der bei seiner Arbeit Fehler begeht. *Beizer* (1995) nennt z.B. für professionelle Kräfte drei Fehler pro tausend Tastenanschläge als durchschnittliche Fehlerrate.

Reproduzierbarkeit bzw. Regressionsfähigkeit Reproduzierbarkeit ist ein entscheidender Faktor für eine effektive Fehlerbehebung. Nur wenn entdeckte Ausfälle reproduziert werden können, ist eine Fehlerursachenfindung und die sich anschließende Fehlerbehebung möglich (vgl. Kap. 2.3). Nach Behebung bekannter Fehler müssen Regressionstests durchgeführt werden, um sicherzustellen, dass die Fehler wirklich behoben und keine neuen Fehler eingefügt wurden. Diese Regressionsfähigkeit der automatischen Testdurchführung ist ein weiterer wichtiger Punkt auf dem Weg zu fehlerarmer Software.

Wiederverwendbarkeit Werden die Tests in Form von Skripten, Bibliotheken oder in anderer Form festgehalten, können evtl. Teile der Tests oder sogar ganze Tests in späteren Projekten wieder verwendet werden. Manuelle Tests erzeugen dagegen bei erneuter Durchführung einen erneuten Aufwand, der wiederum mit Kosten verbunden ist. Der automatisierte Test ist damit zwar mit einem höheren Initialaufwand verbunden (s. Abb. 2.8), weil entsprechende Ablaufskripte bzw. −programme vor der erstmaligen Testausführung erstellt werden müssen. Nach mehreren Testwiederholungen relativiert sich dieser Aufwand jedoch beträchtlich im Vergleich zum manuellen Test, da für die Durchführung im Gegensatz zum manuellen Test kein Personaleinsatz mehr erforderlich ist. Bei der automatischen Testdurchführung kann sich der Testersteller anderen

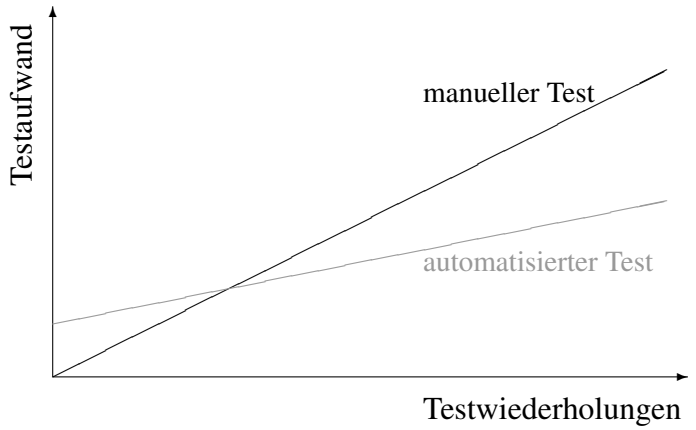


Abbildung 2.8: Testaufwand bei manuellem und automatisiertem Test (*Seiler*, 2006)

Tätigkeiten widmen und z.B. neue Testfälle ermitteln, wodurch evtl. weitere Fehler entdeckt werden können und damit die Qualität des SUT weiter gesteigert wird.

3 Entwicklungs– und Testprozess von Automobilelektronik

Das vorliegende Kapitel ist der Automobilelektronik gewidmet. Nach einer kurzen Betrachtung von Anforderungen verschiedener Benutzergruppen an Automobilelektronik und deren Aufbau wird sowohl der Entwicklungs– als auch der Testprozess inkl. der eingesetzten Testsysteme vorgestellt. Besonderer Fokus liegt dabei auf dem Test– bzw. Absicherungsprozess mittels Hardware–in–the–Loop Testsystemen. Abschließend wird die vorliegende Problemstellung konkretisiert sowie ein daraus abgeleitetes Fazit präsentiert.

3.1 Einleitung

Das heutige Automobil enthält neben zahlreichen mechanischen Komponenten immer mehr elektrische bzw. elektronische Komponenten. Die Verschmelzung von Mechanik, Elektrik/Elektronik (E/E) und Informatik wird als Mechatronik bezeichnet und ermöglicht eine unvorstellbare Fülle von neuen Funktionalitäten. Einige der erkannten Möglichkeiten werden bereits im Automobil umgesetzt, die meisten warten jedoch noch auf ihre Realisierung. *Conrad* (2004) sieht dabei nicht die technischen Möglichkeiten, wie z.B. die Leistungsfähigkeit von Mikroprozessoren, sondern mehr und mehr den Entwicklungs– und Testprozess der Systeme und der darin eingebetteten Software als limitierenden Faktor.

3.2 Anforderungen an Automobilelektronik

In Abb. 3.1 sind einige Benutzergruppen eines Fahrzeugs dargestellt. Jede dieser Gruppen stellt gewisse Anforderungen an ein Fahrzeug und damit verbunden auch an die darin enthaltene Elektronik.

Der Fahrer möchte sein Fahrzeug z.B. zu jedem Zeitpunkt, bei jedem Wetter und an jedem Ort der Erde nutzen können. Daraus resultiert ein geforderter Einsatzbereich für Steuergeräte von $-40°C$ bis $+120°C$ sowie eine Resistenz gegen Feuchte, Nässe und chemische Substanzen wie z.B. Salznebel, Öle oder Treibstoff (*Spitzer*, 2001). Mechanische Einflüsse wie Vibrationen oder extreme Beschleunigungen dürfen ebenfalls die Funktionsfähigkeit nicht negativ beeinflussen. Die Airbag–Elektronik muss auch bei einer extremen Situation, wie

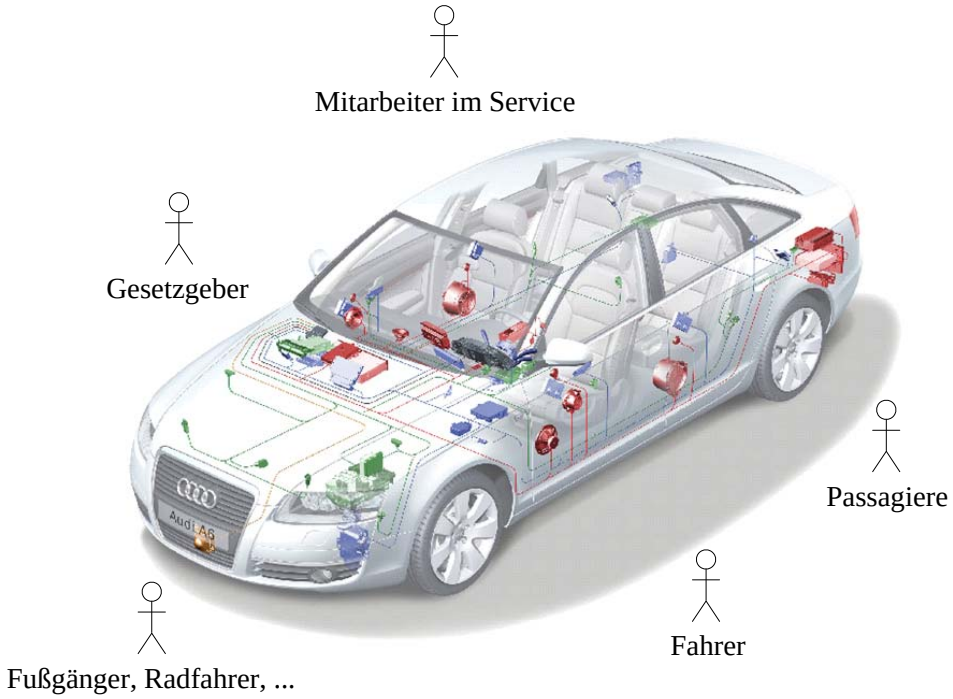


Abbildung 3.1: Benutzergruppen eines Fahrzeug (*Schäuffele & Zurafka*, 2003)

z.B. einem Crash, stets einwandfrei funktionieren. In diesem Zusammenhang spielt auch die Elektromagnetische Verträglichkeit (EMV) eine immer wichtigere Rolle. Vom Gesetzgeber sowie von zahlreichen Richtlinien und Normen werden Rahmenbedingungen wie z.B. Strahlungshöchstwerte, Emissionsgrenzen usw. vorgegeben. Mitarbeiter im Service sind an einer komfortablen Diagnose des Fahrzeugs interessiert, um z.B. bei einer Fehlfunktion schnell die Ursache zu identifizieren und entsprechende Maßnahmen einzuleiten. Passagiere können an einem weiten Unterhaltungsangebot interessiert sein, woraus wiederum Anforderungen an die verwendeten Bussysteme entstehen. Weitere Benutzergruppen wie Fußgänger, Radfahrer, andere Fahrzeuge und Verkehrsteilnehmer stellen viele weitere Anforderungen, die ebenfalls berücksichtigt werden müssen. Die soeben erwähnten mechanischen, elektrischen und ergonomischen Anforderungen sind zwar für die Entwicklung von Automobilelektronik von entscheidender Bedeutung, für die vorliegende Arbeit sind sie jedoch zweitrangig.

Wie bereits in Kap. 2.1.1 angedeutet, liegt der Fokus dieser Arbeit auf der Überprüfung der Korrektheit und Zuverlässigkeit von in Fahrzeugsysteme eingebetteter Software. Die durch die eingebettete Software realisierten Funktionen stellen daher die in Kap. 2.2.1 beschriebenen Testobjekte dar und müssen ihrer funktionalen Spezifikation entsprechen, um als korrekt eingestuft zu werden. Die Zuverlässigkeit der Funktionen wird zudem durch spezielle Tests, wie z.B. durch Simulation von Sensorausfällen oder hohen Buslasten, überprüft (s. Kap. 3.5.1.2).

Im Allgemeinen ist die eingebettete Software über mehrere Fahrzeugkomponenten verteilt.

Die dadurch realisierten Funktionen werden daher auch als verteilte Funktionen bezeichnet. Für die Überprüfung der verteilten Funktionen auf Korrektheit und Zuverlässigkeit sind einige spezielle Eigenschaften (Einbettung, Reaktivität, Vernetzung und Realzeit), welche die hier eingesetzten Fahrzeugsysteme von herkömmlichen Softwaresystemen abgrenzen, zu berücksichtigen (*Schmid*, 2003). Diese speziellen Eigenschaften werden in den folgenden Abschnitten näher betrachtet.

3.3 Aufbau der Automobilelektronik

Automobilelektronik unterteilt sich anhand der Bustopologie im Wesentlichen in die drei Bereiche (s. Abb. 3.2):

- Antrieb und Fahrwerk
- Karosserie bzw. Komfort
- Infotainment

Zum Bereich des Antriebs zählen u.a. der Motor sowie das Getriebe, mit den dazugehörigen Steuergeräten. Bekannteste Vertreter aus dem Bereich der Fahrwerkselektronik sind z.B. das Anti Blockier System (ABS), das Elektronische Stabilitätsprogramm (ESP) sowie die Servolenkung. In letzter Zeit hinzugekommen sind Fahrerassistenzsysteme wie z.B. das Adaptive Cruise Control (ACC) oder das Lane Departure Warning (LDW). Zur Karosserie– bzw. Komfortelektronik gehören z.B. Zugangssysteme wie die Zentralverriegelung und die Diebstahlwarnanlage, aber auch Funktionen zur Verstellung von Sitzen, Spiegeln und nicht zuletzt auch die Beleuchtung des Innen– und Außenraums. Das Navigationssystem, Telefon und Radio bilden einige Bestandteile des Infotainment.

Abb. 3.2 hebt die drei Bereiche hervor und führt die dem jeweiligen Bereich zugeordneten Komponenten auf. Die räumliche Anordnung der einzelnen Komponenten zeigt dagegen Abb. 3.1 auf Seite 26.

3.3.1 Systeme und Komponenten

Kernbausteine von Fahrzeugen sind Systeme, welche sich in beliebig viele Subsysteme unterteilen können. Subsysteme können wiederum aus beliebig vielen weiteren Subsystemen bestehen. Durch diese Dekomposition entsteht zunächst eine funktionale Systemarchitektur, die anschließend auf eine physikalische Systemarchitektur (s. Abb. 3.2) abgebildet wird. Einige Beispiele von Fahrzeugsystemen sind z.B. das Motormanagement, die Klimaregelung oder die Lichtsteuerung. Dabei sind die in einem System/Subsystem definierten Funktionen in der Regel auf mehrere Komponenten sowie über die verschiedenen Bereiche Antrieb,

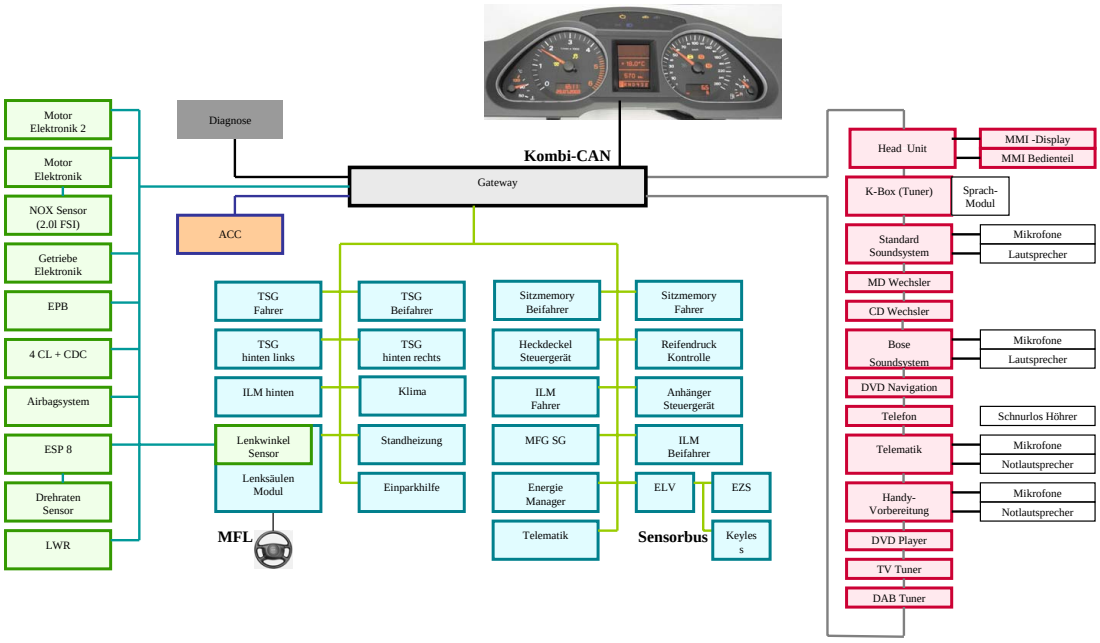


Abbildung 3.2: Bustopologie eines AUDI A6. Antrieb (links), Komfort (mitte), Infotainment (rechts) (*Derichsweiler*, 2005)

Komfort, Infotainment verteilt. Der Begriff verteilte Funktion wurde in diesem Zusammenhang bereits eingeführt (vgl. Kap. 3.2). Jede einzelne Komponente liefert damit einen gewissen Beitrag zur Gesamtfunktion des Fahrzeugs. Die volle Funktionalität wird jedoch erst durch das Zusammenwirken aller Komponenten erreicht. In diesem Zusammenhang spricht man auch von Einbettung, da das System in ein technisches Umfeld eingebettet ist und auch nur darin seine volle Funktionsfähigkeit entfalten kann. Für die bereits angesprochene Funktionsüberprüfung ist daher die Nachbildung des technischen Umfeldes inkl. aller benötigten Schnittstellen zwingend erforderlich.

Anstatt des Begriffs Komponente wird im Bereich der Automobilelektronik auch der Begriff Steuergerät (SG) oder aber auch die englische Bezeichnung Electronic Control Unit (ECU) verwendet. Einer Komponente können dabei mehrere Aktoren und Sensoren zugeordnet sein.

3.3.2 Aktoren und Sensoren

Der aktuelle Zustand des Fahrzeugs wird den Komponenten über zahlreiche in die Fahrzeugtopologie integrierte Sensoren mitgeteilt. Sensoren messen physikalische Größen wie z.B. Beschleunigungen, Temperaturen und liefern ein analoges oder digitales Ausgangssignal, welches vom Steuergerät zunächst aufbereitet wird (s. Abb. 3.3). Abhängig von der jeweiligen Funktionalität des Steuergerätes werden entsprechende Ausgangssignale berechnet und zur weiteren Verarbeitung aufbereitet. Aktoren setzen die in den Signalen kodierten

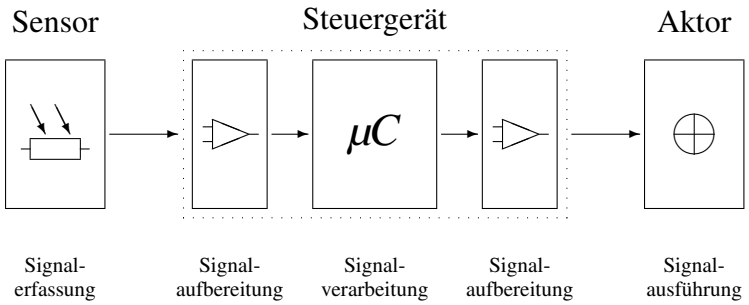


Abbildung 3.3: Aufbau eines Steuergerätes und dessen Sensor–/Aktor–Ansteuerung (*Schmid*, 2003)

Befehle durch Ansteuerung von Motoren, Pumpen oder Ventilen um und ermöglichen so eine Beeinflussung der anfangs durch die Sensoren gemessenen physikalischen Größen.

Diese ständige Interaktion mit der technischen Umgebung ist eine Eigenschaft von reaktiven Systemen. Für Fahrzeugsysteme, als klassische reaktive Systeme, sind bei der Berechnung der Reaktion neben Informationen aus dem technischen Umfeld die aktuellen, inneren Zustände der Steuergeräte sowie evtl. deren Vorgeschichte relevant.

Das Verhalten von Fahrzeugsystemen kann daher nicht durch einzelne Wertepaare beschrieben werden sondern wird durch Folgen von Eingabegrößen bestimmt. Neben dieser Abhängigkeit von zeitbehafteten Größen kann auch eine direkte Zeitabhängigkeit existieren. Ein Eingabevektor $\vec{e}$ eines Fahrzeugsystems hat damit im Allgemeinen die folgende Form (*Simmes*, 1997):

$$\vec{e}(t) = (e_1(t), ..., e_n(t), t) \tag{3.1}$$

Dabei stellt n die Anzahl der Eingabegrößen $e_1 ... e_n$ und t die Zeit dar. Neben dem Wert der Eingabegrößen ist auch ihr zeitlicher Bezug von Bedeutung, wodurch ein vollständiger Tests des eingebetteten Systems unmöglich ist.

3.3.3 Systemvernetzung

Auf mehrere Komponenten verteilte Funktionen benötigen einen Datenaustausch zwischen diesen Komponenten, um die Funktionen koordinieren zu können. Eine so genannte Kommunikationsmatrix beschreibt diese Kommunikation zwischen den verschiedenen Komponenten. Die Spalten der Matrix stellen dabei die Komponenten dar, während die Zeilen der Matrix den definierten Botschaften entsprechen. Jede Botschaft wird von genau einer Komponente gesendet und von mindestens einer Komponente empfangen, wobei eine Mehrfachnutzung von Botschaften die Regel ist. Die Matrix enthält den Identifier (ID) sowie weitere

Angaben, z.B. über den Typ (z.B. Broadcast) und die Sendeart (zyklisch, dynamisch), jeder Botschaft.

In den einzelnen Bereichen sind die Komponenten über Bussysteme vernetzt. Das Gateway realisiert die Vernetzung der einzelnen Bereiche (s. Abb. 3.2) untereinander. Je nach Bereich und den dort herrschenden Anforderungen an die Übertragungsrate kommen unterschiedliche Bussysteme zum Einsatz. Im Antriebs– und Komfortbereich ist der CAN (Control Area Network) Bus etabliert, während sich im Infotainmentbereich der MOST (Media Oriented System Transport) Bus aufgrund seiner hohen Datenrate durchgesetzt hat.

Neben Anforderungen an die geforderte Übertragungsrate der Bussysteme bestehen jedoch auch zeitliche Anforderungen für die Datenverarbeitung der Fahrzeugsysteme, die im Allgemeinen als Realzeitanforderungen bezeichnet werden. Die notwendigen Realzeitanforderungen leiten sich aus der Dynamik des zugeordneten technischen Prozesses ab, in den das jeweilige System eingebettet ist. Dabei kommt es weniger auf Schnelligkeit einer Reaktion sondern vielmehr auf die Einhaltung definierter Zeitschranken an (Rechtzeitigkeit). Eine zu frühe Reaktion kann ebenso wie eine zu späte Reaktion verheerende Konsequenzen nach sich ziehen. Für die Funktionsprüfung von eingebetteten Systemen ist daher neben der korrekten, logischen Funktionalität die Prüfung des Realzeitverhaltens[1] von entscheidender Bedeutung.

Die Vernetzung der Systeme durch Bussysteme bringt viele Vorteile mit sich, wie z.B. eine Reduzierung des Gewichts bzw. Volumens, da weniger Leitungen verlegt werden müssen. Die damit mögliche Mehrfachnutzung von Sensorik spart noch einmal Gewicht und Kosten. Zudem wird die Montagezeit aufgrund einer möglichen Modulbauweise verringert. Weniger Steckkontakte beherbergen weniger potenzielle Fehlerquellen und führen damit zu einer weiteren Qualitätssteigerung. Neue Komponenten lassen sich schnell in das vorhandene Netzwerk integrieren, wodurch eine hohe Erweiterungsfähigkeit des Systems erreicht wird. Änderungen können dabei meist durch Software realisiert werden.

Doch neben den Vorteilen der Vernetzung ergeben sich auch Nachteile, wie z.B. der enorme Anstieg der Komplexität des Gesamtsystems. Durch diese fahrzeugweite Vernetzung können sich die einzelnen Systeme gegenseitig beeinflussen, auch wenn dies ursprünglich nicht vorgesehen war. Entwickler können sich daher bei der System–Entwicklung bzw. –Anpassung nicht mehr auf ihr Einzelsystem beschränken, sondern müssen funktionale Abhängigkeiten im Gesamtsystem geeignet berücksichtigen.

3.4 Entwicklungsprozess der Automobilelektronik

Die Experten sind sich einig, dass aufgrund des immer höheren Stellenwertes von Elektronik im Automobil der Entwicklungsprozess von Automobilelektronik immer mehr zum bestim-

[1] *Wegener* (2001) stellt z.B. mit dem evolutionären Test ein auf den Test des Zeitverhaltens von Realzeit–Systemen spezialisiertes Testverfahren vor.

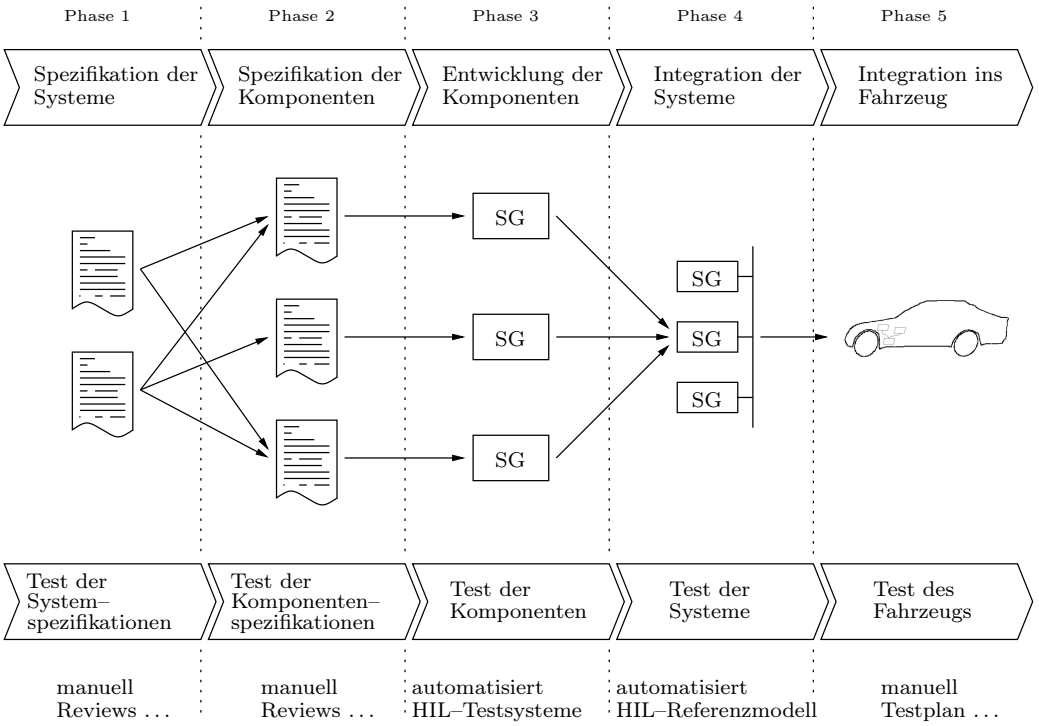


Abbildung 3.4: Schematischer Entwicklungs– und Testprozess für E/E–Systeme (*Schmid*, 2003)

menden Faktor in der Fahrzeugentwicklung wird. Der Entwicklungsprozess unterteilt sich grob in die drei Phasen:

- Spezifikationsphase
- Entwicklungsphase
- Integrationsphase

Jede Phase wird dabei von entsprechenden Tests begleitet (s. Abb. 3.4). In den folgenden drei Abschnitten wird jede dieser Phasen inkl. der zugehörigen Tests überblicksartig vorgestellt. Eine detaillierte Betrachtung des bei der AUDI AG bestehenden Testprozesses findet in Kap. 3.5 statt.

3.4.1 Spezifikationsphase

Während der Spezifikationsphase werden zuerst die später im Fahrzeug enthaltenen Systeme, Subsysteme und Komponenten (Steuergeräte) in Form von System– und Komponentenspezifikationen festgehalten (s. Abb. 3.4). Dabei müssen die Anforderungen der verschiedenen Benutzergruppen eines Fahrzeugs (s. Abb. 3.1) berücksichtigt werden.

Sind die Systeme, Subsysteme und Komponenten spezifiziert, kann die Verteilung der Systeme und Subsysteme (funktionale Systemarchitektur) auf die einzelnen Komponenten (physikalische Systemarchitektur) beginnen. Beeinflusst wird diese Verteilung von Systemen auf Komponenten vor allem durch Faktoren wie Platz– und Raumangebot, Kosten, Verkabelung sowie die Erfahrung der Entwickler (*Schmid*, 2003). Wie auch schon die Systemspezifikationen sind die Komponentenspezifikationen meist informeller Art und unterliegen damit einer gewissen Interpretationsfreiheit. Um diese Fehlerquelle zu beseitigen, werden den Spezifikationen zunehmend ausführbare Modelle beigefügt, die die exakte Funktion der Komponente beschreiben. Vermehrt wird auch anstatt gewöhnlicher Textverarbeitungsprogramme, das von der Firma *Telelogic AB* entwickelte Tool DOORS (Dynamic Object Oriented Requirements System) zur Unterstützung der Spezifikationsphase eingesetzt.

Jede Anforderung ist in DOORS durch eine ID eindeutig gekennzeichnet, so dass die Möglichkeit zur späteren Verknüpfung[2] von Tests mit den dazugehörigen Anforderungen besteht. Neben dieser eindeutigen Kennzeichnung sollten Anforderungen weiterhin eindeutig, widerspruchsfrei, verständlich und testbar sein. Änderungen und Entscheidungen sollten zudem in einer Historie festgehalten werden (*Bender*, 2005). Genügen die Anforderungen bestimmten Voraussetzungen, so lässt sich die Qualität der Anforderungsdokumente mittels spezieller Metriken messen (*Recknagel & Rupp*, 2006).

Bereits in der Spezifikationsphase werden erste Prüfungen der Anforderungen durchgeführt. Hierbei kommen meist manuelle Verfahren wie z.B. Reviews der Komponenten– und Systemspezifikationen zum Einsatz. Liegt die Spezifikation etwa als Modell in formaler Form vor, so ist auch eine automatisierte Prüfung möglich. In der Praxis ist diese automatisierte Prüfung bisher jedoch nicht weit verbreitet (s. Anh. A). In diesem Zusammenhang präsentieren *Esser & Struss* (2007) einen natürlichsprachlichen Formalismus für Anforderungsspezifikationen. Ausgehend von diesen Spezifikationen werden automatisiert ausführbare Modelle erstellt, mit denen das beschriebene System z.B. auf Widerspruchsfreiheit und Vollständigkeit überprüft werden kann.

3.4.2 Entwicklungsphase

Die Entwicklung der Komponenten erfolgt durch verschiedene Zulieferer auf Grundlage der in der Spezifikationsphase entstandenen Dokumente (System– und Komponentenspezifikation). Im Lauf der Entwicklungsphase werden dem Fahrzeughersteller bzw. Original Equipment Manufacturer (OEM) vom Zulieferer verschiedene Prototypen (A–, B–, C–Muster) zur Verfügung gestellt, anhand derer der aktuelle Entwicklungsstand der jeweiligen Komponente beurteilt werden kann. Jeder Prototyp durchläuft dabei einen eigenen Entwicklungsprozess.

[2] Im Allgemeinen handelt es sich bei Verknüpfungen von Tests mit Anforderungen um $n:m$ Beziehungen mit $n,m > 1$. Sind alle Anforderungen mindestens einem Test zugeordnet, spricht man in diesem Zusammenhang von Anforderungsüberdeckung. Metriken zur Bewertung der Verknüpfung von Anforderungen mit Tests präsentieren z.B. *Rosenberg et al.* (1998).

A–Muster stellen Konzeptmuster dar und dienen der Überprüfung der Grundfunktionalität. In Form und Ausführung entsprechen A–Muster meist nicht dem Entwicklungsziel.

B–Muster sind funktionsfähige Muster, die eine ausreichende Betriebssicherheit für Erprobung auf dem Prüfstand (s. Kap. 3.5.1.2) und im Fahrzeug gewährleisten. Sie werden in der Regel in seriennaher Ausführung aus Hilfswerkzeugen hergestellt. Auf dem Weg zum C–Muster entstehen mehrere B–Muster–Stände. Jeder Stand stellt dabei eine Weiterentwicklung des vorherigen Standes dar und enthält im Idealfall bereits Behebungen aller entdeckten Fehler.

C–Muster werden unter Serienbedingungen aus Serienwerkzeugen gefertigt. Dabei müssen sämtliche Anforderungen bzgl. Funktion, Zuverlässigkeit und Störsicherheit eingehalten werden. Auch hier kommen meist mehrere C–Muster zum Einsatz.

Der Entwicklungsprozess von E/E–Komponenten wird von zahlreichen und aufwendigen Tests beim OEM und beim Zulieferer begleitet (s. Abb. 3.4). Der Komponententest setzt sich zusammen aus der Komponentenerprobung, der Softwareprüfung sowie dem Kommunikationstest (s. Abb. 3.5). Ziel der Komponentenerprobung ist, festzustellen, wie sich die einzelnen Komponenten unter verschiedenen Umwelteinflüssen verhalten. Dazu zählen neben Temperatur– und Vibrationstests u.a. Robustheitstests gegen elektrische Störungen. Die Softwareprüfung beinhaltet neben so genannten Software–Assessments, bei denen der Software–Entwicklungsprozess der verschiedenen Zulieferer unter die Lupe genommen wird, ebenfalls die funktionale Prüfung der einzelnen E/E–Komponenten mittels Hardware–in–the–Loop (HIL) Simulatoren (s. Kap. 3.5.1.2). Der Kommunikationstest überprüft, ob die einzelnen Komponenten später in einer vernetzten Umgebung miteinander kommunizieren können. Jede Komponente muss hierbei die definierten Busprotokolle einhalten. All diese Tests werden bereits beim Zulieferer unter Führung des OEM durchgeführt.

3.4.3 Integrationsphase

Liegen die ersten B–Muster vor, so beginnt die Integration der E/E–Komponenten zum Gesamtsystem unter der Führung des OEM. Der OEM hat hierbei die Rolle des Integrators, denn nur er kennt alle Abhängigkeiten im Gesamtsystem.

Abb. 3.4 zeigt bereits die der Integrationsphase zugeordneten Tests. Der entsprechende entwicklungsbegleitende Testprozess von E/E–Systemen bei der AUDI AG ist Gegenstand der nächsten beiden Abschnitte.

3.5 Testprozess der Automobilelektronik

Der Entwicklungsprozess von Automobilelektronik enthält einen separaten Testprozess. Die verschiedenen entwicklungsbegleitenden Tests bei der AUDI AG zeigt Abb. 3.5. Sie bauen

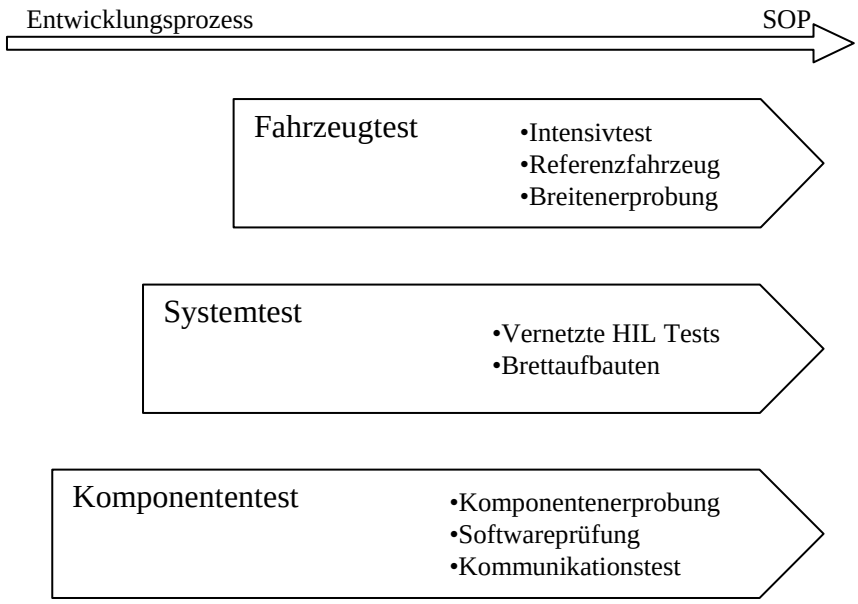


Abbildung 3.5: Entwicklungsbegleitende Tests (*Sterler*, 2005)

aufeinander auf und beginnen zu jeweils unterschiedlichen Zeitpunkten im Entwicklungsprozess. In der Regel enden alle entwicklungsbegleitenden Tests mit dem Serienanlauf bzw. dem Start of Production (SOP).

Während der Komponententest (vgl. Kap. 3.4.2) einzelne E/E−Komponenten im Fokus hat und meist beim Zulieferer stattfindet, betrachtet der Systemtest die Integration der einzelnen E/E−Komponenten zu einem übergeordneten System. Hierbei stehen neben Brettaufbauten wiederum (vernetzte) HIL Tests im Vordergrund. Brettaufbauten stellen die komplette E/E−Architektur von Fahrzeugsystemen dar. Die Systemumgebung wird hierbei jedoch nicht berücksichtigt. Auf diese Weise können verschiedene Fahrzeugvarianten ohne den Einsatz kostspieliger Prototypenfahrzeuge auf ihre korrekte Kommunikationsfähigkeit hin überprüft werden. Diese Überprüfung findet weitgehend manuell statt. Anders verhält es sich bei den Funktionsprüfungen mittels vernetzter HIL Tests. Hier ist eine hohe Automatisierung möglich, wodurch die Tests zu einem hohen Maß reproduzierbar sind. Die Verwendung von realzeitfähigen Umgebungsmodellen ermöglicht zudem die Überprüfung der vernetzten Funktionalität in einer simulierten aber realitätsnahen Umgebung. Auf diese Weise können Grenzbereiche bzw. Extremsituationen gefahrlos überprüft werden.

Abgerundet werden die bisher erwähnten Komponententests sowie Tests der vernetzten Systeme durch Tests am realen Fahrzeug. Dieser Fahrzeugtest beinhaltet u.a. Intensivtests an speziellen Prototypen in Zusammenarbeit mit den Zulieferern. Die Prototypen enthalten alle aktuellen Hard− und Softwarestände und stellen den jeweils aktuellen Entwicklungsstand dar.

Der Systemtest bzw. speziell vernetzte HIL Tests stellen das für diese Arbeit relevante Um-

feld dar. Für eine tiefere Betrachtung des Testsystems Hardware–in–the–Loop sowie des Bezugs zur Arbeit wird an dieser Stelle auf die nächsten zwei Abschnitte verwiesen.

3.5.1 Testsysteme

Hartmann (2001) definiert ein Testsystem als „ein rechnergestütztes Werkzeug, welches den automatisierten Test von elektronischen Komponenten und Systemen eines Kraftfahrzeugs gestattet.“ Diese Definition wird auch im Rahmen dieser Arbeit verwendet.

Im Folgenden werden zunächst Testsysteme klassifiziert und anschließend Hardware–in–the–Loop als das für diese Arbeit relevante Testsystem näher vorgestellt.

3.5.1.1 Klassifizierung von Testsystemen

In den verschiedenen Entwicklungsphasen sind die zu entwickelnden Systeme in der jeweils passenden Entwicklungsstufe verfügbar. Tab. 3.1 zeigt Kombinationen auf und klassifiziert die Testsysteme anhand des Sachverhalts, ob die Umwelt bzw. das SUT real vorhanden ist oder simuliert werden muss.

		Umwelt	
		Simuliert	Real
SUT	Simuliert	Software–in–the–Loop	Rapid Prototyping
	Real	Hardware–in–the–Loop	Onboard Test

Tabelle 3.1: Klassifikation von Testsystemen (*Hartmann*, 2001).

Software–in–the–Loop (SIL) Testsysteme zeichnen sich dadurch aus, dass weder die Umwelt noch das SUT real vorhanden ist. Dementsprechend wird SIL in frühen Phasen der Entwicklung eingesetzt.

Rapid Prototyping ermöglicht den Test eines simulierten Systems unter realen Umweltbedingungen.

Hardware–in–the–Loop (HIL) steht dagegen für den Test realer Komponenten in einer simulierten Umwelt. Üblicherweise wird HIL ab den ersten zur Verfügung stehenden B–Mustern bis zum Serienanlauf des Fahrzeugs eingesetzt.

Onboard Testsysteme Sobald erste Prototypenfahrzeuge zur Verfügung stehen, können Onboard Testsysteme eingesetzt werden.

Die Realitätsnähe der Testsysteme nimmt zu, je mehr reale Komponenten im Testsystem vorhanden sind. Mit steigender Realitätsnähe sinken jedoch die Manipulationsmöglichkeiten

des SUT und damit auch die erreichbare Testtiefe. Interne Abläufe des SUT sind z.B. nicht mehr ohne weiteres zugänglich.

Neben den oben vorgestellten Testsystemen existieren viele weitere Testsysteme. Model−in−the−Loop (MIL) Testsysteme können eingesetzt werden, falls z.B. die Funktion als ausführbares Modell vorliegt. Existiert bereits der später im Steuergerät verwendete Prozessor, kann auch ein Processor−in−the−Loop (PIL) Testsystem zum Einsatz kommen. *Bock et al.* (2005) binden für den Test neuer Fahrerassistenzsysteme sogar das ganze Fahrzeug in eine virtuelle Fahrzeugumgebung mittels des Testsystems Vehicle−in−the−Loop (VIL) ein.

Der Fokus dieser Arbeit liegt auf dem Bereich der HIL Testsysteme, die Gegenstand des nächsten Abschnitts sind.

3.5.1.2 Hardware−in−the−Loop Testsysteme

Hardware−in−the−Loop Testsysteme kommen ab der Verfügbarkeit der ersten B−Muster bis zum Serienanlauf und darüber hinaus zum Einsatz. Sowohl einzelne Komponenten als auch zu einem Verbund vernetzte Komponenten können als SUT in ein HIL Testsystem integriert werden.

Ein HIL Testsystem besteht aus drei wesentlichen Bausteinen:

- SUT
- Realzeit−Rechner
- Kontroll−PC

Das SUT (z.B. ein Steuergerät) liegt in Form von realer Hardware vor und ist in eine Realzeit−Simulationsumgebung auf einem Realzeit−Rechner eingebunden (s. Abb. 3.6). Diese virtuelle Steuergeräteumgebung besteht aus einer Aktor− und Sensorsimulation sowie aus einem Modell, das die Steuergeräteumgebung in Realzeit berechnet. Über die Sensorsimulation wird das SUT mit verschiedenen Eingangssignalen, die bestimmte Szenarien (z.B. Drücken der Warnblinktaste) darstellen, konfrontiert. Es reagiert auf diese Eingabe (vgl. Kap. 3.3.2), indem es entsprechende Ausgangssignale zur Ansteuerung bestimmter Aktoren ausgibt (z.B. Signale zur Ansteuerung der Blinker). Diese Ausgangssignale werden über die Aktorsimulation dem Umgebungsmodell zur Verfügung gestellt. Das Modell berechnet wiederum eine entsprechende Reaktion der Umgebung (z.B. Ansteuerung der Blinker). Diese Veränderung der virtuellen Umgebung nimmt das SUT wiederum durch die Sensorsimulation wahr und der Kreis schließt sich. Weiterer Bestandteil eines HIL Testsystems ist ein Kontroll−PC zur Steuerung, Visualisierung und Automatisierung der Simulation auf dem Realzeit−Rechner.

Wesentliche Vorteile von HIL Testsystemen sind:

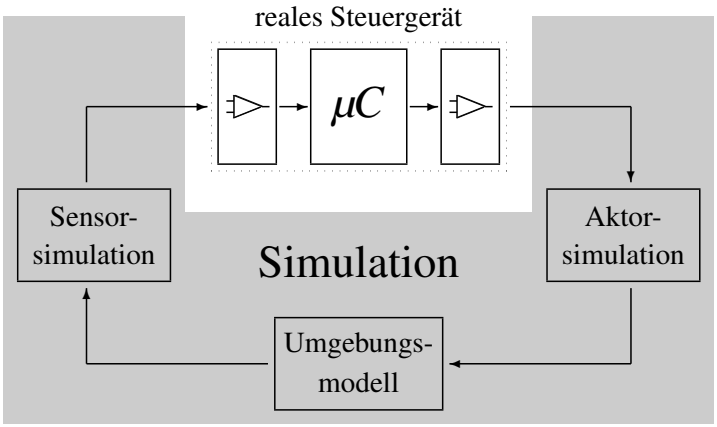


Abbildung 3.6: Hardware–in–the–Loop Simulation

- Reduzierung der Entwicklungszeit
- Kosteneinsparung
- Grenzbereichsabdeckung
- extreme Umgebungsbedingungen (Kälte, Hitze, ...)
- frühe Funktionserprobung
- gezielte Fehlersimulation
- Automatisierbarkeit
- Reproduzierbarkeit
- Auslastung

Die Entwicklungszeit kann durch Simultaneous Engineering (SE), die gleichzeitige Entwicklung von Steuergerät und Fahrzeug, deutlich reduziert werden. Noch bevor ein Fahrzeugprototyp existiert, können mittels HIL erste Steuer– und Regelungsfunktionen in einer virtuellen Fahrzeugumgebung getestet werden. Kostspielige Feldversuche werden durch Laborversuche ersetzt. Zudem können Versuche in Grenzbereichen bzw. Gefahrensituationen gefahrlos durchgeführt werden. Extreme Umgebungsbedingungen lassen sich durch Parametervariationen am simulierten Umgebungsmodell beliebig einstellen. Dadurch wird eine deutlich höhere Testabdeckung als beim Test mittels Prototypen erreicht. Hauptfokus von HIL Tests ist die Überprüfung der in Anforderungsdokumenten spezifizierten Funktionen auf korrektes logisches und zeitliches sowie zuverlässiges Verhalten. Hierfür ist die Auswahl einer geeigneten Stichprobe von Tests notwendig, anhand derer beurteilt werden kann, ob sich das System entsprechend seiner funktionalen Spezifikation korrekt oder fehlerhaft verhält.

Neben dieser Funktionserprobung wird das SUT jedoch auch gezielt mit automobilspezifischen Fehlerbildern (z.B. mit Unterbrechungen, Kurzschlüssen oder Vertauschungen von Leitungen aber auch mit Sensorausfällen, unplausiblen Sensorwerten, falschen CAN−Botschaften, ...) konfrontiert, um z.B. die korrekte Diagnosefunktionalität zu überprüfen. Weitere automobilspezifische Testinhalte sind z.B. Ruhestrommessungen, Tests bei verschiedenen Spannungsverläufen sowie die Überprüfung des Wake−up bzw. Sleep Verhaltens. Gezielte Stresssituationen (z.B. hohe Buslast), in denen das SUT an seine Belastungsgrenzen herangeführt wird, runden das Testspektrum ab.

Durch die Möglichkeit zur automatischen Testdurchführung können die HIL Simulatoren im Idealfall[3] rund um die Uhr ausgelastet werden. Alle durchgeführten Tests sind damit zu einem hohen Grad reproduzierbar.

HIL Testsysteme stellen somit eine kostengünstige und gefahrlose Möglichkeit dar, reale E/E−Komponenten bzw. die darin eingebettete Software in frühen Phasen des Entwicklungsprozesses auf ihre korrekte und zuverlässige Funktionsfähigkeit zu überprüfen.

3.6 Testerstellungsprozess an Hardware−in−the−Loop Testsystemen

Der Testerstellungsprozess zur Erprobung von durch eingebettete Software realisierten, verteilten Funktionen auf korrektes und zuverlässiges Verhalten mittels HIL Testsystemen stellt einen Teilprozess eines übergeordneten Testprozesses dar und unterteilt sich bei der AUDI AG hauptsächlich in die zwei Bereiche (*Kiffe*, 2006):

- Test−Anforderungs−Management
- Test−Ausführungs−Management

Die Sammlung von Testideen, die Analyse des SUT, die Erstellung informeller[4] Testspezifikationen sowie die Erteilung eines Testauftrags sind Bestandteile des Test−Anforderungs−Management. Die informellen Testspezifikationen stellen die Grundlage für die spätere Modellierung, Entwicklung und Durchführung von Tests dar. Nach Durchführung der Tests wird das Testergebnis in Form eines Testberichts festgehalten. Der vollständige Prozess sowie die dem Test−Ausführungs−Management zugeordnete konzerneinheitliche Testautomatisierung EXtended Automation Method (EXAM) inkl. aller damit verbundenen Begriffe wird in Anh. B näher vorgestellt.

[3] Wartungsarbeiten sowie Probeläufe neuer Tests verhindern eine 100%−ige Auslastung der HIL Testsysteme.

[4] Da die konkrete Form der Testspezifikation von sehr vielen Rahmenbedingungen abhängen kann und deren Detaillierungsgrad sehr unterschiedlich ist, wird sie im skizzierten Testerstellungsprozess als informelle Testspezifikation bezeichnet (*Kiffe*, 2006).

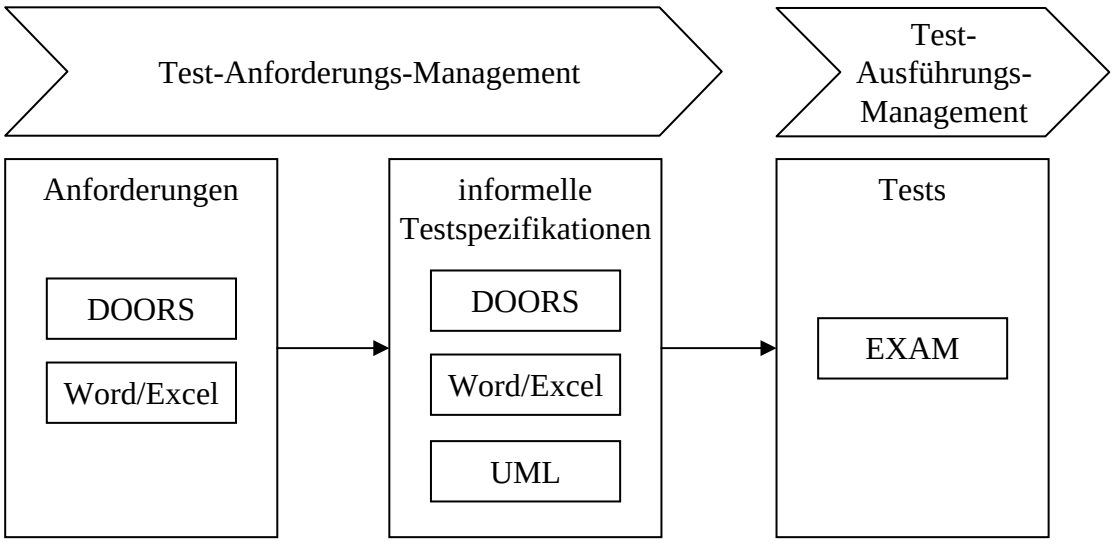


Abbildung 3.7: EXAM−Testerstellungsprozess

Zurzeit wird das Test−Anforderungs−Management von Tools wie z.B. DOORS, Word und Excel zur Spezifikation der Anforderungen unterstützt (s. Abb. 3.7). Die UML[5] bzw. das ARTiSAN Studio, als CASE−Tool[6] der Firma *ARTiSAN Software Tools, Ltd.*, stehen neben den bereits erwähnten Tools zusätzlich zur Spezifikation von Tests zur Verfügung.

Größtenteils werden die Testfälle, wie in der Praxis üblich (vgl. Kap. 2.4), Ad−hoc oder mittels der „Technik“ des „Fehler erraten“, ermittelt. Eine systematische Testfallermittlung ist selten. Zudem weisen die informellen Testspezifikationen einen geringen Formalisierungsgrad auf.

3.6.1 Problemstellung

Das bei dem soeben skizzierten Testprozess aufgeführte Defizit der unsystematischen Ermittlung von Testfällen verbunden mit dem Einsatz von informellen Testspezifikationen bringt folgende Probleme mit sich, die eine umfassende Prüfung der immer komplexer werdenden Systeme zunehmend schwieriger gestalten:

- Testqualität vom Testersteller abhängig
- Testinhalte schwer diskutierbar
- Testziele und Testendekriterien schwer festzulegen

[5] Die Unified Modeling Language (UML) stellt einen Standard der *Object Management Group* (OMG) zur grafischen Notation für die objektorientierte Analyse, das Design und die Entwicklung von Softwaresystemen dar (*Schläfer*, 2004).

[6] CASE ist die Abkürzung für Computer−Aided Software/Systems Engineering.

- Testabdeckung nicht messbar
- Strukturierung von Testinhalten schwierig
- Identifikation redundanter Testinhalte schwierig

Zum einen ist die Qualität der ermittelten Tests sehr von der Erfahrung sowie vom Qualifikationsniveau des Testerstellers anhängig. Die spezifizierten Testinhalte sind meist nur dem Testersteller selbst bekannt und können schwer mit anderen Personen diskutiert werden. Zum anderen sind Aussagen bzgl. der Testabdeckung sowie die Definition von messbaren Testzielen verbunden mit geeigneten Testendekriterien aufgrund des geringen Formalisierungsgrades der textuellen Testspezifikationen kaum möglich. Eine sinnvolle Strukturierung von ähnlichen Testinhalten bereitet ebenso Schwierigkeiten wie die Identifikation redundanter Testinhalte.

3.7 Fazit

Die Ermittlung geeigneter Tests für die soeben dargestellte Überprüfung der Fahrzeugsysteme auf korrekte und zuverlässige Funktionalität ist aufgrund der hohen Systemkomplexität eine höchst anspruchsvolle Aufgabe. Sie stellt die wichtigste Testaktivität (vgl. Kap. 2.4) dar, denn allein für die ausgewählte Stichprobe von Testfällen lässt sich eine Aussage bzgl. des korrekten oder fehlerhaften Verhaltens angeben. Für alle unberücksichtigten Fälle kann keine Aussage getroffen werden. Da ein Test aller möglichen Szenarien nicht durchführbar ist, bedarf es einer gewissen Systematik bei der Auswahl der Stichprobe.

Leider wird die Auswahl der Stichprobe von Testfällen zumeist Ad–hoc oder auf der Erfahrung der Testersteller basierend, d.h. ohne methodische Unterstützung, durchgeführt. Die ausgewählten Testfälle werden in so genannten informellen Testspezifikationen festgehalten. Damit ist die Qualität der Tests hauptsächlich von der Erfahrung bzw. dem Qualifikationsniveau des Testerstellers abhängig. Der geringe Formalisierungsgrad der Testspezifikationen erlaubt außerdem keine Bewertung des darin enthaltenen Testumfangs. Eine sinnvolle Strukturierung von Testinhalten ist ebenso schwierig wie die Identifikation von redundanten Testinhalten sowie die Definition von angemessenen Testendekriterien.

Am Markt existieren zahlreiche Testverfahren, die zudem durch geeignete Werkzeuge unterstützt werden. Bisher konnten sich diese Verfahren in der Praxis jedoch nicht etablieren (*Armbrust et al.*, 2004b).

Ziel dieser Arbeit ist daher, wie bereits in Kap. 1.2 aufgeführt, die Identifikation geeigneter, systematischer Testverfahren sowie deren Integration in den bestehenden Testprozess zur Behebung der im vorangegangenen Abschnitt geschilderten Probleme. Deshalb ist ein entsprechendes Konzept zur Integration der ausgewählten, systematischen Testverfahren aufzustellen und in einem Praxistest auf Praxistauglichkeit zu validieren. Das validierte Konzept ist umzusetzen und eine geeignete Werkzeugunterstützung bzw. –anbindung bereitzustellen.

4 Auswahl existierender Testverfahren

In diesem Kapitel werden existierende Testverfahren im Hinblick auf den Einsatz im zuvor beschriebenen Testerstellungsprozess bewertet und die vielversprechendsten für das weitere Vorgehen ausgewählt.

4.1 Klassifikation analytischer Testverfahren

An dieser Stelle wird überblicksartig auf dynamische Testverfahren (s. Anh. A.1) als das für diese Arbeit relevante Themengebiet eingegangen. Sie stellen ein Teilgebiet der analytischen Testverfahren dar. Eine detaillierte Klassifikation analytischer Testverfahren findet in Anh. A statt.

Dynamische Testverfahren führen das SUT mit Testdaten in seiner realen Umgebung aus. Daher ist eine Aussage über die korrekte oder fehlerhafte Funktion des SUT nur für die ausgewählte Stichprobe von Testfällen möglich. Die korrekte Funktion des SUT für alle Eingaben kann nur durch Test aller möglichen Eingaben sichergestellt werden. Dieser vollständige Test wird als erschöpfender Test bezeichnet. Seine Durchführung ist für reale Systeme in der Regel nicht möglich, da unendlich viele, unterschiedliche Programmpfade existieren (s. Anh. A.2). Aufgrund dieser Schwäche des Tests bedarf es einer Systematik bei der Auswahl der Stichprobe von Testfällen. Diese Aktivität der Testfallermittlung stellt die wichtigste Testaktivität dar und beeinflusst die Qualität des Tests in besonderer Weise (vgl. Kap. 2.4).

Vor diesem Hintergrund werden im nächsten Abschnitt existierende dynamische Testverfahren im Hinblick auf deren Tauglichkeit zur Lösung der in Kap. 3.6.1 aufgeführten Probleme bewertet und ausgewählt.

4.2 Auswahl dynamischer Testverfahren

Aufgrund der zahlreichen Einsatzgebiete, verschiedener Testziele und unterschiedlichster Teststrategien existiert nicht das allgemeingültige Testverfahren für eingebettete Systeme und die darin eingebettete Software (*Broekman & Notenboom*, 2003). Jede Problemstellung erfordert eine spezielle Herangehensweise verbunden mit entsprechend speziellen Testverfahren. Zu diesem Schluss kommt auch *Grimm* (1995), der in seiner Arbeit eine effektive

Teststrategie vorschlägt, die eine Kombination von verschiedenen dynamischen Testverfahren, strukturorientierten und funktionalen Testverfahren, darstellt.

Im Folgenden erfolgt daher eine Bewertung verschiedener dynamischer Testverfahren im Hinblick auf den Einsatz im beschriebenen Testprozess von Fahrzeugkomponenten sowie zur Lösung der in Kap. 3.6.1 aufgeführten Probleme. Dabei werden folgende drei Unterkategorien dynamischer Testverfahren betrachtet:

- strukturorientierte Testverfahren
- funktionale Testverfahren
- weitere Testverfahren

Jeder Kategorie ist im Folgenden ein Abschnitt gewidmet.

4.2.1 Strukturorientierte Testverfahren

Strukturorientierte Testverfahren (s. Anh. A.2) leiten die Testfälle aus der inneren Struktur des SUT (Software) ab. Sie sind daher nicht in der Lage, fehlende Funktionen des Programms aufzudecken. Allein der Vergleich der Implementierung mit der Spezifikation durch den funktionalen Test erkennt derartige Fehler im SUT. Die in Anh. A.2 definierten Überdeckungsmaße erlauben zwar eine sehr gute Bewertung der erreichten Testabdeckung, geben aber keinen Hinweis darauf, in welchem Umfang die Funktionalität des SUT getestet wurde. Eine hundertprozentige Zweigüberdeckung sagt z.B. wenig über die Qualität des Tests aus, sofern in der funktionalen Spezifikation spezifizierte Funktionen nicht im SUT realisiert sind. Lediglich die Konsistenz der Realisierung kann durch strukturorientierte Testverfahren überprüft werden.

Für den Test von E/E−Komponenten spielen strukturorientierte Testverfahren beim OEM eine untergeordnete Rolle. Dies ist u.a. auf die Tatsache zurückzuführen, dass die einzelnen Komponenten nicht vom OEM sondern von verschiedenen Zulieferern entwickelt werden. Die Zulieferer geben ihr Know−how nicht preis und liefern eine Black−Box, die sie anhand der funktionalen Spezifikation des OEM entwickelt haben. Die innere Struktur der Komponenten bleibt dem OEM damit verborgen. Deshalb werden strukturorientierte Testverfahren im weiteren Verlauf der Arbeit nicht weiter betrachtet.

4.2.2 Funktionale Testverfahren

Funktionale Testverfahren (s. Anh. A.3) leiten die Testfälle, im Gegensatz zu den strukturorientierten Testverfahren, aus der funktionalen Spezifikation des SUT ab. Die Qualität der funktionalen Spezifikation ist daher ein wesentlicher Faktor für die Qualität der daraus

abgeleiteten Testfälle. Das SUT ist für den Testersteller mit Ausnahme der funktionalen Spezifikation eine Black−Box.

Die Begründung, ein Programm gegen seine funktionale Spezifikation zu testen, liegt darin, dass strukturorientierte Testverfahren nicht in der Lage sind, fehlende Funktionen des Programms aufzudecken. Funktionale Testverfahren sind dafür nicht in der Lage, die konkrete Implementierung geeignet zu berücksichtigen. Es werden z.B. keine Informationen darüber geliefert, ob alle implementierten Funktionen des Programms benötigt werden oder ob Datenobjekte manipuliert werden, die keinen Einfluss auf das Ein− bzw. Ausgabeverhalten des SUT haben. Dafür lassen sich fehlende und fehlerhafte Funktionen identifizieren.

Wie im Abschnitt zuvor beschrieben, entwickeln Zulieferer die E/E−Komponenten anhand der vom OEM erstellten funktionalen Spezifikation. Beim OEM werden die Testfälle ebenfalls aus der funktionalen Spezifikation abgeleitet, weshalb in diesem Zusammenhang auch vom spezifikationsorientierten bzw. anforderungsbasierten Testen gesprochen wird. Existierende funktionale Testverfahren (s. Anh. A.3) kommen dabei trotz ihrer Prädestiniertheit eher selten im beschriebenen Testprozess von in Fahrzeugsysteme eingebetteter Software zum Einsatz (s. Kap. 4.3).

Folgende funktionale Testverfahren werden im Hinblick auf den Einsatz im skizzierten Testerstellungsprozess betrachtet:

Funktionale Äquivalenzklassenbildung Durch die funktionale Äquivalenzklassenbildung (s. Anh. A.3.1) kann die Testfallanzahl erheblich reduziert werden. Daher bildet die Äquivalenzklassenbildung die Grundlage für viele der vorgestellten funktionalen Testverfahren.

Ursache−Wirkungs−Analyse Die Ursache−Wirkungs−Analyse (s. Anh. A.3.2) ist zwar sehr mächtig, da Inkonsistenzen oder Unvollständigkeiten in der Spezifikation durch Bildung des Ursache−Wirkungs−Graphen entdeckt werden können. Bei der Erstellung des Ursache−Wirkungs−Graphen wird jedoch eine komplexe Struktur (funktionale Spezifikation) durch eine andere (Ursache−Wirkungs−Graph) ersetzt. Dieser Graph wird schnell komplex, unübersichtlich und ist im fortgeschrittenen Stadium nur sehr schwer änderbar (*Ostrand & Balcer*, 1988). Erfahrene Anwender sind daher unabdingbar.

Category−Partition−Method Auch die Category−Partition−Method (s. Anh. A.3.3) ist ähnlich wie die Ursache−Wirkungs−Analyse eine mächtige Methode. Jedoch bedarf es auch hier eines erfahrenen Anwenders, der zudem Kenntnisse in der Test Specification Language benötigt (*Simmes*, 1997).

Classification−Tree Method Als systematisches Testverfahren enthält die Classification−Tree Method (s. Anh. A.3.4) prinzipiell die ganze Mächtigkeit der Äquivalenzklassenbildung und Category−Partition−Method, bietet jedoch zusätzlich eine grafi-

sche Notation zur Erstellung des Klassifikationsbaumes. Die systematische Vorgehensweise ermöglicht eine strukturierte Partitionierung des Eingabedatenraums, wodurch die Übersichtlichkeit über den Eingabedatenraum gefördert wird. Außerdem definiert sie Überdeckungsmaße, anhand derer der aktuelle Teststand gemessen und bewertet werden kann. Die CTM ist nach *Chen & Poon* (1998) leicht zu erlernen und in Forschung und Industrie relativ weit verbreitet. Zahlreiche Anwendungsbeispiele, auch im Bereich der Automobilindustrie, belegen den erfolgreichen Einsatz der Classification−Tree Method. Zudem existieren Werkzeuge, welche die Methode bei der Erstellung der Klassifikationsbäume grafisch unterstützen (s. Kap. 5.4). Aus diesen Gründen wird für das weitere Vorgehen die Classification−Tree Method zur Systematisierung der Testfallermittlung ausgewählt. Kap. 5 widmet sich deshalb ausschließlich der Classification−Tree Method sowie ihrer Erweiterung, der Classification−Tree Method for Embedded Systems.

Simmes (1997) führt in seiner Arbeit eine Bewertung funktionaler Testverfahren nach definierten Entscheidungskriterien in einer Nutzwertanalyse hinsichtlich des Systemtests von Kfz−Steuergeräten durch. „Die gewählten Entscheidungskriterien berücksichtigen die Anforderungen, die sich aus der Anwendung der Testverfahren in der Praxis der industriellen Großserienentwicklung ergeben.“ Dabei erhält die Classification−Tree Method in den Kriterien „Systematik“, „Handhabung“, „Erlernbarkeit“ und „Darstellung“ den höchsten Erfüllungsgrad von zehn Punkten. Die Kriterien „Leistung“ und „Aufwand“ werden mit acht von zehn Punkten bewertet. Diese sehr gute Bewertung bekräftigt die getroffene Entscheidung zugunsten der Classification−Tree Method.

4.2.3 Weitere Testverfahren

Einige Testverfahren (z.B. Zufallstest, Grenzwertanalyse und Test spezieller Werte) verfolgen völlig unterschiedliche Zielsetzungen und werden daher unter dem Oberbegriff „weitere Testverfahren“ (s. Anh. A.4) zusammengefasst.

Das Fehlen einer deterministischen Strategie beim Zufallstest kann sowohl als Stärke aber auch als Schwäche aufgefasst werden. Testersteller neigen dazu, bestimmte Testfälle zu erzeugen, die auch bei der Implementierung des Programms als nahe liegend betrachtet worden sind und für die sich das Programm folglich gutartig verhält. Die Regellosigkeit der Testfallermittlung bzw. die Gleichbehandlung aller Eingabedaten ohne Beachtung menschlicher Präferenzen bietet aus diesem Grunde eine Möglichkeit zur Entdeckung von Fehlern, an die der Testersteller nicht gedacht hat. Diese Unvorhersagbarkeit des Zufallstests bezeichnen jedoch *Armbrust et al.* (2004a) als Nachteil: „Da es sich um zufällig erzeugte Daten handelt, lässt sich prinzipiell nichts Genaues über Abdeckung u.ä. aussagen. Statistische Aussagen sind selbstverständlich möglich, diese sind jedoch immer mit einer gewissen Unsicherheit

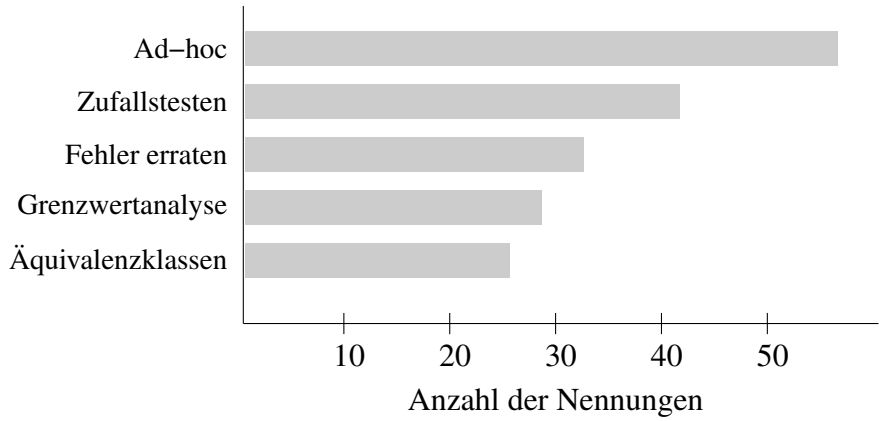


Abbildung 4.1: Auszug aus in der Praxis eingesetzten Testtechniken (*Armbrust et al.*, 2004b)

behaftet.“ Zudem gestaltet sich die Aktivität der Testauswertung (vgl. Kap 2.4) sehr schwierig. Die Existenz eines Orakels wird vorausgesetzt, das für jede erdenkliche Eingabe die korrekte Ausgabe bzw. das korrekte Verhalten kennt.

Alle weiteren Testverfahren eignen sich als ergänzende bzw. optimierende Verfahren und sollten deshalb prinzipiell berücksichtigt werden.

4.3 Einsatz der Testverfahren in der Praxis

Leider wird die in Kap. 2.4 vorgestellte Testaktivität der Testfallermittlung in der Praxis häufig unsystematisch, d.h. ohne den Einsatz geeigneter Methoden, durchgeführt. Dies belegt u.a. die von *Armbrust et al.* (2004b) durchgeführte Online−Umfrage. Abb. 4.1 zeigt einen Auszug der Ergebnisse dieser Umfrage bei der u.a. die in der Praxis eingesetzten Testtechniken von 94 Unternehmen angegeben wurden. Spitzenreiter mit über 50 Nennungen ist dabei die Ad−hoc Testfallermittlung, gefolgt vom Zufallstest. Platz drei nimmt das Erraten von Fehlern als eingesetzte „Technik“ ein, und erst ab Platz vier erscheinen die Grenzwertanalyse und die funktionale Äquivalenzklassenbildung als erste systematische Testverfahren.

An dieser Stelle besteht daher ein enormes Verbesserungspotenzial, das im Rahmen dieser Arbeit, durch die Integration der beiden systematischen Testverfahren (CTM und CTM/ES) in den bestehenden Testprozess, ausgeschöpft werden soll.

5 Detaillierte Betrachtung der ausgewählten Testverfahren

Klassifikationsbäume sind seit vielen Jahren in verschiedenen Bereichen wie z.B. der Philosophie, der Medizin und vielen anderen unter dem Namen Classification And Regression Trees (CART) im Einsatz[1]. Im Bereich des Softwaretests werden sie seit 1993 eingesetzt (*Grimm & Grochtmann*, 1993). Seit dieser Einführung beschäftigen sich einige weitere Gruppen mit deren Weiterentwicklung. Zu nennen sind hier zum einen *Chen & Poon* (1997). Zum anderen existieren zahlreiche Anwendungen in der Industrie (z.B. *Rumprecht* (1994), *Simmes* (1997), *Conrad* (2004)).

In den folgenden Abschnitten wird zunächst die Klassifikationsbaum-Methode (engl. Classification−Tree Method (CTM)) und anschließend die Erweiterung zur Klassifikationsbaum−Methode für eingebettete Systeme (engl. Classification−Tree Method for Embedded Systems (CTM/ES)) vorgestellt. Dabei werden innerhalb dieser Arbeit meistens die englischen Namen bzw. Abkürzungen der beiden Methoden verwendet. Zudem werden weitere Erweiterungen der CTM mit dem zur Verfügung stehenden grafischen Werkzeug, dem Classification−Tree Editor eXtended Logics (CTE XL), aufgezeigt.

5.1 Classification−Tree Method (CTM)

„Die grundsätzliche Idee der Klassifikationsbaum-Methode ist es, zuerst die Menge der möglichen Eingaben für das Testobjekt getrennt auf verschiedene Weisen unter jeweils einem geeigneten Gesichtspunkt zu zerlegen, um dann durch Kombination dieser Zerlegungen zu Testfällen zu kommen“ (*Grimm*, 1995).

Die Classification−Tree Method wird durch ein schrittweises Vorgehen systematisiert und durch eine einfache und anschauliche, grafische Notation unterstützt. Das schrittweise Vorgehen erfolgt durch die folgenden drei Schritte (*Grimm*, 1995):

1. Definition des SUT
2. Klassifikation des Eingabedatenraums

[1] Auch im Automobilbereich werden CART bereits eingesetzt, um z.B. Signale von Airbagsensoren zu analysieren und darauf aufbauend die Auslösealgorithmen des Airbags zu entwerfen (*Botsch & Nossek*, 2007).

a) Definition von testrelevanten Gesichtspunkten

b) vollständige, disjunkte Zerlegung in Äquivalenzklassen[2]

c) Iteration

3. Testfallermittlung

Im Folgenden wird jeder dieser Schritte einzeln betrachtet. Ein konkretes Anwendungsbeispiel der CTM anhand einer vereinfachten Funktion Richtungsblinken findet sich in Kap. 6.2.2.

5.1.1 Die Schritte der Classification−Tree Method

5.1.1.1 Schritt 1: Definition des SUT

Voraussetzung für die Definition bzw. Identifikation der einzelnen Testobjekte bzw. SUT sind detailliere Spezifikationsdokumente. Dabei kann sowohl die Struktur als auch der Inhalt der Dokumente relevante Informationen für die Testobjektidentifikation liefern. Das SUT stellt die Wurzel des Klassifikationsbaumes dar und wird durch ein abgerundetes Rechteck mit dickem Rahmen im Kopf des Klassifikationsbaum−Diagramms repräsentiert (s. Abb. 5.1).

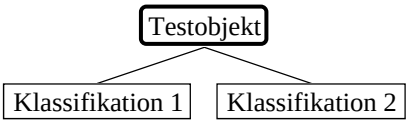


Abbildung 5.1: Klassifikationen im Klassifikationsbaum

„Eine Zerlegung des Systems in geeignete SUT ist aufgrund der Komplexität von Realzeit−Systemen sehr wichtig und stellt eine anspruchsvolle Aufgabe für den Prüfer dar“ (*Rumprecht*, 1994). Diese Reduzierung der Komplexität des übergeordneten Testobjekts ist ein zentraler Punkt, der bei der Anwendung der Methode geeignet berücksichtigt werden muss. Hierfür kann z.B. die später vorgestellte Anwendungspragmatik (s. Kap. 5.3.1) zum Einsatz kommen. Das korrekte Zusammenspiel der einzelnen, bereits getesteten, Teile wird nach der Einzelprüfung im Integrationstest durch geeignete Integrationsstrategien (vgl. Kap. 2.2.6) überprüft.

5.1.1.2 Schritt 2: Klassifikation des Eingabedatenraums

Nach der Definition des SUT erfolgt die Klassifikation seines gesamten Eingabedatenraums.

[2]Der Begriff der Klasse gehört ebenfalls zum Sprachumfang der bereits in Kap. 3.6 erwähnten UML. Im weiteren Verlauf der Arbeit werden Äquivalenzklassen aus Gründen der Übersicht manchmal auch als Klassen bezeichnet. Die Zuordnung des Begriffs zur CTM bzw. zur UML ergibt sich aus dem jeweiligen Kontext.

Schritt 2a: Definition von testrelevanten Gesichtspunkten Hierbei identifiziert der Testersteller zunächst testrelevante Gesichtspunkte des SUT. Jeder Gesichtspunkt soll möglichst eine begrenzte und damit übersichtliche Unterscheidung der möglichen Eingaben für das SUT erlauben. Die vom Testersteller identifizierten Gesichtspunkte werden Klassifikationen genannt und werden auf der Ebene unterhalb der Wurzel des Klassifikationsbaumes eingesetzt. Klassifikationen werden durch umrandete Rechtecke dargestellt. Abb. 5.1 zeigt die Klassifikationen `Klassifikation 1`[3] und `Klassifikation 2` im Klassifikationsbaum.

Schritt 2b: vollständige, disjunkte Zerlegung in Äquivalenzklassen Der nächste Teilschritt sieht für jede eingeführte Klassifikation eine weitere Unterteilung der möglichen Eingabewerte in so genannte Äquivalenzklassen vor. Die Äquivalenzklassenbildung erfolgt hierbei mit dem Ziel, dass ein Test mit einem beliebigen Wert aus einer Äquivalenzklasse äquivalent ist zu einem Test mit irgendeinem anderen Wert aus dieser Äquivalenzklasse. Grundlage hierfür bildet die funktionale Äquivalenzklassenbildung (s. Anh. A.3.1).

Die Aufteilung unterliegt zwei Kriterien. Zum einen soll es sich um eine vollständige Aufteilung der Klassifikation handeln, zum anderen sollen die gebildeten Äquivalenzklassen disjunkt sein. Daraus resultiert die eindeutige Zuordnung eines Wertes zu genau einer Äquivalenzklasse.

In Abb. 5.2 wird die Klassifikation `Klassifikation 1` beispielhaft in `Klasse 1.1` und `Klasse 1.2` vollständig und disjunkt zerlegt, während sich Klassifikation `Klassifikation 2` in insgesamt drei disjunkte Äquivalenzklassen vollständig aufteilt:

- `Klasse 2.1`
- `Klasse 2.2`
- `Klasse 2.3`

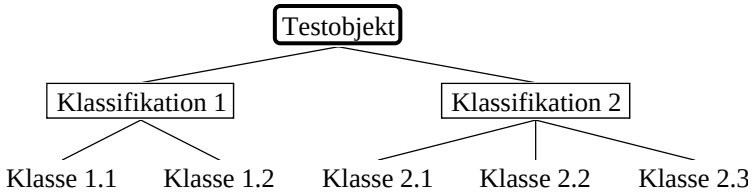


Abbildung 5.2: Klassifikationen und Äquivalenzklassen im Klassifikationsbaum

Die Grenzwertanalyse (vgl. Anh. A.4.2.1) kann hierbei angewendet werden, indem z.B. für jeden bekannten Grenzwert ein Bereich unterhalb des Grenzwertes, der Grenzwert selbst

[3] Durch diese Textformatierung sind im weiteren Verlauf der Arbeit Elemente aus referenzierten Abbildungen gekennzeichnet.

sowie der Bereich oberhalb des Grenzwertes als eigene Äquivalenzklasse angelegt werden. Repräsentanten für jede Äquivalenzklasse können manuell vorgegeben oder aber durch geeignete Algorithmen zufällig ausgewählt werden.

Schritt 2c: Iteration Manchmal kann es sinnvoll bzw. nötig sein, einige Äquivalenzklassen weiter zu unterteilen. Dafür wendet man die Klassifikation iterativ auf Äquivalenzklassen an und erhält eine immer tiefer verzweigte Baumstruktur. Neue Gesichtspunkte bzw. Klassifikationen erweitern den Baum dabei um weitere Ebenen. In Abb. 5.4 ist diese Möglichkeit im Klassifikationsbaum anhand der Äquivalenzklasse `Klasse 2.1` beispielhaft dargestellt.

5.1.1.3 Schritt 3: Testfallermittlung

Nachdem alle Details des Eingabedatenraums im Klassifikationsbaum abgebildet wurden, besteht der nächste Schritt der Classification–Tree Method in der Testfallermittlung. Testfälle entstehen durch die Kombination von Blättern des Baumes, also den nicht weiter klassifizierten Äquivalenzklassen. Blatt und nicht weiter klassifizierte Äquivalenzklasse werden in diesem Zusammenhang synonym verwendet.

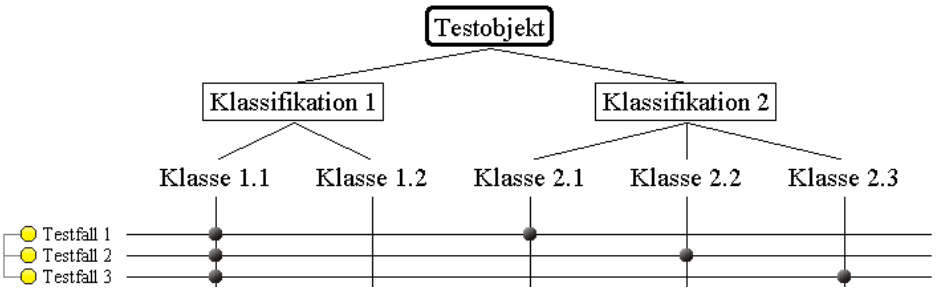


Abbildung 5.3: Klassifikationsbaum mit Kombinationstabelle und drei (logischen) Testfällen

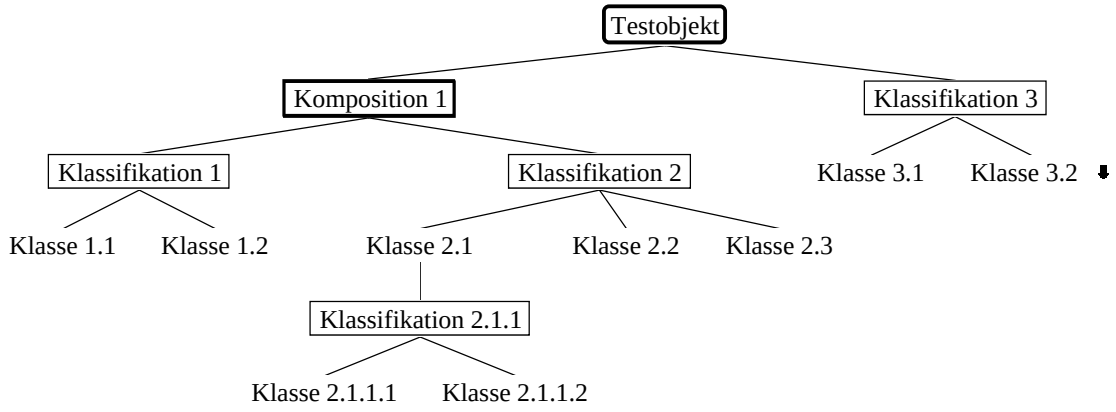
Als Unterstützung zur Testfallermittlung wird eine Kombinationstabelle[4] verwendet. Den Kopf dieser Tabelle bildet der Klassifikationsbaum, während die Blätter des Baumes die Spalten bilden. Eine Zeile in der Kombinationstabelle entspricht jeweils einem Testfall. Durch Markieren der Kreuzungspunkte von Spalten mit Zeilen werden den einzelnen Testfällen die entsprechenden Blätter zugeordnet. In Abb. 5.3 ist ein Klassifikationsbaum mit Kombinationstabelle und drei verschiedenen Testfällen dargestellt. Jeder der drei Testfälle berücksichtigt verschiedene Eingabekonstellationen an das Testobjekt.

[4]Einige Autoren verwenden statt des Begriffs Kombinationstabelle auch den Begriff Entscheidungs– bzw. Testfalltabelle.

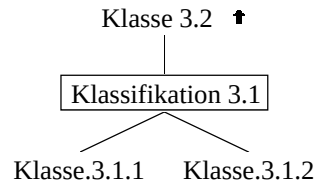
5.1.2 Strukturierungsmöglichkeiten

Zur Strukturierung eines Klassifikationsbaumes existiert das Klassifikationsbaumelement Komposition (siehe `Komposition 1` in Abb. 5.4(a)). Mittels der Komposition (Rechteck mit dickem Rahmen) können inhaltlich zusammengehörende Klassifikationen gruppiert werden.

Weitere Elemente zur Strukturierung des Klassifikationsbaumes stellen die Klassifikationsverfeinerung sowie die Klassenverfeinerung dar. Beide Elemente sind durch einen zusätzlichen Pfeil rechts neben dem Namen der Klassifikation bzw. Äquivalenzklasse gekennzeichnet. In Abb. 5.4(a) ist eine Klassenverfeinerung der Äquivalenzklasse `Klasse 3.2` beispielhaft enthalten. Die mittels der Klassenverfeinerung ausgeblendeten Inhalte zeigt Abb. 5.4(b).



(a) Klassifikationsbaum mit Komposition und Klassenverfeinerung



(b) Inhalt der Klassenverfeinerung

Abbildung 5.4: Klassifikationsbaum mit dem Strukturierungssymbol Komposition, der iterativen Anwendung von Klassifikationen auf Klassen sowie einer Klassenverfeinerung

5.1.3 Überdeckungsmaße

Ein Problem beim Testen besteht darin, festzulegen, wann ausreichend getestet wurde bzw. den beim Test erzielten Testumfang in irgendeiner Weise quantitativ zu beurteilen (vgl. Kap. 2.2.5). Strukturorientierte Verfahren liefern zwar Überdeckungsmaße, sie liefern jedoch keine Aussage darüber, in welchem Umfang die Funktionalität des SUT getestet wurde. So

sagt z.B. eine hundertprozentige Zweigüberdeckung (s. Anh. A.2) wenig über die Qualität des Tests aus, sofern in der funktionalen Spezifikation spezifizierte Funktionen nicht im SUT realisiert sind.

„Die Klassifikationsbaum-Methode bietet nun erstmals Ansatzpunkte für die Definition von Maßen für den funktionalen Test. Da sie in den einzelnen Schritten der Testfallermittlung zwar ein systematisches Vorgehen gewährleistet, der Prozess aber nicht absolut deterministisch verläuft, hängt die Aussagekraft der Maße natürlich von den gewählten Gesichtspunkten und dem Detaillierungsgrad der vorgenommenen Klasseneinteilung ab“ (*Grimm*, 1995). Die Aussagekraft von Maßen eines „schlecht“ klassifizierten Baumes ist eben nur so gut wie die Klassifizierung selbst. Zur Beurteilung der Komplexität des Testproblems und des damit verbundenen Überdeckungsgrades können diese Maße jedoch sehr aufschlussreich sein.

Als Minimalkriterium gibt *Grimm* (1995) die mindestens einmalige Berücksichtigung jeder als testrelevant in den Klassifikationsbaum aufgenommenen Äquivalenzklasse an. Das Überdeckungsmaß

$$CTC_{min} = \frac{actclass}{maxclass} \tag{5.1}$$

(Classification−Tree Coverage) gibt das Verhältnis zwischen der Anzahl beim Test berücksichtigter (actclass) und der insgesamt definierten Blätter (maxclass) an. Bei einem Test sollte prinzipiell eine vollständige Blattüberdeckung ($CTC_{min} = 1$), d.h. die Berücksichtigung jedes eingeführten Blattes in mindestens einem Testfall, angestrebt werden. Schließlich hat der Testersteller beim Design des Klassifikationsbaumes genau diese Baumstruktur gewählt und damit jedes einzelne Blatt des Baumes für testrelevant befunden.

Ein Maximalkriterium ist theoretisch durch die Berücksichtigung aller möglichen Blätterkombinationen gegeben. Dabei erwähnt *Grimm* (1995), dass es prinzipiell Blätter geben kann, die logisch nicht miteinander vereinbar sind und die somit auch nicht zu Testfällen kombiniert werden dürfen. Außerdem kann es vorkommen, dass Blätter zwar logisch vereinbar, aber von Spezifikationsseite nicht möglich sind.

Das Maximalkriterium erlaubt die Definition eines Überdeckungsmaßes, anhand dessen die Testabdeckung bewertet werden kann. Das Maß

$$CTC_{max} = \frac{actfall}{maxfall} \tag{5.2}$$

gibt das Verhältnis der Anzahl der zum Test berücksichtigten (actfall) zu der Anzahl theoretisch möglicher Blätterkombinationen (maxfall) an. Wird das Maß nicht auf alle theoretisch möglichen (maxfall), sondern nur auf die logisch möglichen (logfall) beziehungsweise nur auf die durch die Spezifikation möglichen Blätterkombinationen (spezfall) bezogen, so gilt:

$$CTC_{max} = \frac{actfall}{maxfall} \quad \leq \quad CTC'_{max} = \frac{actfall}{logfall} \quad \leq \quad CTC''_{max} = \frac{actfall}{spezfall} \tag{5.3}$$

Welcher CTC_{max}−Überdeckungsgrad jedoch erreicht werden sollte, kann nicht pauschal für alle Problemstellungen angegeben sondern muss von Fall zu Fall entschieden werden. „Die Erfahrung hat gezeigt, daß bei realen Tests die Anzahl der testrelevanten Kombinationen eher in der Größenordnung des Minimal− als des Maximalfalls liegt“ (*Grimm*, 1995). Nur Systeme mit sehr hohen Sicherheitsanforderungen und hoher Abhängigkeit der verschiedenen Klassifikationen rechtfertigen die Forderung nach $CTC''_{max} = 1$.

Einen Kompromiss aus CTC_{min} und CTC_{max} stellt das so genannte n−weise Kombinationskriterium CTC_n dar. Dabei muss jede mögliche Kombination von n Äquivalenzklassen in mindestens einem Testschritt berücksichtigt werden. Als praktikables Maß nennen *Lamberg et al.* (2004) CTC_2 also die vollständige paarweise Kombination von Blattklassen.

Büchner (2002) gibt als Daumenregel für eine ausreichende Testabdeckung die Summe aller vorhandenen Blattklassen unter Berücksichtigung des Minimalkriteriums als Größenordnung für die Anzahl der notwendigen Testfälle an.

Im weiteren Verlauf der vorliegenden Arbeit werden lediglich die Maße CTC_{min} und CTC_{max} sowie die Daumenregel von *Büchner* (2002) betrachtet.

5.2 Classification−Tree Method for Embedded Systems (CTM/ES)

Conrad (2004) erweitert die Classification−Tree Method zur Classification−Tree Method for Embedded Systems (CTM/ES) und ergänzt damit „die Modell−basierte Entwicklung durch einen neuartigen Ansatz zur systematischen Auswahl und Beschreibung von Testszenarien für die in Steuerungs− und Regelsysteme eingebettete Software.“ Diese Erweiterung beruht im Wesentlichen auf der Kombination der CTM mit einer im Automobilbereich verbreiteten Technik zur Beschreibung zeitabhängiger Stimulisignale mittels des Stimulus Editor des Werkzeugs CONTROLDESK TEST AUTOMATION der Firma *dSPACE GmbH*. Als grafisches Werkzeug der CTM/ES kommt ebenfalls der Classification−Tree Editor (CTE) zum Einsatz (s. Kap. 5.4).

Im Rahmen der abstrakten Beschreibung von Testszenarien[5] wird zunächst ein schnittstellenbasierter Klassifikationsbaum auf der Basis von ausführbaren Modellen erstellt. Anstatt abstrakter Testfälle, wie im Fall der CTM, werden Testsequenzen, bestehend aus einzelnen Testschritten, in der Kombinationstabelle spezifiziert. Die Testschritte können mit Stützstellen versehen werden, so dass die Vorgabe eines zeitlichen Ablaufs möglich ist. Weiterhin bietet die CTM/ES durch so genannte Transitionen die Möglichkeit, die Übergänge zwischen den einzelnen Testschritten einer Testsequenz näher zu spezifizieren. Interpolationsvorschriften für jede Transition wie z.B. Sprung, Rampe, Sinus und weitere Vorschriften sind definiert

[5] *Conrad* (2004) verallgemeinert die Begriffe Testfall und Testsequenz zu dem Begriff Testszenario.

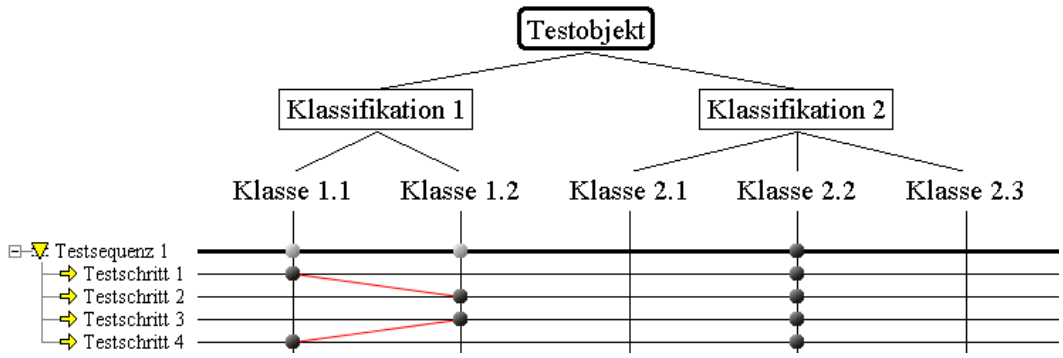


Abbildung 5.5: Kombinationstabelle mit Testsequenz bestehend aus vier Testschritten und zwei Transitionen

und werden durch den Linientyp (keine Linie, durchgezogene Line, gestrichelte Line, ...) der Transition bestimmt.

Die in Abb. 5.5 spezifizierte `Testsequenz 1` beschreibt, ausgehend vom Ausgangszustand in dem `Klasse 1.1` und `Klasse 2.2` ausgewählt sind, den Wechsel von `Klasse 1.1` zu `Klasse 1.2` und im letzten Testschritt wieder den Wechsel zurück zu `Klasse 1.1` mittels der Rampen–Interpolationsvorschrift. Auf diese Weise wird ein zeitlicher Ablauf zur Beeinflussung des SUT vorgegeben. Wäre `Klassifikation 1` z.B. als Geschwindigkeit aufgeführt und würden die zwei Äquivalenzklassen zwei verschiedenen Geschwindigkeitswerten entsprechen, so würde die aufgeführte Testsequenz einen linearen Anstieg vom ersten zum zweiten Geschwindigkeitswert sowie die anschließende Rückkehr zum Ausgangswert beschreiben.

5.2.1 Überdeckungsmaße

Zur Markierung der Blattklassen in der Kombinationstabelle nennt *Conrad* (2004) fünf verschiedene Kombinationsregeln:

- minimal combination
- one factor at a time
- n–wise combination
- random combination
- complete combination

Diese Kombinationsregeln basieren jedoch lediglich auf kombinatorischen Gesichtspunkten und berücksichtigen keine Informationen aus der funktionalen Spezifikation. In der Praxis tritt im Allgemeinen eine Mischung der fünf verschiedenen Regeln auf.

Auch bei einem CTM/ES−Klassifikationsbaum sollten prinzipiell alle Blätter in Testsequenzen berücksichtigt werden ($CTC_{min} = 1$). Wie oft und in welchem Kontext eine Äquivalenzklasse ausgewählt wird, bleibt jedoch dem Testersteller überlassen. Er muss sich dabei an der funktionalen Spezifikation orientieren und nach Möglichkeit jede darin enthaltene Anforderung mit mindestens einer Testsequenz abdecken. Alle weiteren in Abschnitt 5.1.3 definierten, auf der CTM basierenden, Überdeckungsmaße haben ebenso wenig Aussagekraft bei CTM/ES−Klassifikationsbäumen, da der zeitliche Ablauf der Testsequenzen dabei nicht geeignet berücksichtigt wird.

5.3 Weitere Erweiterungen der Classification−Tree Method

Die folgenden Unterabschnitte beinhalten kurze Abhandlungen von Arbeiten, die auf der Erweiterung bzw. Anpassung der Classification−Tree Method beruhen. Detailliertere Informationen finden sich in der aufgeführten Literatur.

5.3.1 Anwendungspragmatik

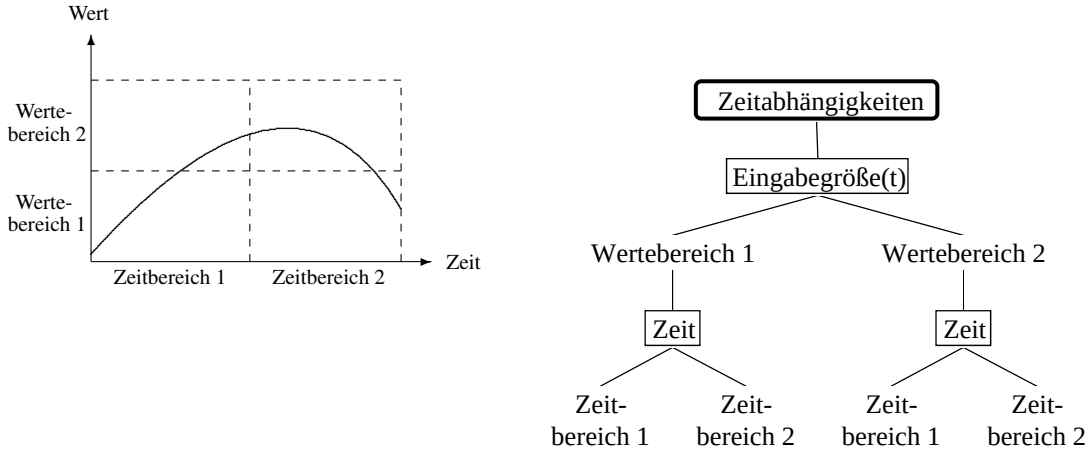
Bereits *Grimm* (1995) erachtet die Entwicklung einer Pragmatik für den praktischen Einsatz der CTM als hilfreich, da „das Auffinden der geeigneten Klassifikationsgesichtspunkte und Fallunterscheidungen aus der informellen Spezifikation sowohl Intuition und Kreativität als auch Erfahrung mit dem Umgang der Methode erfordern.“

Die von *Simmes* (1997) entwickelte Anwendungspragmatik gibt an, wie sich die zwei Charakteristika reaktiver Systeme

- Zeitbedingungen
- innere Zustände

mit der Classification−Tree Method abbilden lassen. Außerdem werden ein Vorgehen in Abhängigkeit des aktuellen Entwicklungsstandes definiert und Regeln zur Beherrschung der Komplexität angegeben.

Die folgenden Abschnitte lehnen sich stark an die Arbeiten von *Heldwein* (1996) und *Simmes* (1997) an.



(a) Eingabegröße(t) (b) Klassifikationsbaum

Abbildung 5.6: Abbildung von Zeitbedingungen im Klassifikationsbaum (*Simmes*, 1997)

5.3.1.1 Abbildung der Charakteristika von Steuergeräten

Zeitbedingungen Steuergeräte sind in einen technischen Prozess eingebunden und unterliegen daher den zeitlichen Anforderungen (vgl. Realzeitanforderungen in Kap. 3.3.3) des technischen Prozesses. Ein– und Ausgangssignale ebenso wie das Gesamtsystem sind daher zeitbehaftet (vgl. Kap. 3.3.2).

Folgende Zeitabhängigkeiten, die durch die CTM dargestellt werden müssen, identifiziert *Simmes* (1997) zwischen Eingangsgrößen und Steuergerätereaktionen:

1. „Nach Ablauf einer spezifizierten Zeit erfolgt, evtl. abhängig von der Änderung der Eingangsgrößen, eine Reaktion des Systems.“
2. „Um eine Systemreaktion hervorzurufen muß eine Eingangsgröße einen Wert mindestens eine spezifizierte Zeitspanne annehmen.“
3. „Zwei Eingangsgrößen müssen ihren Wert gleichzeitig ändern.“
4. „Eine Eingangsgröße muß innerhalb einer festgelegten Zeit, nachdem ein Ereignis eingetroffen ist oder eine andere Eingangsgröße ihren Wert geändert hat, ebenfalls ihren Wert ändern, um eine Reaktion hervorzurufen.“

Dieser zeitliche Bezug lässt sich, wie in Abb. 5.6 dargestellt, mittels der CTM abbilden. Zunächst wird die zeitbehaftete Eingangsgröße bzgl. der für den Test relevanten Werte klassifiziert (s. Abb. 5.6(a)). Danach wird jeder dabei entstandene Wertebereich unter dem Gesichtspunkt des zeitlichen Verhaltens betrachtet (s. Abb. 5.6(b)). „Bei der Wahl dieser Klassen sollten Vermutungen bzw. Informationen über das Verhalten des Systems in den spezifizierten Zeitbereichen und Grenzwertkriterien einfließen“ (*Simmes*, 1997). Insbesondere

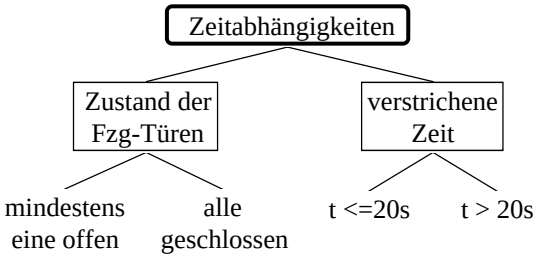


Abbildung 5.7: Modellierung von direkt zeitbezogenen Reaktionen (*Simmes*, 1997)

muss bei einem Test überprüft werden, wie sich das System bei Einhaltung bzw. Verletzung der Zeitbedingung verhält.

Für die Abbildung der oben beschriebenen Zeitabhängigkeiten zeigen Abb. 5.7 und 5.8 mögliche Realisierungen. *Simmes* (1997) betont, dass die vorgestellten Beispiele zur Modellierung der Zeitabhängigkeiten mit der CTM sicherlich auch auf anderem Wege hätten dargestellt werden können, da „die Frage, wie die Zeitabhängigkeit im Klassifikationsbaum modelliert wird, vom Testproblem und dem Testziel“ abhängt. Die vorgestellten Beispiele zeigen, dass die Abbildung von Zeitbezügen prinzipiell möglich ist. Als strenge Modellierungsvorschriften sind die Beispiele jedoch nicht anzusehen.

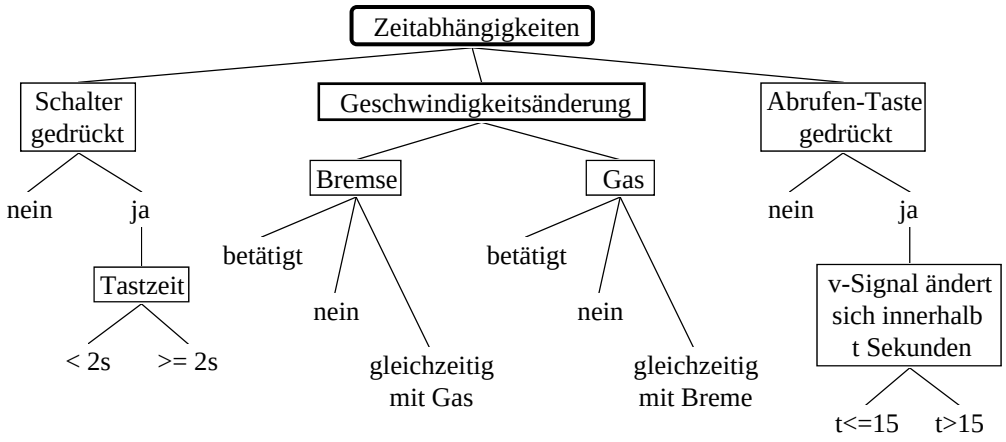


Abbildung 5.8: Modellierung von Zeitbezügen mit der CTM am Beispiel einer Geschwindigkeitsregelanlage (*Simmes*, 1997)

Berücksichtigung innerer Zustände Gemäß der Charakteristika reaktiver Systeme hängt das Verhalten des Systems nicht nur von den Eingaben, sondern auch von seiner Vorgeschichte ab. Daher muss bei der Definition der Testfälle der aktuelle Zustand des SUT berücksichtigt werden.

Für den Umgang mit Zuständen rät *Simmes* (1997) bei einer kleinen Anzahl innerer Zustände zur direkten Aufnahme dieser Zustände in den Klassifikationsbaum des SUT. Bei mehr als etwa fünf Zuständen wird die Erstellung eines eigenen Klassifikationsbaumes für jeden Zustand empfohlen. Jeder Zustand wird damit selbst zum SUT.

5.3.1.2 Anwendung in Abhängigkeit vom Entwicklungsstand

Zur Anwendung der CTM in Abhängigkeit vom Entwicklungsstand liefert *Simmes* (1997) bzw. *Heldwein* (1996) zwei Ansätze:

- fahrzeugorientierter Ansatz
- steuergeräteorientierter Ansatz

Beim fahrzeugorientierten Ansatz „werden testrelevante Situationen (Fahr– oder Bediensituationen) aus dem vorgesehenen Einsatzbereich des Fahrzeugs oder Fahrzeugsystems abgeleitet. Dabei wird das Fahrverhalten in Bezug auf die Funktionalität des zu testenden Steuergeräts betrachtet“ (*Simmes*, 1997). Er ist daher schon in sehr frühen Entwicklungsphasen einsetzbar. Einen Nachteil im fahrzeugorientierten Ansatz stellt das Vorhandensein von Redundanzen in der erstellten Testfallspezifikation dar.

Der steuergeräteorientierte Ansatz vermeidet Redundanzen, benötigt jedoch detaillierte Spezifikationsdokumente, wodurch er erst in späteren Entwicklungsphasen einsetzbar ist. Dabei werden direkt die inneren Zustände des Steuergerätes identifiziert und für jeden ermittelten Zustand ein Klassifikationsbaum erstellt.

5.3.1.3 Techniken zur Beherrschung der Komplexität

Sind mehr als ein oder zwei Klassifikationsbäume zum Test des SUT notwendig, so wird die Anwendbarkeit der CTM maßgeblich durch Techniken zur Beherrschung der Komplexität bestimmt. *Simmes* (1997) unterscheidet dabei zwischen

- überschaubaren Systemen und
- komplexen Systemen

und definiert für jedes System ein entsprechendes Vorgehen. Ein überschaubares System ist durch wenige Zustandsübergänge zwischen den verschiedenen Zuständen charakterisiert, die zudem durch einfache Bedingungen ausgelöst werden, während bei komplexen Systemen eine große Anzahl an Übergängen existiert.

Wesentlicher Unterschied zwischen den zwei verschiedenen Herangehensweisen ist die unterschiedliche Behandlung der Zustandsübergänge. Bei überschaubaren Systemen erfolgt

die Spezifikation der Zustandsübergänge innerhalb jedes Klassifikationsbaumes, während bei komplexen Systemen ein zusätzlicher Klassifikationsbaum, der so genannte Übergangsbaum, eingeführt wird. Im Übergangsbaum ist das gesamte Übergangsverhalten zwischen allen Zuständen des Gesamtsystems spezifiziert.

5.3.2 Model−based black−box testing

Model−based black−box testing (MB^{3}T) wurde von *Conrad et al.* (2004) vorgestellt. MB^{3}T kombiniert den Einsatz der CTM mit dem Einsatz der CTM/ES und unterteilt sich in drei Hauptaktivitäten:

1. Anforderungsbasiertes Testdesign
2. Modellbasiertes Testdesign
3. Konsistenzprüfung

Die CTM wird dabei zum Spezifizieren anforderungsorientierter[6] Testfälle eingesetzt. „Anforderungsorientierte Testfälle sind jedoch abstrakt und somit nicht ausführbar. Deshalb können sie nicht direkt zur Testdurchführung eingesetzt werden. Es ist notwendig, zusätzliche ausführbare Testszenarien zu spezifizieren, die das SUT über seine Schnittstellen stimulieren können“ *Sadeghipour et al.* (2005). Diese ausführbaren Testszenarien werden mithilfe der CTM/ES spezifiziert bzw. aus funktionalen Modellen abgeleitet. Abschließend erfolgt eine Konsistenzprüfung zwischen anforderungsorientierten und modellbasierten Testszenarien. Damit wird u.a. sichergestellt, dass alle anforderungsorientierten Testszenarien durch modellbasierte Testszenarien abgedeckt sind.

MB^{3}T eignet sich aufgrund des hohen Aufwands durch die zwei verschiedenen Blickwinkel bei der Entstehung der Testszenarien insbesondere für Software mit hohen Sicherheits− und Zuverlässigkeitsanforderungen.

Dieser kombinierte Einsatz der CTM mit der CTM/ES bildet zum Teil die Grundlage für das im folgenden Kapitel vorgestellte Konzept zur Kopplung der beiden systematischen Testverfahren und zur Integration in den in Kap. 3.6 vorgestellten Testerstellungsprozess.

5.3.3 Integrated Classification−Tree Method

Die von *Chen & Poon* (1997) entwickelte Integrated Classification−Tree Method (ICTM), berücksichtigt eine systematischere Vorgehensweise bei der Erstellung des Klassifikationsbaumes als die CTM (*Armbrust et al.*, 2004a). Dabei kommt eine so genannte „Classification

[6] Anstatt des Begriffs anforderungsbasierter Testfall verwendet *Conrad et al.* (2005) auch die Begriffe logischer Testfall bzw. logisches Testszenario.

Hierarchy Table“ zum Einsatz, deren Zeilen und Spalten jeweils die identifizierten Klassifikationen des SUT darstellen. Die Tabelle enthält sämtliche Beziehungen zwischen den identifizierten Klassifikationspaaren. Diese Relationen werden durch drei so genannte „hierarchical operators“ beschrieben (*Chen et al.*, 2000). Ausgehend von der manuell erstellten Klassifikationshierarchietabelle generiert schließlich ein Algorithmus den Klassifikationsbaum automatisch.

Nachteil dieses Vorgehens ist der große Aufwand bei einer großen Anzahl von ermittelten Klassifikationen. Bei n Klassifikationen müssen in der Klassifikationshierarchietabelle n^2 Relationen definiert werden.

5.3.4 Attributierte Klassifikationsbäume

Eine weitere Erweiterung der CTM stellt die von *Lützkendorf & Bothe* (2003) präsentierte Attributierung von Klassifikationsbäumen dar. Durch diese Anreicherung der Klassifikationsbäume mit Attributen kann der manuelle Schritt von der Testfallspezifikation zur Testdatengenerierung (vgl. Kap. 2.4) automatisiert werden.

Zur Erreichung dieser Automatisierung ist zwar etwas mehr Aufwand bei der Erstellung und Wartbarkeit der attributierten Klassifikationsbäume erforderlich. Dieser zusätzliche Aufwand ist jedoch angesichts der folgenden Vorteile vertretbar. Zum einen wird die Wartbarkeit von spezifizierten Testfällen deutlich verbessert. Außerdem lassen sich für große Testfallmengen automatisch konsistente Testdaten erzeugen. Erste Erfahrungen liegen in Form einer Praxisanwendung vor (*Lützkendorf*, 2001).

Diese Attributierung von Klassifikationsbäumen stellt den Ausgangspunkt für die in Anh. C vorgestellte Schnittstelle ctmes2exam zwischen dem CTE XL und der Testautomatisierung EXAM (s. Anh. B) dar.

5.4 Classification−Tree Editor eXtended Logics

Grafische Unterstützung erhält die CTM sowie die CTM/ES durch den Classification−Tree Editor (CTE). Es existieren zwei Ausführungen des CTE (*Razorcat Development GmbH* (2007) und (*Pitschinetz & Wegener*, 2007)). Alle in dieser Arbeit erstellten Klassifikationsbäume wurden mit Hilfe des Classification−Tree Editor eXtended Logics CTE XL von *Pitschinetz & Wegener* (2007) erstellt. Bei der Erstellung des Klassifikationsbaumes achtet der CTE XL auf die Einhaltung der syntaktischen Regeln der Methode. Die Erstellung der Kombinationstabelle wird ebenfalls durch den CTE XL unterstützt sowie weitere Report− und Analysefunktionen.

Der CTE XL gliedert sich in vier Bereiche (s. Abb. 5.9). Im Bereich rechts oben kann der Klassifikationsbaum erstellt werden. Der Bereich links oben dient zur Anzeige von Eigen-

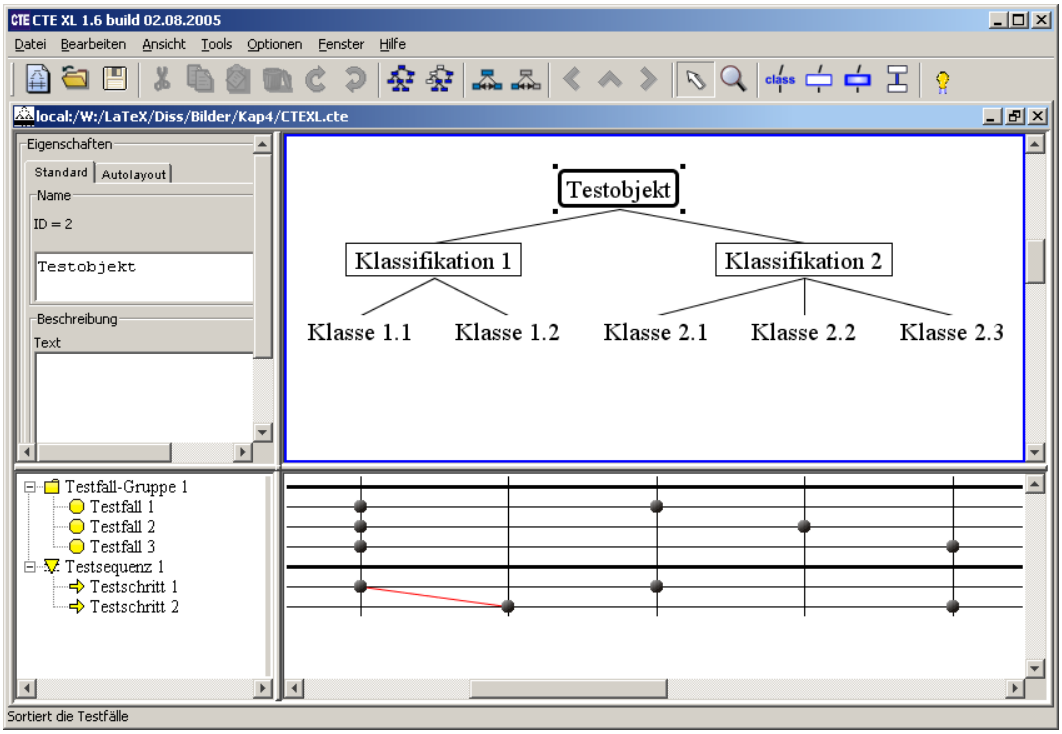


Abbildung 5.9: Screenshot des CTE XL

schaften ausgewählter Elemente. Der untere Bereich stellt die Kombinationstabelle dar. Darin können auf der linken Seite sowohl Testfälle als auch Testsequenzen definiert und in Testfall–Gruppen strukturiert werden. Testsequenzen setzen sich immer aus mehreren Testschritten zusammen. Auf der rechten Seite werden den definierten Testfällen bzw. den einzelnen Testschritten Blattklassen des Klassifikationsbaumes zugeordnet.

Einige Klassifikationen bzw. Äquivalenzklassen machen im Zusammenspiel mit anderen Klassifikationen bzw. Äquivalenzklassen keinen Sinn und sind für den Testfall daher nicht von Belang. Solche Abhängigkeiten können mittels Abhängigkeitsregeln im CTE XL durch den Testersteller spezifiziert werden. Sind in einem Testfall/Testschritt unverträgliche Kombinationen markiert, so hebt der CTE XL den betreffenden Testfall/Testschritt farblich hervor. Zudem bietet der Editor eine Anbindung an Requirement Management Werkzeuge wie z.B. DOORS, wodurch die Verknüpfung von Testszenarien mit Anforderungen ermöglicht wird.

Jedes im CTE XL erstellte Element besitzt gewisse Eigenschaften. Diese Eigenschaften werden bei Markierung des Elements im Eigenschaftsbereich oben links unter verschiedenen Reitern angezeigt. Einige Eigenschaften wie z.B. `Name` oder `Beschreibung` sind für alle Elemente vorhanden. Andere Eigenschaften existieren nur für ausgewählte Elemente. Neue Eigenschaften können im CTE XL in der Menüleiste unter „Tools → Tags...“ an alle Elemente des Klassifikationsbaumes sowie der Kombinationstabelle hinzugefügt werden. Jedes

CTE XL–Element besitzt zudem eine eindeutige ID, welche unter dem Reiter `Standard` eingesehen werden kann (s. Abb. 5.9).

Die Bedienung des Tools ist sehr intuitiv. Außerdem existiert ein Handbuch, welches in der Menüleiste unter „Hilfe → Hilfe“ zugänglich ist.

6 Konzept zur Kopplung und Integration der ausgewählten Testverfahren in den Testprozess

In diesem Kapitel wird ein neuartiges Konzept zur Kopplung und Integration der beiden in Kap. 4.2 ausgewählten, systematischen Testverfahren (CTM und CTM/ES) in den geschilderten Testerstellungsprozess, in Anlehnung an die Methode MB^{3}T vorgestellt. Hintergrund dieser Integration ist die Behebung der in Kap. 3.7 aufgezeigten Probleme bei der Testfallermittlung. Mit Hilfe der CTM und der CTM/ES kann das Test−Anforderungs−Management bzw. speziell die wichtigste Testaktivität, die Testfallermittlung, systematisiert und durch den Einsatz des CTE XL geeignet unterstützt werden. Neben der Integration der beiden Testverfahren wird zudem eine mögliche Kopplung des CTE XL zu der bei der AUDI AG eingesetzten Testautomatisierung EXAM (s. Anh. B) skizziert. An einem Praxisbeispiel findet abschließend eine Demonstration der vorgeschlagenen Vorgehensweise statt.

Das aufgezeigte Konzept wird in Kap. 7 einem Praxistest unterzogen. Die finale Umsetzung des Konzeptes behandelt dagegen Kap. 8.

6.1 Konzept zur Kopplung und Integration der ausgewählten Testverfahren

Das vorgeschlagene Konzept zur Kopplung und Integration der CTM sowie der CTM/ES in den bei der AUDI AG bestehenden Testprozess von Automobilelektronik mittels HIL Testsystemen lehnt sich an die von *Conrad et al.* (2004) vorgestellte Methode MB^{3}T (model−based black−box testing) an (vgl. Kap 5.3.2). Das Konzept wird in Kap. 6.2 anhand eines praxisnahen Beispiels der Funktion Richtungsblinken exemplarisch vorgestellt.

Die CTM wird im vorgestellten Konzept ausgehend von Anforderungsdokumenten ebenso wie bei der MB^{3}T−Methode zum Spezifizieren anforderungsorientierter bzw. logischer Testfälle eingesetzt (s. Abb. 6.1). Ab hier unterscheidet sich dagegen das Vorgehen von dem der MB^{3}T−Methode. Funktionale Modelle, aus denen mittels der CTM/ES schnittstellenbasierte Klassifikationsbäume abgeleitet werden können, liegen wie im Fall der Vorgehensweise

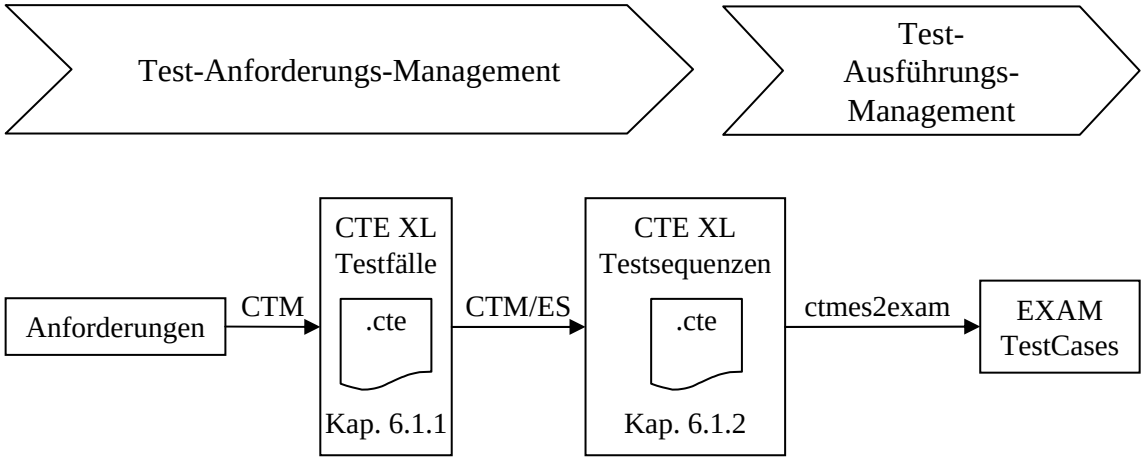


Abbildung 6.1: Konzeptvisualisierung

nach der MB^{3}T–Methode nicht vor. Daher wird die CTM/ES im vorliegenden Fall zur Konkretisierung der anforderungsorientierten Testfälle verwendet. Dabei entstehen Testsequenzen, welche die spätere Testdurchführung ermöglichen. Eine Konsistenzprüfung zwischen CTM–Testfällen und CTM/ES–Testsequenzen wie im Fall der Methode MB^{3}T ist nicht notwendig, weil die CTM/ES–Testsequenzen nicht unabhängig spezifiziert sondern aus den zuvor spezifizierten CTM–Testfällen abgeleitet werden.

In den folgenden Unterkapiteln wird der angestrebte Einsatz (s. Abb. 6.1) der CTM bzw. der CTM/ES sowie deren toolgestützte Anbindung an den vorherrschenden Testprozess, wie er in Kap. 3.6 aufgezeigt wurde, näher vorgestellt. Als Tool kommt der bereits in Kap. 5.4 vorgestellte CTE XL zur Erstellung der Klassifikationsbäume inkl. Kombinationstabelle zum Einsatz, der sowohl die CTM als auch die CTM/ES grafisch unterstützt.

6.1.1 Einsatz der CTM

Die CTM kommt im Test–Anforderungs–Management zum Spezifizieren anforderungsorientierter bzw. logischer Testfälle zum Einsatz (s. Abb. 6.1). Grundlage für die Anwendung der CTM sind daher Anforderungsdokumente. Im Automobilbereich liegen funktionale Spezifikationen meist in textueller Form vor (vgl. Kap. 3.4.1). Davon ausgehend wird ein Klassifikationsbaum nach der CTM im CTE XL erstellt (vgl. Kap. 5.1.1). Anschließend erfolgt die Definition logischer Testfälle in der Kombinationstabelle durch die Kombination geeigneter Blattklassen des Klassifikationsbaumes (s. Kap. 6.2.2).

In Abb. 6.2 sind ein beispielhafter CTM–Klassifikationsbaum sowie drei logische bzw. anforderungsorientierte Testfälle in der darunter liegenden Kombinationstabelle dargestellt[1].

[1] Einen praxisnahen CTM–Klassifikationsbaum zeigt Abb. 6.9 auf S. 76.

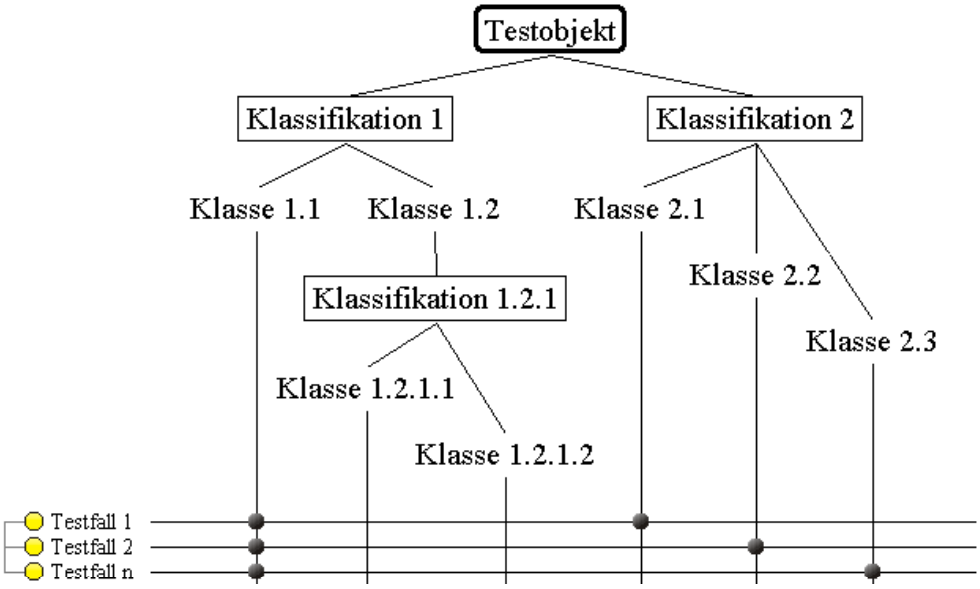


Abbildung 6.2: CTM−Klassifikationsbaum inkl. drei logischen Testfällen

Jeder dieser spezifizierten Testfälle wird im folgenden Schritt mittels der CTM/ES in eine Testsequenz überführt. Jede Testsequenz beschreibt den späteren Testablauf des anforderungsbasierten Testfalls in einer detaillierten Form.

6.1.2 Einsatz der CTM/ES

Die CTM/ES wird zur Konkretisierung der mittels der CTM spezifizierten, anforderungsbasierten Testfälle eingesetzt (s. Abb. 6.1). Die spezifizierten Testsequenzen sind im Grenzbereich zwischen dem Test−Anforderungs−Management und dem Test−Ausführungs−Management anzuordnen, da sie sowohl Spezifikations− als auch Modellierungsanteile besitzen (s. Abb. B.2).

Bevor der Testersteller den konkreten Testablauf als Testsequenz in plattformunabhängiger Form in der Kombinationstabelle angeben kann, muss der CTM−Klassifikationsbaum (s. Abb. 6.2) in einen CTM/ES−Klassifikationsbaum (s. Abb. 6.3) transformiert werden. Einige Klassifikationen und Klassen können bei dieser Transformation ohne Änderungen übernommen werden. Dazu zählen z.B. Klassifikationen, die globale Randbedingungen des Testfalls beschreiben und sich während des gesamten Testablaufs nicht verändern. Klassifikationen, die direkt das Verhalten des SUT im CTM−Klassifikationsbaum beschreiben, werden jedoch teilweise vollständig durch neue Klassifikationen im CTM/ES−Klassifikationsbaum ersetzt. Ein Praxisbeispiel für den soeben beschriebenen Transformationsprozess findet sich in Kap. 6.2.3.2.

Ergebnis des Einsatzes der CTM/ES für den allgemeinen Klassifikationsbaum aus Abb. 6.2

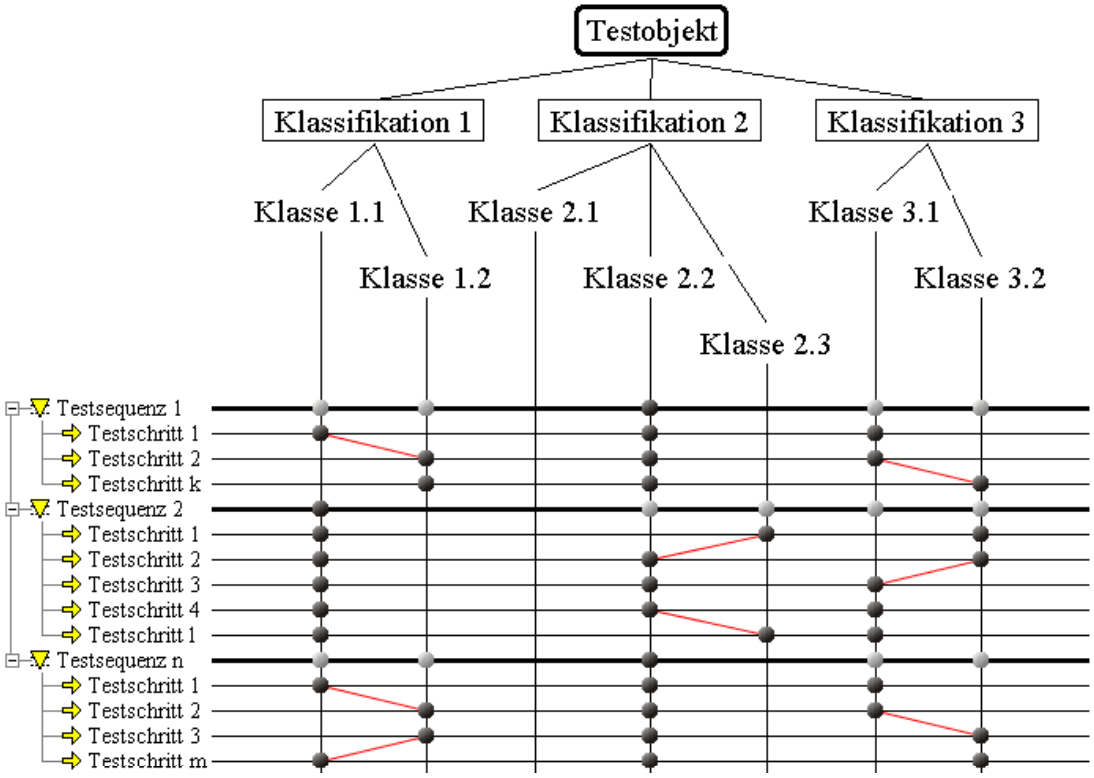


Abbildung 6.3: CTM/ES−Klassifikationsbaum inkl. drei Testsequenzen

ist ein Klassifikationsbaum wie z.B. in Abb. 6.3 dargestellt[2]. In der Kombinationstabelle befindet sich jetzt für jeden in Abb. 6.2 spezifizierten logischen Testfall eine Testsequenz, die den konkreten Ablauf des ursprünglichen Testfalls in Form von einzelnen Testschritten vorgibt bzw. präzisiert.

Mit Hilfe des entwickelten Schnittstellenprototyps ctmes2exam (s. Abb. 6.1 und Kap. C) zwischen dem CTE XL und dem ARTiSAN Studio lassen sich die im CTE XL spezifizierten Testsequenzen als TestCases[3] der eingesetzten Testautomatisierung EXAM zur Verfügung stellen. Für diese Überführung ist neben der Spezifikation der Testsequenzen, die Angabe weiterer EXAM−spezifischer Inhalte im CTE XL notwendig. Klassifikationsbaumelemente müssen z.B. mit entsprechenden Elementen der Testautomatisierung verknüpft werden. Diese Verknüpfung basiert auf der bereits in Kap. 5.3.4 vorgestellten Attributierung von Klassifikationsbäumen. Eine ausführliche Beschreibung des entwickelten, auf der CTM/ES basierenden, Schnittstellenprototyps zwischen dem CTE XL und dem ARTiSAN Studio bzw. EXAM findet sich in Anh. C.

Die jetzt in EXAM vorliegenden TestCases (s. Abb. 6.1) sind nach durchgeführter Test-

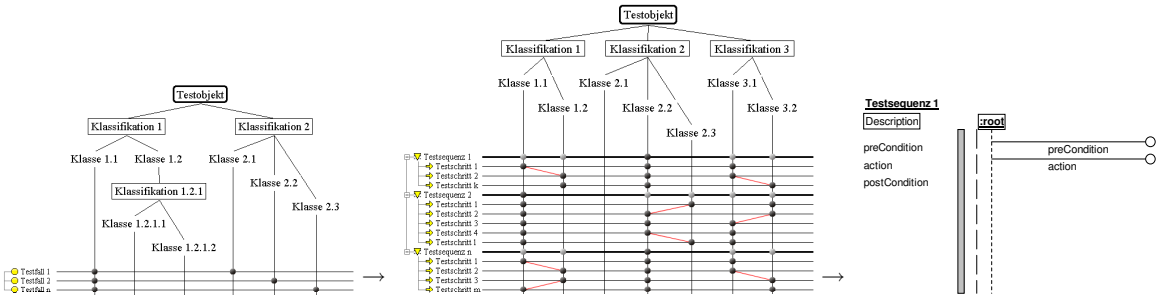
[2]Einen praxisnahen CTM/ES−Klassifikationsbaum zeigt Abb. 6.12 auf S. 82.

[3]Der Begriff TestCase wird in Anh. B im Zusammenhang mit der bei der AUDI AG eingesetzten Testautomatisierung EXAM eingeführt.

komposition (s. Anh. B.4) ausführbar. Allerdings fehlt zum vollständigen TestCase noch die Bewertung der Ausgaben des SUT (s. Anh. B.5.2). Mithilfe der CTM/ES lassen sich nur Eingaben des SUT spezifizieren. Die generierten TestCases müssen daher manuell um geeignete TestSequenzen[4] zur Bewertung der Ausgaben des SUT erweitert werden.

6.1.3 Zusammenfassung

Die soeben vorgestellte Vorgehensweise ist in Abb. 6.4 nochmals zusammenfassend visualisiert.



(a) Logische Testfälle (CTM) (b) Testsequenzen (CTM/ES) (c) EXAM TestCases ohne `postCondition`

Abbildung 6.4: Visualisierung der angestrebten Vorgehensweise

Zunächst wird ausgehend von Anforderungsdokumenten ein anforderungsorientierter Klassifikationsbaum für ein darin identifiziertes SUT erstellt. Anschließend erfolgt die Spezifikation logischer Testfälle in der Kombinationstabelle (s. Abb. 6.4(a)). Ausgehend von diesem CTM−Klassifikationsbaum wird ein CTM/ES−Klassifikationsbaum erstellt. Alle logischen Testfälle werden dabei in Testsequenzen umgewandelt (s. Abb. 6.4(b)). Die spezifizierten Testsequenzen lassen sich mit Hilfe der entwickelten Schnittstelle ctmes2exam (s. Anh. C) nach EXAM überführen (s. Abb. 6.4(c)). Alle automatisch generierten TestCases enthalten dabei bereits den plattformunabhängigen Testablauf und sind nach durchgeführter Testkomposition ausführbar.

Unterstützt wird diese Vorgehensweise werkzeugseitig durch den CTE XL, der sich wie in Abb. 6.5 dargestellt in den bestehenden Testerstellungsprozess (vgl. Kap. 3.6) integriert. Testfälle können auf diese Weise systematisch mittels der CTM in der Kombinationstabelle spezifiziert werden. Der bestehende Formalisierungsgrad sowie die existierenden Überdeckungsmaße ermöglichen Definitionen von messbaren Testzielen und die Bewertung des bestehenden Testumfangs. Ähnliche Testfälle lassen sich einfach strukturieren, wodurch u.a. redundante Testinhalte leicht identifiziert werden können.

[4] Ebenso wie auch der Begriff TestCase wird der Begriff TestSequenz in Anh. B eingeführt.

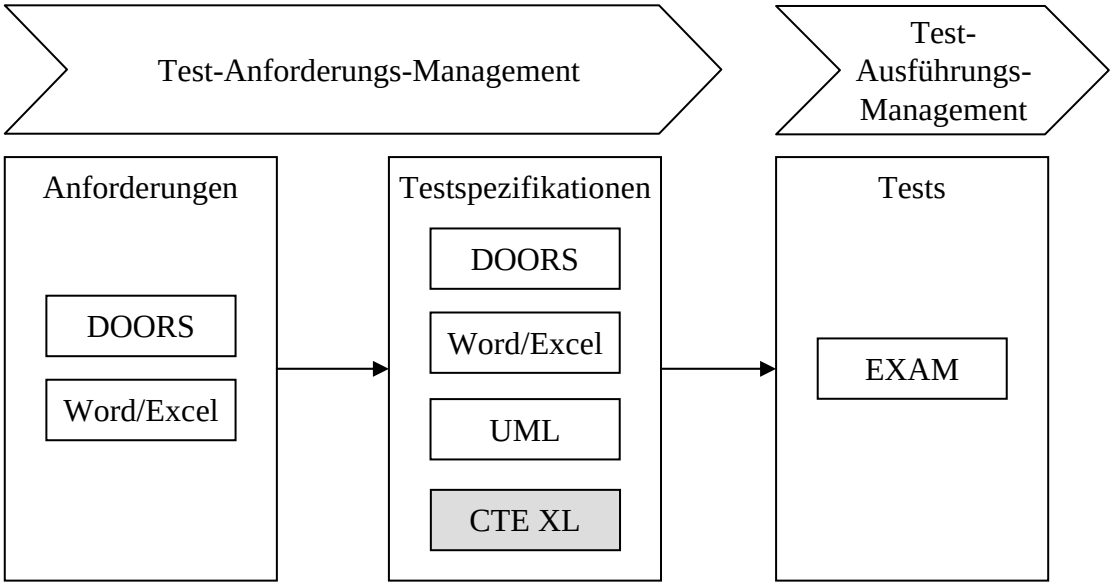


Abbildung 6.5: Einbindung des CTE XL in den bestehenden EXAM−Testerstellungsprozess

6.2 Anwendungsbeispiel

Das soeben vorgestellte Konzept wird im vorliegenden Abschnitt an einem vereinfachten Praxisbeispiel der Funktion Richtungsblinken veranschaulicht. Ausgehend von der Anforderungsdefinition wird zunächst die CTM angewandt. Dabei spezifizierte, logische Testfälle werden anschließend durch den Einsatz der CTM/ES in Testsequenzen überführt.

6.2.1 Funktionale Spezifikation

Folgende realitätsnahe, funktionale Spezifikation der Funktion Richtungsblinken soll als Grundlage für das weitere Vorgehen dienen:

1. Das Schaltermodul−Lenksäule (SMLS) kann ein Richtungsblinken anfordern.
2. Für die Blinkfunktion gilt der erweiterte Betriebsspannungsbereich 6,5 bis 26 Volt.
3. Die Funktion Richtungsblinken muss bei Klemme15=aus (Kl15 aus) deaktiviert werden[5].
4. Beim Ausschalten der Blinkanforderung während der Hellphase (HP) wird der begonnene Zyklus zu Ende geführt. Wird ein gültiger Blinkerwechsel erkannt, muss sofort umgeschaltet werden.

[5]Die Bezeichnung Klemme15 (Kl15) steht im Automobilbereich stellvertretend für die Zündung. Kl15 aus ist daher gleichzusetzen mit ausgeschalteter Zündung.

5. Bei einem Nachtriggern der Blinkanforderung in gleicher Richtung noch während der Dunkelphase (DP) muss die Dunkelphase vor dem nächsten Zuschalten der Hellphase eingehalten werden. Ansonsten wird mit der Hellphase der Blinkzyklus begonnen.

6. Hell– und Dunkelphase haben eine Dauer von jeweils 400ms.

Aus Gründen der Übersichtlichkeit wurden nicht alle Anforderungen der Funktion Richtungsblinken berücksichtigt.

6.2.2 Anwendung der CTM

An dieser Stelle blicken wir zunächst auf die drei wesentlichen Schritte der Classification–Tree Method (vgl. Kap. 5.1.1) zurück:

1. Definition des SUT
2. Klassifikation des Eingabedatenraums
   a) Definition von testrelevanten Gesichtspunkten
   b) vollständige, disjunkte Zerlegung in Äquivalenzklassen
   c) Iteration
3. Testfallermittlung

In den folgenden Unterkapiteln wird jeder dieser Schritte anhand der oben definierten Anforderungen der Funktion Richtungsblinken nach und nach durchgeführt.

6.2.2.1 Definition des SUT

Als erstes wird das SUT definiert, wofür entsprechend detaillierte und strukturierte Anforderungsdokumente die Grundlage darstellen. Liegen diese Dokumente nicht vor und/oder ist der Detaillierungs– oder Strukturierungsgrad gering, kann sich die Definition des SUT unter Umständen sehr schwer gestalten.

Die oben beschriebene Funktion Richtungsblinken ist eine Subfunktion der Funktion Blinken, welche wiederum eine Subfunktion der Funktion Licht und Beleuchtung ist. Sie ist je nach Ausstattungsvariante auf bis zu sieben der in Abb. 3.2 dargestellten Komponenten verteilt.

Ausgehend von dieser Strukturierung der Funktionsbeschreibung lässt sich bereits die Subfunktion Richtungsblinken als eigenständiges SUT identifizieren (vgl. Kap. 5.1.1.1). Die Wurzel des Klassifikationsbaumes erhält daher den Namen `Richtungsblinken`

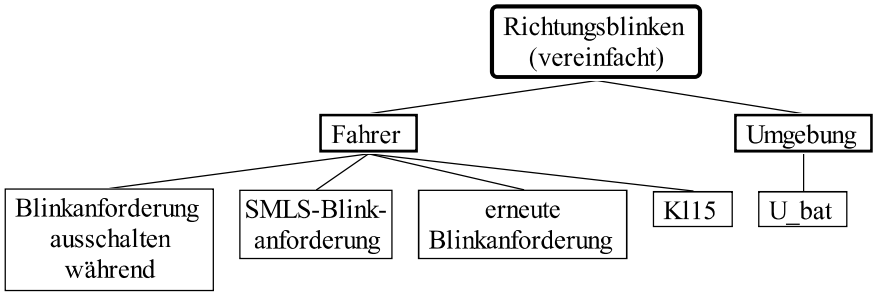


Abbildung 6.6: Erste CTM–Klassifikationen der vereinfachten Funktion Richtungsblinken

`(vereinfacht)` (s. Abb. 6.6). Weitere SUT bilden weitere Subfunktionen wie z.B. Warnblinken, Crashblinken und Notbremswarnblinken der Funktion Blinken. Von einer weiteren Subfunktion Blinkerpriorisierung werden Abhängigkeiten zwischen den verschiedenen Subfunktionen der Funktion Blinken dargestellt. Die folgende Anwendung der CTM und CTM/ES beschränkt sich jedoch auf die durch die in Kap. 6.2.1 definierten Anforderungen beschriebene, vereinfachte Funktion Richtungsblinken.

6.2.2.2 Klassifikation des Eingabedatenraums

Nachdem das SUT `Richtungsblinken (vereinfacht)` definiert ist, erfolgt die Klassifikation seines Eingabedatenraums.

Definition von testrelevanten Gesichtspunkten Zunächst werden Gesichtspunkte, die das SUT beeinflussen (könnten), gesucht. Quelle hierfür ist wiederum die funktionale Spezifikation (vgl. Kap. 6.2.1). Aus den darin definierten Anforderungen lassen sich Gesichtspunkte ableiten, anhand derer die erste Klassifikation des identifizierten SUT vorgenommen wird. Bevor jedoch einzelne Klassifikationen identifiziert werden, können zunächst zwei Kompositionen abgeleitet werden. Eine Komposition fasst Aktionen des Fahrers zusammen und wird deshalb mit dem Namen `Fahrer` versehen. Unter einer zweiten Komposition können die Umgebung beeinflussende Größen zusammengefasst werden. Sie erhält deshalb den Namen `Umgebung`.

Aus der ersten Anforderung leiten wir die Klassifikation `SMLS-Blinkanforderung` ab. Sie beinhaltet die Möglichkeit einer Blinkanforderung durch das SMLS[6]. Die zweite Anforderung wiederum liefert die Batteriespannung `U_bat` als relevante Klassifikation. Anforderung drei liefert den Status der `Kl15` als relevanten Aspekt. Die vierte und fünfte Anforderung gibt

[6]Natürlich erfolgt die Blinkanforderung durch den Fahrer, indem er den Blinkerhebel in die entsprechende Position bewegt. Das SMLS fungiert als Schnittstelle zwischen dem Fahrer und der Funktion Richtungsblinken und gibt den Fahrerwunsch an das Fahrzeugsystem weiter. Daher wird aus der Funktionssicht von einer Blinkanforderung durch das SMLS gesprochen.

jeweils den Hinweis, dass der Moment, in dem ein Blinker ein– bzw. ausgeschaltet wird, relevant für das Verhalten des SUT ist. Deshalb wird die Klassifikation `Blinkanforderung ausschalten während` angelegt, die den Ausschaltvorgang einer Blinkanforderung näher beschreibt. Die Klassifikation `erneute Blinkanforderung` betrachtet dagegen, ob eine weitere Blinkanforderung auftritt. Alle bisher identifizierten Gesichtspunkte finden sich als Klassifikationen in Abb. 6.6.

Anforderung sechs beschreibt das Sollverhalten der Funktion Richtungsblinken. Hell– und Dunkelphase müssen danach jeweils 400*ms* dauern, falls nicht, wie in Anforderung vier beschrieben, ein Blinkerwechsel stattfindet. Da es sich hierbei nicht um Bestandteile des Eingabedatenraums handelt, werden diese Informationen auch nicht in den Klassifikationsbaum aufgenommen.

Vollständige, disjunkte Zerlegung in Äquivalenzklassen Nun werden die identifizierten Gesichtspunkte vollständig und disjunkt in Äquivalenzklassen zerlegt.

Für die Batteriespannung `U_bat` werden folgende drei Äquivalenzklassen gewählt:

- Unterspannung (`unter`)
- Normalspannung (`normal`)
- Überspannung (`über`)

Hintergrund dieser Aufteilung sind weitere Anforderungen, die den erweiterten Betriebsspannungsbereich (vgl. Kap. 6.2.1) näher spezifizieren. Eine feinere Unterteilung der Batteriespannung kann bei Bedarf vorgenommen werden. Für unser Beispiel wird jedoch angenommen, dass die soeben durchgeführte Aufteilung ausreichend ist.

Die `Kl15` kann entsprechend der zwei Möglichkeiten Zündung ein und Zündung aus vollständig und disjunkt in folgende zwei Äquivalenzklassen unterteilt werden:

- `ein`
- `aus`

Für die Klassifikation `SMLS-Blinkanforderung` ergeben sich gemäß den zwei Positionen des Blinkerhebels für eine Blinkanforderung nach links und rechts die zwei Äquivalenzklassen:

- `links`
- `rechts`

Eine Blinkanforderung kann in zwei Zuständen zurückgenommen werden: in der Hellphase (HP) und in der Dunkelphase (DP) eines Blinkzyklus. In den Anforderungen wird die Anzahl der Hell– und Dunkelphasen nicht näher betrachtet. Aus diesem Grund werden folgende vier[7] Äquivalenzklassen für die Klassifikation `Blinkanforderung ausschalten während` definiert:

- `1. HP`
- `1. DP`
- `x. HP`
- `x. DP`

Damit kann eine Rücknahme der Blinkanforderung während der ersten Hell– bzw. Dunkelphase, aber auch zu einem späteren Zeitpunkt während einer beliebigen Hell– bzw. Dunkelphase erfolgen.

Entweder folgt einer `SMLS-Blinkanforderung` eine `erneute Blinkanforderung` oder nicht. Dementsprechend wird die Klassifikation `erneute Blinkanforderung` in die zwei Äquivalenzklassen zerlegt:

- `ja`
- `nein`

In Abb. 6.7 ist der zu diesem Zeitpunkt aktuelle Stand des Klassifikationsbaumes dargestellt.

Iteration Nachdem die ersten Klassifikationen erstellt und auch schon in Äquivalenzklassen unterteilt wurden, stellt sich die Frage, ob eine weitere Klassifikation an einigen Stellen Sinn macht.

Falls in unserem Beispiel eine erneute Blinkanforderung vorliegt, so ist u.a. der Zeitpunkt dieser erneuten Anforderung für das Verhalten des SUT relevant. Aus diesem Grund wird unterhalb der Äquivalenzklasse `erneute Blinkanforderung→ja`[8] eine weitere Klassifikation `Anforderung während laufendem Blinkzyklus` hinzugefügt. Auch hierbei wird wieder zwischen folgenden zwei Äquivalenzklassen unterschieden:

[7] Eine fünfte Äquivalenzklasse mit Namen `Reaktionszeit` ist denkbar, da eine gewisse Zeitspanne (ca. 100*ms*) zwischen einer Blinkanforderung und der tatsächlichen Umsetzung dieser Anforderung (Ansteuerung der jeweiligen Blinkerlampen) vergeht. Es ist durchaus möglich, die Blinkanforderung innerhalb der Reaktionszeit des Systems zurückzunehmen, d.h. bevor die Anforderung umgesetzt wird. Dieser Fall sowie einige weitere Spezialfälle werden jedoch aus Gründen der Übersichtlichkeit nicht betrachtet.

[8] Die Schreibweise `A→B` stellt im Folgenden die Kombination von einer Äquivalenzklasse B und der dazugehörigen Klassifikation A dar.

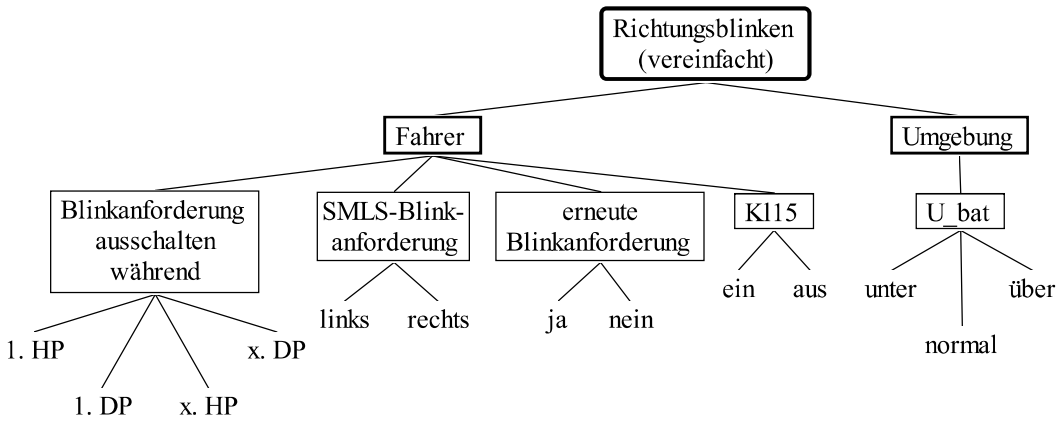


Abbildung 6.7: CTM−Klassifikationsbaum der vereinfachten Funktion Richtungsblinken

- `ja`
- `nein`

Zudem kann nach der `Art der Anforderung` einer vorliegenden erneuten Blinkanforderung unterschieden werden. Als einzige Möglichkeiten ergeben sich hier die zwei Äquivalenzklassen:

- `Retrigger`
- `Gegentrigger`

Eine Retriggeranforderung stellt eine erneute Blinkanforderung mit gleicher Richtung dar (z.B. zweimal Linksblinken). Die Gegentriggeranforderung berücksichtigt entsprechend eine erneute Blinkanforderung in die Gegenrichtung (z.B. einmal Links− gefolgt von einmal Rechtsblinken). Anstatt `Retrigger`−Anforderung und `Gegentrigger`−Anforderung könnten die Äquivalenzklassen auch mit `links` und `rechts` benannt werden. Ob die erneute Blinkanforderung eine Gegentriggeranforderung oder eine Retriggeranforderung darstellt, würde sich in diesem Fall erst durch die entsprechende Auswahl der Äquivalenzklasse der Klassifikation `SMLS-Blinkanforderung` entscheiden. Beide Bezeichnungen sind jedoch äquivalent. Das aufgeführte Beispiel macht von der ersten Möglichkeit Gebrauch. An dieser Stelle wird der in Kap. 5.1.3 bereits angesprochene nichtdeterministische Entstehungsprozess von Klassifikationsbäumen besonders gut deutlich.

Schließlich kann noch die neue Klassifikation `Anforderung ausschalten während` mit den vier bereits bekannten Äquivalenzklassen hinzugefügt werden:

- `1. HP`

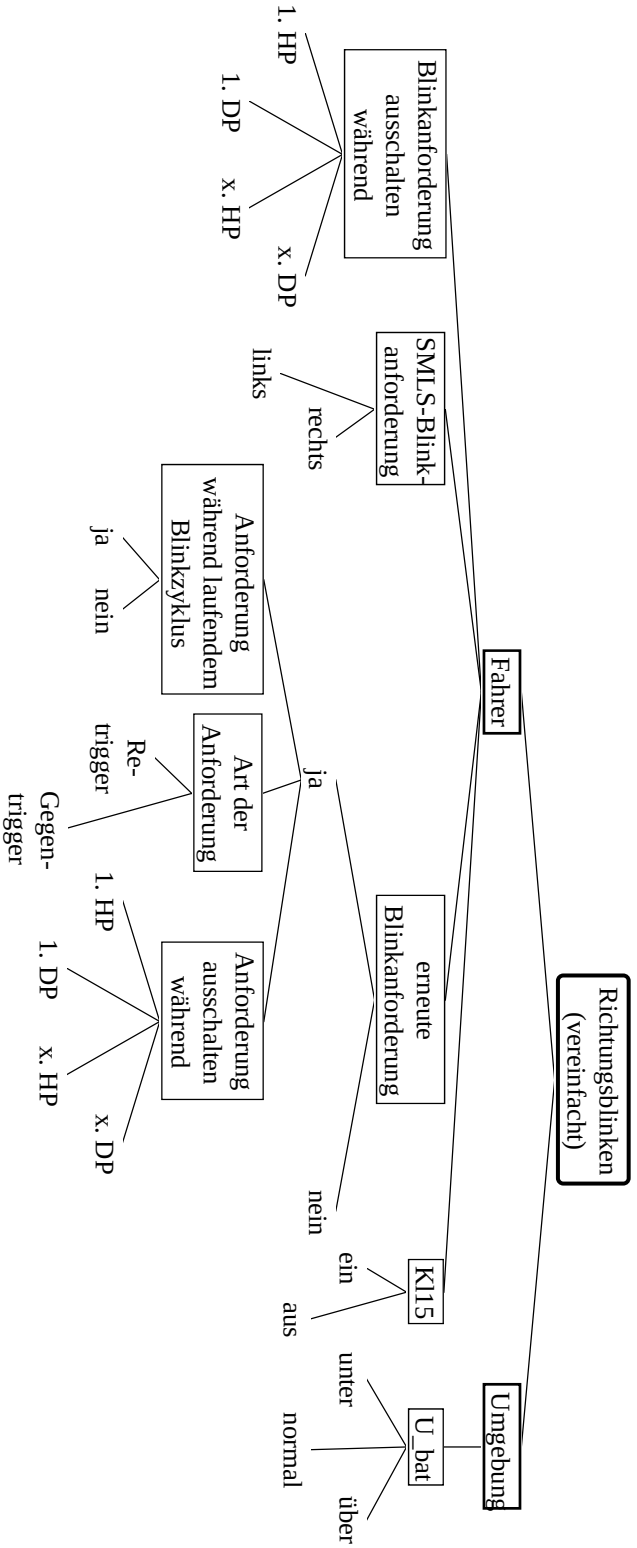


Abbildung 6.8: Vollständiger CTM–Klassifikationsbaum der vereinfachten Funktion Richtungsblinken

- 1. DP
- x. HP
- x. DP

Abb. 6.8 zeigt den so erweiterten und vollständigen[9] Klassifikationsbaum.

6.2.2.3 Testfallermittlung

Nachdem der Klassifikationsbaum vollständig erstellt ist, kann der Schritt der Testfallermittlung folgen. Die hierbei erstellten Testfälle werden auch als logische bzw. anforderungsorientierte Testfälle bezeichnet. Logische Testfälle beschreiben den Testfall auf einer hohen Abstraktionsebene und beinhalten daher keine konkreten Testabläufe. Eine detaillierte Beschreibung des Testfalls kann jedem Testfall in textueller Form im Eigenschaftsbereich des CTE XL hinzugefügt werden. Abb. 6.9 zeigt den vollständigen Klassifikationsbaum inkl. Kombinationstabelle mit sechs beispielhaften, logischen Testfällen.

Testfall a_Ausschalten_Blinkfunktion_links_1HP beschreibt einen Testfall, der bei den Randbedingungen Kl15→ein sowie U_bat→normal stattfindet. Dabei wird eine Blinkanforderung vom SMLS für Linksblinken während der ersten Hellphase abgebrochen. Eine erneute Blinkanforderung tritt nicht auf.

Testfall b_Ausschalten_Blinkfunktion_rechts_1HP unterscheidet sich vom vorangegangenen Testfall lediglich durch eine andere Auswahl der Klassifikation SMLS-Blinkanforderung. Statt nach links wird hier nach rechts geblinkt.

Testfall c_Blinkerwechsel_links_rechts_1HP beschreibt den Abbruch einer Blinkanforderung nach links durch eine Blinkanforderung nach rechts in der ersten Hellphase. Der Abbruch der erneuten Blinkanforderung erfolgt wiederum in der ersten Hellphase.

Während der vierte Testfall ähnlich aufgebaut ist, fällt in den letzten zwei Testfällen ein neues Symbol (graues Dreieck) auf. Dabei handelt es sich um die so genannte „don't care"–Markierung. Sie drückt aus, dass die markierten Äquivalenzklassen ganz bewusst an einer Stelle nicht beachtet werden. Für den Testfall sind sie daher nicht relevant. Im vorliegenden Fall wird die „don't care"–Markierung in der Klassifikation Blinkanforderung ausschalten während verwendet. Bei Berücksichtigung der ausgewählten Äquivalenzklasse Kl15→aus wird der Blinker nach Anforderung zwei aus Kap. 6.2.1 deaktiviert bzw. gar nicht erst aktiviert. Daraus folgt, dass die Blinkanforderung auch nicht in einer Hell– oder Dunkelphase ausgeschaltet werden kann. Solche Abhängigkeiten können mittels der in Kap. 5.4 bereits

[9] Die Vollständigkeit eines Klassifikationsbaumes wird durch den Ersteller des Klassifikationsbaumes festgelegt. Natürlich können jederzeit weitere Klassifikationen in den Baum hinzugefügt werden. In unserem Fall könnte z.B die Anzahl der erneuten Anforderungen relevant sein. Für den hier betrachteten Fall wird der Klassifikationsbaum in der vorliegenden Form jedoch als vollständig betrachtet.

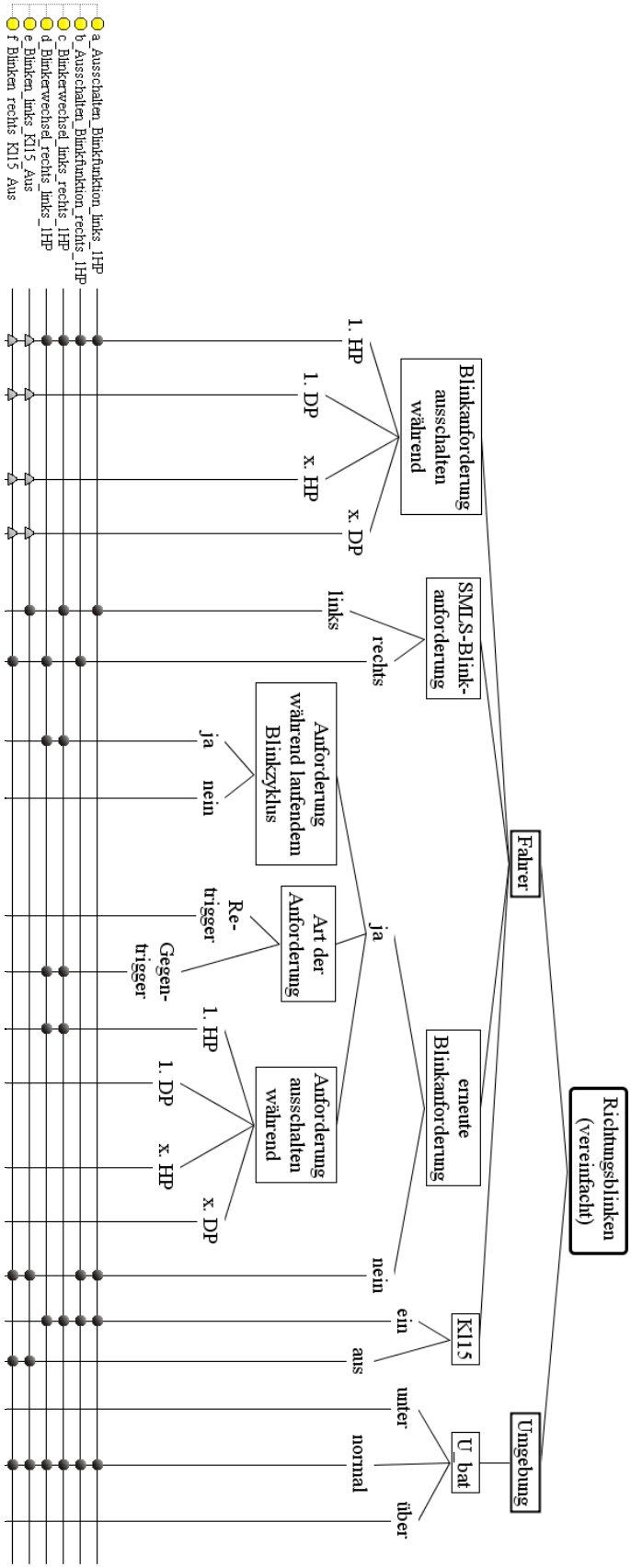


Abbildung 6.9: Vollständiger CTM–Klassifikationsbaum der vereinfachten Funktion Richtungsblinken inkl. Kombinationstabelle

angesprochenen Abhängigkeitsregeln im CTE XL in den Klassifikationsbaum einbezogen werden.

Damit sind sechs beispielhafte Testfälle für das durch die Anforderungen aus Kap. 6.2.1 beschriebene SUT, die vereinfachte Funktion Richtungsblinken, erstellt. Die in Kap. 5.1.3 definierten Überdeckungskriterien haben für den in Abb. 6.9 dargestellten Klassifikationsbaum folgende Werte:

$$CTC_{min} = \frac{actclass}{maxclass} = \frac{10}{20} = 0.5$$

$$CTC_{max} = \frac{actfall}{maxfall} = \frac{6}{4 \cdot 2 \cdot (2 \cdot 2 \cdot 4 + 1) \cdot 2 \cdot 3} = \frac{6}{816} = 0.007$$

Der Wert $CTC_{min} = 0.5$ deutet auf die Tatsache hin, dass genau die Hälfte der spezifizierten Blattklassen bisher bei Testfällen nicht berücksichtigt wurde. Weitere Testfälle sollten daher definiert werden, die diese bisher nicht berücksichtigten Äquivalenzklassen enthalten. Erst bei Berücksichtigung aller existierenden Blattklassen erfüllt der Test dieser Funktion das von *Grimm* (1995) definierte Minimalkriterium ($CTC_{min} = 1$). Zur Erfüllung der von *Büchner* (2002) angegebenen Daumenregel (vgl. Kap. 5.1.3) sind ca. 20 Testfälle notwendig, die zudem das Minimalkriterium erfüllen müssen. Auf diese Weise können messbare und realistische Testziele sowie entsprechende Testendekriterien sehr einfach definiert werden.

Die Kombinatorik lässt insgesamt 816 verschiedene Klassenkombinationen in der Kombinationstabelle zu, so dass CTC_{max} bei sechs Testfällen einen Wert von 0.007 annimmt. Anhand dieser Zahl wird die bereits in Kap. 5.1.3 angedeutete, schwache Aussagekraft des Überdeckungsmaßes CTC_{max} sehr gut deutlich.

6.2.3 Anwendung der CTM/ES

Aufbauend auf den mittels der CTM definierten logischen Testfällen kommt nun die CTM/ES zum Einsatz. Nach Anpassung des CTM−Klassifikationsbaumes werden die logischen Testfälle in Testsequenzen überführt. Jede Testsequenz besteht aus einzelnen Testschritten und gibt somit den konkreten Testablauf vor.

Auch bei der Anwendung der CTM/ES werden wieder die drei Schritte der Classification−Tree Method angewendet, wobei statt Testfällen nun Testsequenzen in der Kombinationstabelle definiert werden. Der letzte Schritt der Methode wird daher in Testsequenzermittlung[10] umbenannt.

[10] Eigentlich findet keine Ermittlung von Testsequenzen statt, da diese bereits als Testfälle mittels der CTM ermittelt wurden. Vielmehr handelt es sich hier um die Konkretisierung der ermittelten Testfälle. Daher könnte auch der Begriff Testfallkonkretisierung oder Testsequenzdefinition für den letzten Schritt der CTM/ES im vorliegenden Fall verwendet werden. Da die CTM/ES jedoch auch eigenständig angewendet werden kann, wird der Begriff Testsequenzermittlung beibehalten.

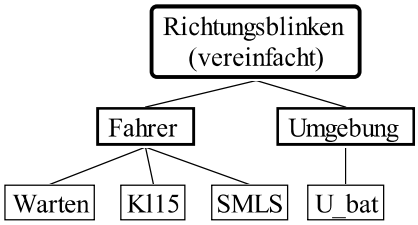


Abbildung 6.10: CTM/ES−Klassifikationen der vereinfachten Funktion Richtungsblinken

6.2.3.1 Definition des SUT

Das SUT ist bereits durch die Anwendung der CTM definiert worden. Es handelt sich nach wie vor um die vereinfachte Funktion Richtungsblinken. Der Name `Richtungsblinken (vereinfacht)` wird daher übernommen und findet sich im Kopf des Klassifikationsbaum−Diagramms in Abb. 6.10 wieder.

6.2.3.2 Klassifikation des Eingabedatenraums

Zunächst erfolgt die Transformation[11] des CTM−Klassifikationsbaumes in einen CTM/ES−Klassifikationsbaum.

Definition von testrelevanten Gesichtspunkten Einige Gesichtspunkte können aus dem bereits erstellten Klassifikationsbaum ohne Änderungen übernommen werden. In unserem Fall sind dies neben den zwei Kompositionen

- `Fahrer`
- `Umgebung`

die beiden Klassifikationen

- `U_bat`
- `Kl15`

Mit einer kleinen Anpassung der betrachteten Äquivalenzklassen kann auch die bereits bekannte Klassifikation `SMLS-Blinkanforderung` übernommen werden. Die Anpassung beinhaltet das Hinzufügen einer neuen Äquivalenzklasse, die der Rücknahme einer Blinkanforderung durch das SMLS entspricht. Der Name der Klassifikation wird zu `SMLS` umbenannt.

[11] Solch ein Transformationsprozess wird z.B. in *Conrad et al.* (2005) näher betrachtet.

Komplizierter gestaltet sich die Übernahme der Klassifikationen, die den zeitlichen Ablauf beschreiben. Der mit Hilfe der Classification–Tree Method erstellte Klassifikationsbaum (s. Abb. 6.9) beinhaltet hierfür u.a. die zwei Klassifikationen:

- `Blinkanforderung ausschalten während`
- `Anforderung während laufendem Blinkzyklus`

Beide Klassifikationen weisen auf die Wichtigkeit des Zeitpunktes einer bestimmten Aktion hin. Soll die Blinkanforderung zu einem gewissen Zeitpunkt (z.B. während einer Hellphase) ausgeschaltet werden, so muss dieser Zeitpunkt spezifiziert werden können. Solch eine zeitliche Abhängigkeit kann auch als die Aktion „Warten des Fahrers auf den richtigen Moment“ interpretiert werden. In den Klassifikationsbaum (s. Abb. 6.10) wird deshalb unterhalb der Komposition `Fahrer` eine Klassifikation mit dem Namen `Warten` aufgenommen.

Vollständige, disjunkte Zerlegung in Äquivalenzklassen Nach Schritt 2b der CTM/ES werden die alten und neuen Gesichtspunkte nun vollständig und disjunkt in Äquivalenzklassen zerlegt.

Die Klassifikation `Warten` nimmt, wie schon erwähnt, eine Sonderstellung unter den Klassifikationen ein. Sie beschreibt einen Zeitraum, in dem der Fahrer ganz bewusst nichts unternimmt, um auf ein bestimmtes Ereignis zu warten. Relevante Ereignisse in unserem Beispiel sind z.B. das Auftreten einer Hell– bzw. Dunkelphase in den beobachteten Blinkzyklen sowie das Ende des letzten Blinkzyklus.

Analog zu den vier Äquivalenzklassen der Klassifikation `Blinkanforderung ausschalten während` aus Abb. 6.9 werden vier entsprechende Äquivalenzklassen eingeführt:

- `auf 1. HP` 0–400ms
- `auf 1. DP` 400–800ms
- `auf x. HP`
- `auf x. DP`

Jede der vier Äquivalenzklassen beschreibt eine spezielle Wartezeit[12]. Genauer gesagt repräsentiert jede Äquivalenzklasse ein bestimmtes Zeitintervall. Hinter der x.–ten Hellphase verbirgt sich z.B. das Zeitintervall $[800ms, 1200ms[$, $[1600ms, 2000ms[$, $[2400ms, 2800ms[$, $[3200ms, 3600ms[$, usw. Zusätzlich wird eine weitere Äquivalenzklasse

[12] Die Vorgabe definierter Zeitpunkte kann, wie in Kap. 5.2 beschrieben, auch durch das Hinzufügen von Stützstellen an jeden Testschritt realisiert werden. Dadurch könnte auf die Klassifikation `Warten` verzichtet werden. Im vorliegenden Fall wird diese Variante jedoch nicht verwendet.

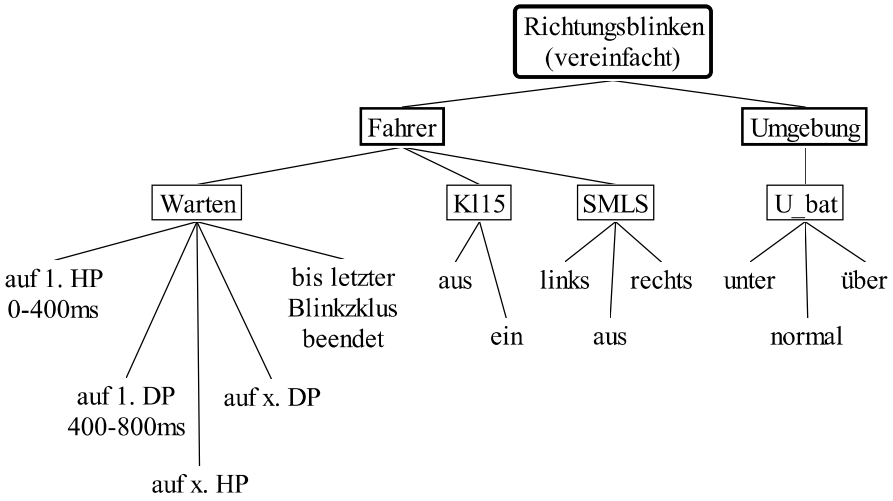


Abbildung 6.11: Vollständiger CTM/ES−Klassifikationsbaum der vereinfachten Funktion Richtungsblinken

- `bis letzter Blinkzyklus beendet`

der Klassifikation `Warten` eingeführt, um die Berücksichtigung einer erneuten Blinkanforderung nach dem Ende des letzten Blinkzyklus der ersten Blinkanforderung zu ermöglichen.

Die Einführung dieser fünften Äquivalenzklasse verstößt zwar gegen die Forderung nach Disjunktheit der Äquivalenzklassen einer Klassifikation, da der Zeitbereich bereits durch die ersten vier Äquivalenzklassen vollständig und disjunkt aufgeteilt wird. Aufgrund der Sonderstellung der Klassifikation `Warten` (s.o.) kann dies jedoch akzeptiert werden.

Die Klassifikationen `Kl15` und `U_bat` können all ihre Blätter

- `ein`
- `aus`

und

- `unter`
- `normal`
- `über`

ohne Änderung übernehmen.

Die Klassifikation `SMLS` unterteilt sich, gemäß den drei möglichen Positionen des Blinkerhebels für Linksblinken, Ruhestellung und Rechtsblinken, dagegen in die folgenden drei Äquivalenzklassen:

- links
- aus
- rechts

Abb. 6.11 zeigt den entsprechenden Klassifikationsbaum.

Iteration Der nächste Schritt beinhaltet die Iteration von Klassifikationen auf Äquivalenzklassen, also eine weitere Unterteilung existierender Äquivalenzklassen falls möglich bzw. nötig. Im betrachteten Beispiel ist eine weitere Unterteilung nicht notwendig, da bereits alle identifizierten Testinhalte in der in Abb. 6.11 abgebildeten Klassifikationsbaumstruktur berücksichtigt sind.

6.2.3.3 Testsequenzermittlung

Als dritter und letzter Schritt der CTM/ES wird nun die Testsequenzermittlung durchgeführt. Dabei wird jeder mit Hilfe der CTM ermittelte Testfall in eine Testsequenz überführt und auf diese Weise konkretisiert. Hierfür kommt die Kombinationstabelle zum Einsatz, deren Spalten die Blätter des Klassifikationsbaumes und die Zeilen die Testsequenzen bzw. Testschritte darstellen (vgl. Kap. 5.2). Ist ein Kreuzungspunkt von Spalte mit Zeile markiert, so wird die entsprechende Äquivalenzklasse bei diesem Testschritt berücksichtigt. Ein Testschritt setzt sich somit aus einer geeigneten Auswahl von Blättern des Klassifikationsbaumes zusammen. Eine zum aufgeführten Beispiel passende Kombinationstabelle ist in Abb. 6.12 dargestellt. Transitionen haben hierbei lediglich die Aufgabe, die relevante Änderung zwischen den einzelnen Testschritten deutlicher hervorzuheben. Die Möglichkeit zur Spezifikation des Übergangs (vgl. Kap. 5.2) wird nicht verwendet.

Die erste Testsequenz `a_Ausschalten_Blinkfunktion_links_1HP` in Abb. 6.12 geht von folgender Startbedingung (1. Testschritt) aus. Die `Kl15` befindet sich in Position `ein`, das `SMLS` in Neutralstellung und die Batteriespannung `U_bat` im Bereich `normal`. Ausgehend von dieser Vorbedingung wird der Blinkerhebel (`SMLS`) in Position für Linksblinken gesetzt (2. Testschritt). Dem folgt eine Wartezeit von $0-400ms$ (3. Testschritt) sowie die Rückstellung des Blinkerhebels in Neutralstellung (4. Testschritt). Durch diese Wartezeit wird die Rücknahme der Blinkanforderung während der ersten Hellphase realisiert.

In der zweiten Testsequenz gilt die gleiche Vorbedingung wie in der ersten Testsequenz. Davon ausgehend wird der Blinkerhebel in Position für Rechtsblinken gesetzt. Es wird wieder einen Zeitraum von $0-400ms$ gewartet. Im letzten Testschritt erfolgt schließlich wieder die Rückstellung des Blinkerhebels.

Testsequenz `c_Blinkerwechsel_links_rechts_1HP` beginnt wie Testsequenz `a_Ausschalten_Blinkfunktion_links_1HP`. Erst ab Testschritt `5. SMLS-RB rechts einschalten` besteht ein Unterschied, der eine Gegentriggeranforderung beschreibt.

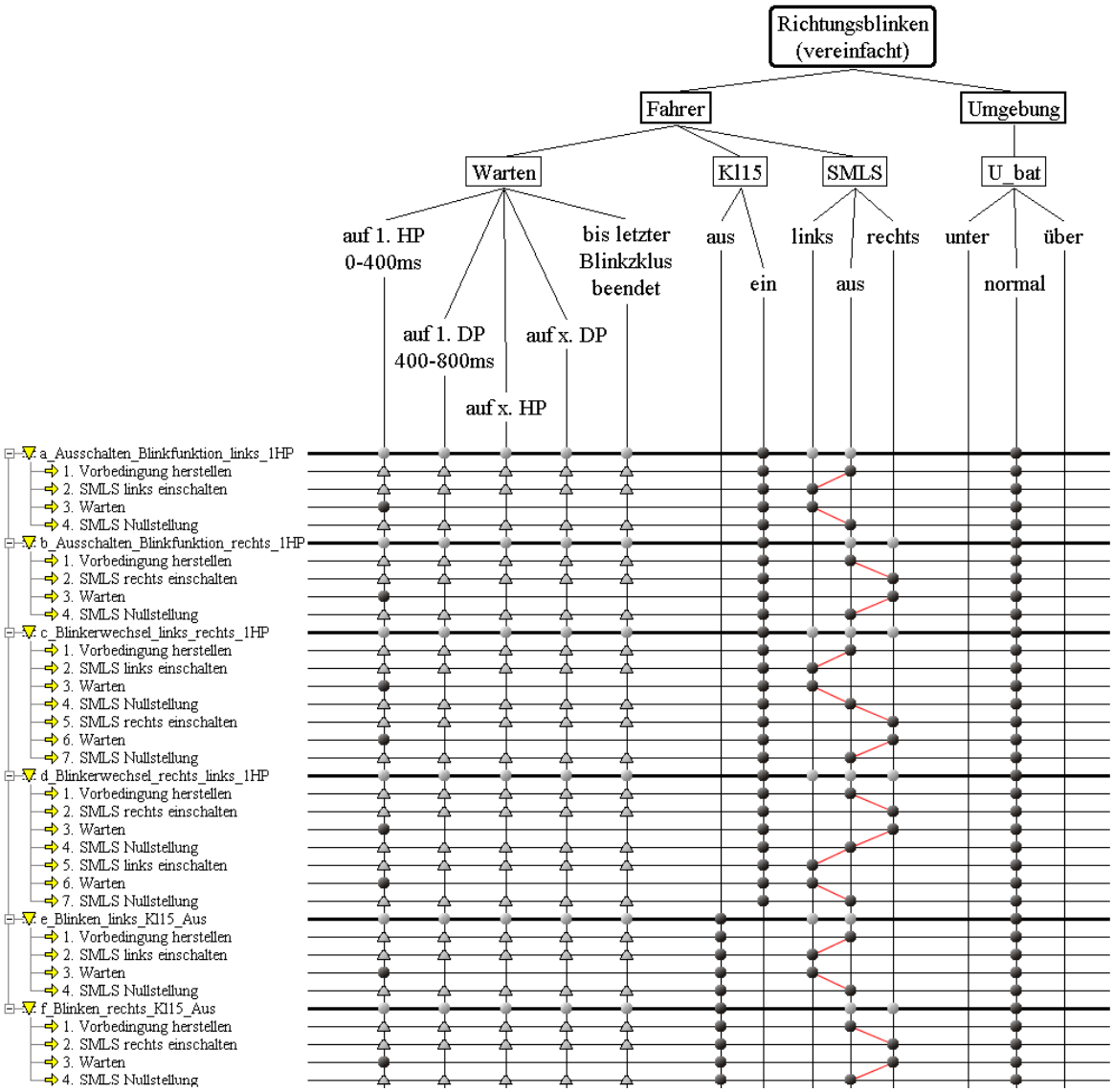


Abbildung 6.12: CTM/ES−Klassifikationsbaum für die vereinfachte Funktion Richtungsblinken inkl. Kombinationstabelle. Jeder logische Testfall aus Abb. 6.9 ist hierbei durch eine entsprechende Testsequenz detailliert bzw. konkretisiert.

Die vierte Testsequenz ähnelt der dritten Testsequenz und wird deshalb nicht näher beschrieben. Auch die Testsequenzen fünf und sechs sind ähnlich aufgebaut.

Jede in Abb. 6.12 spezifizierte Testsequenz beschreibt den genauen Ablauf, des in Abb. 6.9 definierten übergeordneten, logischen Testfalls. Ausgehend von diesen Testsequenzen ist eine automatische Überführung der darin spezifizierten Testinhalte als ausführbare TestCases in die bei der AUDI AG eingesetzte Testautomatisierung EXAM (s. Anh. B) möglich. Der hierfür entwickelte Prototyp (ctmes2exam−Schnittstelle) wird in Anh. C aufbauend auf die-

sem Beispiel vorgestellt.

Weitere Testsequenzen sollten weitere, bisher nicht berücksichtigte, Testinhalte enthalten wie z.B. `U_bat→über` und `U_bat→unter`. Auch die Berücksichtigung aller Äquivalenzklassen der Klassifikation `Warten` in weiteren Testsequenzen ist anzustreben. Schließlich wurden diese Äquivalenzklassen beim Aufbau des Klassifikationsbaumes als relevant eingestuft. Dieser Sachverhalt wird durch das Erreichen des Minimalkriteriums $CTC_{min} = 1$ beschrieben. Da jede einzelne Kombinationsmöglichkeit von unterschiedlichen Ausgangsbedingungen aufgerufen werden kann, stellt das Erreichen von $CTC_{min} = 1$ jedoch kein repräsentatives Testendekriterium für CTM/ES−Klassifikationsbäume dar. Wie bereits in Kap. 5.2.1 erläutert, besitzt das existierende Überdeckungsmaß CTC_{max} (vgl. Kap. 5.1.3) ebenfalls wenig Aussagekraft für CTM/ES−Klassifikationsbäume, weil der zeitliche Ablauf in den Testsequenzen dabei nicht berücksichtigt wird.

7 Konzeptvalidierung durch Praxistest

Das im vorangegangenen Kapitel vorgestellte neuartige Konzept zur Integration der CTM und CTM/ES in den geschilderten Testerstellungsprozess (vgl. Kap. 3.6) wird im vorliegenden Kapitel anhand eines Praxistests validiert. Dabei wird zunächst das Konzept des Praxistests und anschließend seine Durchführung vorgestellt. Abschließend werden die Ergebnisse des Praxistests sowie die daraus abgeleiteten Konsequenzen für die ursprünglich angestrebte Integration der beiden systematischen Testverfahren in den Testerstellungsprozess präsentiert.

7.1 Konzept des Praxistests

Ziel des Praxistests ist die Bewertung der vorgestellten Vorgehensweise (vgl. Kap. 6.1) im Hinblick auf deren Praxistauglichkeit unter den Randbedingungen des bei der AUDI AG existierenden Testerstellungsprozesses. Aus diesem Grund werden Testersteller (erfahrene und unerfahrene), als potenziell künftige Anwender der angestrebten Vorgehensweise, für die Durchführung und Bewertung der Praxistests herangezogen.

Der Praxistest unterteilt sich in einen Kurzzeitpraxistest und einen Langzeitpraxistest und ist in den nächsten zwei gleichnamigen Abschnitten ausführlich beschrieben. Abb. 7.1 zeigt in einer Gegenüberstellung den genauen Ablauf der zwei unterschiedlichen Praxistests.

7.1.1 Kurzzeitpraxistest

Einige Tage vor dem Beginn des Kurzzeitpraxistests wird mit dem jeweiligen, erfahrenen Testersteller die Auswahl eines bereits geprüften und für den jeweiligen Bereich (Antrieb, Infotainment, Komfort (vgl. Kap. 3.3)) repräsentativen SUT vorgenommen. Für diese Durchführung der Kurzzeitpraxistests an bereits geprüften SUT durch erfahrene Testersteller gibt es folgende drei Gründe:

- Der Testersteller ist bereits mit den relevanten Testinhalten am SUT vertraut. Der Praxistest kann daher ohne eine lange Systemanalyse bzw. Einarbeitung in das SUT durchgeführt werden.

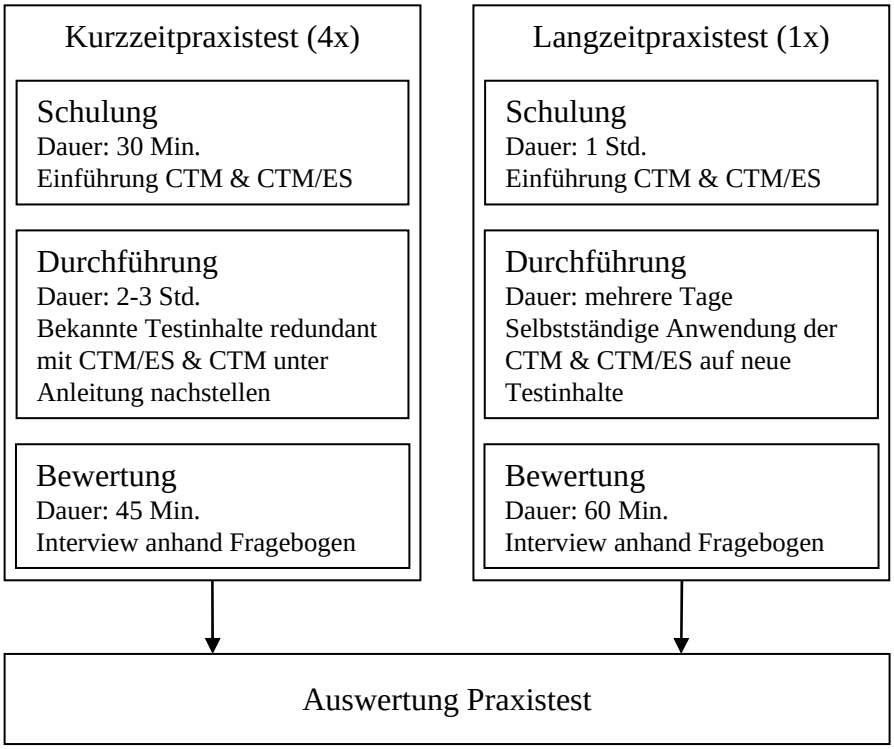


Abbildung 7.1: Ablauf des Praxistests

- Zudem können die bestehenden, herkömmlich abgeleiteten Testfälle vom Testersteller selbst anhand der existierenden Überdeckungsmaße der zwei Methoden bewertet werden.
- Außerdem können die Testersteller aufgrund ihrer langjährigen Erfahrung die Anwendbarkeit der Methoden sowie der in Kap. 6.1 vorgeschlagenen Vorgehensweise im existierenden Testerstellungsprozess (vgl. Kap. 3.6) beurteilen.

Jeder Kurzzeitpraxistest hat einen Zeitrahmen von insgesamt ca. vier Stunden und unterteilt sich in die drei Phasen (s. Abb. 7.1):

1. Schulung (30 Min.)
2. Durchführung (2–3 Std.)
3. Bewertung (45 Min.)

In der ersten halben Stunde werden dem ausgewählten Testersteller die zwei Testverfahren inkl. CTE XL vorgestellt, ebenso wie die angestrebte Integration der Verfahren in den bestehenden Testprozess nach der in Kap. 6.1 vorgestellten Vorgehensweise. Dieser Schulungsphase folgt die Durchführungsphase des Praxistests. Darin stellt der Testersteller unter

Betreuung des Versuchsleiters bereits implementierte TestCases mittels der beiden Verfahren nochmals nach. Ziel dieser Durchführungsphase ist es, festzustellen, ob alle bereits spezifizierten Testinhalte mit Hilfe der CTM/ES sowie der CTM nachgestellt werden können, und welche Vor– und Nachteile die Testersteller zum konventionellen Vorgehen dabei sehen. Der Durchführungsphase folgt die Bewertungsphase, in der jeder Testersteller anhand eines zuvor erstellten Fragebogen interviewt wird.

7.1.2 Langzeitpraxistest

Der Langzeitpraxistest unterteilt sich ebenso wie der Kurzzeitpraxistest in die drei Phasen (s. Abb. 7.1):

1. Schulung (60 Min.)
2. Durchführung (mehrere Tage)
3. Bewertung (60 Min.)

Der Langzeitpraxistest erstreckt sich im Gegensatz zum Kurzzeitpraxistest über mehrere Tage. Ein unerfahrener Testersteller wird dabei zunächst in einer ca. einstündigen Schulungsphase ebenfalls mit den beiden Testverfahren, dem CTE XL sowie mit der in Kap. 6.1 vorgestellten Vorgehensweise vertraut gemacht. In der sich anschließenden mehrtägigen Durchführungsphase werden die Testverfahren vom ausgewählten Testersteller aufeinander aufbauend auf ein bisher noch nicht geprüftes SUT, nach der vorgestellten Vorgehensweise, angewandt. Zum Abschluss erfolgt wiederum eine Bewertungsphase, in der ein leicht abgewandelter Fragebogen zum Einsatz kommt.

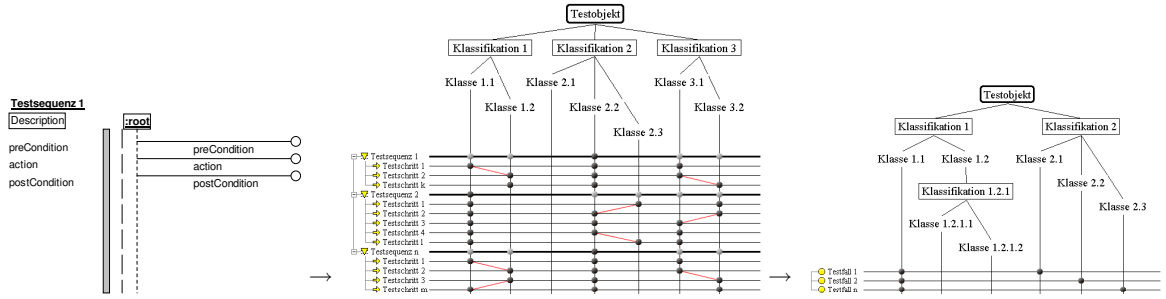
Ziel des Langzeitpraxistests ist die Evaluierung der vorgeschlagenen Vorgehensweise anhand eines konkreten Praxisbeispiels sowie die Untermauerung der aus den Kurzzeitpraxistests abgeleiteten Ergebnisse. Durch den Einsatz der Methoden von einem unerfahrenen Testersteller kann zudem die Erlernbarkeit der Methoden beurteilt werden.

7.2 Durchführung des Praxistests

Für den Kurzzeitpraxistest standen insgesamt vier erfahrene Testersteller aus den drei Bereichen Antrieb, Infotainment und Komfort zur Verfügung, während der Langzeitpraxistest von einem unerfahrenen Testersteller im Komfortbereich durchgeführt wurde.

7.2.1 Kurzzeitpraxistest

Nachdem dem Testersteller die beiden Testverfahren sowie die geplante Einbindung in den Testprozess bzw. in EXAM vorgestellt wurde, begann die Durchführung des jeweiligen Praxistests. Dabei wurde zunächst ein Klassifikationsbaum aus bereits implementierten TestCases des ausgewählten SUT im Sinne der CTM/ES erstellt. Dieser Klassifikationsbaum diente anschließend als Kopf der Kombinationstabelle in der die bereits implementierten TestCases als Testsequenzen nachgestellt wurden.



(a) EXAM TestCases (b) Testsequenzen (CTM/ES) (c) Logische Testfälle (CTM)

Abbildung 7.2: Vorgehen beim Kurzzeitpraxistest

Im nächsten Schritt folgte die Übertragung der erstellten Testsequenzen in logische Testfälle im Sinne der CTM. Auch hierbei wurde zunächst ein Klassifikationsbaum erstellt und danach die Kombinationstabelle mit Inhalten gefüllt. Abb. 7.2 veranschaulicht dieses Vorgehen[1] beim Kurzzeitpraxistest.

Nachdem beide Klassifikationsbäume erstellt und die Kombinationstabellen mit Inhalten gefüllt waren, erfolgte die Bewertung des durchgeführten Praxistests durch den Testersteller (siehe Kap. 7.2.3).

7.2.2 Langzeitpraxistest

Bei dem durchgeführten Langzeitpraxistest orientierte sich der Testersteller dagegen direkt an der in Kap. 6.1 vorgestellten Vorgehensweise und wendete im ersten Schritt die CTM auf das ausgewählte SUT an (vgl. Kap. 5.1.1). Zunächst wurde ein CTM−Klassifikationsbaum erstellt und anschließend logische Testfälle anhand der vorhandenen Anforderungsdokumente definiert.

In einem zweiten Schritt wurde der CTM−Klassifikationsbaum in einen CTM/ES−Klassifikationsbaum umgewandelt. Anschließend wurden die logischen Testfälle

[1] Zu beachten ist, dass das Vorgehen beim Kurzzeitpraxistest eine Art Reverse Engineering darstellt und sich somit von der angestrebten Vorgehensweise in der Reihenfolge der einzelnen durchzuführenden Schritte unterscheidet.

in Testsequenzen überführt und so ein konkreter Testablauf vorgegeben. Visualisiert wird das Vorgehen beim Langzeitpraxistest in Abb. 6.4 auf S. 67.

Während der gesamten Durchführungsphase des Langzeitpraxistests unterstützte der Versuchsleiter den Testersteller nur bei Bedarf. Die abschließende Bewertung wurde, wie auch schon beim Kurzzeitpraxistest, anhand eines Fragebogens in Form eines Interviews durchgeführt.

7.2.3 Bewertung

Für die Bewertung der durchgeführten Praxistests kam ein zuvor erstellter Fragebogen zum Einsatz. Der Fragebogen für den Langzeitpraxistest war dabei eine leichte Abwandlung des Fragebogens des Kurzzeitpraxistests. Der Fragenkatalog wurde mit dem Testersteller nach der Durchführungsphase des Praxistests in Form eines Interviews abgearbeitet. Dabei kam keine starre Bewertungsskala zum Einsatz. Der Testersteller hat auf jede Frage mit eigenen Worten geantwortet. Diese wurden stichpunktartig vom Versuchsleiter notiert.

Einige Fragen aus den zwei Fragebögen sind im Folgenden aufgeführt. Sie wurden bei Bedarf jeweils einmal bezogen auf die CTM sowie einmal bezogen auf die CTM/ES gestellt.

- Wie schnell haben Sie sich mit der/den Methode/n vertraut gemacht?
- War es möglich, den Eingabedatenraum des SUT vollständig auf einen Klassifikationsbaum abzubilden? Welche Situationen waren dabei schwierig bzw. nicht abbildbar?
- Hat der Einsatz der Methode/n bisher nicht berücksichtigte Testinhalte aufgedeckt?
- Können Sie nach Anwendung der Methode/n den erreichten Testumfang bewerten?
- Erleichtert der Einsatz der Methode/n die Kommunikation mit anderen Personen über den spezifizierten Testinhalt?
- Bringt der Einsatz der Methode/n Vor– bzw. Nachteile gegenüber Ihrer bisherigen Vorgehensweise beim Spezifizieren von TestCases?

Jeder Testersteller hatte zudem die Gelegenheit, zum Abschluss des Fragebogens eigene Bemerkungen anzugeben.

7.3 Ergebnisse des Praxistests

Die folgende Präsentation der Ergebnisse erfolgt für Kurzzeitpraxistest und Langzeitpraxistest gemeinsam. Aufgrund der größeren Teilnehmeranzahl liegt die Gewichtung der Auswertung deutlich auf der Seite der Kurzzeitpraxistests. Der Langzeitpraxistest bestätigt jedoch

die Ergebnisse der Kurzzeitpraxistests in sehr hohem Maße und trägt damit sehr zur Untermauerung der Bewertung der Kurzzeitpraxistests bei.

Zur Auswertung der Bewertung der Testersteller kommt eine 6–stufige Skala (+++ bis −−−) zum Einsatz. Ein +/− steht dabei für einen als positiv/negativ bewerteten Punkt. Die Anzahl der Symbole spiegelt die Häufigkeit der Nennungen durch die Testersteller wider. Ein Symbol bedeutet lediglich die einmalige Nennung, zwei Symbole stehen für zwei– und dreimalige Nennungen. Drei Symbole kennzeichnen Punkte, die von vier bzw. fünf Testerstellern genannt wurden.

Die folgenden zwei Unterkapitel enthalten die Ergebnisse unterteilt nach CTM und CTM/ES. Neben der Vorstellung der Ergebnisse werden die Ergebnisse darin mit anderen durchgeführten Untersuchungen zum Einsatz der Classification–Tree Method verglichen und im Hinblick auf den geplanten Einsatz bewertet.

7.3.1 Ergebnisse zum Einsatz der CTM

Die Auswertung der Fragebögen ergibt für die CTM die in Tab. 7.1 dargestellten Ergebnisse.

Bewertung	Stichpunkt
+++	Erlernbarkeit
+++	Übersichtlichkeit (Klassifikationsbaum)
+++	Kommunikationsmittel
++	grafische Repräsentation
++	Strukturierung
++	neue Testinhalte
++	existierende Überdeckungskriterien
−−	Unübersichtlichkeit bei komplexen Problemen
−−	Dynamik der Testentwicklung eingeschränkt
−−	höherer zeitlicher Aufwand
−−	zeitliches Verhalten
−−−	noch ein Tool
−−−	fehlendes Sollverhalten

Tabelle 7.1: Ergebnisse CTM

+++ Erlernbarkeit Alle befragten Testersteller haben ausgesagt, dass sie sich sehr schnell mit der CTM vertraut gemacht haben. Gleichzeitig geben die Testersteller jedoch an, dass ein gewisser Grad an Erfahrung notwendig ist, um die Methode selbstständig anzuwenden. Konkrete Praxisbeispiele sind dabei nach ihrer Aussage für die Anschauung bzw. Erfahrungsbildung sehr förderlich.

+++ Übersichtlichkeit (Klassifikationsbaum) Positiv hervorzuheben ist weiterhin die grafische Darstellungsweise des Klassifikationsbaumes, die den Testerstellern eine gute Übersicht über den Parameterraum bzw. die möglichen Randbedingungen des SUT liefert. Ihre Kreativität für das Auffinden testrelevanter Klassifikationen wird durch diese Visualisierung stark gefördert.

+++ Kommunikationsmittel Die Testersteller sehen den Einsatz der CTM bzw. den dabei entstehenden Klassifikationsbaum inkl. Kombinationstabelle als gutes Kommunikationsmedium, um anderen Personen die spezifizierten Testinhalte schnell und einfach zu vermitteln.

++ grafische Repräsentation, Strukturierung Ebenso förderlich sehen die Testersteller die Anwendung der CTM für den Überblick über ähnliche Testinhalte, damit verbunden für die spätere Strukturierung der anforderungsorientierten Testfälle und schließlich zur Identifikation redundanter Testinhalte.

++ neue Testinhalte Mehrere bisher nicht berücksichtigte Testinhalte, die meist Randbedingungen von Testfällen darstellten, konnten aufgrund der guten Visualisierungsform der Testinhalte im Klassifikationsbaum im Rahmen der Kurzzeitpraxistests identifiziert werden.

++ existierende Überdeckungskriterien Auch die Möglichkeit der Bewertung des erreichten Testumfangs anhand der in Kap. 5.1.3 definierten Überdeckungskriterien sehen die Testersteller als positiv an.

−− Unübersichtlichkeit bei komplexen Problemen Trotz der vorhandenen Strukturierungsmöglichkeiten von Klassifikationsbäumen (vgl. Kap. 5.1.2) und der guten Übersicht, die der Klassifikationsbaum bietet, haben einige Testersteller Bedenken, dass ein Klassifikationsbaum ab einer gewissen Breite/Tiefe bzw. bei komplexeren Problemen unübersichtlich werden könnte.

−− Dynamik der Testentwicklung eingeschränkt Bedenken haben einige Testersteller weiterhin bzgl. der Integration der Verfahren bzw. des CTE XL in den vorhandenen Testprozess. Durch den Einsatz eines weiteren Tools könnte die Dynamik der Testentwicklung leiden[2] (s. auch Punkt „noch ein Tool“).

−− höherer zeitlicher Aufwand Negativ bewertet wurde zudem der höhere zeitliche Aufwand, der bei der Erstellung eines Klassifikationsbaumes investiert werden muss. Im Gegensatz zur häufig vorgefundenen Ad–hoc Vorgehensweise, bei der die ersten Testfälle sehr schnell spezifiziert werden können, lassen sich bei der vorgeschlagenen Vorgehensweise erste Testfälle erst spezifizieren, nachdem der entsprechende Klassifika-

[2] Manchmal ist es notwendig schnelle Änderungen an einem vorhandenen TestCase vorzunehmen, um z.B. den TestCase bei neuen Randbedingungen auszuführen. Dabei ist der Einsatz eines weiteren Tools, verbunden mit der sich ergebenden Toolkette möglicherweise hinderlich.

tionsbaum erstellt und für vollständig erklärt wurde. Ist der Klassifikationsbaum jedoch einmal erstellt, können sehr schnell weitere Tests spezifiziert oder vorhandene Tests angepasst werden.

−− **zeitliches Verhalten** Ein weiterer Punkt, der von mehreren Testerstellern bemängelt wurde, ist die schwierige bzw. nur mit Erfahrung durchführbare Abbildung von zeitlichem Verhalten in eine Klassifikationsbaumstruktur. Der dabei entstehende Baum kann bei einem komplexen SUT ebenfalls sehr komplex werden, so dass die bereits angesprochene Übersichtlichkeit des Klassifikationsbaumes verloren gehen könnte.

−−− **noch ein Tool** Im derzeitigen Testprozess sind bereits zahlreiche Tools im Einsatz (s. Abb. B.4). Durch den Einsatz eines weiteren Tools erhöht sich die Komplexität des Testprozesses. Von entscheidender Bedeutung ist daher eine benutzerfreundliche Integration der Testverfahren sowie des CTE XL als weitere Werkzeugunterstützung in den vorhandenen Testprozess.

−−− **fehlendes Sollverhalten** Als weiteres Manko der Methode wurde die fehlende Berücksichtigung des Sollverhaltens genannt. Jeder automatisch in das UML−Modell generierte TestCase muss vor der Ausführung manuell um geeignete Auswerteroutinen erweitert werden. Das Ziel von Testverfahren ist jedoch die Ermittlung von Testfällen und nicht die Spezifikation des Sollverhaltens des SUT. Dies ist Aufgabe der in Kap. 2.2.4 vorgestellten Testorakel.

Die als negativ bewerteten Punkte „Unübersichtlichkeit bei komplexen Problemen“, „höherer zeitlicher Aufwand“ und „zeitliches Verhalten“ können durch folgende zwei Bemerkungen ein wenig entschärft werden:

- Die Einarbeitung in neue Testinhalte erfordert immer einen gewissen zeitlichen Aufwand, so dass aufgrund der bisher vorgestellten positiven Aspekte dieser höhere zeitliche Aufwand akzeptiert werden kann. Zudem ist zu erwarten, dass nach einer gewissen Einarbeitungsphase dieser Aufwand geringer wird.

- Neben Ansätzen zur Beherrschung von Komplexität wurde von *Simmes* (1997) speziell für die Problematik der Abbildung zeitlichen Verhaltens in eine Klassifikationsbaumstruktur eine Anwendungspragmatik (vgl. Kap. 5.3.1) für den praktischen Einsatz der CTM erarbeitet, „die dem Funktionsentwickler verdeutlicht, wie er die speziellen Charakteristika reaktiver eingebetteter Systeme im Fahrzeug mit der Klassifikationsbaum-Methode abbilden kann.“ Aufgrund des limitierten Zeitkontingents wurde diese Anwendungspragmatik im durchgeführten Praxistest nicht behandelt. Bei entsprechender Berücksichtigung ist zusätzlich mit einer Steigerung der Übersichtlichkeit bei komplexen Problemen zu rechnen.

Wie bereits in Kap. 4.2 aufgeführt, enthält die Arbeit von *Simmes* (1997) auch eine Bewertung der CTM nach definierten Entscheidungskriterien in einer Nutzwertanalyse hinsichtlich des Systemtests von Kfz−Steuergeräten. Dabei erhält die CTM in den Kriterien „Systematik“, „Handhabung“, „Erlernbarkeit“ und „Darstellung“ den höchsten Erfüllungsgrad von zehn Punkten. Die Kriterien „Leistung“ und „Aufwand“ werden mit acht von zehn Punkten bewertet. Diese Ergebnisse stimmen gut mit den Ergebnissen der durchgeführten Praxistests überein.

Die vorgestellten Ergebnisse stimmen ebenfalls gut mit der von *Yu et al.* (2003) durchgeführten Studie bzgl. der positiven Punkte Erlernbarkeit, Übersichtlichkeit, grafische Repräsentation, Strukturierung und neue Testinhalte überein. Im negativen Bereich gibt es ebenfalls eine Übereinstimmung bzgl. der Unübersichtlichkeit bei komplexen Problemen.

Die CTM war vor dem Zeitpunkt des Praxistests keinem beteiligten Testersteller bekannt. Dieser geringe Bekanntheitsgrad funktionaler Testverfahren ist nach *Chen & Poon* (2004) auf die Tatsache zurückzuführen, dass funktionale Testverfahren weniger systematisch sind als strukturorientierte Testverfahren und daher in Software−Engineering Vorlesungen weniger ausführlich behandelt werden.

7.3.2 Ergebnisse zum Einsatz der CTM/ES

Die Auswertung der Fragebögen ergibt für die CTM/ES die in Tab. 7.2 aufgeführten Ergebnisse.

Bewertung	**Stichpunkt**
+++	Erlernbarkeit
+++	Übersichtlichkeit (Klassifikationsbaum)
++	grafische Repräsentation
++	neue Testinhalte
+	Änderbarkeit
−	alleiniger Einsatz der CTM/ES
−−	Unübersichtlichkeit bei komplexen Problemen
−−	Dynamik der Testentwicklung eingeschränkt
−−	Kontrollstrukturen
−−	UML flexibler
−−−	noch ein Tool
−−−	fehlendes Sollverhalten
−−−	existierende Überdeckungskriterien
−−−	Unübersichtlichkeit (Kombinationstabelle)

Tabelle 7.2: Ergebnisse CTM/ES

Neben der leichten Erlernbarkeit der CTM/ES wurde, wie auch schon im vorherigen Abschnitt vorgestellt, die gute Übersichtlichkeit des Klassifikationsbaumes von den Testerstellern hervorgehoben. Neue, nicht berücksichtigte Testinhalte wurden ebenso wie beim Einsatz der CTM gefunden. Zwar wird zudem die schnelle Spezifikation von Testsequenzen bei einem vorhandenen Klassifikationsbaum sowie die schnelle Erweiterbarkeit des Klassifikationsbaumes von einem Testersteller als positiv erwähnt (Änderbarkeit), die restlichen Punkte richten sich jedoch alle gegen die Integration der CTM/ES in den bestehenden Testerstellungsprozess bzw. gegen die Anbindung an die Testautomatisierung EXAM.

Viele der in Tab. 7.2 aufgeführten Punkte wurden bereits im vergangenen Kapitel vorgestellt und werden aus diesem Grund an dieser Stelle nicht nochmal erläutert. Lediglich neu hinzugekommene Punkte werden im Folgenden ausführlicher diskutiert.

– alleiniger Einsatz der CTM/ES Der alleinige Einsatz der CTM/ES ist nach Meinung eines Testerstellers nicht sinnvoll. Die anforderungsorientierten Testfälle, die sich hinter den Testsequenzen verbergen, müssen aufgrund des hohen Detaillierungsgrades der Testsequenzen bereits vorhanden bzw. spezifiziert sein.

−− Kontrollstrukturen, UML flexibler Weiterhin vermissen einige Testersteller Kontrollstrukturen[3] in der CTM/ES und erachten die UML als flexibler bzgl. der Implementierung von TestCases.

−−− existierende Überdeckungskriterien Von allen Testerstellern wurde zudem die nicht vorhandene Möglichkeit zu Aussagen über unberücksichtigte Transitionen (vgl. Kap. 5.2) bemängelt. Zwar sollte auch hier das Minimalkriterium $CTC_{min} = 1$ prinzipiell erreicht werden. Eine quantitative Bewertung des erreichten Testumfangs anhand der definierten Überdeckungskriterien wie im Falle der CTM ist damit leider nur in sehr geringem Maße möglich (vgl. Kap. 5.2.1). Das alleinige Berücksichtigen einer Äquivalenzklasse in einer Testsequenz ist für eine Bewertung des Testumfangs nicht ausreichend, da der Zeitpunkt einer Aktion bzw. der aktuelle Zustand des SUT ebenfalls für das Verhalten des SUT relevant ist.

−−− Unübersichtlichkeit (Kombinationstabelle) Ein weiterer Punkt, der von allen Testerstellern bemängelt wurde, ist die schlechte Übersicht über die Testsequenzen in der Kombinationstabelle. Zu viele Details sind darin bei praxisnahen Testinhalten enthalten. Dies behindert sehr stark den Überblick über den Inhalt der Testsequenzen und wirkt der angestrebten Verbesserung der Übersichtlichkeit entgegen.

Einige der aufgeführten negativen Aspekte (z.B. Kontrollstrukturen) können auch im Fall der CTM/ES etwas entschärft werden. Insgesamt bleibt das Feedback der Testersteller zum Einsatz der CTM/ES und der damit verbundenen Integration in EXAM jedoch negativ.

[3] Ansätze zur Berücksichtigung von Kontrollstrukturen in der Kombinationstabelle finden sich in *Conrad* (2001) sowie in *Conrad* (2004).

Dem Autor sind keine existierenden Untersuchungen zum Einsatz der CTM/ES bekannt, so dass keine Vergleiche der vorliegenden Ergebnisse wie im Abschnitt zuvor durchgeführt werden können.

Wie auch schon die CTM war die CTM/ES keinem Testersteller vor der Durchführung des Praxistests bekannt.

7.4 Konsequenzen für die Integration der Testverfahren in den Testprozess

Für die Integration der beiden vorgestellten systematischen Testverfahren in den Testprozess von in Fahrzeugsysteme eingebetteter Software mittels der Testautomatisierung EXAM ergeben sich basierend auf den im Kap. 7.3 vorgestellten Ergebnissen des Praxistests folgende Konsequenzen.

7.4.1 Konsequenzen für die Integration der CTM/ES

Auf die Integration der CTM/ES in den vorhandenen Testerstellungsprozess zur Unterstützung des Test−Anforderungs−Management wird verzichtet. Zu viel Kritik ist bei den Praxistests an dem Einsatz der CTM/ES sowie der vorgestellten Vorgehensweise aufgekommen (vgl. Tab. 7.2). Die Unübersichtlichkeit der Testsequenzen in Kombination mit der nicht möglichen Aussage über nicht berücksichtigte Transitionen sind die zwei wesentlichen Argumente gegen die in Kap. 6.1.2 vorgestellte Anbindung der CTM/ES an EXAM.

Aus diesem Grund wird die als Prototyp vorhandene, auf der CTM/ES basierende, Schnittstelle ctmes2exam zwischen dem CTE XL und EXAM nicht weiter entwickelt. Den zum Zeitpunkt des Praxistests existierenden Prototypenstand beinhaltet Anh. C.

7.4.2 Konsequenzen für die Integration der CTM

Die in Kap. 6.1.1 aufgezeigte Integration der CTM in das Test−Anforderungs−Management erfolgt in abgewandelter Weise. Zwar wurde auch der Einsatz der CTM zum Teil negativ bewertet. Das positive Feedback, wie z.B. der gute Überblick über ähnliche Testinhalte und die damit verbundene Möglichkeit zur Diskussion der Testinhalte mit anderen Personen, zur Strukturierung der Testfälle ebenso wie das Vorhandensein von Überdeckungskriterien, überwiegt jedoch (vgl. Tab. 7.1). Zudem konnten die negativen Aspekte „Unübersichtlichkeit bei komplexen Problemen“, „höherer zeitlicher Aufwand“ und „zeitliches Verhalten“ weitgehend entschärft werden (siehe Kap. 7.3.1). Die angepasste Umsetzung des in Kap. 6.1 vorgestellten Integrationskonzeptes unter Berücksichtigung der Ergebnisse der durchgeführten Praxistests ist Gegenstand des nächsten Kapitels.

8 Konzeptumsetzung

Aufbauend auf den im vorherigen Kapitel vorgestellten Ergebnissen des durchgeführten Praxistests sowie den daraus abgeleiteten Konsequenzen für die Integration der Testverfahren in den Testerstellungsprozess beschreibt dieses Kapitel die endgültige Anbindung des CTE XL, als wesentlicher Bestandteil der CTM, an das Test−Ausführungs−Management bzw. an die bei der AUDI AG eingesetzte Testautomatisierung EXAM. Die entwickelte Schnittstelle ctm2exam zur Überführung der Testfall−Struktur aus dem CTE XL nach EXAM wird dabei ausführlich vorgestellt. Abschließend erfolgt eine kurze Vorstellung eines Praxiseinsatzes der modifizierten Vorgehensweise anhand eines aktuellen Entwicklungsprojektes.

8.1 Einbindung der CTM in den Testprozess

Die Einbindung der CTM bzw. des CTE XL in den existierenden Testerstellungsprozess erfolgt in Anlehnung an die bereits in Abb. 6.5 dargestellte Weise. Die bestehende EXAM−Toollandschaft (s. Abb. B.4) des Test−Anforderungs−Management wird dabei um den CTE XL, als Werkzeug zur systematischen Testfallermittlung, erweitert und eine Anbindung an das Test−Ausführungs−Management bzw. die Testautomatisierung EXAM in Form einer Schnittstelle (ctm2exam) hergestellt. Die entsprechende Testfall−Struktur in der erstellten .cte−Datei wird mittels der entwickelten Schnittstelle nach EXAM bzw. in das im ARTiSAN Studio geöffnete UML−Modell an eine beliebige Position überführt (s. Abb. 8.1). Dabei wird für jeden im CTE XL spezifizierten Testfall die EXAM−spezifische Ordnerstruktur (s. Abb. B.11) inkl. eines TestCase, eines untergeordneten Objekt−Sequenz−Diagramms sowie eines Use−Case−Diagramms angelegt (s. Abb. 8.5).

Im Gegensatz zu der in Anh. C vorgestellten Schnittstelle ctmes2exam zwischen dem CTE XL und EXAM wird mittels der in diesem Kapitel vorgestellten Schnittstelle ctm2exam jedoch keinerlei Testimplementierung im ausgewählten UML−Modell generiert. Durch den Einsatz der CTM wird lediglich eine Testfall−Struktur im CTE XL definiert (s. Abb. 8.1). Diese Testfall−Struktur wird von der entwickelten Schnittstelle in ein UML−Modell überführt. Darin muss die generierte Testfall−Struktur (Teststruktur) manuell um die Testimplementierung erweitert werden. Diese Aufgabe fällt der Rolle des Testdesigners (s. Anh. B.4.1) im Test−Ausführungs−Management zu.

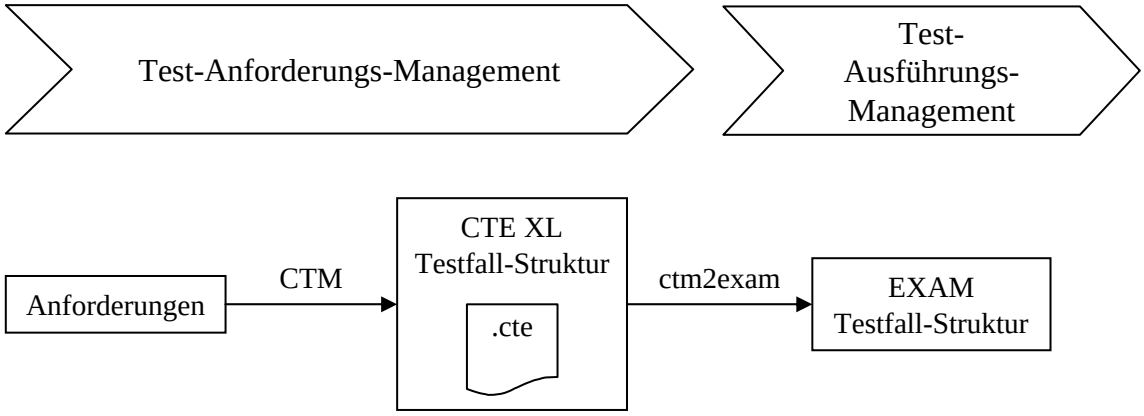


Abbildung 8.1: Einbindung der Schnittstelle ctm2exam in das EXAM–Prozessmodell

Durch diese Integration der CTM bzw. des CTE XL in den Testerstellungsprozess von in Fahrzeugelektronik eingebetteter Software besteht nach Aussagen von Testerstellern ein guter Überblick über die spezifizierten Testfälle wodurch die Testfälle einfacher strukturiert, diskutiert und vermittelt werden können (vgl. Kap 7.3.1). Zudem lässt sich der spezifizierte Testumfang anhand der existierenden Überdeckungsmaße, die wiederum eine Grundlage für die Definition messbarer Testziele und Testendekriterien darstellen, bewerten. Somit werden fast alle in Kap. 3.6.1 genannten Probleme der Testfallerstellung behoben. Einzig die Testqualität ist nach wie vor vom Qualifikationsniveau des jeweiligen Testerstellers abhängig. Sie lässt sich jedoch aufgrund der grafischen Repräsentation besser und schneller mit anderen Personen diskutieren und damit auch einfacher bewerten.

8.2 Schnittstellenbeschreibung CTM–EXAM (ctm2exam)

8.2.1 Anforderungen

Um von der entwickelten Schnittstelle ctm2exam verwertbar zu sein, muss die mittels des CTE XL erstellte .cte–Datei gewissen Anforderungen entsprechen.
Zum einen muss jede Testfall–Gruppe und jeder Testfall in der Kombinationstabelle über die neue[1] Eigenschaft `artisanId` vom CTE XL–Typ „TextualTag“ verfügen. Zusätzlich muss jeder Testfall mit der Eigenschaft `version`, ebenfalls vom CTE XL–Typ „TextualTag“, ausgestattet sein. Die Inhalte dieser beiden neuen Eigenschaften werden im weiteren Verlauf dieses Kapitels näher beschrieben. Zum anderen muss genau eine Testfall–Gruppe als „Wurzel“ in der Kombinationstabelle existieren (z.B. `rootPackage_CTM` in Abb. 8.5(a)). Unterhalb dieser obersten Testfall–Gruppe kann eine beliebige Struktur aus Testfall–Gruppen und

[1] Das Hinzufügen neuer Eigenschaften an ausgewählte Elemente des CTE XL wurde in Kap. 5.4 vorgestellt.

Testfällen existieren. Testsequenzen, als weitere existierende Elemente der Kombinationstabelle, sind dabei nicht zugelassen.

An das UML–Modell wird lediglich eine Anforderung gestellt. Darin muss, neben den Standardelementen (s. Anh. B.5), der Stereotyp mit dem Namen «testModule» enthalten sein. Die Verwendung dieses Stereotyps wird in Kap. 8.3 näher erläutert.

8.2.2 Datenstruktur

Die Funktionsweise der Schnittstelle gliedert sich in zwei Bereiche. Zunächst wird die ausgewählte .cte–Datei geparst und relevante Inhalte in einer internen Datenstruktur (s. Abb. 8.2) abgelegt. Ausgehend von dieser Datenstruktur beginnt der Import, der im CTE XL angelegten Teststruktur, in das geöffnete UML–Modell an die markierte Stelle bzw. das Update der im UML–Modell enthaltenen Teststruktur anhand der CTE–Teststruktur.

Die interne Datenstruktur, in der sowohl die Inhalte aus der .cte–Datei als auch aus dem UML–Modell abgelegt werden, ist in Abb. 8.2 als UML–Klassen–Diagramm dargestellt. Darin finden sich die drei Klassen:

- CombinationTableElement
- Testcase
- Testcase–Group

Relevante bzw. zulässige Inhalte im CTE XL stellen (neben den Elementen eines Klassifikationsbaumes) lediglich Testfälle (`Testcase`) und Testfall–Gruppen (`Testcase-Group`) in der Kombinationstabelle dar. Beide Elemente sind Spezialisierungen eines Kombinationstabellenelementes (`CombinationTableElement`). Das `CombinationTableElement` besitzt fünf relevante Attribute und vererbt diese an `Testcase` und `Testcase-Group`.

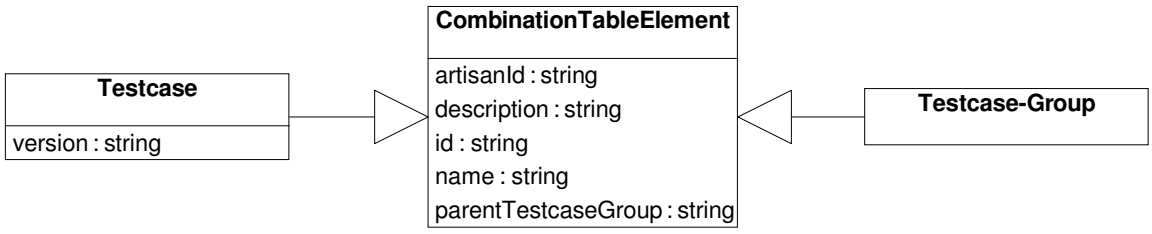


Abbildung 8.2: UML–Klassen–Diagramm der internen Datenstruktur der ctm2exam–Schnittstelle

Jedes zulässige Element der Kombinationstabelle verfügt über einen Namen (`name`), eine CTE XL–ID (`id`), eine Beschreibung (`description`), eine ARTiSAN–ID (`artisanId`) sowie über eine Verknüpfung des jeweiligen Elements zu der übergeordneten Testfall–Gruppe (`parentTestcaseGroup`). Der Testcase verfügt zusätzlich über das Attribut `version`, welches die aktuelle Version des Testfalls enthält. Alle in Abb. 8.2 aufgeführten Attribute sind in der ctm2exam–Schnittstelle als Typ „string“ realisiert.

Die CTE XL–ID wird durch den CTE XL vergeben, während die ARTiSAN–ID vom ARTiSAN Studio generiert wird. Beide IDs werden von der Schnittstelle automatisch in der internen Datenstruktur für jedes darin enthaltene Element abgelegt. Im Fall der ARTiSAN–ID ist dies jedoch erst nach Generierung des entsprechenden Elementes im UML–Modell möglich. Der Eintrag in `parentTestcaseGroup` wird ebenfalls automatisch durch die Schnittstelle vorgenommen. Er ergibt sich aus der durch den Benutzer spezifizierten Testfallstruktur im CTE XL mittels Testfall–Gruppen.

Name und Beschreibung eines CTE XL–Elementes bzw. die Version jedes Testfalls müssen durch den Benutzer manuell definiert werden. Hierfür steht im CTE XL die Eigenschaft `Standard` mit den zwei Feldern `Name` und `Beschreibung` (s. Abb. 5.9) sowie die neu hinzugefügte Eigenschaft `version`, als separates Feld, zur Verfügung (vgl. Kap. 8.2.1).

8.2.3 Benutzerinteraktion

Die Benutzerinteraktion wird über eine grafische Benutzeroberfläche (s. Abb. 8.3) realisiert. Für die Steuerung der ctm2exam–Schnittstelle stehen dabei folgende Funktionen zur Verfügung:

- Auswahl des EXAM–Verzeichnisses
- Auswahl einer .cte–Datei
- Start des Imports
- Start des Update
- Statusmeldungen der Verarbeitung
- Löschen der Statusmeldungen
- Schließen der Anwendung

Der Benutzer hat zum einen die Möglichkeit, den Pfad zum EXAM–Verzeichnis (s. Anh. B.4) auf seinem Client anzugeben. Die Standardeinstellung „C:\EXAM“ zeigt Abb. 8.3. Zudem kann die jeweilige .cte–Datei ausgewählt werden. Wurde eine entsprechende .cte–Datei ausgewählt, kann entweder die Import– oder die Updatefunktion aktiviert werden.

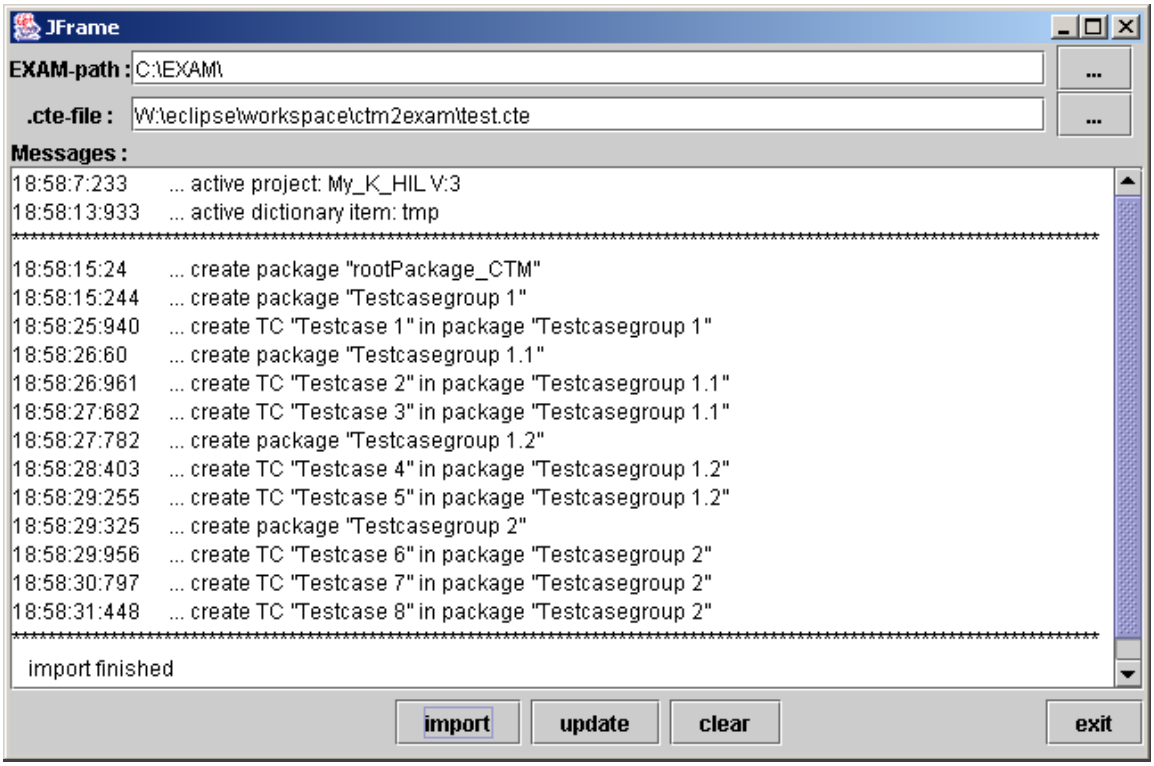


Abbildung 8.3: Benutzeroberfläche der ctm2exam–Schnittstelle

Daraufhin werden dem Benutzer Statusmeldungen zum aktuellen Fortschritt angezeigt. Zusätzlich zur Ausgabe von übersichtlichen Statusmeldungen in der Benutzeroberfläche wird die Datei „ctm2exam.log“ im Installationsverzeichnis (s. Kap. 8.2.5) erstellt. Darin finden sich alle durchgeführten Aktionen in detaillierter Form.

8.2.4 Funktionalität der ctm2exam–Schnittstelle

Die nächsten vier Unterabschnitte beschreiben die Funktionalität der entwickelten Schnittstelle (ctm2exam) zwischen dem CTE XL und dem ARTiSAN Studio, wie sie bereits in Abb. 8.1 aufgezeigt wurde.

Nach einer kurzen Betrachtung allgemeiner Überprüfungen werden die Import– und Updatefunktion, als Hauptfunktionalitäten der Schnittstelle, näher vorgestellt. Abschließend werden die definierten Fehlermeldungen kurz aufgeführt.

8.2.4.1 Allgemeine Überprüfungen

Nachdem der Import oder das Update gestartet wird, erfolgen zunächst einige allgemeine Überprüfungen. Die Überprüfungen der Importfunktion sind als Ablaufdiagramm

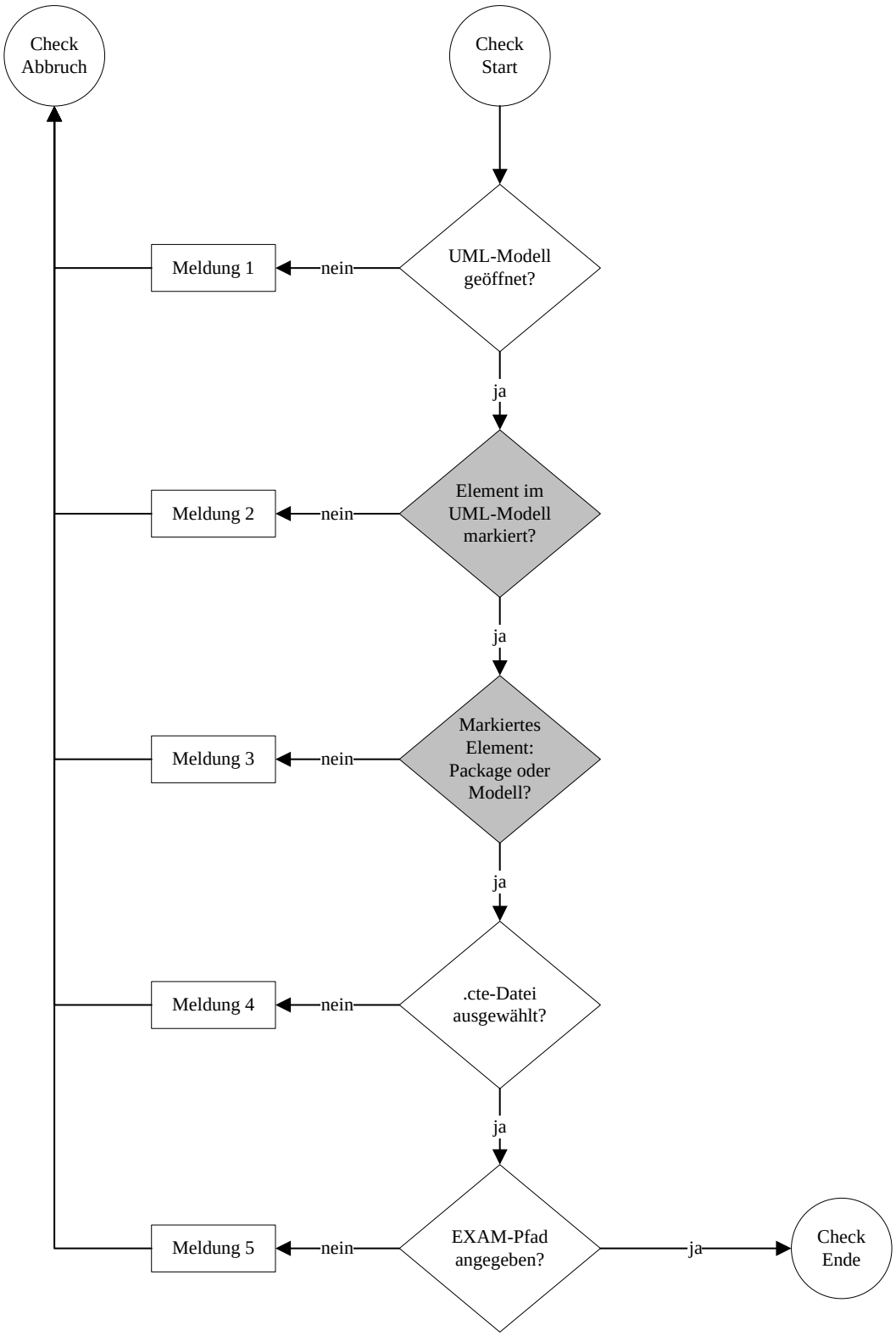


Abbildung 8.4: Ablaufdiagramm der allgemeinen Überprüfungen der Importfunktion

in Abb. 8.4 aufgeführt. Bei der Updatefunktion sind die Überprüfungen `Element im UML–Modell markiert` sowie `Markiertes Element Package oder Modell` nicht notwendig und aus diesem Grund in Abb. 8.4 grau hinterlegt.

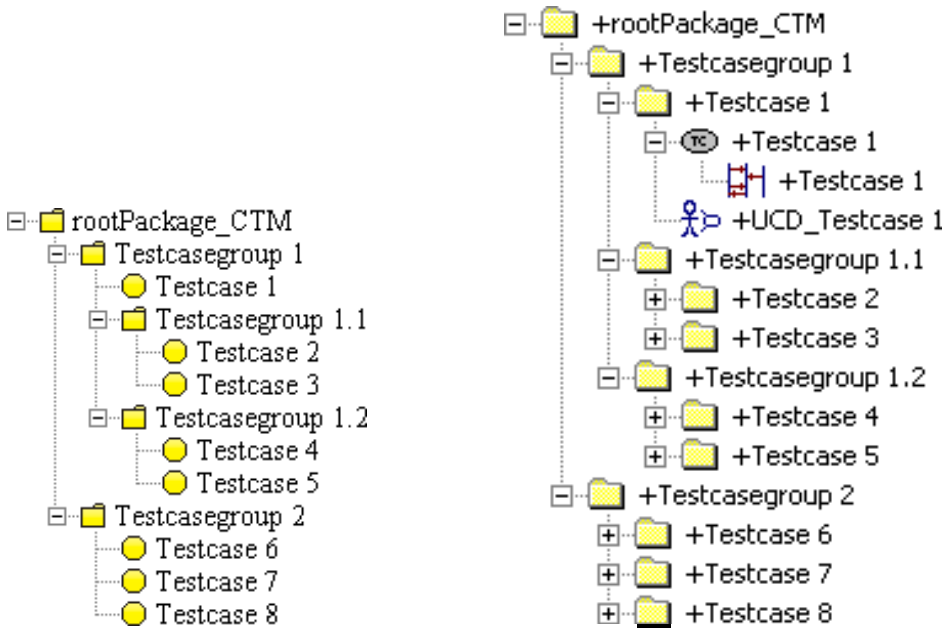
Als erstes überprüft die Checkfunktion, ob ein UML–Modell im ARTiSAN Studio geöffnet und ein Element darin markiert ist. Ist dies der Fall, wird weiterhin überprüft, ob es sich bei dem markierten UML–Modell–Element um ein Package im UML–Modell oder evtl. um das UML–Modell selbst handelt. Entspricht das markierte UML–Modell–Element den geforderten Kriterien, wird weiterhin überprüft, ob eine .cte–Datei ausgewählt und auch vorhanden ist. Abschließend findet die Überprüfung des angegebenen EXAM–Pfades auf Existenz statt. Wurden alle Überprüfungen mit „ja“ beantwortet, wird die eigentliche Import– bzw. Updatefunktion aufgerufen (s. Kap. 8.2.4.2 bzw. Kap. 8.2.4.3). Wurde dagegen eine der vorgestellten Überprüfungen mit „nein“ beantwortet, erhält der Anwender eine entsprechende Fehlermeldung und der Import bzw. das Update der Teststruktur wird nicht gestartet. Alle in den folgenden Ablaufdiagrammen enthaltenen Fehlermeldungen sind in Kap. 8.2.4.4 gesammelt aufgeführt.

8.2.4.2 Import der Teststruktur

Nachdem alle allgemeinen Überprüfungen durchgeführt wurden, müssen vor dem eigentlichen Import, der im CTE XL erstellten Teststruktur (s. Abb. 8.5(a)) in das geöffnete UML–Modell, noch zwei weitere import–spezifische Überprüfungen durchgeführt werden. Beim Einlesen der CTE–Struktur in die bereits erwähnte interne Datenstruktur (vgl. Kap. 8.2.2) wird eine Prüfung der CTE–Struktur auf unzulässige Elemente, z.B. enthaltene Testsequenzen, vorgenommen. In einem weiteren Schritt wird geprüft, ob der Name der im CTE XL definierten obersten (top–level) Testfall–Gruppe im Konflikt mit dem Namen eines bereits existierenden Package, an der ausgewählten Stelle im UML–Modell, steht. Existiert an der markierten Stelle im UML–Modell kein gleichnamiges Package, beginnt der Import der im CTE XL erstellten Teststruktur in das ausgewählte UML–Modell. Das zur Importfunktion zugehörige Ablaufdiagramm, inkl. der zwei Überprüfungen, zeigt Abb. 8.6.

Ausgehend von der Teststruktur in der ausgewählten .cte–Datei (z.B. Abb. 8.5(a)) wird die in Kap. 8.2.2 vorgestellte interne Datenstruktur mit Inhalten befüllt und anschließend die Generierung der Teststruktur im UML–Modell (z.B. Abb. 8.5(b)) vorgenommen. Dabei werden alle im CTE XL spezifizierten Testfälle sowie die erstellte Strukturierung mittels Testfall–Gruppen vollständig in das geöffnete UML–Modell mittels einer rekursiven Funktion überführt (s. Abb. 8.6). Zunächst wird die oberste Testfall–Gruppe (z.B. `rootPackage_CTM` in Abb. 8.5(a)) als Wurzel–Package im UML–Modell unterhalb der markierten Stelle generiert. In der Eigenschaft `Beschreibung` von Testfall–Gruppen definierte textuelle Inhalte werden dem korrespondierenden UML–Package im Eintrag `Description`[2] ohne Änderung

[2] Im vorliegenden Kapitel werden durch diese Textformatierung, neben der bereits erwähnten Kennzeichnung von Inhalten referenzierter Abbildungen, auch EXAM– und CTE XL–spezifische Inhalte hervorgehoben.



(a) Teststruktur im CTE XL (b) Teststruktur im UML–Modell

Abbildung 8.5: Gegenüberstellung der Test– bzw. Packagestrukturen (ctm2exam)

hinzugefügt.

Enthält die oberste Testfall–Gruppe Testfälle, werden diese in das UML–Modell überführt. Dabei wird für jeden CTE XL–Testfall ein TestCase in einem gleichnamigen Package im UML–Modell erstellt (s. Anh. B.5.2). In der Eigenschaft `Standard` definierte textuelle Beschreibungen eines CTE XL–Testfalls werden als `Description` an den entsprechenden TestCase angehängt. Zudem werden die Tag Definitions `testCaseID`, `testCaseState` und `testCaseVersion` des generierten TestCase automatisch mit Inhalten gefüllt. Die Tag Definition `testCaseID` setzt sich folgendermaßen zusammen:

„Packagename des TestCase“.„Name des TestCase“ (z.B. Testcasegroup_1.1.Testcase_2)

Die Tag Definition `testCaseState` wird bei erstmaligem Import immer mit dem Wert „specified“ belegt, während `testCaseVersion` bei der erstmaligen Generierung immer einen Initialwert von „1.0.0“ erhält.

Nachdem alle Testfälle der obersten Testfall–Gruppe in das UML–Modell importiert wurden, wird von der rekursiven Funktion überprüft, ob weitere Testfall–Gruppen (z.B. `Testcasegroup 1` und `Testcasegroup 2` in Abb. 8.5(a)) in der aktuellen Testfall–Gruppe (z.B. `rootPackage_CTM` in Abb. 8.5(a)) enthalten sind. Enthält die aktuelle Testfall–Gruppe keine weiteren, untergeordneten Testfall–Gruppen wird die Importfunktion beendet. Falls jedoch weitere Testfall–Gruppen vorhanden sind, wird zunächst für jede Testfall–Gruppe ein Package im UML–Modell erstellt. Anschließend wird für jedes erstellte UML–Package

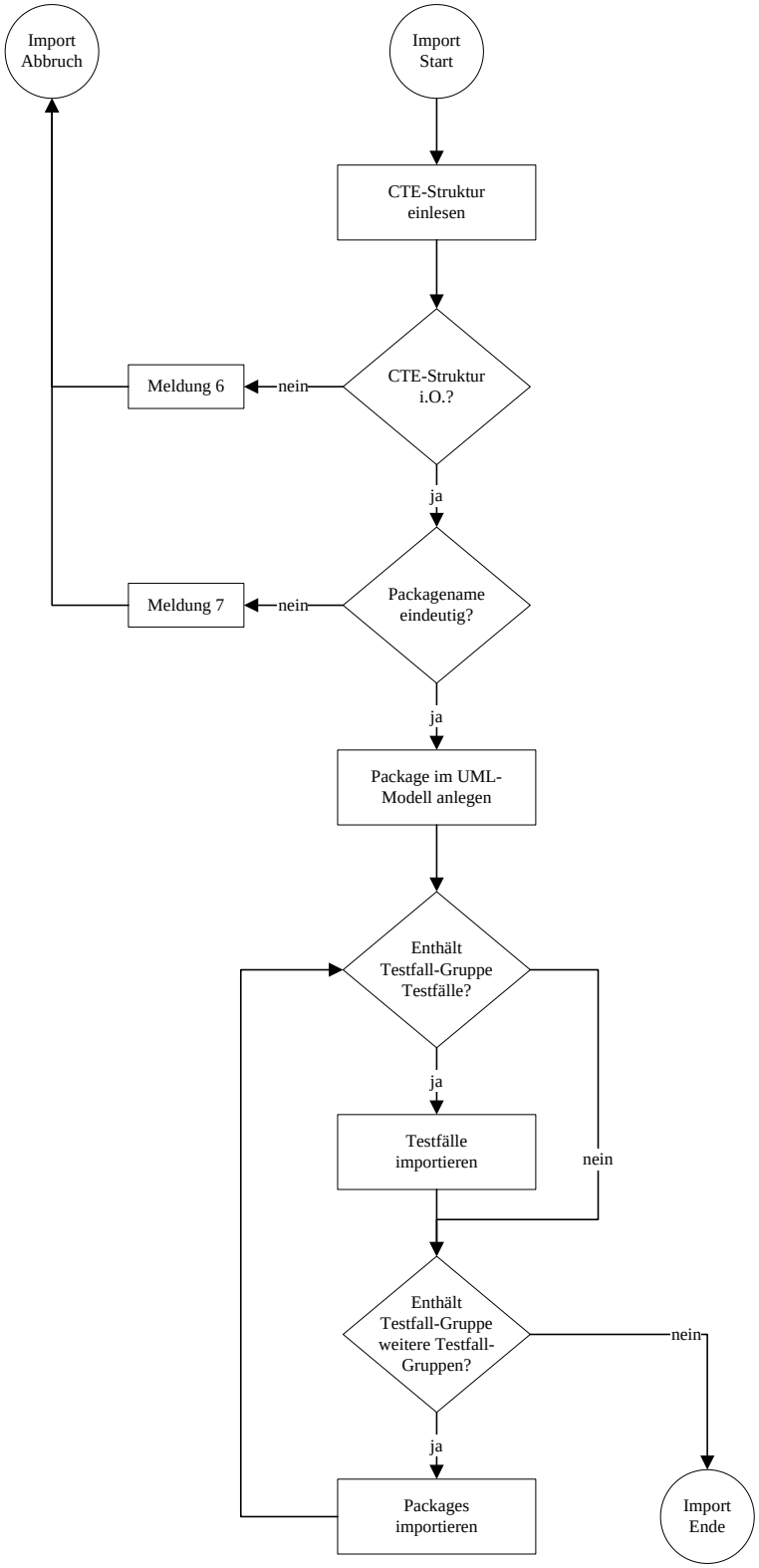


Abbildung 8.6: Ablaufdiagramm der Importfunktion

überprüft, ob darin TestCases zu erstellen sind. Auf diese Weise wird die im CTE XL spezifizierte Teststruktur in das ausgewählte UML−Modell überführt. Die zur Abb. 8.5 zugehörigen Statusmeldungen bei der Generierung der beispielhaften Teststruktur im UML−Modell sind in Abb. 8.3 im Feld `Messages` dargestellt.

Nach erfolgreichem Import der Teststruktur in das ausgewählte UML−Modell wird eine Kopie der verwendeten .cte−Datei unterhalb des angegebenen EXAM−Verzeichnisses abgelegt. Hierfür wird, falls noch nicht vorhanden, zunächst ein cte−Verzeichnis (C:\EXAM\cte) im EXAM−Verzeichnis angelegt. Darin wird von der Schnittstelle eine Packagestruktur erstellt, die dem UML−Modellpfad des markierten UML−Modell−Elementes entspricht. Auf diese Weise ist die Archivierung der .cte−Dateien realisiert, die in ähnlicher Form von weiteren, eingesetzten EXAM−Tools (s. Abb. B.4) verwendet wird.

8.2.4.3 Update der Teststruktur

Beim Update einer im UML−Modell bereits existierenden Teststruktur anhand der im CTE XL spezifizierten Teststruktur sind, wie auch schon bei der im vorherigen Abschnitt vorgestellten Importfunktion, neben den allgemeinen Überprüfungen (s. Abb. 8.2.4.1) einige spezielle Überprüfungen notwendig. Abb. 8.7 zeigt das Ablaufdiagramm der Updatefunktion inkl. der angesprochenen update−spezifischen Überprüfungen.

Als erstes wird die oberste Testfall−Gruppe der erstellten Teststruktur in der .cte−Datei identifiziert. Daraufhin wird überprüft, ob eine Verknüpfung zum geöffneten UML−Modell durch einen entsprechenden Eintrag in der Eigenschaft `artisanId` des Wurzel−Package in der Kombinationstabelle vorhanden ist. Enthält die Eigenschaft `artisanId` einen Eintrag, wird nach dem Package mit dieser ID im geöffneten UML−Modell gesucht. Existiert das gesuchte Package im UML−Modell, wird jede Struktur (UML und CTE) in eine separate interne Datenstruktur (vgl. Kap. 8.2.2) abgebildet. Dem Einlesen folgt ein Vergleich beider Strukturen. Sind beide Strukturen inkonsistent, weil z.B. Elemente aus der UML−Struktur gelöscht wurden, diese aber noch in der CTE−Struktur vorhanden sind, wird eine entsprechende Meldung ausgegeben. Der Anwender hat daraufhin die Möglichkeit, das erneute Anlegen der gelöschten Inhalte vorzunehmen oder das Update abzubrechen. Sind beide Teststrukturen konsistent, erfolgt das eigentliche Update der Teststruktur im UML−Modell.

Falls neue Kombinationstabellenelemente (Testfall−Gruppen, Testfälle) im CTE XL spezifiziert sind, werden diese zunächst an entsprechender Position im UML−Modell erzeugt. Das Vorgehen hierbei orientiert sich an der Importfunktion und wurde bereits im vorangegangenen Abschnitt ausführlich behandelt. Wurden dagegen Elemente innerhalb einer Struktur verschoben, so werden die entsprechenden Elemente im UML−Modell anhand der CTE−Struktur neu positioniert. Falls Kombinationstabellenelemente aus der CTE−Struktur gelöscht wurden, so werden auch die entsprechenden Elemente aus dem UML−Modell entfernt. Die CTE−Struktur stellt damit den Master für die Aktualisierung dar. Die Struktur im UML−Modell wird immer entsprechend der CTE−Struktur angepasst.

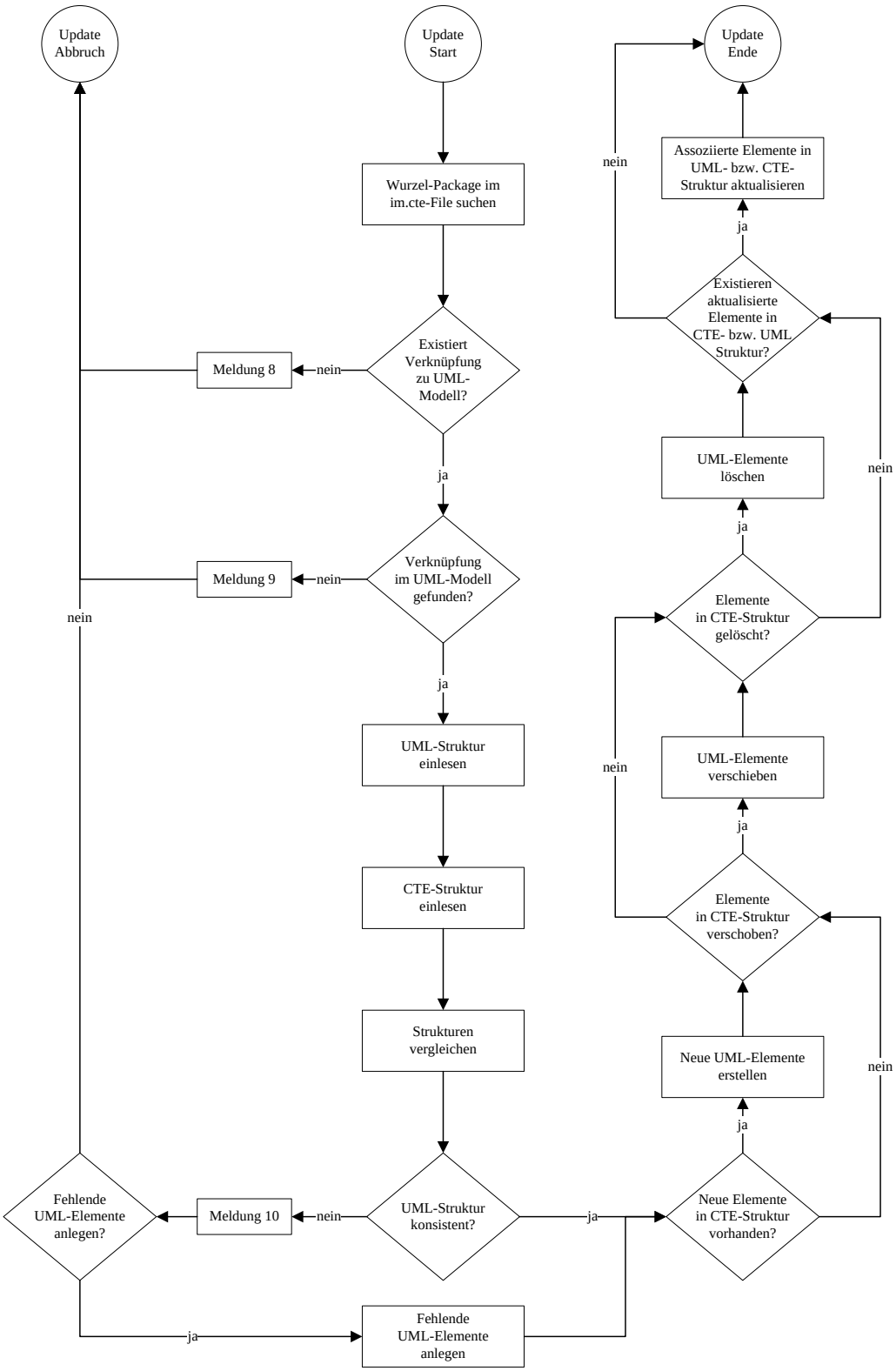


Abbildung 8.7: Ablaufdiagramm der Updatefunktion

Neben den soeben erwähnten Funktionen (Erstellen, Verschieben, Löschen) ist auch eine bidirektionale Aktualisierung ausgewählter Inhalte der beiden Teststrukturen durch die entwickelte Schnittstelle realisiert. Übersteigt die Versionsnummer (Eigenschaft `version`) eines Testfalls in der CTE−Struktur die Versionsnummer (`testCaseVersion`) des entsprechenden TestCases im UML−Modell wird der TestCase im UML−Modell aktualisiert. Dabei wird der Name, die Beschreibung des TestCase sowie die neue Versionsnummer angepasst. Verfügt dagegen ein TestCase im UML−Modell über eine höhere Versionsnummer, werden die erwähnten Inhalte des entsprechenden Testfalls in der CTE−Struktur aktualisiert und in der dazugehörigen .cte−Datei festgehalten.

8.2.4.4 Definierte Fehlermeldungen

Die in den vorangegangenen Abschnitten (s. Abb. 8.4, Abb. 8.6 und Abb. 8.7) erwähnten Fehlermeldungen enthalten folgende Inhalte:

Meldung 1 „no model in ARTiSAN Studio opened, please open model in ARTiSAN Studio“

Meldung 2 „no element in model selected, please select package− or model−element“

Meldung 3 „selected element in model is not of type model or package, please select package− or model−element in model“

Meldung 4 „no .cte−file selected, please select .cte−file“

Meldung 5 „no EXAM−path specified, please specify EXAM−path“

Meldung 6 „selected .cte−file contains unallowed contents (e.g. testsequences), please delete these contents“

Meldung 7 „name of top level testcase−group in selected .cte−file already exists as package in selected model element, please change name of top level testcase−group in selected .cte−file or select another position in model“

Meldung 8 „top level testcase−group contains no entry in artisanId, update impossible, please check if contents of .cte−file already imported in model“

Meldung 9 „no element in model with artisanId of top level testcase−group found, update impossible, please check if contents of .cte−file already imported in model“

Meldung 10 „uml− and cte−structure are inconsistent, please establish consistency or abort update“

Zahlreiche weitere Fehlermeldungen sind vorhanden. Diese werden jedoch aufgrund der hohen Anzahl an dieser Stelle nicht weiter behandelt.

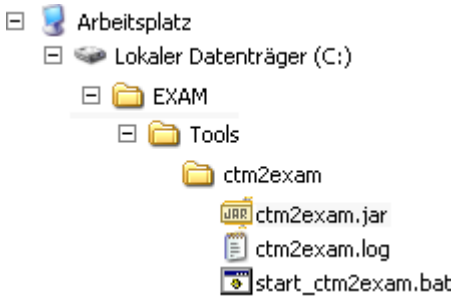


Abbildung 8.8: Das ctm2exam Installationsverzeichnis

8.2.5 Installationsverzeichnis

Wie auch alle anderen EXAM–Tools kann die entwickelte Schnittstelle ctm2exam im Verzeichnis C:\EXAM\Tools auf jedem Client, der das ARTiSAN Studio nutzt, installiert werden. Im Installationsverzeichnis ctm2exam befindet sich ein .jar–Archiv sowie eine Batchdatei zum Starten der Anwendung (s. Abb. 8.8). Nach Ausführung der Anwendung wird darin die bereits in Kap. 8.2.3 erwähnte Logdatei „ctm2exam.log“ abgelegt.

8.3 Berücksichtigung der vorhandenen DOORS–EXAM Schnittstelle

Um die in einem UML–Modell enthaltenen Testfälle in übersichtlicherer Form darzustellen, existiert eine Schnittstelle zwischen EXAM und DOORS. Diese Schnittstelle exportiert alle markierten und mit dem Stereotyp «testModule» gekennzeichnete Testfälle aus dem UML–Modell in ein DOORS–Dokument. Neben der TestCase–Packagestruktur werden dabei die textuellen Beschreibungen der einzelnen TestCases berücksichtigt.

Von der entwickelten Schnittstelle werden alle generierten Packages mit dem Stereotyp «testModule» versehen. Verbunden mit der Einhaltung der EXAM–spezifischen Strukturierung von TestCases (s. Abb. B.11) steht die generierte Teststruktur der bestehenden DOORS–EXAM Kopplung uneingeschränkt zur Verfügung. Der CTE XL integriert sich damit nahtlos in die bestehende Toollandschaft. Alle bisher eingesetzten Tools sind im vollem Umfang weiterhin nutzbar.

8.4 Praxiseinsatz

Im vorliegenden Abschnitt wird kurz erläutert, wie die angepasste Vorgehensweise sowie die entwickelte Schnittstelle ctm2exam vom Autor in einem aktuellen Entwicklungsprojekt der

AUDI AG erfolgreich eingesetzt werden. Bei dem Entwicklungsprojekt handelt es sich um eine weitere Subfunktion der Funktion Blinken (vgl. Kap. 6.2.2.1), die Funktion Notbremswarnblinken (NBWB). Dabei werden bei einer detektierten Notbremsung die Warnblinker des Fahrzeugs automatisch aktiviert, um folgende Fahrzeuge auf die bestehende Gefahrensituation aufmerksam zu machen.

Abb. 8.9 zeigt den für die Funktion Notbremswarnblinken vollständigen Klassifikationsbaum. In der Kombinationstabelle sind darin insgesamt 29 Testfälle spezifiziert. Die definierten Überdeckungsmaße nehmen damit folgende Werte an:

$$CTC_{min} = \frac{actclass}{maxclass} = \frac{33}{33} = 1$$

$$CTC_{max} = \frac{actfall}{maxfall} = \frac{29}{(4+5+5+1)\cdot 2\cdot 3\cdot 4\cdot 3\cdot 2\cdot 2\cdot 2} = \frac{29}{8640} = 0.003$$

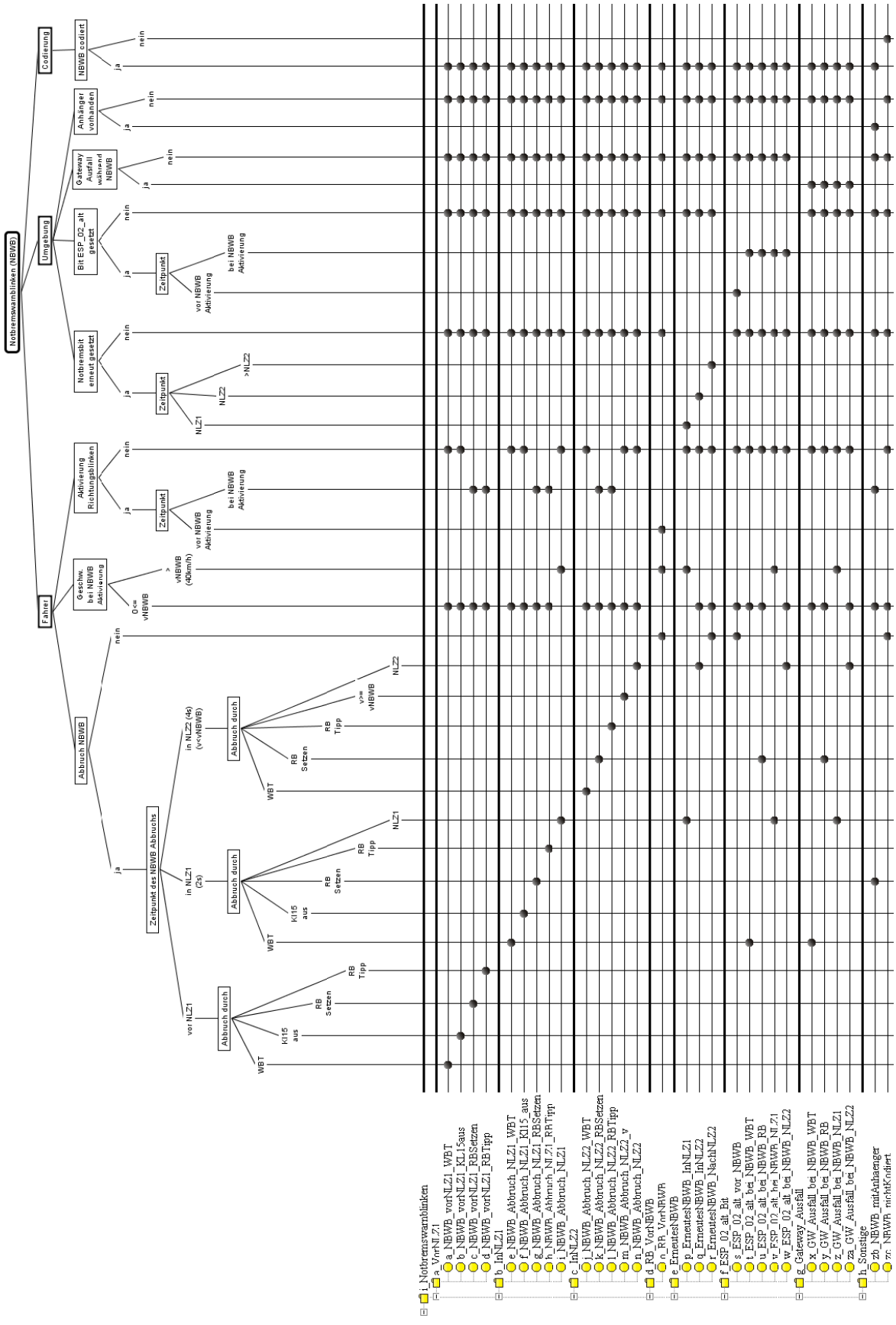
Jede der 33 im Klassifikationsbaum enthaltenen Blattklassen ist dabei in mindestens einem Testfall berücksichtigt ($CTC_{min} = 1$), womit gleichzeitig jeder vom Autor als Testersteller identifizierte Gesichtspunkt in mindestens einem Testfall enthalten ist.

Die Kombinatorik erlaubt insgesamt 8640 verschiedene Kombinationen der Blattklassen. Davon sind 29 Kombinationen durch die spezifizierten Testfälle realisiert. CTC_{max} nimmt damit den Wert von 0.003 an. Anhand dieses Wertes wird die geringe Aussagekraft des Überdeckungsmaßes CTC_{max} nochmals deutlich (vgl. Kap. 6.2.2.3).

Für den vorliegenden Fall erweist sich dagegen die von *Büchner* (2002) eingeführte Daumenregel (vgl. Kap. 5.1.3) als sehr zutreffend. Bei insgesamt 33 Blattklassen im Klassifikationsbaum werden vom Autor 29 Testfälle als ausreichende Testabdeckung für Funktionstests der Funktion Notbremswarnblinken erachtet. Die Anzahl spezifizierter Testfälle liegt damit in der Größenordnung der Summe aller Blattklassen bei gleichzeitiger Berücksichtigung des Minimalkriteriums ($CTC_{min} = 1$).

Die zum Klassifikationsbaum bzw. zur Kombinationstabelle aus Abb. 8.9 zugehörige Teststruktur im EXAM–UML–Modell zeigt Abb. 8.10. Unterhalb des Package `i_Notbremswarnblinken` befinden sich neben den im CTE XL spezifizierten Testfall–Gruppen (z.B. `a_VorNLZ1`) und Testfällen (z.B. `a_NBWB_vorNLZ1_WBT`) weitere manuell hinzugefügte Inhalte (z.B `_subSequences`, `i_Notbremswarnblinken_SUITE`), die zur Ausführung der TestCases in EXAM notwendig sind (s. Anh. B.5).

Bei der Erstellung der Teststruktur der Funktion Notbremswarnblinken war die CTM sowie der CTE XL dem Autor eine große Hilfe. In Diskussionen mit anderen Testerstellern konnten die in der Kombinationstabelle spezifizierten Testfälle sowie die verschiedenen, im Klassifikationsbaum dargestellten, Hauptmerkmale der Funktion Notbremswarnblinken schnell und einfach vermittelt werden. Ebenso förderlich war die Methode zur Strukturierung der spezifizierten Testfälle. Somit bestätigt der Autor die meisten, aus dem durchgeführten Praxistest (vgl. Kap. 7.3.1) ermittelten, Ergebnisse zum Einsatz der CTM. Einige Punkte, wie z.B. der


Notbremswarnblinken (NBWB)
Fahrer
Umgebung
Codierung
Abbruch NBWB
Zeitpunkt des NBWB Abbruchs
vor NLZ1
in NLZ1 (2s)
in NLZ2 (4s) (v<vNBWB)
Abbruch durch
WBT
KI15 aus
RB Setzen
RB Tipp
v>= vNBWB
nein
ja
Gesch. bei NBWB Aktivierung
0<= vNBWB
> vNBWB (40km/h)
Abblendung Richtungsblinken
Zeitpunkt
vor NBWB Aktivierung
bei NBWB Aktivierung
Notbremslicht eingeschaltet
NLZ1
NLZ2
>NLZ2
Bit ESP_02_alt gesetzt
Gateway Ausfall während NBWB
Anhänger vorhanden
NBWB codiert
1_Notbremswarnblinken
a_VorNLZ1
a_NBWB_vorNLZ1_WBT
b_NBWB_vorNLZ1_KL15aus
c_NBWB_vorNLZ1_RBSetzen
d_NBWB_vorNLZ1_RBTipp
b_inNLZ1
e_NBWB_Abbruch_NLZ1_WBT
f_NBWB_Abbruch_NLZ1_KI15_aus
g_NBWB_Abbruch_NLZ1_RBSetzen
h_NRWB_Abbruch_NLZ1_RBTipp
i_NBWB_Abbruch_NLZ1
c_inNLZ2
j_NBWB_Abbruch_NLZ2_WBT
k_NBWB_Abbruch_NLZ2_RBSetzen
l_NBWB_Abbruch_NLZ2_RBTipp
m_NBWB_Abbruch_NLZ2_v
n_NBWB_Abbruch_NLZ2
d_RB_VorNBWB
o_RB_VorNRWB
e_ErneutesNBWB
p_ErneutesNBWB_InNLZ1
q_ErneutesNBWB_InNLZ2
r_ErneutesNBWB_NachNLZ2
f_ESP_02_alt_Bit
s_ESP_02_alt_vor_NBWB
t_ESP_02_alt_bei_NBWB_WBT
u_ESP_02_alt_bei_NBWB_RB
v_FSP_02_alt_bei_NRWB_NLZ1
w_ESP_02_alt_bei_NBWB_NLZ2
g_Gateway_Ausfall
x_GW_Ausfall_bei_NBWB_WBT
y_GW_Ausfall_bei_NBWB_RB
z_GW_Ausfall_bei_NBWB_NLZ1
za_GW_Ausfall_bei_NBWB_NLZ2
h_Sonstige
Zb_NBWB_mitAnhaenger
zc_NBWB_nichtKodiert

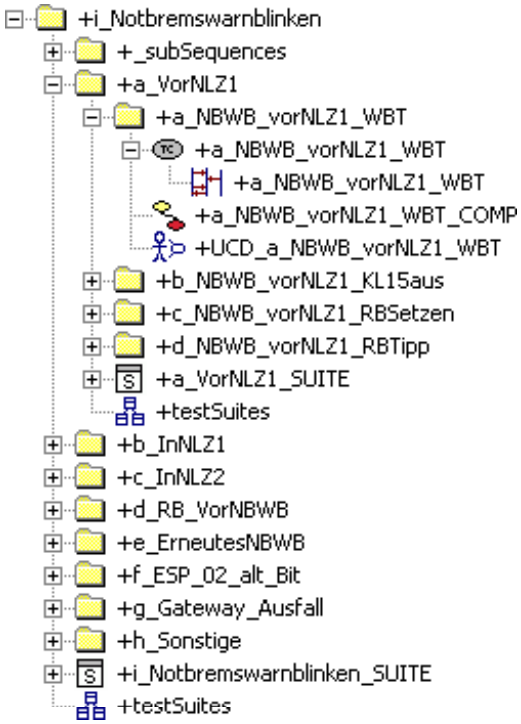


Abbildung 8.10: Teststruktur der Funktion Notbremswarnblinken im EXAM−UML−Modell

höhere zeitliche Aufwand, werden vom Autor dagegen nicht bestätigt. Dabei ist jedoch zu bemerken, dass der Autor sich bereits seit längerer Zeit mit der CTM befasst und daher eine gewisse Erfahrung im Umgang mit der Methode besitzt.

9 Zusammenfassung

Die Komplexität der in hohem Grade vernetzten Fahrzeugsysteme nimmt stetig zu. Die hohen Anforderungen an Sicherheit und Qualität können neben einem strukturierten Entwicklungsprozess nur durch ausgiebige und systematische Tests gewährleistet werden. Testobjekte stellen dabei einzelne oder zu übergeordneten Systemen vernetzte Steuergeräte bzw. die darauf in Form von eingebetteter Software vorliegenden, verteilten Funktionen dar. Während des Entwicklungsprozesses liegen die Testobjekte in verschiedenen Ausprägungen vor. Der Test beginnt mit Komponententests beim Zulieferer und endet mit System– und Fahrzeugtests beim Automobilhersteller.

Dynamische Testverfahren ermöglichen den Test der logischen Funktionalität sowie die Überprüfung der geforderten Realzeitanforderungen in der realen Umgebung des Testobjekts. Strukturorientierte Testverfahren (besser bekannt als White–Box Testverfahren) scheiden für den geschilderten Testprozess aufgrund der Tatsache aus, dass die für diese Testverfahren notwendige Kenntnis der realen Implementierung von den verschiedenen Zulieferern im Allgemeinen nicht zur Verfügung gestellt wird. Funktionale Testverfahren, die auch unter dem Namen Black–Box Testverfahren bekannt sind, berücksichtigen keine Informationen über die innere Struktur des Testobjekts zur Ermittlung von Testfällen, sondern stützen sich allein auf seine funktionale Spezifikation. Sie kommen in der Praxis eher selten zum Einsatz (*Armbrust et al.*, 2004b). Zurückzuführen ist dies nach *Chen & Poon* (2004) zum Teil auf den geringen Bekanntheitsgrad funktionaler Testverfahren.
Dies hat zur Folge, dass die Qualität der meist „Ad–hoc“ ermittelten Tests sehr von der Erfahrung und vom Qualifikationsniveau des Testerstellers anhängt und die genauen Testinhalte meist nur dem Testersteller selbst bekannt sind. Außerdem sind Aussagen bzgl. der Testabdeckung sowie die Definition von messbaren Testzielen verbunden mit geeigneten Testendekriterien aufgrund des geringen Formalisierungsgrades der textuellen Testspezifikationen kaum möglich. Eine sinnvolle Strukturierung von ähnlichen Testinhalten bereitet ebenso Schwierigkeiten wie die Identifikation redundanter Testinhalte im Testprozess.

Aufbauend auf der Bewertung funktionaler Testverfahren wurden die Classification–Tree Method (CTM) sowie die Classification–Tree Method for Embedded Systems (CTM/ES), als existierende systematische Testverfahren, zur Unterstützung des bei der AUDI AG existierenden Testprozesses ausgewählt. In Anlehnung an die Methode MB^{3}T (model–based black–box testing) von *Conrad et al.* (2004) wurde ein neuartiges Konzept zur Kopplung und Integration der beiden Testverfahren in den geschilderten Testprozess von Automobilelektronik bzw. von in Fahrzeugsystemen eingebetteter Software präsentiert. Beide Testver-

fahren kommen dabei im Test−Anforderungs−Management des Testprozesses zum Einsatz. Eine prototypisch entwickelte Schnittstelle stellt die spezifizierten Testsequenzen dem nachgelagerten Test−Ausführungs−Management in ausführbarer Form zur Verfügung.

In einem Praxistest wurde das vorgestellte Integrationskonzept von unerfahrenen und erfahrenen Testerstellern anhand realer Beispiele aktueller Testprojekte validiert. Die Ergebnisse des Praxistests zeigen eine gute Übereinstimmung mit vergleichbaren Untersuchungen zum Einsatz der CTM. Besonders hervorzuheben ist für die CTM die Übereinstimmung in den Punkten Erlernbarkeit, Übersichtlichkeit, grafische Repräsentation, Strukturierung und neue Testinhalte. Spezifizierte Testfälle können durch den Einsatz der CTM gut strukturiert und mit anderen Personen diskutiert werden. Außerdem lassen sich fehlende und redundante Inhalte leicht identifizieren. Existierende Überdeckungsmaße ermöglichen zudem die quantitative Bewertung des Testumfangs sowie die Definition von Testzielen und Testendekriterien. Im vorgestellten Kontext wurde der Einsatz der CTM/ES von den am Praxistest beteiligten Testerstellern größtenteils negativ beurteilt. Vor allem die Unübersichtlichkeit in den Testsequenzen sowie die nicht mögliche Aussage über relevante, aber nicht berücksichtigte, Übergänge zwischen einzelnen Testschritten der Testsequenzen hatte eine Anpassung des vorgestellten Integrationskonzeptes als Konsequenz.

Das modifizierte Integrationskonzept verzichtet daher auf den Einsatz der CTM/ES im geschilderten Testprozess. Die CTM wird jedoch wie geplant zur Systematisierung der Aktivität der Testfallermittlung im Testprozess herangezogen. Anforderungsbasierte Testfälle werden dabei nach der CTM mittels der existierenden Werkzeugunterstützung, dem CTE XL, in einer Testfall−Struktur erstellt. Eine speziell entwickelte Schnittstelle (ctm2exam) zwischen dem Test−Anforderungs−Management und dem Test−Ausführungs−Management stellt diese Testfall−Struktur der eingesetzten, auf UML−Modellen basierenden, Testautomatisierung EXtended Automation Method (EXAM) zur Verfügung. Die in den UML−Modellen automatisch generierte Testfall−Struktur muss dabei jedoch, im Gegensatz zum zuvor verfolgten Integrationskonzept, manuell vom Testdesigner um die Testimplementierung erweitert werden. Der CTE XL wird somit im Test−Anforderungs−Management als Werkzeug zur Testspezifikation eingesetzt. Die im geschilderten Testprozess existierende Trennung zwischen Testspezifikation und Testimplementierung wird dadurch stärker hervorgehoben. Außerdem wird durch den vorgestellten Einsatz der CTM sowie des CTE XL die Systematik bei der Ermittlung von Tests gefördert. Damit verbessert sich die Strukturierung von ähnlichen Testfällen, was wiederum zur Vermeidung von Redundanzen in den Testspezifikationen führt. Die darin spezifizierten Testinhalte können zudem besser und schneller anderen Personen vermittelt werden. Außerdem sind Aussagen über die erreichte Testabdeckung ebenso möglich wie die Angabe von messbaren Testendekriterien. Somit sind fast alle genannten Probleme der Testfallerstellung durch die präsentierte Integration der CTM in den existierenden Testprozess behoben worden. Allein die Testqualität ist nach wie vor vom Qualifikationsniveau und der Erfahrung des jeweiligen Testerstellers abhängig. Sie lässt sich jedoch aufgrund der grafischen Repräsentation mit anderen Personen diskutieren und damit auch einfacher be-

werten.
Abschließend wurde das Vorgehen des angepassten Integrationskonzeptes vom Autor in einem aktuellen Entwicklungsprojekt der AUDI AG erfolgreich eingesetzt. Die aus dem zuvor durchgeführten Praxistest abgeleiteten Ergebnisse zum Einsatz der CTM konnten dabei größtenteils bestätigt werden.

Zusammenfassend ist festzuhalten, dass die CTM eine mächtige Methode für den systematischen Test von in Fahrzeugsysteme eingebetteter Software darstellt. Ihr erfolgreicher, praxisnaher Einsatz wurde an mehreren Anwendungsbeispielen aus dem Automobilbereich im Verlauf dieser Arbeit aufgezeigt. Dennoch kann sie nicht als alleiniges Testverfahren für eingebettete Software bzw. Systeme angesehen werden. Wie *Grimm* (1995) bemerkt, ist ein effektiver Test eingebetteter Systeme nur durch eine Kombination mehrerer Testverfahren zu erreichen. Aus diesem Grund kann das in dieser Arbeit vorgeschlagene Integrationskonzept der CTM in den bei der AUDI AG existierenden Testprozess nur ein erster Schritt in die richtige Richtung sein. Weitere existierende Testverfahren, auch strukturorientierte, müssen in die bestehenden Testprozesse integriert werden.

Doch nicht nur die Testprozesse müssen verbessert werden, auch die Entwicklungsprozesse verlangen nach Erneuerung. Qualität darf nicht erst beim Test in ein Produkt „hineingeprüft“ werden, sondern muss bereits ab der Produktdefinition entlang des gesamten Entwicklungsprozesses entsprechend berücksichtigt werden. Qualitativ hochwertige Systeme lassen sich nur dann entwickeln, wenn die ihnen zugeordneten Anforderungsdokumente eine entsprechend hohe Qualität aufweisen. Vollständige und eindeutig formulierte Anforderungen sind daher eine Grundvoraussetzung für die Entwicklung qualitativ hochwertiger Systeme.

In der modellbasierten Entwicklung, „die durch den durchgängigen Einsatz ausführbarer Modelle in allen konstruktiven Entwicklungsaktivitäten gekennzeichnet ist“ (*Conrad*, 2004), können Anforderungen eindeutig formuliert werden. Durch den soeben beschriebenen durchgängigen Einsatz von ausführbaren Modellen entlang des gesamten Entwicklungsprozesses können Tests bereits sehr frühzeitig initiiert werden. Bestimmte Fehler können dadurch entsprechend früh identifiziert und behoben werden. Dies führt zu Systemen mit einer geringeren Fehlerquote, welche wiederum in kürzeren Entwicklungszyklen und somit kostengünstiger entwickelt werden können. Diese neuen Möglichkeiten zur Entwicklung neuer Software−Systeme gilt es konsequent auszunutzen, um die stetig steigende Komplexität der Systeme zukünftig besser zu beherrschen.

A Klassifikation analytischer Testverfahren

Analytische Qualitätssicherungsverfahren lassen sich in Abhängigkeit von ihrer Zielsetzung in drei Klassen unterscheiden (*Balzert*, 1998). Abb. A.1 auf Seite 118 zeigt diese Klassifikation.

Analysierende Verfahren vermessen und/oder stellen bestimmte Eigenschaften von Systemkomponenten dar. Unterkategorien bilden hierbei z.B. die Analyse der Bindungsart, Metriken, Grafiken und Tabellen sowie die Anomalienanalyse.
Metriken wie z.B. die zyklomatische Zahl (McCabe–Metrik) oder die Halstead–Metrik ermöglichen es, Eigenschaften wie die strukturelle Komplexität, die Programmlänge oder den Grad der Kommentierung quantitativ zu ermitteln. Die Ergebnisse gestatten einen Quervergleich mit bisherigen Erfahrungswerten. Abweichungen deuten auf eine Anomalie hin, sowohl im positiven als auch im negativen Sinne. Eine genauere Untersuchung ist dann angebracht. Tabellen und Grafiken erlauben es, ein Programm unter speziellen Gesichtspunkten zu analysieren und die Ergebnisse in übersichtlicher und interpretierbarer Form darzustellen, z.B. cross reference–Listen, Variablenverwendungslisten usw.

Verifizierende Verfahren dagegen beziehen sich ausschließlich auf das Beweisen der Korrektheit von Programmen. Bei der Programmverifikation wird mit formalen Mitteln der theoretischen Informatik versucht, das gegebene Programm auf Korrektheit hin zu prüfen. „Die Akzeptanz formaler Techniken in der ‚konventionellen' Softwareentwicklung ist relativ gering. Ursachen sind sicherlich die in vielen Fällen unzureichende Anwendbarkeit dieser Techniken und die höhere Bereitschaft, Risiken aufgrund von Restfehlern einzugehen " *Liggesmeyer* (2005b). Hinzukommen mangelhafte Rahmenbedingungen, wie z.B. das Fehlen einer hinreichend formalen Spezifikation oder eines geeigneten Werkzeugs.

Testende Verfahren haben das Ziel, Fehler zu erkennen und lassen sich ebenfalls in weitere Kategorien einteilen (s. Abb. A.1): den statischen und dynamischen Test.

Statische Testverfahren analysieren den Quellcode, anstatt das SUT auszuführen, um Fehler zu finden. Am häufigsten werden für den statischen Test manuelle

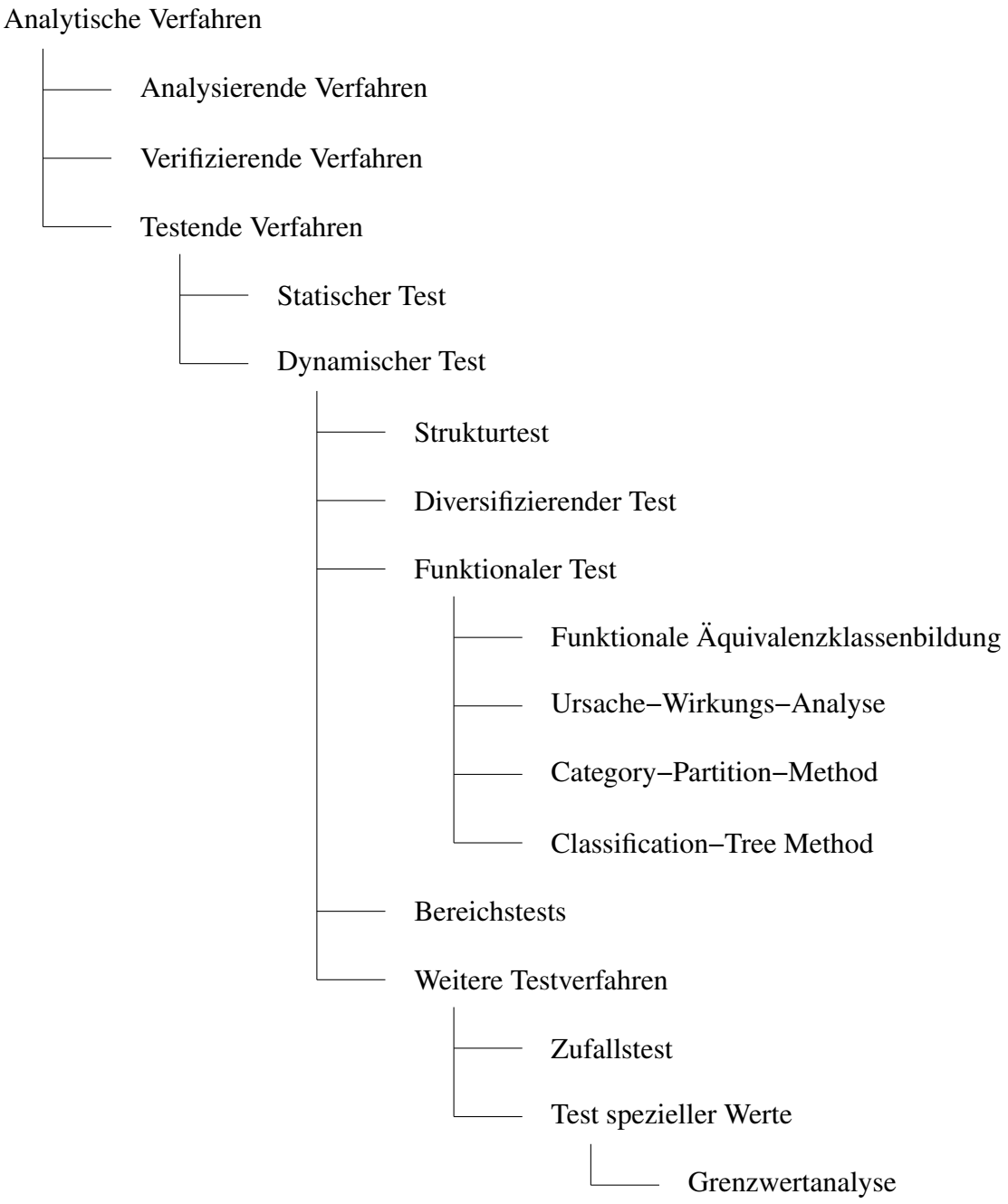


Abbildung A.1: Klassifikation analytischer Testverfahren (*Balzert*, 1998)

Prüfmethoden wie Code Inspections, Reviews und Walkthroughs eingesetzt. Dabei wird der Quellcode von einer Gruppe von Testern in Augenschein genommen. Bei Code Inspections wird versucht, die der Implementierung zugrunde liegende Logik zu erfassen. Walkthroughs dagegen verwenden einige wenige Testfälle, anhand derer der Programmablauf nachgestellt wird.

Dynamische Verfahren dagegen führen den Quellcode aus und sind Gegenstand des nächsten Abschnitts.

A.1 Dynamische Testverfahren

Dynamische Testverfahren stellen die wichtigsten analytischen Qualitätssicherungsmaßnahmen dar und unterteilen sich je nach Testreferenz in weitere Kategorien (s. Abb. A.1), die alle die folgenden gemeinsamen Merkmale besitzen (*Liggesmeyer*, 1990):

- „Das übersetzte, ausführbare Programm wird mit konkreten Eingabewerten versehen und ausgeführt."
- Das Programm kann in seiner realen Umgebung getestet werden.
- Es handelt sich um Stichprobenverfahren, d.h. die Korrektheit des getesteten Programms wird nicht bewiesen.

Da die dynamischen Testverfahren die Ausführung des Programms mit konkreten Testdaten fordern, ist eine Aussage über die korrekte oder fehlerhafte Funktion des Programms nur für die gewählten Testdaten möglich. Die korrekte Funktion des Programms für alle Eingaben kann daher nur durch den Test aller möglichen Eingaben sichergestellt werden. Dieser vollständige Test wird als erschöpfender Test bezeichnet. Seine Durchführung ist für reale Programme in der Regel nicht möglich, da unendlich viele, unterschiedliche Pfade existieren[1]. Schon *Dijkstra* formulierte diesen Sachverhalt im Jahre 1972 folgendermaßen: „Program testing can be used to show the presence of bugs, but never to show their absence." Aufgrund dieser „Schwäche" des Tests bedarf es einer Systematik bei der Auswahl der Testfälle.

Ziel von dynamischen Testverfahren ist nach *Liggesmeyer* (1990) die Erzeugung einer Stichprobe von Testfällen mit folgenden Eigenschaften:

- repräsentativ
- fehlersensitiv
- redundanzarm

[1] Bereits für ein triviales Programm, das zwei ganze Zahlen mit jeweils 16 Bit addiert, sind 2^{32} Kombinationen für einen vollständigen Test notwendig.

- ökonomisch

Die erzeugten Eingaben sollen geeignete Stellvertreter der möglichen Eingaben sein. Der dynamische Test hat als Ziel die Entdeckung von Fehlern, dementsprechend sollen die Testdaten geeignet sein, Fehler aufzuspüren. Außerdem sind eine geringe Redundanz und eine hohe Wirtschaftlichkeit in der ausgewählten Stichprobe von Testfällen anzustreben.

A.1.1 Klassifikation dynamischer Testverfahren

Dynamische Testverfahren (s. Abb. A.1) lassen sich weiter nach der ihnen zugrunde liegenden Testreferenz klassifizieren.

Strukturtest Werden die Testdaten anhand des Kontroll– oder Datenflusses des Programms abgeleitet, dann liegt ein dynamischer Strukturtest bzw. White–Box–Test vor (s. Anh. A.2). Die Struktur bzw. der Quellcode des Programms muss hierbei bekannt sein.

Diversifizierender Test Diversifizierende Tests, auch back–to–back Tests genannt, vergleichen Ergebnisse verschiedener, unabhängig voneinander entwickelter Programmversionen (n–version programming). Jedoch können auch durch künstlich eingefügte Fehler verschiedene Programmversionen erzeugt werden, die dann vergleichend getestet werden.

Funktionaler Test Beim Funktionstest, funktionalen Test bzw. Black–Box–Test wird die funktionale Spezifikation benutzt, um Testfälle abzuleiten. In Anh. A.3 werden die in Abb. A.1 aufgelisteten funktionalen Testverfahren näher betrachtet.

Bereichstest Der Bereichstest geht davon aus, dass sowohl Spezifikation als auch Implementierung alleine nicht ausreichend sind, um ein Programm ausreichend zu testen. Der Bereichstest ist damit eine Mischung aus Struktur– und funktionalem Test.

Weitere Testverfahren Weiterhin gibt es Testverfahren, die sich nicht so einfach in die bisher genannten Kategorien einteilen lassen. Diese Verfahren werden unter dem Begriff weitere Testverfahren zusammengefasst. Dazu gehören z.B. der Zufallstest, der Test spezieller Werte und die Grenzwertanalyse. Auch diese Testverfahren werden in einem separaten Abschnitt (s. Anh. A.4) näher betrachtet.

A.2 Strukturorientierte Testverfahren

Strukturorientierte Testverfahren leiten die Testfälle aus der inneren Struktur (z.B. Kontrollflussgraph) des SUT (Software) ab. Im Folgenden werden die drei bekanntesten strukturorientierten Testverfahren bzw. deren Überdeckungsmaße kurz vorgestellt.

Anweisungsüberdeckung Der Anweisungsüberdeckungstest stellt den kleinsten strukturorientierten Überdeckungstest dar. Jede Anweisung im Kontrollflussgraph des SUT wird dabei mindestens einmal ausgeführt.

Zweigüberdeckung Auch der Zweigüberdeckungstest basiert auf dem Kontrollflussgraphen und fordert die mindestens einmalige Ausführung jeder darin enthaltenen Kante bzw. jedes enthaltenen Zweiges. Der Zweigüberdeckungstest beinhaltet damit auch den Anweisungsüberdeckungstest und wird in der Praxis als das Minimalkriterium für den Strukturtest von Softwaresystemen angesehen.

Pfadüberdeckung Im Gegensatz zu den zwei soeben vorgestellten Testverfahren ist der Pfadüberdeckungstest in der Praxis meist nicht durchführbar. Er fordert die vollständige Ausführung aller existierenden Programmpfade und stellt somit das umfassendste strukturorientierte Testverfahren dar.

A.3 Funktionale Testverfahren

Bei funktionalen Testverfahren werden die Testfälle, im Gegensatz zu den strukturorientierten Testverfahren, allein aus der funktionalen Spezifikation abgeleitet. Damit ist das SUT für den Testersteller mit Ausnahme der Spezifikation ein schwarzer Kasten, daher auch der Name Black−Box−Test.

Die Begründung, ein Programm gegen seine funktionale Spezifikation zu testen, liegt darin, dass der Strukturtest nicht in der Lage ist, fehlende Funktionalitäten des Programms aufzudecken. Allein der Vergleich der Implementierung mit der Spezifikation durch den Funktionstest erkennt derartige Fehler zuverlässig, sofern die in Abb. 2.3 vorgestellte Wirkkette zur Fehlererkennung vollständig durchlaufen wurde. Der Funktionstest ist hingegen nicht in der Lage, die konkrete Implementierung geeignet zu berücksichtigen. Es werden keine Informationen darüber geliefert, ob alle implementierten Funktionen des Programms benötigt werden, oder ob Datenobjekte manipuliert werden, die keinen Einfluss auf das Ein− bzw. Ausgabeverhalten des SUT haben.

Da die Testfälle bei funktionalen Testverfahren allein aus der Spezifikation abgeleitet werden, erfüllt ein vollständiger Funktionstest in der Regel nicht die Minimalanforderungen einfacher Strukturtests. Untersuchungen zeigen, dass ein Funktionstest oft nur zu einer Zweigüberdeckungsrate von ca. 70 % führt (*Balzert*, 1998).

Der Test von E/E−Komponenten durch den Automobilhersteller lässt sich in diese Kategorie einordnen. Die Zulieferer geben ihr Know−how nicht preis und liefern eine Black−Box, die sie anhand der funktionalen Spezifikation entwickelt haben. Die Testfälle werden beim Automobilhersteller durch Testersteller ebenfalls aus dieser funktionalen Spezifikation abgeleitet. Diese Aktivität der Testfallermittlung (vgl. Kap. 2.4) gilt es bzgl. der in Anh. A.1

aufgeführten Kriterien zu optimieren, denn sie beeinflusst die Qualität des Tests in besonderer Weise.

Die wichtigsten funktionalen Testverfahren werden im Folgenden kurz vorgestellt. Für einen tieferen Einblick wird auf die zahlreich existierende Literatur verweisen (z.B. *Liggesmeyer* (1990), *Riedemann* (1997), *Spillner & Linz* (2005)).

A.3.1 Funktionale Äquivalenzklassenbildung

Das Prinzip der funktionalen Äquivalenzklassenbildung besteht darin, die Ein– und Ausgabedaten in so genannte Äquivalenzklassen einzuteilen. Nach *Myers* (1979) gelten folgende Regeln zur Bildung von Äquivalenzklassen:

- Falls eine Eingabebedingung einen Wertebereich spezifiziert, so sind eine gültige Äquivalenzklasse und zwei ungültige Äquivalenzklassen zu bilden. Beispiel:

Eingabebereich	$1 \leq x \leq 10$	$x \in \mathbf{N}, \mathbf{R}$
Eine gültige Äquivalenzklasse	$1 \leq x \leq 10$	
Zwei ungültige Äquivalenzklassen	$x < 1, \quad x > 10$	

- Falls eine Eingabebedingung eine Menge von Werten spezifiziert, die unterschiedlich behandelt werden, so ist für jeden Wert eine eigene Äquivalenzklasse zu bilden. Für alle Werte, mit Ausnahme der gültigen Werte, ist eine ungültige Äquivalenzklasse zu bilden. Beispiel:

Eingabebedingung	Monatsname
Zwölf gültige Äquivalenzklassen	Januar, ..., November, Dezember
Eine ungültige Äquivalenzklasse	z.B. Julian

- Falls eine Eingabebedingung eine Situation festlegt, die zwingend erfüllt sein muss, so ist eine gültige Äquivalenzklasse und eine ungültige Äquivalenzklasse zu bilden. Beispiel:

Eingabebedingung	Erstes Zeichen muss eine Zahl sein.
Eine gültige Äquivalenzklasse	Erstes Zeichen ist eine Zahl.
Eine ungültige Äquivalenzklasse	Erstes Zeichen ist keine Zahl.

- Falls Grund zu der Annahme besteht, dass Elemente einer Äquivalenzklasse unterschiedlich behandelt werden, so ist diese Äquivalenzklasse entsprechend aufzutrennen.

Die Äquivalenz der Werte einer Klasse bezüglich eines Tests bedeutet, dass jeder Wert der Klasse entweder ein Fehlverhalten hervorruft oder dass alle Werte der Klasse zu einem korrekten Verhalten führen. Für jede Äquivalenzklasse wird schließlich ein Repräsentant ausgewählt, und nur dieser wird bei einem Test berücksichtigt. Damit sind nur die Kombinationen

der Repräsentanten zu testen und nicht alle möglichen Kombinationen der Eingabedaten. Daraus resultiert eine deutliche Testfallminderung.

Im Zusammenhang mit der Methode der funktionalen Äquivalenzklassenbildung wird von gültigen und ungültigen Äquivalenzklassen gesprochen. Eine gültige Äquivalenzklasse enthält gültige Eingaben, also Werte für die ein definierter Wertebereich vorgesehen ist. Alle Werte außerhalb dieses Wertebereichs werden ungültigen Äquivalenzklassen zugeordnet. Eine ungültige Äquivalenzklasse besteht daher aus unerlaubten, aber möglichen Eingabewerten. Eine Behandlung der ungültigen Äquivalenzklassen ist essentiell, um festzustellen wie sich der Prüfling in diesen Situationen verhält. *Bergsmann & Achtert* (2003) formulieren diesen Sachverhalt folgendermaßen: „Ein Programm muss nicht nur untersucht werden, um festzustellen, ob es nicht tut was es tun soll, sondern es muss auch untersucht werden, ob es etwas tut, was es nicht tun soll.“ Ein Test einer gültigen Äquivalenzklasse kann daher als Funktionstest und ein Test einer ungültigen Äquivalenzklasse als Stresstest bezeichnet werden.

Zu Testfällen gelangt man, indem möglichst viele gültige Äquivalenzklassen miteinander kombiniert werden. Sind alle gültigen Äquivalenzklassen geprüft, so berücksichtigt man pro weiteren Testfall jeweils eine ungültige Äquivalenzklasse. Dies findet so lange statt, bis alle ungültigen Äquivalenzklassen durch Testfälle abgedeckt sind.

A.3.2 Ursache–Wirkungs–Analyse

Die Ursache–Wirkungs–Analyse beruht auf der Arbeit von *Elmendorf* (1973) und berücksichtigt, im Gegensatz zur Äquivalenzklassenbildung, Wechselwirkungen und Abhängigkeiten zwischen Ein– und Ausgabedaten. Auf diesem Weg können Fehler entdeckt werden, die an bestimmte Eingabekombinationen gebunden sind. Als Hilfsmittel kommt der Ursache–Wirkungs–Graph zum Einsatz, welcher die Beziehungen zwischen den Ursachen und den Wirkungen sowie weitere Abhängigkeiten enthält. Aus dem Graphen werden schließlich Testfälle abgeleitet.

A.3.3 Category–Partition–Method

Die Category–Partition–Method wurde von *Ostrand & Balcer* (1988) entwickelt, und geht wie die Ursache–Wirkungs–Analyse ebenfalls von einer Kombination unterschiedlicher Ursachen aus. Der Unterschied liegt jedoch darin, dass keine Überlegungen bezüglich der Wirkungen bzw. der Sollergebnisse in die Testfallermittlung einfließen (*Schmid*, 2003). Die Spezifikation der Äquivalenzklassen und aller anderen Elemente der Category–Partition–Method erfolgt in einer Skriptsprache, der so genannten Test Specification Language.

A.3.4 Classification–Tree Method

Die Classification–Tree Method (*Grimm*, 1995) wurde Anfang der 90er Jahre von Ingenieuren bei der Daimler–Benz AG entwickelt. Sie baut auf der Category–Partition–Method auf und dient der systematischen Analyse des Eingabedatenraums eines Testobjekts. Dabei wird der Eingabedatenraum nach verschiedenen Aspekten klassifiziert. Die Kombination der verschiedenen Aspekte führt schließlich zu Testfällen. Zudem existiert ein grafisches Werkzeug, das die Methode unterstützt. Näheres zur Classification–Tree Method, als ausgewähltes Testverfahren, findet sich in Kap. 5.

A.4 Weitere Testverfahren

Einige Testverfahren können keiner der bisher vorgestellten Kategorien eindeutig zugeordnet werden, da sie zum Teil völlig unterschiedliche Zielsetzungen verfolgen. Diese Testverfahren sind in diesem Unterabschnitt zusammengefasst. Sie sind weniger als alleinige Verfahren anzuwenden, sondern stellen ergänzende bzw. optimierende Verfahren dar, die in Kombination mit den vorgestellten funktionalen Testverfahren anzuwenden sind.

A.4.1 Zufallstest

Beim Zufallstest wird die Belegung der Eingangsvariablen für jeden Testfall zufällig gewählt. Somit liegt dem Zufallstest keine deterministische Strategie zugrunde.

A.4.2 Test spezieller Werte

Dieser Oberbegriff fasst eine Reihe von Testverfahren zusammen, die für die Testdaten bestimmte Aspekte fordern. Der Test spezieller Werte ist damit in eigentlichem Sinne kein Testverfahren, da ihm kein generelles Konzept zugrunde liegt. Er dient vielmehr der Ergänzung oder Optimierung von Testfällen. Die Testfallermittlung erfolgt auf Basis von Erfahrungswerten, wobei der Testersteller seiner Kreativität freien Lauf lassen kann. Ein guter Testersteller entwickelt mit der Zeit ein Gefühl für Fehler. Fehleranfällige Programmteile, Tücken einer Programmiersprache und Schwierigkeiten beim Portieren auf eine bestimmte Plattform sind dem Testersteller gut bekannt. Außerdem ist er sich der Unvollständigkeit von Spezifikationen bewusst und misstraut sogar Compilern und anderen Werkzeugen, da auch diese Fehler enthalten können.

A.4.2.1 Grenzwertanalyse

Das bekannteste Testverfahren aus der Familie des Tests spezieller Werte ist die Grenzwertanalyse. Deshalb wird sie hier als gesonderter Punkt behandelt. Grundlage für die Grenz-

wertanalyse bildet die Erfahrung, dass Grenzbereiche besonders häufig mit Fehlern belastet sind. Dies erklärt sich zum Teil durch Vertauschung der Operatoren „$<$“ und „$\leq$“ sowie „$>$“ und „$\geq$“ bei der Programmierung.

B EXtended Automation Method (EXAM)

Dieses Kapitel beschreibt die im Volkswagen Konzern eingesetzte Plattform zur Automatisierung von Tests im Bereich von Hardware–in–the–Loop Testsystemen. Der Fokus liegt dabei auf Aspekten, die für diese Arbeit relevant sind. Sie werden entsprechend detailliert vorgestellt. Die restlichen Aspekte der Automatisierungsplattform werden der Vollständigkeit halber erwähnt, auf Details wird dabei jedoch nicht näher eingegangen.

B.1 Einleitung

Aufgrund des immer höheren Stellenwertes von Elektronik im Automobil wird der Entwicklungsprozess von Automobilelektronik immer mehr zum bestimmenden Faktor in der Fahrzeugentwicklung (vgl. Kap. 3.4). Zur Sicherstellung der hohen Sicherheits– und Qualitätsanforderungen kommt dem Testprozess eine besondere Bedeutung zu. Dabei werden im gesamten Volkswagen Konzern zahlreiche Testsysteme (vgl. Kap. 3.5.1) eingesetzt, die mittels verschiedenster Testautomatisierungslösungen betrieben werden.

„Ein Prozess, der die Komplexität eines Fahrzeugs und alle erforderlichen Varianten berücksichtigt, führt sehr schnell in die Problematik einer Softwareentwicklung (*Schieber & Derichsweiler*, 2004)“. Anforderungen die an Softwaresysteme gestellt werden, gelten daher auch für Testsysteme. Dies sind:

- Strukturierung
- Versionierung
- Verwaltung und Organisation von schnell wachsenden Codebeständen
- Portabilität

Als spezielle Anforderungen für Testsysteme nennen *Schieber & Derichsweiler* (2004) zusätzlich:

- Regressionsfähigkeit

- Testautomatisierung

Zudem fordern sie den Einsatz von Standards und Werkzeugen, die sich in der klassischen Softwareentwicklung bewährt haben.

B.2 Konzerneinheitliche Testautomatisierung EXAM

Um der heterogenen Tool– und Prozesslandschaft sowie der stetig steigenden Systemkomplexität entgegenzuwirken, wurde die konzerneinheitliche Testautomatisierung EXtended Automation Method (EXAM) entwickelt. „EXAM ist ein strategisches Projekt des Volkswagen Konzerns zur Bereitstellung einer leistungsfähigen Plattform für eine gemeinsame, abgestimmte Entwicklung von Test–Programmen für die Erprobung von elektronischen Steuergeräten mittels Prüfstandstechnik im Allgemeinen bzw. Hardware–in–the–Loop–Systemen im Besonderen“ (*Köhler & Kiffe*, 2006).

Wesentliche Merkmale von EXAM sind:

- grafische Modellierung von Testinhalten mit der Unified Modeling Language (UML)
- Trennung zwischen Testbeschreibung und ihrer Implementierung
- Mehrfachnutzung von Funktionsbibliotheken
- Rollentrennung
- konzernweite Entwicklung von Tests im Team
- automatische Generierung von prüfstandspezifischem Code

Testinhalte werden nicht mehr für einen bestimmten HIL Simulator programmiert, sondern mit der UML plattformunabhängig modelliert. Diese Trennung zwischen Testbeschreibung und ihrer Implementierung fördert eine hohe Wiederverwendungsrate bzw. Austauschbarkeit der modellierten Testinhalte und Funktionsbibliotheken im Volkswagen Konzern. Tests können so konzernweit in übergreifenden Teams entwickelt werden, was Zeit und damit auch Kosten spart. Die freiwerdenden Ressourcen können z.B. der intensiveren Erprobung dienen und somit zu einer Steigerung der Qualität führen. Jeder Anwender von EXAM nimmt eine oder mehrere definierte Rollen ein und konzentriert sich auf die mit der jeweiligen Rolle verbundenen Aufgaben. Weiteres Merkmal von EXAM ist die automatische Generierung von plattformspezifischem Code ausgehend von der plattformunabhängigen Testbeschreibung durch UML–Modelle. Damit verfolgt EXAM in einem weiteren Sinne den Ansatz der Model Driven Architecture[1] (*Kiffe*, 2006).

[1] Die Model Driven Architecture (MDA) ist ebenso wie UML ein OMG–Standard, der eine Softwarearchitektur zum Ziel hat, die Geschäfts– und Applikationslogik von der darunter liegenden Plattformtechnologie trennt (*Schläfer*, 2004).

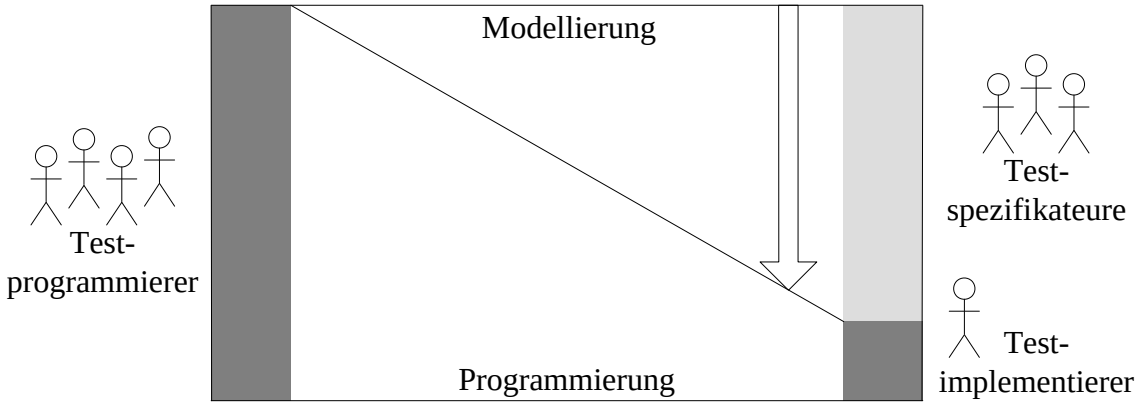


Abbildung B.1: Anteile Programmierung und Modellierung (*Voigt*, 2006)

Abb. B.1 zeigt die Aufspaltung der ursprünglich reinen Programmierung in Programmierung und Modellierung. Die ursprünglichen Testprogrammierer, die sowohl Spezialkenntnisse in der Software–Entwicklung als auch Spezialkenntnisse bzgl. der Testinhalte vorweisen mussten, werden in zwei Gruppen[2] geteilt. Testimplementierer kümmern sich nur noch um die Programmierung bzw. Entwicklung von Funktionsbibliotheken, während sich die Testspezifikateure ausschließlich mit der plattformunabhängigen Modellierung der Testinhalte beschäftigen. Jede Gruppe konzentriert sich auf ihr Spezialgebiet und kann dadurch effizienter arbeiten.

Ermöglicht wird diese Trennung zum einen durch die Verwendung von standardisierten Funktionsbibliotheken sowie zum anderen durch die Verfolgung einer konsequenten Interface– und Single–Source–Strategie. Zudem kommt ein CASE–Tool, das ARTiSAN Studio zum Einsatz.

Da EXAM der ständigen Weiterentwicklung[3] unterworfen ist, wird im Folgenden nur ein kleiner und für die weitere Arbeit relevanter Einblick in EXAM gegeben. Tiefergehende Details sind den erwähnten Literaturangaben zu entnehmen.

B.3 EXAM–Prozessmodell

„EXAM versteht sich als System für Design und die Durchführung von Testfällen. Es übernimmt gemäß Definition keine Requirementsengineering–, Testtracking– oder Test–Management–Aufgaben, es existieren aber Schnittstellen zu benachbarten Systemen (z.B. DOORS)“ *Köhler & Kiffe* (2006).

In Abb. B.2 ist das EXAM–Prozessmodell dargestellt, wobei die Abgrenzung von EXAM

[2]Weitere Gruppen bzw. Rollen werden in Anh. B.4.1 vorgestellt.
[3]Diese Arbeit basiert auf der EXAM Version 0.8 (Stand Oktober 2006).

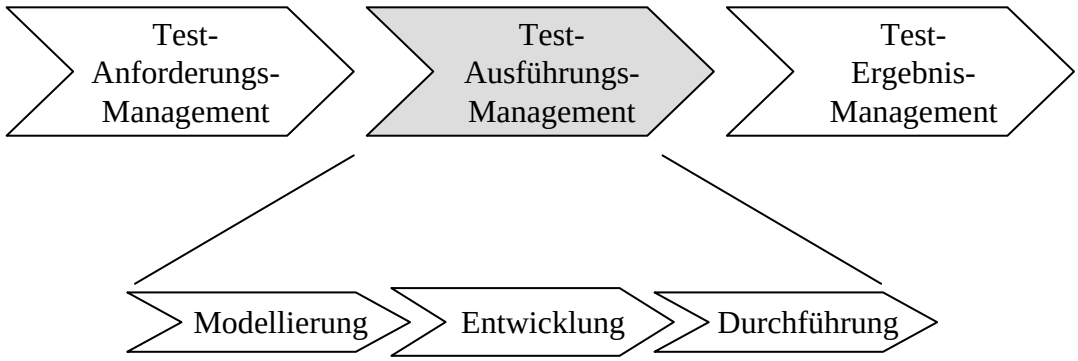


Abbildung B.2: EXAM–Prozessmodell (*Kiffe*, 2006)

bzw. dem Test–Ausführungs–Management zum Test–Anforderungs–Management sowie zum Test–Ergebnis–Management folgendermaßen realisiert ist.

Die Sammlung von Testideen, die Analyse des SUT, die Erstellung informeller Testspezifikationen sowie die Erteilung eines Testauftrags ist Bestandteil des Test–Anforderungs–Management. Die informellen Testspezifikationen stellen die Grundlage für die spätere Modellierung, Entwicklung und Durchführung von Tests in EXAM dar. Nach Durchführung der Tests wird das Testergebnis in Form eines Testberichts festgehalten. Die anschließende Interpretation des Testergebnisses sowie die eventuelle Einleitung eines Fehlerabstellprozesses (vgl. Kap. 2.3) ist Teil des Test–Ergebnis–Management.

Blicken wir auf die in Kap. 2.4 vorgestellten Testaktivitäten zurück, so lässt sich folgende Zuordnung zu dem soeben vorgestellten Prozessmodell vornehmen. Die Aktivitäten Testplanung, Testfallermittlung und Sollergebnisbestimmung fallen gemäß EXAM dem Test–Anforderungs–Management zu. Alle anderen Testaktivitäten lassen sich dem Test–Ausführungs–Management zuordnen, wobei die Testdatengenerierung der Modellierung und Generierung entspricht und Testdurchführung, Monitoring und Testauswertung der Durchführung zuzuordnen sind. Die testübergreifenden Aktivitäten Testorganisation und Testdokumentation stellen auch in EXAM übergreifende Tätigkeiten dar und betreffen daher das gesamte Test–Ausführungs–Management.

Die vorliegende Arbeit konzentriert sich auf die beiden Bereiche Test–Anforderungs–Management und Test–Ausführungs–Management, wobei aus dem Test–Ausführungs–Management nur die Modellierungsphase relevant ist. Das Test–Ergebnis–Management wird nicht weiter betrachtet. Bisher wird das Test–Anforderungs–Management von Tools wie z.B. DOORS, Word und Excel zur Spezifikation der Testfälle unterstützt (s. Abb. B.3). Auch die UML bzw. das ARTiSAN Studio als wesentlicher Bestandteil von EXAM wird zur Erstellung der informellen Testspezifikationen eingesetzt. Eine systematische Testfallermittlung ist dabei jedoch selten. Testfälle werden, wie in der Praxis üblich (vgl. Kap. 2.4), größtenteils Ad–hoc oder mittels der „Technik“ des „Fehler erraten“ ermittelt.

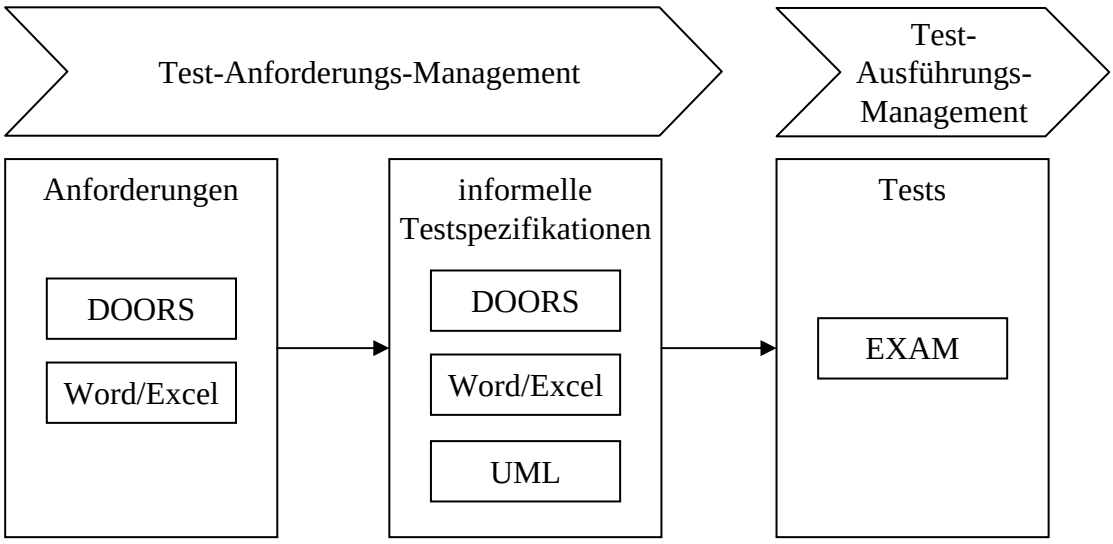


Abbildung B.3: Eingesetzte Tools im EXAM−Prozessmodell

Mit Hilfe der in Kap. 5 vorgestellten CTM und CTM/ES kann das Test−Anforderungs−Management bzw. speziell die wichtigste Testaktivität der Testfallermittlung systematisiert und durch den Einsatz des CTE XL geeignet unterstützt werden. Aus diesem Grund wird die Integration der beiden systematischen Testverfahren sowie des zugehörigen Tools in das soeben skizzierte Prozessmodell angestrebt. Ein entsprechendes Integrationskonzept wird in Kap. 6 vorgestellt und in Kap. 7 einem Praxistest unterzogen. Die finale Realisierung behandelt Kap. 8.

B.4 Der EXAM−Testprozess

„Zur Durchführung einer stufenweisen Integration der Elektronikkomponenten und deren Funktionsabsicherung ist ein eigener Prozess notwendig“ (*Schieber & Derichsweiler*, 2004). Dieser Testprozess wurde bereits in Kap. 3.4.3 vorgestellt. Im Folgenden wird der Testprozess mit Fokus auf HIL Testsysteme sowie die konzerneinheitliche Testautomatisierung EXAM betrachtet.

Neben dem ARTiSAN Studio als Modellierungswerkzeug für Testinhalte wird EXAM von weiteren speziell entwickelten Tools[4] unterstützt. Die entsprechende Toolkette ist in Abb. B.4 abgebildet.

[4] Das UML−Modell stellt die einzige Informationsquelle dar (Stichwort Single−Source). Die eingesetzten Tools verarbeiten die darin modellierten Testinhalte mithilfe des OLE Automation Interface des ARTiSAN Studio. Sie sind standardmäßig auf jedem Client unter C:\EXAM installiert. Zusätzlich existieren einige weitere hier nicht aufgeführte Tools.

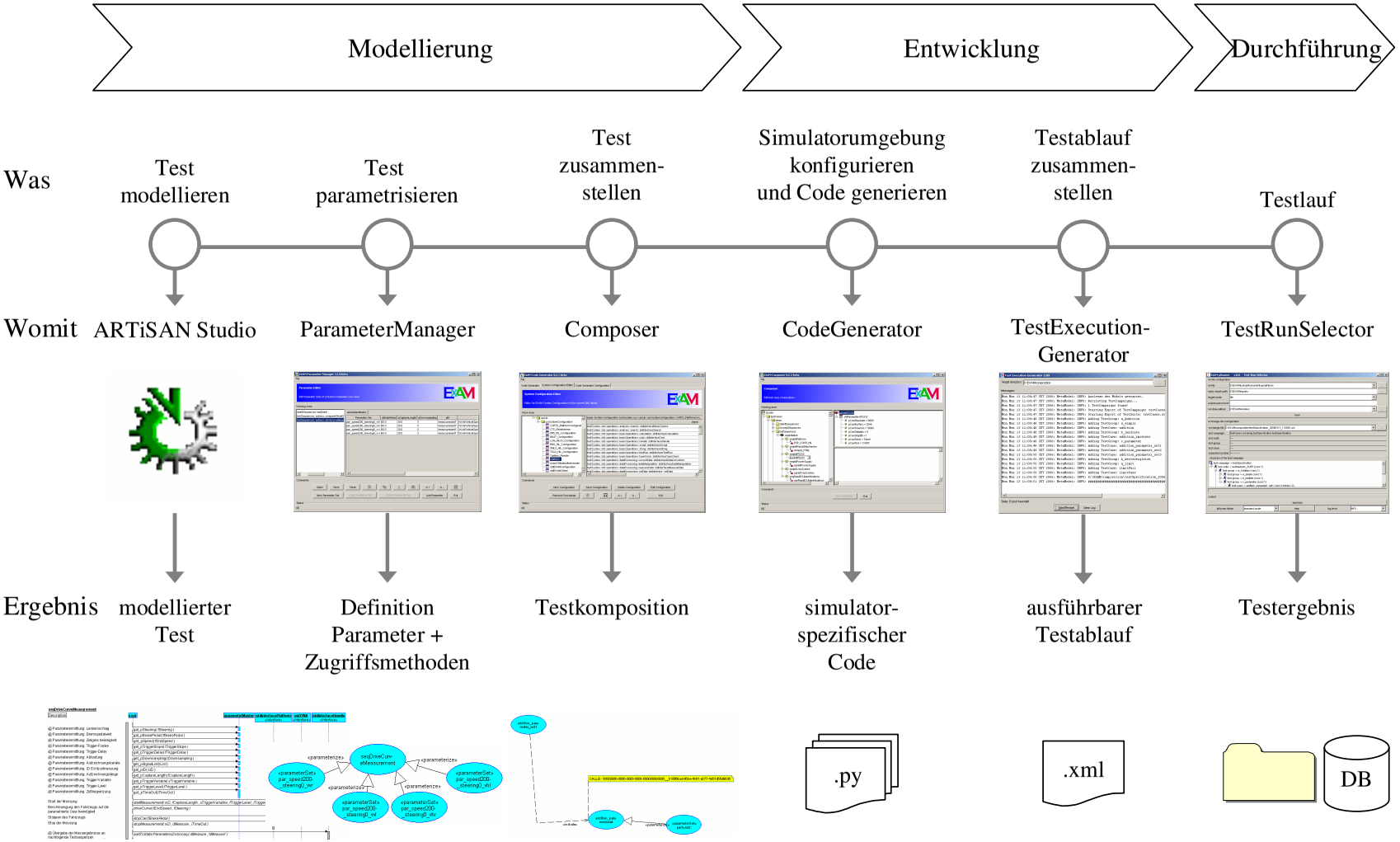


Abbildung B.4: Testprozess und Toolkette in EXAM (*Voigt*, 2006)

Zunächst werden ausgehend von informellen Testspezifikationen in der Modellierungsphase Tests im ARTiSAN Studio modelliert und bei Bedarf durch geeignete Parametersätze parametrisiert. Dadurch können Tests z.B. an verschiedene Randbedingungen angepasst werden, wodurch der Testablauf in unterschiedlichen Szenarien wieder verwendbar ist. In der sich anschließenden Testkomposition erfolgt die Zuweisung von Parametersätzen zu Tests, was die Zusammenstellung des späteren Testablaufs ermöglicht. Inhalt der Entwicklungsphase ist die Konfiguration der Simulatorumgebung sowie die Generierung des plattformspezifischen Codes in Form von Python Dateien. Der TestExecutionGenerator liefert ausgehend vom modellierten Testablauf eine .xml−Datei, die den ausführbaren Testablauf beinhaltet und als Schnittstelle zur Durchführungsphase fungiert. In der Durchführungsphase kann der in der .xml−Datei enthaltene Testablauf noch geringfügig variiert (z.B. Auskommentieren einzelner Tests, Mehrfachausführung von Tests) werden, bevor er auf dem ausgewählten HIL Simulator ausgeführt wird. Ergebnis der Durchführungsphase ist ein Testergebnis, das entweder in einer Datenbank (DB) oder in einem Filesystem abgelegt wird. Anhand dieses Testergebnisses lässt sich ein Testbericht generieren.

B.4.1 Rollen im Testprozess

In Anh. B.1 wurde die Gruppen− bzw. Rollenaufteilung als ein wesentliches Merkmal in EXAM angesprochen und in Abb. B.1 sind bereits zwei erste Benutzerrollen aufgeführt worden. Zuvor wurde im Zusammenhang mit dem Testprozess jedoch immer der Begriff des Testers bzw. Testerstellers, als dem Testprozess zugeordnete Person, verwendet. Dieser allgemeine Begriff wird im Folgenden weiter klassifiziert. U.a. sind folgende Rollen im und um den EXAM−Prozess definiert (*Kiffe*, 2006):

- **Testmanager** Die Verantwortung für das Testprojekt und dessen Durchführung und Koordination liegt beim Testmanager. Zudem erteilt er den Testauftrag gemäß einer informellen Testspezifikation.
- **EXAM−Anwender** Alle in EXAM vorkommenden Anwenderrollen sind unter dem Oberbegriff EXAM−Anwender zusammengefasst.
  - **Testadministrator** „Der Testadministrator verwaltet den Testpool bzw. einen Teil davon. Er hat den Überblick über vorhandene Testfälle und veröffentlicht ggf. neue Testfälle. Zudem ist er für die Versionierung und Standardisierung von Testfällen zuständig.“
  - **Testdesigner** Der Testdesigner ist wiederum ein Oberbegriff für folgende Rollen im Bereich der Modellierungs− und Entwicklungsphase.
    - **Testspezifikateur** Der Testspezifikateur ist für die Modellierung der formalen Testfälle in UML zuständig. Dabei abstrahiert er von der speziellen Ausprägung eines Testfalls und ermöglicht dadurch die Portierung des Testfalls auf andere Testsysteme und SUT.

- **Testimplementierer** „Der Testimplementierer modelliert alle konkreten Implementierungsklassen der EXAM−Bibliothek, die für die Ausführung eines abstrakten Testfalls erforderlich sind. Dies kann entweder in Plattformcode oder in UML erfolgen.“
- **Testcomposer** Der Testcomposer ordnet den TestSequenzen eines TestCase existierende Parametersätze zu.
- **Testablaufmodellierer** Der Testablaufmodellierer modelliert den Testablauf im UML−Modell.

- **Testanwender** Zum Aufgabenbereich des Testanwenders gehört die Ausführung vorhandener Testfälle sowie die Überwachung des Testablaufs.
- **Testergebnisreviewer** „Der Testergebnisreviewer sichtet die im Testablauf erzeugten Testergebnisse und prüft diese auf Korrektheit bzgl. des Ablaufs in EXAM. Er dokumentiert aufgetretene Probleme. Definitionsgemäß führt er keine inhaltliche/fachliche Bewertung der Ergebnisse durch.“

Jeder Rolle sind dabei zum Teil spezielle Werkzeuge zugeordnet. Der Testspezifikateur und der Testimplementierer z.B. operieren hauptsächlich im ARTiSAN Studio, während der Testanwender und der Testergebnisreviewer einige der anderen in Abb. B.4 dargestellten Tools einsetzen.

B.5 Das EXAM−UML−Modell

Das UML−Modell ( ) enthält alle in Abb. B.14 dargestellten Schichten mit Ausnahme der untersten zwei. Es befindet sich in einem speziellen Container (Repository) auf einem zentralen Server. Die EXAM−Anwender greifen über Clients auf das Repository zu. Durch diese Client−Server Architektur des ARTiSAN Studio können mehrere Anwender gleichzeitig an einem UML−Modell arbeiten, ohne sich gegenseitig zu behindern. Die bereits angesprochene Testentwicklung in konzernübergreifenden Teams wird hierdurch in besonderem Maße gefördert.

Die Strukturierung in einem UML−Modell ist durch Packages ( ) realisiert, womit auch schon das erste UML−Artefakt[5] eingeführt ist. Abb. B.5 zeigt die Grundstruktur von EXAM−UML−Modellen. Auf oberster Ebene befinden sich die zwei Packages `testCases`[6] und `testSystem`. Die Inhalte des Package `testCases` bauen auf den Inhalten des Package `testSystem` auf, so dass im Folgenden zuerst der Aufbau und die Elemente dieses Package vorgestellt werden.

[5] Ein UML−Artefakt stellt einen Bestandteil (Klasse, Beziehung, Diagramm, ...) eines UML−Modells dar (*Schläfer*, 2004).

[6] Wie bereits in Kap. 5.1.1.1 erläutert, kennzeichnet diese Textformatierung entsprechende Elemente aus referenzierten Abbildungen.

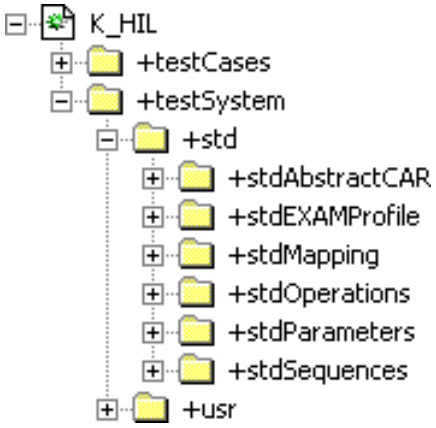


Abbildung B.5: Grundstruktur von EXAM−UML−Modellen am Beispiel des UML−Modells des vernetzten Komfortprüfstandes (`K_HIL`)

Dabei ist zu bemerken, dass EXAM zwar die Artefakte der UML in einer an die objektorientierte Analyse angelehnten Weise verwendet, es jedoch zum Teil deutliche Abweichungen und Erweiterungen bzgl. der bekannten Bedeutung der Artefakte gibt. Weiterhin verwendet EXAM nicht den vollen Umfang an Artefakten sondern lediglich eine ausgewählte Teilmenge davon (*Schläfer*, 2004).

Jedes UML−Artefakt verfügt im UML−Modell über zahlreiche Eigenschaften. Einige dieser Eigenschaften wie z.B. der Name, die Beschreibung oder die ID stehen jedem Artefakt zur Verfügung, andere Eigenschaften sind jedoch speziell an bestimmte Artefakte gebunden.

B.5.1 Package `testSystem`

Das Package `testSystem` enthält alle Funktionalitäten von EXAM, die zur Durchführung der Tests am Prüfstand benötigt werden, in Form einer standardisierten Packagestruktur. Die erste Strukturierungsebene bilden dabei Standardfunktionen (Package `std`) und Benutzerfunktionen (Package `usr`) (s. Abb. B.5).

Die Standardstruktur des Package `std` darf dabei nur von einem speziellen Gremium geändert werden, um die Austauschbarkeit von Funktionsbibliotheken, Mappings und Parameterklassen zwischen UML−Modellen verschiedener Prüfstände sicherzustellen. Das Package `stdOperations` z.B. ist, wie die anderen Packages auch, in weitere Subpackages unterteilt und enthält z.B. arithmetische Operationen, Operationen zur Variablendefinition sowie zur Initialisierung von verschiedenen HIL Simulatoren und diversen Netzgeräten *Raabe* (2005). Ebenso finden sich darin Operationen, um die Testergebnisse in einem Testbericht abzulegen. Die Inhalte der anderen Packages werden zum Teil im Laufe dieses Kapitels beschrieben.

Die Struktur des Package `usr` leitet sich aus dem Package `std` ab. Darin können die

Testdesigner und Testimplementierer alle benötigten Inhalte anlegen, die bisher im Standard nicht enthalten sind. Eine ständige Überprüfung der zahlreichen, über verschiedene UML–Modelle verteilten, Benutzerfunktionen auf standardisierbare Inhalte wird regelmäßig durchgeführt. Auf diese Weise wandern Inhalte aus dem Benutzerbereich in den Standard und stehen bei neuen Releases von EXAM allen EXAM–Anwendern zur Verfügung.

B.5.1.1 Grundbausteine im Package `testSystem`

Zur Modellierung und Programmierung der soeben erwähnten Inhalte verwendet EXAM, neben den bereits vorgestellten, im Wesentlichen folgende Artefakte der UML:

Klasse Funktionsbibliotheken, Parameter, das Mapping, Systemkonfigurationen sowie TestSuiten (s. Anh. B.5.2.1) werden in EXAM als Klassen dargestellt.

Klassen können folgende Artefakte enthalten:

- **Attribut** Attribute stellen Variablen innerhalb von Parameter– und Mappingklassen dar, deren Werte durch get– und set–Methoden gelesen und geschrieben werden können.
- **Operation** Alle Funktionen von Funktionsbibliotheken sind als Operationen von Klassen (so genannten Implementierungsklassen) realisiert. Die Funktionalität der Operation kann entweder als plattformspezifischer Programmcode oder plattformunabhängig in Form eines Objekt–Sequenz–Diagramms vorliegen.

  Operationen können folgende Artefakte enthalten.

  Parameter Parameter stellen Übergabewerte für Operationen dar.

  Objekt–Sequenz–Diagramm Das Objekt–Sequenz–Diagramm stellt die plattformunabhängige Implementierung von Operationen dar.
  Das Beispiel in Abb. B.7 zeigt den Inhalt des Objekt–Sequenz–Diagramms der Operation `Klemme15_Reset`. Darin wird als erstes die Mappingvariable `mSwKlemme15` durch die entsprechende set–Operation `set_mSwKlemme15` mit dem Wert 0.0 belegt. Dem folgt der Aufruf der Operation `wait`, wodurch Wartezeiten in EXAM realisiert werden. Nach 1000 Millisekunden wird der Wert von `mSwKlemme15` wieder auf 1.0 zurückgesetzt. Wie der Name des Objekt–Sequenz–Diagramms andeutet, handelt es sich hierbei um einen Reset der Klemme 15 mit der Dauer von einer Sekunde.

Eine besondere Klasse in EXAM stellt die abstrakte Klasse `root` dar (s. Abb. B.6(a)). Sie bildet den Einstiegspunkt in jedes Objekt–Sequenz–Diagramm und enthält weder Attribute noch Operationen.

Interface Das Interface stellt eine weitere spezielle Klasse (so genannte Interfaceklasse) dar. Es enthält nur abstrakte Operationen, d.h. die Operationen sind darin nur definiert, aber nicht implementiert. Die Implementierung liegt in Form der bereits erwähnten Implementierungsklassen als Programmcode oder als Objekt–Sequenz–Diagramm vor. Dabei müssen alle von der Interfaceklasse definierten Operationen von allen zugeordneten Implementierungsklassen implementiert werden.

Ein besonderes Interface in EXAM stellt das Interface `stdInterfaceControl` (s. Abb. B.6(b)) dar. Es stellt abstrakte Operationen zur Steuerung des Kontrollflusses in einem Objekt–Sequenz–Diagramm zur Verfügung. Eine zugeordnete Implementierungsklasse existiert hierbei nicht.

Klassen–Diagramm Im Klassen–Diagramm sind die Relationen zwischen Implementierungsklassen und deren Interfaceklassen mittels einer Generalisierungsbeziehung mit dem Stereotyp «implement» dargestellt.

Abb. B.8 zeigt ein beispielhaftes Klassen–Diagramm. Auf den Inhalt des dargestellten Klassen–Diagramms wird in Anh. B.5.1.3 näher eingegangen.

Stereotyp Mittels Stereotypen werden UML–Artefakte klassifiziert, wobei einem UML–Artefakt mehrere Stereotypen zugeordnet werden können. Dabei ist auch eine Anpassung des regulären Erscheinungsbilds des assoziierten UML–Artefakts möglich (s. Stereotypen «testCase», «testSequence» und «parameterSet»).

Die für diese Arbeit relevanten Stereotypen in EXAM werden an dieser Stelle nur kurz aufgelistet. Eine detailliertere Erläuterung erfolgt an der jeweiligen Stelle ihres konkreten Einsatzes.

«testCase» s. TestCase in Anh. B.5.2.1

«testSequence» s. TestSequenz in Anh. B.5.2.1

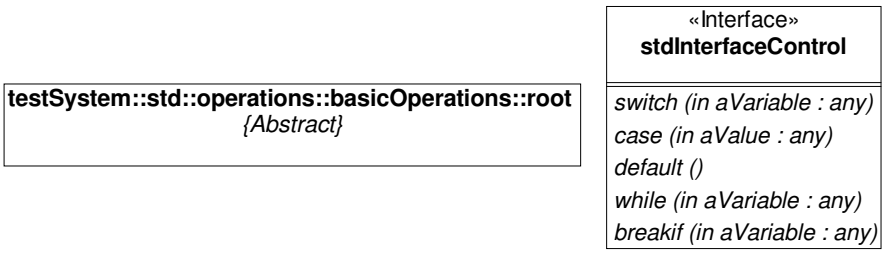
«parameterSet» s. Parametersatz in Anh. B.5.2.1

«parameterize» s. Use–Case–Diagramm in Anh. B.5.2.1

«testModule» s. Anh. 8.3

Stereotypen können weiterhin mit Tag Definitions assoziiert sein.

Tag Definition Tag Definitions spezifizieren den Wertebereich für so genannte Tagged Values. Der Stereotyp «testCase» ist z.B. mit der Tag Definition „duration“ assoziiert. Auf diese Weise kann ein UML–Artefakt als Testfall deklariert und mit zusätzlichen Informationen wie z.B. der Dauer des Testfalls versehen werden. Stereotypen und Tag Definitions sind im Ordner `stdEXAMProfile` (s. Abb. B.5) definiert.



(a) `root` (b) `stdInterfaceControl`

Abbildung B.6: Zwei besondere Klassen in EXAM

B.5.1.2 Systemkonfiguration

Ein UML–Modell enthält zahlreiche Interface– und Implementierungsklassen. Welche spezielle Implementierungsklasse bei einem Testablauf berücksichtigt werden soll, wird in der Systemkonfiguration definiert. Dabei wird jeder Interfaceklasse genau eine Implementierungsklasse zugeordnet. Dadurch können die prüfstandsunabhängig formulierten Testinhalte an verschiedene HIL Plattformen mittels verschiedenen Systemkonfigurationen sehr leicht angepasst werden.

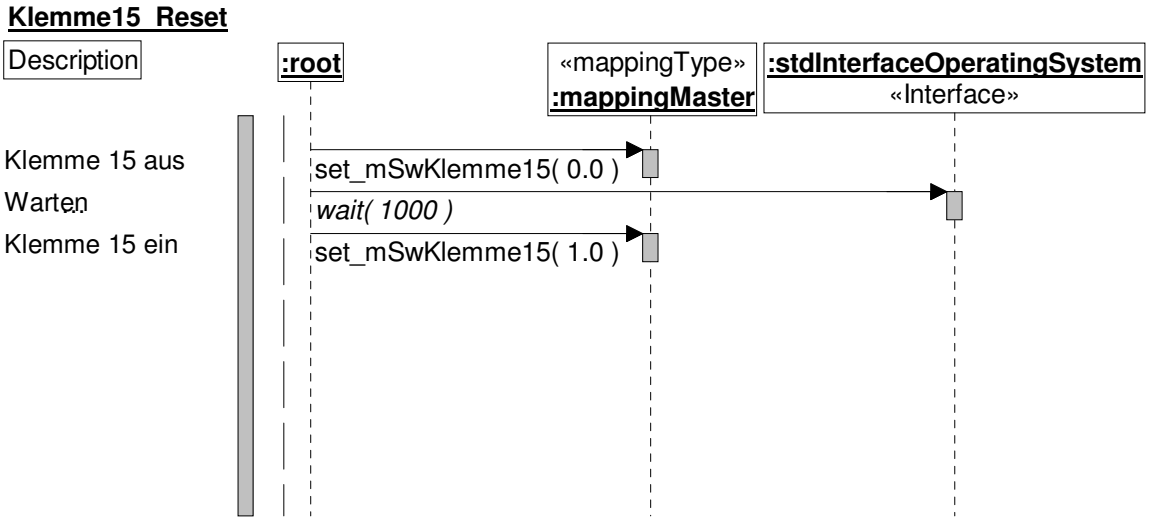


Abbildung B.7: Beispiel eines Objekt–Sequenz–Diagramms

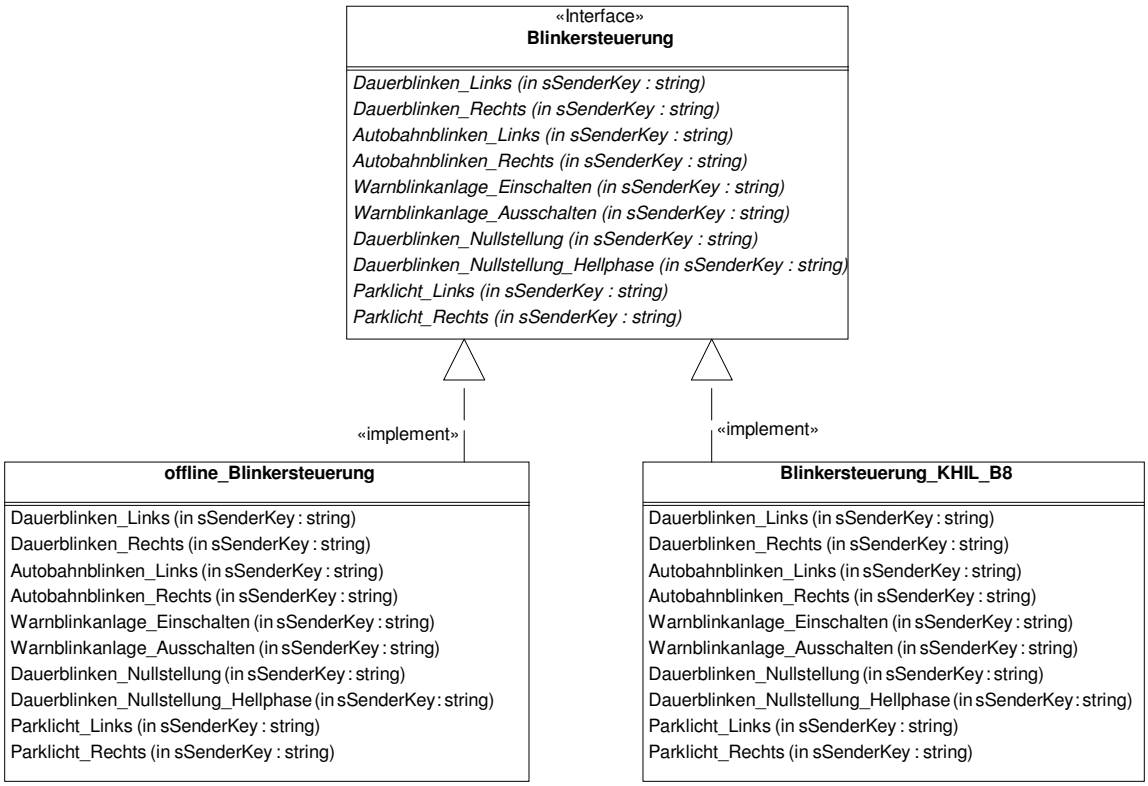


Abbildung B.8: Beispiel eines Klassen−Diagramms

B.5.1.3 Package `stdAbstractCAR`

Besonders hervorzuheben im Package `testSystem` ist das Package `stdAbstractCAR`. Es stellt eine Funktionsbibliothek dar, die neben der abstrakten Abbildung zahlreicher Fahrzeugfunktionen aus Fahrersicht (`FahrzeugFahrer`), die Fahrzeugumgebung beeinflussende Operationen im Package `Umgebung` beinhaltet (s. Abb. B.9).

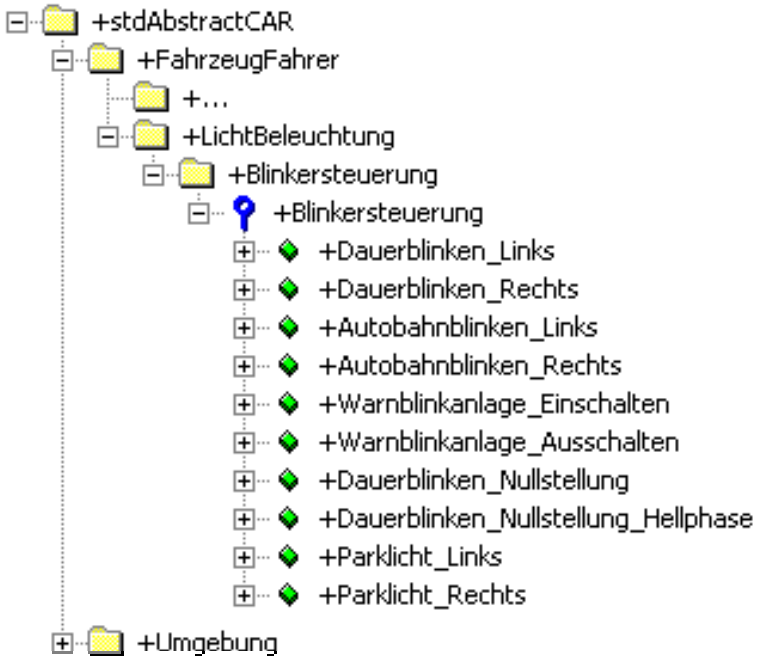


Abbildung B.9: Einblick in die Funktionsbibliothek `stdAbstractCAR`

So ist z.B. im Package `Blinkersteuerung` das gleichnamige Interface enthalten, welches abstrakte Operationen zur Steuerung der Blinker zur Verfügung stellt (z.B. `Dauerblinken_Links`, `Dauerblinken_Rechts`, `Dauerblinken_Nullstellung`). Das Package `Blinkersteuerung` gehört dem übergeordneten Package `LichtBeleuchtung` an, das wiederum dem Package `FahrzeugFahrer` untergeordnet ist. Auf diese Weise entsteht eine hierarchische, funktionsorientierte Packagestruktur, die alle wesentlichen Fahrzeug–Fahrer–Interaktionen beinhaltet.
Mittels Operationen aus dem Package `Umgebung` können z.B. Fahrbahnreibwerte und Fahrbahnsteigungen vorgegeben werden. Komplexere Inhalte wie z.B. die Herstellung eines Crashzustandes sind darin ebenso in abstrakter Form abgebildet.

Entsprechende Implementierungsklassen befinden sich im Package `usr` und enthalten die entsprechenden plattformspezifischen Inhalte. Ein Test, in dem z.B. die abstrakte Operation `Dauerblinken_Links` (Betätigen des Blinkerhebels für Linksblinken) aufgerufen wird, kann nach entsprechender Konfiguration des Systems auf mehreren Simulatorplattformen ausgeführt werden.

Das Klassen−Diagramm in Abb. B.8 zeigt den soeben beschriebenen Sachverhalt. Für die Interfaceklasse `Blinkersteuerung` existieren im vorliegenden Fall die zwei Implementierungsklassen `offline_Blinkersteuerung` und `Blinkersteuerung_K_HIL_B8`. Diese sind über eine Generalisierungsbeziehung mit dem Stereotyp «implement» mit der Interfaceklasse verknüpft. Die Systemkonfiguration (vgl. Anh. B.5.1.2) legt fest, welche dieser zwei möglichen Implementierungsklassen bei der automatischen Codegenerierung berücksichtigt wird.

B.5.2 Package `testCases`

Die Inhalte des Package `testCases` bauen auf den Inhalten des Package `testSystem` auf und enthalten durch den Testdesigner modellierte Testinhalte in Form von Testfällen sowie Inhalte, die den Testablauf der modellierten Testfälle festlegen. Anstatt Testfall verwendet EXAM den Begriff des TestCase[7].

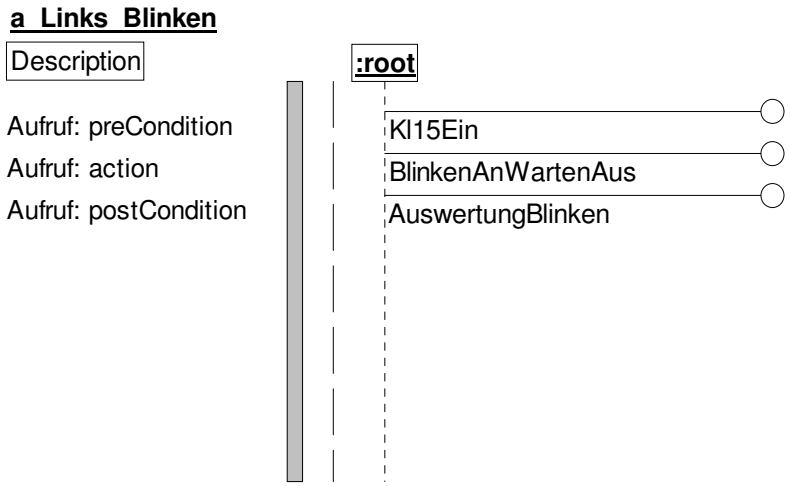


Abbildung B.10: Beispiel eines TestCase−Objekt−Sequenz−Diagramms

Eine einzuhaltende Strukturierung bei der Modellierung von TestCases ist nicht zwingend erforderlich. EXAM verwendet die in Abb. B.10 dargestellte Struktur. Danach gliedert sich ein TestCase in:

- Vorbedingung (preCondition)
- Aktion (action) und
- Nachbedingung (postCondition)

[7] Der EXAM−TestCase ist definiert als die „Abbildung genau eines (1) formal spezifizierten Testsachverhalts in einer abstrakten, formalen und strukturierten Beschreibungsform. Varianten eines gemeinsamen Ablaufs sind jeweils ein separater Testfall“ *Kiffe* (2006).

Die preCondition dient der Herstellung einer definierten Ausgangssituation des TestCase. Während der action wird die eigentliche Testaktion durchgeführt, und die postCondition enthält Operationen zur Bewertung sowie evtl. spezielle Nachbedingungen des TestCase. Diese Strukturierung sowie die Beschreibung des TestCase mittels Artefakten der UML in TestSequenzen wird in EXAM als formale Testspezifikation bezeichnet.

Ausgehend von der soeben vorgestellten Strukturierung von TestCases wird in EXAM für eine zusammengehörige Einheit von TestCases die in Abb. B.11 aufgezeigte Packagestruktur verwendet.

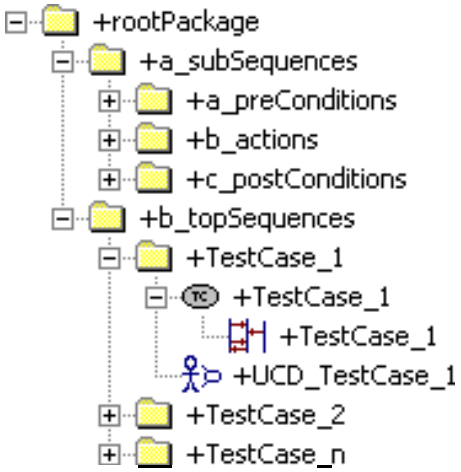


Abbildung B.11: Beispielhafte Packagestruktur zur Ablage von TestCases

Das Package `rootPackage` auf oberster Ebene spiegelt meist ein bestimmtes SUT (z.B. Steuergerät, Funktion, ...) wider. Der Packagename sollte daher entsprechend gewählt werden. Darin findet zunächst eine Aufteilung in die zwei Packages `a_subSequences` und `b_topSequences` statt. Die in Abb. B.10 inkludierten TestSequenzen sind je nach Typ (preCondition, action, postCondition) in den entsprechenden Subpackages des Package `a_subSequences` enthalten. Das Package `b_topSequences` enthält dagegen TestCases, die aufgrund ihres Inhaltes bzw. der Zugehörigkeit zum SUT gruppiert wurden. Eine weitere Unterteilung des Package `b_topSequences` nach Sub−SUT ist möglich.

B.5.2.1 Grundbausteine im Package `testCases`

Zur Modellierung der soeben erwähnten TestCases sowie zur Festlegung des Testablaufs in so genannten Testkampagnen verwendet EXAM im Wesentlichen folgende Artefakte der UML:

⬭ **Use−Case** Use−Cases stehen in EXAM je nach zugeordnetem Stereotyp für unterschiedliche Kontexte. Folgende Variationen von Use−Cases existieren:

TestCase Ein Use–Case, dem der Stereotyp «testCase» zugewiesen ist, wird innerhalb von EXAM als TestCase bezeichnet. TestCases symbolisieren konkrete Testfälle, denen immer ein Objekt–Sequenz–Diagramm untergeordnet ist, das den Testablauf in einer formalen Form (s.o.) abbildet.

Objekt–Sequenz–Diagramm Objekt–Sequenz–Diagramme, die TestCases untergeordnet sind (s. Abb. B.11), haben meist die in Abb. B.10 dargestellte Form. Anstatt Operationsaufrufen werden darin existierende TestSequenzen durch die so genannte Include–Beziehung aufgerufen. Im dargestellten Fall wird zuerst die TestSequenz `Kl15Ein` aufgerufen. Ihr folgt die TestSequenz `BlinkenAnWartenAus` und abschließend die TestSequenz `AuswertungBlinken`.

TestSequenz Ein Use–Case dem der Stereotyp «testSequence» zugewiesen ist, wird innerhalb von EXAM als TestSequenz bezeichnet. Auch einer TestSequenz ist immer ein Objekt–Sequenz–Diagramm zugeordnet.

Objekt–Sequenz–Diagramm Im Objekt–Sequenz–Diagramm ist die zeitliche Abfolge von Operationen (s. Abb B.7) und Use–Case–Includes (s. Abb B.10) enthalten. Mischformen sind dabei erlaubt.

Innerhalb dieser Arbeit wird der Begriff der TestSequenz in dieser Schreibweise immer in dem soeben vorgestellten Kontext verwendet. Dies soll Verwechslungen mit den in Kap. 5.2 eingeführten Testsequenzen vermeiden.

Parametersatz Ein Use–Case, dem der Stereotyp «parameterSet» zugewiesen ist, wird innerhalb von EXAM als Parametersatz bezeichnet. Use–Cases, die als Parametersätze verwendet werden, ist immer ein Objekt–Kollaborations–Diagramm untergeordnet.

Objekt–Kollaborations–Diagramm Das Objekt–Kollaborations–Diagramm enthält Instanzen existierender Parameterklassen. Abb. B.12 zeigt z.B. eine Instanz der Parameterklasse `parameterMaster` im Objekt–Kollaborations–Diagramm `GlobalParameters`. Darin sind alle mit Werten belegten Parameter (Attribute der Parameterklasse) inkl. der Werte selbst enthalten. Der Parameter `pCaptureLength`, der der Aufzeichnungslänge der Echtzeitmessung entspricht, wird darin mit dem Wert 20 belegt. Dies führt bei Ausführung des TestCase zu einer Aufzeichnungslänge von 20 Sekunden. Die übrigen Parameter dienen ebenfalls der Steuerung der erwähnten Echtzeitmessung.

Use–Case–Diagramm Auch das Use–Case–Diagramm wird in EXAM in unterschiedlichen Kontexten eingesetzt. Grundsätzlich werden darin Relationen zwischen verschiedenen Use–Cases beschrieben, so z.B. die Zuordnung von Parametersätzen

zu Testsequenzen mittels einer Generalisierungsbeziehung[8], die wiederum mit dem Stereotyp «parameterize» versehen ist. In Abb. B.13 sind z.B. der TestSequenz `BlinkenAnWartenAus` drei Parametersätze zugewiesen. Mehrfachvererbung zwischen Parametersätzen ist dabei möglich. Der Parametersatz `GlobalParameters` ist aus Abb. B.12 bereits bekannt. Die übrigen Parametersätze erben alle dort definierten Parameter und überschreiben diese mit ihren Werten oder erweitern den Parametersatz entsprechend.

TestCompositionDiagram Jedes Use–Case–Diagramm, dem der Stereotyp «testCaseComposition» zugeordnet ist, wird als TestCompositionDiagram bezeichnet. Es stellt das Ergebnis der Testkomposition (s. Abb. B.4) dar und enthält für jeden TestCase die vollständige Auflistung aller dem TestCase untergeordneten TestSequenzen sowie den für jede TestSequenz ausgewählten Parametersatz.

Testsuite Eine Klasse, welcher der Stereotyp «testSuite» zugewiesen ist, wird innerhalb von EXAM als Testsuite bezeichnet. Eine Testsuite stellt eine abgeschlossene Einheit dar, die die Ablaufreihenfolge der darin enthaltenen TestCases festgelegt. Für jede Testsuite wird ein eigener Testbericht erstellt.

Testkampagne Objekt–Sequenz–Diagramme, denen der Stereotyp «testCampaign» zugewiesen ist, werden innerhalb von EXAM als Testkampagnen bezeichnet. Testkampagnen stellen die einzige ablauffähige Einheit in EXAM dar. Ihr Inhalt legt die Aufrufreihenfolge von Testsuiten fest.

GlobalParameters

Instance:stdParameters::parameterMaster
pCaptureLength=20 pDownsampling=2 pID=RB pSignalList=['mSw_SMLS_BlinkenLinksRechts','mSw_ILMH_BlinkenLinksOnOff','mSw_ILMH_BlinkenRechtsOnOff'] pTimeOut=35 pTriggerDelay=-1 pTriggerLevel=0.9 pTriggerSlope=1 pTriggerVariable='mSw_SMLS_BlinkenLinksRechts'

Abbildung B.12: Beispiel eines Objekt–Kollaborations–Diagramms

[8]Die Generalisierungsbeziehung zwischen Parametersatz und Testsequenz wird nicht im eigentlichen Sinne der Vererbung verwendet. Die Kennzeichnung mittels des erwähnten Stereotyps spiegelt diesen Sachverhalt wider.

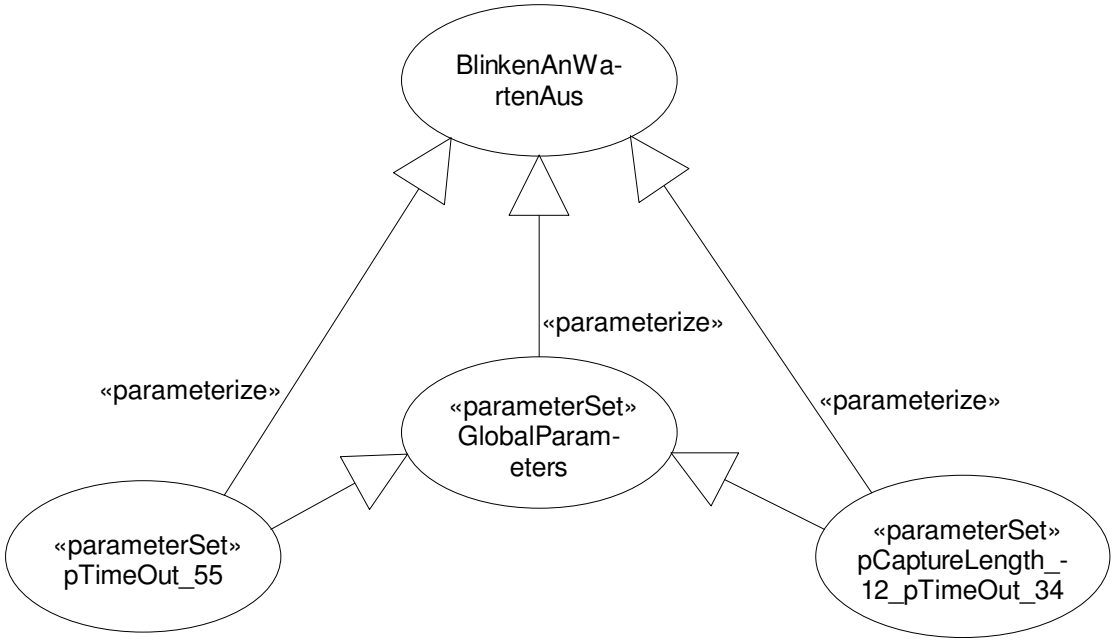


Abbildung B.13: Beispiel eines Use–Case–Diagramms

B.6 Schichtenkonzept

Ein entscheidender Faktor für den Erfolg von EXAM ist die wartungsfreundliche und leicht erweiterbare Strukturierung der verschiedenen UML–Modelle (vgl. Anh. B.5) durch ein Schichtenkonzept (*Schläfer*, 2004). Nach Erweiterung durch *Kiffe* (2006) nimmt das Schichtenkonzept die in Abb. B.14 dargestellte Form an, wobei die einzelnen Schichten aufeinander aufbauen.

Der untersten Schicht, dem `Target Layer`, sind alle Komponenten des Testsystems zugeordnet. Dazu zählen z.B. Hardware–in–the–Loop Prüfstände verschiedenster Hersteller, verschiedenste Netzteile sowie zahlreiche Software–Systeme (CANoe, INCA, ...). Darüber befindet sich der `Driver Layer`, der Schnittstellen zur Ansteuerung der im `Target Layer` vorhandenen Komponenten bereitstellt. Die nächste Schicht bildet der `Adaption Layer`, in dem die Schnittstellen des `Driver Layer` in die Softwareumgebung eines konkreten EXAM–Systems adaptiert werden. Auf dieser Adaption aufbauend setzt der `Library Layer` ein, in dem alle Implementierungsklassen von EXAM entweder in Form von Objekt–Sequenz–Diagrammen oder als plattformspezifischer Programmcode enthalten sind. Der `Interface Layer` ermöglicht die bereits in Anh. B.2 angesprochene Trennung zwischen Testbeschreibung und Testimplementierung. Für jede im `Library Layer` enthaltene Implementierungsklasse existiert im `Interface Layer` eine Interfaceklasse. Werden im `Test Case Layer` nur Klassen aus dem `Interface Layer` zur Testbeschreibung verwendet, dann sind die Tests plattformunabhängig beschrieben und lassen sich entsprechend auf verschiedene Hardware–in–the–Loop Prüfstände portieren. Abgeschlossen wird das Schich-

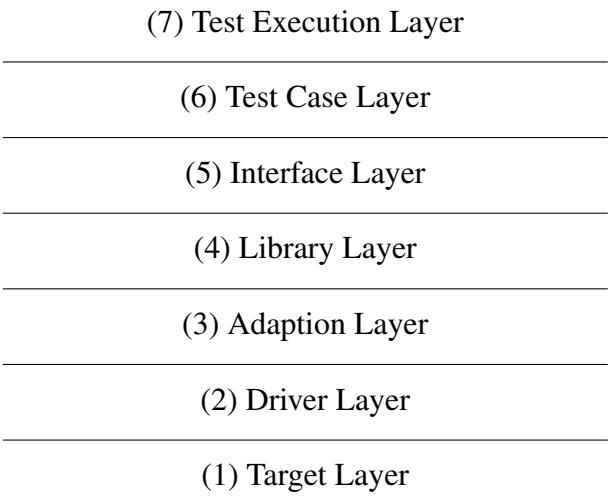


Abbildung B.14: EXAM−Schichtenmodell (*Kiffe*, 2006)

tenmodell durch den `Test Execution Layer`. Darin wird die Struktur des Testablaufs festgelegt.

C Schnittstellenbeschreibung CTM/ES–EXAM (ctmes2exam)

Dieses Kapitel beinhaltet die Vorstellung der Schnittstelle (ctmes2exam), die basierend auf der Classification–Tree Method for Embedded Systems (CTM/ES) zwischen dem Classification–Tree Editor eXtended Logics (CTE XL) und der Testautomatisierung EXAM bzw. dem verwendeten CASE–Tool, dem ARTiSAN Studio, prototypenhaft implementiert wurde. Mit Hilfe dieser Schnittstelle werden im CTE XL definierte Testsequenzen automatisch als TestCases in ein UML–Modell überführt. Nach durchgeführter Testkomposition ist jeder generierte TestCase ausführbar.

Der entwickelte Prototyp diente der Überprüfung des in Kap. 6 vorgestellten Integrationskonzeptes, welches aufgrund der Ergebnisse des Praxistests (vgl. Kap. 7) in der ursprünglich präsentierten Form nicht umgesetzt wurde. Die Realisierung verfügt daher über keine grafische Benutzeroberfläche. Zur Entwicklung wurde die Entwicklungsumgebung *Eclipse* verwendet.

In den ersten beiden Abschnitten (C.1 und C.2) werden zunächst die notwendigen Erweiterungen des CTE XL vorgestellt, die für den Import in das UML–Modell notwendig sind. Der dritte Abschnitt (C.3) geht auf die Funktionsweise der Schnittstelle ein und enthält eine ausführliche Beschreibung der automatisch erstellten Struktur im UML–Modell.

C.1 Definition neuer Eigenschaften von Klassifikationsbäumen

Die für die Automatisierung notwendige Attributierung von Klassifikationsbäumen (vgl. Kap. 5.3.4) beruht auf der Erweiterung der standardmäßig zur Verfügung stehenden Element–Eigenschaften von Klassifikationsbäumen. Neue Eigenschaften können im CTE XL in der Menüleiste unter „Tools → Tags...“ an alle Elemente des Klassifikationsbaumes und der Kombinationstabelle hinzugefügt werden. Dabei kann zwischen verschiedenen „Tagtypen“ gewählt werden. Verwendet werden ausschließlich die beiden Typen „VariableMap“ (s. Abb. C.1) und „TextualTag“ (s. Abb. C.2), wobei der Typ „VariableMap“ nur für die zwei neuen Eigenschaften `GlobalParameters` und `TestSequenceParameters` zum Einsatz kommt. Alle anderen neu eingeführten Eigenschaften sind vom Typ „TextualTag“.

Diese zusätzlich eingeführten Eigenschaften werden im Zusammenhang mit dem jeweiligen CTE XL–Element in den folgenden zwei Abschnitten detailliert vorgestellt. Abschnitt C.1.1 geht dabei auf die Elemente des Klassifikationsbaumes ein, während Abschnitt C.1.2 Elemente der Kombinationstabelle betrachtet.

C.1.1 Elemente des Klassifikationsbaumes

Von der entwickelten Schnittstelle werden lediglich die drei Elemente Wurzel, Klassifikation und Äquivalenzklasse des Klassifikationsbaumes berücksichtigt. Äquivalenzklassen und Klassifikationen sind nur relevant, falls sie sich auf der untersten Ebene befinden. Äquivalenzklassen auf unterster Ebene wurden in Kap. 5.1.1.3 bereits als Blattklassen benannt. Klassifikationen die Blattklassen enthalten, werden deshalb im weiteren Verlauf als Blattklassifikationen bezeichnet.

C.1.1.1 Wurzel

Die Wurzel des Klassifikationsbaumes hat lediglich einen zusätzlichen Reiter mit Namen

- `GlobalParameters`

im Eigenschaftsfenster erhalten.

`GlobalParameters` ermöglicht die Definition von globalen Parametern für alle Testsequenzen. Abb. C.1 zeigt ein Beispiel. Die dort aufgeführten Attribute haben einen `Namen` sowie einen `Wert`.

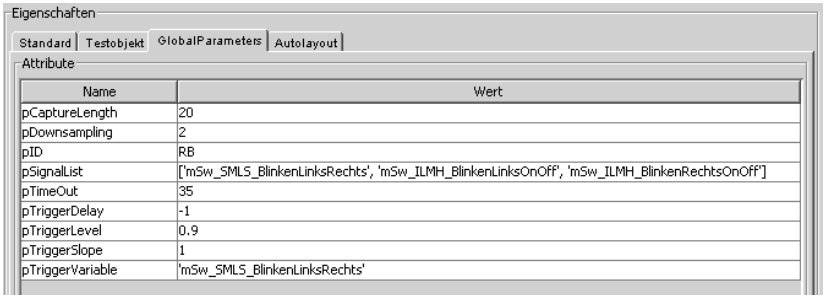


Abbildung C.1: Eigenschaft `GlobalParameters` des Klassifikationsbaumelementes Wurzel vom Typ „VariableMap“

Unter `Name` enthaltene Einträge stellen Parameter in EXAM dar, wie z.B. die Aufzeichnungslänge der Messung (`pCaptureLength`) oder die Liste der Aufzeichnungsvariablen (`sSignalList`). Der Name der Parameter muss hierbei identisch mit dem

UML–Modellnamen des Parameters sein. Im Eintrag `Wert` kann der Inhalt des Parameters hinterlegt werden.

C.1.1.2 Klassifikation

Klassifikationen enthalten im Gegensatz zur Wurzel mehrere Erweiterungen:

- `PreConditionOrder`
- `InterfaceID`
- `Variable`
- `GetOperationID`

`PreConditionOrder` resultiert aus der Notwendigkeit, eine Reihenfolge für Operationsaufrufe für die Herstellung einer definierten Vorbedingung festzulegen. Der erste Testschritt einer im CTE XL definierten Testsequenz beschreibt eine Vorbedingung, aus der das Testobjekt heraus stimuliert wird. Ob jedoch in dem in Abb. C.3 aufgeführten Beispiel zuerst die Aktion `Kl15 ein` oder `U_bat normal` hergestellt werden soll, bleibt ungewiss. Hier schafft der neue Reiter `PreConditionOrder` Abhilfe, indem jeder Klassifikation eine eindeutige Zahl, beginnend bei 1, zugeordnet wird. Dadurch wird die Aufrufreihenfolge der Vorbedingung für alle TestCases eindeutig festgelegt. Klassifikationen mit `PreConditionOrder` $= -1$ werden bei der Herstellung der Vorbedingung nicht betrachtet.

`InterfaceID` Jede Blattklassifikation muss ein Interface aus dem UML–Modell zugeordnet werden. Diese Verknüpfung zwischen UML–Element und CTE XL–Element ist derzeit über die UML–Modell–ID des UML–Elementes realisiert. Die ID muss mittels des Tools TELLME[1] im UML–Modell ermittelt werden und manuell in den Reiter `InterfaceID` der zugehörigen Klassifikation eingefügt werden. Abb. C.2 zeigt einen Beispieleintrag für die Eigenschaft `InterfaceID` vom Typ „TextualTag“.

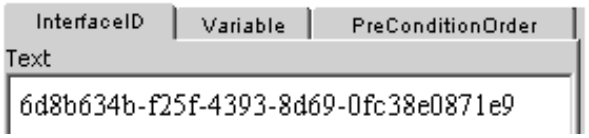


Abbildung C.2: Eigenschaft `InterfaceID` des Klassifikationsbaumelementes Klassifikation vom Typ „TextualTag“ inkl. enthaltener UML–Modell–ID

[1] Das Tool TELLME ermöglicht eine detaillierte Einsicht in UML–Modelle des ARTiSAN Studio.

`Variable` lautet eine weitere neue Eigenschaft von Klassifikationen. Hier wird der Name der Variablen eingetragen, die später im Objekt–Sequenz–Diagramm erscheint. Die Benennung folgt dabei keinen besonderen Regelungen, jedoch muss sie zwischen den verschiedenen Klassifikationen eindeutig sein. Gleiche Einträge führen zu Problemen (mehrere gleichnamige Variablen im jeweiligen Objekt–Sequenz–Diagramm). Eine Empfehlung lautet daher, die in EXAM für Objekt–Sequenz–Diagramme üblichen Bezeichnungen zu verwenden: z.B. für die Klassifikation `Warten` die Variable `iWaitKey`, für die Klassifikation `U_bat` die Variable `sU_batKey` usw.

`GetOperationID` Unter diesem Reiter wird die UML–Modell–ID der getOperation der zur Klassifikation gehörenden Variablen angegeben.

C.1.1.3 Äquivalenzklasse

Äquivalenzklassen wurden folgende drei Eigenschaften hinzugefügt:

- `OperationID`
- `Representative`
- `StepDescription`

`OperationID` Jeder Blattklasse wird eine Operation eines Interfaces aus dem UML–Modell zugeordnet. Das Interface ist bereits durch seine UML–Modell–ID in der über der Äquivalenzklasse liegenden Klassifikation gekennzeichnet. Die Zuordnung einer UML–Operation zur Äquivalenzklasse findet auf die gleiche Weise statt: durch Einfügen der UML–Modell–ID in den Reiter `OperationID`.

`Representative` Blattklassen benötigen für die spätere Ausführung des TestCases einen konkreten Stellvertreter für die ganze Äquivalenzklasse. Dieser Vertreter wird im Reiter `Representative` angegeben. Im Beispiel der Klassifikation `U_bat` stehen hinter den abstrakten Äquivalenzklassen `unter`, `normal` und `über` z.B. die Vertreter `8.0`, `13.2` und `18.0`. Diese Werte erscheinen als Übergabeparameter der aufgerufenen Operation im automatisch erzeugten Objekt–Sequenz–Diagramm.

`StepDescription` bietet die Möglichkeit den Kommentar jeder Zeile im Objekt–Sequenz–Diagramm automatisch zu erzeugen. Dabei werden die einzelnen Testschritte miteinander verglichen und jeweils eine Äquivalenzklasse für jeden Testschritt als relevant eingestuft. Aus diesem Grund darf sich von Testschritt zu Testschritt immer nur eine Äquivalenzklasse ändern. Dadurch steht die relevante Äquivalenzklasse eindeutig

fest. Eine manuelle Benennung der später erwähnten Testschritte (Anh. C.1.2.2) ist daher nicht notwendig. Es wird der Eintrag aus `StepDescription` für dem Kommentar im Objekt−Sequenz−Diagramm übernommen.

C.1.2 Elemente der Kombinationstabelle

Lediglich Testsequenzen und Testschritte, als Elemente der Kombinationstabelle, werden von dem Prototypen für den Import in das UML−Modell ausgewertet. Testfälle und Packages werden nicht berücksichtigt.

C.1.2.1 Testsequenz

Die Testsequenz besitzt, ebenso wie die Wurzel, nur einen neuen Reiter

- `TestSequenceParameters`

im Eigenschaftsfenster.

`TestSequenceParameters` Der Inhalt dieses zusätzlichen Reiters ist sehr ähnlich zum einzigen neuen Reiter der Wurzel. Wir erinnern uns, dass mittels der Eigenschaft `Global-Parameters` die Definition globaler Parameter für den gesamten Baum inkl. Testsequenzen möglich war. Mittels `TestSequenceParameters` können die globalen Parameter durch lokale Parameter überschrieben werden. Auf diese Weise kann z.B. die Aufzeichnungslänge eines TestCase verändert oder aber auch eine andere Signalliste angegeben werden (s. Abb. C.17).

Weitere Optionen wie z.B. die Erweiterung der Signalliste sind denkbar, wurden jedoch bisher nicht umgesetzt.

C.1.2.2 Testschritt

Dem Testschritt sind drei neue Eigenschaften zugewiesen:

- `StepType`
- `SubSequenceName`
- `PossibleVariations`

`StepType` In EXAM enthält ein TestCase folgende Struktur: preCondition, action und postCondition (vgl. Kap B.5.2). Diese Struktur muss im CTE XL manuell vorgegeben werden. Gelegenheit dazu bietet der Reiter `StepType` jedes Testschrittes. Jeder Testschritt muss entweder den Eintrag preCondition oder action besitzen[2], wobei der erste Testschritt immer als preCondition markiert sein muss. Alle als preCondition markierten Testschritte zählen zur Vorbedingung. Als action markierte Testschritte werden der action–TestSequenz zugeordnet und bei der Aufzeichnung der Messung mittels der Operation `startMeasurement` berücksichtigt.

Die nächsten beiden neuen Eigenschaften von Testschritten sind nur von Interesse, falls Subsequenzen in der Kombinationstabelle definiert werden.

`SubSequenceName` Mittels der neuen Eigenschaft `SubSequenceName` können mehrere zusammenhängende Testschritte zu Subsequenzen gruppiert werden. Im Reiter `SubSequenceName` muss der Name der Subsequenz für jeden zur Subsequenz zugehörigen Testschritt angegeben werden. Dabei gilt die Regel, dass höchstens ein Testschritt (der erste oder der letzte Testschritt) einen anderen Eintrag in `StepType` haben darf als die restlichen Testschritte der definierten Subsequenz.
Abb. C.3 zeigt sechs Testsequenzen. In der ersten Testsequenz `a_Links_Blinken` sind die Testschritte 2–5 durch den Eintrag „BlinkenAnWartenAus“ unter `SubSequenceName` als Subsequenz definiert (s. auch Tab. C.4 in Anh. C.1.3). Ist ein Testschritt keiner Subsequenz zugeordnet, so muss derzeit der Eintrag „none“ unter `SubSequenceName` eingefügt werden.

`PossibleVariations` Zusätzlich können im Reiter `PossibleVariations` mögliche Variationen der definierten Subsequenz angegeben werden. Falls in anderen Testsequenzen Variationen der einmal definierten Subsequenz vorkommen, so werden diese bei der automatischen Generierung der Testinhalte in das UML–Modell durch die entsprechende Subsequenz sowie passende Parametersätze ersetzt. Falls also Abläufe in der Kombinationstabelle wiederkehren und sich nur durch die Auswahl unterschiedlicher Äquivalenzklassen unterscheiden, sollte von dieser Möglichkeit Gebrauch gemacht werden. Dadurch wird das UML–Modell nicht unnötig mit Elementen belastet und die angestrebte hohe Wiederverwendungsrate wird erhöht.

Abb. C.3 zeigt im Eigenschaftsfenster im Reiter `possibleVariations` für den markierten[3] Testschritt 3 den Eintrag:

$$\underbrace{p8 : p36 \quad p42;}_{Pflicheintrag} \quad \underbrace{p10 : sameID}_{Optional}$$

[2] Der Eintrag postCondition wird derzeit nicht ausgewertet.
[3] Um die Einträge der übrigen Testschritte einzusehen, wird wiederum auf Tab. C.4 in Anh. C.1.3 verwiesen.

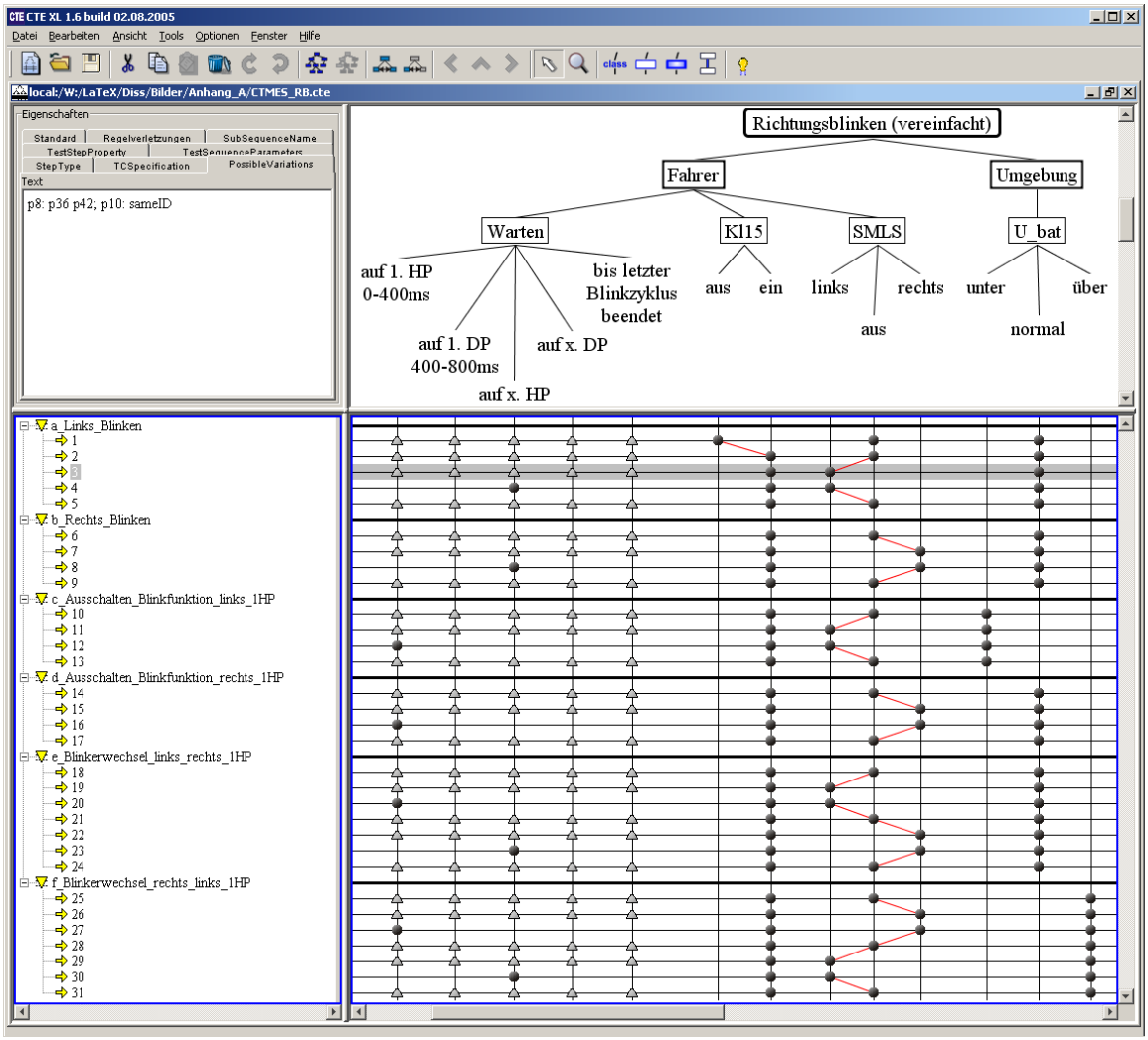


Abbildung C.3: Screenshot des CTE XL

Bisher ist die Darstellung der Subsequenzvariationen noch sehr kryptisch. Es werden die IDs der CTE XL–Elemente (vgl. Kap. 5.4) zur Festlegung von Variationen verwendet.

Der oben dargestellte Eintrag wird durch das Semikolon in zwei Untereinträge aufgespalten. Der erste Untereintrag „p8: p36 p42“ ist ein Pflichteintrag und beschreibt eine mögliche Variation in der Klassifikation `SMLS` mit der ID 8. Für die Variation zur Auswahl stehende Äquivalenzklassen sind die Äquivalenzklassen mit der ID 36 bzw. 42, also die Äquivalenzklassen `links` bzw. `rechts`. Falls keine Variation in einem Testschritt gewünscht ist, so kann dies mit dem Eintrag „none“ spezifiziert werden.
Der zweite Untereintrag „p10: sameID“ ist optional und legt fest, dass für die Klassifikation mit der ID[4] 10, also `U_bat`, die gleiche Äquivalenzklasse ausgewählt sein muss wie im Test-

[4]Jeder CTE XL–ID muss ein „p“ vorangestellt werden. Diese Forderung ergibt sich aus dem XML–Datenformat in dem der CTE XL die erstellten .cte–Dateien ablegt.

schritt zuvor. D.h. die ausgewählte Äquivalenzklasse von `U_bat` darf sich von Testschritt zu Testschritt nicht ändern, ganz egal welche Äquivalenzklasse einmal ausgewählt ist. Damit können Subsequenzvariationen bei verschiedenen sich nicht ändernden Randbedingungen realisiert werden.

Einträge der ersten Kategorie dürfen nur einmal pro Testschritt vorkommen und müssen immer an erster Stelle stehen. Einträge der zweiten Kategorie können dagegen für jede im Klassifikationsbaum vorhandene Klassifikation einmal pro Testschritt definiert werden, wobei bereits eine in der ersten Kategorie vorhandene Klassifikation in der zweiten Kategorie nicht zulässig ist.

Hintergrund dieser Möglichkeit zur Definition von Subsequenzen ist die bereits erwähnte „Entlastung“ des UML−Modells und die damit verbundene und angestrebte hohe Wiederverwendungsrate von spezifizierten Testinhalten im UML−Modell (vgl. Anh. B.1).

C.1.3 Relevante Eigenschaften des Klassifikationsbaumes

Aufbauend auf dem in Abb. C.3 dargestellten Klassifikationsbaum inkl. Kombinationstabelle wird im Anh. C.3 der automatische Import der Testinhalte in das UML−Modell beschrieben. Zum besseren Verständnis der Funktionsweise der Schnittstelle gibt dieser Abschnitt einen Überblick über alle für die beschriebene Schnittstelle relevanten Eigenschaften des in Abb. C.3 dargestellten Klassifikationsbaumes. Die Eigenschaften sind nach ihrem zugehörigem Element (Klassifikation, Äquivalenzklasse, Testsequenz, Testschritt) im CTE XL in Form von vier Tabellen strukturiert.

Name	ID	PreConditionOrder	InterfaceID	Variable
`Warten`	4	−1	7ebc0ec39294	iWaitKey
`Kl15`	6	2	0fc38e0871e9	iKl15Key
`SMLS`	8	3	5f0045e9aacf	sSMLSKey
`U_bat`	10	1	a96606eb80bf	sU_batKey

Tabelle C.1: Eigenschaften von Klassifikationen

Name	ID	OperationID	Representative	StepDescription
auf 1.HP 0-400ms	12	7d666aa6b4bb	200	Warten
auf 1.DP 400-800ms	16	7d666aa6b4bb	500	Warten
auf x.HP	14	7d666aa6b4bb	5000	Warten
auf x.DP	18	7d666aa6b4bb	5400	Warten
bis letzter Blinkzyklus beendet	80242	7d666aa6b4bb	1500	Warten
aus	22	7c6112a0c010	aus	Zündung ausschalten
ein	20	637ad80401c2	ein	Zündung einschalten
links	36	f657d67489f8	links	SMLS links einschalten
aus	39	9d85938fd9df	aus	SMLS Nullstellung
rechts	42	f49cca053432	rechts	SMLS rechts einschalten
unter	45	6861c87211e2	8.0	Unterspannung herstellen
normal	48	6861c87211e2	13.2	Normalspannung herstellen
über	51	6861c87211e2	18.0	Überspannung herstellen

Tabelle C.2: Eigenschaften von Äquivalenzklassen

Name	ID	TestSequence–Parameters
a_Links_Blinken	54	pCaptureLength=12, pTimeOut=34
b_Rechts_Blinken	74	pTimeOut=55
c_Ausschalten_Blinkfunktion_links_1HP	2825	–
d_Ausschalten_Blinkfunktion_rechts_1HP	33963	–
e_Blinkerwechsel_links_rechts_1HP	33992	pCaptureLength=12
f_Blinkerwechsel_rechts_links_1HP	33997	pCaptureLength=11

Tabelle C.3: Eigenschaften von Testsequenzen

Name	ID	StepType	SubSequenceName	PossibleVariations
1	48593	preCondition	none	–
2	58	preCondition	BlinkenAnWartenAus	p10: p45 p48 p51
3	66	action	BlinkenAnWartenAus	p8: p36 p42; p10: sameID
4	62	action	BlinkenAnWartenAus	p4: p12 p14 p16 p18; p8: sameID; p10: sameID
5	70	action	BlinkenAnWartenAus	none; p10: sameID
6	78	preCondition	none	–
7	82	action	none	–
8	86	action	none	–
9	90	action	none	–
10	33941	preCondition	none	–
11	33945	action	none	–
12	33951	action	none	–
13	33957	action	none	–
14	33861	preCondition	none	–
15	33866	action	none	–
16	33870	action	none	–
17	33875	action	none	–
18	33891	preCondition	none	–
19	33896	action	none	–
20	33932	action	none	–
21	33900	action	none	–
22	33905	action	none	–
23	34099	action	none	–
24	34119	action	none	–
25	33909	preCondition	none	–
26	33914	action	none	–
27	33927	action	none	–
28	33918	action	none	–
29	33923	action	none	–
30	34106	action	none	–
31	34113	action	none	–

Tabelle C.4: Eigenschaften von Testschritten

C.2 Weitere Anpassungen

Neben den bisher erwähnten Erweiterungen sind weitere Anpassungen notwendig, um die automatische Generierung von Testinhalten im UML−Modell zu ermöglichen. Diese Anpassungen beziehen sich jedoch nicht auf den CTE XL und darin enthaltene Klassifikationsbaum−Elemente. Einige Standardinterfaces sowie Standardoperationen müssen mit der entwickelten Schnittstelle verknüpft werden. Dies sind z.B. Operationen wie `popList`, `calculateAddition`, `startMeasurement` sowie alle in Abb. B.6 enthaltenen Operationen und Klassen. Diese Verknüpfungen sind derzeit direkt im Java−Quellcode der Schnittstelle über die UML−Modell−ID des jeweiligen Elementes realisiert. Auch hier besteht noch Optimierungspotenzial. Für den entwickelten Prototyp ist die bestehende Lösung jedoch vollkommen ausreichend.

C.3 Generierte Struktur im UML−Modell

Durch die in den vorherigen Abschnitten eingeführten neuen Eigenschaften von Klassifikationsbäumen sowie die weiteren Anpassungen ist eine Anbindung des CTE XL an EXAM möglich. Nachdem der Klassifikationsbaum sowie die Testsequenzen erstellt wurden, müssen die erforderlichen Erweiterungen des Klassifikationsbaumes (s. Anh. C.1) mit Inhalten gefüllt werden. Danach kann die entwickelte Schnittstelle (ctmes2exam) ausgehend von der .cte−Datei, welche im XML−Format vorliegt, den Import der spezifizierten Testinhalte in ein UML−Modell vornehmen.

Die Funktionsweise der Schnittstelle lässt sich in drei Bereiche unterteilen. Zuerst wird die CTE−Datei von einem Parser durchlaufen und alle relevanten Inhalte in einer internen Datenstruktur (vgl. Kap. 8.2.2) abgelegt. Danach wird die Datenstruktur analysiert und einige Umstrukturierungen automatisch vorgenommen. Anschließend beginnt der Import der spezifizierten Testinhalte in das aktuell geöffnete UML−Modell.

Beim Import wird zunächst eine statische Struktur an dem ausgewählten Ort in der Packagestruktur des UML−Modells angelegt. Der Ort wird derzeit direkt in der Schnittstelle definiert und mit dem Namen `rootPackage` versehen. Abb. C.4 zeigt den Aufbau der statischen Packagestruktur, welche sich an der standardmäßig verwendeten Struktur von Packages mit Testinhalten (vgl. Anh. B.5.2) orientiert.

Im Package `rootPackage`[5] befindet sich das Package `a_subSequences` und das Package `b_topSequences`. Package `a_subSequences` unterteilt sich weiterhin in die Packages `a_preConditions`, `b_actions` und `d_defined`.

[5] Anstatt des Namens `rootPackage` könnte z.B. der Name der durch die Wurzel des Klassifikationsbaumes ausgewählten Testobjekts verwendet werden.

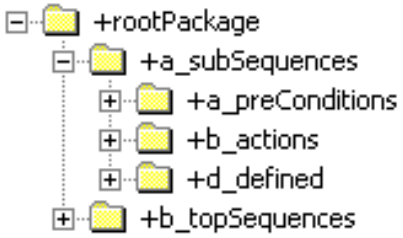


Abbildung C.4: Statische Packagestruktur

Das Package `a_subSequences` beinhaltet alle vorkommenden Subsequenzen. Im Package `a_preConditions` befinden sich TestSequenzen, die Vorbedingungen entsprechen. Package `b_actions` enthält alle action−TestSequenzen. Vom Testspezifikateur manuell definierte Subsequenzen befinden sich inkl. aller Parametersätze für eventuelle Variationen der Subsequenzen im Package `d_defined`.

Im Package `b_topSequences` befinden sich dagegen alle im CTE XL definierten Testsequenzen, wobei im UML−Modell jede Testsequenz durch einen TestCase repräsentiert wird.

Die in Abb. C.4 aufgezeigte statische Packagestruktur wird während des Imports nach und nach mit Inhalten gefüllt. Die folgenden Abschnitte beschreiben das Entstehen sowie den Inhalt der einzelnen Packages. Der Inhalt der jeweiligen Packages hängt dabei natürlich wesentlich von den im CTE XL spezifizierten Inhalten ab. Alle in den folgenden Abschnitten aufgeführten Grafiken orientieren sich an einem durchgängigen Beispiel. Sie wurden automatisch durch die Schnittstelle im ausgewählten UML−Modell erzeugt. Lediglich das Layout der generierten Elemente wurde zur besseren Visualisierung leicht angepasst. Das verwendete Beispiel (Klassifikationsbaum, Testsequenzen und Testschritte) ist in Abb. C.3 dargestellt. Eine detaillierte Auflistung aller für den Import relevanten Inhalte ist in Anh. C.1.3 enthalten. Das Beispiel stellt eine leichte Abwandlung des in Kap. 6.2 vorgestellten Beispiels der Funktion Richtungsblinken dar. Der Klassifikationsbaum ist identisch, lediglich einige Testsequenzen sind verschieden.

C.3.1 Package `a_subSequences`

Im Package `a_subSequences` befinden sich Packages für die TestSequenzen, die einen definierten Ausgangszustand für den Test herstellen (`a_preConditions`), die die eigentliche Testaktion durchführen (`b_actions`) und die die vom Anwender definierten Subsequenzen beherbergen (`d_defined`).

Aufgrund des nicht berücksichtigten Sollverhaltens durch das Testverfahren muss das Package `c_postCondition`, welches normalerweise TestSequenzen zur Auswertung des Tests bereitstellt, manuell vom Anwender angelegt und mit für jeden TestCase spezifischen Inhalten gefüllt werden. Im Folgenden werden daher nur die Packages `a_preConditions`,

b_actions und d_defined betrachtet.

C.3.1.1 Package a_preConditions

Das Package a_preConditions enthält immer ein Package mit dem Namen firstStepOfPreCondition (s. Abb. C.5). Weitere Packages (z.B. Kl15Ein) können enthalten sein, falls in der .cte–Datei Testsequenzen definiert sind, die mehrere als Vorbedingung (Reiter StepType = „preCondition“) markierte Testschritte enthalten.

Package firstStepOfPreCondition Inhalt dieses Packages ist eine TestSequenz (Use–Case mit S) in der der Ausgangszustand, der durch den ersten Testschritt einer Testsequenz im CTE XL beschrieben wird, hergestellt wird.

Neben einem Use–Case–Diagramm ist für jeden erkannten Ausgangszustand zusätzlich ein Parametersatz (Use–Case mit P) enthalten. Das Use–Case–Diagramm beschreibt die Zuweisung der verschiedenen Parametersätze zur firstStepOfPreCondition–TestSequenz, falls verschiedene Ausgangszustände im ersten Testschritt der spezifizierten Testsequenzen im CTE XL definiert sind. In Abb. C.5 ist beispielhaft der Inhalt des Package firstStepOfPreCondition dargestellt. Darin befinden sich vier verschiedene Parametersätze, die vier verschiedene Ausgangszustände mittels der TestSequenz firstStepOfPreCondition herstellen können. Dies entspricht den vier verschiedenen Anfangstestschritten im betrachteten Beispiel (s. Abb. C.3).

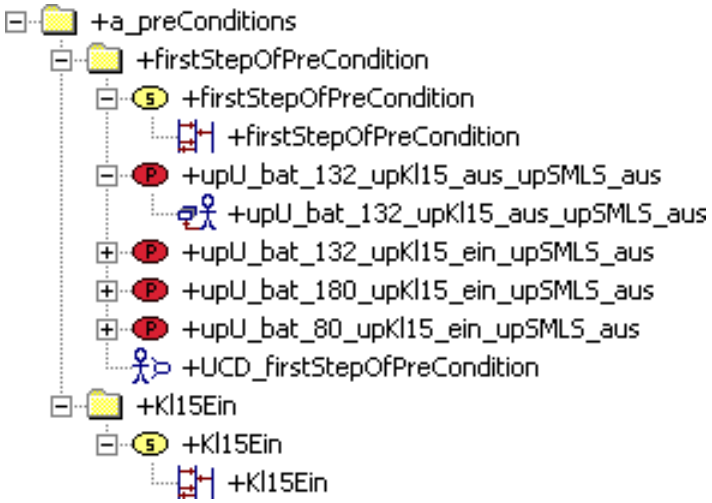


Abbildung C.5: Packagestruktur von a_preConditions

Wie bereits angedeutet, kommt dem ersten Testschritt einer im CTE XL definierten Testsequenz besondere Bedeutung zu. Der erste Testschritt einer Testsequenz beschreibt einen Ausgangszustand, aus dem heraus das Testobjekt stimuliert wird. Hierbei kommt der Inhalt

der Klassifikationseigenschaft `PreConditionOrder` jeder Blattklassifikation zum Tragen. Die Aktion mit dem niedrigsten Eintrag in `PreConditionOrder` wird als erstes ausgeführt, gefolgt von dem nächst höheren Eintrag. Klassifikationen mit `PreConditionOrder`$= -1$ werden bei der `firstStepOfPreCondition`–TestSequenz nicht berücksichtigt. Tab. C.1 zeigt die Einträge der Eigenschaften `PreConditionOrder` und `Variable` der Klassifikationen im Klassifikationsbaum.

Die Klassifikation `Warten` wird in der `firstStepOfPreCondition`–TestSequenz aufgrund der Eigenschaft `PreConditionOrder` $= -1$ nicht berücksichtigt. Die Herstellung des spezifizierten Ausgangszustandes erfolgt gemäß der definierten Reihenfolge. Als erstes wird die Batteriespannung gesetzt. Dem folgt die Herstellung des gewünschten Klemme 15 Status. Als letzte Aktion in der `firstStepOfPreCondition`–TestSequenz wird die Stellung des SMLS gesetzt. Diese Reihenfolge findet sich im entsprechenden Objekt–Sequenz–Diagramm in Abb. C.6 wieder.

Für jede im ersten Testschritt einer Testsequenz im CTE XL ausgewählte Klasse wird anhand der Klasseneigenschaft `Operation` entschieden, ob in die `firstStepOfPreCondition`–TestSequenz ein einziger Operationsaufruf oder eine SWITCH–Anweisung eingefügt wird. Sind alle direkten Nachbarklassen der für diesen Testschritt relevanten Äquivalenzklasse der gleichen Operation zugeordnet, so wird ein einziger Operationsaufruf inkl. dem Variablennamen, der in der Klassifikationseigenschaft `Variable` (s. Tab. C.1) angegeben ist, in das Objekt–Sequenz–Diagramm eingefügt. Der Aufruf `set_mValVoltage(sU_batKey)` in der fünften Zeile in Abb. C.6 ist hierfür ein Beispiel.

Sind die direkten Nachbarklassen jedoch unterschiedlichen Operationen zugeordnet, kommt die SWITCH–Anweisung zum Einsatz. Als Übergabeparameter bekommt die SWITCH–Anweisung den Eintrag aus der jeweiligen Klassifikationseigenschaft `Variable`. Die `firstStepOfPreCondition`–TestSequenz enthält für jede in der jeweiligen Klassifikation enthaltene Klasse einen einzelnen Aufruf. Dieser wird durch eine CASE–Anweisung mit dem Eintrag aus der Klasseneigenschaft `Representative` versehen. Ein Beispiel für diese Realisierung ist ebenfalls in Abb. C.6 enthalten (Zeilen 7–13). Abhängig vom Inhalt der Variablen `iKl15Key` wird entweder die Zündung eingeschaltet oder die Zündung und der Motor ausgeschaltet.

Ob alle Nachbarklassen einer ausgewählten Klasse der gleichen Operation zugeordnet sind, wird anhand des Eintrags in der Klasseneigenschaft `OperationID` entschieden. Enthalten alle Klassen einer Klassifikation die gleiche ID, dann sind alle Klassen der gleichen Operation zugeordnet. Andernfalls sind die Klassen unterschiedlichen Operationen zugeordnet, die jeweils durch ihre ID eindeutig gekennzeichnet sind.

Da unterschiedliche Ausgangsbedingungen in den verschiedenen Testsequenzen vorkommen können, ist die `firstStepOfPreCondition`–TestSequenz parametrierbar gestaltet. Die verschiedenen Konstellationen von Klassen werden identifiziert und für jede Konstellation ein

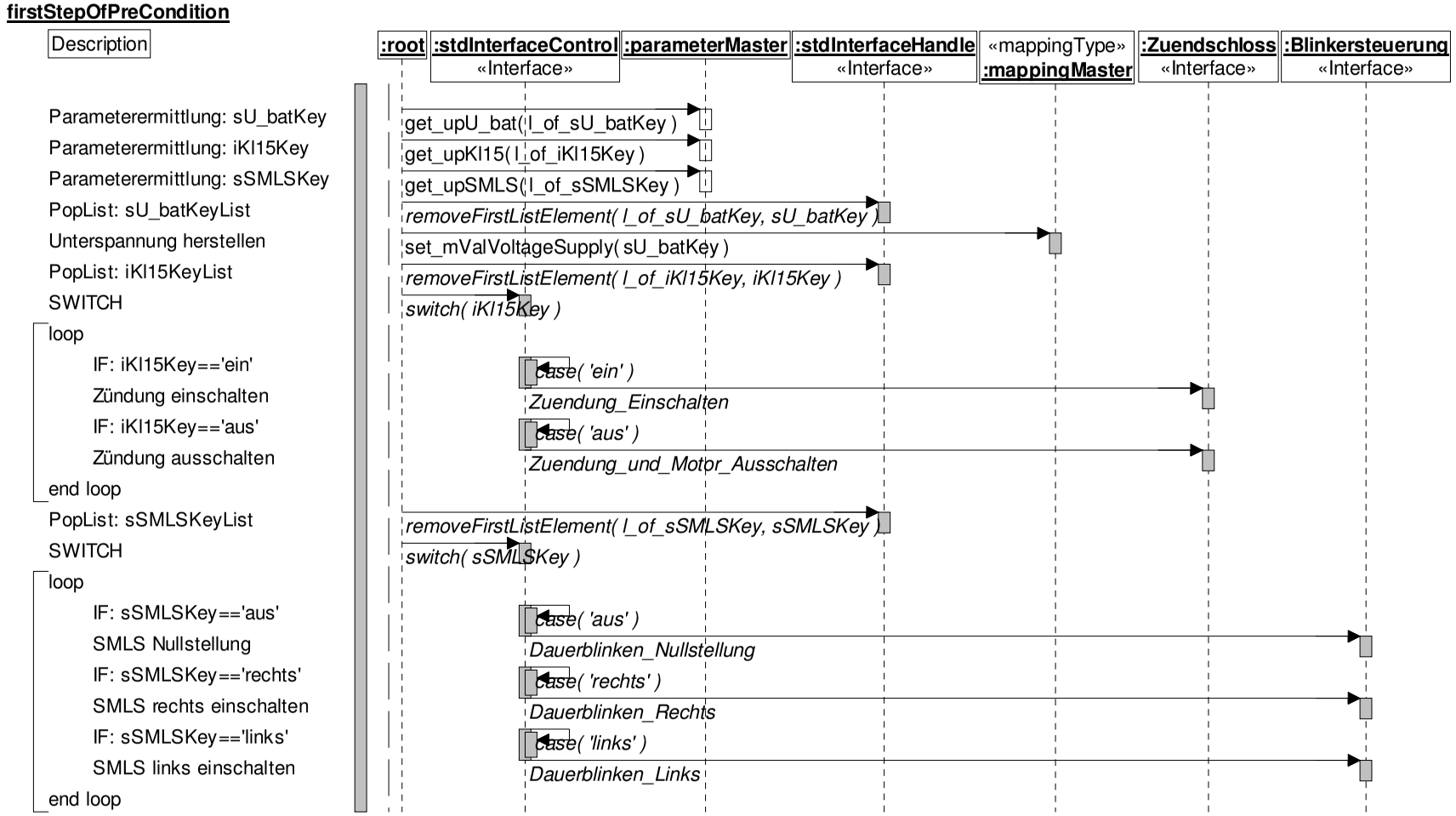


Abbildung C.6: Objekt–Sequenz–Diagramm der `firstStepOfPreCondition`–TestSequenz

eigener Parametersatz angelegt. Konkrete Werte der verwendeten Variablen werden zu Beginn der `firstStepOfPreCondition`–TestSequenz aus dem jeweiligen Parametersatz eingelesen und in einer Liste[6] gespeichert. Vor dem jeweiligen Operationsaufruf wird der Eintrag aus der Liste entfernt und je nach Situation direkt in den Operationsaufruf oder aber in die oben vorgestellte SWITCH–CASE Struktur integriert.

Im bereits erwähnten Use–Case–Diagramm werden die einzelnen Use–Cases miteinander mithilfe der «parameterize» Beziehung verknüpft. Falls mehrere Testsequenzen einen identischen ersten Testschritt enthalten, wird jeweils nur ein einziger Parametersatz für die `firstStepOfPreCondition`–TestSequenz angelegt. Abb. C.7 zeigt die Parameterkonstellation, die sich aus dem in Abb. C.3 aufgeführten Beispiel ergibt. Darin sind vier verschiedene Parametersätze der `firstStepOfPreCondition`–TestSequenz zugeordnet. Jeder Parametersatz stellt eine andere Ausgangsbedingung her.

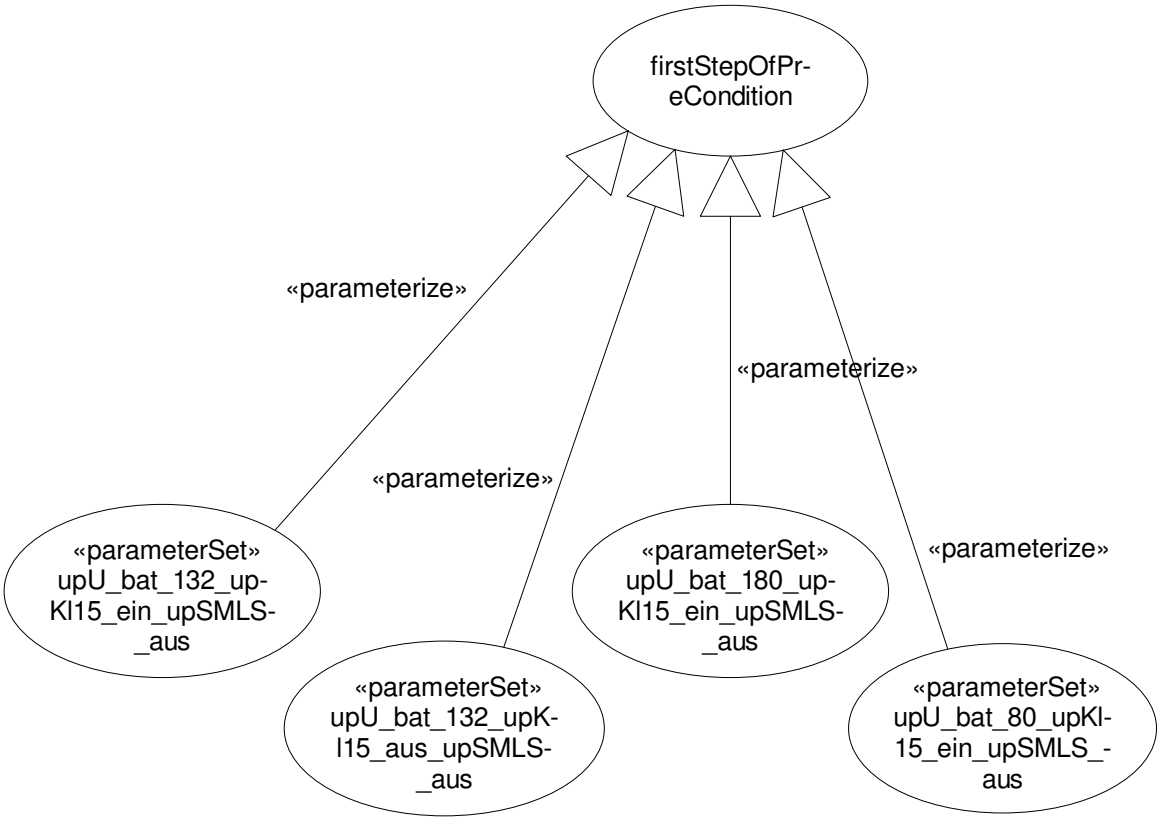


Abbildung C.7: Use–Case–Diagramm `UCD_firstStepOfPreCondition`

Die Benennung der Parametersätze ist bisher nicht optimal. Derzeit werden die Namen der enthaltenen Parameter sowie die entsprechenden Werte der Parameter im Namen abgelegt. Falls sehr viele Parameter enthalten sind und/oder falls sehr lange Parameterwerte vorliegen, treten Probleme bei der Codegenerierung mit der maximalen Zeichenlänge[7] auf.

[6] Der Grund für den Einsatz einer Liste wird in Anh. C.3.1.2 ersichtlich.

[7] Von den existierenden Tools zur Codegenerierung werden derzeit lediglich Pfadlängen von bis zu 255 Zei-

Weitere Packages Falls im CTE XL Testsequenzen spezifiziert wurden, die mehrere Testschritte als Vorbedingung (Testschritteigenschaft `StepType`=„preCondition“) markieren, enthält das Package `a_preConditions` neben dem Package `firstStepOfPreCondition` weitere Packages (z.B. Package `Kl15Ein` in Abb. C.5). Die Namen der Packages setzen sich aus den Namen der in den Testschritten ausgewählten Klassifikationen und Klassen zusammen. In unserem Beispiel wurden in der ersten Testsequenz `a_Links_Blinken` aus Abb. C.3 die ersten beiden Testschritte als Vorbedingung (Eigenschaft `StepType`=preCondition) markiert. Der zweite Testschritt unterscheidet sich vom ersten in der Klassifikation `Kl15` durch die Klasse `ein`. Die Kombination der Namen dieser zwei Elemente führt zu dem Packagenamen `Kl15Ein`. Dabei wird der erste Buchstabe der Äquivalenzklasse in einen Großbuchstaben gewandelt.

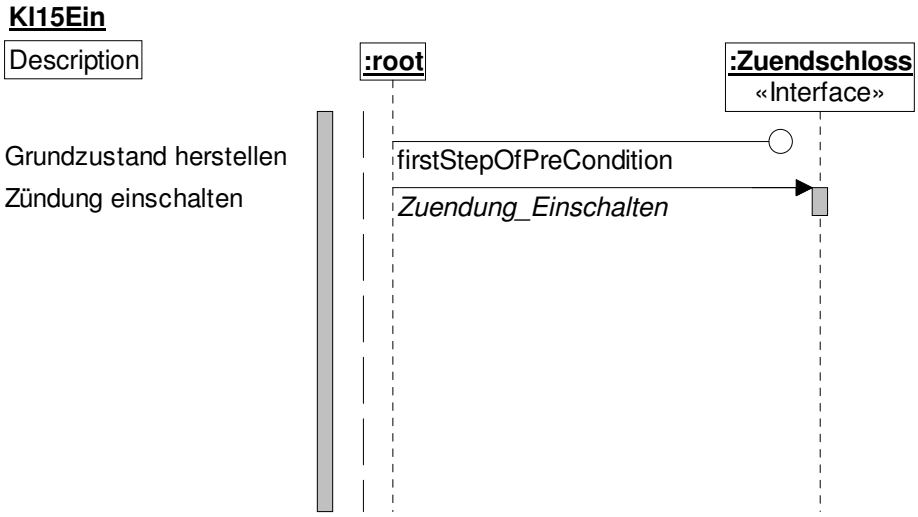


Abbildung C.8: TestSequenz `Kl15Ein`

Im jeweiligen Package befindet sich ein gleichnamiges Objekt–Sequenz–Diagramm (s. Abb. C.8). Darin wird zunächst die `firstStepOfPreCondition`–TestSequenz inkludiert (vgl. Anh. B.5.2). Dies entspricht dem Inhalt des ersten Testschrittes. Es folgen weitere Aufrufe, die ausgehend von dem spezifizierten Ausgangszustand die gewünschte Vorbedingung herstellen. Für jeden als Vorbedingung spezifizierten Testschritt ist ein Aufruf im Objekt–Sequenz–Diagramm enthalten. In unserem Beispiel ist dies der Aufruf der entsprechenden Operation zum Einschalten der Zündung.

Die „weiteren Packages“ werden erst generiert, nachdem das Package `d_defined` (s. Anh. C.3.1.2) vollständig generiert wurde, denn es können definierte Subsequenzen als Vorbedingung definiert sein. Diese Subsequenzen müssen an entsprechender Stelle im soeben vorgestellten Objekt–Sequenz–Diagramm berücksichtigt werden.

chen ohne Probleme verarbeitet. Sind im UML–Modell Pfade enthalten, die länger sind, können diese nicht verarbeitet werden.

C.3.1.2 Package `d_defined`

Falls der Anwender mit Hilfe der Testschritteigenschaft `SubSequenceName` Subsequenzen definiert hat[8], wird das Package `d_defined` mit Inhalten gefüllt. Dabei wird in allen Testsequenzen nach möglichen Variationen der spezifizierten Testsequenz gesucht. Wurde eine weitere Abfolge von Testschritten erkannt, die in das spezifizierte Muster hineinpasst, so werden die entsprechenden Testschritte für die weitere Bearbeitung markiert. Dafür wird die bereits vorhandene Testschritteigenschaft `SubSequenceName` verwendet.

Sind alle Subsequenz–Variationen identifiziert, beginnt der Import. Dabei wird für jede spezifizierte Subsequenz ein eigenes Package mit dem Namen der Subsequenz im Package `d_defined` angelegt. Jedes Subsequenz–Package beinhaltet eine TestSequenz, pro Subsequenz–Variation einen Parametersatz sowie ein Use–Case–Diagramm in dem die Beziehungen zwischen den Parametersätzen und der TestSequenz dargestellt sind. In Abb. C.9 ist eine beispielhafte Packagestruktur visualisiert.

In unserem Beispiel (s. Abb. C.3) wurden von der im CTE XL spezifizierten Subsequenz `BlinkenAnWartenAus` insgesamt vier Variationen in allen weiteren spezifizierten Testsequenzen identifiziert.

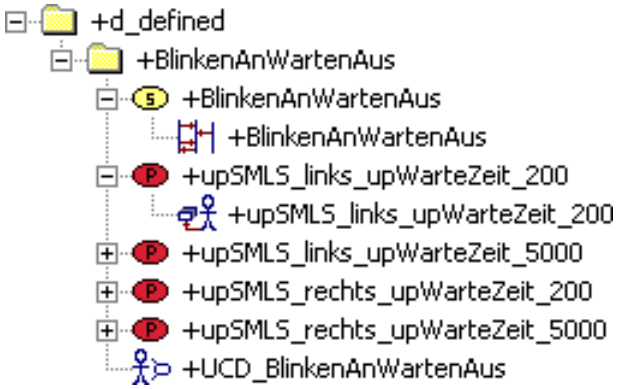


Abbildung C.9: Packagestruktur von `d_defined`

Das für jede im CTE XL spezifizierte Subsequenz generierte Objekt–Sequenz–Diagramm enthält am Anfang zunächst Aufrufe, um die Einträge aus den Parametersätzen einzulesen. Dem folgen Aufrufe der einzelnen Operationen, wobei auch hier von der in Anh. C.3.1.1 vorgestellten Vorgehensweise mit Einzelaufrufen bzw. SWITCH–CASE Konstrukten Gebrauch gemacht wird. Abb. C.10 zeigt ein Beispiel für die Umsetzung der definierten Subsequenz `BlinkenAnWartenAus` im UML–Modell.

An dieser Stelle erklärt sich auch der Einsatz einer Liste, in welche die Parameterwerte eingefügt werden. Vor jedem Aufruf einer Operation muss die entsprechende Operation mit

[8] Details zur Definition von Subsequenzen finden sich in Anh. C.1.2.2 in der Beschreibung der Testschritteigenschaft `SubSequenceName` sowie `PossibleVariations`.

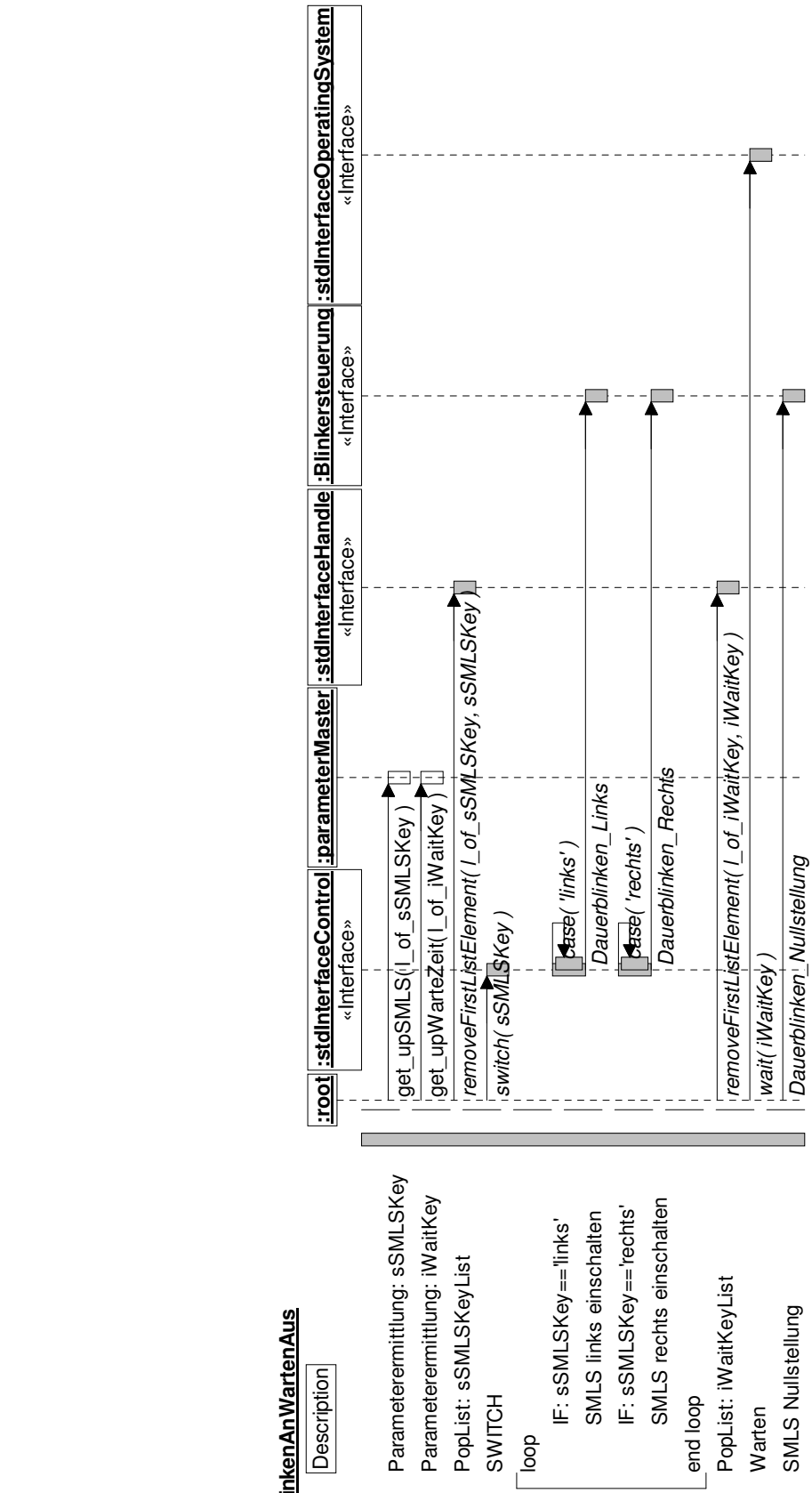


Abbildung C.10: Umsetzung der definierten Subsequenz `BlinkenAnWartenAus` in UML

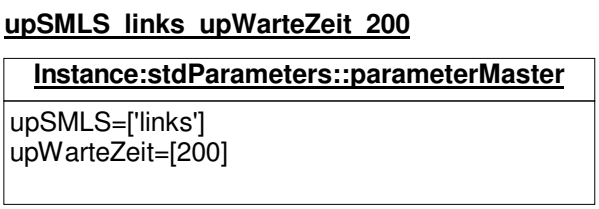


Abbildung C.11: Parameterwerte in Listenform

einem Parameter versehen werden[9]. Es besteht die Möglichkeit, dass eine Operation in einer spezifizierten Subsequenz mehrere Male an verschiedenen Stellen aufgerufen wird. Vor jedem Aufruf wird ein Parameter für den jeweils aktuellen Aufruf benötigt. Daher liegen in den Parametersätzen die Parameterwerte in Listenform vor (s. Abb. C.11). Vor jedem Aufruf einer Operation wird aus der zugehörigen Liste der zuständige Parameterwert entfernt und in den jeweiligen Kontext eingebunden.

C.3.1.3 Package `b_actions`

Im Package `b_actions` befinden sich action−TestSequenzen. Die zur Generierung des Packages relevanten Informationen stammen aus den Testschritten, die in der Eigenschaft `StepType` den Eintrag „action“ enthalten. Der Name jedes hier erzeugten Packages setzt sich wie auch schon im Fall der preCondition−TestSequenz aus den einzelnen durchgeführten Aktionen zusammen. Falls eine Variation einer definierten Subsequenz erkannt wird, so wird der Name der Subsequenz mit in den Packagenamen integriert (s. Abb. C.12).

In jedem Package befindet sich eine TestSequenz inkl. der identifizierten Parametersätze sowie ein Use−Case−Diagramm in dem die Beziehungen der im Package enthaltenen Use−Cases dargestellt sind. Das Objekt−Sequenz−Diagramm der TestSequenz enthält als erstes Aufrufe von Operationen, um die entsprechenden Parameterwerte einzulesen (s. Abb. C.13). Diesen Aufrufen zur Parameterermittlung folgt der Operationsaufruf zum Starten der Messung mittels der bereits im UML−Modell existierenden Operation `startMeasurement`. Diese Operation enthält zahlreiche der zuvor ermittelten Parameter als Übergabewerte. Nach dem Aufruf der `startMeasurement`−Operation folgt die für jeden Testschritt mittels der Eigenschaft `StepType`=„action“ spezifizierte Stimulation des Testobjekts. Beendet wird die Sequenz mit Operationsaufrufen zum Stoppen der Messung (`stopMeasurement`) und zum Verknüpfen der Messdaten mit dem aktuellen Test (`linkMeasureFileToCurrentTest`).

Neben den Packages mit den action−TestSequenzen befindet sich im Package `b_actions` auch noch ein Parametersatz mit dem Namen `GlobalParameters` (s. Abb. C.12 und

[9] Bei einem Einzelaufruf der Operation wird der Parameter direkt mit der Operation verknüpft. Kommt eine SWITCH−CASE Anweisung zum Einsatz wird diese mit dem Parameter versehen.

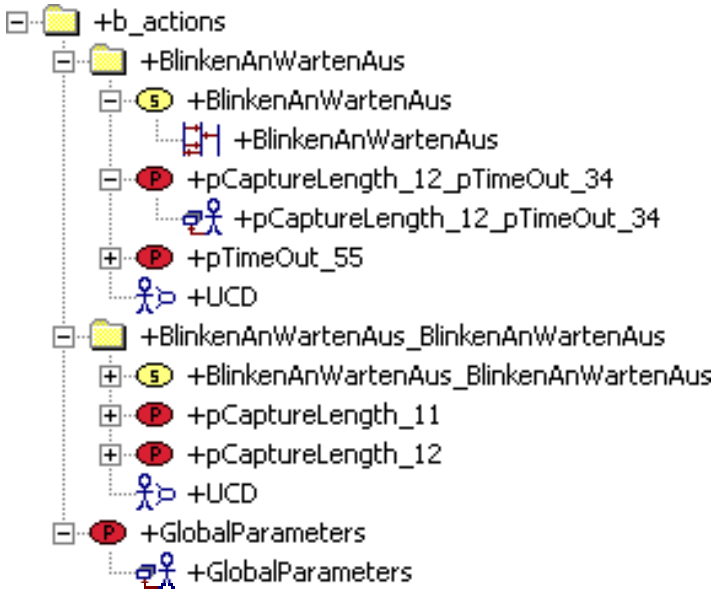


Abbildung C.12: Packagestruktur von `b_actions`

Abb. C.14), der die in der Wurzel des Klassifikationsbaumes angegebenen globalen Parameter enthält (vgl. Eigenschaft `GlobalParameters` in Abb. C.1). Dieser Parametersatz ist in allen Use–Case–Diagrammen der einzelnen action–TestSequenzen enthalten und stellt damit allen action–TestSequenzen die global definierten Parameter zur Verfügung. Natürlich können diese globalen Parameterwerte durch lokale Parameterwerte überschrieben werden. Lokale Parameterwerte für eine Testsequenz bzw. einen TestCase können in der Testsequenzeigenschaft `TestSequenceParameters` festgehalten werden. Die Erweiterung der Use–Case–Diagramme um lokale Parametersätze wird erst bei der Generierung der einzelnen TestSequenzen durchgeführt (s. Anh. C.3.2).

C.3.2 Package `b_topSequences`

Nachdem das Package `b_actions` fast vollständig gefüllt ist, beginnt die Überführung der Testsequenzen in TestCases in das Package `b_topSequences`. Beim Import der in der Kombinationstabelle spezifizierten Testsequenzen wird für jede Testsequenz ein Package mit dem Namen der Testsequenz angelegt. Abb. C.15 zeigt die Packagestruktur von `b_topSequences` für das verwendete Beispiel. Das jeweilige Package beinhaltet einen TestCase. Im untergeordneten Objekt–Sequenz–Diagramm wird dabei auf die bereits im Package `a_subSequences` vorhandenen Elemente zurückgegriffen. Diese werden in entsprechender Reihenfolge in das Objekt–Sequenz–Diagramm eingefügt.

Das zum TestCase `a_Links_Blinken` zugehörige Objekt–Sequenz–Diagramm ist in Abb. C.16 dargestellt. Darin werden zwei Use–Cases inkludiert. Die erste Zeile `preCondition` ruft die TestSequenz `Kl15Ein` aus dem Package `Kl15Ein` im Package

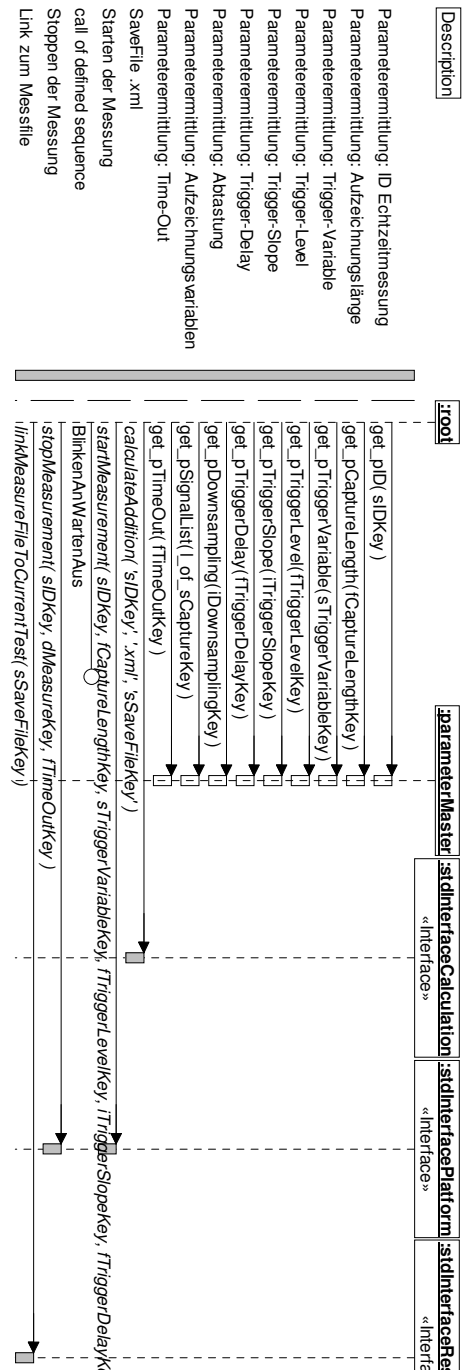


Abbildung C.13: action−TestSequenz

GlobalParameters

Instance:stdParameters::parameterMaster
pCaptureLength=20 pDownsampling=2 pID=RB pSignalList=['mSw_SMLS_BlinkenLinksRechts','mSw_ILMH_BlinkenLinksOnOff','mSw_ILMH_BlinkenRechtsOnOff'] pTimeOut=35 pTriggerDelay=-1 pTriggerLevel=0.9 pTriggerSlope=1 pTriggerVariable='mSw_SMLS_BlinkenLinksRechts'

Abbildung C.14: Globale Parameter

`a_preConditions` auf. Dahinter steckt das Objekt–Sequenz–Diagramm aus Abb. C.8. In der zweiten Zeile `action` wird die eigentliche Testaktion durchgeführt, die in Anh. C.3.1.3 in Abb. C.13 bereits vorgestellt wurde.

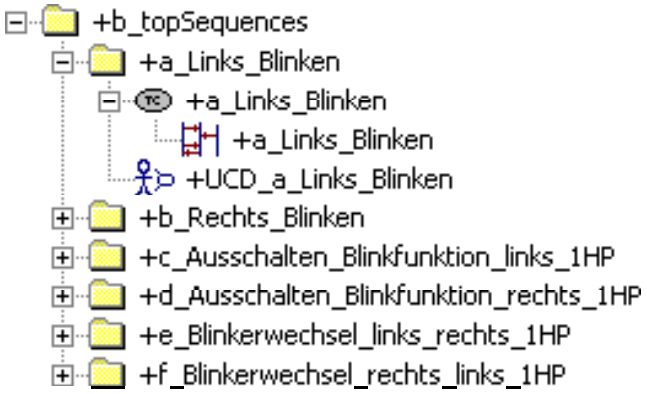


Abbildung C.15: Packagestruktur von `b_topSequences`

Bei dieser Gelegenheit wird das noch unvollständige Package `b_actions` im Package `a_subSequences` vervollständigt. Falls einer Testsequenz lokale Parameterwerte zugeordnet sind, welche die globalen Werte ersetzen sollen, wird an entsprechender Stelle (z.B. im Package der zugehörigen action–TestSequenz (vgl. Anh. C.3.1.3)) im Package `b_actions` ein neuer Parametersatz mit den lokalen Werten erzeugt sowie die notwendigen Beziehungen zwischen den verschiedenen Use–Cases im Use–Case–Diagramm erzeugt. Abb. C.17 zeigt ein Beispiel für ein vollständiges Use–Case–Diagramm des Package `BlinkenAnWartenAus` im Package `b_actions`.

Die action–TestSequenz `BlinkenAnWartenAus` kann sowohl mit den globalen Parametern als auch mit den speziellen Einschränkungen durch die zwei zusätzlichen Parametersätze `pCaptureLength=12_pTimeOut=34` und `pTimeOut=55` parametriert werden.

Nach durchgeführter Testkomposition, in der die Parametersätze den einzelnen TestSe-

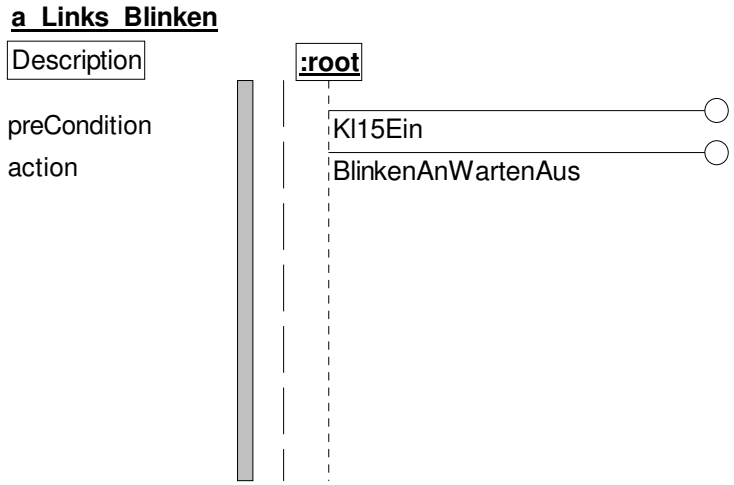


Abbildung C.16: Objekt−Sequenz−Diagramm des TestCase `a_Links_Blinken` aus Abb. C.15

quenzen zugewiesen werden sowie der Auswahl der gewünschten Zielplattform, können die TestCases ausgeführt werden. Dies ist jedoch nicht mehr Gegenstand der entwickelten Schnittstelle. Es wird daher an dieser Stelle auf Anh. B.4 verwiesen.

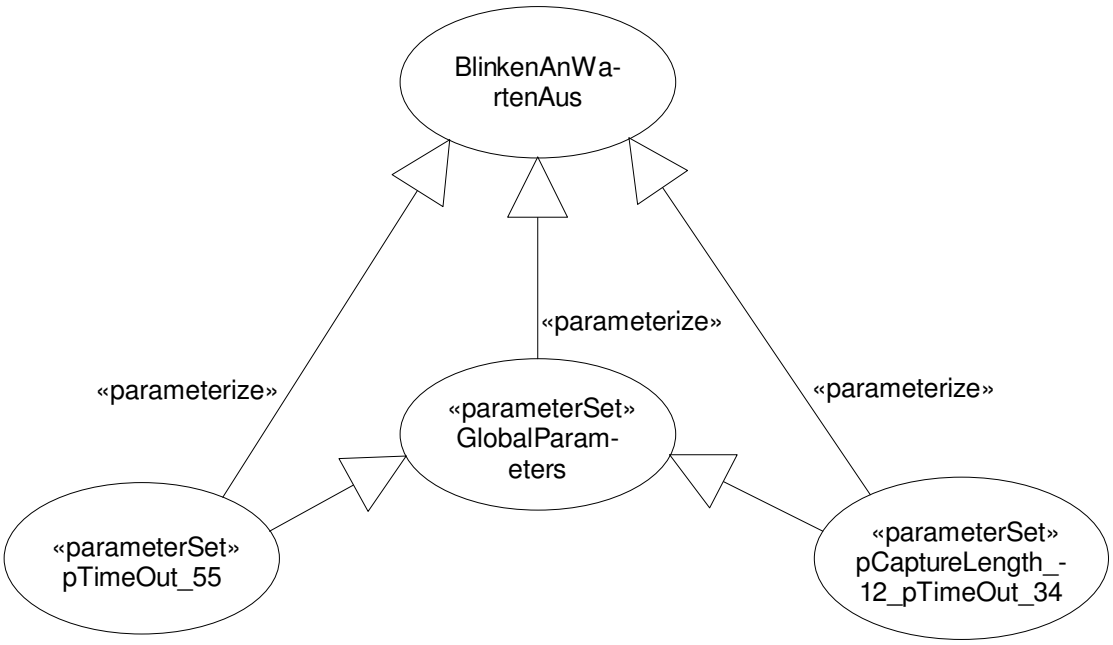


Abbildung C.17: Vollständiges Use−Case−Diagramm des action−Use−Case `BlinkenAnWartenAus` aus Abb. C.12

Literaturverzeichnis

Abran, A.; Moore, J. W.; Bourque, P.; Dupuis, R., (Hrsg.), 2004. Guide to the Software Engineering Body of Knowledge: 2004 Version – SWEBOK. IEEE Computer Society, Los Alamitos. URL http://www.swebok.org (Letzter Zugriff am 18.09.2007).

Armbrust, O.; Ochs, M.; Snoek, B., 2004a. Stand der Forschung von Software–Tests und deren Automatisierung. IESE–Report 068.04/D, Fraunhofer IESE (Institut für Experimentelles Software Engineering), Kaiserslautern. URL http://publica.fraunhofer.de/eprints/N–21987.pdf (Letzter Zugriff am 18.09.2007).

Armbrust, O.; Ochs, M.; Snoek, B., 2004b. Stand der Praxis von Software– Tests und deren Automatisierung – Interviews und Online–Umfrage. IESE–Report 093.04/D, Fraunhofer IESE (Institut für Experimentelles Software Engineering), Kaiserslautern. URL http://publica.fraunhofer.de/eprints/N–24381.pdf (Letzter Zugriff am 18.09.2007).

ARTiSAN Software Tools, Ltd., 2007. Firmenhomepage. URL http://www.artisansw.com (Letzter Zugriff am 18.09.2007).

Balzert, H., 1998. Lehrbuch der Software–Technik: Software–Management, Software–Qualitätssicherung, Unternehmensmodellierung. Spektrum Akademischer Verlag, Heidelberg.

Beizer, B., 1995. Black Box Testing – Techniques for Functional Testing of Software and Systems. John Wiley & Sons, Inc.

Bender, K., (Hrsg.), 2005. Embedded Systems – qualitätsorientierte Entwicklung. Qualitätssicherung bei Embedded Software. Springer, Berlin.

Bender, K.; Jack, P.; Koç, A.; Péter, I.; Megyeri, G., 2001. Qualitätssicherung eingebetteter Software: Methoden und Best–Practices. Technische Universität München, Lehrstuhl für Informationstechnik im Maschinenwesen.

Bergmann, J., 1998. Funktionsprüfung der Steuerungssoftware intelligenter technischer Produkte. Dissertation, Technische Universität München, Lehrstuhl für Informationstechnik im Maschinenwesen.

Bergsmann, J.; Achtert, W., 2003. Software–Qualitätsmanagement – Ansätze aus konstruktiver und analytischer Sicht. In *Fachvortrag im Rahmen der LSZ–Konferenz*

Software−Testing. Wien. URL http://www.software−quality−lab.at (Letzter Zugriff am 18.09.2007).

Bock, T.; Siedersberger, K. H.; Zaverl, M.; Breu, A.; Maurer, M., 2005. Simulations− und Testumgebung für Fahrerassistenzsysteme − Vehicle in the Loop. *VDI Berichte 1900: Erprobung und Simulation in der Fahrzeugentwicklung − Mess− und Versuchstechnik* .

Botsch, M.; Nossek, J. A., 2007. Feature Selection for Change Detection in Multivariate Time-Series. In *Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2007)*, Seiten 590 – 597.

Broekman, B.; Notenboom, E., 2003. Testing Embedded Software. Addison−Wesley.

Büchner, F., 2002. Using the classification tree method. *Embedded.com* URL http://www.embedded.com/showArticle.jhtml?articleID=15201712 (Letzter Zugriff am 18.09.2007).

Chen, T. Y.; Poon, P. L., 1997. Construction of classification trees via the classification−hierarchy table. *Information and Software Technology* 39, Seiten 889–896.

Chen, T. Y.; Poon, P. L., 1998. On the effectiveness of classification trees for test case construction. *Information and Software Technology* 40, Seiten 765–775.

Chen, T. Y.; Poon, P. L., 2004. Experience with teaching black−box testing in a computer science/software engineering curriculum. *IEEE Transactions on Education* 47(1), Seiten 42–50.

Chen, T. Y.; Poon, P. L.; Tse, T. H., 2000. An integrated classification−tree methodology for test case generation. *International Journal of Software Engineering and Knowledge Engineering* 10(6), Seiten 647–679.

Conrad, M., 2001. Beschreibung von Testszenarien für Steuergeräte−Software: Vergleichskriterien und deren Anwendung. In *10. Internationaler Kongress Elektronik im Kraftfahrzeug*. VDI, Baden−Baden.

Conrad, M., 2004. Modell−basierter Test eingebetteter Software im Automobil − Auswahl und Beschreibung von Testszenarien. Deutscher Universitätsverlag / GWV Fachverlage GmbH, Wiesbaden.

Conrad, M.; Fey, I.; Sadeghipour, S., 2004. Systematic Model−Based Testing of Embedded Control Software: The MB^{3}T Approach. In *Proc. ICSE 2004 Workshop W14S on Software Engineering for Automotive Systems (SEAS'04)*, Seiten 17–25. Edinburgh.

Conrad, M.; Fey, I.; Sadeghipour, S., 2005. Systematic Model−Based Testing of Embedded Automotive Software. In *Electronic Notes in Theoretical Computer Science 111*, Seiten 13–26.

David, A.; Rempfer, M., 2005. Elektronik im Auto – Chancen und Risiken. In *ADAC Fachgespräch: Mehr Sicherheit durch Elektronik im Auto? Chancen und Risiken moderner Fahrerassistenzsysteme*. Allgemeiner Deutscher Automobil–Club e.V., Berlin.

DeMarco, T., 1982. Controlling Software Projects: Management, Measurement & Estimation. Yourdon Press, New York.

Derichsweiler, F., 2005. Erfahrungsbericht: Modellbasierte Testautomatisierung bei Audi. In *Darmstädter Kolloquium „Softwarequalitätssicherung durch Testen – Status Quo und Visionen"*. Technische Universität Darmstadt, FG Echtzeitsysteme, Institut für Datentechnik. URL http://www.es.tu–darmstadt.de (Letzter Zugriff am 18.09.2007).

Dijkstra, E. W., 1972. Notes on Structured Programming. Structured Programming. Academic Press, London.

DIN 55350-11, 1995. Begriffe zu Qualitätsmanagement und Statistik – Teil 11: Begriffe des Qualitätsmanagements. Normenausschuss Qualitätsmanagement, Statistik und Zertifizierungsgrundlagen (NQSZ) im DIN Deutsches Institut für Normung e. V., Berlin.

DIN EN ISO 9000, 2000. Qualitätsmanagementsysteme – Grundlagen und Begriffe. Normenausschuss Qualitätsmanagement, Statistik und Zertifizierungsgrundlagen (NQSZ) im DIN Deutsches Institut für Normung e. V., Berlin.

dSPACE GmbH, 2006. Firmenhomepage. URL http://www.dspace.de (Letzter Zugriff am 18.9.2007).

Eclipse, 2007. Entwicklungsumgebung für die Programmiersprache Java, Version 3.0.1. URL http://www.eclipse.org (Letzter Zugriff am 18.09.2007).

Elmendorf, W. R., 1973. Cause–effect graphs in functional testing. Technischer Report 00.2487, IBM Systems Development Division, Poughkeepsie.

Esser, M.; Struss, P., 2007. Obtaining Models for Test Generation from Natural-language-like Functional Specifications. In *G. Biswas et. al. (eds.), DX'07, 18th International Workshop on Principles of Diagnosis*, Seiten 75–82. Nashville.

Fastenmeier, W., 2005. Mehr Verkehrssicherheit oder Risiko durch Elektronik im Auto? In *ADAC Fachgespräch: Mehr Sicherheit durch Elektronik im Auto? Chancen und Risiken moderner Fahrerassistenzsysteme*. Allgemeiner Deutscher Automobil–Club e.V., Berlin.

Frühauf, K.; Ludewig, J.; Sandmayr, H., 2004. Software–Prüfung: Eine Anleitung zum Test und zur Inspektion. vdf, Hochschulverlag an der ETH Zürich.

Grimm, K., 1995. Systematisches Testen von Software – Eine neue und effektive Teststrategie. Dissertation, Technische Universität Berlin, Fachbereich Informatik.

Grimm, K.; Grochtmann, M., 1993. Systematischer Software–Test mit der Klassifikationsbaum–Methode. In *23. GI–Jahrestagung, Dresden, 27.9.–1.10.93*, Reihe Informatik Aktuell, Seiten 252–259. Springer–Verlag, Berlin.

Guddat, U., 2003. Automatisierte Tests von Telematiksystemen im Automobil. Dissertation, Fakultät für Informations– und Kognitionswissenschaften, Eberhard Karls Universität Tübingen.

Hartmann, N., 2001. Automation des Tests eingebetteter Systeme am Beispiel der Kraftfahrzeugelektronik. Dissertation, Universität Fredericiana Karlsruhe, Institut für Technik der Informationsverarbeitung.

Heldwein, A., 1996. Funktionaler Test von Steuergeräten im Kraftfahrzeug mit der Klassifikationsbaum–Methode. Diplomarbeit, Daimler–Benz AG, Fachhochschule für Technik Esslingen, Fachbereich Mechatronik, Studiengang Feinwerktechnik.

Hennell, M. A.; Hedley, D.; Riddell, I. J., 1987. Automated testing techniques for real–time embedded software. In *Lecture Notes in Computer Science – ESEC '87 – Proceedings of the 1st European Software Engineering Conference, Strasbourg*.

IEEE Std 1044–1993, 1993. IEEE Standard Classification for Software Anomalies. The Institute of Electrical and Electronics Engineers, New York.

IEEE Std 610.12–1990, 1990. IEEE Standard Glossary of Software Engineering Terminology. The Institute of Electrical and Electronics Engineers, New York.

ISO/IEC 9126-1, 2001. Software engineering – Product quality – Quality model. ISO/IEC – International Organisation for Standardisation / International Electrotechnical Commission, Geneva.

Kiffe, G., 2006. EXAM Konzeptpapier – Konzerneinheitliche Testautomatisierung im Volkswagen Konzern. AUDI AG, Ingolstadt.

Köhler, F.; Kiffe, G., 2006. EXAM: Interview mit den EXAM–Projektleitern bei der AUDI AG und beim Volkswagen Konzern. *Micronova Kundenzeitschrift* (2), Seiten 16. URL http://www.micronova.de (Letzter Zugriff am 18.09.2007).

Lamberg, K.; Beine, M.; Eschmann, M.; Otterbach, R.; Conrad, M.; Fey, I., 2004. Model–based testing of embedded automotive software using MTest. In *Proc. of SAE World Congress 2004, Detroit, SAE Technical Paper Series 2004–01–1593*.

Liggesmeyer, P., 1990. Modultest und Modulverifikation. BI Wissenschaftsverlag, Mannheim.

Liggesmeyer, P., 2005a. Qualitätssicherung sicherheitskritischer Systeme. In *Darmstädter Kolloquium „Softwarequalitätssicherung durch Testen – Status Quo und Visionen“*. Technische Universität Darmstadt, FG Echtzeitsysteme, Institut für Datentechnik. URL http://www.es.tu–darmstadt.de (Letzter Zugriff am 18.09.2007).

Liggesmeyer, P., 2005b. Software Engineering eingebetteter Systeme: Grundlagen – Methodik – Anwendungen, Kapitel Prüfung eingebetteter Systeme, Seiten 205–225. Elsevier GmbH, München.

Lützkendorf, S., 2001. Softwaretest in Reverse Engineering–Prozessen. Diplomarbeit, Humboldt–Universität zu Berlin, Math.–Naturwiss. Fakultät II, Institut für Informatik, Lehrstuhl für Softwaretechnik.

Lützkendorf, S.; Bothe, K., 2003. Attributierte Klassifikationsbäume zur Testdatenbestimmung. In *TAV 19: Vortrag zum Treffen der Fachgruppe TAV der Gesellschaft für Informatik*. Köln.

Morell, L. J., 1990. A theory of fault–based testing. *IEEE Transactions on Software Engineering* 16(8), Seiten 844–857.

Myers, G. J., 1979. The Art of Software Testing. John Wiley & Sons, Inc., New York.

Object Management Group, 2007. URL http://www.omg.org (Letzter Zugriff am 18.09.2007).

Ostrand, T. J.; Balcer, M. J., 1988. The category–partition method for specifying and generating functional tests. *Communications of the ACM* 31(6), Seiten 676–686.

Pitschinetz, R.; Wegener, J., 2007. URL http://www.systematic–testing.com (Letzter Zugriff am 18.09.2007).

Pol, M.; Koomen, T.; Spillner, A., 2002. Management und Optimierung des Testprozesses: ein praktischer Leitfaden für erfolgreiches Testen von Software mit TPI und TMap. dpunkt.verlag GmbH, Heidelberg.

Raabe, D., 2005. Entwicklung und Verifizierung eines Tools zur automatischen Library–Generierung zur Überprüfung und Manipulation von CAN–Botschaften unter ITS (Integrated Test Services). Diplomarbeit, AUDI AG, Fachhochschule Hildesheim/Holzminden/Göttingen.

Razorcat Development GmbH, 2007. Firmenhomepage. URL http://www.razorcat.de (Letzter Zugriff am 18.09.2007).

Recknagel, M.; Rupp, C., 2006. Messbare Qualität in Anforderungsdokumenten. *Vertikal* (2), Seiten 12–17. ap Verlag GmbH.

Riedemann, E. H., 1997. Testmethoden für sequentielle und nebenläufige Software–Systeme. B. G. Teubner, Stuttgart.

Rosenberg, L. H.; Hammer, T. F.; Huffman, L. L., 1998. Requirements, Testing, and Metrics. In *Sixteenth Annual Pacific Northwest Software Quality Conference*, Seiten 107–122. Portland.

Rumprecht, B., 1994. Systematischer Test von Realzeit–Systemen mit der Klassifikationsbaum–Methode. Diplomarbeit, Technische Universität Berlin, Fachbereich Informatik, Institut für Angewandte Informatik.

Sadeghipour, S.; Lim, M.; Conrad, M., 2005. Modellbasierter Test sicherheitsrelevanter Steuergerätesoftware. *AUTOMOTIVE* (9–10), Seiten 94–96. Carl Hanser Verlag GmbH.

Schieber, R.; Derichsweiler, F., 2004. Testautomatisierung in der Automobilbranche. *OBJEKTspektrum* (4), Seiten 64–68. SIGS–DATACOM GmbH.

Schläfer, M., 2004. Konzept–Papier: ITS – Integrated Test Services. AUDI AG, Ingolstadt.

Schmid, H., 2003. Konzeption einer pragmatischen Testmethodik für den Test von eingebetteten Systemen. Dissertation, Universität Ulm, Fakultät für Mathematik und Wirtschaftswissenschaften, Abteilung angewandte Informationsverarbeitung.

Schäuffele, J.; Zurafka, T., 2003. Automotive Software Engineering: Grundlagen, Prozesse, Methoden und Werkzeuge. Viehweg Verlag, Wiesbaden.

Seiler, F., 2006. Absicherung verteilter Funktionen im Bereich Body Electronic B8. Bericht über das zweite Praxissemester, AUDI AG, Ingolstadt.

Simmes, D., 1997. Entwicklungsbegleitender Systemtest für elektronische Steuergeräte. Dissertation, Technische Universität München, Lehrstuhl für Informationstechnik im Maschinenwesen.

Spillner, A.; Linz, T., 2005. Basiswissen Softwaretest. dpunkt.verlag GmbH, Heidelberg.

Spitzer, B., 2001. Modellbasierter Hardware–in–the–Loop Test von eingebetteten elektronischen Systemen. Dissertation, Universität Fridericiana Karlsruhe, Fakultät für Elektrotechnik und Informationstechnik.

Sterler, G., 2005. Qualität und Innovationen als Erfolgsfaktoren. *Elektronik Automotive* (5), Seiten 85–88. WEKA Fachzeitschriften–Verlag GmbH.

Telelogic AB, 2007. Firmenhomepage. URL http://www.telelogic.com (Letzter Zugriff am 18.09.2007).

Vigenschow, U., 2005. Objektorientiertes Testen und Testautomatisierung in der Praxis – Konzepte, Techniken und Verfahren. dpunkt.verlag GmbH, Heidelberg.

Vitrián, E. S., 2004. Beitrag zur Ermittlung von Kosten und Nutzen der präventiven Qualitätsmethoden QFD und FMEA. Dissertation, Technische Universität Berlin, Fakultät V – Verkehrs– und Maschinensysteme.

Voigt, V., 2006. EXAM – Einführung einer konzernweiten Testautomatisierung bei der Volkswagen AG. *Micronova Kundenzeitschrift* (2), Seiten 13–15. URL http://www.micronova.de (Letzter Zugriff am 18.09.2007).

Wegener, J., 2001. Evolutionärer Test des Zeitverhaltens von Realzeit–Systemen. Dissertation, Humboldt–Universität zu Berlin.

Wohnhaas, A., 1997. Rechnergestützte Steuergeräte–Entwicklung. *ATZ – Automobiltechnische Zeitschrift* 99(12), Seiten 744–754. Vieweg Verlag / GWV Fachverlage GmbH.

Yu, Y. T.; Poon, P. L.; Ng, S. P.; Chen, T. Y., 2003. On the use of the classification–tree method by beginning software testers. In *Proceedings of the 2003 ACM symposium on applied computing*, Seiten 1123–1127. Melbourne.