UML-basierte Testfall- und Systemmodelle für die Eisenbahnleit- und -sicherungstechnik

Von der Fakultät für Maschinenbau

der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung der Würde

eines Doktor-Ingenieurs (Dr.-Ing.)

genehmigte Dissertation

von:	Dipl.-Ing. Volker Knollmann
aus (Geburtsort):	Bramsche

eingereicht am:	10. September 2007
mündliche Prüfung am:	11. Dezember 2007
Referenten:	Prof. Dr.-Ing. Karsten Lemmer
	Prof. Dr.-Ing. Jörn-Uwe Varchmin

2007

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Verkehrssystemtechnik des Deutschen Zentrums für Luft- und Raumfahrt in Braunschweig.

Ich danke daher dem Institutsleiter Herrn Prof. Dr.-Ing. Karsten Lemmer als meinem Doktorvater für die intensive und wohlwollende Unterstützung meiner Arbeit. Besonders der mir gewährte Freiraum, eigene Ideen zu entwickeln und umzusetzen, hat wesentlich zum Gelingen meiner Forschungstätigkeiten beigetragen.

Herrn Prof. a. D. Dr.-Ing. Jörn-Uwe Varchmin danke ich für die Übernahme des Korreferats und die angenehme Begleitung meiner universitären bzw. akademischen Ausbildung über gut neun Jahre.

Mein Dank gilt außerdem Herrn Prof. Dr.-Ing. Dr. h. c. Eckehard Schnieder, Leiter des Instituts für Verkehrssicherheit und Automatisierungstechnik, für die Übernahme des Vorsitzes der Prüfungskommission.

Ebenfalls danke ich der Siemens Transportation Systems AG, insbesondere Herrn Dr. Koch und Herrn Dr. Suwalski, ohne deren signifikante inhaltliche und finanzielle Unterstützung meine Arbeit nicht möglich gewesen wäre. Herrn Uwe Steinke verdanke ich wertvolle und anregende Diskussionen und einen tiefen Einblick in modellbasierte Entwicklungsansätze bei Siemens TS.

Darüber hinaus möchte ich mich bei allen Kollegen und Studenten bedanken, die in den letzten Jahren zum Gelingen meiner Arbeit beigetragen haben. Die angenehme und herzliche Arbeitsatmosphäre im Institut werde ich stets in guter Erinnerung behalten. Ebenso danke ich allen meinen Freunden, vor allem vom USC Braunschweig, die mich stets motiviert haben und mir während schwieriger Phasen meiner Arbeit mit viel Verständnis und Unterstützung begegnet sind.

Schließlich gilt mein besonderer Dank meiner Familie, insbesondere meinen Eltern, die mir den Weg durch das Studium ebneten und durch ihr beständiges Interesse am Fortschritt dieser Arbeit wesentlich zu deren Fertigstellung beigetragen haben.

Braunschweig, im Dezember 2007

Inhaltsverzeichnis

Vorwort i

Abbildungsverzeichnis v

Tabellenverzeichnis ix

Kurzfassung xi

Abstract xiii

1 Einleitung 1
1.1 Das System Eisenbahn . . . 1
1.2 Entwicklung von Leit- und Sicherungstechnik . . . 2
1.3 Bedeutung des Testens . . . 3
1.4 Einsatz von Simulationen und Modellen . . . 5
1.5 Lösungsansatz und wichtigste Ergebnisse . . . 6
1.6 Gliederung und Struktur der Arbeit . . . 8

2 Modelle und Systembeschreibungen 9
2.1 Einführung . . . 9
2.2 Modelle in der Verkehrstechnik . . . 14
2.3 Modellierung von Leit- und Sicherungstechnik . . . 15
2.4 Betriebliche Modelle . . . 19
2.5 Modellierung von Tests . . . 26

3 Beschreibungsmittel 28
3.1 Definitionen und Begriffe . . . 28
3.2 Für Systembeschreibungen notwendiger Sprachumfang . . . 31
3.3 Vorstellung einiger Beschreibungsmittel . . . 32
3.3.1 TTCN-3 . . . 37
3.3.2 LUSTRE und SCADE . . . 38
3.3.3 XML und railML . . . 40

4 Die UML und ihre Profile 43
4.1 Historie . . . 43
4.2 Bestandteile der aktuellen UML-Spezifikation . . . 43
4.3 Die Diagramme der UML . . . 46
4.3.1 Strukturdiagramme . . . 46
4.3.2 Verhaltensdiagramme . . . 48
4.4 UML-Werkzeuge und das BMW-Prinzip . . . 49
4.5 Die Systems Modeling Language (SysML) . . . 50
4.6 Das UML 2 Testing Profile (U2TP) . . . 52

5 Entwicklungsprozesse 54
5.1 Einführung . . . 54
5.2 Das V-Modell . . . 55
5.2.1 V-Modell 97 und V-Modell XT: Ein Überblick . . . 55
5.2.2 Iterativer, objektorientierter Entwurf mit dem V-Modell . . . 58
5.3 Weitere Prozesse und Vorgehensmodelle . . . 60
5.3.1 Das Wasserfallmodell . . . 60
5.3.2 Das Spiralmodell . . . 60

5.3.3 Agile Prozesse . . . 63
5.4 Systementwicklung und Zulassung . . . 64
5.5 Anforderungserfassung und -verfolgung . . . 66

6 Tests und Testumgebungen 68
6.1 Übersicht . . . 68
6.2 Testumgebungen . . . 72
6.3 Modellierung von Tests . . . 75
6.3.1 Beschreibung der Testumgebung mit U2TP . . . 76
6.3.2 Beschreibung des Testverhaltens mit U2TP . . . 78
6.3.3 Testdaten und zeitliche Randbedingungen in U2TP . . . 80
6.3.4 Aufbau von Testbibliotheken . . . 80
6.4 Automatische Testfallerzeugung . . . 81
6.4.1 Testfallerzeugung aus Verhaltensdiagrammen . . . 85
6.4.2 Testfallerzeugung durch Äquivalenzklassenbildung . . . 86

7 Verknüpfung von System- und Testfallbeschreibungen in UML 91
7.1 Lösungsansatz . . . 91
7.2 Anforderungsverwaltung und -verfolgung . . . 93
7.2.1 „Vertikales“ Nachverfolgen von Anforderungen . . . 95
7.2.2 „Horizontales“ Nachverfolgen von Anforderungen . . . 97
7.2.3 Auswertung der Anforderungsbeziehungen . . . 98
7.3 Codeerzeugung aus dem UML-Modell . . . 100
7.4 TTCN-3-Erzeugung aus dem UML-Modell . . . 101
7.5 Zusammenfassende Darstellung der gezeigten Verfahren . . . 102

8 Fallstudie 106
8.1 Die Werkzeugkopplung . . . 106
8.1.1 Das Zusammenspiel der Programme . . . 107
8.1.2 Anpassung zwischen Artisan Studio und CCS . . . 109
8.1.3 Interaktion der Pythonskripte mit dem UML-Modell . . . 111
8.1.4 Skripte zur TTCN-3-Erzeugung und zum Anforderungsmanagement . . . 112
8.2 Kurzbeschreibung des modellierten Systems . . . 116
8.3 Das Modell . . . 117
8.3.1 Die Anforderungsmodellierung . . . 118
8.3.2 Die Systemmodellierung . . . 118
8.3.3 Die Testmodellierung . . . 120
8.3.4 Vom Testmodell auf die Testanlage . . . 125
8.4 Ergebnisse und Bewertung der Fallstudie . . . 126

9 Zusammenfassung und Ausblick 127

A Abkürzungsverzeichnis 130

B Literaturverzeichnis 134

Abbildungsverzeichnis

1.1 Die Fahrzeugantennen der verschiedenen Zugsicherungssysteme einer Lokomotive der Baureihe 185 (nach [Bau02]) . . . 2
1.2 Schematische Darstellung der Entwicklungs- und Testschritte bei Anwendung des V-Modells . . 3
1.3 Zusammenspiel von Spezifikationsdokumenten und Expertenwissen des Testers bei Test- und Entwicklunsgvorgängen . . . 4
1.4 Einbettung des Phasenmodells aus dem STEP-X-Projekt in das Anforderungs- und Testmanagement (Quelle: [HH03b]) . . . 6
1.5 Schematische Darstellung des Lösungsansatzes dieser Arbeit: kombinierte UML-Modelle für alle Entwicklungs- und Testvorgänge . . . 7

2.1 Merkmale und Pragmatikausprägungen von Modellen nach [Sta73] und [Kla79] . . . 10
2.2 Transformation der Information über ein System vom Modellierer über das Modell bis zu dessen Interpretation durch den Modellnutzer . . . 11
2.3 Schematische Darstellung der zunehmenden Formalisierung der Systembeschreibung im Laufe eines Top-Down-Entwicklungsprozesses . . . 12
2.4 Zusammenspiel mehrerer eigenständiger Simulationen in einer Kosimulation . . . 14
2.5 Verschiedene, nach Ebenen sortierte Einsatzgebiete von Modellen in der Verkehrstechnik (zur Ebeneneinteilung s. auch [ESS94, Sch92]) . . . 15
2.6 Im Rahmen eines Entwicklungsprozesses für sicherheitsrelevante Systeme notwendige Dokumente (eisenbahn- bzw. ETCS-spezifische Sicht) . . . 17
2.7 Gesetze und Verordnungen für den Betrieb von Voll- und Straßenbahnen in Deutschland . . . 20
2.8 Zentrale Rolle betrieblicher Regeln im Gesamtsystem Eisenbahn . . . 22
2.9 Betriebsprozess „Ausfahrt eines Zuges“ als UML-Aktivitätsdiagramm . . . 23
2.10 Betriebsprozess “Einstellen einer Zugfahrstraße mit einem ESTW“ als UML-Aktivitätsdiagramm 24
2.11 Ausschnitt aus der Studie zur Darstellung der Beziehungen zwischen funktionalen Anforderungen an Leit- und Sicherungstechnik [WE07] . . . 26

3.1 Die Komponenten Symbole, Syntax und Semantik eines Beschreibungsmittels, ihre gegenseitigen Abhängigkeiten und Spezialisierungen . . . 29
3.2 Definition eines deutschen KFZ-Kennzeichens durch Verwendung der BNF (ohne Sonderkennzeichen und ohne unzulässige Erkennungsnummern) . . . 30
3.3 Überblick über die in dieser Arbeit angesprochenen Beschreibungsmittel und den Grad ihrer Formalität (ohne Sortierung innerhalb der einzelnen Gruppen) . . . 32
3.4 Endlicher Automat einer im Rahmen dieser Arbeit modellierten und mit dem Model Checker „UPPALL“ verifizierten SiFa (Ausschnitt) . . . 35
3.5 Temporallogische Aussagen über das in Abbildung 3.4 gezeigte System, durch die die *liveness properties* und *safety properties* mit Hilfe des Model Checkers „UPPAAL“ bewiesen werden können 36
3.6 Elemente einer TTCN-3-Testumgebung und ihre Interaktion . . . 38
3.7 Beispiel eines SCADE-Netzes . . . 39
3.8 Einsatz von railML im DLR-Bahnlabor RailSiTe zur Kopplung von Editoren und Simulationsumgebung durch den Austausch statischer Infrastruktur- und Szenariodaten . . . 41

4.1 Historische Entwicklung und Verflechtung verschiedener objekt-orientierter Methoden und Notationen, die schließlich in der UML aufgehen (Quelle: http://www.oose.de/uml_historie.htm) 44
4.2 Ebenen der Metamodellierung bei OMG-Sprachen . . . 45
4.3 Schematische Darstellung des BMW-Prinzips und seine Einbettung in das Entwicklungsumfeld 50
4.4 Komponenten der Systems Modeling Language (SysML) . . . 51
4.5 Bestandteile des UML 2 Testing Profiles . . . 53

5.1 Der klassische V-förmige Entwicklungsablauf, der bis einschließlich des „V-Modell 97“ verwendet wurde (Quelle: [Koo97]) . . . 56
5.2 Die wichtigsten Elemente des „V-Modell XT“ und ihre gegenseitigen Beziehungen als UML-Klassendiagramm . . . 57
5.3 Auswirkungen spät im Entwicklungsprozess eingebrachter Änderungen (Quelle: [Bla04]) . . . 58
5.4 Verschachtelung mehrerer V-Prozesse für die Komponenten eines ETCS-Fahrzeuggerätes . . . 60
5.5 Das Wasserfallmodell als Originalabbildung aus [Roy87]. . . . 61
5.6 Das Spiralmodell nach Barry W. Boehm (Quelle: [Boe88]) . . . 62
5.7 Weiterentwicklung des Wasserfallmodells mit seinen langen Entwicklungsphasen über iterative Ansätze (z. B. Spiralmodell) bis hin zum eXtreme Programming . . . 63
5.8 Verschiedene Wege zu einem EU-Konformitätsnachweis . . . 65

6.1 Beispielprogramm zur Veranschaulichung des Komplexitätsproblems beim Testen (aus [MSBT04]) 69
6.2 Schematische Darstellung des Unterschieds zwischen BlackBox- und WhiteBox-Testverfahren . 70
6.3 Integrationsreihenfolge eines fiktiven Beispielsystems . . . 72
6.4 Schematischer Aufbau und Datenfluss einer HiL-Testumgebung . . . 73
6.5 Bandbreite der für eine LST-Simulation samt Visualisierung auszutauschenden Daten . . . 75
6.6 Zusammenstellung mehrerer Testkomponenten zu einer individuellen Testkonfiguration (*Test Context*) . . . 77
6.7 Exemplarische Interaktion zwischen Testkomponenten und SUT in einem Sequenzdiagramm . . 78
6.8 Zustandsdiagramm eines einfachen Arbiters, der eine *worst-case*-Entscheidung vornimmt. Das `v` in den *Guard-Conditions* sei der Parameter der `setVerdict`-Methode des Arbiters. . . . 79
6.9 Exemplarisches Sequenzdiagramm, das den Einsatz von Timern zeigt . . . 80
6.10 Baumstruktur der mittels U2TP erfassten RailSiTe-Module (Ausschnitt) . . . 82
6.11 U2TP-Testkontext für eine Laborkonfiguration mit simuliertem EVC . . . 83
6.12 U2TP-Testkontext für eine Laborkonfiguration zum HiL-Test eines echten EVCs . . . 84
6.13 Ausschnitt eines Aktivitätsdiagramms zur Testfallerzeugung mit TDE/UML . . . 87
6.14 Einfache Testtopologie für die Ableitung von Stellwerkstests mit Hilfe von Klassifikationsbäumen 88
6.15 Klassifikationsbaum für ein alleinstehendes KS-Signal ohne Fahrtrichtungsanzeiger und mit zwei möglichen Geschwindigkeitssignalisierungen . . . 89
6.16 Klassifikationsbaum einer einfachen Topologie aus zwei Bahnhöfen mit eingleisiger Strecke, zwei Blocksignalen und Erlaubniswechsel . . . 89

7.1 Verbreitete Arbeitsabläufe in der Softwareentwicklung . . . 92
7.2 Verbreitete Arbeitsabläufe in der Softwareentwicklung . . . 92
7.3 Komponenten des gemeinsamen SysML- / U2TP-Modells und die Rückverfolgung von Anforderungen . . . 94
7.4 Beispiel für ein SysML-Anforderungsdiagramm mit funktionalen, verschachtelten Anforderungen 96
7.5 Beispiel für die Zuordnung zwischen funktionalen Anforderungen und Systemanforderungen . 97
7.6 „Internal Block Diagram“ der SysML mit der Verknüpfung zwischen Anforderungs- und Implementierungselementen . . . 98
7.7 Beispiel einer UML-Verknüpfung zwischen SysML-Anforderung und U2TP-Testfall . . . 99
7.8 Auszug aus einem U2TP-Modell zur Erzeugung von TTCN-3-Testskripten . . . 103
7.9 Aus dem in Abbildung 7.8 gezeigten UML-Modell erzeugter TTCN-3-Code . . . 104

8.1 Zusammenspiel und ausgetauschte Informationen der an der Fallstudie beteiligten Programme 107
8.2 Werkzeugkopplung im STEP-X-Projekt . . . 108
8.3 Diagramm der zur Kopplung von Artisan Studio und CCS notwendigen Timerklassen . . . 109
8.4 Zustandsdiagramm der zum logischen Timer gehörenden Klasse . . . 110
8.5 Einfaches Pythonskript zur Ausgabe aller Elemente mit Namen und Typ, die sich in einem UML-Modell befinden . . . 112
8.6 Ausschnitt aus der Klassenhierarchie zur Kapselung der COM-Objekte in speziellen Handlerklassen . . . 112
8.7 Sequenzdiagramm eines Testablauf mit Referenzen auf andere Sequenzdiagramme . . . 114
8.8 Grafische Oberfläche zur Erzeugung von TTCN-3-Skripten aus einem UML-Modell . . . 115
8.9 Grafische Oberfläche zur Analyse der Anforderungsbeziehungen innerhalb eines UML-Modells 116
8.10 Vereinfachtes Blockdiagramm des in der Fallstudie untersuchten Systems (grau hinterlegt) . . . 117
8.11 Strukturdiagramm der obersten Ebene der Systemarchitektur . . . 119

8.12 Modellierung der Umschaltung zwischen RailSiTe-Modus und interaktivem Modus in der Zustandsmaschine des Protokollhandlers (oberste Ebene einer hierarchisierten FSM 121
8.13 Klassifikationsbaum für die Applikationsebene des Kommunikationsprotokolls 122
8.14 Beispiel eines Unterbaums zu Abbildung 8.13; dargestellt sind die Parameter zum Speichern einer Balisentelegrammsequenz 122
8.15 Klassifikationsbaum für die Testfallableitung auf Ebene der Sicherungsschicht der DSP-RailSiTe-Kommunikation 123
8.16 UML-Klassendiagramm des U2TP-Datenpools für die Testtelegramme der Sicherungsschicht des DSP-RailSiTe-Protokolls (Ausschnitt) 124
8.17 Ausschnitt eines der Fallstudie entnommenen Sequenzdiagramms zur Testfallbeschreibung . . . 125

Tabellenverzeichnis

2.1 Einige Chancen und Risiken für den Einsatz modellbasierter Techniken in der Entwicklung von Eisenbahnleit- und -sicherungstechnik . . . 18
2.2 Inhalte der generischen Norm EN 61508 und der aus ihr für die Entwicklung von Bahnanwendungen abgeleiteten Normen der CENELEC-Reihe . . . 19

3.1 Systemeigenschaften, die zur vollständigen Systembeschreibung durch den Sprachumfang des Beschreibungsmittels abgedeckt werden müssen (vgl. auch Systemaxiome nach [Sch92, ESS94]). 32

4.1 Übersicht über alle UML-Diagramme und ihren jeweiligen Verwendungszweck . . . 47

6.1 Übersicht über die Anzahl der von CTE erzeugten Testfälle für die untersuchten Szenarien unter verschiedenen Konfigurationen . . . 90

7.1 Die wichtigsten Transformationsbeziehungen zwischen U2TP (UML) und TTCN-3 (in Anlehnung an [Ebn04]) . . . 102

8.1 Klassen der `profileHandler`-Ebene zur Interpretation der UML-Modelle unter SysML- und U2TP-Aspekten . . . 113

Kurzfassung

Der ungebrochene Trend zu immer umfangreicheren und komplexeren Automatisierungssystemen macht auch vor der Leit- und Sicherungstechnik für Eisenbahnen nicht Halt. Ein Beispiel dafür ist die Einführung des europaweit einheitlichen Zugsicherungssystems ETCS (European Train Control System), das zur Zeit auf ersten Strecken den kommerziellen Betrieb aufnimmt.

Einrichtungen wie ETCS zeichnen sich dadurch aus, dass sie Sicherheitsverantwortung für Menschen, Umwelt und Material übernehmen. An ihre Entwicklung sind daher besonders hohe Anforderungen zu stellen. Leider zeigen reale Projekte, dass sich diese Anforderungen unter Verwendung herkömmlicher Entwicklungsmethoden nur mit hohem Aufwand erfüllen lassen. Aus diesem Grund besteht reges Interesse an neuen Ansätzen für die Planung, die Umsetzung und vor allem den Test sicherheitsrelevanter Applikationen.

Die vorliegende Arbeit stellt einen Beitrag zur Weiterentwicklung der Entwurfsmethoden für derartige Systeme dar. Sie greift aktuelle Forschungsthemen aus dem Umfeld der Modellierungssprache UML (Unified Modeling Language) auf und kombiniert sie zu einem durchgängigen Verfahren für die konsistente System- und Testfallbeschreibung in einem einzigen UML-Modell.

Die Modellierung basiert auf den UML-Profilen SysML (System Modeling Language) und U2TP (UML 2 Testing Profile), die beide als offizielle Standards von der OMG (Object Management Group) spezifiziert sind. Diese Dissertation zeigt, wie ausgehend von den genannten UML-Profilen

- ein einziges, kombiniertes System- und Testmodell aufgestellt werden kann
- Beziehungen zwischen Anforderungen, Implementierung und Testfall hinterlegt werden können
- Anforderungsbeziehungen automatisiert ausgewertet werden können, um die Modellkonsistenz sicherzustellen und die Implementierungs- bzw. Testfallabdeckung zu bestimmen
- C++-Code für einen digitalen Signalprozessor aus dem Modell generiert wird
- Testskripte auf TTCN-3-Basis (Testing and Test Control Notation, Version 3) automatisch erzeugt werden
- eine Werkzeugkette aufgebaut werden kann, die alle angesprochenen Funktionen unterstützt.

Zum Nachweis der Umsetzbarkeit des Verfahrens wird im Rahmen einer Fallstudie veranschaulicht, wie sich eine Komponente des eisenbahntechnischen Labors RailSiTe® (Rail Simulation and Testing) des Deutschen Zentrums für Luft- und Raumfahrt in einem kombinierten Testfall- und Systemmodell abbilden lässt.

Ergänzend zu den methodisch-theoretischen Grundlagen der kombinierten UML-Modelle und zur umfangreichen Fallstudie werden im Rahmen dieser Arbeit auch begleitende Aspekte des Modellierungsverfahrens betrachtet. Dazu zählen beispielsweise Vergleiche mit anderen Beschreibungsmitteln (z. T. in Form kürzerer Fallstudien) oder die Einbettung objekt-orientierter Modellierung in herkömmliche Entwicklungsprozesse.

Im Ergebis zeigt sich, dass das Verfahren und die Toolkette funktionieren und sich in einem realen Entwicklungsprojekt einsetzen lassen. Die erwarteten Vorteile wie Wiederverwendung von Modellteilen, effizientere Testabläufe, formalere Systembeschreibungen oder die Vermeidung von Notationsbrüchen im Prozess zeigen sich sowohl in der theoretischen Herleitung als auch in der praktischen Anwendung der Methode. Somit ist zu hoffen, dass die hier vorgestellten Techniken künftig auch außerhalb ihres ursprünglichen wissenschaftlichen Umfelds eingesetzt werden.

Abstract

The continuous introduction of more sophisticated and more complex automation systems is of high relevance for operations- and control devices for railway applications. One example of this trend is the Euopean Train Control System (ETCS). It is harmonized for all European member states and currently deployed for commercial use on selected lines.

Applications like ETCS take responsibility for passengers, environment and the rolling stock material. For this reason, the development of those applications is characterized by high demands on testing, documentation and quality assurance. Unfortunately, real projects have prooven to achieve these high demands only with tremendous efforts as long as they stick to their traditional development processes and methods. Therefore, a wide spread exploration for new concepts regarding planing, implementation and – most of all – testing of safety relevant systems is under its way.

This thesis contributes to the search for new design methods for such systems. It combines recent achievements in the environment of the Unified Modeling Language (UML) to an integrated approach for a consistent testcase- and system description in a single UML model.

The model comprises the UML profiles SysML (System Modeling Language) and U2TP (UML 2 Testing Profile), which are officially specified and released by the Object Management Group (OMG). This thesis shows how based on these two profiles

- a single, combined system- and testcase model can be build
- relationships between requirements, implementation- and testcase can be established
- requirement relationships can be evaluted automatically to ensure model consistency and to determine implementation and testcase coverage
- C++-code can be generated directly from the model
- TTCN-3-code (Testing and Test Control Notation, Version 3) can be derived automatically from the model
- a toolchain can be set up which supports all mentioned functions

In order to proove the feasibility of this approach, a case study has been performed which shows how a component of the railway laboratory RailSiTe® (Rail Simulation and Testing) of the German Aerospace Center has been described using a combined testcase- and system model.

Apart from the case study and from the theoretical basics of combined UML models, this thesis deals with surrounding aspects of the modeling method. This includes comparisons with other means of description or embedding object-oriented modeling in traditional development processes.

The overall result is that both the approach and the toolchain are working properly and can be applied in real world projects. The anticipated advantages like re-usage of model items, more efficient testing, more formal system descriptions and avoidance of discontinuous notations in the whole process appeared in the theoretical derivation as well as in the practical usage of the method. So there is reason to believe that this approach will find future applications outside its scientific origin.

1 Einleitung

Ziel der vorliegenden Arbeit ist die Vorstellung neuer Ansätze für einen strukturierten modellbasierten Entwurf von Eisenbahnleit- und -sicherungstechnik (LST), der besonders die Themen Anforderungsmanagement, Testfallbeschreibung und Testautomatisierung berücksichtigt. Wichtigstes Element des in dieser Arbeit vorgestellten Verfahrens ist die Verwendung eines zentralen UML-Modells zur kombinierten Beschreibung von System- und Testfalleigenschaften. Es wird sich zeigen, dass dieser Ansatz erhebliche Vorteile für den gesamten Entwicklungsprozess von der Anforderungserfassung bis hin zur Integrations- und Testphase bedeutet. Von diesen Prozessschritten finden vor allem die entwicklungsbegleitenden Testvorgänge in dieser Arbeit besondere Berücksichtigung, da sie für sicherheitsrelevante Systeme – wie beispielsweise Eisenbahnsicherungstechnik – eine besondere Bedeutung haben.

Als Einführung in diese Domäne beschreiben die folgenden Abschnitte zunächst die wichtigsten Eigenschaften des Systems Eisenbahn und geben eine kurze Übersicht über die derzeitigen Entwicklungsverfahren für LST. Darauf aufbauend wird die Bedeutung des Testens und die Notwendigkeit des Einsatzes von Modellen und Simulationen hervorgehoben.

Nach dieser Vorstellung der relevanten Themenbereiche werden die wichtigsten neuen Methoden- und Lösungsansätze dieser Arbeit kurz vorgestellt. Danach schließt dieses Kapitel mit einer Übersicht über die Gliederung dieser Dissertation.

1.1 Das System Eisenbahn

Wie Pachl darstellt, zeichnet sich das System Eisenbahn durch zwei Kerneigenschaften aus [Pac00]:

Spurführung und bewegliche Fahrwegelemente: Die Spurführung macht es unmöglich, Hindernissen wie beispielsweise entgegenkommenden Fahrzeugen auszuweichen. Es muss daher sichergestellt sein, dass die vor einem Fahrzeug liegende Strecke frei ist. Daraus resultiert die Forderung nach Gegen- und Folgefahrschutz (Kollisionsvermeidung mit entgegenkommenden oder vorausfahrenden Fahrzeugen).

Die Spurführung wird unter Zuhilfenahme beweglicher Fahrwegelemente wie z. B. Weichen oder Gleissperren realisiert. Bei falscher Einstellung dieser Elemente besteht eine Gefährdung der transportierten Passagiere oder Güter durch Entgleisen des Zuges. Außerdem könnte ein Zug fehlgeleitet werden, was allerdings nicht sicherheitskritisch ist, solange der Gegenfahrschutz gewährleistet bleibt. Der vor einem Fahrzeug liegende Streckenabschnitt muss also zur Verhinderung von Entgleisungen oder Fehlleitungen richtig eingestellt und gegen (unbeabsichtigtes) Verstellen gesperrt sein. Daraus resultiert die Forderung nach der Fahrwegsicherung.

Geringe Reibung zwischen Rad und Schiene: Die Reibung zwischen Rad und Schiene (Stahl auf Stahl) beträgt ungefähr ein Achtel der Reibung zwischen Auto und Straße (Gummi auf Asphalt). Dadurch ergeben sich, verglichen mit dem Straßenverkehr, erheblich längere Bremswege, durch die das Fahren eines Zuges „auf Sicht“ auf geringe Geschwindigkeiten begrenzt ist (erlaubt sind maximal 30 – 40 km/h, je nach nationaler Betriebsordnung). Daraus resultiert, dass bei höheren Geschwindigkeiten der geforderte Gegen- und Folgefahrschutz sowie die Fahrwegsicherung durch technische oder betriebliche Verfahren sichergestellt sein muss, da eine Überprüfung durch den Lokführer nicht mehr rechtzeitig erfolgen kann.

Die Entwicklung der (Digital-)Technik und das Streben, höhere Streckenauslastungen bei geringerem Personalbedarf zu ermöglichen, hatten in den letzten Jahren und Jahrzehnten den massiven Einzug komplexer Automatisierungstechnik zur Folge, um die oben beschriebenen Sicherungsfunktionen zu realisieren.

Beispiele dafür sind die zunehmende Verbreitung elektronischer Stellwerke (ESTW) und die große Anzahl mikroprozessorbasierter Sicherungssysteme auf einer modernen Lokomotive wie z. B. der Baureihe 185. Abbildung 1.1 soll dies verdeutlichen: jede der dort aufgeführten, nationalen Zugsicherungen der BR 185 besteht aus einem eigenen, in der Regel mehrkanalig ausgeführten Rechnersystem mit eigener Software, eigenen Anzeigen etc. Neben den vielfältigen Sicherungseinrichtungen kommen noch weitere Systeme für die Zugsteuerung, Kommunikation, Passagierinformation, Komforteinrichtungen usw. zum Einsatz. Die Anzahl softwaregesteuerter Systeme in einem modernen Zug ist also erheblich.

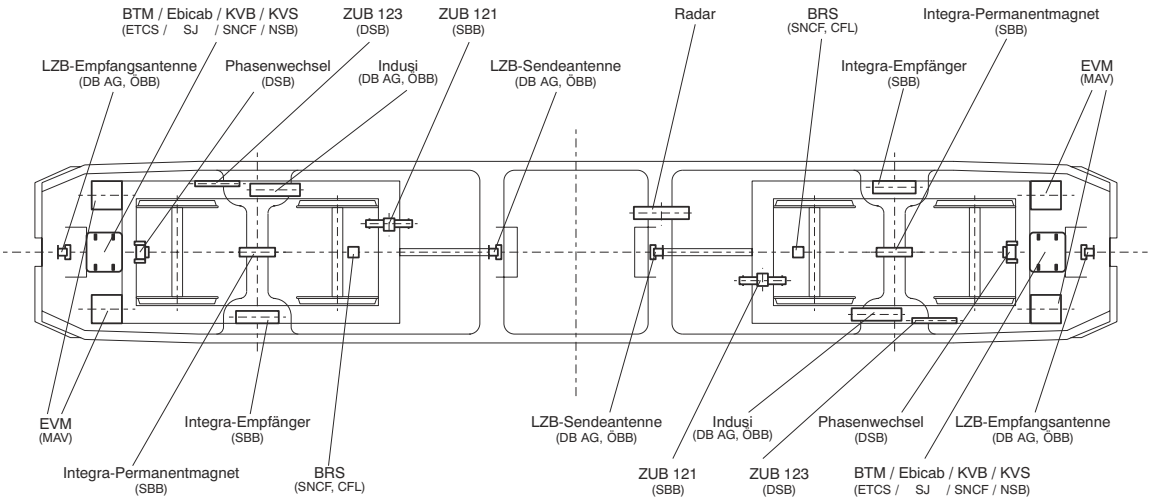

Abbildung 1.1: Die Fahrzeugantennen der verschiedenen Zugsicherungssysteme einer Lokomotive der Baureihe 185 (nach [Bau02])

Wenn entscheidende sicherheitsrelevante Funktionen wie Folge- und Gegenfahrschutz oder die Fahrwegsicherung durch solche technischen Systeme realisiert werden, übernehmen sie zwangsläufig eine hohe Sicherheitsverantwortung, da ein Ausfall oder ein Fehlverhalten des Gerätes die Gefährdung oder sogar den Verlust von Menschenleben zur Folge haben könnte.

Dieser Sicherheitsverantwortung muss durch entsprechende Entwicklungsprozesse, ausführliche Tests und behördliche Zulassung der Systeme Rechnung getragen werden. Ein Kapitel dieser Arbeit befasst sich daher genauer mit den zugrunde liegenden Normen und Zulassungsbedingungen für die Entwicklung sicherheitskritischer Systeme.

1.2 Entwicklung von Leit- und Sicherungstechnik

Für die Entwicklung von LST kommen bis heute fast ausschließlich auf dem V-Modell [Koo97, Koo05] basierende Prozesse zum Einsatz. Abhängig von der Größe des Entwicklungsprojektes weist die Ablauforganisation unterschiedlich viele Zwischenstufen und Verknüpfungen zu anderen Prozessen auf (z. B. Projektmanagement, Produktmanagement, Qualitätssicherung). Der grundsätzliche Ablauf entspricht jedoch immer dem in Abbildung 1.2 gezeigten Vorgehen.

Die Entwicklungsvorgaben, also die Anforderungen des Auftraggebers bzw. Anwenders, liegen dabei typischerweise in Prosaform vor. Die Anwendung formaler Methoden oder fertiger Textmuster, wie sie beispielsweise von Rupp vorgeschlagen werden [RS04], ist derzeit praktisch nicht gegeben.

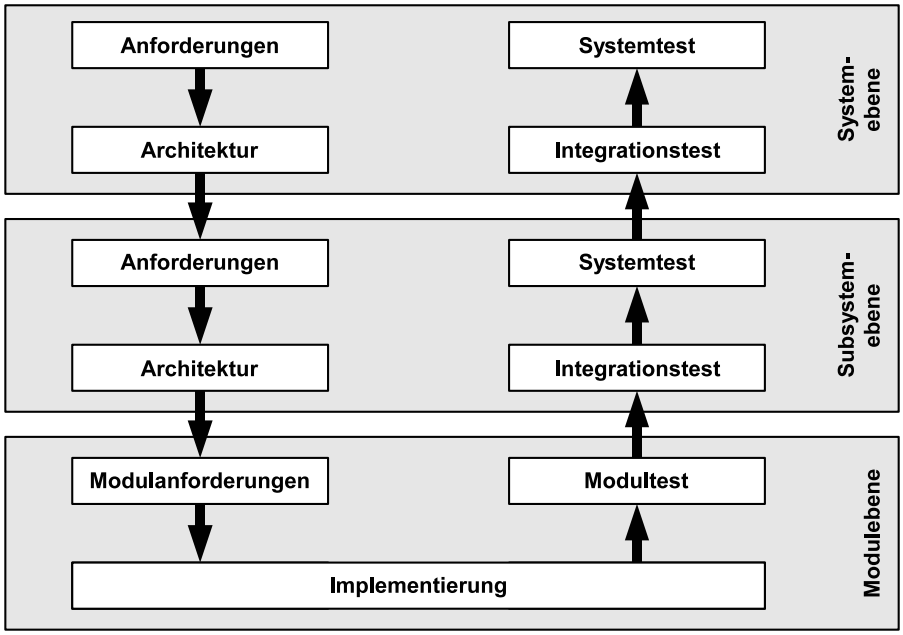

Abbildung 1.2: Schematische Darstellung der Entwicklungs- und Testschritte bei Anwendung des V-Modells

Das gleiche trifft häufig auch auf die innerhalb der Entwicklung angefertigten Spezifikationen wie Schnittstellen- oder Systemarchitekturbeschreibungen zu. Mitunter sind auf diesen Ebenen aber bereits strukturierte Darstellungen vorzufinden, die beispielsweise jede Anforderung nach einem festen Schema als Tabellenzeile erfassen. Allerdings ist auch diese Form der Beschreibung – obwohl strukturiert – hochgradig informal.

Infolgedessen treten über den gesamten Entwicklungsprozess Widersprüche, Unvollständigkeiten und Auslegungsfragen hinsichtlich der zugrunde liegenden Dokumente auf. Letztlich liegen diese Probleme und Defizite in der Komplexität der zu entwickelnden Systeme begründet. Komplexe Sachverhalte können nicht durch einfache Beschreibungen definiert werden, sondern verlangen komplexe Darstellungen. Mit informalen Beschreibungsmitteln, wie der natürliche Sprache, sind solche komplexen Darstellungen allerdings nur unzureichend und uneinheitlich zu beherrschen. Besonders offensichtlich wird dies unter Berücksichtigung der unterschiedlichen Rollen und Hintergründe der am Projekt beteiligten Personen (eher technisch oder betrieblich orientierte Personen, Programmierer, Manager, Gutachter, ...). In einem solch heterogenen Nutzerkreis können mit rein textuellen Beschreibungen nur selten bei allen Beteiligten das gleiche Systemverständnis und identische Vorstellungen und Assoziationen hervorgerufen werden.

Dabei erhöhen die „Reibungsverluste“ durch Mehraufwand an Kommunikation die Entwicklungskosten ebenso wie spät erkannte Fehler oder Fehlentwicklungen. Entscheidend dabei ist, dass die Beseitigung eines Fehlers um so aufwändiger und teurer wird, je später er im Projektverlauf gefunden wurde.

Eine verbesserte Qualität der Spezifikationsdokumente – möglichst durch den Einsatz formaler Beschreibungen – und frühes Testen versprechen eine signifikante Reduzierung der Entwicklungskosten. Aufgrund der geringen Stückzahlen in der Fertigung von Eisenbahnleit- und -sicherungstechnik entspricht eine deutliche Senkung der Entwicklungskosten auch eine deutliche Senkung des Produktpreises, was für den Kunden natürlich von entscheidender Bedeutung ist.

1.3 Bedeutung des Testens

Im letzten Abschnitt wurde bereits angedeutet, dass das Testen einen erheblichen Einfluss auf die Entwicklungskosten hat. Es darf also bei der Planung nicht vernachlässigt werden und muss als ein integraler Be-

standteil einer jeden Produktentwicklung berücksichtigt werden. Obwohl dies seit Langem bekannt ist, wird Testen dennoch häufig als etwas Separates, als etwas Außenstehendes wahrgenommen: es wird primär als Verursacher von zeitlichen und finanziellen Mehraufwänden gesehen, die sich darüber hinaus auch noch schwer abschätzen lassen.

Es gibt seit einiger Zeit Bestrebungen, diese negative Wahrnehmung von Tests zu verbessern und durch geeignete Vorgehensweisen die Verbindung von Entwicklung und Test effizienter zu gestalten. Exemplarisch für einen solchen Ansatz sei hier das Projekt „STEP-X“ genannt, in dem am Beispiel eingebetteter Systeme aus dem Automobilbereich ein konsistenter, modellbasierter Entwicklungs- und Testprozess erarbeitet wurde. Zur Verbesserung der Testabläufe wurde im Rahmen von STEP-X ein vollständiges Testrahmenwerk abgeleitet, das neben UML- bzw. petrinetzbasierter Modellierung auch die (automatisierte) Testfallableitung und die Auswahl geeigneter Hardware berücksichtigt [Hor05]. Im Ergebnis zeigte sich nicht nur eine gesteigerte Produktqualität, sondern auch eine Reduzierung der Testaufwände bei besserer Beherrschbarkeit und Planbarkeit des Testprozesses.

Neben zeitlichen und finanziellen Risiken kann aber Testen auch personelle Herausforderungen mit sich bringen. Wie Myers sehr anschaulich zeigt, sind die Anforderungen an einen guten Tester erheblich höher als die an einen guten Entwickler [MSBT04]. Ein Tester braucht zunächst die Fähigkeiten eines Entwicklers, um das System und seine Realisierung zu verstehen. Darüber hinaus benötigt er aber vor allem Erfahrung und Instinkt, um Tests durchzuführen und zu definieren, die tatsächlich ein Fehlverhalten des Systems hervorrufen und damit einen Fehler offenbaren.

Dieses Expertenwissen ist bis heute die entscheidende Grundlage für die Ableitung der meisten Testfälle. Ein solches Vorgehen birgt jedoch eine Reihe entscheidender Nachteile, weshalb seit einigen Jahren verstärkt nach Methoden und Techniken gesucht wird, um die Testfallableitung systematischer und nachvollziehbarer zu gestalten und sie von einem individuellen Tester und seinem Expertenwissen zu entkoppeln.

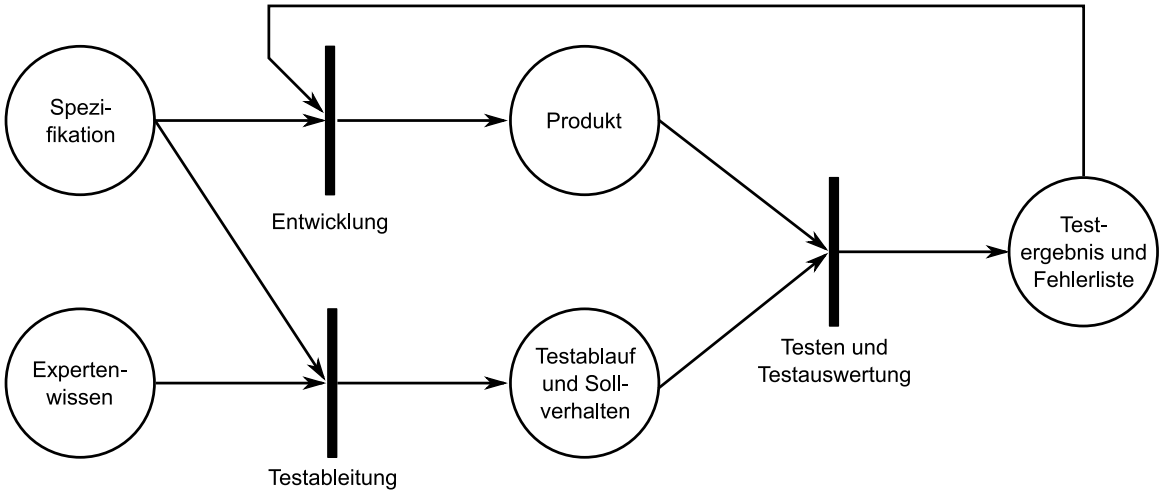

Abbildung 1.3: Zusammenspiel von Spezifikationsdokumenten und Expertenwissen des Testers bei Test- und Entwicklunsgvorgängen

Allerdings zeigt Abbildung 1.3, dass der Tester und sein Expertenwissen nicht die einzigen Voraussetzungen für gelungene Tests sind. Da Testen das Ziel hat, über die Korrektheit des Systemverhaltens zu urteilen, muss vor allem definiert werden, was das korrekte Verhalten tatsächlich ist. An dieser Stelle schließt sich der Kreis zu den weiter oben erwähnten Spezifikationsdokumenten: nur wenn die Anforderungen das Systemverhalten vollständig, eindeutig und widerspruchsfrei beschreiben, können zugehörige Testfälle abgeleitet und ausgewertet werden [1].

[1] Der bekannte Ausspruch „It's not a bug, it's a feature!“ beschreibt diesen Zusammenhang sehr treffend

Aus diesem Grund ist ein entscheidender Aspekt dieser Arbeit die formalere Darstellung von Systemspezifikationen, um dadurch eine geeignete Grundlage für eine methodische und überprüfbare Testfallableitung zu schaffen. Gleichzeitig soll durch eine formalere, grafische Repräsentation der Testfälle und Testumgebungen eine automatische Testausführung ermöglicht und die Wiederverwendung bzw. Anpassung von Tests erleichtert werden.

1.4 Einsatz von Simulationen und Modellen

Art und Umfang von Tests sind eng an den Entwicklungsablauf gebunden. Daher sind entscheidende Aussagen zum Systemverhalten prozessbedingt häufig erst spät möglich, was Komplikationen und Mehrkosten mit sich bringt. Exemplarisch wird dies anhand des V-Modells aus Abbildung 1.2 deutlich. Es zeigt, dass bei der Entwicklung zunächst *top-down* eine Dekomposition des Systems vorgenommen wird und dadurch die Anforderungen immer weiter bis auf die unterste Ebene (üblicherweise die sogenannte Modulebene) heruntergebrochen werden. Die Implementierung setzt dann *bottom-up* auf der untersten Ebene ein und führt über die kontinuierliche Integration von Teilkomponenten schließlich zum Gesamtsystem.

Dieses Vorgehen hat den entscheidenden Nachteil, dass Aussagen zum Verhalten des Gesamtsystems erst zu einem sehr *späten* Entwicklungszeitpunkt möglich sind. Die Annahmen und Entscheidungen, die in einem *frühen* Stadium der Top-Down-Phase getroffen wurden und grundlegend für die weitere Verfeinerung der Spezifikationsdokumente sind, können erst in einer *späten* Phase der Integration überprüft und notfalls korrigiert werden. Die späte Aufdeckung und Beseitigung von Fehlern führt jedoch, wie bereits erwähnt wurde, zu hohen Mehrkosten.

Seit einigen Jahren versucht man daher, diesen prinzipbedingten Einschränkungen des V-Modells (und anderer „schwergewichtiger“ Prozesse wie beispielsweise des Wasserfall- oder Spiralmodells) durch den Einsatz von Simulationen und Modellen zu begegnen, falls eine Verschlankung der Prozessabläufe nicht möglich ist.

Durch Modelle sollen das System bzw. seine wesentlichen Teile vor Beginn der Implementierung nachgebildet werden. Diese Nachbildung wird dann gegen den Einsatzzweck des Systems validiert oder gegen das geforderte Verhalten verifiziert. Nach abgeschlossener Überprüfung kann dann die Implementierung beginnen, deren Ergebnisse wiederum gegen das Modell verifiziert werden können. Letzten Endes nimmt so das Modell die Rolle der Spezifikation ein. Dieses Vorgehen wird als modellbasierte Entwicklung oder kurz MDD bezeichnet („Model Driven Development“).

Entscheidend ist, dass das MDD vor allem eine konsequente Umsetzung der Forderung aus Abschnitt 1.2 nach formaleren Systembeschreibungen darstellt, denn ein ausführbares Modell kann als vollständig formale Systemspezifikation angesehen werden. Und selbst ein nicht ausführbares, semiformales Modell kann Fehler beispielsweise durch Konsistenzprüfungen oder Erzeugung von Codeteilen von Beginn an wirkungsvoll verhindern.

Die positiven Effekte digitaler Spezifikationen und ausführbarer Modelle werden unter anderem durch die Ergebnisse des bereits erwähnten Projekts STEP-X belegt. Im Rahmen von STEP-X wurde ein Strukturierter Entwicklungsprozess am Beispiel von X-by-Wire-Applikationen vorgestellt. Der Prozess (s. Abb. 1.4 greift dazu intensiv auf Modelle und Simulationen zurück. Bereits auf Ebene des funktionalen Grobentwurfs und des Architekturentwurfs wird das Systemdesign simulativ überprüft, was signifikant zur Fehlervermeidung bzw. frühzeitigen Fehleraufdeckung beiträgt. Durch Einsatz eines Regel-Checkers kann darüber hinaus die Konsistenz der entstandenen Modelle sichergestellt werden [Mut03, MP03]. Als Besonderheit werden diese Modelle innerhalb von STEP-X nicht nur zur Unterstützung der Entwicklungs- und Testvorgänge eingesetzt, sondern auch zur Diagnose der Systeme im späteren Einsatzumfeld herangezogen. Durch Analyse der im Modell hinterlegten Strukturinformationen – Komponenten und deren (Kommunikations-)Beziehungen – können Diagnosesysteme konfiguriert und parametriert und somit der Diagnosevorgang vereinfacht werden

[HH03a]. Durch die Ableitung der Diagnoseinformationen aus dem Entwicklungsmodell ist darüber hinaus die Diagnose eng mit dem Entwicklungsprozess verknüpft. Die drei genannten STEP-X-Kernaspekte (Systementwicklung, Test und Diagnose) sind in ein umfassendes Anforderungs- und Testmanagement integriert, das die einzelnen Prozessphasen miteinander verbindet und dadurch die Grundlage des System- und Testentwurfs darstellt (Abb. 1.4).

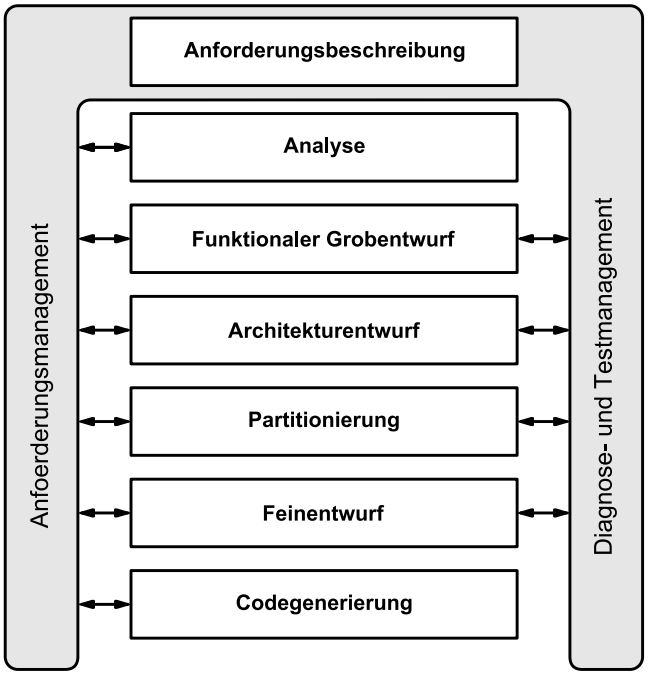

Abbildung 1.4: Einbettung des Phasenmodells aus dem STEP-X-Projekt in das Anforderungs- und Testmanagement (Quelle: [HH03b])

Mit Hinblick auf den Systemtest ist die Simulation der Systemumgebung mindestens ebenso wichtig wie die bereits angesprochene reine Systemsimulation. Durch die Umgebungssimulation kann ein Entwicklungsprototyp mit quasi-realen Stimuli beaufschlagt werden, um aus den Reaktionen eine Aussage zur Konformität des Prüflingsverhaltens treffen zu können. Dieses Vorgehen wird auch als „Hardware-in-the-Loop“ (HiL) bezeichnet. Liegt in früheren Entwicklungsschritten noch kein vollständig integrierter Prototyp bestehend aus Zielhard- und -software vor, kann auch nur die Applikation oder deren Modell in die Umgebungssimulation eingebunden werden. In diesem Falle spricht man von Software- bzw. Model-in-the-Loop (SiL / MiL).

Bei diesem Vorgehen ist also nicht nur ein Modell des Systems, sondern auch seiner (Test-)Umgebung notwendig. Diese Arbeit soll daher zeigen, wie diese beiden Modellklassen auf Basis der „Unified Modeling Language“ (UML) in einem einzigen, konsistenten Gesamtmodell erfasst und als zentrales Element im Entwicklungsprozess eingebunden werden können.

1.5 Lösungsansatz und wichtigste Ergebnisse

Wie eingangs bereits angedeutet, wurde das im vorigen Absatz beschriebene „konsistente Gesamtmodell“ im Rahmen dieser Arbeit durch den Einsatz der UML unter Verwendung erst kürzlich spezifizierter UML-Profile realisiert. Während in früheren Ansätzen häufig auf die Kombination mehrerer Beschreibungsmittel zurückgegriffen werden musste (das thematisch sehr eng verwandte STEP-X-Projekt verwendet beispielsweise UML und Petrinetze), erlaubt die Verfügbarkeit dieser Profile, alle relevanten Informationen in einem einzigen Modell zu hinterlegen. Dies ermöglicht eine geschlossenere Darstellung und reduziert Schnittstellen-

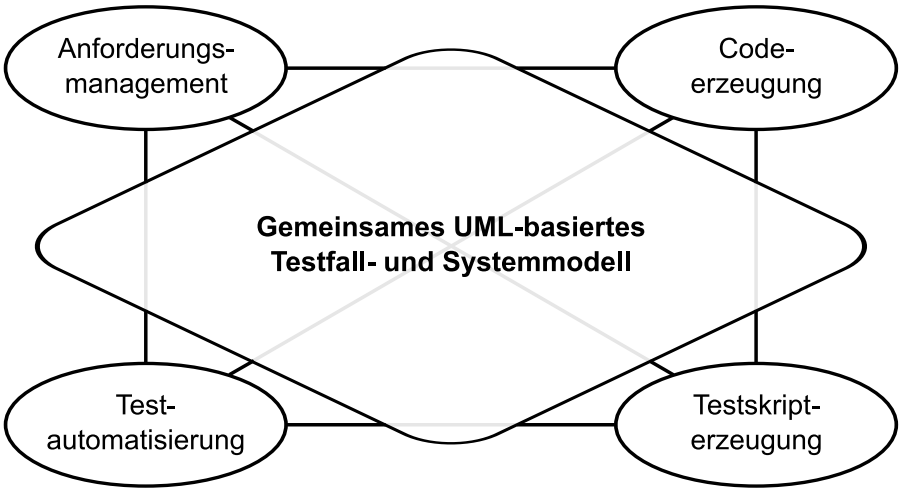

Abbildung 1.5: Schematische Darstellung des Lösungsansatzes dieser Arbeit: kombinierte UML-Modelle für alle Entwicklungs- und Testvorgänge

und Werkzeugaufwände. Hinsichtlich der Testbeschreibungen beschränkt sich diese Arbeit auf Black- und GreyBox-Tests. Diese Limitierung ergibt sich aus den Eigenschaften des für Testbeschreibungen ausgewählten UML-Profils. Für eine Erweiterung des hier vorgestellten UML-Ansatzes um WhiteBox-Testmöglichkeiten sei auf [Hor05, Lig90, Lig02] verwiesen.

Neben aktuellen Entwicklungen auf dem Gebiet der UML und der UML-Profile flossen auch Ergebnisse aus der Analyse realer Anforderungsdokumente für Zugsicherungssysteme, Erfahrungen aus System- und Konformitätstests für ETCS (ETCS: European Train Control System) und Erkenntnisse aus dem Aufbau des DLR-Simulationslabors RailSiTe® (Rail Simulation and Testing) in diese Arbeit ein.

Auf dieser breiten Grundlage wird die vorliegende Dissertation herausarbeiten, wie sich die vorgeschlagenen kombinierten Testfall- und Systemmodelle methodisch und technisch realisieren lassen und wie sie die Systementwicklung in entscheidenden Punkten unterstützen (Abb. 1.5):

- Systemanforderungen, -architektur und -design lassen sich zusammen mit Testdaten und Testfällen in einem einzigen UML-Modell ablegen.

- Aus dem UML-Modell erzeugter Code kann ohne weitere Maßnahmen direkt auf einem digitalen Signalprozessor ausgeführt werden.

- Aus den im UML-Modell hinterlegten Tests lassen sich direkt ausführbare Testskripte in der Sprache TTCN-3 erzeugen.

- Die im UML-Modell hinterlegten Beziehungen zwischen Anforderungen, Implementierung und Testfällen lassen sich automatisiert auswerten und erlauben jederzeit Rückschlüsse auf die Modellkonsistenz und die Fortschritte bei der Implementierung und der Testfallabdeckung.

- Die automatische Erzeugung von Testfällen auf Basis von Klassifikationsbäumen ist eine wirkungsvolle Unterstützung im Entwicklungsprozess und lässt sich in das UML-basierte Verfahren integrieren.

Die Machbarkeit des Lösungsvorschlags wird anhand einer praktischen Umsetzung im Rahmen einer Fallstudie empirisch nachgewiesen.

1.6 Gliederung und Struktur der Arbeit

Die Arbeit beleuchtet zunächst das wissenschaftliche Umfeld, in das die System- und Testfallmodellierung von Eisenbahnleit- und -sicherungstechnik eingebettet ist. Dazu zählt vor allem die Auseinandersetzung mit Modellen und Beschreibungsmitteln, aber auch eine intensive Betrachtung des Themas Testen. Auf diesen Betrachtungen aufbauend werden die verwendeten UML-Profile sowie das methodische und praktische Vorgehen zu deren Kopplung vorgestellt. Eine Fallstudie zeigt schließlich die Anwendbarkeit und Realisierbarkeit des vorgeschlagenen Ansatzes. Am Ende werden die wichtigsten Ergebnisse zusammengefasst und ein Ausblick auf weiterführende Arbeiten gegeben. Im Einzelnen ergibt sich daraus die folgende Gliederung:

Dieses Kapitel umreisst Kontext der Arbeit, erläutert ihre Ansätze und stellt die Gliederung der nachfolgenden Kapitel vor.

In Kapitel 2 werden die Eigenschaften von Modellen, deren Komponenten und Anwendungsbereiche eingeführt. Darüber hinaus wird der Einsatz von Modellen für die Darstellung betrieblicher und technischer Vorgänge in der Eisenbahndomäne sowohl theoretisch wie auch an Beispielen verdeutlicht.

Im darauf folgenden Kapitel 3 wird der Begriff des Beschreibungsmittels eingeführt. Besonderes Augenmerk wird dabei auf den Formalitätsgrad verschiedener Beschreibungsmittel und deren Spezifikation gelegt. Basierend auf den im vorangegangenen Kapitel definierten Modellkomponenten wird der notwendige Sprachumfang von Beschreibungsmitteln für eine vollständige Systembeschreibung abgeleitet. Die Kurzbeschreibung einiger Beschreibungsmittel rundet das Kapitel ab.

Eine intensive Auseinandersetzung mit der Unified Modeling Language ist Gegenstand des Kapitels 4. Darin werden unter anderem die Geschichte der UML und der Aufbau ihrer Spezifikation erläutert. Das dabei eingeführte Prinzip der Metamodellierung wird von den anschließenden Abschnitten über die UML-Profile SysML und U2TP aufgegriffen und angewendet.

Der Frage nach geeigneten Vorgehensmodellen für die objektorientierte Entwicklung sicherheitsrelevanter Systeme wird in Kapitel 5 nachgegangen. Aufgrund seiner weiten Verbreitung wird das V-Modell dabei genauer untersucht. Darüber hinaus sind das Wasserfall- und Spiralmodell ebenso wie agile Methoden Gegenstand der Analyse. Da gerade im Eisenbahnbereich der Entwicklungsprozess eng mit den Themen Zulassungen und Genehmigungen verknüpft ist, werden auch diese Aspekte zusammen mit dem ebenfalls wichtigen Anforderungsmanagement behandelt.

Das umfangreiche Gebiet Testen ist Gegenstand des Kapitels 6. WhiteBox- und BlackBox-Testmethoden werden darin ebenso betrachtet wie Testumgebungen, Testautomatisierung und die Testfallbeschreibungssprache TTCN-3. Als ein für die nachfolgende Fallstudie wichtiger Punkt werden ebenfalls aktuelle Methoden der automatischen Testfallerzeugung dargestellt.

Kapitel 7 legt die methodischen Grundlagen für die Verknüpfung von System- und Testfallmodellen. Dazu wird intensiv auf die Nachverfolgung von Anforderungen auf die Implementierungs- und Testfallebene sowie auf die Code- und TTCN-3-Erzeugung aus dem Modell eingegangen.

Die zentrale Fallstudie in Kapitel 8 führt alle zuvor erarbeiteten theoretischen Ansätze in einem praktischen Beispiel zusammen. Dafür wird zunächst die Kopplung der in der Fallstudie eingesetzten Werkzeuge genau beschrieben, um danach die einzelnen Modellkomponenten des Beispielsystems im Detail vorzustellen.

Das abschließende Kapitel 9 fasst die wichtigsten Ergebnisse zusammen und bietet einen Ausblick auf mögliche weiterführende Arbeiten.

2 Modelle und Systembeschreibungen

Wie bereits in der Einleitung erwähnt, ist die Modellierung von Eisenbahnleit- und -sicherungstechnik ein Schwerpunkt dieser Arbeit. Daher folgt in diesem Kapitel eine Einführung in den Modellbegriff und die verschiedenen Aspekte und Komponenten eines Modells.

Daran schließt sich ein allgemeiner Überblick über die Verwendung von Modellen in der Verkehrstechnik an, bevor dann explizit auf die Modellierung von Leit- und Sicherungstechnik eingegangen wird.

Über die Erstellung technischer Modelle aus dem Eisenbahnbereich hinaus wird anschließend auch die Modellierung betrieblicher Prozesse behandelt.

Das Kapitel schließt mit einem kurzen Vorgriff auf die Modellierung von Tests ab, auf die in einem späteren Teil der Arbeit genauer eingegangen wird.

2.1 Einführung

Der Begriff „Modell“ wird im Alltag und in der (wissenschaftlichen) Literatur in unterschiedlichsten Bedeutungen und Definitionen verwendet. Entstanden ist der Begriff aus dem lateinischen „Modulus“ bzw. dem italienischen “Modello“ und diente als Bezeichnung für „Muster“, „Vorlage“ oder „Maß“ und wurde vornehmlich in der Architektur und Kunst verwendet [SS99].

Mittlerweile erstreckt sich der Gebrauch des Begriffes auf viele weitere Felder, wie beispielsweise auf mathematische Modelle aus der Physik oder Verhaltensmodelle in der Psychologie. Die Verwendung geht hin bis zu den Software- und Systemmodellen aus den Ingenieurswissenschaften und der Informatik, die Gegenstand dieser Arbeit sind.

Wie in Abbildung 2.1 dargestellt, sind allen Modellen dabei folgende drei Merkmale gemein [Sta73]:

Abbildung: Ein Modell ist stets die Repräsentation eines wie auch immer gearteten Originals (das wiederum ein Modell sein kann).

Verkürzung: Ein Modell gibt nicht alle Eigenschaften des Originals wieder, sondern nur die, die dem Modellierer oder Modellbenutzer relevant erscheinen.

Pragmatik: Modelle ersetzen das Original für bestimmte Nutzer, für ein bestimmtes Zeitintervall und für bestimmte Operationen oder Handlungen. Ein Modell ist also nur unter bestimmten Randbedingungen anwendbar. Bei der Verwendung eines Modells muss daher sorgfältig geprüft werden, ob es für den jeweiligen Einsatz geeignet ist oder ob der Gültigkeitsbereichs des Modells bereits verlassen wurde.

Die Einsatzgebiete von Modellen variieren mit ihrer Pragmatik, können aber nach [Kla79] in vier Gruppen aufgeteilt werden:

- Funktionsersatz, -erweiterung oder -verstärkung (z. B. Regler)
- Verhaltensregulation (z. B. Training in einem Flugsimulator)
- Erkenntnisgewinnung (z. B. Klimarechenmodelle)
- Kenntnisvermittlung und -speicherung (z. B. anatomische Bildtafeln in der Medizin)

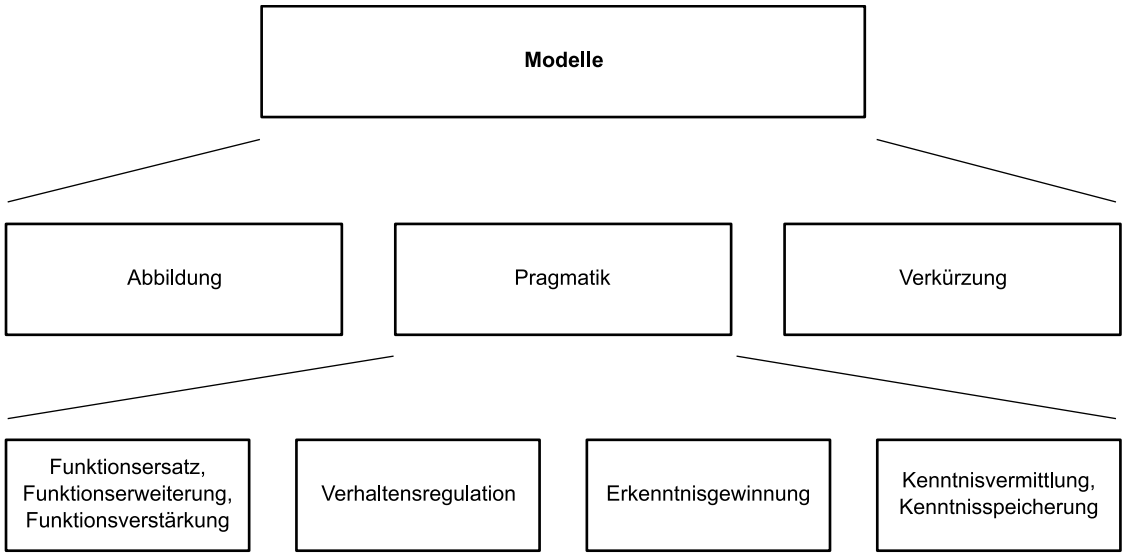

Abbildung 2.1: Merkmale und Pragmatikausprägungen von Modellen nach [Sta73] und [Kla79]

Die Modelle, die in dieser Arbeit behandelt werden, sind überwiegend den beiden letztgenannten Punkten zuzuordnen. Dabei fällt die Erkenntnisgewinnung primär mit der Phase der Modellerstellung zusammen, da begleitend zur Modellierung eines neu zu entwickelnden Systems viele Fragen aufgeworfen werden, die sich nicht direkt aus den Vorgaben – z. B. der Anforderungsspezifikation – beantworten lassen. Der Modellierungsvorgang an sich dient daher bereits der Verbesserung des Systemverständnisses und der frühzeitigen Klärung unpräziser oder widersprüchlicher Vorgaben.

Die Kenntnisvermittlung setzt ein, wenn das Modell als Vorgabe für nachfolgende Entwicklungsschritte verwendet wird. Es dient dann der Weitergabe des im vorherigen Schritt gewonnen Systemverständnisses an andere Arbeitsgruppen oder andere Entwickler. Ebenso kann aber auch gegenüber dem Auftraggeber das Systemverständnis der Entwickler veranschaulicht werden. Der Kunde kann dadurch leichter einschätzen, ob seine Anforderungen richtig gedeutet und umgesetzt werden, was die Risiken einer späteren Systemvalidierung mindert.

Ein entscheidender Punkt bei der Kenntnisvermittlung durch Modelle ist die Frage der semantischen Eindeutigkeit. Nur wenn die Interpretation des Modells durch den Modellersteller und den Modellbenutzer identisch ist, wird der Modellinhalt „richtig“ vermittelt (siehe dazu auch Abbildung 2.2). Dazu müssen die Elemente, aus denen sich das Modell zusammensetzt (*Symbole*) und ihre Kombinationen (*Syntax*) in ihrer Bedeutung (*Semantik*) möglichst präzise und unzweideutig definiert sein. Symbole, Syntax und Semantik sind durch das *Beschreibungsmittel* vorgegeben, in dem das Modell abgefasst ist. Aber selbst wenn der Modellnutzer exakt die gleiche Information aus dem Modell zieht, die der Modellersteller basierend auf seinem „Notationswissen“ (dem Wissen um die syntaktischen und semantischen Möglichkeiten des Beschreibungsmittels) in das Modell überführt hat, kommt es schon alleine durch die unterschiedlichen Persönlichkeiten und Hintergründe der beteiligten Personen zu „Defekten“ im wahrgenommenen Bild des Originalsystems bzw. des gewünschten Zielsystems. Diese Defekte sind prinzipbedingt und grundsätzlich unabhängig vom verwendeten Beschreibungsmittel. Naturgemäß treten diese Defekte bei informalen Beschreibungsmitteln (Prosatext) allerdings am stärksten auf. (Semi-)Formale Beschreibungen zwingen den Ersteller dagegen zu größerer Sorgfalt und decken Widersprüche, Auslassungen und Inkonsistenzen in der Beschreibung leichter auf.

Das Beschreibungsmittel stellt somit gewissermaßen das Rückgrat eines Modells dar und limitiert durch seinen Sprachumfang auch den möglichen Modellinhalt. Aspekte, die sich mit dem gewählten Beschreibungsmittel nicht ausdrücken lassen, können nicht in das Modell aufgenommen werden. Daher muss vor Beginn der

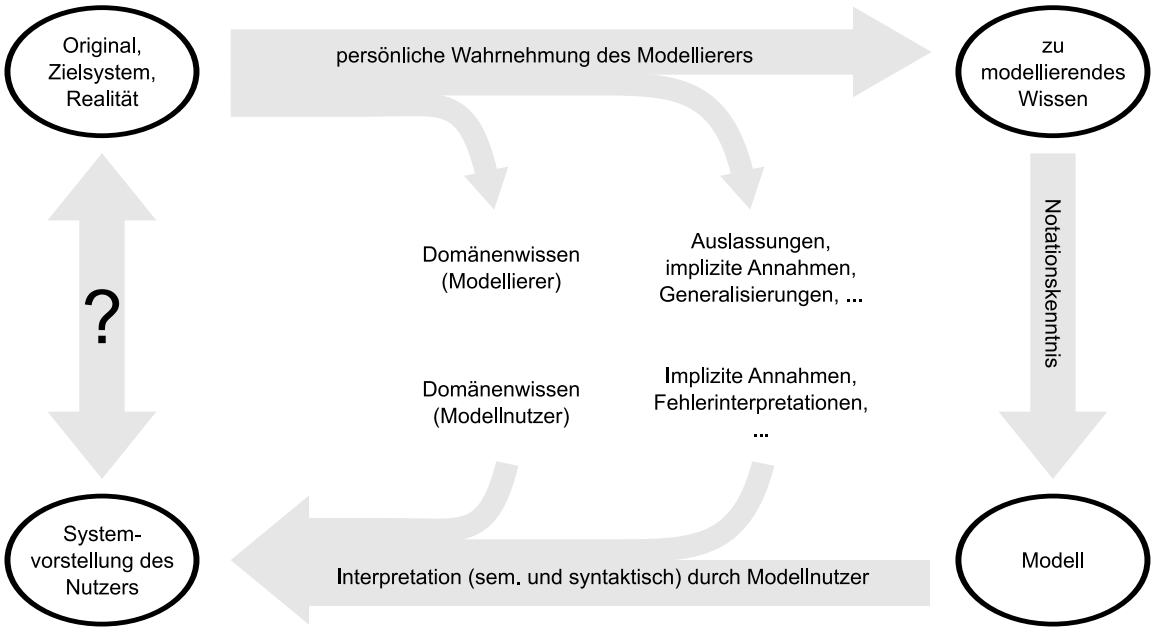

Abbildung 2.2: Transformation der Information über ein System vom Modellierer über das Modell bis zu dessen Interpretation durch den Modellnutzer

Modellierung das geeignete Beschreibungsmittel sorgfältig gewählt werden. Sollte sich später im Laufe der Modellerstellung die gewählte Notation als unzulänglich herausstellen, ist der dadurch notwendige Wechsel des Beschreibungsmittels üblicherweise nur mit erheblichem Aufwand möglich. Das Folgekapitel geht daher genauer auf Beschreibungsmittel und ihre Eigenschaften ein und stellt darüber hinaus einige gängige Notationen ausführlicher vor.

Formale Modelle

Die oben erwähnte Eindeutigkeit bei der Interpretation des Modells ist dann erreicht, wenn das Modell unter Rückgriff auf ein formales Beschreibungsmittel erstellt wurde. Ein solches „formales Modell“ kann mit geeigneten Werkzeugen auf einem Computer lauffähig gemacht und simuliert werden. Ein einfaches Beispiel für ein solches Modell ist die Beschreibung eines gewünschten Systemverhaltens in einer gängigen Hochsprache wie C oder Java. Mit Hilfe des zugehörigen Compilers bzw. Interpreters kann das „Modell“ ausführbar gemacht und „simuliert“ werden.

Ausführbare Modelle bieten den Vorteil, dass das Modellverhalten interaktiv oder sogar in der späteren Systemumgebung überprüft werden kann. Kunde bzw. Entwickler können auf diese Weise sehr leicht eine Validierung des Modells vornehmen und dadurch die Qualität des voraussichtlichen Entwicklungsergebnisses besser beurteilen und beeinflussen.

Dieser Ansatz wird schon seit einiger Zeit bei der Entwicklung neuer Applikationen verfolgt. Ein gängiges Konzept in diesem Zusammenhang ist „Rapid Application Development“ (RAD). RAD zielt vornehmlich auf die schnelle und möglichst automatisierte Erzeugung von Benutzeroberflächen und ist daher für die Programmierung eingebetteter Systeme (z. B. der Sicherungsrechner einer Lok) nur bedingt geeignet.

Ein RAD-verwandtes Vorgehen ist die grafische Programmierung von Steuerungs- oder Regelungsalgorithmen, die dann durch einen geeigneten Cross-Compiler auf einem eingebetteten Zielsystem lauffähig gemacht werden (beispielsweise „Matlab-Simulink“ (Modeller) zusammen mit dem „Realtime Workshop“ (Compiler) und der „DSpace-Box“ (Zielrechner, Laufzeitumgebung)). Der Test der so erzeugten Programme erfolgt in der simulierten oder echten Betriebsumgebung der Applikation, so dass sehr schnell belastbare Aussagen über die Qualität der implementierten Algorithmen getroffen werden können. Dieser Umstand und die Tatsache,

dass die „Round-Trip-Time“ – der Zeitaufwand zwischen einer Änderung an den Programmquellen bis zum Vorliegen der neu compilierten, lauffähigen Applikation – sehr gering ist, hat zu der Bezeichnung „Rapid Prototyping“ für dieses Vorgehen geführt. Allerdings lässt sich die Komplexität der Sicherungsprogramme eines Stellwerks oder eines Zuges mit diesen Methoden nicht beherrschen, so dass auch dieser modellbasierte Ansatz im Bereich der LST keine signifikante Verbreitung gefunden hat.

Generell ist die Einführung *formaler* Modelle in herkömmliche top-down-basierte Entwicklungsprozesse nicht einfach. Die Forderung nach möglichst früher Verwendung des formalen Modells im Entwicklungsprozess (um maximalen Nutzen aus dem Modell zu ziehen) kollidiert mit dem grundsätzlichen Paradigma des Top-Down-Ansatzes (Bild 2.3), nach dem das Systemverhalten zunächst unscharf formuliert wird, um dann in weiteren Entwicklungsschritten zunehmend konkretisiert zu werden. Der Einsatz eines formalen Modells auf einer zu frühen Entwicklungsstufe kann daher in der Regel gar nicht realisiert werden. Daher muss, um maximalen Nutzen aus dem Einsatz formaler Modelle zu ziehen und um Nachteile zu vermeiden, der Übergangspunkt auf formale Systembeschreibungen im Entwicklungsprozess sorgfältig abgewogen werden.

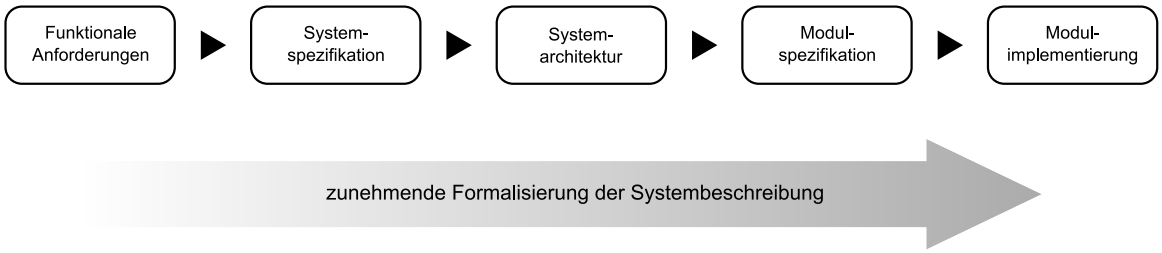

Abbildung 2.3: Schematische Darstellung der zunehmenden Formalisierung der Systembeschreibung im Laufe eines Top-Down-Entwicklungsprozesses

Um in frühen Entwicklungsstadien dennoch von den Vorteilen einer modellbasierten Systembeschreibung profitieren zu können, sollte dort auf *semiformale* Beschreibungen zurückgegriffen werden. Diese erlauben eine gewisse Unschärfe bei der Modellierung und sind in jedem Fall einer informalen – sprich: rein textuellen – Beschreibung vorzuziehen. Dies ist auch der Ansatz der vorliegenden Arbeit, die die Verwendung von UML zur System- und Testfallbeschreibung in frühen Entwicklungsstadien vorstellt.

Modellkomponenten und Modellklassen

Um ein System in all seinen Ausprägungen zu erfassen, muss ein Modell verschiedene Aspekte des Systems erfassen. Dies sind[1]:

- *statische* Eigenschaften wie Systemarchitektur, Abgrenzung zur Außenwelt, Dekomposition in Subsysteme und Schnittstellen; kurz: alle Systemaspekte, die keiner zeitlichen Änderung unterliegen.

- *dynamisches* Verhalten wie Systemzustände, Zustandsübergänge, Datenaustausch zur Umwelt oder zu Subsystemen; kurz: alle Systemaspekte, die von der Zeit und der Systemhistorie abhängen. Dazu zählen sowohl die *temporale Dynamik*, die das System*verhalten* über der Zeit beschreibt, als auch die *kausale Dynamik*, die die System*funktionen*, ihre Zusammenhänge und ihre Auswirkungen auf den Systemzustand definiert [JS01].

- *nicht-funktionale* Anforderungen wie Zuverlässigkeit, Verfügbarkeit, Wartung / Instandhaltung, Sicherheit (RAMS(S) – Reliability, Availability, Maintainability, Safety, Security), Dokumentation, Prozess- und Entwicklungsabläufe, etc. In der Praxis werden vor allem die nicht-funktionalen Anforderungen häufig in ihrer Bedeutung unterschätzt und bei der Systemspezifikation vernachlässigt [RS04].

[1] Auf eine etwas andere Aufteilung in Struktur, Verhalten, Zustand und Funktion wird in Abschnitt 3.2 eingegangen

Hinsichtlich der Erfassung des Systemverhaltens – der dynamischen Aspekte – können Modelle in zwei grundsätzliche Klassen unterteilt werden. Die erste Klasse bilden die sogenannten zeitkontinuierlichen Modelle, in denen die Zeit stetig voranschreitet und die für gewöhnlich durch Differentialgleichungen dargestellt werden. Beispiele für solche Modelle finden sich in der Regelungs- und der Nachrichtentechnik oder in der Elektrotechnik bei der Analyse von Wechselstromnetzwerken.

Die zweite Klasse umfasst die zeitdiskreten Verhaltensbeschreibungen, in denen die Zeit in Takte oder Zeitscheiben eingeteilt ist und angenommen wird, dass sich das System zwischen zwei Takten nicht verändert. Eine Änderung des Systemzustandes ist also nur zu diskreten Zeitpunkten möglich und erfolgt nicht kontinuierlich wie bei der erstgenannten Modellklasse. Zeitdiskrete Modelle sind in der Informatik und der Elektrotechnik weit verbreitet und für gewöhnlich als sogenannte Zustandsautomaten (z. B. als Moore- oder Mealy-Automat, Beschreibung in [Bal01]) realisiert. Eine weitere wichtige Form zeitdiskreter Modelle sind diskretisierte Lösungen an sich kontinuierlicher Differentialgleichungen. Derartige Modelle basieren auf den gleichen Grundlagen wie zeitkontinuierliche Systeme, allerdings erfolgt die Berechnung bzw. Simulation unter Berücksichtigung diskreter Zeitschritte (beispielsweise bei digitalen Filtern).

Häufig kann ein System nicht eindeutig einer Verhaltensklasse zugeordnet werden, da es sowohl diskrete wie kontinuierliche Aspekte umfasst. Anschaulich wird dies bei der Betrachtung eines Autoantriebs: die Gangschaltung kann als Zustandsautomat betrachtet werden, bei dem es zu diskreten Übergängen kommt (Gangwechsel). Die Fahrdynamik (also die Bestimmung von Momentanposition, -geschwindigkeit und -beschleunigung) wird dagegen am besten als kontinuierliches System basierend auf Differentialgleichungen erfasst. Eine Modellierung des Gesamtsystems würde daher kontinuierliche *und* diskrete Aspekte umfassen. Beschreibungen derartiger Systeme werden daher auch als *hybride* Modelle bezeichnet.

Simulationen hybrider Modelle stellen eine besondere Herausforderung dar, da herkömmliche Simulationsumgebungen entweder diskrete oder kontinuierliche Modelle verarbeiten können. Außerdem kommt es in der Regel zu einem Notationsbruch bei der Modellierung, da die meisten Beschreibungsmittel nur eine Form der Verhaltensbeschreibung zulassen.

Als gängige Lösung wird eine getrennte Simulation des diskreten und des kontinuierliches Teils und die dynamische Kopplung der beiden Simulationen (eine sogenannte Kosimulation, Abb. 2.4) herangezogen. Die Kopplung darf eventuell an die Simulation gestellte Echtzeitanforderungen nicht verletzen und muss außerdem die Schnittstellen der jeweiligen Simulationsumgebungen bedienen können (diese sind nicht einheitlich). Neben der dynamischen Kopplung muss auch sichergestellt sein, dass statische Aspekte wie das Simulationsszenario und Simulationsparameter in beiden Umgebungen identisch sind.

Für die Kopplung von Simulationsumgebungen gibt es einige generische Ansätze wie beispielsweise „EXITE“ der EXTESSY AG[2], aber in der Regel sind individuelle bzw. proprietäre Lösungen notwendig.

Selbstverständlich ist eine Kosimulation nicht auf einen diskreten und einen zeitkontinuierlichen Teil beschränkt. Kopplungen verschiedener Simulationsumgebungen können ganz allgemein immer dann eingesetzt werden, wenn zwei eigenständig simulierte Teilsysteme zu einer Gesamtsimulation ergänzt werden sollen. Ein Beispiel dafür ist das DLR-Bahnlabor RailSiTe® (Rail Simulation and Testing), in dem eine modulare ETCS-Simulation mit einer monolithischen Stellwerkssimulation und einer ebenfalls monolithischen 2D-Visualisierung gekoppelt wurde. Dabei wurde hohe Sorgfalt auf die Konsistenz und Integrität der statischen Daten (z. B. Streckentopologie) und einen zeitgerechten Austausch der dynamischen Daten (z. B. Zugpositionen) gelegt[KGL07].

Das eben erwähnte RailSiTe ist nur ein Beispiel für die Umsetzung der in dieser Einführung beschriebenen Modellierungs- und Simulationskonzepte. Die folgenden Abschnitte bieten daher einen Überblick über die weitere Verwendung von Simulationen und Modellen im Umfeld der Eisenbahnleit- und -sicherungstechnik. Sie betrachten dazu die Verkehrstechnik auf Gesamtsystemebene sowie technische und betriebliche Model-

[2] http://www.extessy.com

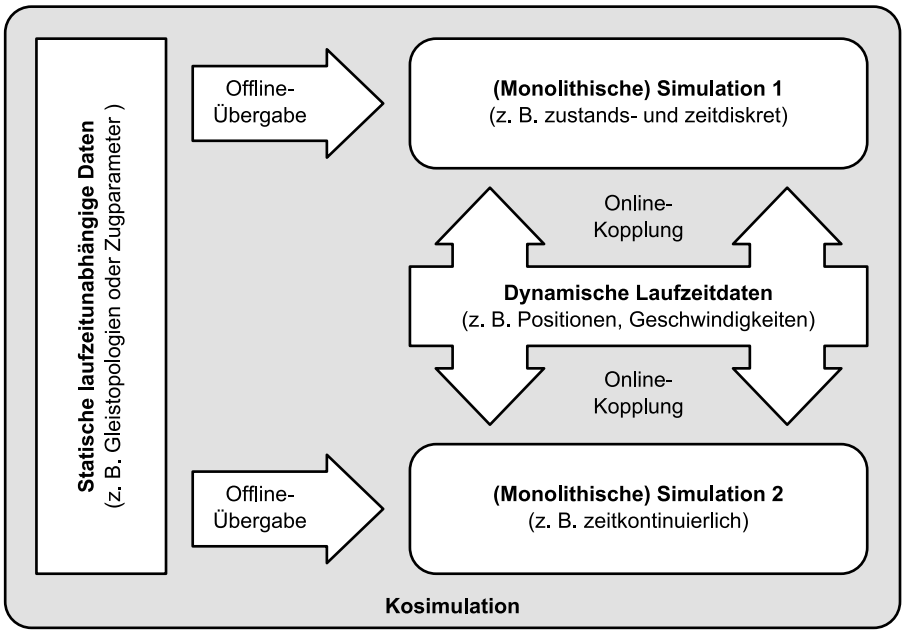

Abbildung 2.4: Zusammenspiel mehrerer eigenständiger Simulationen in einer Kosimulation

le aus der Eisenbahndomäne und zeigen auf, mit welchen Ansätzen und Pragmatiken in diesen Bereichen gearbeitet wird.

2.2 Modelle in der Verkehrstechnik

Modelle und Simulationen werden in der Verkehrstechnik schon seit geraumer Zeit eingesetzt. Anwendungen wie das im letzten Absatz erwähnte Simulationslabor RailSiTe stellen nur den letzten Schritt einer Entwicklung dar, in deren Verlauf eine Vielzahl von Modellierungs- und Analysemethoden für die Verkehrstechnik adaptiert oder entwickelt wurden.

Im Sinne der vier grundsätzlichen Modellzwecke (siehe Seite 9) werden Modelle des Gesamtsystems „Verkehr“ am häufigsten zur Erkenntnisgewinnung eingesetzt (vgl. Abbildung 2.5). Durch die Erfassung von Verkehrsstatistiken und die Ableitung der Beziehungen zwischen den verschiedenen Messgrößen wird beispielsweise versucht, das zukünftige Verkehrsaufkommen und den Mobilitätsbedarf zu prognostizieren. Die daraus gewonnenen Erkenntnisse dienen dann *strategischen* Entscheidungen wie beispielsweise der Netzentwicklung (Straßenbau, Schienenbau). Der zeitliche Horizont solcher Modelle liegt typischerweise im Bereich von Jahren und Jahrzehnten.

Auf *dispositiver* Ebene werden Modelle zur Prognose von Fahrgastzahlen im Tages- oder im Jahresverlauf (Tagesgang / Jahresgang) herangezogen. Als Grundlage werden auch hier statistische Erfassungen herangezogen, aber auch besondere Randbedingungen (wie beispielsweise Großereignisse) finden Berücksichtigung. Die Prognoseergebnisse dienen der Fahrplanentwicklung sowie der Personal- und Fahrzeugplanung. Der zeitliche Horizont liegt dabei im Bereich einiger Tage bis hin zu mehreren Monaten.

Auf *taktischer* bzw. *operativer* Ebene werden vor allem Steuerungs- und Entscheidungsfindungsprozesse betrachtet. In diesem Bereich finden sich auch die später in der Fallstudie vorgestellten Leit- und Sicherungstechnikmodelle wieder. Allerdings können auch Modelle zur Fahrdynamik, zur Abstandshaltung von Fahrzeugen oder zum Fahrerverhalten dieser Gruppe zugeordnet werden. Modelle dieser Klasse berücksichtigen das Verhalten im Bereich von Sekunden bis hin zu mehreren Stunden.

Vor allem die langfristige Prognose der Verkehrsentwicklung ist ausgesprochen vielschichtig. Sie muss ver-

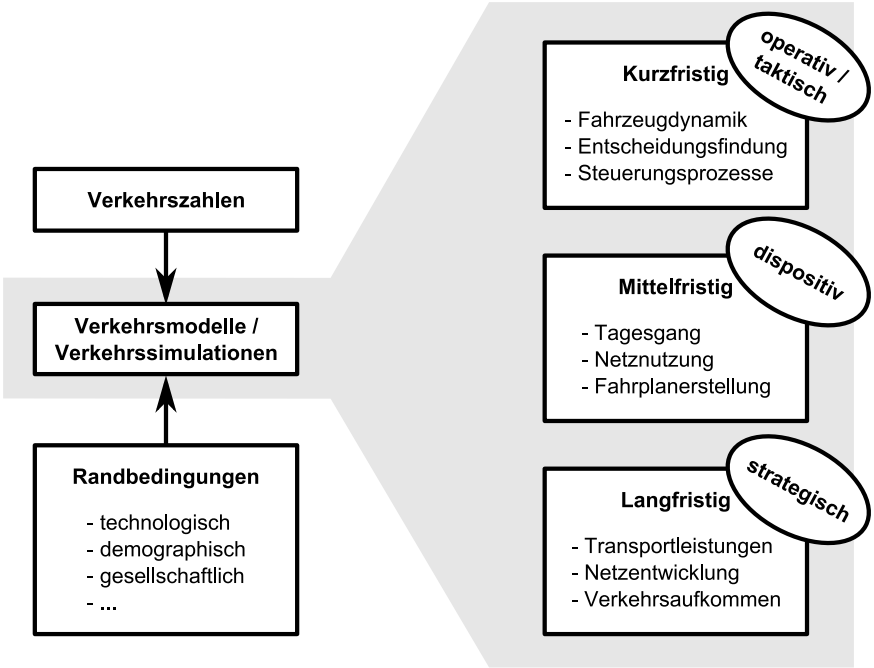

Abbildung 2.5: Verschiedene, nach Ebenen sortierte Einsatzgebiete von Modellen in der Verkehrstechnik (zur Ebeneneinteilung s. auch [ESS94, Sch92])

schiedenste Randbedingungen berücksichtigen (technisch, gesellschaftlich, demografisch, ...) und ist durch ihre strategische Ausrichtung Grundlage für die Entscheidung über den Einsatz großer Geldmittel. Dazu zählt vor allem der Ausbau der Infrastruktur bzw. die Netzentwicklung allgemein, aber auch die Entwicklung neuer Technologien. Erst eine langfristige Prognose kann neue Trends und Märkte aufzeigen und ist somit eine wichtige Grundlage für Innovation und Entwicklung.

Die hier auf Gesamtverkehrsebene vorgestellte Unterteilung in vier Ebenen kann auch auf einzelne Verkehrsträger heruntergebrochen werden. Eine Ebenenunterteilung für die Aufgaben im Bahnverkehr stellt beispielsweise Fay vor [Fay99]. Allgemein stehen in der Eisenbahndomäne entweder *technische* oder *betriebliche* Aspekte auf operativ/taktischer oder dispositiver Ebene im Vordergrund. Entsprechend wird der folgende Abschnitt die Beschreibung technischer Systeme behandeln, während die Modellierung betrieblicher Prozesse im darauf folgenden Abschnitt ab Seite 19 betrachtet wird.

2.3 Modellierung von Leit- und Sicherungstechnik

Spezifischer als die zuvor erläuterten Modelle auf Ebene des Gesamtsystems Verkehr ist die Erstellung technischer Modelle für die Eisenbahndomäne. Primärer Zweck dieser technischen Modelle ist die Unterstützung des Entwicklungsprozesses. Mit ihnen werden System- und Testanforderungen (selten), Systemstruktur und Systemverhalten (häufiger) erfasst. Dabei variiert von Unternehmen zu Unternehmen aber auch von Projekt zu Projekt der Einsatzpunkt, ab dem auf Modelle zurückgegriffen wird.

Aus Sicht der Entwicklung lassen sich mit Modellen zwei grundsätzliche Schnittstellen bedienen:

interne Schnittstellen: Austausch von Informationen primär zwischen verschiedenen Entwickler- oder Testergruppen, aber auch zum Management. Im letzteren Fall ist häufig das für das Modellverständnis notwendige Vorwissen (z. B. Kenntnis der Notation) nicht sehr ausgeprägt, so dass in diesem Fall nur bedingt auf Modelle zurückgegriffen werden kann. Für den ersten Aspekt, die Kommunikation auf technischer Ebene, sind Modelle allerdings hervorragend geeignet.

externe Schnittstellen: Austausch von Informationen zum Kunden und zu anderen an der Entwicklung beteiligten Partnern (andere Firmen innerhalb eines Konsortiums, Gutachter, Zulassungsbehörden). Charakteristischerweise ist die Nutzergruppe dieser Schnittstelle ist sehr heterogen zusammengesetzt und umfasst Personen mit Fokussierung auf Betrieb, Technik, Sicherheit oder Prozess- oder Qualitätsmanagement.

An diesen Schnittstellen wird eine Vielzahl von Dokumenten ausgetauscht, wie die Abbildung 2.6 zeigt. Von diesen Dokumenten eignet sich naturgemäß vor allem der Zweig der „Technischen Spezifikationen / Definitionen“ für eine modellbasierte Darstellung. Dazu zählen funktionale und nicht-funktionale Anforderungen, Schnittstellenbeschreibungen, Systemarchitektur / Dekomposition und Ergebnisse der Systemanalyse. Ebenso gut sind formale, modellbasierte Darstellungen von Testspezifikationen möglich, wie spätere Kapitel dieser Arbeit zeigen werden.

Die Beschreibung von Anforderungen und ihrer geplanten Umsetzung findet, wie bereits erwähnt, bis heute überwiegend in textueller Form statt. Als zentrale Dokumente gelten dabei das *Lastenheft* (auftraggeberseitige Anforderungen) und das *Pflichtenheft* (auftragnehmerseitiges Fachfeinkonzept bzw. Lösungsansätze). Beide Begriffe sind in der DIN 69901 bzw. DIN 69905 definiert und erläutert [DIN87, DIN97]. Eine Bespielgliederung für ein umfassendes Lasten- und Pflichtenheft kann auch der VDI/VDE-Richtlinie 3694 entnommen werden [VDI91]. Im allgemeinen zielen modellbasierte Ansätze darauf ab, die durch die DIN- und VDI/VDE-Vorschriften geforderten Inhalte des Pflichtenheftes und, wenn möglich, auch des Lastenheftes durch ein entsprechendes Systemmodell zu ersetzen, auf dem dann die weitere Entwicklung aufsetzen kann.

Eine Auswertung einiger realer Entwicklungsprojekte ergab jedoch, dass die zahlreichen Möglichkeiten modellbasierter Darstellungen momentan nur selten umfassend genutzt werden [Uht06]. Beispielsweise hatten alle Projekte gemein, dass an der externen Schnittstelle (vor allem zum Kunden) nach wie vor nur Volltextdokumente (Prosa) ausgetauscht wurden. Bestenfalls war eine tabellarische, strukturierte Aufstellung von Anforderungen anzutreffen. Darüber hinausreichende formale Dokumente wurden nicht verwendet. Dabei könnten gerade durch eine präzisere Darstellung (nicht-)funktionaler Anforderungen sowie von Architektur- und Implementierungsansätzen viele Unklarheiten von Anfang an vermieden werden.

Zwar ist eine zu starke Formalisierung initialer Anforderungen im Sinne der zu Abbildung 2.3 gehörigen Ausführungen schwierig, der Übergang auf eine zumindest semiformale Darstellung jedoch wünschenswert. Saghi und Tarnai skizzieren in diesem Zusammenhang, wie eine schrittweise Einführung formaler Techniken in den Entwicklungsablauf gestaltet werden kann [ST03]. Kennzeichnend für die Einführung ist dabei eine längere Übergangszeit, in der formale, semiformale und informale System- und Anforderungsbeschreibungen parallel bzw. ergänzend verwendet werden. Im Laufe dieser in verschiedene Phasen eingeteilten Übergangszeit wird zunächst die Systembeschreibung formalisiert, um anschließend die Kundenanforderungen in eine formale Darstellung zu überführen.

Hinsichtlich der Systembeschreibung ist festzustellen, dass aktuell vor allem statische Modellaspekte modellbasiert erfasst werden. An erster Stelle ist hier der UML-gestützte Entwurf der Softwarearchitektur zu nennen. Eine Ausweitung dieser Methode auf die gesamte Systemarchitektur ist jedoch nicht zu beobachten. Vereinzelt wird auch Systemverhalten mit Hilfe von Aktivitäts- oder Zustandsdiagrammen erfasst, allerdings bleibt das in der Eisenbahndomäne eine Ausnahme.

Entsprechend wenig Verbreitung findet die automatische Erzeugung von Code aus Modellen Verwendung. Auch hier ist am ehesten die Erzeugung der grundlegenden Klassendefinitionen aus UML-Strukturdiagrammen zu nennen. Nur vereinzelt ist die Umsetzung von Systemverhaltensmodellen in Code (beispielsweise aus Zustandsdiagrammen mit Hilfe von Werkzeugen wie „StateMate“ oder „Stateflow®“) anzutreffen.

Betrachtet man diese Ansätze aus der Perspektive des gesamten Entwicklungsprozesses, wird offensichtlich, dass die ergänzende Einführung modellbasierter Techniken in bestehende Umgebungen aus mehreren Gründen durchaus problematisch sein kann (vgl. Tab. 2.1):

Abbildung 2.6: Im Rahmen eines Entwicklungsprozesses für sicherheitsrelevante Systeme notwendige Dokumente (eisenbahn- bzw. ETCS-spezifische Sicht)

Ausweitung der „Toollandschaft“: Die Einführung neuer Techniken bedingt auch immer eine Einführung neuer Werkzeuge, an denen die Mitarbeiter geschult werden müssen und die eine Einarbeitungszeit verlangen. Darüber hinaus fallen Lizenz- und Wartungskosten an.

Erlernen einer zusätzlichen Methode: Alle an der Entwicklung beteiligten Personen müssen die neuen Methoden und Techniken erlernen und vor allem *akzeptieren*. Da dies initial einen erhöhten Arbeitseinsatz erfordert, der den ohnehin häufig anzutreffenden Zeitdruck noch verschärft, muss der Nutzen der Neuerung unmittelbar erkennbar sein.

Pflege und Weiterentwicklung von Altsystemen: Vorhandene Daten über Altsysteme (in der Softwareentwicklung auch als *Legacy System* bezeichnet) liegen selbstverständlich nicht modellbasiert vor. Eine Übertragung auf die neuen Methoden und Werkzeuge bedeutet in der Regel einen Aufwand, der nicht gerechtfertigt ist, da neben der Konvertierung der Systembeschreibung auch Folgekosten für neue Sicherheitsnachweise, Gutachten, Validierungen etc. anfallen können. Daher muss an Altsystemen parallel noch mit “alten“ Methoden weitergearbeitet werden. Bedenkt man die langen Lebenszyklen von Eisenbahnleit- und -sicherungstechnik, ist dies ein Punkt von hoher Relevanz.

Notationsbruch innerhalb des Prozesses: in allen analysierten Projekten wurden modellbasierte Verfahren nur punktuell eingesetzt. Dies führt dazu, dass System- und Testfallbeschreibungen auf unterschiedlichen Dekompositionsstufen in unterschiedlichen Formaten abgelegt werden müssen. Das bedeutet Mehraufwände für die wiederholte Darstellung des prinzipiell gleichen Sachverhaltes. Darüber hinaus müssen die daraus entstehenden redundanten Beschreibungen konsistent gehalten werden, was vorwiegend in Handarbeit sichergestellt werden muss. Und nicht zuletzt erfordern die mehrfachen Notationen immer wieder ein Umdenken der Entwickler und die genaue Kenntnis der verschiedenen Beschreibungsmittel.

Vor allem die letzten beiden Punkte stellen ein großes Hindernis für die Einführung neuer Methoden dar und erklären den bislang eher zurückhaltenden Einsatz von Modellen in der Eisenbahndomäne. Der im Rahmen der Fallstudie vorgestellte Ansatz dieser Arbeit zur Verwendung eines einzigen, homogenen Modells für alle relevanten Aspekte des Entwicklungsprozesses (Anforderungen, Architektur, Verhalten, Tests) kann zumindest das Problem des Notationsbruchs entschärfen. Dafür ist allerdings die Neuausrichtung des gesamten Entwicklungsprozesses notwendig.

Chancen	Risiken
• CENELEC-konforme Entwicklung	• Ausweitung der Toollandschaft
• Erhöhung der Produktqualität	• Mehraufwand durch Einführung zusätzlicher Methoden
• Mittel- bis langfristig verringerte Entwicklungsaufwände	• Inkompatibilität zu bestehenden Altsystemen
• Bessere Dokumentation des Expertenwissens	• Notationsbruch innerhalb des Prozesses

Tabelle 2.1: Einige Chancen und Risiken für den Einsatz modellbasierter Techniken in der Entwicklung von Eisenbahnleit- und -sicherungstechnik

Generell ist die Frage nach der Motivation für die Einführung neuer, modellbasierter Verfahren zu stellen. Da die Einführung zunächst mit einem unter Umständen erheblichen Mehraufwand verbunden ist, kann kein kurzfristiger wirtschaftlicher Nutzen als Entscheidungsgrundlage angeführt werden. Eine ökonomische

Rentabilität ist eher mittel- bis langfristig zu erwarten, so dass eine entsprechende Umstellung der Prozesse abgesehen von allen technischen Aspekten vor allem eine strategische Managemententscheidung bedeutet.

Diese Entscheidung wird durch die seit 1999 eingeführten CENELEC-Normen (Comité Européen de Normalisation Electrotechnique, s. Tab. 2.2) DIN EN 50126 ff. [DIN99, DIN01, DIN03] beeinflusst, die für neue Leit- und Sicherungstechnikentwicklungen im Eisenbahnbereich anzuwenden sind. Unter anderem fordert diese Normenreihe den Einsatz von Modellen in der Entwicklung, lässt die Art und den Umfang des Einsatzes aber offen.

Norm	**Inhalt**
EN 61508	Generische Norm, die die funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme zum Gegenstand hat. Unter anderem werden darin Anforderungen an Hard- und Software gestellt und verschiedene Sicherheitsintegritätsstufen (*Safety Integrity Level*, SIL) definiert.
EN 50126	Definiert Anforderungen an Zuverlässigkeit, Verfügbarkeit, Wartbarkeit und Sicherheit (RAMS) von Bahnanwendungen. Ähnlich wie in EN 61508 wird dabei der gesamte Lebenszyklus des Gerätes vom Konzept bis zur Außerbetriebnahme berücksichtigt.
EN 50128	Implementiert den softwarespezifischen Teil der EN 61508 für Bahnanwendungen. Auch hier wird wieder der gesamte Softwarelebenszyklus betrachtet und ein besonderes Augenmerk auf die Softwarespezifikation und SIL-Zuordnung gelegt.
EN 50129	Beschränkt sich auf die elektronischen Systeme für Bahnanwendungen. Dabei stehen Analysen und Ausfallarten verschiedener elektronischer Bauelemente und Systeme im Vordergrund.

Tabelle 2.2: Inhalte der generischen Norm EN 61508 und der aus ihr für die Entwicklung von Bahnanwendungen abgeleiteten Normen der CENELEC-Reihe

Zu den erwarteten Vorteilen des Einsatzes technischer Modelle in der Entwicklung zählen vor allem die Erhöhung der Produktqualität (verbleibende Fehleranzahl, Sicherheit) und die langfristige Verringerung des Entwicklungsaufwands durch geringere Fehler- und Änderungsraten, Wiederverwendung von Modellen oder Modellteilen, erhöhter Wartbarkeit und besserer Dokumentation des vorhandenen „Expertenwissens“.

Aus der Analyse der bestehenden Verbreitung modellbasierter Techniken und der oben genannten Randbedingungen ist für die Hersteller von Leit- und Sicherungstechnik ein unmittelbarer Handlungsbedarf abzuleiten, um auch künftig normgerechte Produktentwicklung zu betreiben. Die im Rahmen der Fallstudie in dieser Arbeit vorgestellten Verfahren können ein Ansatz sein, entsprechende Modellierungsverfahren in künftigen Entwicklungsprozessen zu implementieren.

2.4 Betriebliche Modelle

Neben der zuvor behandelten Modellierung von LST ist auch die formale Beschreibung betrieblicher Prozesse von Bedeutung. Sie stellen, wie in den folgenden Absätzen genauer erläutert wird, die operativen Randbedingungen für den Einsatz von Eisenbahnleit- und -sicherungstechnik dar. Daher verdienen sie bei der Entwicklung entsprechender Geräte besondere Beachtung.

Abstrakt betrachtet sind LST-Systeme lediglich technische Hilfsmittel zur Umsetzung der betrieblichen Regeln. Sollte der vorgesehene oder vorgeschriebene Betrieb mit einem LST-Gerät nicht möglich sein, wurde das falsche System entwickelt[3]. Um in dieser Hinsicht möglichst eindeutige Entwicklungsvorgaben zu machen, ist eine formale Beschreibung der relevanten Betriebsprozesse wünschenswert. Sie stellen somit einen Teil der Kundenanforderungen dar, gegen die ein System im Laufe der Entwicklung getestet werden muss.

[3] Es ist Aufgabe der *Validierung*, dies festzustellen

Grundsätzlich wird der Eisenbahnbetrieb in Deutschland durch die BOStrab (Verordnung über den Bau und Betrieb der Straßenbahnen) und die EBO (Eisenbahnbau- und -betriebsordnung) geregelt, wovon letztere auf dem AEG (Allgemeines Eisenbahngesetz) beruht (Abb. 2.7). AEG und EBO, die nur für Vollbahnen anzuwenden sind, werden auf dem Gebiet der DB durch sogenannte Konzernrichtlinien (KoRil) umgesetzt bzw. ergänzt, die den Betrieb für die Mitarbeiter verbindlich regeln. Vergleichbar mit einem Top-Down-Prozess erhöht sich also die Detailtiefe der Beschreibung vom AEG über die EBO bis zu den KoRil kontinuierlich. Auf Seiten der Straßenbahnen werden die Regelungen der BOStrab noch durch das Personenbeförderungsgesetz (PBerfG) ergänzt, das neben Vorschriften zu Planfeststellungs- und Zulassungenverfahren auch Vorgaben für die Gestaltung von Fahrplänen und Fahrpreisen macht. Im Folgenden soll sich die weitere Betrachtung betrieblicher Regeln und Prozesse allerdings auf Vollbahnen beschränken und Straßenbahnen nicht weiter berücksichtigen. Insbesondere ist bei den Vollbahnen ein Blick auf die Situation in anderen europäischen Ländern notwendig.

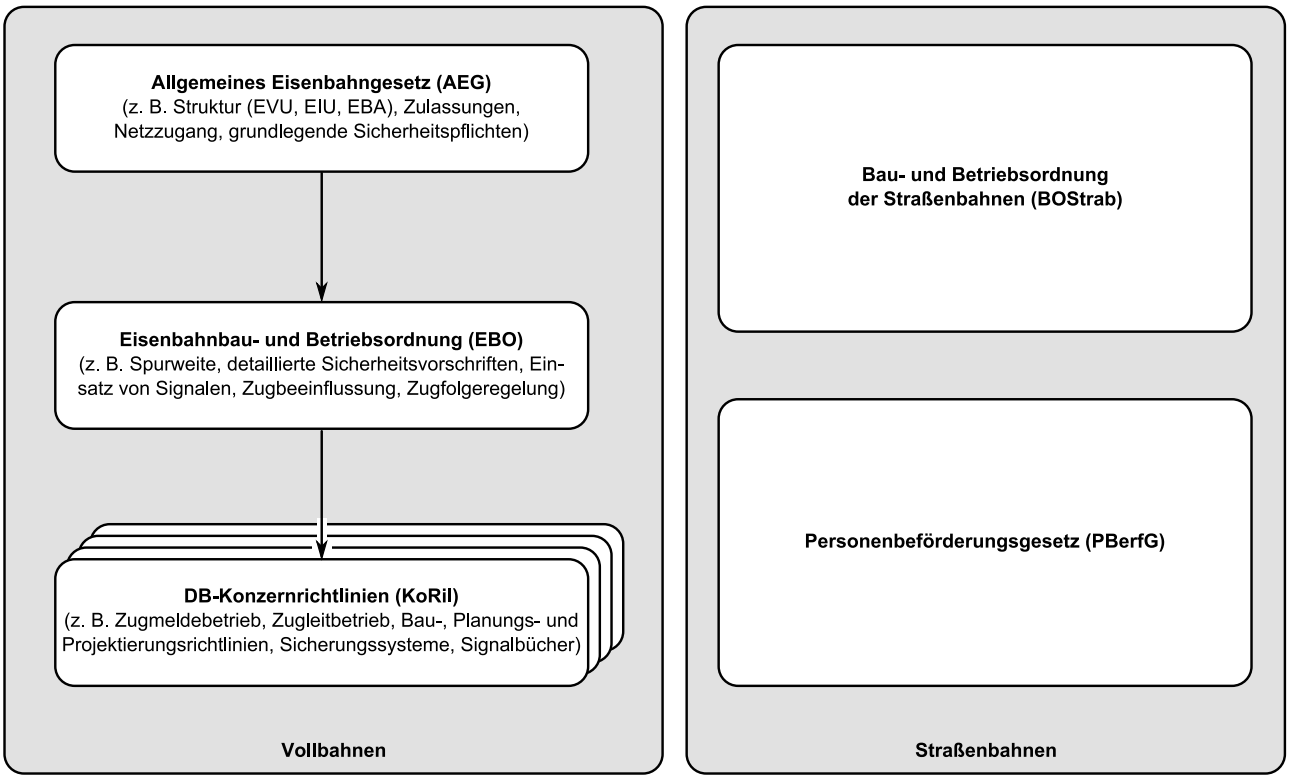

Abbildung 2.7: Maßgebliche Gesetze und Verordnungen für den Betrieb von Voll- und Straßenbahnen in Deutschland

Die starke Heterogenität, die sich in Europa im Bereich der Leit- und Sicherungstechnik findet und die durch die Einführung von ETCS verringert werden soll, spiegelt sich auch auf betrieblicher Ebene wider. Jedes Land besitzt eigene Betriebsregeln, die sich zum Teil erheblich von einander unterscheiden [4].

Durch die in den letzten Jahren vollzogene Öffnung der europäischen und weltweiten Märkte für LST besteht nun zunehmender Bedarf an einer formalen und international verständlichen Darstellung dieser betrieblichen Regelwerke. Sie werden bei der Produktentwicklung vor allem im System- bzw. Feldtest und bei der Validierung eingesetzt (beispielsweise bei Stellwerken, Funkstreckenzentralen oder Fahrzeuggeräten).

Da Systemtest und Validierung gegen Ende des Entwicklungsprozesses stattfinden, kommt der korrekten Interpretation der Betriebsregeln eine um so größere Bedeutung zu, denn die Beseitigung von Fehlern, die in

[4] Beispiel: in Deutschland werden gestörte Bahnübergänge *langsam* geräumt, um angemessen auf eine mögliche Gefährdung reagieren zu können. In Frankreich werden sie dagegen *schnell* geräumt, um die Zeit der möglichen Gefährdung gering zu halten

einer solch späten Entwicklungsphase aufgedeckt werden, ist mit vergleichsweise hohen Kosten verbunden[5]. Folgende Szenarien sind in diesem Zusammenhang denkbar:

- Die Regeln werden zu Beginn der Entwicklung falsch interpretiert, so dass eine Validierung des Systems durch Dritte fehlschlägt.
- Die Regeln werden bei der Entwicklung nicht oder nicht ausreichend berücksichtigt, so dass das System bei der Validierung funktionale Lücken aufweist.
- Die Regeln werden beim Systemtest falsch interpretiert, so dass Tests möglicherweise „falsch positiv“ oder „falsch negativ“ gewertet werden.
- Die Regeln wurden beim Systemtest nicht oder nicht ausreichend berücksichtigt, so dass keine ausreichende Testabdeckung gegeben ist.

Das Risiko einer unvollständigen oder falschen Einbeziehung der betrieblichen Regelwerke wird vor allem dann deutlich, wenn man sich den Umfang der in informaler Prosaform vorliegenden Regelwerke vor Augen hält. Dazu einige Beispiele:

- In Deutschland umfassen alleine die betrieblichen „Kernprozesse“ in Form der KoRil 408 („Züge fahren und rangieren“), KoRil 436 („Zugleitbetrieb“) und KoRil 437 („Signalisierter Zugleitbetrieb“) ca. 2000 Seiten. Hinzu kommen weitere Vorschriften zum Beispiel für die Bedienung und Instandhaltung von Signalanlagen und Stellwerken (KoRil 482) und für die Bedienung der Zugsicherung (PZB 90/INDUSI, KoRil 483). Darüber hinaus hat der Gesetzgeber eine Eisenbahnsignalordnung (ESO) erlassen, die durch die KoRil 301 ergänzt wird (gemeinsames Signalbuch der deutschen Bahn und der ehemaligen Reichsbahn).
- Die Fahrdienstordnung der Schweiz (vergleichbar mit den KoRil der DB) füllt 612 Seiten, hinzu kommen weitere Regelungen, etwa ergänzende Bestimmungen zur Eisenbahnverordnung (EBV), die EBV selber, eine Signalisationsverordnung (SSV) und das grundlegende Eisenbahngesetz (EBG) sowie eine Reihe von Weisungen, Rundschreiben, Anleitungen, Checklisten, Leitfäden und Merkblättern (Quelle: [Bun06]).
- In Schweden gab es bis vor kurzem fünf unterschiedliche Regelwerke für den Betrieb von Stellwerken, die erst 2003 nach Abschluss eines mehrjährigen Projektes harmonisiert wurden. Dazu wurden die Anforderungen an die Stellwerkslogik mittels UML formalisiert erfasst [TE04].

Grundsätzlich besteht auch ein eisenbahnbetriebliches Modell aus einem statischen und einem dynamischen Teil. Dem statischen Teil sind beispielsweise Organisationsstrukturen oder eine Hierarchie der am betrieblichen Prozess beteiligten Entitäten zuzurechnen (beispielsweise einen ICE 3 als Spezialisierung von Zügen mit Mehrfachtraktion, die ihrerseits eine Spezialisierung von Zügen mit elektrischem Antrieb sind, die eine Spezialisierung von Schienenfahrzeugen sind, etc.).

Durch den dynamischen Teil wird die Interaktion zwischen den Prozessbeteiligten repräsentiert. Diese Interaktion entspricht in der Regel dem Austausch von Nachrichten wie beispielsweise Zugmeldungen zwischen Fahrdienstleitern, Fahrbefehle an Triebwagenführer oder Stellkommandos an bewegliche Fahrwegelemente.

Betriebliche Prozesse im Kontext des Gesamtsystems Eisenbahn

Aus Sicht der Leit- und Sicherungstechnik spezifizieren betriebliche Modelle Verhalten auf einer höheren, abstrakteren Ebene. Sie stellen den Kontext dar, in den die LST eingebettet ist und haben daher den Anspruch,

[5] Dieser Aspekt wird im Kapitel über Entwicklungsprozesse noch genauer beleuchtet

eher das „Gesamtsystem Eisenbahn“ wiederzugeben. Offenkundig wird dieser übergeordnete, integrierende Charakter betrieblicher Modelle vor dem Hintergrund, dass sich technische Modelle entweder auf Komponenten der Strecken- *oder* der Fahrzeugseite beschränken, während betriebliche Beschreibungen das Zusammenspiel beider Subsysteme zum Gegenstand haben.

Das UML-Klassendiagramm in Bild 2.8 verdeutlicht die zentrale Rolle betrieblicher Regeln innerhalb des Gesamtsystems Eisenbahn. Der Eisenbahnbetrieb in einem bestimmten Netzbereich, der einem Eisenbahninfrastrukturunternehmen (EIU) zuzuordnen ist, wird von einem oder mehreren Eisenbahnverkehrsunternehmen (EVU) realisiert, die die jeweilige Transportleistung erbringen (Trennung von Netz und Betrieb). Neben dem EIU und dem EVU sind auf Gesamtsystemebene betriebliche Regeln erforderlich, die zum einen auf gesetzlichen Vorgaben (z. B. dem AEG oder der EBO) basieren, die zum anderen aber auch vom EIU vorgegeben werden. Aus diesen Regeln folgen bestimmte Anforderungen an die strecken- oder fahrzeugseitige Technik, die in geeigneter Weise entworfen und parametriert werden muss. Ebenso muss das operative Personal, das direkt am Eisenbahnbetrieb beteiligt ist, regelkundig sein. Daher stehen, wie in Bild 2.8 dargestellt, sowohl Technik als auch Personal in direkter Abhängigkeit von den betrieblichen Regeln.

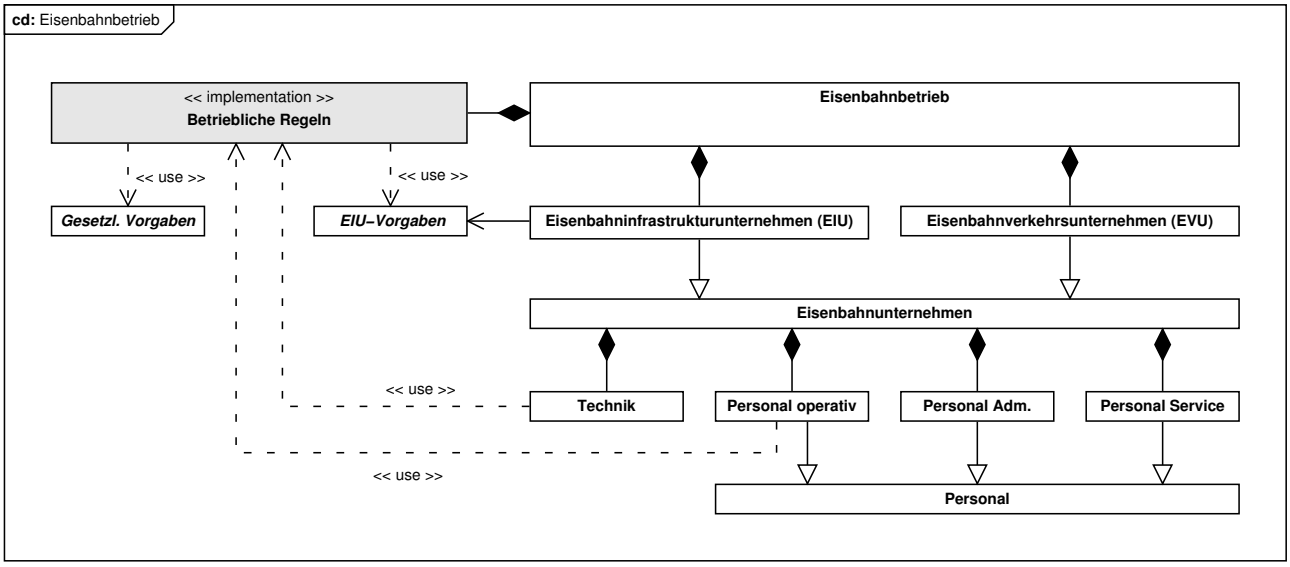

Abbildung 2.8: Zentrale Rolle betrieblicher Regeln im Gesamtsystem Eisenbahn

Vor dem Hintergrund der wechselseitigen Abhängigkeit von Betriebsregeln und Technik werden auch die erheblichen Schwierigkeiten bei der Einführung europaweit *einheitlicher* Zugsicherungs*technik* deutlich, da diese in einem Umfeld extrem *heterogener* Betriebs*regeln* stattfindet. Zur Lösung oder Vermeidung dieser Gegensätze muss der logisch nächste Schritt nach der (so gut wie abgeschlossenen) Standardisierung der Technik daher die Vereinheitlichung betrieblicher Regeln sein [Mey05].

Als Beispiel für die erwähnte enge Verbindung von Technik und Betrieb sei hier die Ausfahrt eines Zuges aus einem Bahnhof gezeigt (Bild 2.9). Zugleich soll verdeutlicht werden, wie ein solcher Vorgang in UML-Notation beschrieben werden kann.

Ein betrieblicher Vorgang kann auf oberster Ebene als *Aktivität* aufgefasst werden, also als eine „durchzuführende in sich abgeschlossene Abfolge von Tätigkeiten, die nach ihrer Durchführung ein im Vorfeld festgelegtes Ergebnis liefern“ [Ver00]. Eine Aktivität beschreibt den Ablauf systeminterner Vorgänge und ist als solches vergleichbar mit einem klassischen Geschäftsprozess. Die Erfassung und Analyse von Geschäftsprozessen wird in zahlreichen Büchern behandelt (beispielsweise [FBH05] und [Stö05]) und soll hier daher nicht weiter vertieft werden. Der Geschäftsprozess darf nicht mit einem Anwendungsfall verwechselt werden, da der Anwendungsfall die Interaktion des Systems mit seiner Umwelt adressiert (*BlackBox*) und die systemin-

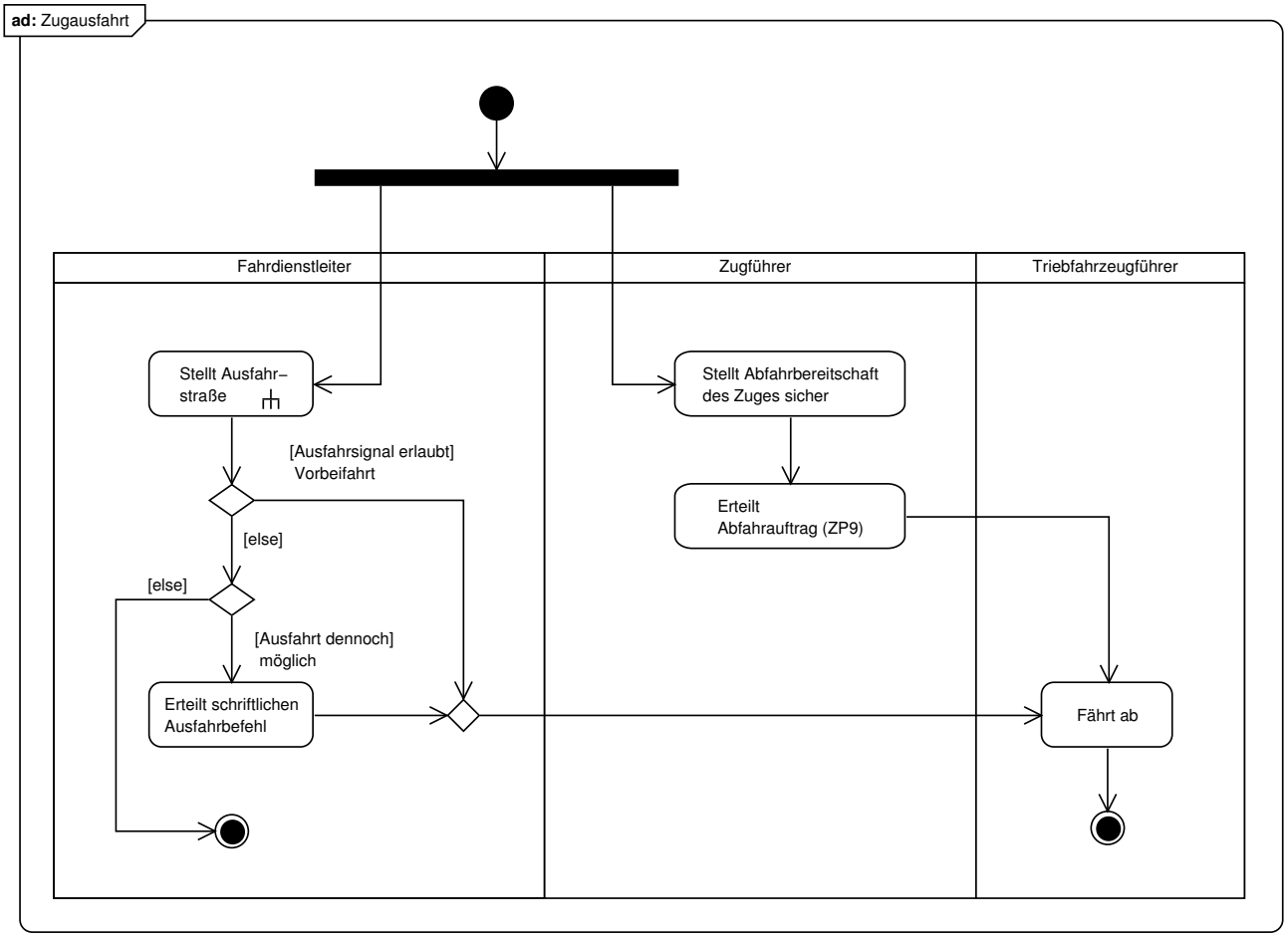

Abbildung 2.9: Betriebsprozess „Ausfahrt eines Zuges“ als UML-Aktivitätsdiagramm

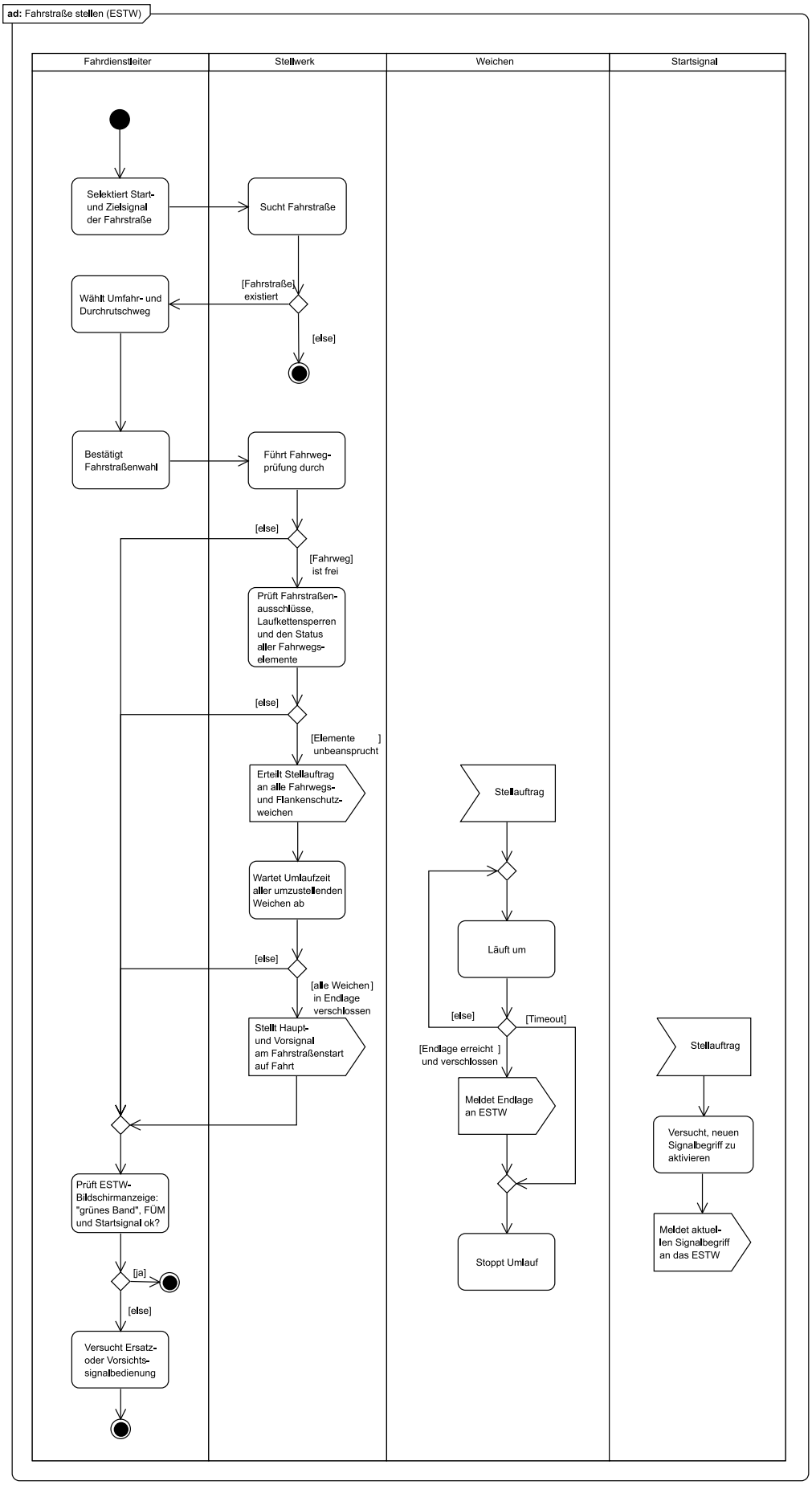

Abbildung 2.10: Betriebsprozess "Einstellen einer Zugfahrstraße mit einem ESTW" als UML-Aktivitätsdiagramm

ternen Vorgänge (*WhiteBox*) unberücksichtigt lässt. Einem Anwendungsfall kann jedoch ein Geschäftsprozess als „Implementierung“ hinterlegt sein.

Bild 2.9 zeigt den oben erwähnten Vorgang „Ausfahrt eines Zuges“ als UML-Aktivitätsdiagramm (zur Semantik von UML-Aktivitätsdiagrammen siehe [Obj05c]). Besonders hervorzuheben ist Möglichkeit zur Darstellung paralleler Vorgänge (hier: „Stellt Ausfahrstraße“ und “Stellt Abfahrbereitschaft des Zuges sicher“), die Erfassung alternativer Abläufe durch Entscheidungsknoten und die mit der Version 2.0 der UML eingeführte Petrinetzsemantik. Der letzte Punkt bedeutet konkret, dass die Aktion “Fährt ab“ erst ausgeführt wird, wenn alle zu ihr führenden Aktivitäten / Aktionen[6] abgeschlossen sind.

Darüber hinaus lassen sich Aktivitätsdiagramme hierarchisieren. Das rechenartige Symbol in der Aktivität „Stellt Ausfahrstraße“ zeigt an, dass die zugehörige Aktivität in einem untergeordneten Diagramm genauer erläutert ist (Bild 2.10). In diesem nachgeordneten Diagramm wird neben der Interaktion zwischen Fahrdienstleiter und Stellwerk auch die oben angesprochene enge Verbindung von Technik und Betrieb ersichtlich, die sich in den dargestellten stellwerksinternen Überprüfungsvorgängen widerspiegelt.

Dabei ist die Grenze zwischen Betriebspersonal und Technik in diesem Kontext nicht unverrückbar. Ein Teil der im Beispiel dem Stellwerk zugeordneten Aktionen (z. B. die Fahrwegsprüfung) werden bei Verwendung mechanischer Stellwerke durch den Fahrdienstleiter ausgeführt. Das zeigt, dass aus betrieblicher Sicht eine bestimmte Menge „generischer“ Vorgänge implementiert sein muss, die aber – je nach technischer Ausstattung – unterschiedlichen Entitäten zugeordnet werden können. Diese Zuordnung kann durch Verwendung grafischer, objektorientierter Beschreibungsmittel besonders deutlich herausgestellt werden (siehe dazu beispielsweise die als „Swimlanes“ bezeichneten Spalten in Bild 2.9 und Bild 2.10). Das Thema der Funktionsallokation bei der Entwicklung von Eisenbahnleit- und -sicherungssystemen wird ausführlich auch von [Mey04] behandelt.

Funktionale Sicht auf betriebliche Prozesse

Eine implementierungs- und technikneutrale Zusammenstellung von Funktionen und Anforderungen für Leit- und Sicherungstechnik wurde auch im Rahmen eines Forschungsprojektes am DLR-Institut für Verkehrssystemtechnik erarbeitet [WE07]. Die vielfältigen Beziehungen und Abhängigkeiten zwischen einzelnen Funktionen, die in herkömmlichen tabellarischen oder baumartigen Darstellungen nicht hinreichend erfasst werden können, wurden dabei durch den Einsatz von UML-Diagrammen unter Anwendung des SysML-Profils [Obj06a] übersichtlich dargestellt. Abbildung 2.11 zeigt ein Beispieldiagramm aus diesem Projekt, in dem die zur Überwachung der Fahrwegelemente notwendigen Aktivitäten als *Block Definition Diagram* der SysML dargestellt sind. Bei dem gezeigten Ausschnitt handelt es sich um einen Teil der mittleren Abstraktionsebene; es ist sowohl noch eine übergeordnete, abstraktere Ebene als auch eine untergeordnete, konkretere Ebene vorhanden.

Bedingt durch den funktionalen, implementierungs- und technikneutralen Ansatz der Studie wurde bei der Erstellung der Diagramme eine BlackBox-Sichtweise eingenommen. Dadurch gleichen die Ergebnisse eher Anwendungsfällen als den in den Aktivitätsdiagrammen (Abb. 2.9 und 2.10) gezeigten „Geschäftsprozessen“. Eingedenk der Vielfalt betrieblicher Regelungen (sprich: Geschäftsprozesse) auf nationaler und internationaler Ebene ist aber eine sorgfältige anwendungsfallorientierte Anforderungsanalyse ein durchaus zielführender Schritt hin zu einer Harmonisierung der Regelungen.

Durch diese Analyse kann erfasst werden, was aus funktionaler Sicht durch die Betriebsregeln realisiert werden soll. Ein wahrscheinliches Ergebnis ist, dass viele Länder grundsätzlich gleiche Funktionen in unterschiedlichen Bestimmungen und Prozessen umgesetzt haben. Zeigt man die funktionale Gleichwertigkeit verschiedener Regeln, ist es vermutlich einfacher, sich auf einheitliche Prozesse zu einigen.

Die Darstellung von Anwendungsfällen wird durch die UML ebenfalls unterstützt. Das dafür vorgesehene

[6] Aktionen sind, anders als Aktivitäten, „atomar“ und können nicht weiter in untergeordnete Vorgänge unterteilt werden.

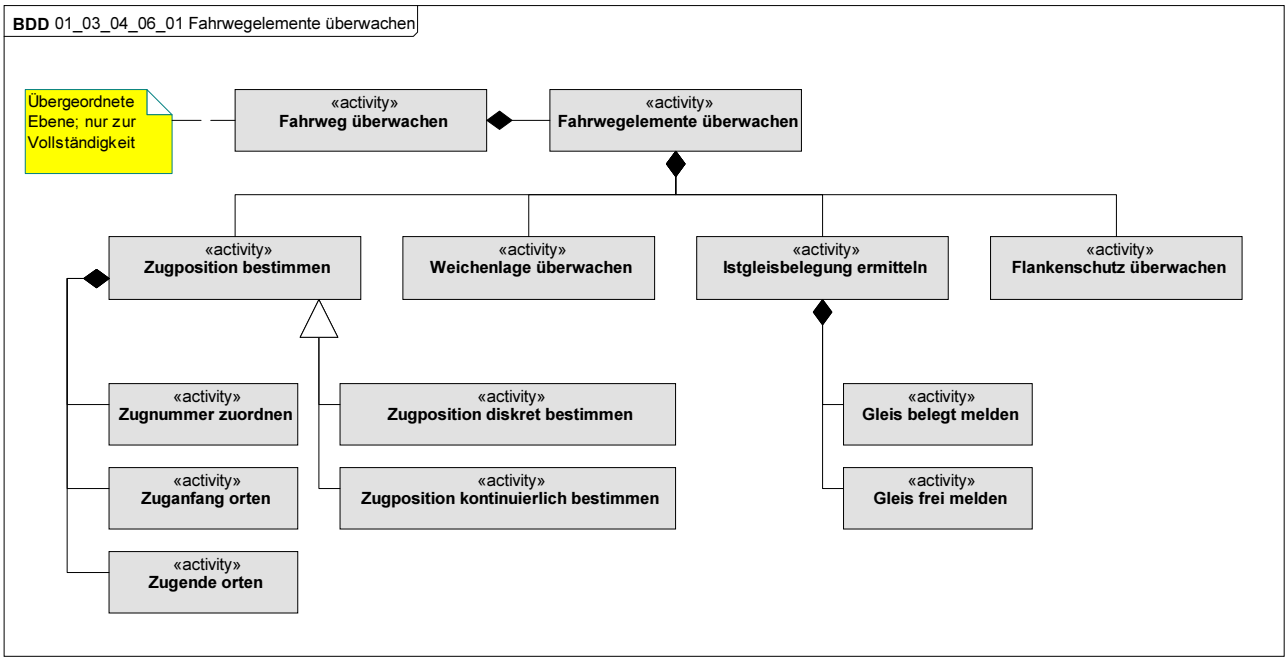

Abbildung 2.11: Ausschnitt aus der Studie zur Darstellung der Beziehungen zwischen funktionalen Anforderungen an Leit- und Sicherungstechnik [WE07]

Anwendungsfalldiagramm erfasst die mit einem Anwendungsfall verknüpften externen Entitäten (sogenannte Akteure), den Namen des Anwendungsfalls sowie Querbeziehungen zwischen Anwendungsfällen. Dieses Diagramm kann allerdings nur eine grobe Übersicht geben. Für eine ausführliche Darstellung von Anwendungsfällen sollte auf natürlichsprachliche Beschreibungen zurückgegriffen werden, die strukturiert auf Basis von Vorlagen beschrieben werden können. Solche Vorlagen erfassen beispielsweise Vor- und Nachbedingungen eines Anwendungsfalls, die beteiligten Akteure, den Auslöser, das Ergebnis und den grundsätzlichen Ablauf des Anwendungsfalls. [Coc01] enthält eine umfangreiche Sammlung von Vorlagen, die an den spezifischen Einsatzzweck angepasst werden können.

Je nach Modellzweck und Sichtweise können mit der UML also sowohl WhiteBox- als auch BlackBox-Beschreibungen betrieblicher Vorgänge erstellt werden. Dafür sind Anwendungsfall-, Aktivitäts- und Sequenzdiagramme am besten geeignet. Die damit mögliche formale Erfassung der Betriebsprozesse erscheint vor dem Hintergrund der Harmonisierung der europäischen Betriebsregeln und der Entwicklung betriebsgerechter, international verwendbarer Leit- und Sicherungstechnik als dringend geboten.

2.5 Modellierung von Tests

Ähnlich wie die beiden bisher betrachteten Themen – technische Systeme und Betriebsprozesse – lassen sich auch Tests durch Modelle beschreiben. Dazu ist in Analogie zu den vorangegangenen Abschnitten auch bei Testmodellen eine Zweiteilung in Statik und Dynamik bzw. Struktur und Verhalten sinnvoll.

Strukturelle Eigenschaften eines Tests beziehen sich in erster Linie auf den Testaufbau, also das Zusammenspiel von Testumgebung, Tester und Prüfling (SUT, „System under Test“). Insbesondere betrifft dies die zu verwendenden Schnittstellen des SUT und die Konfiguration der Testumgebung. Beispielsweise kann eine Testumgebung aus mehreren Komponenten und Modulen bestehen, die je nach Testzweck anders zusammengestellt werden. Für eine lückenlose Dokumentation und einwandfreie Reproduzierbarkeit der Tests muss die Konfiguration der Testumgebung daher ebenfalls erfasst werden.

Das Verhalten eines Tests umfasst vor allem den Testablauf, also die einzelnen Testschritte, die die Stimuli

am oder die Reaktionen vom Prüfling beschreiben. Diese Beschreibung darf dabei nicht nur den Normalablauf abdecken, sondern muss auch alternative Pfade für den Fall vorsehen, dass der Prüfling anders reagiert als erwartet.

Eng mit der Frage alternativer Testabläufe ist auch die Frage des finalen Testurteils verbunden. Im Rahmen eines Testlaufs muss die Entscheidung fallen, ob ein SUT den Test bestanden hat oder nicht. Da die Nichteinhaltung eines Testschrittes nicht automatisch zu einem Nichtbestehen des Prüflings führt (z. B. wenn optionale Eigenschaften geprüft werden), ist dies keineswegs eine einfache binäre Ja-Nein-Entscheidung. Beispielsweise kann ein Test das Ergebnis „bestanden“, „bestanden mit Abweichungen“, nicht bestanden“ oder auch „unentscheidbar“ haben.

Für eine vollständige Beschreibung sind daher die Kriterien, nach denen letztlich die Entscheidung getroffen wird, ebenfalls im Modell wiederzugeben. Typischerweise sind sie als Verhaltensspezifikation einer speziellen Testkomponente – des sogenannten *Arbiters* – implementiert.

Zweck von Testmodellen

Wie alle anderen Modelle besitzen auch Testmodelle eine bestimmte Pragmatik und dienen somit einem bestimmten Zweck. Üblicherweise sind dies:

Dokumentation: Durch die Steigerung der Dokumentationsqualität wird dem Tester die Ausführung der Tests und die Bewertung der Testergebnisse deutlich erleichtert. Gegenüber externen Gutachtern und Validieren stellt eine qualitativ hochwertige Spezifikation ebenfalls einen erheblichen Vorteil dar.

Wiederverwendung: Die Modelle von Testkomponenten und Testabläufen können in Bibliotheken gesammelt und so wiederverwendet werden. Dadurch kann ein Tester in Nachfolgeprojekten erhebliche Zeit bei der Erstellung der Testbeschreibungen sparen. Das Expertenwissen, dass der Entwicklung der Tests zugrunde liegt, kann also in entsprechenden Bibliotheken „konserviert“ werden.

Testautomatisierung: Liegen Tests in einer formalen Notation vor, besteht grundsätzlich die Möglichkeit der automatischen Testausführung und -auswertung. Dadurch kann der hohe Personalaufwand, der noch immer für viele Testkampagnen kennzeichnend ist, reduziert werden. Gleichzeitig steigt durch exakte Reproduzierbarkeit der Testabläufe die Qualität der Ergebnisse, da der „Faktor Mensch“ keine verfälschenden Einflüsse mehr auf den Test hat.

Zur detaillierten Beschreibung von Testmodellen sei auf den Abschnitt 6.3 ab Seite 75 verwiesen. Die Ausführungen im Rahmen dieses Kapitels sollen nur einen ersten Überblick über das Thema darstellen und die Modellierung von Tests im Kontext der Modellierung technischer Systeme und betrieblicher Prozesse einführen, um dadurch die notwendigen Grundlagen für die nachfolgenden Kapitel zu schaffen.

Insbesondere betrifft dies das nächste Kapitel, in dem auf verschiedene Beschreibungsmittel, deren Bestandteile und Eigenschaften eingegangen wird. Der Sprachumfang des Beschreibungsmittels ist wesentlich dafür verantwortlich, inwieweit die in diesem Kapitel erläuterten statischen, dynamischen und nicht-funktionalen Aspekte betrieblicher, technischer und testorientierter Modelle geeignet abgebildet werden können. Somit sind Modell und Beschreibungsmittel eng miteinander verflochten und stehen in gegenseitiger Wechselwirkung, wie die folgenden Seiten zeigen werden.

3 Beschreibungsmittel

Da der Wahl eines geeigneten Beschreibungsmittels bei der Modellierung eine entscheidende Bedeutung zukommt, sollen in diesem Kapitel Eigenschaften, Anforderungen und Einsatzgebiete verschiedener Beschreibungsmittel näher betrachtet werden.

Nach einer allgemeinen Einführung in Beschreibungsmittel werden einige Notationen genauer vorgestellt. In weiteren Abschnitten werden dann ausgewählte Sprachen, die unter anderem Gegenstand von Fallstudien waren, detailliert betrachtet. Ein Vergleich der vorgestellten Beschreibungsmittel bildet den Abschluss dieses Kapitels.

3.1 Definitionen und Begriffe

Ein Beschreibungsmittel dient der Speicherung der im vorangegangenen Kapitel eingeführten Systemeigenschaften (statische, dynamische und nicht-funktionale Komponenten) in grafischer Form. „Grafisch“ kann in diesem Kontext jede Form von Zeichen, Buchstaben oder anderer Symbole umfassen. Somit sind gängige Alphabete (beispielsweise das lateinische) ebenfalls den Beschreibungsmitteln zuzurechnen.

Ein Beschreibungsmittel muss in drei Punkten wohl definiert sein. Dies sind die Menge der verfügbaren *Symbole*, die *Syntax* und die *Semantik* (vgl. auch Abb. 3.1). Folgende Eigenschaften kennzeichnen diese Begriffe:

Symbole: Die Symbole stellen die Menge der darstellbaren Zeichen einer Sprache dar (Alphabet). Durch Kombination (z. B. Aneinanderreihung) der Symbole können die Wörter einer Sprache erzeugt werden. So ist beispielsweise die Menge der deutschen Wörter das Alphabet bzw. die Symbolmenge der Sprache „Deutsch“. Die Kombination dieser Symbole, also die deutschen Sätze, stellen die Wörter der Sprache „Deutsch“ dar[1].

Syntax: Durch die Syntax werden die gültigen Zeichenkombinationen eines Beschreibungsmittels erfasst. Im Sinne des obigen Beispiels der Sprache „Deutsch“ definiert die Syntax, welche Kombination deutscher Wörter einen gültigen deutschen Satz ergeben. Ein Wort einer Sprache entspricht einer Aussage oder einem Ausdruck, beispielsweise einer Zeile einer Programmiersprache. Die Syntax weist einer Sprache durch die Spezifikation der gültigen Symbolkombinationen eine innere Struktur zu. Die verfügbaren Symbole und die zugehörige Syntax werden häufig unter dem Begriff *Semiotik* zusammengefasst.

Semantik: Die Semantik ordnet den Symbolen und ihren Kombinationen eine Bedeutung zu. Sie ist damit wesentlich für die Mächtigkeit einer Sprache verantwortlich, da sich nur die Eigenschaften und Aspekte eines Systems, die durch die Semantik des gewählten Beschreibungsmittels abgedeckt sind, auch tatsächlich erfassen lassen.

Je nach Formalitätsgrad einer Sprache sind diese drei Aspekte unterschiedlich scharf zu spezifizieren. Die höchsten Ansprüche stellen formale Beschreibungsmittel, die für alle drei Sprachkomponenten eine mathematische Basis fordern, wie die folgenden Absätze zeigen. Im Anschluss daran wird auf die Beschreibung semiformaler Notationen eingegangen.

[1] Die umgangssprachliche Bedeutungen von „Zeichen“ und „Wort“ sind in diesem Beispiel gegenüber der Definition einer (formalen) Sprache im Sinne der Informatik verschoben: Zeichen und Wörter der Informatik sind Wörter und Sätze im umgangssprachlichen Gebrauch.

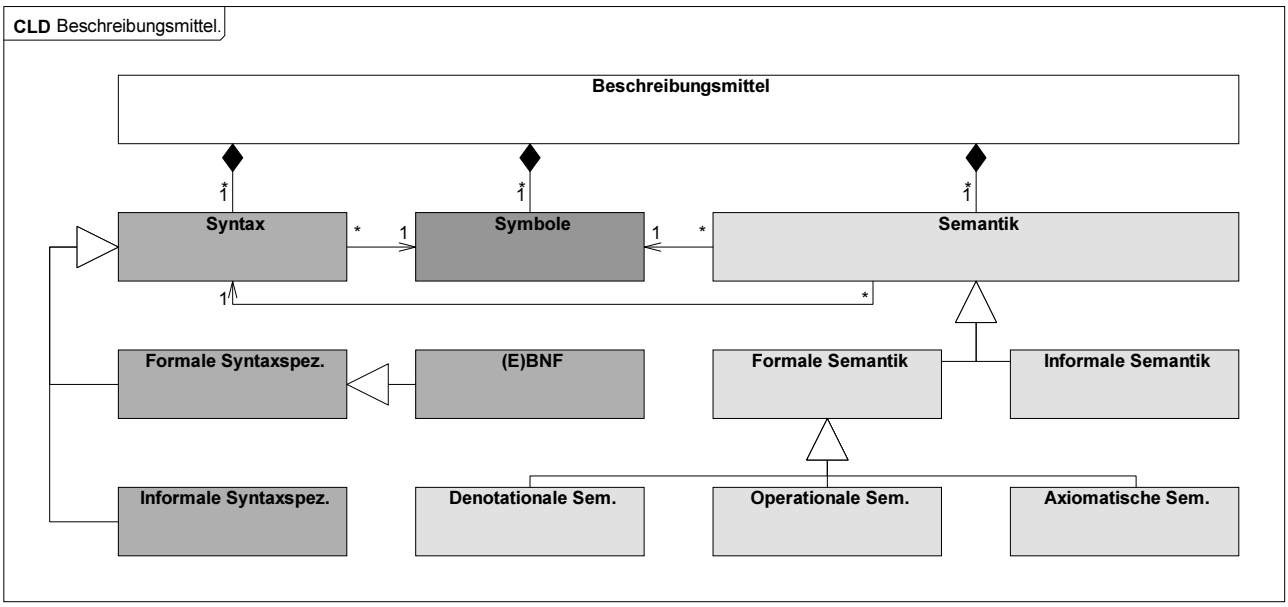

Abbildung 3.1: Die Komponenten Symbole, Syntax und Semantik eines Beschreibungsmittels, ihre gegenseitigen Abhängigkeiten und Spezialisierungen

Spezifikation formaler Beschreibungsmittel

Vor allem bei künstlich geschaffenen Beschreibungsmitteln wie Programmier- oder Modellierungssprachen besteht die Notwendigkeit, das „Beschreibungsmittel zu beschreiben". Selbstverständlich ist dafür wieder ein Beschreibungsmittel notwendig, das irgendwo beschrieben worden sein muss usw.

Bei der „Beschreibung von Beschreibungsmitteln" gelangt man also auf eine Metaebene, auf der unweigerlich auf bekannte (Umgangs-)Sprachen – also Prosa – oder auf Mathematik zurückgegriffen werden muss[2].

Die Definition der Symbole ist vergleichsweise einfach, da es sich im Wesentlichen um die Aufzählung einer Menge handelt. Auch wenn diese Mengendefinition prinzipiell mathematisch leicht gehandhabt werden kann, setzt bereits dieser Schritt vielen exakten Sprachdefinitionen praktische Grenzen. Als Beispiel sei hier nur die Aufzählung aller deutschen Wörter genannt, die die Symbolmenge der Sprache „Deutsch" bilden.

Für die Beschreibung der Syntax stehen verschiedene Methoden zur Verfügung, von denen die bekannteste die *Backus-Naur-Form* oder auch *Backus-Normalform* ist, die mit BNF abgekürzt wird (siehe auch Abb 3.1). Sie wurde erstmals zur formalen Spezifikation der Programmiersprache Algol-60 eingesetzt [Bac59] und hat sich seitdem weiter verbreitet. Auch moderne Sprachen wie Pascal oder die weiter unten noch intensiver diskutierte *Testing and Test Control Notation, Version 3* (TTCN-3) basieren auf einer BNF-Grammatik.

Die BNF verwendet nur wenige Konstrukte und ist daher leicht erlernbar. Sie stellt Regeln auf, nach denen bestimmte Platzhalter (sogenannte Nichtterminalsymbole) durch normale Zeichen (Terminalsysmbole) und (andere) Nichtterminale ersetzt werden dürfen. Abbildung 3.2 zeigt exemplarisch die Definition eines deutschen Autokennzeichens in BNF-Notation. Darin bedeutet `<Bezeichner>` ein Nichtterminalsymbol, `::=` eine Zuweisung, `[]` einen optionalen Teil und `|` eine Alternative.

Durch die Beschränkung der linken Seite einer Syntaxregel auf genau ein alleinstehendes Nichtterminalsymbol erlaubt die BNF nur die Definition einer Untermenge möglicher Syntaktiken (sogenannte *kontextfreie* Sprachen). Andere Grammatiken lassen komplexere Regeln zu und ermöglichen so weiterführende Sprachkonstrukte. Eine Klassifikation der verschiedenen Sprachtypen wird durch die *Chomsky-Hierarchie* vorgenommen, auf die beispielsweise in einem Lehrbuch von Hedtstück genauer eingegangen wird [Hed03].

Die Syntaxregeln einer Sprache bzw. eines Beschreibungsmittels geben, wie bereits erwähnt, nur Aufschluss

[2] Es gibt einige Sprachen, die „in sich selber" definiert sind. Beispielsweise wird für die Spezifikation der UML-Superstruktur UML verwendet, vgl. Kapitel 4.

```
<unterscheidungszeichen> ::= A | AA | AB | ABG | ABI | ... | Z | ZI | ZW
<buchstabe> ::= A | B | C | ... | Z
<ziffer> ::= 0 | 1 | 2 | ... | 9
<erkennungsnummer1> = <buchstabe> <buchstabe> Leerzeichen <ziffer> [<ziffer>] [<ziffer>]
<erkennungsnummer2> = <buchstabe> Leerzeichen <ziffer> [<ziffer>] [<ziffer>] [<ziffer>]
<erkennungsnummer> = <erkennungsnummer1> | <erkennungsnummer2>
<kennzeichen> = <unterscheidungszeichen> - <erkennungsnummer>
```

Abbildung 3.2: Definition eines deutschen KFZ-Kennzeichens durch Verwendung der BNF (ohne Sonderkennzeichen und ohne unzulässige Erkennungsnummern)

über die erlaubten Symbolkombinationen (z. B. Zeichenketten). Ihre Bedeutung – die Semantik – ist dadurch aber noch nicht definiert. Ein einfaches Beispiel soll diesen Sachverhalt verdeutlichen: die *Zeichenketten* „456“, „0456“, „00456“ bedeuten alle die *Zahl* 456.

Für ein formales Beschreibungsmittel ist somit nicht nur die eindeutige, vollständige und widerspruchsfreie mathematische Definition der Syntax notwendig, sondern auch eine ebenso geartete Semantikspezifikation. Die am weitesten verbreiteten Varianten formaler Semantiken sind (Abb. 3.1):

Denotationale Semantik: die Bedeutung von Ausdrücken wird durch mathematische Funktionen beschrieben, die auf den Zustandsraum des Systems wirken.

Operationale Semantik: durch Definition eines vereinfachten Interpreters, der sogenannten *abstrakten Maschine*, werden die Auswirkungen von Ausdrücken der betrachteten Sprache beschrieben.

Axiomatische Semantik: die Inhalte und Aussagen von Ausdrücken werden mittels logischer Schlussregeln ermittelt, die auf ein zuvor definiertes Axiomensystem (der sogenannte Kalkül) angewendet werden.

Erst durch eine abgeschlossene, formale Semantik wird ein Beschreibungsmittel ausführbar und ermöglicht dadurch eine Verhaltenssimulation der hinterlegten Modelle und Systeme am Computer. Vielen Sprachen, beispielsweise der UML, fehlt diese vollständige mathematische Basis. Sie werden daher als semiformale Sprachen bezeichnet.

Spezifikation semiformaler Beschreibungsmittel

Bei dieser Klasse von Sprachen sind die Semantikdefinition und häufig auch Teile der Syntaxspezifikation als Prosatext abgefasst. Durch Verwendung möglichst präziser Formulierungen soll größtmögliche Eindeutigkeit erzielt werden, allerdings liegt es in der Natur der Prosabeschreibung, dass Mehrdeutigkeiten, Widersprüche oder Auslassungen nie ausgeschlossen werden können. Wichtige Elemente einer solchen Sprachbeschreibung sind

- die Angabe, welche Symbole auf welche Weise miteinander kombiniert werden dürfen
- Angaben zu Einschränkungen (engl. *constraints*) hinsichtlich der Kombinierbarkeit von Symbolen oder ihrer Verwendbarkeit innerhalb eines bestimmten Kontexts
- die Semantik des Symbols bzw. der Symbolkombination

Häufig werden diese oft sehr abstrakt formulierten Beschreibungen[3] noch durch Beispiele ergänzt, um ihren Sinn besser zu verdeutlichen.

Exemplarisch für die Definition eines semiformalen Beschreibungsmittels sei die Spezifikation der Unified Modeling Language genannt, auf die im folgenden Kapitel noch genauer eingegangen wird. Bemerkenswert ist dabei die Tatsache, dass die UML teilweise „in sich selbst“ beschrieben ist: auf der Metaebene der „Beschreibung des Beschreibungsmittels UML“ wird wiederum UML eingesetzt, und zusätzlicher Text trägt die restliche, fehlende Information.

[3] Manche Sprachbeschreibung erinnert stark an Gesetzes- oder Vertragstexte.

Bei der Verwendung semiformaler Beschreibungsmittel muss man zwar auf die Ausführbarkeit der Modelle verzichten, allerdings wird dem Anwender dadurch die Freiheit eingeräumt, den Detaillierungsgrad seines Modells in weiten Teilen selbst zu bestimmen. Diese Freiheit kann sehr nützlich sein, wenn ein Modell im Laufe eines Top-Down-Entwicklungsprozesses zunehmend konkretisiert wird, jedoch birgt sie ähnlich wie eine informale Beschreibung die Gefahr von Fehlinterpretationen des Modellinhalts.

3.2 Für Systembeschreibungen notwendiger Sprachumfang

Nach [JS01], [Sch02] und [Mey04] sind die Aspekte *Struktur*, *Verhalten*, *Zustand* und *Funktion* kennzeichnend für ein System. Diese vier Eigenschaften sind eine alternative Darstellung der im vorangegangenen Kapitel eingeführten Unterteilung in statische und dynamische Komponenten einer Systembeschreibung.

Grundlage für die Erfassung dieser Eigenschaften in einem Modell ist die Fähigkeit des zugrunde liegenden Beschreibungsmittels, diese Eigenschaften auch ausdrücken zu können. So sind beispielsweise Petrinetze hervorragend geeignet, um Nebenläufigkeiten und konkurrierende Zugriffe auf eine gemeinsame Ressource nachzubilden. Für eine Darstellung der Systemstruktur und -architektur sind sie allerdings kaum anwendbar.

Mit Blick auf die statischen Systemeigenschaften muss ein Beschreibungsmittel vor allem Strukturelemente (also Komponenten, Subsysteme), Schnittstellen und die Abgrenzung des Systems zur Außenwelt darstellen können. Für diesen Zweck werden üblicherweise alle Arten von Blockdiagrammen verwendet. Häufig liegt ihnen keine feste Syntax und Semantik zugrunde, sondern sie werden vom Ersteller „frei“ entworfen und entweder mit einer Legende oder erläuterndem Text versehen. Eine andere Möglichkeit ist die Verwendung von UML-Klassen- und -Strukturdiagrammen, die zwar komplexer sind, aber dafür über eine (semi-)formale Grundlage verfügen.

Hinsichtlich der dynamischen Systemaspekte müssen vor allem kausale und temporale Zusammenhänge ausgedrückt werden können. Kausale Eigenschaften eines Systems beschreiben die Verknüpfung zwischen (externen) Ereignissen und den damit verbundenen Änderungen des Systemzustandes. Wichtige zeitliche Zusammenhänge, die von einem Beschreibungsmittel erfasst werden müssen, sind:

Antwortzeiten, Übertragungszeiten, Reaktionszeiten, etc: Allgemein ausgedrückt handelt es sich um Zeitvorgaben, die für die Abarbeitung eines Algorithmus, die Verarbeitung eines Events/Interrupts oder die Übertragung einer bestimmten Datenmenge gelten. Die Zeiten werden dabei entweder als Dauer (Zeitspanne) oder Frist (Zeitpunkte) angegeben. Solche Zeitvorgaben sind vor allem bei Echtzeitsystemen von Relevanz, für die die garantierte, rechtzeitige Erledigung bestimmter Aufgaben charakteristisch ist [Sch92]. Die Zeitanforderungen spielen besonders in der Systemspezifikation und in der Implementierung eine Rolle. In der Systemanalyse sind sie dagegen von untergeordneter Bedeutung, da darin häufig von optimalen Bedingungen ausgegangen wird (idealer Prozessor, unendlicher Speicher etc.).

Nebenläufigkeit: Die unabhängige, parallele Bearbeitung unterschiedlicher Aufgaben muss darstellbar sein.

Synchronisation, konkurrierender Zugriff, Prioritäten: Aus der nebenläufigen Bearbeitung verschiedener Aufgaben folgt die Notwendigkeit, eigenständige Handlungsstränge zu synchronisieren bzw. zu vereinen oder neue Handlungsstränge abzuzweigen. Dies muss durch ein Beschreibungsmittel erfasst werden können. Ebenso folgt aus der Nebenläufigkeit die grundsätzliche Möglichkeit des konkurrierenden Zugriffs auf bestimmte Ressourcen. Dieser Konflikt muss in der Beschreibung erkennbar und eine Lösung aufgezeigt sein. Im allgemeineren Fall eines „Konflikts“ kann es vorkommen, dass innerhalb eines Systems zwei oder mehr Zustandsübergänge gleichzeitig ausgeführt werden können. Das Beschreibungsmittel muss für einen solchen Fall entweder Prioritäten vorgeben oder die Definition von Prioritäten erlauben.

Antwortzeiten lassen sich besonders gut durch Sequenzdiagramme darstellen, während für Nebenläufigkeiten und Synchronisationen vor allem Petrinetze und Aktivitätsdiagramme geeignet sind. Für alle diese genannten Diagrammtypen gibt es isolierte Darstellungsformen, aber auch Varianten von UML-Diagrammen[4]. Zur Übersicht sind die statischen und dynamischen Anteile am Sprachumfang eines Beschreibungsmittels in Tabelle 3.1 zusammengefasst. Darüber hinaus führt die nachfolgende Vorstellung einiger Beschreibungsmittel die Stärken und Schwächen der jeweiligen Sprachen hinsichtlich der hier genannten Aspekte genauer aus.

Systemaspekt	**Mögliche grafische Darstellung**
Subsysteme (Struktur, Dekomposition), Schnittstellen, Abgrenzung zur Außenwelt	Allgemeines Blockdiagramm, UML: vor allem Klassen-, Kompositionsstruktur- und Anwendungsfalldiagramme
Kausale Dynamik (Zustandswechsel, Reaktion auf externe Ereignisse)	Zustandsdiagramme, Petrinetze
Temporale Dynamik (Zeitliche Aspekte des Systemverhaltens oder von Zustandsänderungen)	Zeitbewertete Petrinetze, Aktivitäts-, Sequenz- und (eingeschränkt) Zustandsdiagramme

Tabelle 3.1: Systemeigenschaften, die zur vollständigen Systembeschreibung durch den Sprachumfang des Beschreibungsmittels abgedeckt werden müssen (vgl. auch Systemaxiome nach [Sch92, ESS94]).

3.3 Vorstellung einiger Beschreibungsmittel

Für die Analyse und die Realisierung von Eisenbahnleit- und -sicherungstechnik ist eine Vielzahl von Beschreibungsmittel im Einsatz, wobei eine weite Verbreitung nur der natürlichsprachlichen Beschreibung und den gängigen Programmiersprachen wie C / C++ und Pascal vorbehalten ist.

Meyer zu Hörste hat etwa ein Dutzend Tagungen, Symposien und Workshops hinsichtlich der vorgestellten formalen Methoden und der eingesetzten Beschreibungsmittel ausgewertet und in einer umfangreichen Tabelle zusammengestellt [Mey04]. Für eine möglichst breite Übersicht sei daher auf die genannte Arbeit verwiesen. Darüber hinaus führt Bowen auf einer Website [Bowb] knapp einhundert formale Notationen samt Referenzen zu weiterführenden Informationen auf; viele der dort genannten Sprachen besitzen allerdings faktisch keine Relevanz.

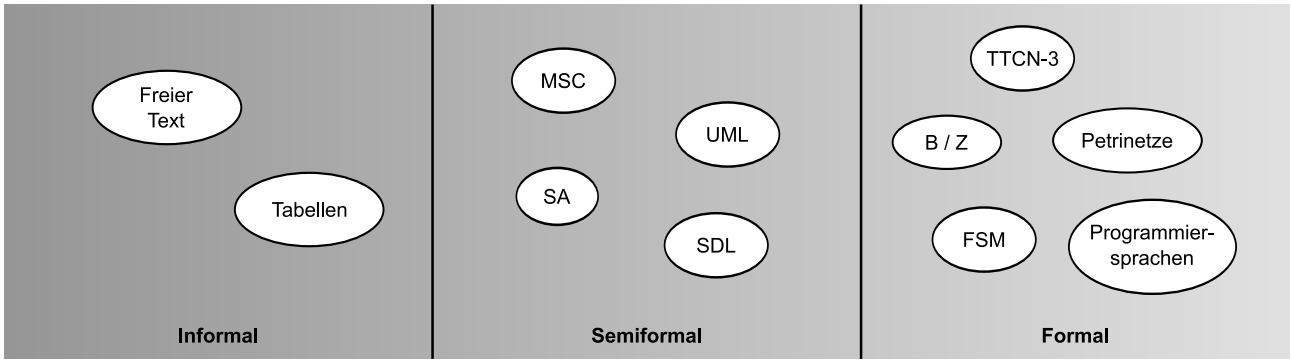

Abbildung 3.3: Überblick über die in dieser Arbeit angesprochenen Beschreibungsmittel und den Grad ihrer Formalität (ohne Sortierung innerhalb der einzelnen Gruppen)

An dieser Stelle seien kurz einige Sprachen erwähnt, die eine gewisse Bedeutung in der Eisenbahnleit- und -sicherungstechnik haben, für die Fallstudie im hinteren Teil dieser Arbeit aber nicht von Bedeutung sind.

[4] Es gibt keine direkten UML-Petrinetze, allerdings wurde mit der UML 2.0 ein Teil der Petrinetzsemantik in die Aktivitätsdiagramme übernommen.

Die für die Fallstudie relevanten Sprachen (UML, TTCN-3) werden später in eigenen Abschnitten bzw. Kapiteln detaillierter erläutert. Eine Übersicht der hier bzw. in Folgekapiteln behandelten Sprachen zeigt auch Abbildung 3.3, die die verschiedenen Beschreibungsmitteln nach Formalitätsgrad ordnet.

Natürlichsprachlicher Text: Klartext bzw. Prosatext ist noch immer das am weitesten verbreitete Beschreibungsmittel für Anforderungen, Schnittstellenbeschreibungen etc. In vielen Entwicklungsprozessen und -projekten wird – abgesehen von der Softwareumsetzung – ausschließlich Prosatext eingesetzt. Eine solche Beschreibung ist informal, (fehl-)interpretierbar (Sprachbarrieren in internationalen Projekten!) und in aller Regel unvollständig. Beispielsweise geht Rupp intensiv auf die Verwendung natürlichsprachlicher Anforderungsbeschreibungen in Entwicklungsprojekten ein [RS04].

Programmiersprachen: Der Einsatz von Programmiersprachen steht üblicherweise am Ende eines Entwicklungsprozesses. Programmiersprachen zählen zu den formalen Sprachen werden je nach zugrunde liegendem Programmierparadigma in imperative und deklarative Sprachen unterteilt.

Zu den imperativen Sprachen zählen beispielsweise C/C++, Pascal und Basic, während die funktionale Sprache Haskell und die logische Sprache Prolog typische Vertreter deklarativer Sprachen sind. Die mit Abstand weiteste Verbreitung haben imperative Sprachen; von wenigen Spezialanwendungen abgesehen sind logische und funktionale Sprachen vor allem für die Forschung interessant und werden daher häufig auch als „Akademikersprachen“ bezeichnet.

B und Z: B und Z sind formale Sprachen, die auf die Arbeiten von Jean-Raymond Abrial zurückgehen. Während Z ausschließlich zur Systemspezifikation dient, lassen sich in B Algorithmen formulieren, die anschließend in ausführbare Sprachen wie Java oder C transformiert werden können. Dafür greift B auf die Hilfe einer abstrakten Maschine zurück, auf der die Programmabläufe und Algorithmen spezifiziert werden (AMN – Abstract Machine Notation).

B und Z ermöglichen eine formale Verifikation des Codes, weshalb sie für den Einsatz in sicherheitsrelevanten Applikationen prädestiniert sind. Ein Beispiel dafür sind die von Siemens in Paris und New York errichteten fahrerlosen U-Bahnen bzw. Shuttles „Meteor“ und „VAL“ , bei denen B zum Einsatz kam (s. auch [ED07, JK06]).

Während Z im Jahre 2002 in eine offizielle ISO-Norm überführt wurde [ISO02], liegt für B nur ein De-Facto-Standard in Form einer Veröffentlichung von Abrial vor [Abr96]. Weitere Informationen und Referenzen zu Z und B sind auch im Internet verfügbar [Bowc, Bowa].

Petrinetze: Petrinetze gehen auf die Arbeit von Carl Adam Petri [Pet62] zurück und bestehen aus bipartiten Graphen zur Darstellung ereignisdiskreter Vorgänge. Sie eignen sich besonders zur Darstellung von Nebenläufigkeiten und wurden im Laufe der Zeit in verschiedene Richtungen weiterentwickelt (z. B. farbige Petrinetze, zeiterweiterte Petrinetze). Je nach Ausprägung sind sie einer vollständigen mathematischen Analyse zugänglich, die unter anderem Aussagen über Verklemmungen (Deadlocks) und die Lebendigkeit des Netzes erlaubt. Der Einsatz von Petrinetzen in der Ingenieurs- und Automatisierungstechnik wird intensiv in Büchern von Abel und Schnieder behandelt [Abe90, SA92] und ist auch Gegenstand der aktuellen Forschung (s. beispielsweise Beiträge in [ST07]).

SDL: Die „Specification and Description Language“ (SDL) ist eine grafische Beschreibungssprache für Kommunikationssysteme oder – allgemeiner – für Systeme aus kommunizierenden Prozessen. Zu ihren Kernkomponenten gehören erweiterte Zustandsmaschinen, die über Signale miteinander kommunizieren. In neueren Versionen unterstützt SDL auch eine objektorientierte Systemmodellierung. Die Sprache wurde von der „International Telecommunication Union“ (ITU) genormt [ITU02] und wird traditionell vor al-

lem in der Telekommunikationsdomäne eingesetzt. Vereinzelt ist aber auch der Einsatz im Bereich der Eisenbahnleit- und -sicherungstechnik zu beobachten, wie beispielsweise Berglehner zeigt [Ber00].

MSC: Die „Message Sequence Charts“ (MSC) wurden ebenfalls von der ITU spezifiziert [ITU04] und entstammen ähnlich wie SDL der Telekommunikationsbranche. In ihnen wird der Austausch von Nachrichten zwischen verschiedenen Entitäten oder Systemkomponenten über der Zeitachse grafisch aufgetragen. Sie ermöglichen die Abbildung verschiedener Nachrichtenarten (synchron / asynchron), zeitlicher Randbedingungen (Timer, Timeouts) und bedingter Kommunikation (via *Guard Conditions*). Mit MSCs lassen sich vor allem Kommunikationsprotokolle oder andere sequentielle Abläufe – wie beispielsweise Testfälle – effektiv beschreiben (siehe auch [ENL04, SG03]).

Zustandsautomaten: Die Automatentheorie stellt eines der Kerngebiete der theoretischen Informatik dar. Sie behandelt die Repräsentation von Systemen in Form von *Zuständen* und den sie verbindenden *Transitionen*. Die bildliche Darstellung erfolgt dabei in der Regel als gerichteter Graph aus Kreisen (Zuständen) und Kanten (Transitionen). Zum Schalten einer Transition müssen bestimmte Vorbedingungen erfüllt sein, die aus externen Stimuli oder aus internen Ereignissen (zum Beispiel Abschluss einer Rechenoperation) bestehen. Die theoretische Informatik kennt eine Vielzahl von Automatenvarianten, die sich vor allem in den folgenden Kriterien unterscheiden:

- Endlichkeit der Zustandsmenge: die Menge der Systemzustände kann unendlich oder endlich sein. Bei der Modellierung realer Systeme findet man in der Regel Automaten mit endlich vielen Zuständen vor; diese Automaten werden dann als *FSM* (engl. für *Finite State Machine*) bezeichnet[5]. Auch wenn reale Systeme einen endlichen Zustandsraum haben, kann dessen Größe schnell erhebliche Ausmaße annehmen. So hat ein Programm mit einer einzigen 8-Byte-Gleitkommazahl (*double precision* in C++) bereits einen Zustandsraum von 2^{64} Zuständen. Diese Systemeigenschaft wird als *Zustandsraumexplosion* bezeichnet und ist das zentrale Problem bei der Modellierung und Analyse von FSM. Das Beispiel zeigt auch, dass ein vollständiger Systemtest, also die Überprüfung *aller* Systemzustände, praktisch nicht machbar ist. Abhilfe kann hier die Partitionierung des Zustandsraumes in sogenannte *Äquivalenzklassen* schaffen [MSBT04, LW00].
- Terminiertheit: Ein Automat kann terminieren, das heißt er erreicht einen Zustand, der keinen Nachfolgezustand mehr hat (entweder weil es keine ausgehenden Transitionen gibt oder weil für keine ausgehende Transition die Vorbedingung erfüllt ist). Beispielsweise sollte ein Automat, der eine numerische Berechnung repräsentiert, immer terminieren, weil sonst niemals ein Ergebnis zurückgeliefert wird. Andererseits ist bei Automaten, die Betriebs- oder Sicherungssysteme repräsentieren, Terminiertheit unerwünscht, weil dies faktisch einem Absturz des Systems gleichkäme. Bei der Verifikation solcher Systeme werden daher sogenannte *Lebendigkeits-* und *Sicherheitsbedingungen* überprüft: erstere bestimmen, dass ein erwünschter Zustand (beispielsweise die Hauptschleife) immer erreicht wird; letztere definieren, dass verbotene Zustände wie Verklemmungen (engl. *Deadlocks*) niemals erreicht werden (zu *liveness properties* und *safety properties* siehe auch [CGP00]). Ähnliche Analysen – beispielsweise der Erreichbarkeitsgraph – sind auch für Petrinetze möglich, siehe dazu u. a. [Jen97].

  Als Beispiel für solche Analysen wurde im Rahmen dieser Arbeit exemplarisch ein SiFa-System (Sicherheitsfahrschaltung, auch als Totmannschaltung bekannt) mit Hilfe endlicher Automaten nachgebildet (Abb. 3.4). Der formale Nachweis bestimmter Systemeigenschaften – beispielsweise das Nichtüberschreiten der Höchstgeschwindigkeit oder die Verklemmungsfreiheit des Systems – wurde dann mit Hilfe des Model Checkers „UPPAAL“[6] geführt. Dazu wurden die in Abbildung 3.5

[5] In der deutschsprachigen Literatur findet sich mitunter auch die Abkürzung *EA* (Endlicher Automat), allerdings ist FSM gebräuchlicher.
[6] http://www.uppaal.com

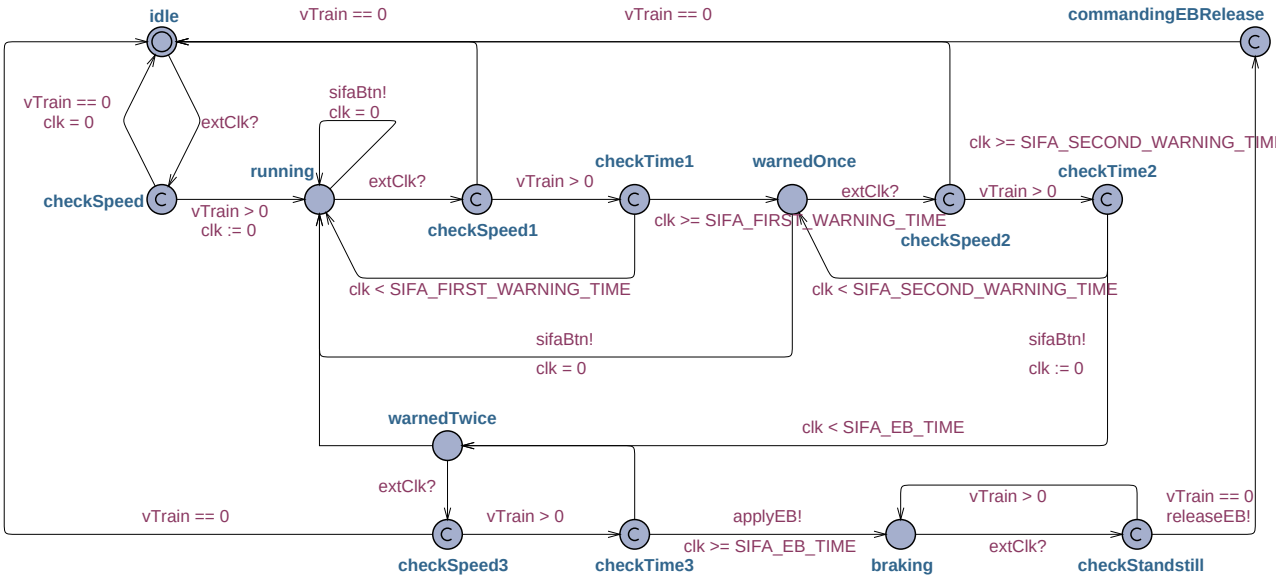

Abbildung 3.4: Endlicher Automat einer im Rahmen dieser Arbeit modellierten und mit dem Model Checker „UPPALL“ verifizierten SiFa (Ausschnitt)

gezeigten temporallogischen Aussagen durch den Model Checker überprüft und ggf. eine Sequenz von Eingangswerten als Gegenbeispiel konstruiert, die zur Verletzung der Aussagen führt.

- Determiniertheit: Es ist möglich, dass in einem Zustand mehrere ausgehende Transitionen vorhanden sind, deren Vorbedingungen gleichzeitig erfüllt sind. In einem solchen Fall ist der Nachfolgezustand nicht eindeutig festgelegt und der Automat daher nicht-deterministisch. Ist der Folgezustand dagegen durch entsprechende Modellierung wie dem Ausschluss konkurrierender Vorbedingungen oder dem Einsatz von Prioritäten immer eindeutig bestimmt, wird der Automat als deterministisch bezeichnet.
- Mächtigkeit der Ein- und Ausgabesprache: die eingehenden Stimuli und die Ausgaben eines Automaten können als Zeichen einer formalen Sprache aufgefasst werden. Dementsprechend eng ist die Verwandtschaft zwischen formalen Sprachen bzw. deren Grammatiken und Zustandsautomaten. Abstrakt betrachtet leistet eine (nicht-deterministische) FSM genau die Erkennung der gültigen Wörter einer formalen Sprache. Entsprechend können durch die benutzen Transitionen bzw. Zustände auch Ausgabewörter erzeugt werden. Die zugehörigen Ein- und Ausgabealphabete können endlich oder unendlich viele Symbole enthalten und die entsprechenden Grammatiken der Ein- und Ausgabesprachen endlich oder unendlich viele Wörter zulassen. Die grundlegenden Arbeiten zu formalen Sprachen und Zustandsautomaten gehen zurück auf den Mathematiker Alan Turing [Tur36] und werden in vielen Lehrbüchern zur theoretischen Informatik behandelt (u. a. in [Hed03]).

Bekannte Klassen von FSM sind die Moore- und Mealyautomaten. Beim Mooreautomaten hängt die Ausgabe nur vom Zustand ab, während beim Mealyautomaten eine Kombination aus Eingabezeichen und Zustand die Ausgabe bestimmt. Beide Automatenklassen sind allerdings gleich mächtig und lassen sich ineinander überführen. Eine Einführung in die praktische Verwendung von Zustandsautomaten in der Softwaretechnik – also etwas abseits der abstrakten theoretischen Informatik – gibt [Bal01].

Strukturierte Analyse (SA): Die SA geht auf die Arbeiten von Tom DeMarco [DeM79] zurück und beschreibt ein System durch einen Satz hierarchischer Datenflussdiagramme (DFD). Die DFD werden durch ein einzelnes Kontextdiagramm ergänzt, das die Schnittstellen des Systems zur Außenwelt wiedergibt. Alle

```
/*
Whenever the EB is applied, the train reaches standstill (read: _ALL_ paths starting at
train.ebApplied will lead to a state with v == 0).
*/
train.ebApplied --> (vTrain == 0)

/*
The train can never exceed its maximum speed.
*/
A[] (vTrain <= V_MAX_TRAIN)

/*
If the train is moving, the SiFa is not idle
*/
A[] ((vTrain > 1) imply (not sifa.idle))

/*
The SiFa is in normal mode as long as the train is moving and the first warning period has
not elapsed (or is just elapsed).
*/
A[] (sifa.running imply ((vTrain > 0) and (sifa.clk <= SIFA_FIRST_WARNING_TIME)))

/*
The SiFa is in first warning mode as long as the train is moving and the first warning
period has elapsed and the second warning threshold has not been reached (or is just elapsed).
*/
A[] (sifa.warnedOnce imply ((vTrain > 0) and (sifa.clk <= SIFA_SECOND_WARNING_TIME) and \
(sifa.clk >= SIFA_FIRST_WARNING_TIME)))

/*
The SiFa is in second warning mode as long as the train is moving and the second warning
period has elapsed and the EB threshold has not been reached (or is just elapsed).
*/
A[] (sifa.warnedTwice imply ((vTrain > 0) and (sifa.clk <= SIFA_EB_TIME) and \
(sifa.clk >= SIFA_SECOND_WARNING_TIME)))

/*
If the SiFa is in braking mode, the train must be moving and the EB time threshold must
have been reached
*/
A[] (sifa.braking imply ((vTrain > 0) and (sifa.clk >= SIFA_EB_TIME)))

/*
Whenever the SiFa is not idle, pressing the SiFa-button will lead to the normal
SiFa-state "running" with resetted timer.
*/
(user.sifaBtnPressed and (not sifa.idle)) --> (sifa.running and (sifa.clk == 0))

/*
If the train is moving and the time for an emergency brake is exceeded
(interval between two SiFa-button-presses), the train is in mode "EB".
*/
A[] (((vTrain >= 1) and (sifa.clk > SIFA_EB_TIME)) imply train.ebApplied)

/*
Make sure that the system never freezes
*/
A[] not deadlock
```

Abbildung 3.5: Temporallogische Aussagen über das in Abbildung 3.4 gezeigte System, durch die die *liveness properties* und *safety properties* mit Hilfe des Model Checkers „UPPAAL“ bewiesen werden können

innerhalb der DFD verwendeten Datentypen werden darüber hinaus in einem Datenwörterbuch aufgeführt.

Die an sich statischen DFD wurden später durch dynamische Komponenten wie Entscheidungstabellen, Kontrollflüsse und Pseudocode ergänzt, um außer der Systemstruktur auch Systemverhalten erfassen zu können.

Durch die strenge Hierarchisierung der DFD ist die Strukturierte Analyse ein typisches Beispiel für einen Top-Down-Entwicklungsprozess. Sie ist daher für gängige V-förmige Entwicklungsprozesse sehr gut geeignet.

Da die Symbolik, Syntax und Semantik der SA sehr schnell und einfach zu erlernen sind und sich die erzeugten Diagramme relativ leicht in (prozeduralen) Code umsetzen lassen, war die SA bis in die frühen 90er Jahre weit verbreitet. Mit der Ablösung des prozeduralen durch das objektorientierte Programmierparadigma wurde die SA jedoch zurückgedrängt, da eine objektorientierte Modellierung mit ihr nicht möglich ist.

Anmerkung: Die SA als solche zählt zu den Methoden und nicht zu den Beschreibungsmitteln. Dennoch sei sie hier erwähnt, weil ihr Einsatz üblicherweise an die Verwendung der für sie charakteristischen Diagramme – Kontext- und Datenflussdiagramme – gebunden ist. Allerdings lassen sich die genannten Diagramme auch außerhalb der SA verwenden und ebenso kann die SA auch mit anderen Diagrammen (beispielsweise farbigen Petrinetzen) durchgeführt werden.

3.3.1 TTCN-3

Anders als die zuvor erwähnten Sprachbeispiele, die sich alle auf die Beschreibung von Systemen beziehen, ist TTCN-3 eine reine Testbeschreibungssprache. TTCN-3, was für „Testing and Test Control Notation, Version 3“ steht, ist der momentan letzte Entwicklungsschritt einer Reihe von Sprachdefinitionen, die durch das ETSI herausgegeben werden und in früheren Versionen als „Tree and Tabular Combined Notation“ bekannt waren. TTCN-3 dient der Beschreibung von Testfällen und Testumgebungen und ist besonders für verteilte Systeme und Telekommunikationssysteme geeignet (die TTCN-3-Entwicklung wird u. a. wesentlich durch Nokia vorangetrieben). Die aktuelle Sprachdefinition [ETS05] stammt vom Juni 2005. Als eine der in der Fallstudie verwendeten Sprachen soll TTCN-3 in diesem Abschnitt genauer betrachtet werden.

Die wesentlichen Elemente innerhalb von TTCN-3 sind Testkomponenten für die Interaktion mit dem Prüfling, ein Arbiter für die Ermittlung des Testurteils und eine Ablaufsteuerung für die Umsetzung aller dynamischen Testaspekte. Der Test an sich wird dabei vorwiegend als Nachrichtenaustausch zwischen den Testkomponenten und dem Prüfling beschrieben. Basierend auf den Prüflingsreaktionen fällt jede Testkomponente ein „lokales“ Urteil für ihren Zuständigkeitsbereich und gibt dieses an den Arbiter weiter. Dieser vereinigt nach vorgegebenen Regeln alle Einzelurteile zu einem Gesamturteil. Die Interaktion zwischen Prüfling, Testkomponenten, Arbiter und Ablaufsteuerung ist auch in Abb. 3.6 wiedergegeben.

Da ausschließlich die Prüflingsschnittstellen für den Test und die Bildung des Testurteils herangezogen werden, ist TTCN-3 nur für BlackBox-Tests geeignet. Hinsichtlich der Testgranularität unterliegt es jedoch keinen Einschränkungen; TTCN-3 kann vom Modul- bis zum System- oder Abnahmetest sinnvoll eingesetzt werden.

Die Sprache TTCN-3 kennt drei verschiedene Repräsentationen, die je nach Anwendungsfall frei gewählt werden können:

- Die *Core Notation* entspricht einer rein textuellen Darstellung und erinnert sehr stark an Programmiersprachen wie C.
- Die *Tabular Notation* erlaubt die Darstellung von Testfällen in Tabellenform, ähnlich wie es in älteren TTCN-Versionen üblich war.

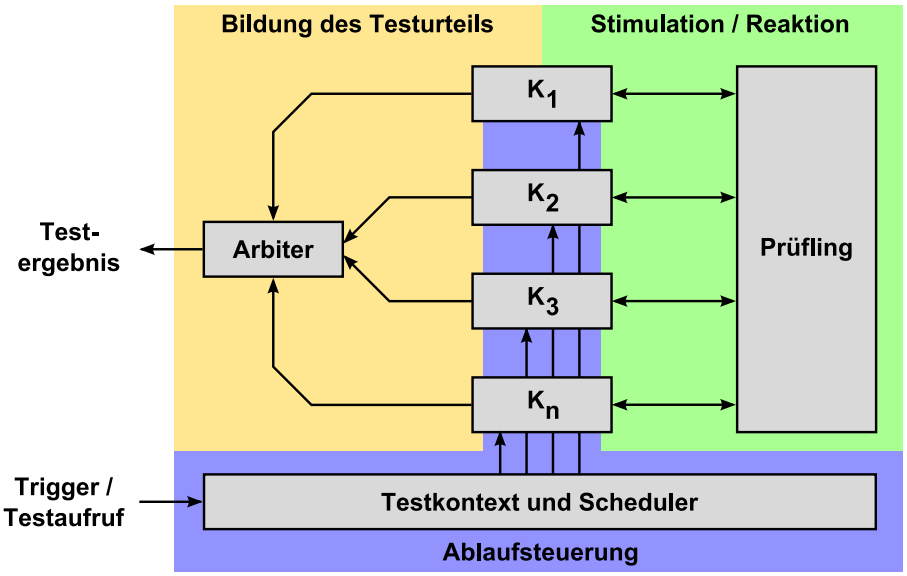

Abbildung 3.6: Elemente einer TTCN-3-Testumgebung und ihre Interaktion

- Das *Graphical Format* ist eine an MSCs angelehnte Möglichkeit, Testabläufe grafisch zu beschreiben.

Eine Alternative zum Graphical Format stellt das UML 2 Testing Profile (U2TP) dar, das eine UML-basierte Darstellung von TTCN-3 ermöglicht, aber auch vollkommen isoliert von TTCN-3 einsetzbar ist. Das U2TP wird in einem späteren Kapitel noch genauer vorgestellt.

Es ist hervorzuheben, dass TTCN-3 einen sehr großen Sprachumfang besitzt und durch seine mächtige Syntax und Semantik sehr komplexe Testbeschreibungen ermöglicht. Dadurch wird es einerseits sehr vielfältig einsetzbar, aber andererseits schwer erlernbar und für Toolhersteller nur mit großem Aufwand realisierbar.

Eine detaillierte Beschreibung und Analyse von TTCN-3 zusammen mit der Implementierung eines exemplarischen TTCN-3-Interpreters ist Gegenstand der Arbeit von Janke [Jan07]. Eine allgemeine Einführung in TTCN-3 und die Erläuterung der verfügbaren Sprachkonstrukte kann auch entsprechenden Fachbüchern entnommen werden [WDT+05].

3.3.2 LUSTRE und SCADE

SCADE ist ein Werkzeug zur Systemmodellierung, das vor allem in der Luftfahrtindustrie schon seit vielen Jahren im Einsatz ist. Das Programm SCADE wird von Esterel Technologies (Frankreich) vertrieben und stellt im Wesentlichen einen grafischen Editor für die Sprache SCADE zur Verfügung, die ihrerseits eine Erweiterung der Sprache LUSTRE ist [CPHP87, HCRP91].

In LUSTRE bzw. SCADE werden Systeme in Form synchroner Datenflüsse modelliert. Die Flüsse werden durch Knoten miteinander verknüpft, indem ein Knoten aus einem oder mehreren Eingangsflüssen einen oder mehrere Ausgangsflüsse erzeugt. Die Transformation zwischen Eingangs- und Ausgangsdaten erfolgt auf Basis formaler Gleichungen (beispielsweise durch boolsche, arithmetische oder temporale Operatoren). Knoten bzw. Netze von Knoten können hierarchisiert werden, d. h. ein Knoten kann aus mehreren Unterknoten bestehen. Das Gesamtsystem wird durch den obersten Knoten der Hierarchie repräsentiert; dessen Ein- und Ausgänge entsprechen den Schnittstellen des Systems zur Außenwelt. Ein Beispiel für ein Netz aus mehreren SCADE-Knoten, die über Datenströme miteinander verbunden sind, zeigt Abb. 3.7.

Diese Kurzbeschreibung verdeutlicht, dass das Grundprinzip der Modellierung mit SCADE dem der Datenflussdiagramme bzw. der SA (s. Seite 35) sehr ähnlich ist. Allerdings schränkt SCADE – im Vergleich zur SA – die Semantik der Datenflüsse in einem wichtigen Punkt ein: in SCADE sind die Datenflüsse *synchron getaktet*,

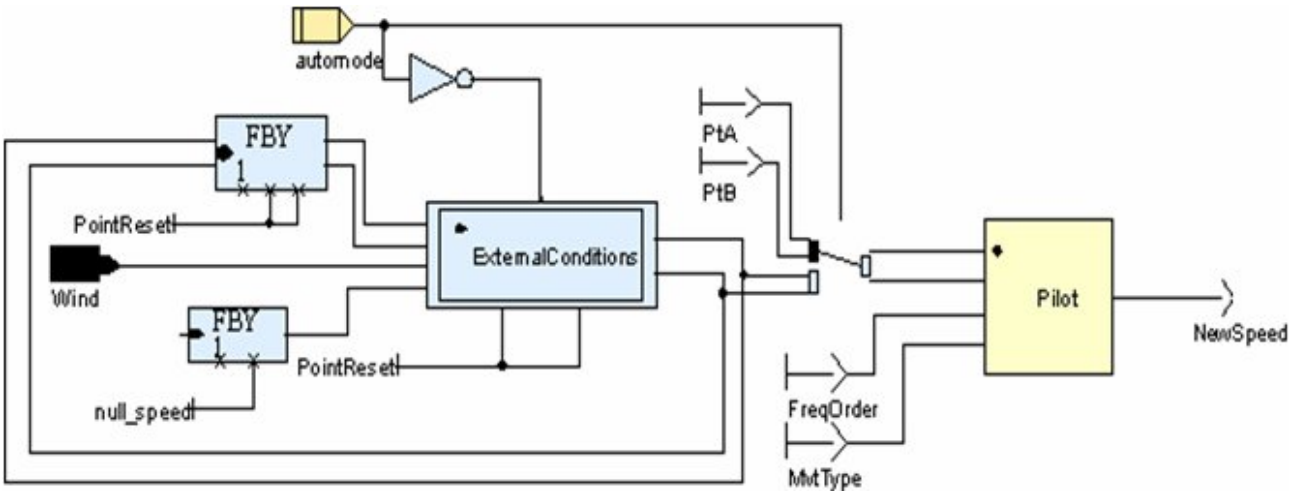

Abbildung 3.7: Beispiel eines SCADE-Netzes aus mehreren Knoten, Eingängen und Ausgängen (Quelle: Esterel Schulungsunterlagen)

das bedeutet, dass zu einem bestimmten Zeitschritt n von allen Datenflüssen das $n$te Element verarbeitet wird. Durch diese Einschränkung werden die Datenflussmaschinen deterministisch und vor allem ausführbar.

Die synchrone Taktung verbietet allerdings direkte Rückkopplungsschleifen. Dieses für viele Steuerungs- und Regelungsalgorithmen essentielle Konstrukt kann in SCADE daher nur realisiert werden, indem einem Eingang der Wert eines Ausgang aus dem *vorangegangenen* Zeitschritt rückgekoppelt wird. Dieses Verfahren ist auch vom Design digitaler Filter bzw. deren mathematischer Analyse mit Hilfe der z-Transformation bekannt. Mit Hinblick auf eine spätere Analyse des Netzwerkes bringt die Verwendung eines solchen Verzögerungsoperators allerdings Schwierigkeiten mit sich. Durch seinen Einsatz erhält das System ein „Gedächtnis", wodurch sein Zustandsraum signifikant vergrößert wird (es gehen nicht mehr nur die aktuellen Werte der Zustandsgrößen in den Zustandsraum ein, sondern auch deren Historie).

Das Programm SCADE unterstützt die Nutzung der Sprache SCADE durch folgende Module:

Editor: Der Editor erlaubt die grafische Modellierung von SCADE-Knoten und Netzen.

Simulator: Mit Hilfe des Simulators kann ein SCADE-Modell Schritt für Schritt ausgeführt und dabei beobachtet werden. Es ist auch möglich (beispielsweise für Regressionstests), zuvor gespeicherte Sequenzen von Eingangswerten zurückzuspielen.

Code Generator: In einem zweistufigen Prozess können SCADE-Dateien zunächst nach LUSTRE übersetzt werden, um anschließend in C oder Ada übertragen zu werden. Eine Besonderheit, die SCADE vor allem für das Design sicherheitsrelevanter Systeme in der Luftfahrt oder Eisenbahnsicherungstechnik interessant macht, ist, dass die Codeerzeugung nach gängigen Entwicklungsnormen *qualifiziert* ist (u. a. gemäß DO-178B, einer zentrale Entwicklungsnorm der Luftfahrtdomäne).

Konkret bedeutet das, dass eine Verifikation zwischen erzeugter Software und dem SCADE-Modell entfallen kann. Der qualifizierte Codegenerator stellt die Gleichwertigkeit von Code und Modell sicher. Die Überprüfung des Systemverhaltens kann daher schon am Modell erfolgen und muss nicht erst auf Codeebene stattfinden. Diese Überprüfung wird durch den „Model Checker" unterstützt.

Model Checker: Der Model Checker nutzt verschiedene Algorithmen des *Model Checking* [CGP00], um einen mathematischen Beweis zu führen, dass bestimmte Systemeigenschaften immer gelten bzw. bestimmte Systemzustände nie erreicht werden (z. B. eine Verklemmung). Der Beweis gilt dabei für jede Sequenz

von Eingangswerten. Sollte ein Beweis nicht erbringbar sein, erzeugt der Model Checker zudem eine Kombination von Eingangswerten als Gegenbeispiel.

In einer kurzen Fallstudie bewährte sich SCADE als effizientes Modellierungswerkzeug, das vor allem für die Entwicklung sicherheitsrelevanter Software interessant ist. Es zeigte sich aber auch, dass die Entwicklung mit SCADE in den passenden, methodischen Rahmen eingebettet sein muss, um richtig zu funktionieren. Insbesondere ist hier der Übergang von der Spezifikation zum Modell zu beachten. SCADE hat den gleichen Grad an Formalität wie eine reguläre Programmiersprache, daher muss auch bei der Verwendung von SCADE die Systemarchitektur und das Systemdesign vor dem Modellieren soweit vorentwickelt werden, wie es bei normaler Programmierung auch der Fall wäre.

SCADE unterstützt die Entwicklung also vor allem bei der Implementierung und später im Test bzw. bei der Zulassung. Eine Hilfestellung bei der Systemanalyse bzw. dem Design der Systemarchitektur kann es nicht geben. Daher sollte zwischen den informalen Kundenanforderungen und dem formalen SCADE-Modell unbedingt ein semiformaler Entwicklungsschritt eingeschoben werden, der z. B. auf UML beruhen kann.

3.3.3 XML und railML

In den letzten Jahren erfreuen sich Datenformate, die auf der eXtensible Markup Language (XML) beruhen, einer zunehmenden Beliebtheit. Die Vorteile, die zu dieser Verbreitung von XML geführt haben, sind:

- Daten können strukturiert in Bäumen abgelegt werden. Einzelne Teile eines Baums können durch eine beliebige Anzahl zusätzlicher Werte attribuiert werden.
- Das Datenformat, zumindest aber die Dokumentstruktur, ist menschenlesbar. Zusammengehörige Bereiche eines Baums werden durch sogenannte *Tags* eingeklammert, die beliebig benannt werden können. Das gesamte Dokument einschließlich seiner Strukturauszeichnung wird im Klartext gespeichert.
- Es gibt eine große Auswahl von Parsern für alle gängigen Programmiersprachen. Die meisten dieser Parser sind *OpenSource*, also im Quelltext verfügbar und ohne Lizenzgebühren in eigenen – auch kommerziellen – Projekten einsetzbar.
- Durch die separaten Beschreibungssprachen „Document Type Definition“ (DTD) und das neuere „XML Schema“ (XSD) können die Struktur und die erlaubten Datentypen eines spezifischen XML-Datenformats beschrieben werden. Ein XML-Dokument kann dann gegen die DTD- oder XSD-Definition auf Gültigkeit untersucht werden (Validierung).

Diesen Vorteilen steht als Nachteil vor allem die Dateigröße von XML-Dokumenten entgegen. Bedingt durch die Klartextspeicherung der Informationen ist der Speicherbedarf deutlich höher als bei binären Dateiformaten.

Es gibt verschiedene Ansätze, um die Dateigröße zu reduzieren (Kompression mit gängigen Packern, Substitution der Tag- und Attributnamen je nach deren Länge und Häufigkeit), aber das pure, unkomprimierte XML ist noch immer mit Abstand am weitesten verbreitet. Da Speicherplatz – unabhängig ob im Hauptspeicher oder auf Datenträgern – verglichen mit den Programmier- und Folgekosten durch binäre, proprietäre Formate sehr günstig ist, werden die größeren Dateien in den meisten Anwendungen für das einfachere Handling in Kauf genommen.

Eine Anwendung von XML-Daten in der Eisenbahndomäne stellt das offene Datenformat „railML®“[7] dar. railML hat es sich zum Ziel gesetzt, ein Austauschformat von Eisenbahninfrastruktur-, Fahrplan- und Fahrzeugdaten zu schaffen, das ohne Lizenzgebühren genutzt werden kann (auch kommerziell). Der Fokus liegt

[7] www.railml.org

dabei auf dem Datenaustausch zwischen Simulationsprogrammen, bei denen bisher jedes Programm sein eigenes, nicht offenes Format verwendet, wodurch die Kopplung verschiedener Tools erheblich erschwert wird.

Die Entwicklung von railML basiert auf der Arbeit Freiwilliger aus Industrie und Forschung, die das Datenformat kontinuierlich vorantreiben und dabei die Anregungen und Wünsche der Anwender berücksichtigen. Die im Rahmen der Arbeit notwendigen Diskussionen werden für jeden zugänglich in einer Newsgroup geführt. Darüber hinaus finden zweimal jährlich Treffen von Entwicklern, Anwendern und Interessierten statt, auf denen die Möglichkeit zum direkten Informationsaustausch gegeben ist.

Der durch den offenen Ansatz potentiell unstrukturierte Entwicklungsprozess wird durch Koordinatoren gelenkt, die die eingehenden Vorschläge und Änderungswünsche sammeln und im Diskussionsforum bzw. auf den Treffen zur Abstimmung stellen.

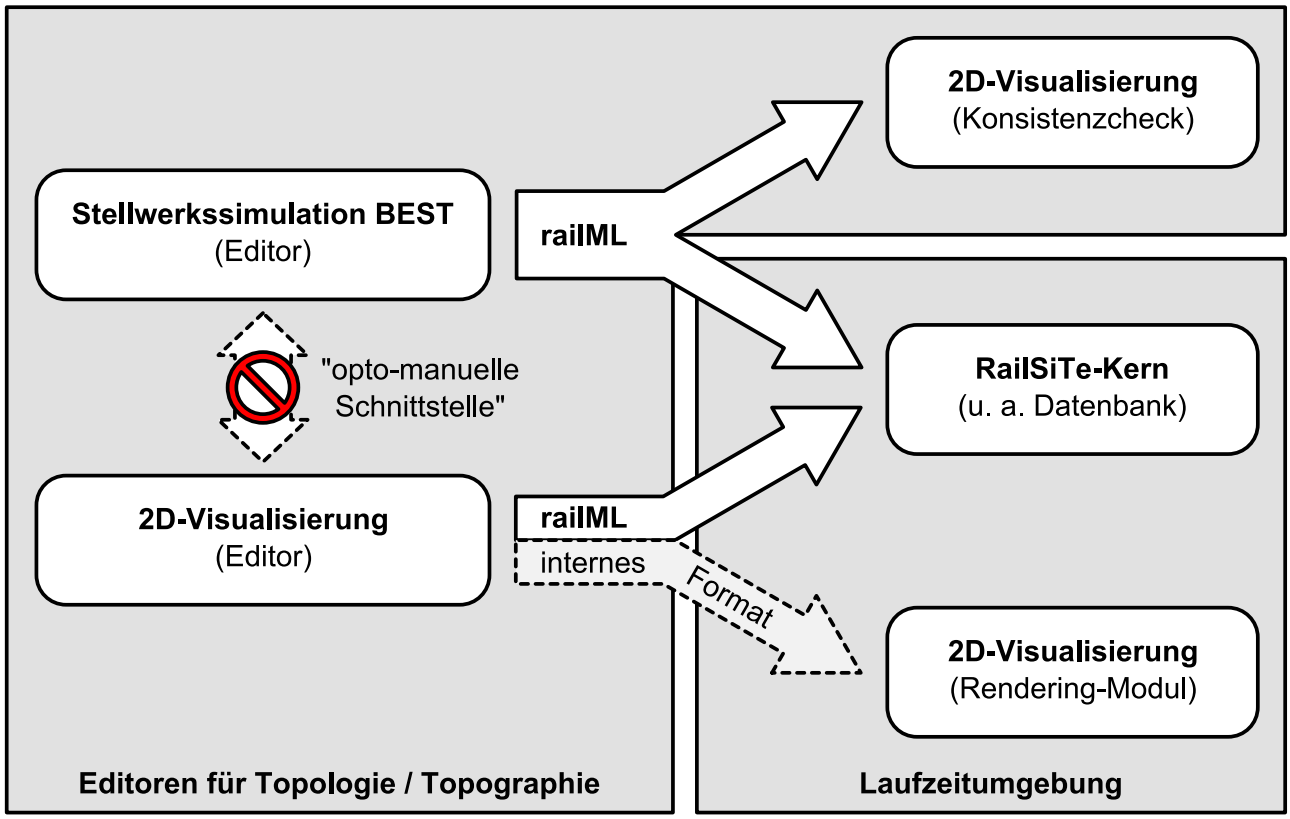

Abbildung 3.8: Einsatz von railML im DLR-Bahnlabor RailSiTe zur Kopplung von Editoren und Simulationsumgebung durch den Austausch statischer Infrastruktur- und Szenariodaten

Die Anzahl und das Spektrum der Nutzer von railML wächst stetig und umfasst Simulationsprogramme wie OpenTrack[8], Fahrplanberechnungsprogramme der Deutschen Bahn und das eisenbahntechnische Labor des DLRs, das RailSiTe. Im RailSiTe wird der Austausch statischer Infrastrukturdaten zwischen der Stellwerkssimulation BEST und dem Laborkern realisiert ([KGL07], Abb. 3.8). Darüber hinaus kann der entsprechende railML-Importfilter natürlich auch zum Einlesen von Infrastrukturbeschreibungen anderer Quellen genutzt werden.

Aktuelle Entwicklungsarbeiten von railML betreffen

- die Entwicklung eines Beschreibungsformats für Stellwerksdaten
- die Erweiterung des Infrastrukturformats um topografische Informationen, z. B. basierend auf GML[9]

[8] www.opentrack.ch, [Hür01]
[9] http://www.opengis.net/gml

- die Nutzung von railML zum Austausch von RBC-Projektierungen bei der Einrichtung und Zulassung neuer ETCS-Strecken (vgl. dazu auch [EV03])

und belegen das rege Interesse, das railML innerhalb der Eisenbahndomäne entgegen gebracht wird.

Abschließende Bemerkungen zu Beschreibungsmitteln

Abgesehen von natürlicher Sprache wurden alle in diesem Kapitel angesprochenen Beschreibungsmittel zunächst in der theoretischen Informatik entwickelt und haben anschließend über verschiedene Anwendungen und Projekte ihren Weg in die Eisenbahndomäne gefunden. Charakteristischerweise gibt es keine eigenständige Sprache, die spezifisch für die Anforderungen des Systems Eisenbahn entwickelt wurde. Mithin stellt jede Adaption oder Portierung einer Sprache in das Eisenbahnumfeld einen Kompromiss dar, der zwischen dem ursprünglichen Einsatzzweck und den Ansprüchen der Eisenbahndomäne geschlossen werden muss. Das sich daraus ergebende Spannungsfeld zwischen Formalitätsgrad, Problemorientierung und Handhabbarkeit einer formalen Technik wird beispielsweise von Einer beschrieben [Ein03].

Zwar wird seit einiger Zeit an sogenannten domänenspezifischen Sprachen (Domain Specific Languages, DSL) geforscht, allerdings wird darunter eher der Zuschnitt auf eine bestimmte „Pragmatikdomäne“ (z. B. Testfallbeschreibung, Systems Engineering, usw.) als auf eine bestimmte „Kontextdomäne“ (z. B. Automobiltechnik, Eisenbahn, Luftfahrt usw.) verstanden.

In ähnlicher Weise trifft dies auch auf die Sprache UML zu, auf die im folgenden Kapitel eingegangen wird. Sie ist zwar nicht domänenspezifisch, aber sie wurde ebenfalls zunächst von der und für die Informatik entwickelt und diffundiert nun langsam in andere Anwendungsfelder. Ihre Entwicklungshistorie und ihre Eigenschaften sind Gegenstand des nächsten Kapitels. Ihr Einsatz wird später in der Fallstudie ausführlich erläutert.

4 Die UML und ihre Profile

Die Unified Modeling Language (UML) ist eine Sprache für die objektorientierte, grafische Darstellung des Aufbaus, des Verhaltens und der Funktionen von Systemen. Der Anwendungsbereich der UML ist dabei nicht auf eine bestimmte Domäne beschränkt und auch nicht ausschließlich für die Beschreibung technischer Systeme geeignet; der weite Einsatzbereich der UML umfasst beispielsweise auch Geschäftsprozesse oder Konzernstrukturen.

In den folgenden Abschnitten sollen die geschichtliche Entwicklung und die Struktur der UML kurz wiedergegeben werden. Darüber hinaus werden die für kombinierte Testfall- und Systemmodelle wichtigen UML-Profile genauer vorgestellt.

4.1 Historie

Parallel zum Aufkommen der objektorientierten Programmierung in den 90er Jahren entstanden auch viele Bestrebungen, dieses neue Programmierparadigma durch geeignete Modelle und Methoden zu unterstützen. Zu diesen Ansätzen zählen unter anderem (vgl. Abb. 4.1):

- Responsibility Driven Design (RDD) von Rebecca Wirfs-Brock [WBWW90]
- Object Modeling Technique (OMT) von James Rumbaugh [RBP+91]
- Object-Oriented Software Engineering (OOSE) von Ivar Jacobson [JCJ92]
- Object Behavior Analysis (OBA) von Kenneth Rubin und Adele Goldberg [RG92]
- Methodology for Object-Oriented Software Engineering of Systems (MOSES) von Brian Henderson-Sellers und Julian M. Edwards [HSE94, HSE93]
- Semantic Object Modeling Approach (SOMA) von Ian Graham [Gra00]
- Object-Oriented Process, Environment and Notation (OPEN) und Object Modeling Language (OML, ein Vorläufer bzw. Konkurrent zur UML) von Ian Graham und Brian Henderson-Sellers [FHSG97, HSFG97]

Diese unterschiedlichen Ansätze wurden dann vor allem von Grady Booch, Ivar Jacobson und James Rumbaugh (alle drei waren damals bei der Rational Software Corporation tätig) zum ersten Entwurf der UML vereint, die von der Object Management Group (OMG) im November 1997 als UML 1.1 veröffentlicht wurde.

Die Entwicklung der UML 1.x, so der Sammelbegriff für die Versionen vor der UML 2, wurde in den kommenden Jahren vorangetrieben (2000: UML 1.3, 2001: UML 1.4), bis schließlich – nach mehreren Entwürfen und Vorversionen – im Juli 2005 die stark überarbeitete Version UML 2.0 veröffentlicht wurde. Die aktuelle Fassung ist UML 2.1.1 vom Februar 2007.

4.2 Bestandteile der aktuellen UML-Spezifikation

Im Vergleich zur UML 1.x wurde bei der UML 2 die Spezifikation von Grund auf neu strukturiert und in vier Teile untergliedert [Obj07b, Obj07a, Obj06b, Obj06c]:

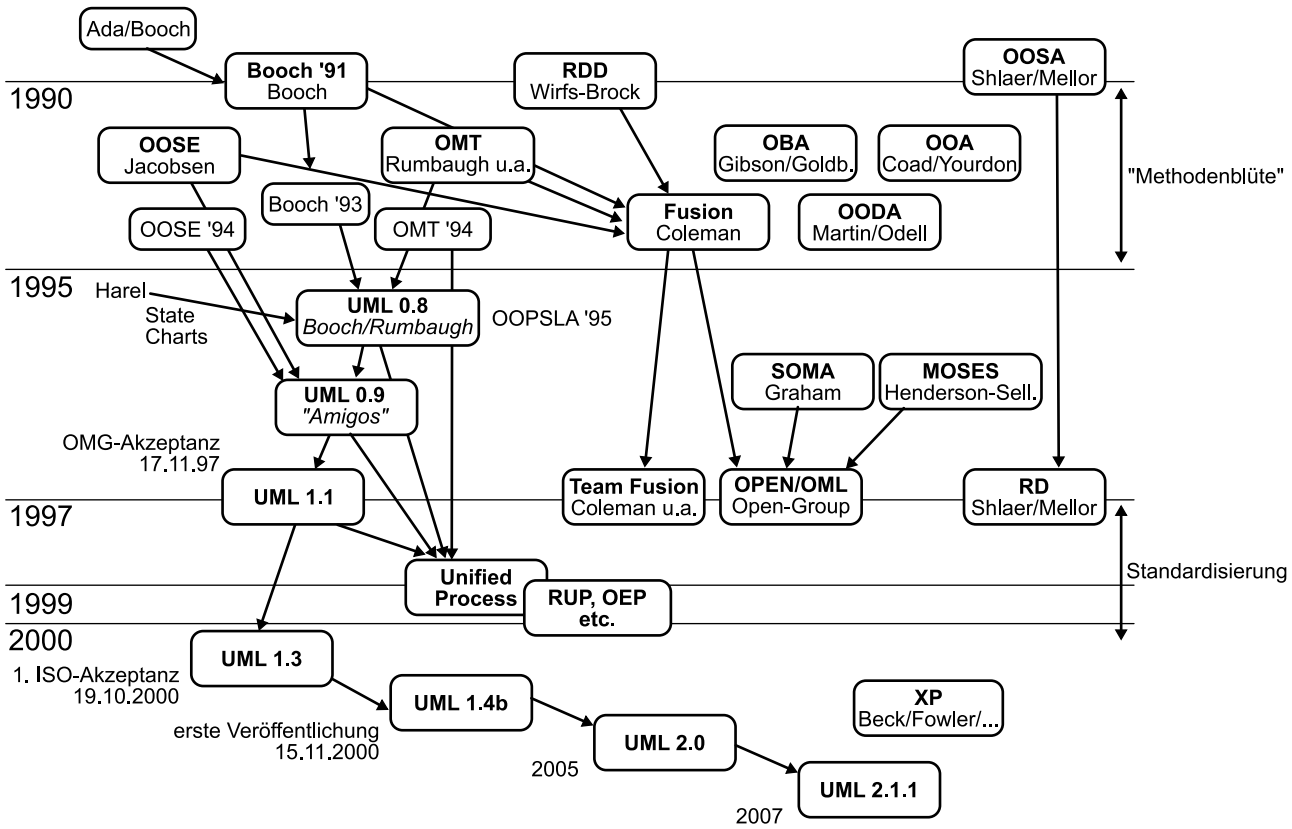

Abbildung 4.1: Historische Entwicklung und Verflechtung verschiedener objekt-orientierter Methoden und Notationen, die schließlich in der UML aufgehen (Quelle: http://www.oose.de/uml_historie.htm)

Infrastructure: Definiert die wesentlichen Grundbausteine klassenbasierter Modellierung wie beispielsweise Klassen, Pakete oder Assoziationen. Unter anderem stellen sich dabei auf Infrastrukturebene auch Beziehungen wie Assoziationen als Klasse dar: sie repräsentieren die gemeinsamen Eigenschaften aller Beziehungen zwischen Elementen. Auf diese Weise stellt die Infrastruktur eine Sammlung von Beschreibungs- bzw. Sprachelementen zur Verfügung, mit deren Hilfe die „eigentliche“ UML beschrieben werden kann.

Superstructure: Baut auf der Infrastruktur auf und definiert die Sprachelemente, aus denen ein UML-Anwender seine individuellen Modelle zusammensetzt. Die Superstruktur definiert also die Sprache, in der die “eigentlichen“ UML-Modelle beschrieben werden. Für diese Sprachdefinition werden die Elemente der Infrastruktur in Form von Paketen von der Superstruktur eingebunden.

Object Constraint Language (OCL): Textuelle Notation von Randbedingungen oder Zusicherungen an Modellelemente. Das können beispielsweise Wertebereiche für Attribute, Zustandsübergangsbedingungen (engl. *guards*) oder Invarianten für Instanzen von Klassen sein.

Diagram Interchange Specification: vereinheitlicht den Austausch von Diagrammen zwischen verschiedenen Anwendungen und wurde erst mit der UML 2 herausgegeben. Die Betonung liegt hier auf dem Austausch von *Diagrammen*, weil für das *Modell* bereits seit UML 1.x ein Austauschformat definiert ist (XMI, XML Metadata Interchange). Da XMI aber nur das Modell und nicht dessen grafische Repräsentation speichert, wurde der Datenaustausch zwischen UML-Programmen erheblich erschwert. Durch die Diagram Interchange Specification soll diese Lücke geschlossen werden.

Das auf den ersten Blick verwirrende Zusammenspiel von Infra- und Superstruktur liegt im Konzept der *Metamodellierung* begründet. Ein Metamodell beschreibt die Sprache, in der ein (untergeordnetes, konkreteres)

Modell beschrieben wird.

Im einfachsten Fall finden zwei Beschreibungsebenen Verwendung: eine abstrakte Ebene mit Klassen und eine konkrete Ebene mit deren Instanzen zur Laufzeit (Objektmodell oder auch Original). Hier findet noch keine Metamodellierung statt.

Wird aber zusätzlich die Spezifikation der Sprache berücksichtigt, in der das Klassenmodell abgefasst wurde, erhöht sich die Anzahl der Modellierungsebenen bereits auf drei. Das Modell der Modellierungssprache des eigentlichen Systems wird dann als *Metamodell* bezeichnet.

Im Falle der UML und verwandter, von der OMG spezifizierter Sprachen kommt noch eine weitere Metaebene hinzu, das Meta-Metamodell (manchmal auch als Meta^2-Modell bezeichnet). Das Meta-Metamodell stellt eine Sprache zur Verfügung, in der das Metamodell des Systemmodells abgefasst werden kann.

Diese vier Abstraktionsebenen werden häufig von M0 (konkret) nach M3 (abstrakt) durchnummeriert (siehe Abb. 4.2). Die Beziehungen zwischen den Ebenen sind durch kontinuierliche Abstraktion (Klassenbildung) oder Konkretisierung (Instantiierung) gekennzeichnet. Prinzipiell spricht nichts gegen höhere Metaebenen als M3, allerdings hat die Praxis gezeigt, dass in den meisten Fällen vier Ebenen ausreichend sind.

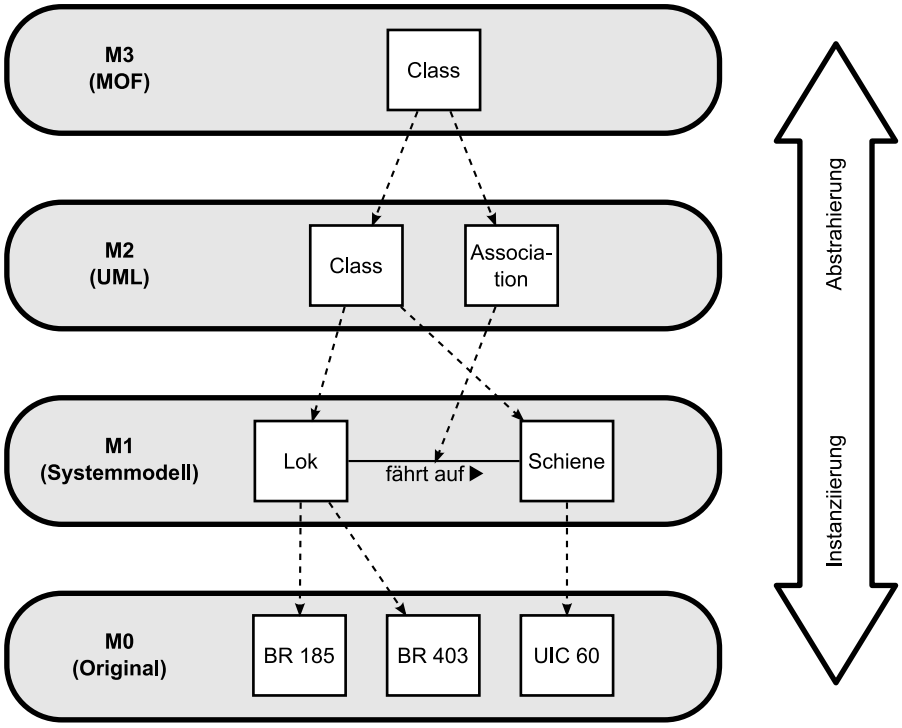

Abbildung 4.2: Ebenen der Metamodellierung bei OMG-Sprachen (die gestrichelten Pfeile symbolisieren eine Instantiierung zwischen verschiedenen Modellierungsebenen)

Die OMG hat als Meta-Metaebene M3 das MOF (MetaObject-Facility) spezifiziert, das seinerseits die Grundlage für andere Modellierungssprachen wie beispielsweise die UML oder das „Common Warehouse Metamodel“ bildet.

In diesem Zusammenhang ist die Ambivalenz der UML-Infrastrukturdefinition hervorzuheben. Es wäre zu erwarten, dass sie als Teil der UML-Spezifikation auf dem MOF aufbauend der Ebene M2 zuzuordnen ist. Tatsächlich sind aber die UML-Infrastruktur und das eigentliche MOF keineswegs aufeinander aufbauend bzw. disjunkt. Vielmehr wird die zentrale Komponente der UML-Infrastruktur, die *InfrastructureLibrary*, vom MOF eingebunden (*„merged“*) und dadurch zu einem Teil des MOF. Somit ist die UML-Infrastruktur eher der Ebene M3 zuzuordnen, während die eigentliche UML-Spezifikation auf der Ebene M2 vornehmlich durch die UML-Superstruktur repräsentiert wird.

Diese zunächst als umständlich erscheinende Definition einer einheitlichen Meta-Metasprache bietet jedoch eine Reihe von Vorteilen:

- Einfacher Daten- und Modellaustausch: alle MOF-basierten Sprachen können ihre Modelle im gleichen Format austauschen. Dies ist die Grundlage von XMI (XMI ist direkt auf dem MOF aufgebaut).
- Gemeinsame Repositories: Modelle unterschiedlicher, aber dennoch MOF-basierter Sprachen können kombiniert in einem einzigen Repository koexisitieren, da sich auf die gleichen Strukturen zur Datenspeicherung zurückgreifen.
- Möglichkeit zur Modelltransformation: dadurch, dass die Bedeutung verschiedener Modellierungskonstrukte auf der Meta-Metaebene spezifiziert ist, können konkrete Systemmodelle in unterschiedliche Repräsentationen transformiert werden. Diesen Gedanken macht sich die MDA (Model Driven Architecture) zunutze, die zwischen plattformunabhängigen und plattformspezifischen Systemmodellen unterscheidet und diese ineinander transformiert.
- Sprachanpassung: der Mechanismus, eine bestehende Sprache an besondere Anforderungen anzupassen, ist im MOF verwurzelt. Durch eine einheitliche Sprache für Modellierungssprachen – genau das ist das MOF – können so auch Sprachanpassungen formal definiert und appliziert werden.
- Einheitliche Sprachbeschreibung: die (textuelle / grafische) Spezifikation von Sprachen und Spracherweiterungen wird durch das MOF vereinheitlicht und führt so zu einer höheren Qualität und Eindeutigkeit der Sprachdefinition.

Vor allem die letzten beiden Punkte werden durch die UML-Anpassungen SysML (Systems Engineering Language) und U2TP (UML 2 Testing Profile) intensiv genutzt. SysML und U2TP sind als MOF-basierte Erweiterungen zur UML angelegt und beschreiben in ihrer Spezifikation [Obj06a, Obj05b] die notwendigen Erweiterungen und Änderungen am UML-Metamodell, um die gewünschte Spezialsprache zu erzeugen. Eine solche Erweiterung der UML wird auch als „UML-Profil“ bezeichnet.

4.3 Die Diagramme der UML

Wie zuvor bereits in den Abschnitten 3.2 und 2.1 erläutert, sind für eine vollständige Systembeschreibung statische und dynamische Aspekte zu berücksichtigen. Dem entsprechend stellt die UML Struktur- und Verhaltensdiagramme zur Verfügung, in denen sich die jeweiligen Systemeigenschaften abbilden lassen. Die Notation, Syntax und Semantik der UML-Diagramme ist Gegenstand einer großen Anzahl an Publikationen. Eine umfassende Beschreibung der UML-Diagramme mit zahlreichen Beispielen findet sich u. a. in [Bal01] (nur UML 1.x), [RHQ+05] und [Kec06]. Aus diesem Grund sollen hier nur kurz die Diagrammtypen der UML eingeführt werden. Eine ganz knappe Darstellung der Diagramme und ihrer Inhalte kann außerdem Tabelle 4.1 entnommen werden. Für alle Details sei auf die Literatur verwiesen.

4.3.1 Strukturdiagramme

Klassendiagramm: Klassen und ihre Beschreibung in Klassendiagrammen bilden den Kern der objektorientierten Modellierung und somit auch der UML. Klassendiagramme stellen die Eigenschaften von Klassen – ihre Methoden und Attribute – und die Beziehungen zwischen Klassen dar. Zu den wichtigsten Beziehungen zählen die Generalisierung (Vererbung), die Assoziation, die Aggregation und die Komposition. Syntax und Semantik des Klassendiagramms haben eine lange Entwicklung hinter sich, wodurch es zwar in seinen Ausdrucksmöglichkeiten sehr mächtig wurde, aber leider auch einen erhöhten Lernaufwand erfordert.

	Diagramm	Zweck, Inhalt
Struktur	Klassendiagramm	Beziehungen zwischen Klassen
	Objektdiagramm	Beziehungen zwischen Objekten (Klasseninstanzen)
	Komponentendiagramm	BlackBox-Darstellung von Subsystemen und Komponenten
	Kompositionsstrukturdiagramm	WhiteBox-Darstellung von Komponenten
	Verteilungsdiagramm	Funktionszuweisung in verteilten Systemen
	Paketdiagramm	Einbinden von Datentypen aus anderen Paketen
Verhalten	Zustandsdiagramm	Darstellung zustandsdiskreten Verhaltens und Zustandsübergänge
	Aktivitätsdiagramm	Abfolge von Aktivitäten inklusive Kontroll- und Datenfluss
	Anwendungsfalldiagramm	Funktionale Sicht auf das System; Interaktion mit der Umgebung
	Kommunikationsdiagramm	Nachrichtenaustausch zwischen Objekten
	Sequenzdiagramm	Nachrichtenaustausch zwischen Objekten
	Timing-Diagramm	Zustandsänderungen über der Zeit
	Interaktionsübersichtsdiagramm	Interaktion als Reihe von Verweisen auf andere Verhaltensdiagramme

Tabelle 4.1: Übersicht über alle UML-Diagramme und ihren jeweiligen Verwendungszweck

Objektdiagramm: Während Klassen eine abstrakte und zeitlich invariante Sicht auf ein System darstellen, repräsentieren Objekte (also die Instanzen von Klassen) eine spezielle Systemkonfiguration zu einem bestimmten Zeitpunkt. Objektdiagramme beschreiben somit die Beziehungen zwischen Objekten in einem bestimmten Systemzustand. Insbesondere lassen sich hier auch Beziehungen zwischen Instanzen derselben Klasse darstellen, was im Klassendiagramm nur eingeschränkt möglich ist.

Die Objekte im Objektdiagramm sind zwar Instanzen von Klassen und daher auf einer niedrigeren Abstraktionsebene als Klassen, aber dennoch sind sie im Sinne der in Abbildung 4.2 noch Teil der Ebene M1. Auf Ebene M0 befinden sich die „echten“ Objekte, während die Objekte im Objektdiagramm nach wie vor Modelle sind, die das Original im Sinne von Abschnitt 2.1 nur verkürzt bzw. innerhalb einer gewissen Pragmatik beschreiben.

Komponentendiagramm: Im Komponentendiagramm werden Teile des Systems zu Komponenten mit eindeutig definierten Schnittstellen zusammengefasst. Ziel ist dabei, eine Austauschbarkeit bzw. Wiederverwendung von Komponenten zu erreichen. Durch den Fokus auf die Schnittstellen der Komponente wird diese gleichsam als *BlackBox* behandelt, deren interne Realisierung nicht genauer spezifiziert ist.

Kompositionsstrukturdiagramm: Im Kompositionsstrukturdiagramm wird im Gegensatz zum Komponentendiagramm eine *WhiteBox*-Sicht auf das System eingenommen. Es zeigt die internen Subsysteme bzw. Module einer Komponente auf, weist ihnen Rollen zu und stellt die internen Datenflüsse dar.

Verteilungsdiagramm: Im Falle von verteilten Systemen – wie Client-Server-Applikationen oder auch dem Zusammenspiel mehrerer Steuergeräte in einem Automobil – kann mit Hilfe von Verteilungsdiagrammen eine Zuweisung von Funktionen auf die verschiedenen Knoten in dem vernetzten System repräsentiert werden. Außerdem werden die Kommunikationskanäle zwischen den Knoten dargestellt. Das Verteilungsdiagramm ähnelt somit dem Kompositionsstrukturdiagramm, ist allerdings weit weniger mächtig und setzt auf einer höheren Abstraktionsebene an.

Paketdiagramm: Klassen und andere (Daten-)Typen werden in der UML in Paketen organisiert. Durch Einbinden von Paketen in andere Modelle werden die darin enthaltenen Elemente zugänglich gemacht und

somit eine Wiederverwendung ermöglicht. Genau diese Einbeziehung von Paketen bzw. Bibliotheken wird im Paketdiagramm dargestellt. Anders als beispielsweise in einem Kompositionsstrukturdiagramm werden keine internen Kommunikationsbeziehungen aufgebaut; es werden lediglich die Typen aus anderen Paketen sichtbar gemacht (genauer: in den Namensraum eingebunden).

4.3.2 Verhaltensdiagramme

Zustandsdiagramm: Im Zustandsdiagramm kann das Verhalten von Klassen oder ihrer Methoden mit Hilfe von endlichen Automaten (siehe auch Seite 34) erfasst werden. Zustände können hierarchisiert und zusammengesetzt werden und zum Auslösen von Zustandsübergängen stehen verschiedene Ereignisse wie Events, Methodenaufrufe oder Timer zur Verfügung. Zustandstransitionen können darüber hinaus durch *Guards* mit Nebenbedingungen belegt werden, die für eine erfolgreiche Aktivierung der Transition erfüllt sein müssen.

Aktivitätsdiagramm: Das Aktivitätsdiagramm ist entfernt mit einem Flussdiagramm vergleichbar und erlaubt die Notation einer Abfolge von Aktivitäten, die mit Hilfe sogenannter *Swimlanes* einzelnen Systemteilen zugeordnet werden können. Anders als das Zustandsdiagramm trifft es keine Aussage über den (internen) Systemzustand, sondern stellt eher eine Handlungsabfolge dar. Es ermöglicht eine übersichtliche Darstellung paralleler Aktivitätsabläufe und stellt auch Notationselemente für deren Synchronisation zur Verfügung. Darüber hinaus kann neben dem Kontrollfluss auch ein Objekt- bzw. Datenfluss in das Diagramm aufgenommen werden. Außerdem können Signale versendet und empfangen werden und durch „Unterbrechungskanten“ ist auch die Semantik für interrupt-ähnliches Verhalten hinterlegt.

Anwendungsfalldiagramm: Die rein funktionalen Anforderungen, die ein System erfüllen muss, lassen sich in Anwendungsfalldiagrammen hinterlegen. Sie stellen basierend auf Szenarien die Interaktion des Systems mit seiner Umwelt dar und definieren zugleich die Systemgrenze. Die Anwendungsfälle sind vollkommen losgelöst von der späteren Implementierung und sind als sehr abstrakte Sicht auf das System zu verstehen. Sie stellen auch eine wichtige Grundlage für spätere (funktionale) Tests dar.

Kommunikationsdiagramm: Das Kommunikationsdiagramm ist als dynamisches Pendant zum Objektdiagramm zu verstehen. In ihm wird der Datenaustausch zwischen Objekten in einer bestimmten Situation notiert. Durch Voranstellen von Sequenznummern vor die Einzelnachrichten wird die Reihenfolge des Nachrichtenaustauschs genau festgelegt. Die grafische Anordnung der Kommunikationspartner innerhalb der Diagrammfläche kann dabei beliebig gewählt werden.

Sequenzdiagramm: Ähnlich wie ein Kommunikationsdiagramm repräsentiert auch das Sequenzdiagramm einen Nachrichtenaustausch zwischen Objekten, jedoch mit wesentlich strengerer Syntax. Die Kommunikationspartner sind alle auf einer horizontalen Achse angeordnet und der Nachrichtenaustausch wird durch Pfeile zwischen ihnen dargestellt. Die Vertikale dient dabei als Zeitachse. Die dynamische Erzeugung und Zerstörung von Objekten, alternative Nachrichtenabfolgen, Kontrollflüsse und zeitliche Randbedingungen lassen sich präzise in ein Sequenzdiagramm übernehmen. Die Sequenzdiagramme gehen zurück auf die *Message Sequence Charts* (MSC, siehe auch Seite 34), die schon seit längerer Zeit vor allem in der Telekommunikationsindustrie eingesetzt werden.

Timing-Diagramm: Das Timing-Diagramm stellt gewissermaßen eine Verschmelzung von Zustands- und Sequenzdiagramm dar. Es stellt verschiedene Systemzustände auf der vertikalen Achse dar und nutzt die Horizontale als Zeitachse. So kann der Zustandsverlauf auch von mehreren Objekten über der Zeit aufgetragen werden. Eintreffende Nachrichten, die einen Zustandswechsel auslösen, werden als senkrechte Pfeile ebenfalls in das Diagramm übernommen.

Interaktionsübersichtsdiagramm: Interaktionsübersichtsdiagramme sind als Variation von Aktivitätsdiagrammen zu verstehen. Anstelle „normaler“ Aktivitäten enthalten sie Verweise auf andere Interaktionsdiagramme wie beispielsweise Sequenzdiagramme. Sie verdeutlichen also den Kontrollfluss zwischen Interaktionen auf einer übergeordneten Ebene. Genau wie Aktivitätsdiagramme verfügen auch Interaktionsübersichtsdiagramme über Möglichkeiten zur Darstellung und Synchronisierung paralleler Abläufe.

4.4 UML-Werkzeuge und das BMW-Prinzip

Für die Anfertigung der zuvor beschriebenen Diagramme ist natürlich die Unterstützung durch ein geeignetes Werkzeug notwendig, obwohl die UML als solches – als reines Beschreibungsmittel – natürlich werkzeug- und plattformneutral ist. Prinzipiell wäre sogar eine rein manuelle Modellierung basierend auf Papierzeichnungen möglich, allerdings ist das selbstverständlich nicht sinnvoll.

Für die computergestützte Modellierung steht eine Vielzahl von Programmen zur Verfügung, die den Anwender auf unterschiedlichsten Ebenen unterstützen. Das Spektrum reicht dabei von eher einfachen Zeichenprogrammen (wie beispielsweise Microsoft Visio oder das OpenSource-Programm Dia[1]) bis hin zu echten Modellierungswerkzeugen wie Rational Rose von IBM, Rhapsody von Telelogic, Together von Borland oder Artisan Studio von Artisan Software[2].

Die letztgenannten Werkzeuge nutzen die oben beschriebene Metamodellierung, um ein Modell des UML-Modells zu speichern. Das erlaubt Funktionen wie beispielsweise Modelltransformationen, Zusammenführen von Modellen, Konsistenzprüfungen oder die automatische Erzeugung von Dokumentation. Darüber hinaus ermöglichen diese Werkzeuge auch die Erzeugung von Quellcode (überwiegend aus den Klassen- und Zustandsdiagrammen) für unterschiedlichste Programmiersprachen.

Ein Beschreibungsmittel und dessen Unterstützung durch ein Werkzeug ist allerdings noch nicht ausreichend, um eine strukturierte Systementwicklung durchzuführen. Beides ist dafür notwendig, aber nicht hinreichend. Vielmehr müssen Beschreibungsmittel und Werkzeug noch durch ein zielgerichtetes Vorgehen, die *Methode*, ergänzt werden. Erst durch die Kombination von Beschreibungsmittel, Methode und Werkzeug ist eine systematische und effektive Systementwicklung möglich. Das Zusammenwirken dieser drei Komponenten wird auch als das „BMW-Prinzip“ bezeichnet ([Sch99], Abb. 4.3).

Aus methodischer Sicht erfolgt die Modellierung im UML-Umfeld objekt-orientiert (siehe auch [Bal01]). Eine mögliche grundsätzliche Vorgehensweise ist dabei:

1. Erstellung einer funktionalen Systembeschreibung auf Basis von Anwendungsfällen und Anwendungsfalldiagrammen.
2. Identifikation der wichtigsten Klassen und deren Beziehungen; Modellierung dieser Aspekte im Klassendiagramm.
3. Erfassen von Interaktionen in Sequenz- und Zustandsdiagrammen.
4. Ableiten von Methoden für die Klassen aus den Interaktionen.
5. Iterative Wiederholung dieses Vorgangs; dadurch kontinuierliche Verfeinerung und Korrektur des Modells.

Darüber hinaus wird häufig noch zwischen der weitestgehend lösungsneutralen Systemanalyse (OOA - Object Oriented Analysis) und dem darauf aufbauenden Entwurf (OOD - Object Oriented Design) unterschieden. Für größere Systeme ist die separate Analyse ein wichtiger Schritt, um bei allen Beteiligten ein einheitliches

[1] http://www.gnome.org/projects/dia
[2] Alle genannten Produktnamen sind eingetragene Warenzeichen der jeweiligen Hersteller

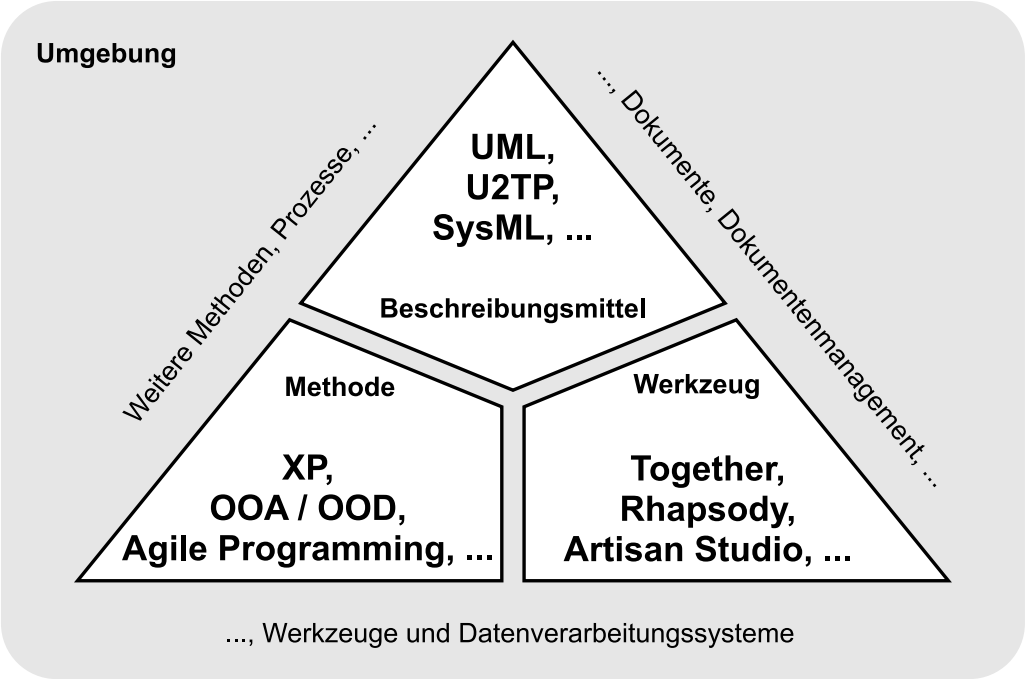

Abbildung 4.3: Schematische Darstellung des BMW-Prinzips und seine Einbettung in das Entwicklungsumfeld

Systemverständnis zu erreichen. Ein Nachteil ist jedoch, dass bei Verwendung von OOA und OOD alles doppelt modelliert wird, nur unter verschiedenen Betrachtungswinkeln. Die MDA versucht diesen Nachteil durch eine automatische Transformation eines plattformunabhängigen Modells (PIM - Platform Independent Model) in ein plattformspezisches Modell (PSM - Platform Specific Model) zu beseitigen, allerdings sind auch dieser Methode Grenzen gesetzt.

Insgesamt muss bei der Anwendung des BMW-Prinzips darauf geachtet werden, dass sich Beschreibungsmittel, Methode und Werkzeug möglichst harmonisch in ihre Umgebung einfügen. Fällt beispielsweise die Wahl der Entwicklungsmethode auf „eXtreme Programming“ und existiert parallel dazu ein Qualitätsmanagementprozess, der die Erstellung umfangreicher Dokumentationen verlangt, entsteht eine Konfliktsituation. Die Einführung neuer BMW-Komponenten erfordert also die Berücksichtigung des gesamten Entwicklungsumfelds, in das das Beschreibungsmittel / die Methode / das Werkzeug eingebettet wird (vgl. Abb. 4.3).

In diesem Zusammenhang kommt den Entwicklern eine besondere Rolle zu, da sie neue Tools, Sprachen oder Verfahren erst erlernen müssen. Besonders eine umfangreiche Sprache wie die UML kann in einer solchen Situation abschreckend und demotivierend wirken und dadurch die Akzeptanz neuer Entwicklungsansätze nachhaltig beeinträchtigen. Aus diesem Grund wurden Anpassungen, sogenannte Profile, der UML geschaffen, um für einen speziellen Kontext eine einfach zu erlernende Spezialsprache zur Verfügung zu stellen. Ein Beispiel für eine solche Spezialsprache ist SysML, die sich besonders an Ingenieure wendet und im folgenden Abschnitt genauer vorgestellt wird.

4.5 Die Systems Modeling Language (SysML)

Ein häufig vorgebrachter Einwand gegen die Verwendung von UML ist deren Komplexität und die dadurch notwendige lange Einarbeitungszeit. Darüber hinaus haben Konzepte und Begriffe aus der Softwareentwicklung die Sprache an vielen Stellen geprägt, was viele potentielle Nutzer, denen ein starker Informatikhintergrund fehlt, abschreckt.

Dieser Nachteil wurde schon früh von Toolherstellern[3] und UML-Nutzern erkannt und daraufhin die Entwicklung eines UML-Profils gestartet, das verglichen mit der UML mit einem reduzierten Sprachumfang auskommt und dessen Semantik auf die Entwicklung und Beschreibung technischer Systeme im allgemeinen ausgerichtet ist. Das Ergebnis dieser Entwicklung ist die „Systems Modeling Language“, kurz SysML.

Zwischenzeitlich gab es zwei konkurrierende SysML-Spezifikationen auf dem Markt, aber mittlerweile wurden beide Fassungen zur „OMG SysML“ vereinigt und als Spezifikationsentwurf von der OMG akzeptiert [Obj06a].

Der Sprachumfang von SysML, der auch durch Abb. 4.4 veranschaulicht wird, erlaubt die Beschreibung einer Reihe von Struktur- und Verhaltensaspekten technischer Systeme[4]:

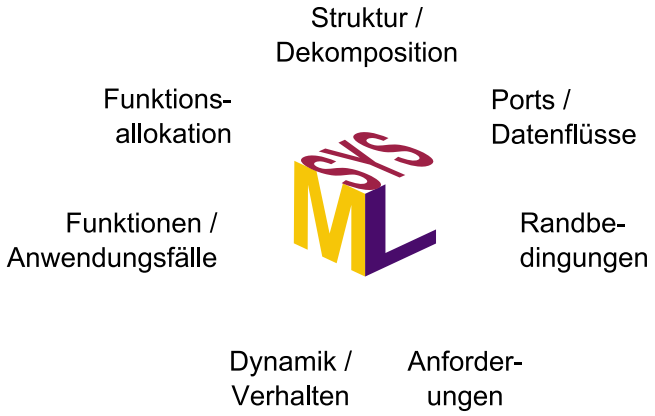

Abbildung 4.4: Komponenten der Systems Modeling Language (SysML)

Dekomposition in Subsysteme und Module, für deren Darstellung Klassen- und Kompositionsstrukturdiagramme herangezogen werden. Das System wird dabei aus `≪Block≫`-Elementen zusammengesetzt, die sowohl für eine BlackBox- wie auch für eine WhiteBox-Sicht auf das System verwendet werden können (Blockdiagramme).

Ports und Datenflüsse repräsentieren Schnittstellen und die ihnen zugeordnete Ströme von Daten, Material oder Energie. Die Ströme können ihre Inhalte sporadisch oder periodisch/kontinuierlich transportieren und werden in Blockdiagrammen eingesetzt, um die Verbindungen zwischen verschiedenen `≪Block≫`-Elementen anzuzeigen.

Randbedingungen und Parameterdiagramme spezifizieren mathematische oder logische Ausdrücke und die zu ihnen gehörigen Parameter, die die physikalischen Eigenschaften des Systems festlegen bzw. begrenzen. Diese Parameter und Randbedingungen können in (domänenspezifischen) Bibliotheken gesammelt und so leicht in anderen Modellen wiederverwendet werden. Die so erfassten Randbedingungen bzw. Zusicherungen können im Systementwurf vor allem während der Analyse- und Designphase eingesetzt werden, um eine physikalisch korrekte Systemimplementierung zu gewährleisten oder die Analyse der Auswirkung von Parameteränderungen zu unterstützen.

Systemverhalten wird in der SysML durch leicht abgewandelte Formen der normalen UML-Verhaltensdiagramme repräsentiert.

Eine rein funktionale Sicht auf das System kann durch Anwendungsfalldiagramme wiedergegeben werden, deren Syntax und Semantik sich nicht von denen der normalen UML unterscheidet.

[3] unter anderem Artisan, Boeing, EADS, Gentleware, I-Logix, IBM und Telelogic
[4] Der Rest des Abschnitts über SysML ist [KL07a] entnommen

Anforderungen werden im Modell als besondere Variante von Klassendiagrammen, den sogenannten Anforderungsdiagrammen, hinterlegt. Diese Diagramme enthalten besonders stereotypierte Klassen mit entsprechenden Parametern („Tags“ oder auch “Tagged Values“; sind eine Eigenschaft des Stereotypen), die eine durchgehende Anforderungsverfolgung über das gesamte Modell hinweg ermöglichen.

Allokationen – also beispielsweise die Zuordnung von Funktionen zu Implementierungselementen – können durch Referenzen beschrieben werden, die entweder in besonderen Tags oder in separaten Kommentarfeldern (sogenannte *callouts*) eingetragen werden.

Durch diese Sprachkonstrukte ist mit SysML eine geeignete Sprache für die Spezifikation, Analyse und das Design komplexer Systeme entwickelt worden. Und obwohl SysML an sich eine domänenspezifische Sprache ist (spezifisch für Systemmodellierung) ist sie nicht spezifisch für ein bestimmtes Anwendungsgebiet wie beispielsweise die Automobil- oder Eisenbahndomäne entworfen worden. Darüber hinaus können mit SysML sowohl Hardware- als auch Softwareaspekte eines Systems beschrieben werden.

Der Umstand, dass SysML nur eine Untermenge der UML verwendet und dadurch leichter und schneller zu erlernen ist, lässt die Sprache als besonders für Ingenieure geeignet erscheinen. Häufig bringen Ingenieure keinen sehr ausgeprägten IT-Hintergrund mit, daher ist das Erlernen der reinen, komplexen UML in vielen Fällen keine Alternative zu traditionellen Systembeschreibungen und Entwicklungen. Hier kann SysML also Abhilfe schaffen.

Die verglichen mit UML geringere Komplexität von SysML kann beispielsweise auch an der Anzahl der verwendeten Diagrammtypen festgemacht werden. Wie oben beschrieben kennt die normale UML 13 unterschiedliche Diagramme, von denen aber nur sieben von SysML genutzt werden. Allerdings definiert SysML auch zwei neue Diagrammtypen – das Anforderungs- und das Parameterdiagramm – von denen das erste allerdings syntaktisch sehr eng am Klassendiagramm angelehnt ist und das zweite von einer sehr geringen Komplexität gekennzeichnet ist.

4.6 Das UML 2 Testing Profile (U2TP)

Ähnlich wie in Form der SysML eine UML-Anpassung für Systems Engineering geschaffen wurde, ist auch ein Sprachprofil für Testbeschreibungen entstanden[5]. Dieser als „UML 2 Testing Profile“ (U2TP) bezeichnete UML-Dialekt definiert auf Basis der UML-Notationselemente und einiger weniger neuer Symbole einen Satz von Diagrammen zur Testfallbeschreibung. Wie für ein UML-Profil üblich, spezifiziert auch das U2TP zu diesem Zwecke verschiedene Stereotypen, Tags und Symbole sowie deren Syntax und semantische Bedeutung. Es vereinbart somit einen Katalog von Elementen, die von allen Benutzern des Profils einheitlich interpretiert werden, und ermöglicht dadurch die eindeutige Beschreibung eines bestimmten Sachverhaltes.

Wie jede andere Systembeschreibung muss auch die Spezifikation von Tests sowohl statische als auch dynamische Aspekte – also Struktur und Verhalten – berücksichtigten. Das U2TP ist dem entsprechend – wie auch Abb. 4.5 illustriert – in folgende Teile gegliedert, auf die in einem späteren Kapitel noch im Einzelnen eingegangen wird:

Test Architecture: Beschreibung des Aufbaus der Testumgebung, Zusammenwirken der einzelnen Komponenten, Konfiguration für verschiedene Testanordnungen

Test Behaviour: Beschreibung des Testverhaltens, der Testschritte, der Sollergebnisse jedes Schrittes und der Reaktionen bei Abweichung der Ergebnisse vom Sollverhalten

[5] Der gesamte Abschnitt wurde größtenteils [KL06] entnommen

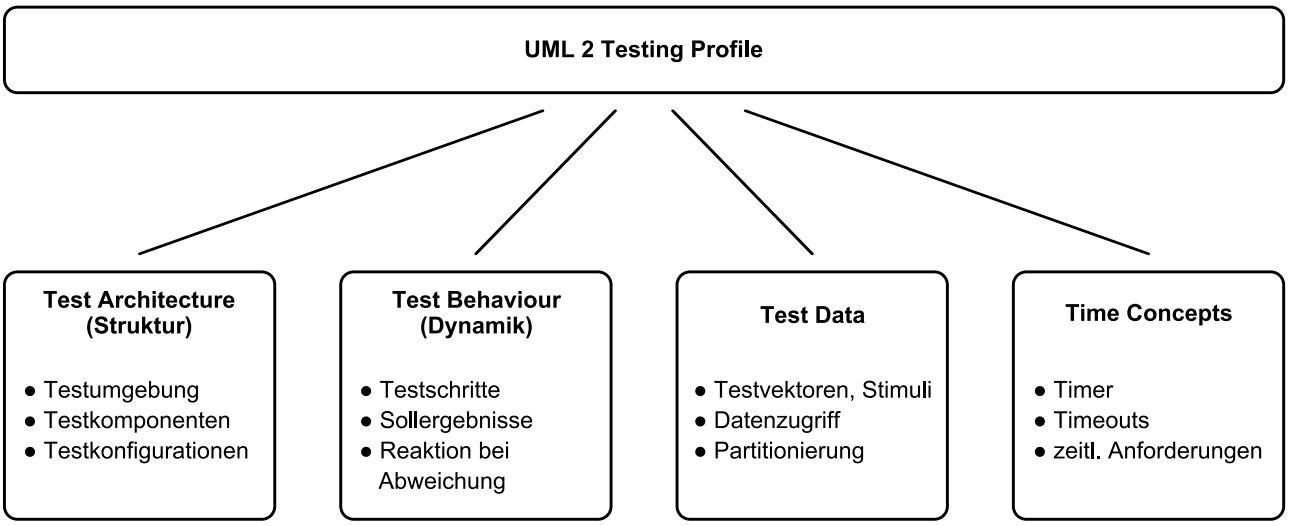

Abbildung 4.5: Bestandteile des UML 2 Testing Profiles

Test Data: Definition von Testdaten, Partitionierung der Testdaten in Untereinheiten, Zugriffsmethoden auf die Testdaten

Time Concepts: Spezifikation von Timern und Timeouts, um zeitliche Anforderungen an den Testablauf beschreiben zu können

Die Konzepte des U2TP beziehen sich rein auf *BlackBox*-Tests und lassen sich auf allen Testebenen vom Modul- bis zum Systemtest anwenden. Bei der U2TP-Spezifikation handelt es sich um eine reine Notations- und Semantikdefinition; sie ist daher werkzeug- und plattformneutral. Ebenso ist die Verwendung des U2TP nicht auf eine bestimmte Domäne beschränkt.

Das U2TP weist eine enge Verwandtschaft mit TTCN-3 auf; im weiteren Sinne kann es sogar als Alternative zur grafischen Notation von TTCN-3 angesehen werden [SDGR03, HW03, SG03]. Eine *generische* Abbildung von U2TP-Elementen auf TTCN-3-Konstrukte ist bereits in der U2TP-Spezifikation enthalten und eröffnet somit eine Möglichkeit, U2TP-Modelle durch Verwendung bestehender TTCN-3-Interpreter zur Ausführung zu bringen. Dies ebnet den Weg zur Testautomatisierung, durch die der Entwicklungsprozess beschleunigt und die Produktqualität erhöht werden kann. Insbesondere sind in diesem Zusammenhang zu nennen:

- Aufwandseinsparung durch automatische Testdurchführung und -auswertung
- reproduzierbarere Resultate dank formaler Testablaufs- und Ergebnisspezifikation; damit einhergehend: bessere Vergleichbarkeit von Tests
- Aufbau von Testfallbibliotheken und Wiederverwendung von Elementen aus anderen Projekten bzw. aus anderen Testfällen des gleichen Projektes

Ein Beispiel, wie eine *konkrete* Abbildung von U2TP-Elementen auf TTCN-3 aussehen kann, wurde im Rahmen dieser Dissertation erarbeitet und wird später in der Fallstudie noch eingehend erläutert. Darüber hinaus kann eine ausführliche Beschreibung des Aufbaus eines TTCN-3-Interpreters und des notwendigen Sprachumfangs für eine automatische Testausführung auch der Arbeit von Janke entnommen werden [Jan07].

Um die oben skizzierten Vorteile einer formalen Testfall- oder Systembeschreibung effektiv zu nutzen, muss sich das Beschreibungsmittel – sei es nun reine UML, SysML, TTCN-3 oder ähnliches – mit dem Ablauf und den besonderen Randbedingungen eines Entwicklungsvorgangs vereinbaren lassen. Daher stellt das folgende Kapitel gängige Entwicklungsprozesse vor und geht auch auf besondere Anforderungen an den Prozess ein, die beispielsweise aus der Entwicklung sicherheitsrelevanter Systeme resultieren.

5 Entwicklungsprozesse

Im vorangegangen Kapitel wurde bereits erläutert, dass ein Beschreibungsmittel alleine noch nicht hinreichend für eine gelungene Systementwicklung bzw. Systemmodellierung ist. Vielmehr bedarf es neben einer geeigneten Werkzeugunterstützung auch einer entsprechenden Methode, also eines zielgerichteten, strukturierten Vorgehens, um ein Beschreibungsmittel effektiv einzusetzen (BMW-Prinzip).

Eine Methode ist ihrerseits Bestandteil eines übergeordneten Vorgehensmodells, das als Entwicklungsprozess bezeichnet wird. Während die Methode nur die Kernaufgaben der Entwicklung behandelt (beispielsweise Analyse, Design und Implementierung), ist der Gültigkeitsbereich des Entwicklungsprozesses weiter gefasst. Er beinhaltet auch Aspekte wie Zulassung, Abnahme oder Dokumentation (vgl. auch Elemente des Mind-Maps auf Seite 17). Der Prozess gibt somit den Rahmen vor, in den die Methode eingebettet ist. Er ist damit wesentlich für die effiziente Anwendung einer Methode bzw. des gesamten BMW-Prinzips verantwortlich.

Aus diesem Grund sollen in den folgenden Abschnitten häufig anzutreffende Entwicklungsprozesse eingeführt und erläutert werden. Ein besonderer Schwerpunkt wird dabei auf dem weit verbreiteten V-Modell liegen, dessen Eignung für objekt-orientierte Entwicklung genauer betrachtet wird.

Neben weiteren Vorgehensmodellen wie dem Spiral- und dem Wasserfallmodell werden darüber hinaus auch der Zusammenhang zwischen Entwicklungsprozess und Zulassungsfragen sowie die Rolle des Anforderungsmanagements Gegenstand dieses Kapitels sein.

5.1 Einführung

Abstrakt betrachtet bedeutet „Systementwicklung“ die Überführung initialer (Kunden-)Anforderungen in ein lauffähiges System, das eben diesen Anforderungen genügt. Die Vorgehensweise bzw. der Ablauf, der dieser Überführung zugrunde liegt, wird durch den Entwicklungsprozess beschrieben.

Ein „Prozess“ ist nach [DIN05] definiert als „Satz von in Wechselbeziehung oder Wechselwirkung stehenden Tätigkeiten, der Eingaben in Ergebnisse umwandelt“, wobei die Norm darauf hinweist, dass Eingaben eines Prozesses häufig Ergebnisse eines anderen Prozesses sind. Ähnlich äußert sich [ISO95]; dort wird ein Prozess als „A set of interrelated activities, which transform inputs into outputs“ bezeichnet.

Als Prozess wird also ein ergebnisorientiertes, zielgerichtetes Handeln verstanden, dass aus einer Anzahl zumeist sequentieller Arbeitsschritte besteht. Durch die Festlegung der Art und Ausführung dieser Arbeitsschritte in einer *Prozessbeschreibung* kann das Vorgehen in einer Organisation vereinheitlicht werden, was zu reproduzierbaren Ergebnissen, erhöhter Qualität und – durch Vermeidung von Doppelarbeit und unnötiger Methodenvielfalt – auch zu Kostensenkungen führt.

Die Tätigkeit eines Unternehmens umfasst eine Vielzahl mehr oder weniger stark miteinander verknüpfter Prozesse, die „systematisch erkannt und gehandhabt“ werden müssen [DIN05]. Zu den häufigsten in produzierenden bzw. entwickelnden Unternehmen beschriebenen Prozessen gehören die Entwicklung, das Projektmanagement, das Produktmanagement, das Qualitätsmanagement und das Personalmanagement.

Da mit der Definition und Beschreibung von Prozessen immer eine Vereinheitlichung von Abläufen und damit eine Einschränkung des Handlungsspielraums der Mitarbeiter einhergeht, ist es wichtig, einen angemessenen Umfang und eine angemessene Granularität der Prozessbeschreibung zu finden. Bei zu schwachen Ablaufbeschreibungen bleibt der erwünschte Effekt aus, während zu restriktive oder detaillierte Vorschriften

zu Ablehnung bei den Mitarbeitern führen oder nur suboptimale Lösungen und Entwicklungsergebnisse zulassen. Der anzustrebende Mittelweg ist besonders dann sehr schmal, wenn kreative Vorgänge – beispielsweise die Entwicklung – reguliert werden sollen. Ein Prozess sollte daher nach Möglichkeit nur einen Handlungsrahmen definieren und ausreichende Flexibilität aufweisen, um bei Bedarf auch abweichende Vorgehensweisen zuzulassen.

Neben dem bereits erwähnten Qualitätssicherungsaspekt bedingen Prozessbeschreibungen vor allem eine klare Abgrenzung von Verantwortlichkeiten und Aufgaben und ermöglichen so erst die effektive Zusammenarbeit verschiedener Teams in größeren Unternehmen und Projekten. Darüber hinaus müssen Prozesse in verschiedenen Domänen bestimmten Kriterien genügen, um eine Marktzulassung des entwickelten Produktes zu erlangen.

Der Entwicklungsprozess befindet sich also in einem Spannungsfeld aus

- Normativen bzw. qualitätssichernden Anforderungen (im Eisenbahnbereich besonders [DIN99, DIN01, DIN03] sowie die Normenreihe ISO 900x)
- Forderungen des Managements nach einem effektiven Prozess
- Forderungen der Mitarbeiter nach möglichst wenig Einschränkungen
- der Entwicklungsmethodik (objektorientiert vs. „klassisch“)
- Schnittstellen zu anderen Prozessen
- Schnittstellen zum Kunden
- Besonderen Eigenschaften des Unternehmens oder der Domäne

In den vergangenen Jahren wurden zahlreiche Ansätze entwickelt, um einen Prozess zu gestalten, der den genannten Anforderungen möglichst gerecht wird. Einige dieser Vorgehensmodelle sollen in den folgenden Abschnitten genauer vorgestellt werden, wobei besonders ausführlich auf das V-Modell eingegangen werden soll. Andere Ansätze wie Wasserfallmodell, Spiralmodell und „Agile Programming“ sollen dagegen nur kurz beleuchtet werden.

5.2 Das V-Modell

5.2.1 V-Modell 97 und V-Modell XT: Ein Überblick

Das V-Modell ist ein Ende der achtziger bzw. Anfang der neunziger Jahre im Auftrag des Bundesverteidigungsministeriums entwickeltes Vorgehensmodell für die Softwareentwicklung komplexer Systeme. Nach seiner Einführung im Militärbereich im Jahre 1991 wurde auch eine Variante für die zivile Nutzung erstellt, die im Sommer 1992 vom Bundesinnenministerium eingeführt wurde und seit 1996 den Status einer verbindlich einzusetzenden Vorschrift hat [ME99].

Im Zuge einer kontinuierlichen Verbesserung hat sich die Prozessbeschreibung vom „V-Modell 92“ über das „V-Modell 97“ zum seit Februar 2005 gültigen “V-Modell XT„ weiterentwickelt. Die vollständige Dokumentation des V-Modells sowie einige Werkzeuge zur Unterstützung der Anwender sind frei verfügbar und dürfen ohne Lizenzkosten verwendet und angepasst werden [Koo05].

Das V-Modell beschreibt, *was* im Rahmen des Entwicklungsprojekts *wann*, *wie* und *womit* getan werden muss und geht dadurch sehr viel weiter als andere Verhaltensmodelle, die sich für gewöhnlich nur auf das „was“ beschränken.

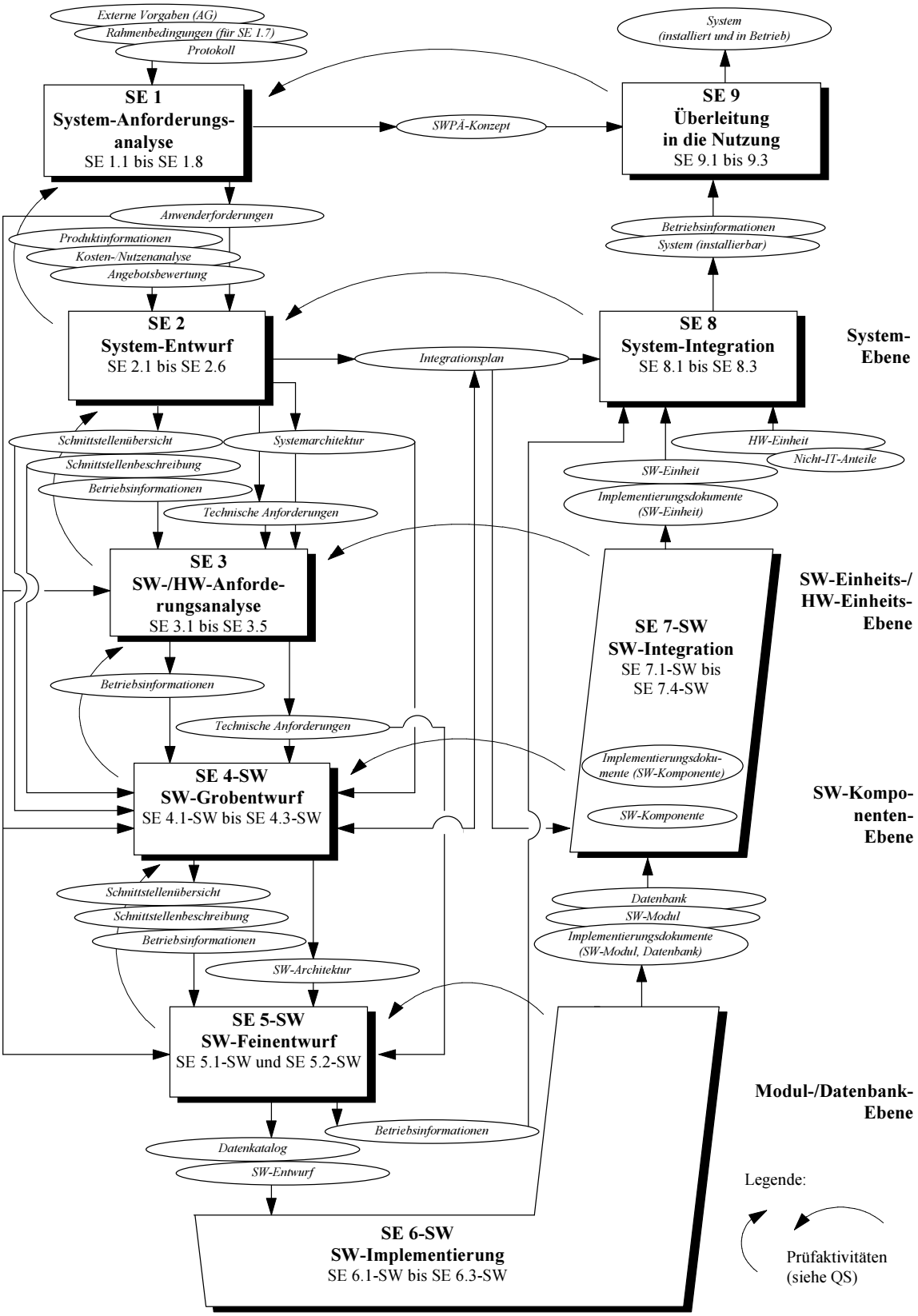

Abbildung 5.1: Der klassische V-förmige Entwicklungsablauf, der bis einschließlich des „V-Modell 97“ verwendet wurde (Quelle: [Koo97])

Der Name „V-Modell“ erklärt sich einerseits aus der Abkürzung für „Vorgehensmodell“ und anderseits auch durch den üblicherweise V-förmig dargestellten Entwicklungsablauf, wie er in Abbildung 5.1 wiedergegeben ist. Charakteristisch ist die Aufteilung des Prozesses in den linken Ast (Entwicklung) und den rechten Ast (Integration und Test). Die Basis des Vs ist durch die Implementierung der kleinsten, „atomaren“ Softwareeinheit, des Moduls, gekennzeichnet. Dieser fixe Ablauf war bis einschließlich des V-Modells 97 zentrales Element der Prozessbeschreibung und hat Eingang in unzählige firmenspezifische Entwicklungsprozesse gefunden.

Mit der Veröffentlichung des V-Modells XT wurde dieser starre, *ablaufzentrierte* Ansatz zugunsten eines *produktzentrierten* Ansatzes aufgeweicht. Das V-Modell XT stellt eine Reihe von „Vorgehensbausteinen“ zur Verfügung, aus denen je nach Projekttyp eine Untermenge für ein konkretes Projekt ausgewählt werden kann. Zu jedem Vorgehensbaustein gehört ein Satz von Aktivitäten, die in *Produkten* resultieren, aber auch Produkte als Eingabe benötigen können. Produkte im Sinne des V-Modells können Dokumente, Hardware, Software oder ähnliches sein[1].

Die Abarbeitungsfolge der Vorgehensbausteine wird ebenfalls durch den Projekttyp determiniert und ist als Folge von „Entscheidungspunkten“ gegeben, die den Projektfortschritt dokumentieren. Als zentrales Kriterium, ob das Projekt an einem Entscheidungspunkt (vergleichbar mit einem Meilenstein) fortschreitet, ist das Vorliegen der im Entscheidungspunkt geforderten Produkte. Der Prozess läuft dadurch produktgesteuert ab. Die vielfältigen Beziehungen zwischen Produkttyp, Vorgehensbausteinen, Aktivitäten etc. sind in der Spezifikation des V-Modells XT [Koo05] ausführlich beschrieben und in Abbildung 5.2 übersichtsartig als UML-Klassendiagramm wiedergegeben.

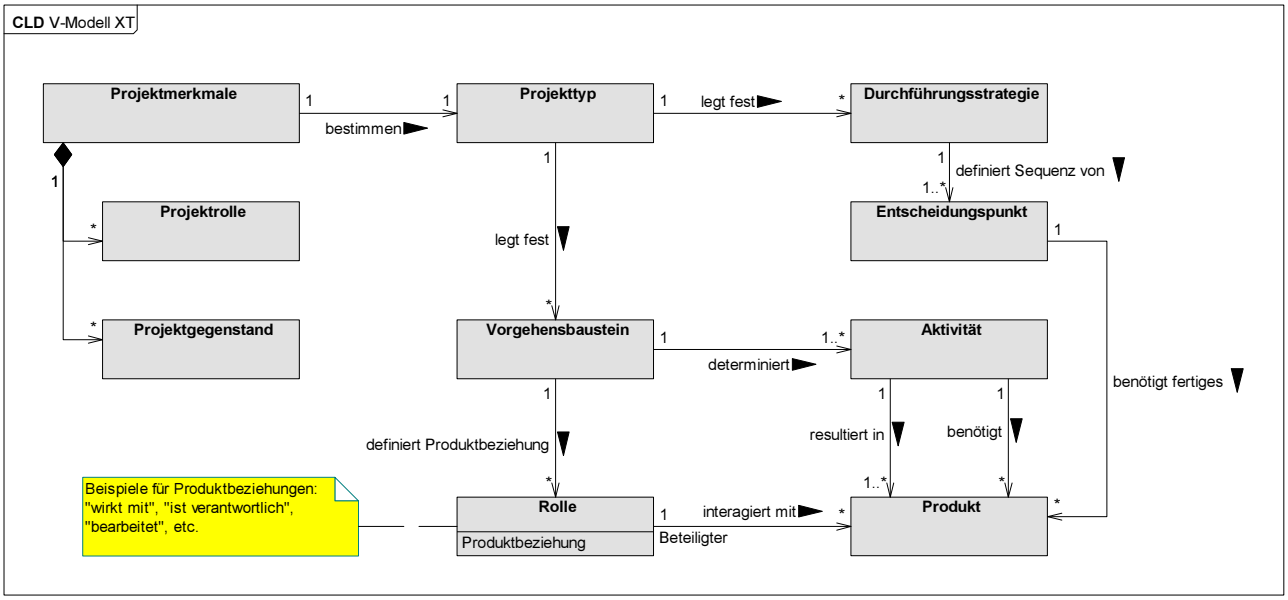

Abbildung 5.2: Die wichtigsten Elemente des „V-Modell XT“ und ihre gegenseitigen Beziehungen als UML-Klassendiagramm

Hinsichtlich der Projekttypen wird zwischen der auftraggeber- und auftragnehmerseitigen Sicht auf eine Systementwicklung unterschieden. Legen also beide Vertragsparteien das V-Modell zugrunde, laufen zwei parallele V-Prozesse ab, die beide auf dem gleichen Vorgehensmodell beruhen, aber unterschiedliche Ausrichtungen haben. Dadurch können bei allen Beteiligten optimal zugeschnittene Varianten des V-Modells eingesetzt werden.

[1] Das V-Modell definiert sich selbst als „Leitfaden zum Planen und Durchführen von Projekten“ [Koo05] und ist somit nicht auf Softwareentwicklung oder technische Systeme beschränkt.

Im Gegensatz zum V-Modell 97 unterstützt die XT-Version darüber hinaus auch neue Entwicklungsmethoden wie beispielsweise agile Techniken [Bec00, WRL05] oder die komponentenbasierte Systementwicklung [Göh98, May05].

Neben der Differenzierung zwischen Auftraggeber- / Auftragnehmersicht und der Berücksichtigung moderner Methoden ist eine dritte wesentliche Eigenschaft des V-Modells XT die vielseitige Anpassbarkeit (engl. *Tailoring*) an Projektarten und Projektgrößen. Durch das optionale Hinzufügen oder Weglassen einzelner Vorgehensbausteine und (Zwischen-)Produkte kann der auf den ersten Blick sehr umfangreiche Prozess auch für kleinere Projekte sinnvoll zurechtgestutzt und eingesetzt werden. Diese Flexibilität drückt sich auch im Namen aus: „XT“ steht für *eXtreme Tailoring*.

5.2.2 Iterativer, objektorientierter Entwurf mit dem V-Modell

Komplexe Softwaresysteme können nur über Zwischenstationen, wie beispielsweise Analyse, Grobentwurf, Feinentwurf und spätere schrittweise Integration, erstellt werden. Dieser Umstand spiegelt sich auch im „traditionellen“ V-Modell aus Abbildung 5.1 wieder, das aufgrund seines Alters noch immer weiter verbreitet ist als sein 2005 verabschiedeter Nachfolger V-Modell XT.

Vielfach werden diese Schritte im Laufe einer Entwicklung aber nur *einmalig* durchlaufen, was zu verschiedenen Problemen führt:

- Anforderungsänderungen oder -verfeinerungen können im laufenden Prozess nur schwer berücksichtigt werden.
- Die Integration der entwickelten Komponenten findet erst relativ spät statt.
- Die Kundeninteraktion erfolgt vor allem am Anfang und Ende des Prozesses.

Anforderungsänderungen und eine zu starke Einbindung des Kunden beinhalten zwar immer auch das Risiko sich ständig ändernder Projektziele, allerdings erhöht eine zu geringe Kundeninteraktion auch die Wahrscheinlichkeit, ein Produkt zu entwickeln, das nicht dem Kundenwunsch entspricht. Tritt diese Erkenntnis und die daraus resultierenden Änderungen erst gegen Ende des Projektes auf, steigen die Kosten der Fehlerbeseitigung erheblich (Abb. 5.3).

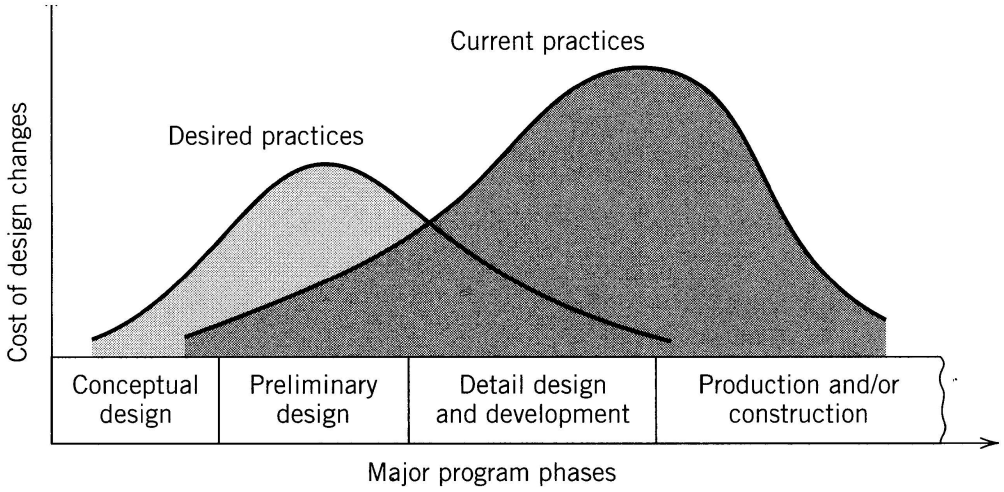

Abbildung 5.3: Auswirkungen spät im Entwicklungsprozess eingebrachter Änderungen (Quelle: [Bla04])

Anstelle eines „großen“ Vs, das einmalig durchlaufen wird, empfiehlt sich daher die Aneinanderreihung mehrerer „kleiner“ Vs, die zu einer kontinuierlichen Verfeinerung des Gesamtsystems führen und nach jedem

Schritt ein ausführbares Testsystem als Zwischenlösung hervorbringen. Dieses Vorgehen wird als inkrementelle Entwicklung bezeichnet und stellt eine der geeignetsten Techniken zur (objektorientierten) Softwareentwicklung dar [ME99, Bal01].

Obwohl bereits mit dem V-Modell 97 die inkrementelle Softwareentwicklung möglich war [DHM98], wird es doch vielfach als ein monolithischer Prozess durchgeführt, was zu den oben genannten Schwierigkeiten führen kann. Das V-Modell XT betont die inkrementelle Entwicklung noch stärker und führt sogar einen eigenen Entscheidungspunkt („Iteration geplant") dafür ein, der zum V-Modell-Kern gehört und damit Teil jedes V-Modell-Projektes ist.

Neben der Einführung iterativer Vorgehensweisen ist vor allem der Einsatz von Modellierungstechniken ein geeigneter Weg, die in Abbildung 5.3 skizzierten Folgen später Fehlererkennung zu vermeiden. Dabei hilft die Modellierung auf mehreren Ebenen:

- Die Anforderungsmodellierung, also die formalisierte bzw. strukturierte Notation von Anforderungen und ihren Wechselbeziehungen ermöglicht eine präzisere Abschätzung der Folgen von Anforderungsänderungen. Sind darüber hinaus Verknüpfungen zwischen Anforderungen, Implementierung und Testfall realisiert (näheres dazu im Folgekapitel), sind jederzeit Aussagen über den aktuellen Entwicklungsstand und die Testabdeckung möglich.

- Die semiformale Modellierung von Systemstruktur und -verhalten schafft bei allen Projektbeteiligten eine einheitliche Vorstellung über das Entwicklungsziel, hilft Architekturfehler zu vermeiden, schafft klare Modularisierung und Schnittstellen und ermöglicht eine einfache Wiederverwendung von Systemkomponenten. Sie kann außerdem als Grundlage für automatische Codeerzeugung dienen.

- Die formale Erfassung des Systemverhaltens ermöglicht vollständige Codeerzeugung für das System und eine formale Verifikation des Systemverhaltens. Die mathematische Beschreibung des Verhaltens ist allerdings sehr aufwendig.

Alle diese genannten Punkte haben zum Ziel, möglichst früh im Prozess möglichst viel Systemverständnis und möglichst testbare Artefakte hervorzubringen, um von Beginn an die Fehlerrate zu minimieren.

Im Rahmen der Analyse eines realen, auf dem V-Modell 97 basierendem, monolithischen Entwicklungsprozesses wurde gezeigt, dass die Verknüpfung von modellbasierter und inkrementeller Entwicklung möglich ist und zu positiven Ergebnissen führt [Uht06]. Im vorliegenden Fall wurde sogar ein an sich monolithischer V-Prozess so „verbogen", dass eine inkrementelle Entwicklung in ihn eingebettet werden konnte.

Die inkrementelle Entwicklung – also die Aneinanderreihung von Vs – darf dabei nicht verwechselt werden mit der hierarchischen Schachtelung V-förmiger Prozesse für einzelne Systemkomponenten, wie sie in Abbildung 5.4 am Beispiel eines ETCS-Fahrzeuggerätes dargestellt ist. Das V des Gesamtsystems zerfällt dabei in mehrere Vs für die beteiligten Subsysteme Onboard Unit (OBU), Driver Machine Interface (DMI), Juridical Recorder (JRU), Specific Transmission Module (STM), Zugsteuergerät (ZSG) und die Anbindung nationaler Leit- und Sicherungstechnik (LST) wie beispielsweise die punkt- und linienförmige Zugbeeinflussung (PZB / LZB). Abbildung 5.4 fokussiert dabei auf die OBU-Entwicklung, die wiederum in einen Hard- und einen Softwareteil zerfällt. Die Softwareentwicklung mündet schließlich in die Implementierung und den Test verschiedener Module, die anschließend zur OBU-Software integriert werden.

Je nach Komplexität des zu entwickelnden Systems ergibt sich somit eine beinahe fraktal anmutende Verschachtelung von V-Prozessen auf unterschiedlichster Ebene, die aber alle nur *einmalig* durchlaufen werden und somit nicht als inkrementelle Entwicklung anzusehen sind. Entsprechend schwierig gestaltet sich die Einbettung objekt-orientierter formaler Methoden in eine solche Umgebung.

Die Anwendung der in den nachfolgenden Kapiteln vorgestellten Modellierungstechniken sollte daher vor dem Hintergrund einer inkrementellen Entwicklung gesehen werden, die zu einer schrittweisen Verfeinerung der Modelle und zu einer schrittweisen Zunahme des Funktionsumfangs führt.

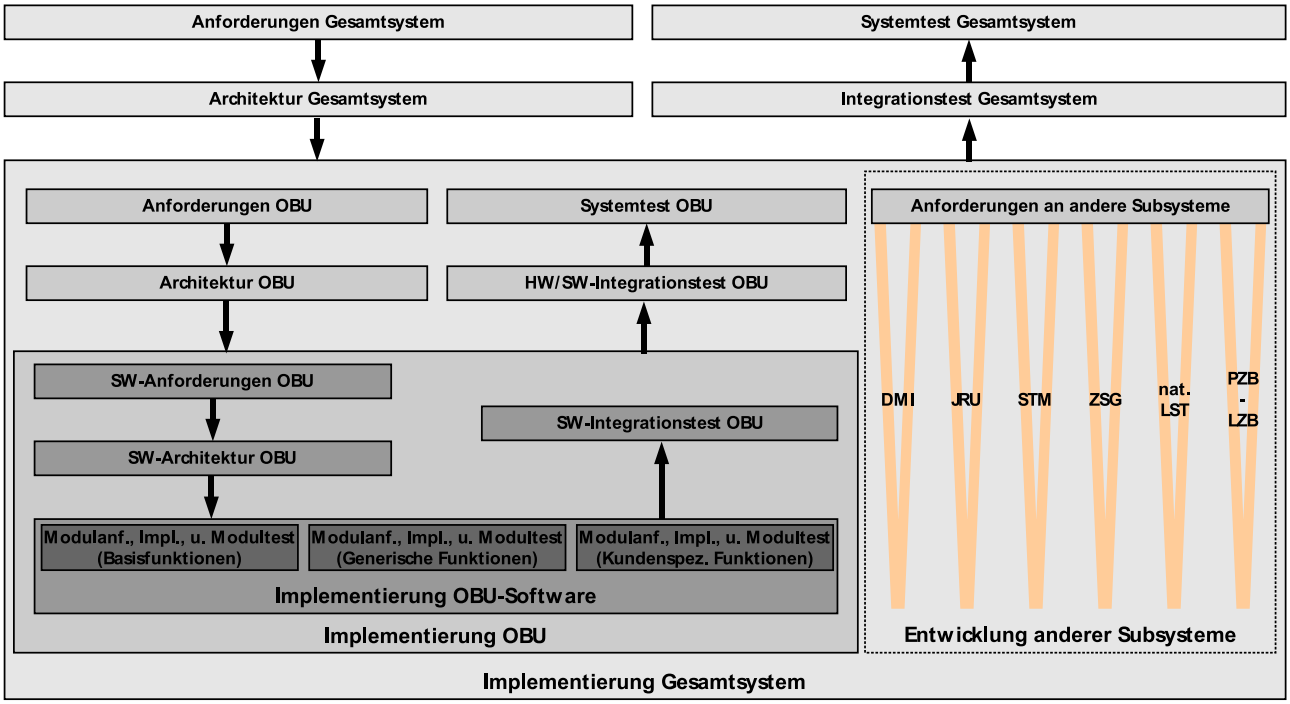

Abbildung 5.4: Verschachtelung mehrerer V-Prozesse für die Komponenten eines ETCS-Fahrzeuggerätes

5.3 Weitere Prozesse und Vorgehensmodelle

5.3.1 Das Wasserfallmodell

Das Wasserfallmodell ist das wohl älteste Vorgehensmodell und wurde 1970 von W. W. Royce vorgeschlagen [Roy87]. Obwohl Royce ursprünglich einen iterativen Prozess im Blick hatte, wurde der Wasserfallprozess zum Synonym für die lineare, sequentielle Systementwicklung von den Anforderungen über die Implementierung hin zur Inbetriebnahme ohne die Möglichkeit, vorangegangene Phasen zu wiederholen (Abb. 5.5).

In der Praxis zeigte sich, dass die Forderung des Wasserfallmodells nach vollständig zu Projektbeginn vorhandenen und „eingefrorenen“ Anforderungen in der Praxis häufig nicht erfüllbar ist. Darüber hinaus erwiesen sich auch die fehlenden Rücksprungsmöglichkeiten in vorangegangene Phasen als problematisch, weil dadurch die Fehlerbeseitigung und das Einfließen von Erfahrungen erschwert wurde. Häufig führte dies zu einer „doppelgesichtigen Softwareentwicklung“ (Zitat [ME99]), in der auf dem Papier eine wasserfallförmige Entwicklung dokumentiert wurde, während de facto andere, zumeist iterative Entwicklungsverfahren zur Anwendung kamen.

Insgesamt suggeriert der Prozess durch seine strikte Phasentrennung und seinen einfachen Aufbau zwar eine gute Steuerbarkeit, allerdings hat er sich nur für kleinere Projekte als anwendbar erwiesen, da mit ihm die Probleme komplexer (Software-)Entwicklungen nicht beherrschbar sind[2].

5.3.2 Das Spiralmodell

Im Jahr 1988 veröffentlichte Barry W. Boehm eine iterative Weiterentwicklung des V-Modells, die er als Spiralmodell bezeichnete [Boe88]. Abbildung 5.6 zeigt das wiederholte Durchlaufen von Anforderungs-, Analyse- und Implementierungsphasen, die schließlich ein einem fertigen Produkt münden (die Vorversionen werden

[2] Interessanterweise hat eine Umfrage unter großen und kleinen Softwareherstellern im Jahre 2003 ergeben, dass viele Unternehmen noch immer das Wasserfallmodell einsetzen, wenn auch „versteckt“ [NL03, LN04].

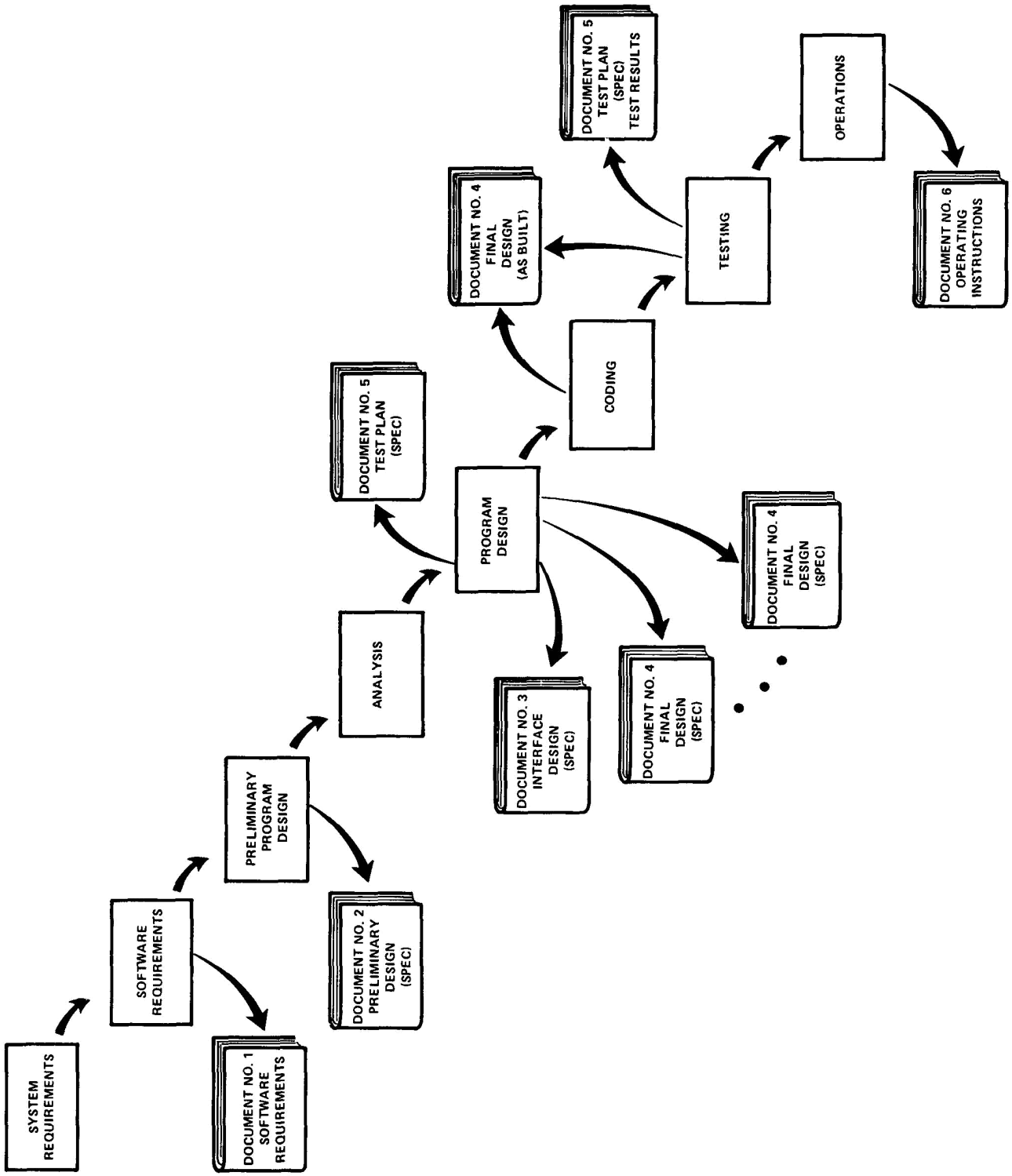

Abbildung 5.5: Das Wasserfallmodell als Originalabbildung aus [Roy87].

als Prototypen bezeichnet). Die Spiral- bzw. Quadrantendarstellung des Vorgehensmodells ermöglicht die Erfassung der Gesamtkosten einer Phase als radiale Information, während die Art der Phase aus dem Winkel eines Punktes im Koordinatensystem abzulesen ist.

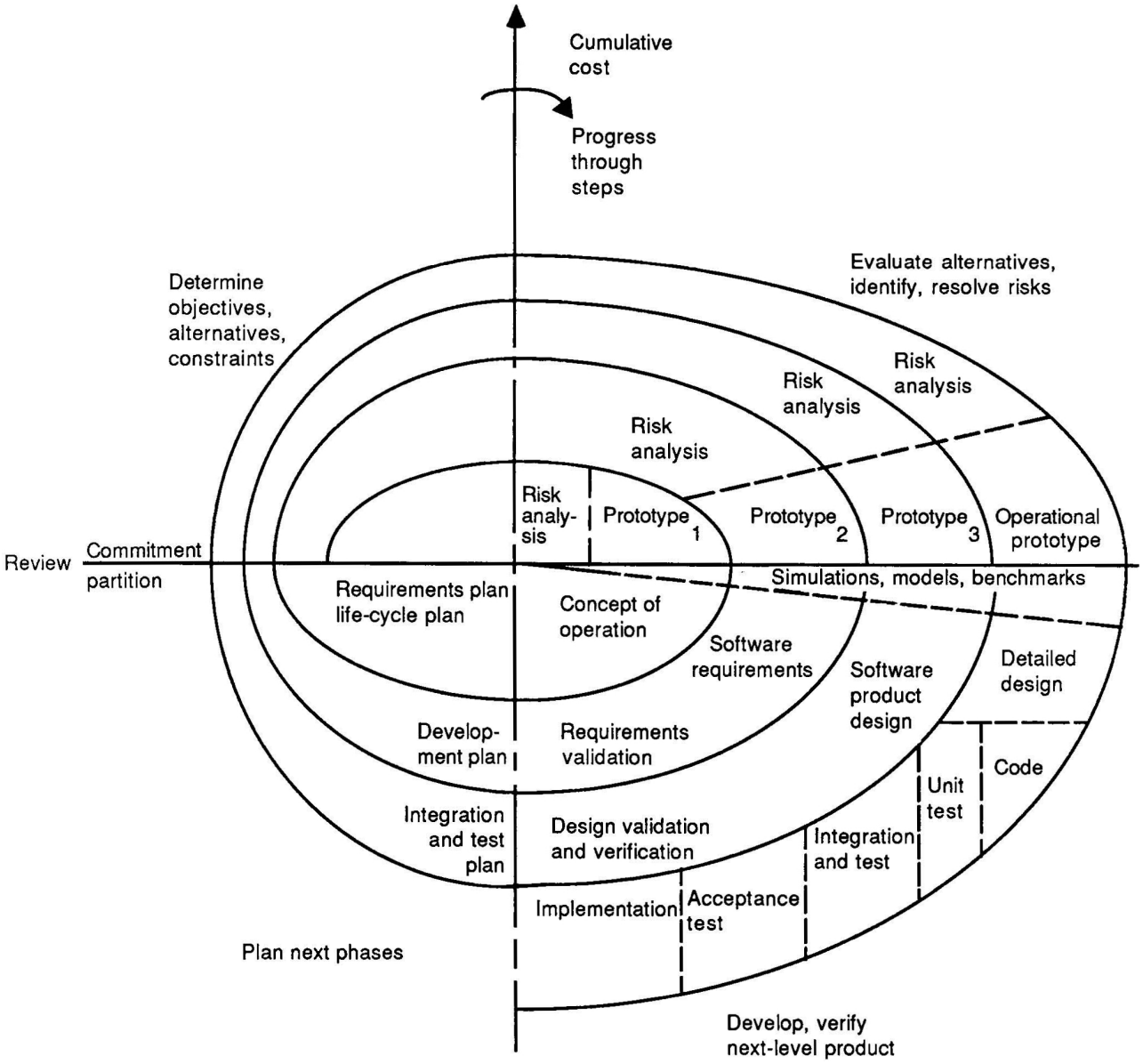

Abbildung 5.6: Das Spiralmodell nach Barry W. Boehm (Quelle: [Boe88])

Neben der wiederholten Ausführung der grundlegenden Wasserfallphasen führt Boehm vor allem eine intensive Risikoanalyse in den Prozess ein, die die Ziele der nachfolgenden Iteration bestimmt. Das Vorgehen ist also insgesamt *risikogetrieben*.

Aufgrund seiner Komplexität und des administrativen Aufwands ist das Spiralmodell in seiner ursprünglichen Form vor allem für große Projekte einsetzbar. Durch Variation der Art der aufeinanderfolgenden Phasen und der Anzahl der Iterationen kann das Spiralmodell jedoch sehr flexibel an andere Bedingungen angepasst werden.

5.3.3 Agile Prozesse

Die bisher vorgestellten Prozesse V-, Wasserfall- und Spiralmodell haben gemeinsam, dass sie vor der Implementierung eine lange Analysephase durchlaufen. In dieser Phase entsteht eine Vielzahl von Anforderungs-, Planungs- und Architekturdokumenten (siehe beispielsweise Abb. 2.6 auf Seite 17 für eine Auswahl der in einem solchen „schwergewichtigen“ Prozess anfallenden Dokumente), aber eben kein Programm. Dieses Phänomen ist auch als „WISCY“-Syndrom bekannt [Wel94], was für „Why isn't Sam coding yet?“ steht.

Die Bemühungen um andere, schlankere Vorgehensmodelle brachten die sogenannten „Agilen Prozesse“ hervor, die sich auf eine stark entbürokratisierte Softwareentwicklung mit wenigen Regeln konzentrieren. Einer der bekanntesten Vertreter dieser Verfahren ist das „eXtreme Programming“ (XP), das vor allem durch Kent Beck bekannt wurde [Bec00, Bec99].

Die agile Entwicklung ist durch stark iteratives bzw. inkrementelles Vorgehen geprägt und setzt auf kontinuierliche, intensive Kommunikation mit allen an der Entwicklung beteiligten Personen (Entwickler, Partner, Auftraggeber). Durch kurze Entwicklungszyklen (vgl. Abb. 5.7) soll der Kunde kontinuierlich über den aktuellen Projektstand informiert bleiben und durch unmittelbares Feedback sollen Anforderungsänderungen oder Fehlentwicklungen sofort behandelt werden.

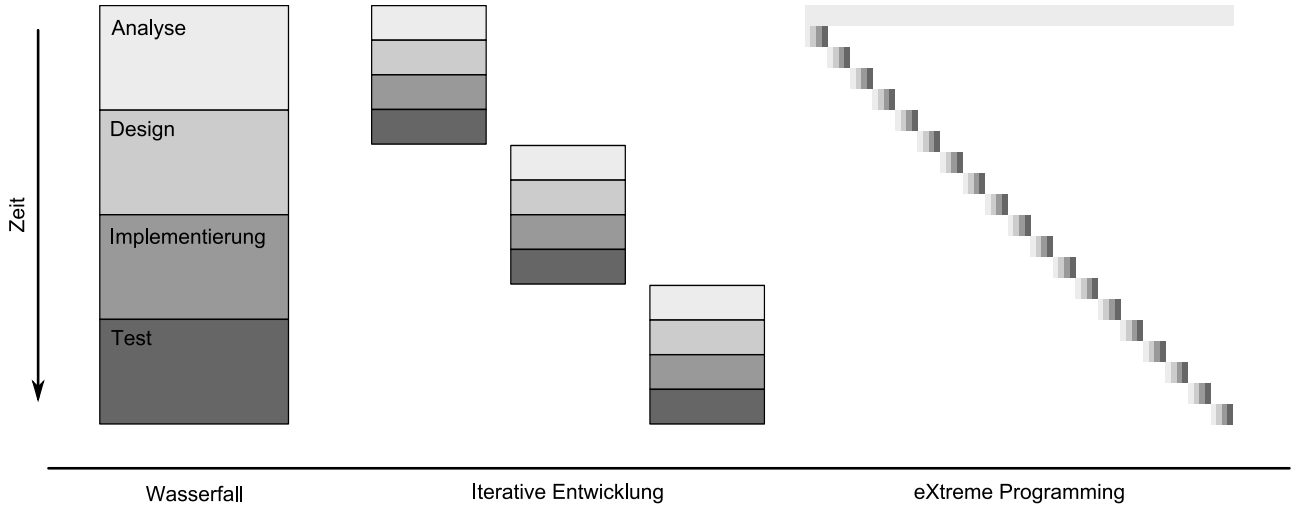

Abbildung 5.7: Weiterentwicklung des Wasserfallmodells mit seinen langen Entwicklungsphasen über iterative Ansätze (z. B. Spiralmodell) bis hin zum eXtreme Programming, das alle diese Phasen in kurzen Entwicklungsschritten vereinigt und im gesamten Entwicklungsprozess immer wiederholt (Quelle: [Bec99]).

Neben rein technischen oder organisatorischen Aspekten werden auch noch soziale bzw. kulturelle Werte von der agilen Entwicklung berührt. Beispiele dafür sind die von allen Beteiligten einzuhaltende 40-Stunden-Woche und der kollektive Code-Besitz (jeder darf jeden Teil des Sourcecodes verbessern bzw. anpassen). Und auch aus methodischer Sicht sind einige zum Teil neuartige Ideen vertreten:

Paarprogrammierung: Zwei Programmierer teilen sich einen Arbeitsplatz. Einer programmiert, der andere redigiert, macht Vorschläge und arbeitet am Konzept mit. Die höheren Aufwände sollen durch den daraus resultierenden besseren Code und die bessere Verbreitung des Code-Wissens (die Zweierteams werden ständig neu zusammengestellt) über das Gesamtprojekt mehr als ausgeglichen werden.

Testgetriebene Entwicklung: Der Programmierer schreibt zuerst den Testfall (Unittest, genaueres dazu im folgenden Kapitel) und erst danach den eigentlichen Code. Alle Tests werden gesammelt und bei Fertigstellung eines jeden Inkrements vollständig ausgeführt (Regressionstest). Ein weit verbreitetes Werkzeug, dass dieses Vorgehen unterstützt, ist JUnit, das wesentlich von Kent Beck mitentwickelt wurde. Dieser Ansatz ist auch als „Test-First“-Verfahren oder „Test Driven Development“ bekannt [Bec02].

Refactoring: Unter Refactoring versteht man, die Codebasis bestehender Programme zu verbessern, ohne die Funktion bzw. das Verhalten des Programms zu verändern. Im sogenannten „kleinen Refactoring“ wird dabei nur auf Codeebene gearbeitet und beispielsweise Funktionsaufrufe oder `if`-Bedingungen entzerrt. Das „große Refactoring“ geht einen Schritt weiter und verändert auch die Softwarearchitektur. Wichtig beim Refactoring ist, dass die durchgeführten Verbesserungen auch zu Änderungen in der Entwicklung führen, um die alten Fehler nicht zu wiederholen. Das Refactoring wurde wesentlich durch Martin Fowler [Fow05] unter Mitarbeit von Kent Beck geprägt.

Allen diesen Ansätzen liegt die Idee zugrunde, dass komplexe Software nicht im Voraus bis ins Detail geplant werden kann, da sich die detaillierten Anforderungen erst mit dem Programm entwickeln. Daher sollen die Prozesse die unvermeidlichen Änderungsanfragen möglichst gut abfangen und behandeln können anstatt zu versuchen, Änderungen zu umgehen oder zu vermeiden. Dies geschieht nicht zuletzt durch den Einsatz objektorientierter Programmiersprachen und deren Unterstützung durch Modellierungstechniken. Ein Beispiel für die Anwendung modellbasierter Techniken in einem agilen Umfeld zeigt Rumpe, der mit UML/P ein auf agile Entwicklungsprozesse zugeschnittenes UML-Profil einführt [Rum05].

Insgesamt werden agile Prozesse erfolgreich eingesetzt[3] und erfreuen sich zunehmender Beliebtheit. Vor allem kommen sie der „intuitiven“ Arbeitsweise von Softwareingenieuren und Programmieren entgegen und tragen dadurch nicht unerheblich zur Arbeitsmoral bei. Allerdings ist ihr hochdynamischer Ansatz nicht für alle Arten von Entwicklung geeignet. Im Bereich sicherheitsrelevanter Systeme ist beispielsweise eine umfangreiche Dokumentation, die wesentlich über die Dokumentation innerhalb einer agilen Entwicklung hinausgeht, Grundlage für eine Zulassung. Daher stellen zulassungspflichtige Systeme besondere Anforderungen an den Entwicklungsprozess, auf die im folgenden Abschnitt eingegangen wird.

5.4 Systementwicklung und Zulassung

Die Tatsache, dass Ausfall oder Fehlverhalten von Systemen mit Sicherheitsverantwortung zu einer Gefährdung von Mensch, Umwelt oder Material führen kann, verlangt bei der Entwicklung derartiger Geräte besondere Sorgfalt. Diese Sorgfalt ist durch ausführliche Dokumentation zu belegen, wobei diese Dokumentation zusammen mit dem entwickelten System von mehreren unabhängigen Personen begutachtet wird.

Voraussetzung für die Inbetriebnahme eines solchen Systems sind verschiedene Tests und Überprüfungen, die zum Teil behördlich abgenommen werden müssen:

Systemtest: Test des Herstellers, ob das System alle vorgesehen Funktionen korrekt realisiert.

Abnahmetest: Test des Auftraggebers, ob der Hersteller die zugesicherten Systemeigenschaften einhält (CAT, Customer Acceptance Test). Der CAT wird in der Regel mit der finalen Installation im Feld anhand typischer Betriebsszenarien durchgeführt. Vorgelagert ist ein Test beim Hersteller, der durch den Kunden überwacht wird (FAT, Factory Acceptance Test).

Validierung: Überprüfung, ob das System für den vorgesehen Verwendungszweck geeignet ist.

Sicherheitsnachweis: Überprüfung der Systemkonstruktion bzw. -realisierung (Hard- und Software) auf Einhaltung von Sicherheitskriterien. Für Bahnanwendungen wird gemäß DIN-EN 50126 [DIN99] vom Kunden (Betreiber der Anlage) eine *Risikoanalyse* erwartet, in der die maximal zulässigen vom System ausgehenden Gefährdungen ausgewiesen werden. Der Hersteller hat in einer *Gefährdungsanalyse* nachzuweisen, dass sein System den Vorgaben entspricht. Dieser Nachweis wird durch die zulassende Behörde überprüft.

[3] Ein Beispiel dafür sind viele OpenSource-Projekte wie beispielsweise der Linux-Kernel, die ohne aufwändige Planung, Analysen und Management-Overhead sehr effizient arbeiten und hochwertigen Code erzeugen

Zulassung: Nach Begutachtung der Konstruktions-, Test- und Sicherheitsnachweisunterlagen entscheidet die zulassende Behörde über eine Genehmigung der Inbetriebnahme des Systems. Die Genehmigung kann sich auf ein einzelnes Gerät beschränken (sogenannte *Einzelzulassung*) oder für eine ganze Geräteserie gelten (*Bauart-* bzw. *Typzulassung*).

Konformität: Innerhalb der EU müssen Produkte bestimmten Kriterien gehorchen, um ein Mindestmaß an Qualität und Funktionalität zu gewährleisten. Diese Anforderungen sind den TSIs (Technische Spezifikation für Interoperabilität) festgehalten. Die Konformität zur TSI wird durch ein EU-weit einheitliches Konformitätsbewertungsverfahren überprüft, an dem neutrale Stellen beteiligt sind. Es gibt mehrere Möglichkeiten, eine Konformitätserklärung zu erreichen, wie Abbildung 5.8 zeigt. Grundsätzlich beruhen alle Verfahren auf einer Kombination von:

- Einsatz überprüfter Qualitätsmanagementprozesse beim Hersteller
- Prüfung von Baumustern in der Entwurfsphase
- Prüfung von Baumustern in der Produktionsphase
- Überwachung der Produktions- und Entwicklungsprozesse

In der Regel wird die Konformität eines Produktes durch die bekannte CE-Kennzeichnung ausgewiesen. Es gibt allerdings Ausnahmen – unter anderem Hochgeschwindigkeitseisenbahnsysteme (z. B. ETCS). Sie müssen einem Konformitätstest unterzogen werden, sind aber nicht mit einer CE-Kennzeichnung ausgestattet. Die für Eisenbahnanwendungen wichtigen TSIs, die der Konformitätsüberprüfung zugrunde liegen, sind [Eur02b], [Eur02a] und [Eur06].

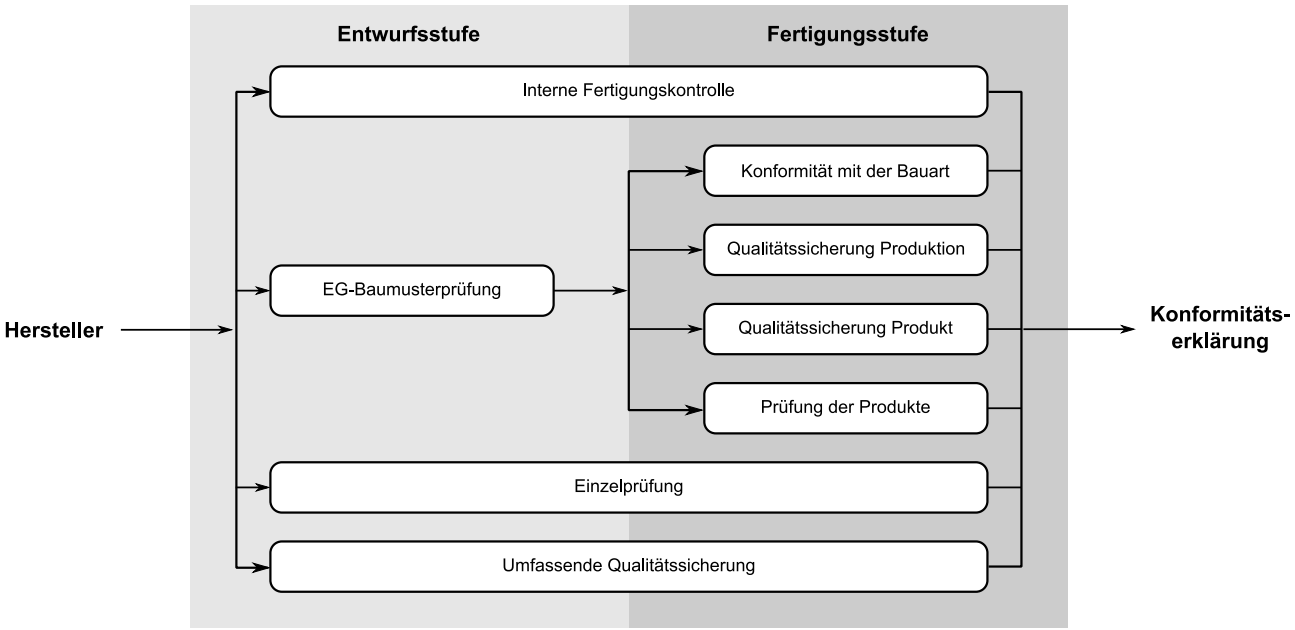

Abbildung 5.8: Verschiedene Wege zu einem EU-Konformitätsnachweis. Die „Interne Fertigungskontrolle“ und der Nachweis der „Konformität mit der Bauart“ bedürfen keiner benannten Stelle. Alle anderen Überprüfungen werden unter Aufsicht einer benannten Stelle durchgeführt.

Der Entwicklungsprozess muss sicherstellen, dass alle für die genannten Überprüfungen notwendigen Unterlagen erzeugt werden. Beispielsweise erfordert [Eur06] für eine Baumusterprüfug die Vorlage folgender Dokumente (Zitat):

- eine allgemeine Beschreibung des Baumusters

- Entwürfe und Fertigungsangaben, beispielsweise Zeichnungen und -pläne von Bauteilen, Unterbaugruppen, Schaltkreisen usw.
- Beschreibungen und Erläuterungen, die zum Verständnis der Entwürfe und Fertigungsangaben sowie zur Instandhaltung und zum Betrieb der Interoperabilitätskomponente erforderlich sind
- Bedingungen für die Integration der Interoperabilitätskomponente in ihre Systemumgebung (Unterbaugruppe, Baugruppe, Teilsystem) und die erforderlichen Schnittstellenbedingungen
- Betriebs- und Instandhaltungsbedingungen der Interoperabilitätskomponente (Betriebsdauer- oder Laufleistungsbeschränkungen, Verschleißgrenzen usw.)
- die technischen Spezifikationen, einschließlich der europäischen Spezifikationen mit einschlägigen Bestimmungen, die ganz oder teilweise angewandt werden
- Beschreibung der zur Erfüllung der TSI gewählten Lösungen, falls die europäischen Spezifikationen nicht vollständig angewandt wurden
- Ergebnisse von Konstruktionsberechnungen, Prüfungen usw.
- Prüfberichte

Das strukturierte Vorgehen bei der Entwicklung des sicherheitsrelevanten Systems, das durch die genannten Unterlagen dokumentiert werden soll, ist am ehesten mit einem V-förmigen Prozess erreichbar. Ein Beispiel für einen solchen aus dem V-Modell abgeleiteten Entwicklungsprozess ist der PEACC+-Prozess der Firma Siemens Transportation Systems, der die CENELEC-konforme Entwicklung [DIN99, DIN01, DIN03] von Eisenbahnleit- und -sicherungstechniksystemen beschreibt [Uht06]. Agile Prozesse erreichen hingegen nicht die notwendige Dokumentationsdichte und können auch weitere Randbedingungen – wie beispielsweise die personelle Trennung von Tester und Programmierer [IEC00] – nicht sicherstellen.

5.5 Anforderungserfassung und -verfolgung

Eine wichtige Tätigkeit innerhalb jedes Entwicklungsprozesses ist die Erfassung und die Verwaltung der Systemanforderungen. Dieser Vorgang wird auch als Requirements Engineering (RE) bezeichnet und von Rupp als „systematischer Weg von der Projektidee über die Ziele zu einem vollständigen Satz von Anforderungen“ definiert [RS04]. Dieser Ansatz – also die möglichst vollständige und detaillierte Erfassung aller Anforderungen vor Umsetzungsbeginn – entstammt eher der Vorgehensweise schwergewichtiger, nicht agiler Entwicklungsprozesse und entspricht den Erfordernissen bei der Entwicklung sicherheitsrelevanter Systeme.

Wichtig bei der Anforderungserfassung ist, die Anforderungen und Wünsche *aller* natürlichen und juristischen bzw. abstrakten Personen und Gruppen (engl. *Stakeholder*) zu erfassen, die Einfluss auf das System haben. Die Menge der Stakeholder kann Anwender, Manager, Käufer, Entwickler, Prüfer/Auditoren, Designer, externe Experten, Controller etc. umfassen, wobei jeder eine andere Sichtweise und andere Prioritäten mit dem zu entwickelnden System verbindet. Im Laufe des RE müssen die Anforderungen dieser verschiedenen Sichten zu einem großen, konsistenten Anforderungskatalog verschmolzen werden.

Besonders wichtig bei der Erstellung dieses Katalogs ist eine klare Formulierung der Anforderungen, da nicht selten schwammige, unpräzise und untestbare Umschreibungen gewählt werden, die im weiteren Verlauf des Projektes für Unklarheit bzw. Falschauslegungen führen können.

Sind die Anforderungen erfasst, setzt das Requirements Management (RM) ein, dass die Umsetzung der Anforderungen und deren Test überwacht. Ziel ist dabei die Verfolgung einer Anforderung zu evtl. von ihr abgeleiteten oder von ihr abhängigen anderen Anforderungen, zu Implementierungselementen und zu den

zugehörigen Testfällen. Das übernächste Kapitel wird darauf eingehen, wie diese Aufgabe mit Unterstützung der UML realisiert werden kann.

Die Behandlung der Anforderungen durchzieht also den gesamten Entwicklungsprozess von der Projektidee bis hin zum Abnahmetest, in dessen Verlauf gegen die ursprünglich aufgestellten Anforderungen getestet wird. Daher ist die enge Verflechtung des RE bzw. RM mit dem Entwicklungsprozess und weiteren unterstützenden Prozessen, wie beispielsweise der Qualitätssicherung, entscheidend. Schließlich ist die Nachverfolgbarkeit und Testfallabdeckung von Anforderungen auch ein wichtiges Kriterium bei der Zulassung sicherheitsrelevanter Systeme und ein qualitativ hochwertiger Anforderungskatalog ist eine wichtige Grundlage für Vertragsverhandlungen, die Systemarchitektur und eventuelles *Claim Management* bei Nachforderungen durch oder an den Kunden.

Die eben erwähnte Testfallabdeckung von Anforderungen und der Abnahmetest des Kunden sind zwei Aspekte des weiten Themenkomplexes „Testen“, auf das im folgenden Kapitel genauer eingegangen werden soll. Es wird ausführlich auf die Grundlagen und Herausforderungen des Testens in verschiedenen Phasen des Entwicklungsprozesses eingehen und unter anderem auch Methoden zur automatisierten Testfallableitung und zur Testbeschreibung mittels UML vorstellen.

6 Tests und Testumgebungen

Testen ist ein wichtiger, integraler Bestandteil eines jeden Entwicklungsprozesses und macht häufig 50 % oder mehr des gesamten Entwicklungsaufwands aus [Rum05]. Aufgrund dieser enormen Bedeutung des Testens für die Entwicklung von (Software-)Systemen soll der Testvorgang in diesem Kapitel im Detail analysiert werden.

Der erste Abschnitt gibt eine Übersicht zu den methodischen Grundlagen des Testens und führt einige Fachbegriffe ein. Der darauf folgende Teil beschreibt die Anforderungen an Testumgebungen, die für die automatische Ausführung von Tests notwendig sind. Exemplarisch wird eine Testumgebung für ETCS-Fahrzeuggeräte vorgestellt und ihre Funktionsweise erläutert.

Die Notation und Modellierung von Tests werden dann im Abschnitt 6.3 genauer betrachtet. Ein Schwerpunkt wird die Modellierung von Tests mit Hilfe des UML 2 Testing Profiles (U2TP) sein, das bereits am Ende von Kapitel 4 kurz vorgestellt wurde.

Abschließend werden verschiedene Methoden zur automatischen Erzeugung von Testfällen kurz vorgestellt und diskutiert. Dabei wird auch der Einfluss solcher Methoden auf den herkömmlichen Test- und Entwicklungsprozess berücksichtigt.

6.1 Übersicht

Myers definiert Testen sehr treffend als

> *Testen ist der Prozeß, ein Programm mit der Absicht auszuführen, Fehler zu finden*

und spricht damit folgende wichtige Punkte an [MSBT04]:

- Testen ist im weitesten Sinne ein destruktiver Vorgang mit dem Ziel der *Falsifizierung*. Aus einem Test, der kein fehlerhaftes oder abweichendes Systemverhalten provozieren konnte, lassen sich keine weitere Rückschlüsse für die Entwicklung ziehen. Auch wird durch einen solchen Test das Vertrauen in den Prüfling nur bedingt erhöht, da korrektes Systemverhalten nur für einen einzigen Satz von Eingabevektoren unter den zum Testzeitpunkt herrschenden weiteren (möglicherweise nicht beeinflussbaren) Randbedingungen nachgewiesen wurde.

- Aufgrund des nahezu unendlich großen Zustandsraumes eines (Software-)Systems und aufgrund der mannigfachen und teilweise unbekannten Verkopplung mit seiner Umwelt (z. B. Beeinflussung durch elektromagnetische Verträglichkeit, EMV) ist es praktisch unmöglich, das Systemverhalten für alle Zustände, Eingangsvektoren und – bei Systemen mit speichernden Eigenschaften – alle Sequenzen von Eingangsvektoren zu überprüfen. Ein vollständiger Systemtest ist also nicht realisierbar.

- Aus den beiden erstgenannten Punkten folgt, dass die Fehlerfreiheit eines Programms nicht nachgewiesen werden kann, da es nicht möglich ist, jeden Systemzustand und das damit verbundene Sollverhalten für eine Überprüfung zu erreichen. Ein Test kann also immer nur die *Anwesenheit* von Fehlern zeigen, nicht deren *Abwesenheit*.

Das oben angesprochene Komplexitätsproblem soll durch ein Beispiel aus [MSBT04] verdeutlicht werden. Abb. 6.1 zeigt ein einfaches Programm, das aus einer äußeren Schleife besteht, die bis zu 20 Mal durchlaufen werden kann. Innerhalb der Schleife gibt es eine Reihe von Verzweigungen, die zu insgesamt fünf unterschiedlichen Pfaden durch den Schleifenkörper führen.

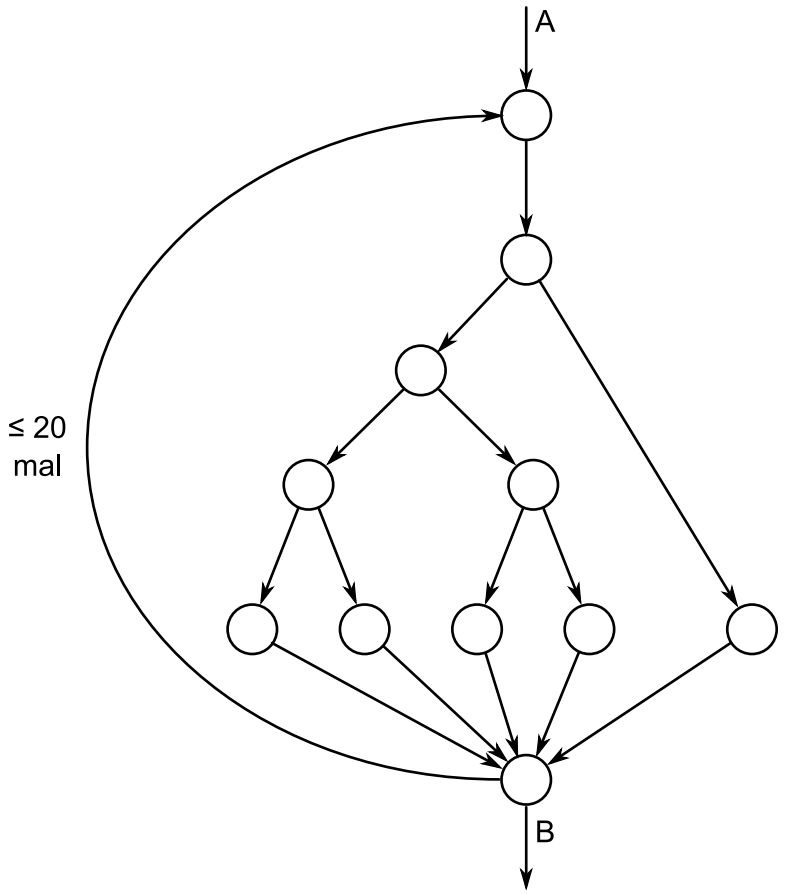

Abbildung 6.1: Beispielprogramm zur Veranschaulichung des Komplexitätsproblems beim Testen (aus [MSBT04])

Bei einem einzelnen Schleifendurchlauf sind also 5^1 Pfade möglich, bei zwei Schleifendurchläufen entsprechend 5^2 usw. Da bis zu 20 Durchläufe möglich sind, müssen alle diese Kombinationen addiert werden: $5^1 + 5^2 + 5^3 + \ldots + 5^{20}$, was zu einer geometrischen Reihe führt und in knapp 120 Billionen möglicher Pfade endet.

Dieses Beispiel zeigt, dass schon bei vergleichsweise einfachen Programmen die Überprüfung aller möglichen Ausführungspfade praktisch unmöglich ist. Durch Berücksichtigung verschiedener Variablenbelegungen, die zu gleichen Ausführungspfaden führen, rein mathematisch unterschiedlichen Programmzuständen entsprechen, wird offensichtlich, dass die Zahl aller denkbaren Programmzustände die Zahl der möglichen Ausführungspfade noch überschreitet. Ein vollständiger Test aller Pfade, geschweige denn aller Zustände, ist somit bei realen Systemen praktisch nicht durchführbar.

BlackBox- vs. WhiteBox-Testing

Das eben erläuterte Beispiel der vollständigen Pfadabdeckung (auch als C2-Abdeckung bezeichnet) ist ein typischer Vertreter sogenannter *WhiteBox*-Testmethoden.

Beim WhiteBox-Testing fließt Wissen über die interne Programmstruktur in die Erstellung und die Auswahl der Testfälle ein. Neben der C2-Abdeckung gehören auch die schwächeren C0- und C1-Abdeckungskriterien in die Kategorie der WhiteBox-Tests, die die vollständige Anweisungs- bzw. Zweigüberdeckung zum Ziel haben.

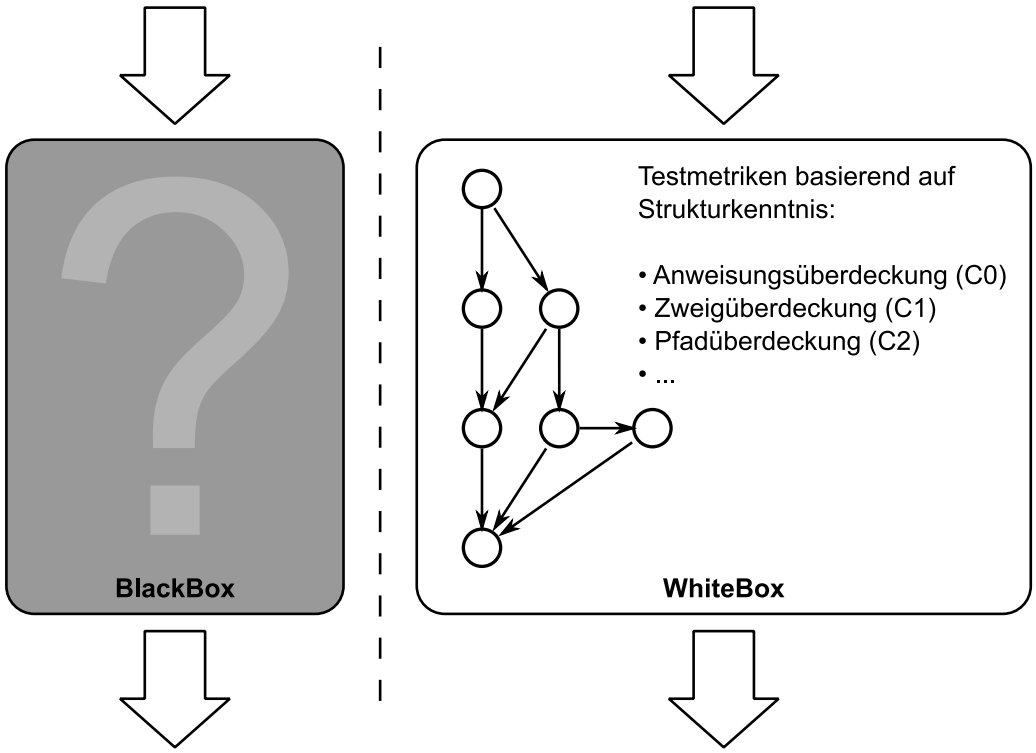

Abbildung 6.2: Schematische Darstellung des Unterschieds zwischen BlackBox- und WhiteBox-Testverfahren. Beim WhiteBox-Test fließt Wissen über die interne Struktur des Prüflings ein, während beim BlackBox-Test allein die Systemspezifikation als Testgrundlage herangezogen werden kann

Im Gegensatz dazu wird beim *BlackBox*-Testing ohne Berücksichtigung der internen Struktur gearbeitet und der Prüfling alleine nach seinem Schnittstellenverhalten beurteilt. Grundlage der Testableitung ist dabei die Softwarespezifikation, deren Umsetzung durch gezielte Stimulation des Prüflings an seinen Schnittstellen und durch Analyse der dadurch hervorgerufenen Reaktionen überprüft werden soll.

Innerhalb eines Softwareentwicklungsprozesses werden WhiteBox-Verfahren vor allem in frühen Phasen der Programmerstellung verwendet. Eine verbreitete Variante sind sogenannte *Unit-Tests*, in denen der Programmierer Testfälle für jeden Codeabschnitt erstellt, der eine Funktion oder einen bestimmten Sonderfall behandelt. Als am effektivsten hat sich dabei die quasi-parallele Programmierung von Code und Testfall erwiesen. Jedem neuen Codeabschnitt wird also sofort ein zugehöriger Testfall gegenübergestellt und in eine Bibliothek aufgenommen. Diese Tests werden dann gebündelt nach jedem größeren Entwicklungssprung erneut ausgeführt (möglichst automatisiert), um die Rückwirkungsfreiheit neuer Codeabschnitte auf die Funktion der bestehenden Codebasis zu überprüfen. Dieser Vorgang wird auch als *Regressionstest* bezeichnet.

Dem gegenüber werden BlackBox-Tests eher in späteren Entwicklungsphasen eingesetzt und auch nicht vom Programmierer, sondern von unabhängigen Personen entwickelt, ausgeführt und analysiert. Durch dieses Vier-Augen-Prinzip soll eine neutrale, auf der Spezifikation basierende Überprüfung des Codes gewährleistet werden, die nicht durch den voreingenommenem Blick des Programmierers verfälscht wird.

Aus den unterschiedlichen Prinzipien von BlackBox- und WhiteBox-Tests lässt sich schließen, dass mit Ihnen unterschiedliche Ziele verfolgt werden. Ein WhiteBox-Test kann beispielsweise keine *fehlenden* Funktionen erfassen, da er strukturbasiert testet und somit nur überprüft „was da ist“.

Im Gegensatz dazu kann durch einen BlackBox-Test keine Aussage darüber getroffen werden, welcher Codeanteil bereits überprüft wurde, weil in ihm nur das Schnittstellenverhalten relevant ist. Durch welche Codeanteile dieses Verhalten hervorgerufen wurde spielt für den Test keine Rolle.

BlackBox- und WhiteBox-Tests können demnach einander nur ergänzen, aber nicht ersetzen. Daher gibt es

Bestrebungen, die Vorteile beider Verfahren im „GreyBox-Testing“ zu vereinigen. Ein Beispiel dafür ist der „Tests-First“-Ansatz der agilen Programmierung, in dem die Testfälle vor dem eigentlichen Programm erstellt werden. In dem Sinne, dass ihnen keine Kenntnis der Prüflingsstruktur zugrunde liegt, sind sie den BlackBox-Tests zuzuordnen. Die Tests selber haben allerdings eher WhiteBox-Charakter, so dass sie auch dieser Gruppe zugeordnet werden könnten. Eine andere Variante von GreyBox-Tests ist die Stimulation des Programms mit speziellen Werten. Verbreitete Beispiele sind Feldgrenzen, leere Zeichenketten, ungültige Eingaben, etc. Das Programm muss diese speziellen Werte korrekt behandeln. Diese Art von Test hat GreyBox-Charakter, da sie zur Bestimmung der speziellen Werte die interne Programmstruktur kennen muss (WhiteBox), der Test an sich aber funktionsorientiert ist (BlackBox).

Testebenen

Im vorangegangenen Abschnitt wurde bereits darauf hingewiesen, dass in unterschiedlichen Phasen der Softwareentwicklung unterschiedliche Teststrategien angewendet werden. Entsprechend wird in diesem Zusammenhang auch von *Testebenen* gesprochen, die in den jeweiligen Entwicklungsabschnitten zum Einsatz kommen (vgl. rechter Ast des V-Modells, Seite 56):

Modultest: Der Modul- oder Unittest schließt sich direkt an die Implementierung der kleinsten Softwareeinheit, dem Modul, an. Typischerweise kommen in ihm WhiteBox-Teststrategien zum Einsatz, die durch Werkzeuge wie beispielsweise JUnit [Bec02] unterstützt werden. Abgesehen vom empirischen, stichprobenartigen Ansatz der Unit-Tests existiert eine Vielzahl weiterer Testmethoden, die auf unterschiedlichsten Ansätzen beruhen (Quellcodeanalyse, Metriken, symbolische Analyse von Wertebereichen auf Quellcodebasis u. v. m). Eine umfassende Übersicht über grundlegende Verfahren zum Modultest gibt [Lig90].

Integrationstest: Werden einzelne Komponenten des Systems zusammengefügt, wird die korrekte Interaktion zwischen ihnen durch Integrationstests überprüft. Grundsätzlich müssen dabei zwei Arten von Integration unterschieden werden:

Software-Software-Integration: Bei der SW-SW-Integration werden einzelne Programmmodule zu einer größeren Einheit verbunden. Die Ausführung und der Test dieser Einheit erfolgt in der Regel auf derselben Plattform (CPU, Rechnerarchitektur, Betriebssystem), auf der auch die Entwicklung stattfindet. Die Stimulation des Prüflings erfolgt durch spezielle Simulatoren, die keine vollständige realitätsnahe Umgebungssimulation bieten, sondern lediglich eine gezielte Stimulation der Schnittstellen ermöglichen. Hardwareschnittstellen, wie beispielsweise Feldbusanschlüsse, werden dabei ebenfalls in Software simuliert. Schwerpunkt beim SW-SW-Integrationstest ist die Überprüfung der aus Datensicht korrekten Interaktion einzelner Programmmodule.

Hardware-Software-Integration: Im Rahmen der HW-SW-Integration werden einzelne Programmmodule erstmals auf der Zielplattform zur Ausführung gebracht und getestet. Neben der Frage der grundsätzlichen Lauffähigkeit des Codes (Cross-Kompilierung, Zielbetriebssystem etc.) steht vor allem die korrekte Verwendung der Hardware- und Betriebssystemschnittstellen im Vordergrund der Tests. Darüber hinaus muss auf der Zielplattform das Zeitverhalten des Programms überprüft werden, da es bedingt durch die Performance und das Scheduling des Zielsystems zu Verletzung von Echtzeitbedingungen oder „Race Conditions“ kommen kann.

Die Testumgebung muss die Stimulation der Prüflingsschnittstellen mit realen Signalen ermöglichen, die auch auf physikalischer Ebene genau der Zielumgebung entsprechen, in der der Prüfling später eingesetzt werden soll. Die Erzeugung der Stimuli kann auf einer exakten Umgebungssimulation beruhen, muss aber nicht.

Systemtest: Der Test der vollständig auf der Zielplattform integrierten Gesamtsoftware wird als Systemtest bezeichnet. In ihm wird das System unter realistischen Einsatzbedingungen einem überwiegend funktionalem Test unterzogen, um vor der Übergabe an den Kunden die Erfüllung der Spezifikation sicherzustellen. Dafür kann als Testumgebung das echte spätere Einsatzumfeld (sogenannte Feldtest) oder ein Labor mit einer realitätsnahen Umgebungssimulation verwendet werden.

Eine Komponente des Systemtests ist der sogenannte „Stresstest", in dem die Gesamtanlage der maximal spezifizierten Belastung ausgesetzt oder sogar außerhalb der Spezifikation betrieben wird. Typische Beispiele für Stressgrößen sind das Nachrichtenaufkommen auf Bussen oder die Anzahl zu verwaltender Objekte. Abhängig vom Testplan und der Systemarchitektur können der Stresstest oder Teile davon bereits bei der HW-SW-Integration durchgeführt werden.

Abnahmetest: Der Abnahmetest erfolgt durch den Kunden nach Integration der Anlage in das Einsatzumfeld. Er stellt einen rein funktionalen Test gegen die Spezifikation dar. Eine andere Bezeichnung ist auch „Akzeptanztest", da nach Bestehen des Tests die Anlage in das Eigentum und in die Verantwortung des Kunden übergeht. Häufig wird zwischen einem Abnahmetest beim Kunden (CAT, Customer Acceptance Test) und einem vorgelagerten Test beim Hersteller vor Auslieferung der Anlage (FAT, Factory Acceptance Test) unterschieden. Der FAT wird durch den Kunden überwacht und begleitet.

Generell gilt, dass fertiggestellte Komponenten so früh wie möglich integriert und getestet werden sollen (Abb. 6.3). So können Fehler im Schnittstellenverhalten frühzeitig aufgedeckt und beseitigt werden. Komplexe Systeme sind daher durch mehrstufige bzw. mehrfache Integrations- und Testschritte gekennzeichnet. Da die Wechselwirkungen zwischen einzelnen Modulen und Subsystemen im Allgemeinen sehr komplex sind, ist die Bestimmung einer geeigneten Integrationsreihenfolge nicht trivial. Liegt ein geeignetes (semi-)formales Systemmodell – beispielsweise in UML – vor, kann es zur Bestimmung der Modulabhängigkeiten und damit zur Bestimmung der Integrationsreihenfolge herangezogen werden.

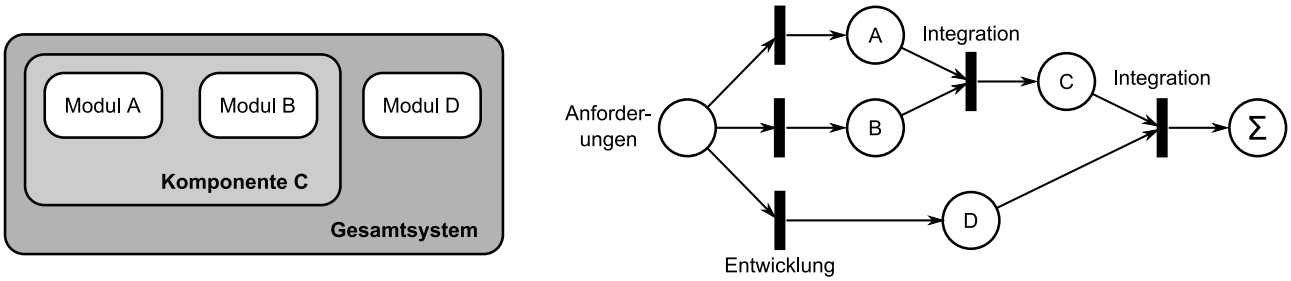

Abbildung 6.3: Beispiel für die Integrationsreihenfolge eines fiktiven Systems: die Module A und B, die zusammen die Komponente C bilden, werden unabhängig von D so früh wie möglich integriert. Erst nach erfolgreicher Integration von A und B erfolgt zusammen mit D die Integration zum Gesamtsystem

Eine möglichst frühe Aufdeckung von Fehlern im Laufe des Entwicklungsprozesses, die unter anderem durch eine frühzeitige Integration von Modulen und Subsystemen erreicht werden soll, reduziert auch die Entwicklungskosten. Untersuchungen haben gezeigt, dass die Kosten für die Beseitigung eines Fehlers steigen, je später der Fehler entdeckt wird (vgl. Abb. 5.3 auf Seite 58). Frühes Testen und ein systematischer, weniger fehlerträchtiger Systementwurf verursachen also initial höhere Aufwände, über den gesamten Entwicklungsprozess betrachtet fallen die Kosten jedoch geringer aus.

6.2 Testumgebungen

Im vorangegangenen Abschnitt wurde bereits erwähnt, dass für unterschiedliche Testebenen unterschiedliche Testumgebungen notwendig sind. Auf Modultestebene ist häufig die Einbindung des Prüflings in ein Ske-

lett leerer Nachbarmodule ausreichend, die alle notwendigen Schnittstellen aufweisen, aber keine zugehörige Funktionalität implementiert haben. Diese „leeren“ Schnittstellenroutinen werden auch als *Stubs* (engl. für „Stummel“) bezeichnet. Der Aufruf eines Stubs durch den Prüfling kann protokolliert und der übergebene Parametersatz zur Testauswertung analysiert werden.

Mit fortschreitender Systemintegration ist auch die Testumgebung anzupassen und nach und nach einer Umgebungssimulation des Prüflingsumfelds anzunähern. Während auf Ebene des Integrationstests eine auf der reinen Datensicht beruhende Simulation noch ausreichend ist, müssen im Systemtest reale Schnittstellenereignisse mit realem Zeitverhalten nachgebildet werden.

Wird der Prüfling in eine entsprechende Simulationsumgebung eingebunden, die einerseits Eingaben bzw. Stimuli für den Prüfling erzeugt und andererseits auf dessen Ausgaben reagiert, liegt ein geschlossener Datenkreislauf zwischen Testumgebung und Prüfling vor. Dient dabei die fertig integrierte Zielhard- und -software als Prüfling, ist entsprechend die Bezeichnung *Hardware-In-The-Loop* (kurz HiL, Abb. 6.4) üblich. Andere, vergleichbare Verfahren sind *Software-In-The-Loop* (SiL, reiner Software-Test ohne Stimulation von Hardwareschnittstellen) oder *Model-In-The-Loop* (MiL, es wird lediglich das Modell eines Systems stimuliert, nicht dessen späterer Code).

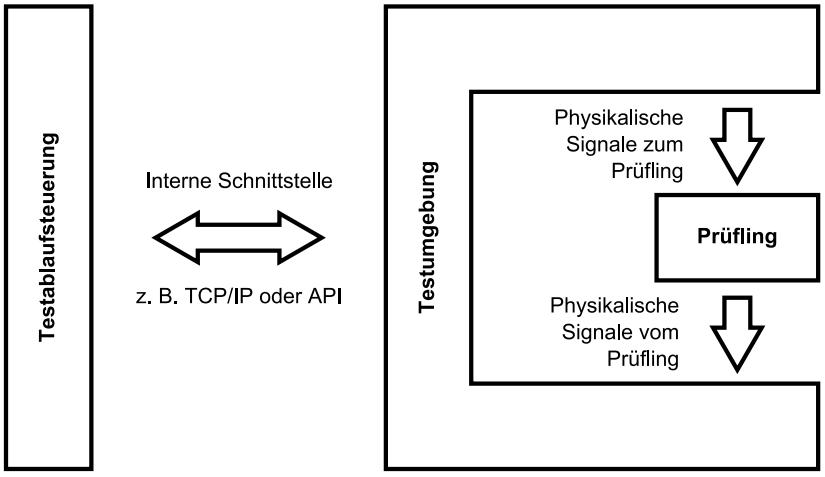

Abbildung 6.4: Schematischer Aufbau und Datenfluss einer HiL-Testumgebung

Im Vergleich zwischen MiL-, SiL-, und HiL-Testumgebungen kommt dem HiL-Test die größte Bedeutung zu, da er als Ersatz für teure und aufwändige Feldtests herangezogen werden kann und dadurch zu einer deutlichen Reduzierung der Entwicklungskosten führt.

Eine HiL-Anlage wie in Abb.6.4 besteht zum einen aus den signalerzeugenden bzw. signalaufzeichnenden Teilen, die direkt mit dem Prüfling interagieren („Testumgebung“ in der Abbildung). Dieser Teil implementiert auch die Umgebungssimulation, also das Wissen über das Verhalten und die Zusammenhänge der Elemente im Umfeld des Prüflings[1]. Zum anderen ist eine zentrale Logik notwendig, durch die das zeitliche Verhalten des jeweiligen Testfalls kontrolliert wird. Diese Logik wird als „Testablaufsteuerung“ bezeichnet und interagiert mit der eigentlichen Testumgebung über eine interne Schnittstelle.

Häufig handelt es sich bei der Testablaufsteuerung um Interpreter für proprietäre oder frei verfügbare Skriptsprachen, die eine einfache HighLevel-Testfallbeschreibung ermöglichen. Der Interpreter löst dann zu jedem Skriptbefehl eine bestimmte Anzahl Schnittstellenaktionen aus. Beispiel: die (rein fiktive) Anweisung `setSpeed(60)` veranlasst die Testumgebung, einem zu testenden Zugsicherungsgerät eine Momentangeschwindigkeit der Lok von 60 km/h vorzuspielen. Dies könnte unter anderem durch die Erzeugung oder Anpassung entsprechender kontinuierlicher Rechteckspannungen geschehen, die mit den Wegimpulsgeber-

[1] Selbstverständlich müssen nur die Teile der Umgebung simuliert werden, zu denen der Prüfling Schnittstellen aufweist

eingängen des Prüflings verbunden sind. *Wie* die Geschwindigkeit simuliert wird, ist Teil des Wissens der Testumgebung. *Warum* in einem gewissen Testkontext eine Geschwindigkeit von 60 km/h simuliert werden soll, ist Teil des Testablaufs und damit des Testskriptes.

Zusammenfassend können die folgenden Punkte als die wichtigsten Anforderungen angesehen werden, denen eine HiL-Testumgebung genügen muss:

- Bedienung der originalen Prüflingsschnittstellen
- Protokollierung und Verarbeitung der Prüflingsreaktionen; die Prüflingsreaktion muss sich auch auf die Umgebungssimulation auswirken können. Ein reines Zurückspielen vordefinierter Stimuli ist im Allgemeinen nicht ausreichend.
- Echtzeitinteraktion mit dem Prüfling; sofern es der Prüfling zulässt, sind allerdings auch optional Zeitraffung und Zeitdehnung vorzusehen.
- Bereitstellung einer Steuerungsschnittstelle, über die der Testablauf kontrolliert werden kann
- Zusammenfassung von Testschritten zu Testsequenzen bzw. Testszenarien, die gebündelt ausgeführt werden können
- Deterministischer und reproduzierbarer Testablauf

HiL-Testumgebungen für Eisenbahnleit- und -sicherungstechnik

In der Eisenbahn-LST bestehen HiL-Tests für gewöhnlich aus simulierten Zugfahrten, in die der Prüfling eingebettet wird. Die Umgebungssimulation muss daher die Dynamik der Zugbewegung ebenso umfassen wie die Kommunikation zwischen fahrzeug- und streckenseitigen Einrichtungen (Balisen, INDUSI-Magneten, GSM-R-Übertragungen, Signalbegriffe, ...). Darüber hinaus muss der Zustand der Umgebung erfasst und angepasst werden. Dazu zählen Weichenlagen, Signalstellungen und Gleisbesetztmeldungen. Soll die Zugfahrt während der Simulation visualisiert werden – was einem Blick aus dem Führerstand auf die Umgebung und die Strecke entspricht – müssen umfangreiche Informationen über die Landschaft, die vollständige Streckentopo*grafie*[2] und das Aussehen von Signalen, entgegenkommenden Zügen, Häusern, Bahnhöfen, etc. hinterlegt werden.

Im RailSiTe®, dem eisenbahntechnischen Labor des DLR, kann eine realistische HiL-Simulation für ETCS einschließlich der beschriebenen Visualisierung vorgenommen werden. Konkret bietet das Labor die Möglichkeit zur HiL-Einbindung realer ETCS-Fahrzeuggeräte oder realer ETCS-Funkstreckenzentralen (RBC, Radio Block Center) und simuliert einen Triebfahrzeugführer- und einen Fahrdienstleiterarbeitsplatz.

Der Einsatzbereich des RailSiTes umfasst ETCS-Konformitätstests gemäß [UNI05a], die betriebliche Validierung neuer bzw. geplanter Strecken ([KGL07]) und die simulative Untersuchung von Migrationsszenarien beim Übergang von existierenden Altsystemen auf neue Konfigurationen wie beispielsweise ETCS [KOJL05, OKJL05].

Für eine derart komplexe Anlage ist es unerlässlich, dass der Datenaustausch und die Datenkonsistenz zwischen den an der Simulation beteiligten Komponenten zu jeder Zeit gewährt bleibt und trotz hohen Kommunikationsaufwands die Echtzeitbedingungen für die Gesamtsimulation nicht verletzt werden.

Wie Abb.6.5 zeigt, überdecken die von der Simulation zu verwaltenden Daten ein breites Spektrum, das von Bedienhandlungen der Tester über infrastruktur- und fahrzeugseitige Zustandsinformationen bis hin zu Texturinformationen für Landschaftsdarstellungen reicht.

Neben rein internen Schnittstellen, die beispielsweise die Visualisierung und die Stellwerkssoftware in die Simulation einbinden [GKJL06, KGL07], muss eine Reihe offizieller Schnittstellen vorgehalten werden, die

[2] Für einfache Simulationen ohne Visualisierung reicht die Streckentopo*logie* ergänzt um wenige Parameter wie Gradienten oder Kurvenradien aus.

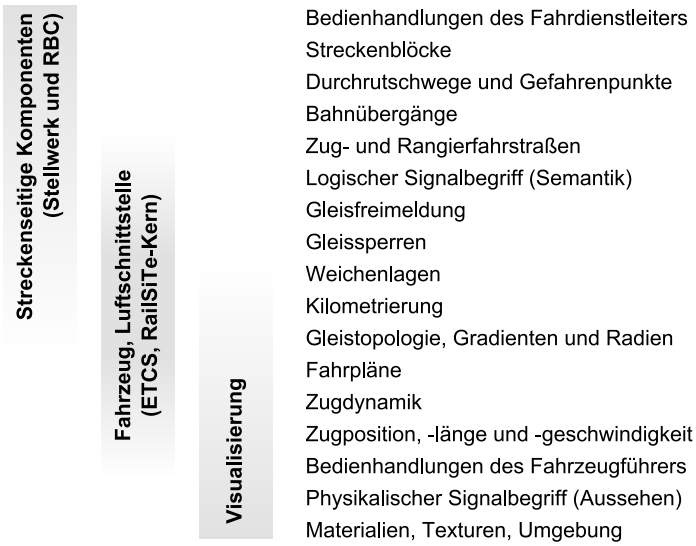

Abbildung 6.5: Bandbreite der für eine LST-Simulation samt Visualisierung auszutauschenden Daten

die Einbindung realer Prüflinge (Hardware) in die Simulation erlauben. Unter anderem sind dies Odometriesignale, Zustandsinformationen über das Bremssystem, Kommunikation über die Luftschnittstelle via Balise, GSM-R und EuroLoop, ISDN-Ports zum Anschluss realer Kommunikationseinrichtungen und einige andere Schnittstellen mehr [UNI05b].

6.3 Modellierung von Tests

Grundsätzlich ist auch ein Testsystem, wie beispielsweise das RailSiTe, ein normales (Software-)System[3]. Es besitzt daher, wie in den Abschnitten 2.1 und 3.2 bereits ausgeführt wurde, Struktur, Zustand, Verhalten und Funktion. Diese Eigenschaften eines Testsystems können daher mit geeigneten Beschreibungsmitteln in der gewünschten Form modelliert werden (wie in den Kapiteln 2 und 3 beschrieben).

Im Folgenden soll aber weniger die Modellierung des internen Verhaltens und der internen Struktur der Testumgebung im Vordergrund stehen, sondern vielmehr die möglichst formale Erfassung der Testfälle, die in einer Testumgebung zur Ausführung gebracht werden. Die Ausführung kann dabei manuell – also durch einen Tester – erfolgen oder aber automatisiert durch die Testumgebung geschehen. Die Vorteile der automatisierten Testausführung liegen neben der offensichtlichen Einsparung von Personalkosten und der wahrscheinlichen Reduzierung der Testzeit vor allem in der exakten Reproduzierbarkeit der Tests und der daraus resultierenden guten Vergleichbarkeit der Testergebnisse.

Die automatische Testausführung verlangt allerdings eine sehr präzise Testspezifikation, die in ihrer Formalität und ihrem (sequentiellen) Charakter einem gewöhnlichen Programm in nichts nachsteht. Tatsächlich werden viele Testumgebungen durch Skriptsprachen wie Ruby (Beispiel: „Test- und Simulationsumgebung“ (TuS) der Firma Siemens Transportation Systems) oder Python (Beispiel: Audi, [SD04]) angesteuert. Der Testfall ist also ein reguläres Skript, abgefasst in der jeweiligen Skriptsprache.

In der Praxis weisen Tests oberhalb der Modulebene allerdings häufig nicht die notwendige Formalität auf, die für eine automatische Testausführung notwendig wäre. Vor allem auf Systemtestebene sind vielfach nur informale, verbale Testfallbeschreibungen oder bestenfalls Testschrittsammlungen in Tabellenform zu finden.

[3] Hier ergibt sich die interessante Rekursion „Wie wird ein Testsystem getestet?“. Am Ende der Rekursion muss stets eine manuelle Verifizierung und Validierung stehen, beispielsweise durch (Stichproben-)Test gegen die Spezifikation, denn selbst eine formale Verifikation würde nur die Richtigkeit eines zugrunde liegenden Algorithmus zeigen, nicht jedoch dessen korrekte Umsetzung in Code oder dessen grundsätzliche Eignung für die Lösung der Problemstellung.

Ein gutes Beispiel dafür ist [UNI05a]. Die darin definierten ETCS-Konformitätstests, die auf Systemtestebene angewendet werden, sind als Tabelle eines Microsoft®-Word-Dokumentes hinterlegt und das für den Testfall von einem virtuellen Zug einzuhaltende Geschwindigkeitsprofil ist in Form einer Bitmap definiert. Diese Form der Testfalldefinition ist für eine nachgelagerte Testautomatisierung denkbar ungeeignet, zumal ein Teil der Daten in einem proprietären Format – Microsoft Word – abgelegt ist.

Es gibt verschiedene Ansätze, die Beschreibung von Testfällen zu verbessern. Als Beispiele seien Arbeiten von Ebner und Ebrecht [Ebn05, Ebn04, EzH04, ESB05, EMzHL07] genannt, wobei sich die erstgenannten Arbeiten an UML und Sequenzdiagrammen orientieren und die letztgenannten vor allem aus Arbeiten an ETCS-Konformitätstests entstanden sind.

Da eines der zentralen Ziele dieser Arbeit die Erstellung eines homogenen System- und Testmodells auf der Basis von UML ist, soll hier der Fokus auf UML-basierten Techniken und dabei insbesondere auf dem UML 2 Testing Profile (U2TP) liegen. Die Grundlagen des U2TP wurden bereits in Abschnitt 4.6 kurz eingeführt. Darauf aufbauend wird auf den folgenden Seiten detaillierter auf die Modellierung von Testfällen mit Hilfe des U2TP eingegangen.

Die entscheidenden Gründe, für die Testfallmodellierung auf das U2TP zurückzugreifen, waren:

- Das U2TP ist ein UML-basiertes Testbeschreibungsmittel. Die Integration mit einem UML-basierten Systemmodell wird dadurch erleichtert.
- Das U2TP ist für BlackBox-Tests konzipiert und damit für die überwiegend funktionalen Tests auf höherer Ebene (Integrations-, System-, Abnahme- und Konformitätstest) besonders geeignet.
- Das U2TP ist in der Lage, neben dem Testablauf auch den Aufbau und die Konfiguration der Testumgebung zu erfassen. Dadurch wird die Qualität der Testbeschreibung erhöht und die Reproduzierbarkeit verbessert.
- Das U2TP kann nahezu vollständig auf TTCN-3 und partiell auch auf JUnit abgebildet werden, wodurch eine Schnittstelle zur automatischen Testausführung gegeben ist.
- Das U2TP ist weder auf eine bestimmte Testebene noch auf eine bestimmte Anwendungsdomäne beschränkt.
- Die U2TP-Spezifikation ist frei verfügbar und wird von der OMG gepflegt.

6.3.1 Beschreibung der Testumgebung mit U2TP

Eine Testumgebung zeichnet sich vor allem durch den Prüfling (SUT, *System Under Test*) und die mit ihm interagierenden Komponenten (*Test Components*) aus. Die Testkomponenten stellen die notwendige Funktionalität zur Stimulation des Prüflings zur Verfügung und nehmen auch dessen Ausgaben auf. Der Prüfling und die Testkomponenten werden als UML-Klassen in entsprechenden Klassen- bzw. Strukturdiagrammen beschrieben und durch die entsprechenden Stereotypen `≪SUT≫` bzw. `≪TestComponent≫` klassifiziert.

Darüber hinaus definiert das U2TP auch Schnittstellen für Klassen, die der Steuerung des Testablaufs dienen. Es handelt sich dabei zum einen um den sogenannten *Arbiter*, der die Rückmeldungen verschiedener Testkomponenten über den Erfolg einzelner Testschritte zu einem Gesamturteil über den vollständigen Test zusammenfasst. Zum anderen wird das Interface für eine *Scheduler*-Klasse eingeführt, die den zeitlichen Ablauf des Tests steuert und ggf. Testkomponenten instantiiert oder startet.

Scheduler und *Arbiter* sind im U2TP nur durch ihre Schnittstelle und ihr grundsätzliches Verhalten beschrieben. Eine tatsächliche Implementierung muss durch den Anwender vor dem Hintergrund der jeweiligen Testumgebung vorgenommen werden.

Zusammenstellung der Testkonfiguration

Die einzelnen Bestandteile der Testumgebung werden durch die bisher genannten Maßnahmen nur „lose“, also alleinstehend, definiert. Dadurch wird es möglich, Bibliotheken verschiedener Testkomponenten aufzubauen und deren Elemente dann zu einem speziellen Testaufbau zu kombinieren. Die Definition der Bibliothekselemente proftiert dabei von objekt-orientierten Techniken bzw. UML-Funktionen wie Vererbung, Generalisierung, Parametrierung und Instantiierung. Mit Hilfe von Vererbung und Generalisierung können Testkomponenten aufeinander aufbauend spezifiziert und in ihren Eigenschaften zunehmend konkretisiert werden. Darüber hinaus ist eine Manipulation des Komponentenverhaltens über das Setzen von Parametern zur Laufzeit (vorzugsweise bei Instantiierung) möglich.

Somit ist über

- abstrakte Klassen (nur Schnittstellenbeschreibung, nicht instantiierbar)
- Vererbung bzw. Generalisierung (Implementierung von Schnittstellenverhalten, Konkretisierung und Verfeinerung der Komponentenfähigkeiten)
- Parametrierung zur Laufzeit

ein kontinuierliche Übergang von generischen zu spezifischen Testkomponenten möglich, die wiederum zu Testkonfigurationen für verschiedene Abstraktionsebenen kombiniert werden können.

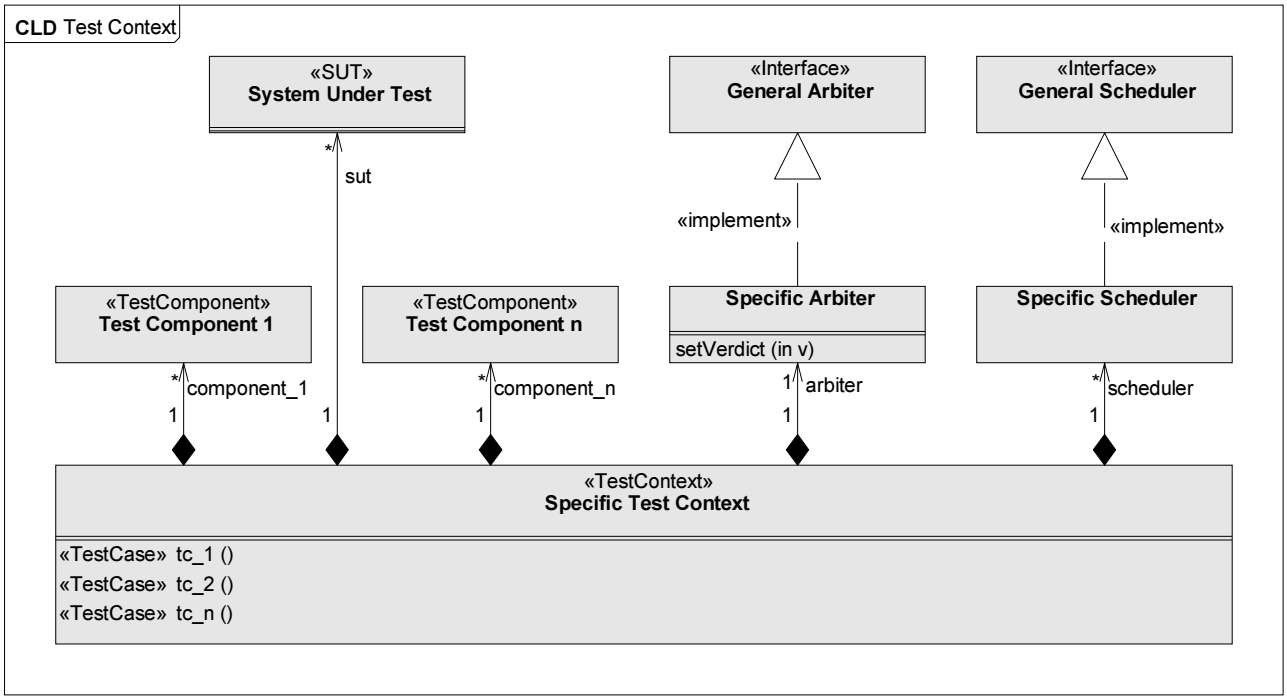

Abbildung 6.6: Zusammenstellung mehrerer Testkomponenten zu einer individuellen Testkonfiguration (*Test Context*)

Die Zusammenstellung einer solchen individuellen Testkonfiguration erfolgt mit Hilfe des sogenannten *Test Context* (Abb. 6.6). Dabei handelt es sich um eine Klasse, die die jeweils zu verwendenden Testkomponenten enthält (als Attribute bzw. Assoziation) und auch die inneren Beziehungen zwischen diesen Komponenten festlegt (in einem Strukturdiagramm). Die Methoden einer solchen mit `≪TestContext≫` stereotypierten Klasse sind die einzelnen Testfälle (klassifiziert durch `≪Testcase≫`), die durch Aufruf von außen ausgelöst werden können und somit das Interface zwischen Tester und Testumgebung darstellen.

6.3.2 Beschreibung des Testverhaltens mit U2TP

Die dynamischen Aspekte einer Testfallbeschreibung werden vor allem durch die Interaktion zwischen den Testkomponenten und dem Prüfling bestimmt. Sie werden mit Hilfe der in der UML gängigen Verhaltensdiagramme (vornehmlich Sequenz-, Aktivitäts- oder Zustandsdiagramme) beschrieben. Üblicherweise sind für diesen Anwendungsfall Sequenzdiagramme die geeignetste Notation, da sich in ihnen die Abfolge des Datenaustauschs an den Prüflingsschnittstellen und auch zeitliche Randbedingungen übersichtlich darstellen lassen. Darüber hinaus ist innerhalb eines Sequenzdiagramms der Verweis bzw. die Einbindung eines anderen Sequenzdiagramms möglich. Über diesen Mechanismus lassen sich daher sehr wirkungsvoll Bibliotheken von Testschrittabfolgen aufstellen, wie auch später in der Fallstudie an einem Beispiel gezeigt wird.

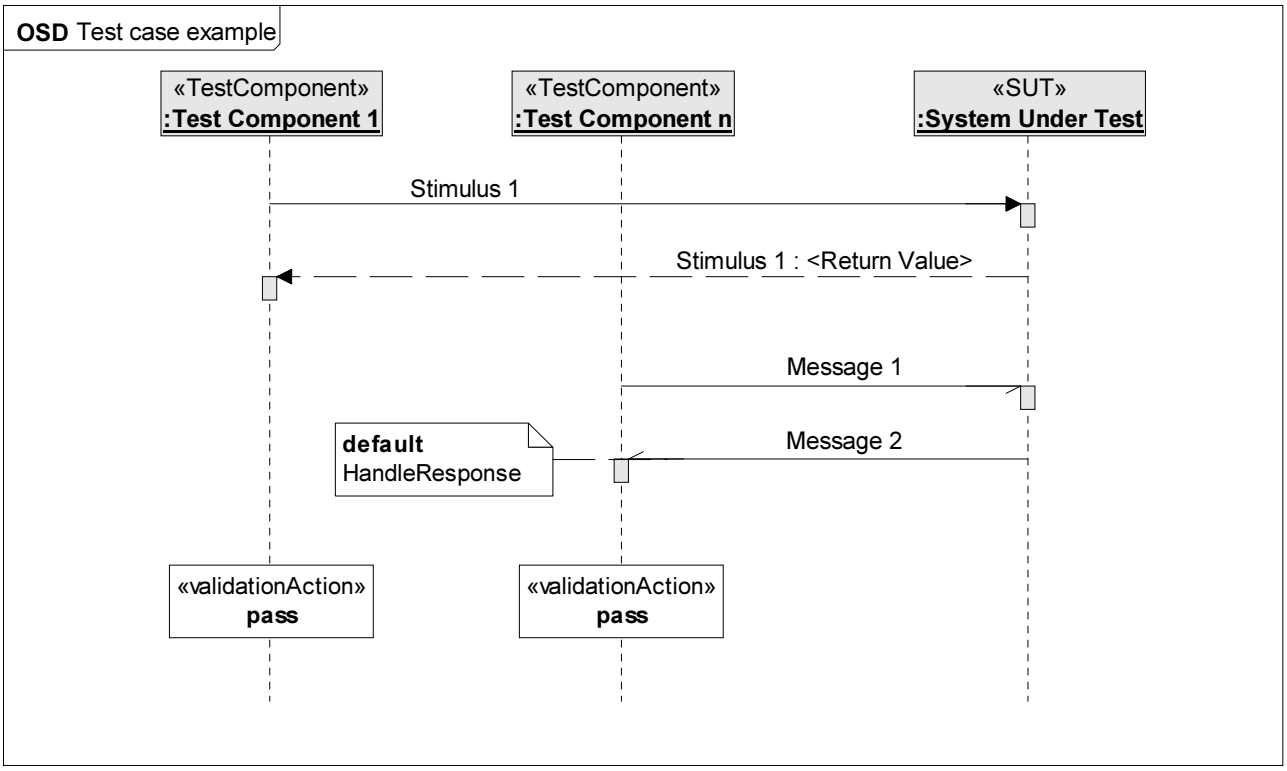

Abbildung 6.7: Exemplarische Interaktion zwischen Testkomponenten und SUT in einem Sequenzdiagramm (Vereinfachung: Lebenslinien zeigen Komponenten anstatt Ports)

Abbildung 6.7 veranschaulicht die grundsätzliche Verwendung von Sequenzdiagrammen zur Darstellung des Datenaustauschs zwischen Testkomponenten und Prüfling. Im oberen Teil der Abbildung stimuliert die erste Testkomponente den Prüfling durch einen synchronen Aufruf und dieser antwortet mit einem Rückgabewert. Im zweiten Teil wird eine asynchrone Kommunikation mittels Austausch von Nachrichten dargestellt.

Für den Fall, dass ein Prüfling während der Testausführung vom Sollverhalten abweicht, muss entschieden werden, wie der weitere Testablauf zu gestalten ist. Dafür sieht das U2TP den Einsatz sogenannter *defaults* vor, der ebenfalls in Abbildung 6.7 dargestellt ist. Semantisch kann das verstanden werden als: „Wenn eine andere Nachricht als `Message 2` eintrifft, rufe das Unterprogramm `HandleResponse` auf!“. Das *default*-Konzept wurde aus der Sprache TTCN-3 (*Testing and Test Control Notation Version 3* [ETS05]) übernommen.

Ein *default* verweist also auf eine separate Verhaltensbeschreibung, die den weiteren Testablauf festlegt und für gewöhnlich als weiteres Sequenz- oder Zustandsdiagramm ausgeführt ist. Folgende Möglichkeiten zur Reaktion auf einen Fehler bestehen:

- Wiederholung des fehlerauslösenden Testschritts (*repeat*)

- Ignorieren des Fehlers und Fortsetzen der Testausführung (*continue*)
- Abbruch des Testablaufs

Durch die Trennung von „Normalablauf“ und Fehlerbehandlung werden die Hauptdiagramme übersichtlicher gehalten und dadurch die Les- und Wartbarkeit des UML-Modells verbessert.

Selbstverständlich muss durch einen *default* auch die Wertung des Testergebnisses verändert werden. Wie oben bereits erwähnt erfolgt die Beurteilung des gesamten Tests durch den Arbiter. Die am Test beteiligten Komponenten melden ihm zur Laufzeit ihre „lokalen“ Testergebnisse, aus denen der Arbiter dann das Gesamtergebnis ermittelt. Im U2TP sind bereits einige Ergebnisvarianten vordefiniert:

pass: Der Prüfling hat sich spezifikationskonform verhalten.

fail: Das Gegenteil von *pass*

inconclusive: Es ist nicht entscheidbar, ob das Verhalten spezifikationskonform ist oder nicht.

error: Es liegt ein Fehler in der Testumgebung vor.

Ein einfacher Arbiter könnte beispielsweise eine *worst-case*-Entscheidung vornehmen. Das Verhalten eines solchen Arbiters zeigt Abbildung 6.8.

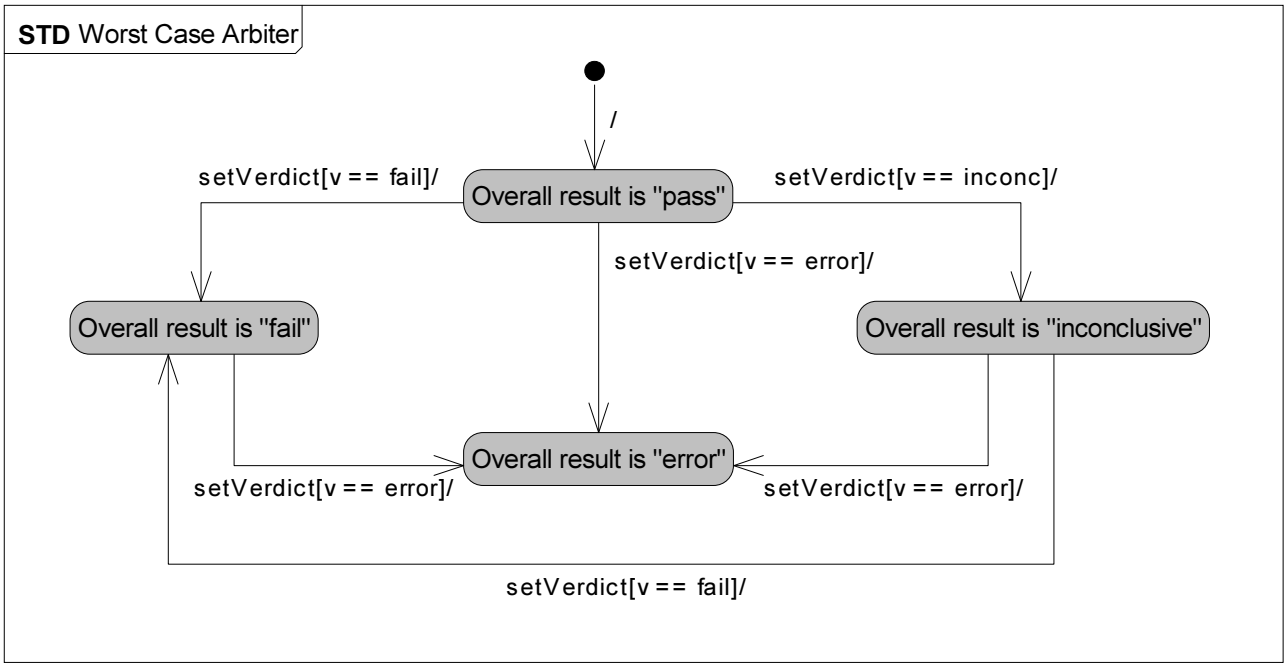

Abbildung 6.8: Zustandsdiagramm eines einfachen Arbiters, der eine *worst-case*-Entscheidung vornimmt. Das `v` in den *Guard-Conditions* sei der Parameter der `setVerdict`-Methode des Arbiters.

Würde eine Testkomponente bei Abweichung des Prüflings vom Sollverhalten ein lokales Urteil *fail* an den Arbiter übertragen, könnte eine andere Komponente nicht mehr ein Gesamtergebnis *pass* hervorrufen. Allerdings sind aber auch andere Arbiter-Implementierungen denkbar, zum Beispiel die Einführung einer Toleranzschwelle, die trotz einer gewissen Anzahl lokaler *fail*-Entscheidungen insgesamt dennoch ein *pass*-Urteil fällt. Aufgrund der vielen Möglichkeiten und Anwendungsfälle definiert das U2TP an dieser Stelle nur ein allgemeines Interface und keine genaue Implementierung eines Arbiters.

Innerhalb der Verhaltensdiagramme wird die Übergabe eines lokalen Urteils an den Arbiter entweder durch explizite Modellierung des Aufrufs seiner `setVerdict`-Methode erfasst oder durch die Einführung einer Aktion, die mit `«validationAction»` stereotypiert ist (Abb. 6.7).

6.3.3 Testdaten und zeitliche Randbedingungen in U2TP

Die für die Ausführung von Tests notwendigen Daten (z. B. Sollergebnisse oder Nachrichtentelegramme) werden im U2TP in sogenannten *Data Pools* organisiert, die wiederum in Untereinheiten (*Data Partitions*) aufgeteilt werden können. Die Notation in UML erfolgt, wie bei Datenstrukturen üblich, als Klassendiagramm, für das die entsprechenden Stereotypen `≪DataPool≫` und `≪DataPartition≫` vom U2TP zur Verfügung gestellt werden.

Für die Beschreibung zeitlicher Zusammenhänge und Abhängigkeiten werden im U2TP Schnittstellen für Timer und Aktionen für deren Steuerung / Interaktion definiert. Timer verfügen über die Grundfunktionen Starten, Stoppen und Auslesen und können nach Ablauf einer wählbaren Zeit eine bestimmte Aktion auslösen (*TimeOutAction*). Darüber hinaus können Testkomponenten verschiedenen Zeitdomänen (sogenannten *time zones*) zugeordnet werden, um auf diese Weise synchron arbeitende Subsysteme zu gruppieren.

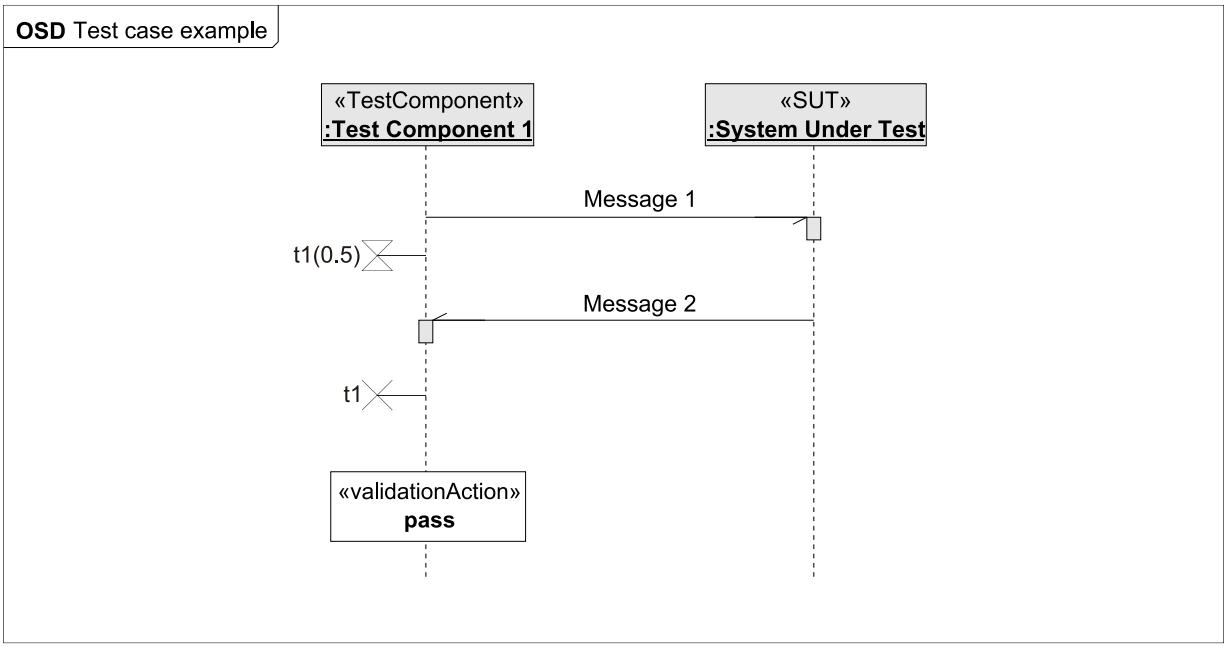

Abbildung 6.9: Exemplarisches Sequenzdiagramm, das den Einsatz von Timern zeigt (Vereinfachung: Lebenslinien zeigen Komponenten anstatt Ports)

Den Einsatz von Timern verdeutlicht Abbildung 6.9 anhand eines fiktiven Beispiels. Nach dem Senden der ersten Nachricht wird ein Timer mit einem Startwert von 0,5 s gestartet und bei Empfang der zweiten Nachricht wieder gestoppt. Sollte es zu einem Timeout kommen, ist die zweite Nachricht nicht rechtzeitig eingetroffen und eine *TimeOutAction* wird ausgelöst, die beispielsweise im *default* des *TestContext* verarbeitet werden könnte (zu *defaults* siehe Abschnitt 6.3.2).

6.3.4 Aufbau von Testbibliotheken

Im vorangegangenen Abschnitt wurde im Zusammenhang mit der Testkontextdefinition bereits erwähnt, dass sich der modulare Aufbau der mittels U2TP hinterlegten Testkonfigurationen zum Aufbau von Komponentenbibliotheken eignet. Dies soll durch ein Beispiel aus dem DLR-Bahnlabor RailSiTe veranschaulicht werden.

Das Labor besteht aus einer verteilten Echtzeitsimulation, die sich aus dem Zusammenspiel vieler Einzelmodule ergibt. Die Konfiguration und Kombination der einzelnen Module ergibt sich aus dem Versuchszweck und kann für jedes Simulationsszenario individuell festgelegt werden. Zur Dokumentation und zur Reproduzierbarkeit eines Versuchs bzw. einer Versuchssreihe ist daher die Modulkonfiguration zwingend zu erfassen.

Aus diesem Grund wurden die wichtigsten Labormodule und ein Prüfling als U2TP-Testkomponenten erfasst und zwei Laborkonfigurationen exemplarisch in UML erfasst. Das sich daraus ergebende Modell mit einer baumartigen Struktur von Modulen, die in unterschiedlichen Testkontexten wiederverwendet werden können, zeigt Abbildung 6.10.

In Abbildung 6.11 werden die Module zu einem Testaufbau – dem U2TP-Testkontext – kombiniert, der für Versuche mit softwaresimulierten ETCS-Fahrzeugrechnern (EVC, European Vital Computer) geeignet ist. Zur besseren Übersicht wurden einige Module weggelassen, die grundsätzlich an jedem Aufbau beteiligt sind.

Als Kontrast zu Abbildung 6.11 zeigt Abbildung 6.12, wie unter Wiederverwendung einiger vorhandener Module und dem Einfügen weniger neuer Module eine Laborkonfiguration für den HiL-Test eines echten ETCS-Fahrzeugrechners definiert werden kann.

In gleicher Weise lassen sich mit Hilfe des U2TP und einem geeigneten UML-Werkzeug Bibliotheken von Testabläufen oder Testdaten erstellen, die dann eine einfache und schnelle Zusammenstellung neuer Testszenarien erlauben. Eine solche Testschrittbibliothek kann beispielsweise aus einer Sammlung von Sequenzdiagrammen mit genau definierten Abläufen bestehen. Auf diese Diagramme mit elementaren Testschritten kann dann aus anderen Sequenzdiagrammen heraus verwiesen werden. Die Fallstudie im übernächsten Kapitel wird ein Beispiel dafür anführen und auch erläutern, wie diese Verweise bei der automatischen Erzeugung von Testcode berücksichtigt werden müssen.

Die Erzeugung solcher Testschritte oder Testfälle kann auf unterschiedliche Weise erfolgen. In vielen Fällen ist eine manuelle Ableitung der Tests aus der Spezifikation (BlackBox) oder der Implementierung (WhiteBox) vorzufinden. Es existieren allerdings auch Ansätze, die Testerzeugung zu automatisieren, wie der folgende Abschnitt ausführt.

6.4 Automatische Testfallerzeugung

Der Begriff „automatische Testerzeugung“ wird in der Praxis und in der Literatur in einem sehr weiten Kontext verwendet. Die Bedeutung reicht von der Transformation (formaler) Testmodelle in Skripte bis hin zur selbstständigen Erzeugung von Testfällen aus einem formalen Systemmodell.

Das Thema wird in der Literatur in zahlreichen Veröffentlichungen behandelt. Innerhalb der Domäne Eisenbahn bzw. innerhalb des Beschreibungsmittels UML/MSCs seien hier exemplarisch [Ebn04, ENL04, SD04, MSS00, SMPS98, CCM+99, SJMS97] genannt.

Dieser Abschnitt wird sich auf die Testerzeugung aus formalen System- oder Verhaltensmodellen beschränken. Die Frage, wie die erzeugten Testfälle in ein ausführbares Format überführt werden können – also beispielsweise die Erzeugung von Skripten aus den erzeugten Testfällen – wird im nächsten Kapitel exemplarisch anhand der TTCN-3-Erzeugung aus UML-Modellen behandelt. Darüber hinaus beziehen sich die im Folgenden beschriebenen Verfahren auf höhere Testebenen (z. B. Integrations- oder Systemtest) und sind auf BlackBox-Ansätze beschränkt. Zahlreiche Verfahren für WhiteBox-Tests können beispielsweise [Lig90] entnommen werden.

Die gängigen Methoden zur automatischen Erzeugung von Systemtestfällen beruhen auf

- einer (graphentheoretischen) Analyse von Verhaltensdiagrammen, mit dem Ziel, alle sinnvollen Pfade bzw. Übergänge innerhalb des Diagramms mit den erzeugten Testfällen anzusprechen. Vorzugsweise werden für solche Methoden Zustands- oder Aktivitätsdiagramme als Grundlage verwendet.

- einer Partitionierung der Eingangsgrößen des Systems in Äquivalenzklassen und der Ableitung von Testfällen als Kombination dieser Äquivalenzklassen. Ergänzend können auch die Ausgangsgrößen des Systems entsprechend partitioniert und kombiniert werden; die Erzeugung der zu den Ausgangsgrößen gehören Eingangsgrößen (Stimuli) muss dann allerdings manuell erfolgen.

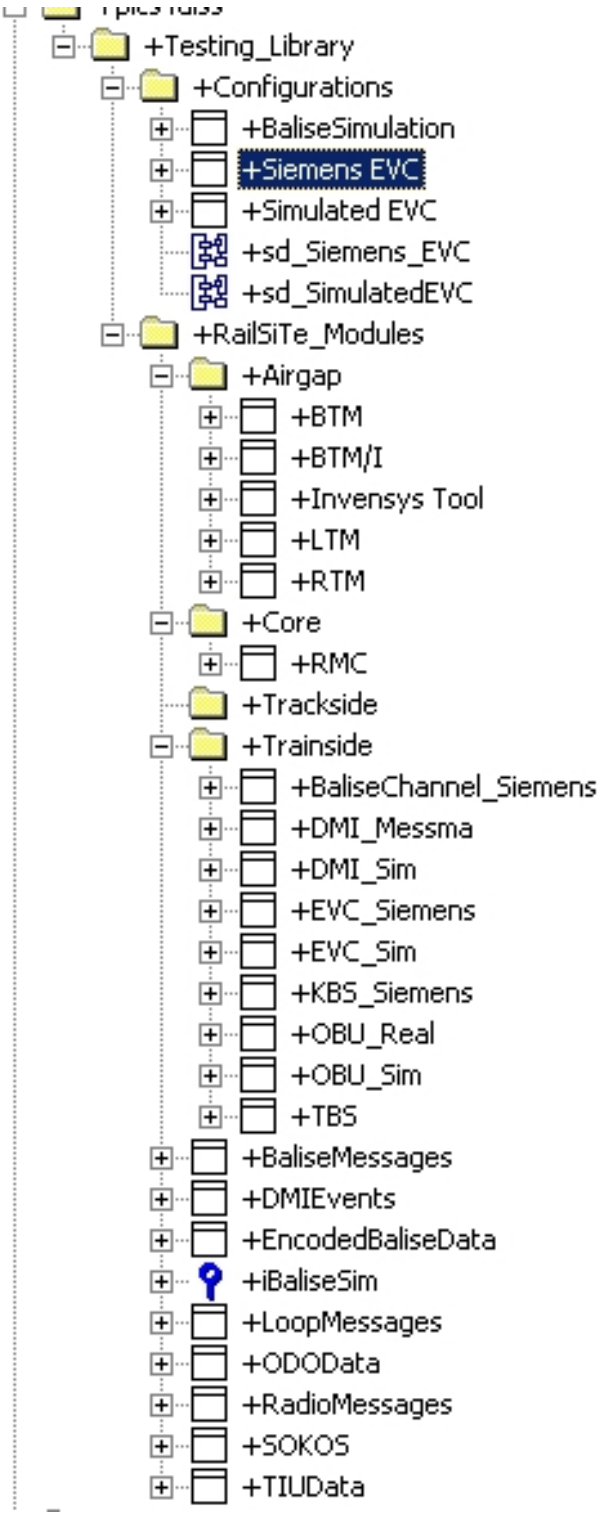

Abbildung 6.10: Baumstruktur der mittels U2TP erfassten RailSiTe-Module (Ausschnitt)

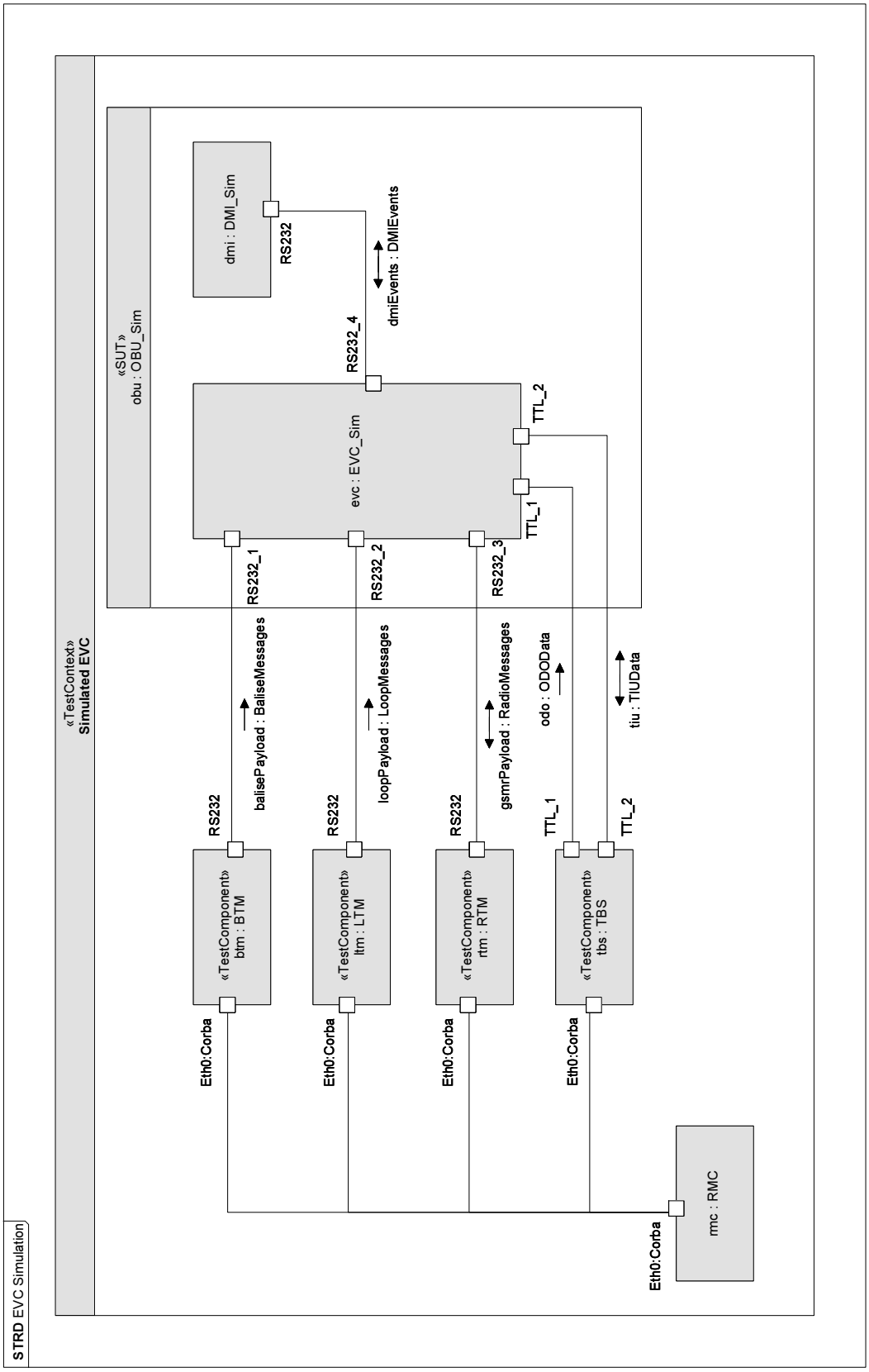

Abbildung 6.11: U2TP-Testkontext für eine Laborkonfiguration mit simuliertem EVC

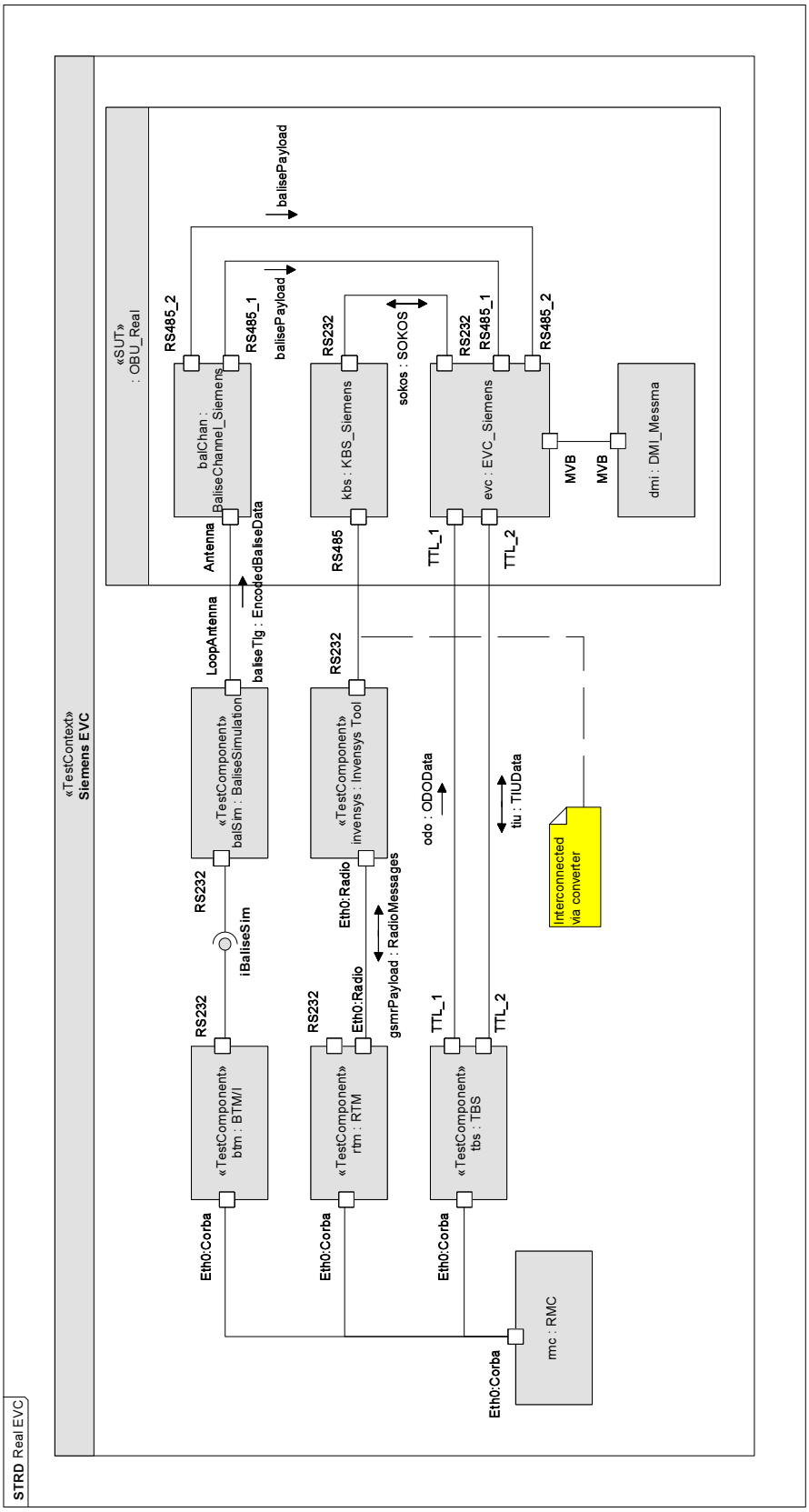

Abbildung 6.12: U2TP-Testkontext für eine Laborkonfiguration zum HiL-Test eines echten EVCs

- einer formalen Verifikation des Systemmodells, durch die vorgegebene Systemeigenschaften als unter allen Umständen gültig bewiesen werden. Erfüllt das System die Eigenschaft nicht, wird durch den Model Checker ein Gegenbeispiel erzeugt. Dieses Gegenbeispiel entspricht einer Sequenz von Eingangswerten. Durch Negation der zu testenden Eigenschaft kann das erzeugte Gegenbeispiel dann als Testfall verwendet werden.

  Verdeutlichung: Angenommen, es soll ein Testfall für die Funktion „Das ETCS-Fahrzeuggerät leitet eine Notbremsung ein“ erzeugt werden. Durch den Model Checker wird dazu die Aussage „Das ETCS-Fahrzeuggerät leitet *niemals* eine Notbremsung ein“ überprüft. Dieser erzeugt erwartungsgemäß eine Sequenz von Eingangsgrößen, die die oben genannte Aussage widerlegt und damit genau die ursprüngliche gewünschte Reaktion hervorruft. Diese Sequenz kann dann im (HiL-)Systemtest zur Stimulation des Prüflings verwendet werden.

  Zu beachten ist, dass dieses Verfahren natürlich nur stichprobenartigen Charakter hat. Sind viele Stimulusabfolgen möglich, die das gewünschte Verhalten auslösen würden, wird selbstverständlich nur eine einzelne erzeugt und getestet. Sollen mehrere Pfade erzeugt werden, müssen entsprechend mehrere Aussagen formuliert werden und durch ergänzende Bedingungen auf die zu erzeugende Testsequenz angepasst werden (Beispiel: „Das Fahrzeuggerät führt niemals eine Notbremsung *im Modus Full Supervision* aus“, „*...im Modus Staff Responsible ...*“ usw.)

Von den drei genannten Verfahren sollen hier die beiden erstgenannten, also die Ableitung aus Verhaltensdiagrammen und die Äquivalenzklassenbildung, genauer behandelt werden. Der dritte Ansatz, das Model Checking, wurde im Rahmen der SCADE-Fallstudie eingesetzt, deren Ergebnisse am Ende dieser Arbeit kurz vorgestellt werden.

6.4.1 Testfallerzeugung aus Verhaltensdiagrammen

Wie bereits erwähnt basiert die Erzeugung von Testfällen aus Verhaltensdiagrammen auf einer graphentheoretischen Analyse der Diagramme. Entsprechende Algorithmen suchen in Abhängigkeit bestimmter Metriken nach Pfaden durch die Graphen, auf denen alle Knoten bzw. alle Kanten besucht werden.

Basiert die Pfadsuche auf *Zustandsdiagrammen*, so ist sichergestellt, dass alle Systemzustände bzw. alle Transition erreicht bzw. ausgelöst werden. Bei Verwendung von *Aktivitätsdiagrammen* bewirkt der Algorithmus, dass alle Aktivitäten und Verzweigungspunkte durchlaufen werden. Bei der Anwendung entsprechender Methodiken lassen sich die genannten zustands- bzw. aktivitätsorientierten Ansätze auch auf Petrinetze abbilden.

Ein Beispiel für die Erzeugung von Systemtestfällen aus Zustandsdiagrammen ist die Ableitung der Testsequenzen für ETCS, die in [UNI05a] enthalten sind. Das ETCS-Verhalten wurde dazu als Knoten-Kanten-Graph mit entsprechenden Zuständen und Übergangsbedingungen auf relativ hoher Ebene beschrieben. Durch einen entsprechenden Algorithmus („Chinese Postman's Problem“) wurden dann Pfade in diesem Graphen gesucht, die jeder für sich eine virtuelle Zugfahrt entlang einer virtuellen Strecke repräsentieren und in deren Verlauf alle relevanten Systemfunktionen ausgelöst werden.

Einen anderen Ansatz verfolgt das Werkzeug *TDE/UML*, das von Siemens Corporate Research entwickelt wurde und noch immer erweitert wird [Vie05, HVFR05]. Im Kern erzeugt es Testfälle aus UML-Aktivitätsdiagrammen, die das Systemverhalten repräsentieren. Mit Hilfe von Pfadüberdeckungsalgorithmen werden Testfälle derart erzeugt, dass alle Pfade des Diagramm erfasst werden.

Das Besondere an TDE/UML ist allerdings, dass diese traditionelle Methode der Pfadüberdeckungstests mit Methoden der Äquivalenzklassenanalyse kombiniert wird. Konkret geschieht das durch Annotation des Verhaltensdiagramms mit sogenannten *Parametern* (so die Bezeichnung in TDE/UML), die ihre möglichen Werte aus Klassifikationsmengen beziehen. Ein Element einer solchen Menge entspricht dabei einer Äquivalenzklasse der zugehörigen Einflussgröße.

Bei der Testfallableitung wird also nicht nur die Pfadüberdeckung, sondern auch die Datenüberdeckung berücksichtigt. Entsprechend können einzelne Pfade für verschiedene Datensätze mehrmals als Testfall durchlaufen werden.

Abbildung 6.13 zeigt ein Beispiel aus einer projektbegleitenden TDE-Fallstudie bei Siemens Transportation Systems. Für Labor- und Feldtests auf Systemebene wurden vor der Inbetriebnahme einer neuen Zugstrecke betriebliche Situationen und Anwendungsfälle in TDE-UML modelliert. Mit Hilfe der Pfad- und Datenüberdeckung konnte daraus ein großer Satz an Testfällen erzeugt werden. Aus diesen Testfällen wurde dann eine repräsentative Untermenge ausgewählt, die einerseits in Papierform den offiziellen Abnahmedokumenten hinzugefügt wurde und die andererseits durch TDE/UML und einem hinterlegten Skripting-Framework in ausführbare Testfälle für die Siemens-Laborumgebung überführt wurde.

Die Erfahrungen aus dieser Fallstudie sind unter anderem, dass der Aufwand für die Auswahl geeigneter Testfälle – einige Dutzend unter Tausenden – anfangs unterschätzt wurde und dass die automatisch erzeugten Testskripte noch viel manuelle Nacharbeit erforderten.

Allerdings zeigte sich positiv, dass große Teile der erarbeiteten Methoden und Datensätze mit geringem Aufwand in einem Nachfolgeprojekt eingesetzt werden konnten und sich die Folgen des höheren Erstaufwands dadurch schon im zweiten Projekt relativierten.

Begünstigt wurde das Projekt durch den Umstand, dass die betrieblichen Szenarien schon in einer semiformalen Form vorlagen, so dass deren Überführung in TDE- bzw. UML-Aktivitätsdiagramme mit geringem Aufwand möglich war. Generell unterstreicht das einen Aspekt, der später in einer Fallstudie zum Thema „SCADE“ noch einmal genauer aufgegriffen wird: der Einsatz formaler bzw. automatisierter Methoden verlagert den Aufwand vom Ende an den Anfang der Entwicklung, weil die Modellierung des Systems bzw. des Sollverhaltens Grundlage der nachfolgenden Schritte ist. Hier ist also management- und entwicklerseitig Geduld und ein langer Atem gefordert, weil sich durch dieses Vorgehen in der ersten Phase wenig Ergebnisse einstellen. Die Resultate werden erst in späteren Entwicklungs- und Testprozess anfallen, aber dann von höherer Güte sein. Und *insgesamt* ist – zumindest über mehrere Projekte – mit einer Reduzierung des Gesamtaufwands zu rechnen.

6.4.2 Testfallerzeugung durch Äquivalenzklassenbildung

Wie von Myers [MSBT04] beschrieben, können die Ein- und Ausgangsgrößen eines Systems in Äquivalenzklassen unterteilt werden (vgl. auch [OB88], *Category-Partition Method*). Bei der Entwicklung von Testfällen müssen dann prinzipiell nur noch ein repräsentativer Wert aus jeder Klasse und gegebenenfalls die Klassengrenzen berücksichtigt werden.

Da aber reale Systeme über viele Eingangsgrößen verfügen, die sich wiederum in viele Äquivalenzklassen unterteilen lassen, ist eine methodische Verfeinerung dieses Ansatzes samt geeigneter Werkzeugunterstützung notwendig. Das Ergebnis der entsprechenden Forschungsarbeiten ist der Ansatz der *Klassifikationsbäume*, der Anfang der 1990er Jahre von Grochtmann, Grimm und Wegener entwickelt wurde [GG93b, GG93a]. Sie bauten unter anderem auf Ostrands Arbeiten auf und erweiterten bestehende Ansätze um eine baumartige, hierarchische Darstellung der Äquivalenzklassenelemente bzw. Kategorien. Um eine effiziente Nutzung der Methode zu ermöglichen, programmierten sie außerdem ein Werkzeug (CTE, Classification Tree Editor), mit dem sich Klassifikationsbäume effizient modellieren und aus ihnen Testfällen ableiten lassen.

CTE unterstützt den Anwender allerdings nicht nur bei der Erstellung der Baumstrukturen, sondern auch bei der Kombination der Klassenelemente zu Testfällen und bei der Aneinanderreihung von Testfällen zu Testsequenzen. Ähnlich wie auch TDE/UML (s. vorangegangener Abschnitt) verschiedene Strategien der Testfallableitung kennt, um die Anzahl der entstehenden Testfälle zu beeinflussen, so implementiert auch CTE entsprechende Mechanismen.

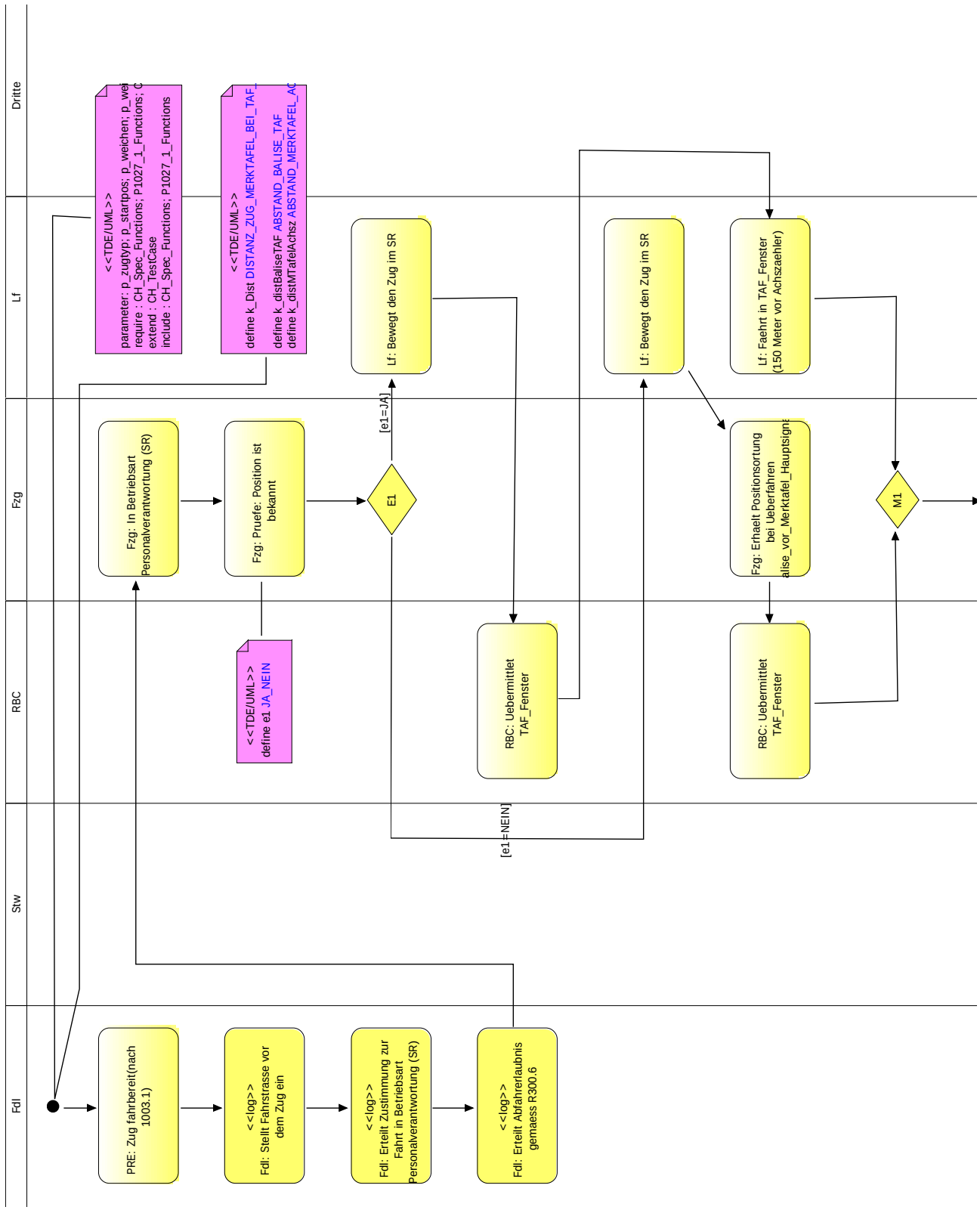

Abbildung 6.13: Ausschnitt eines Aktivitätsdiagramms zur Testfallerzeugung mit TDE/UML

Der erste Ansatz zur Verringerung der Testfallanzahl ist ein rein mathematisch-kombinatorischer Vorgang. Anstatt alle Werte aller Variablen vollständig miteinander zu kombinieren, werden nur noch alle Zweier- bzw. Dreierpaarungen von Werten gebildet, was ab drei bzw. vier Variablen bereits zu einer Reduzierung der Kombinationen und damit zu einer Reduzierung der Testfälle führt. Bei diesem Verfahren werden nach wie vor alle möglichen Werte aller Variablen getestet, aber – wie gesagt – nicht mehr in allen Kombinationen. In vielen Fällen ist das aber sicherlich ein akzeptabler Kompromiss.

Das zweite und sehr viel mächtigere Verfahren der Testfallreduzierung ist die Berücksichtigung logischer Abhängigkeiten zwischen Elementen aus verschiedenen Kategorien. Dies soll durch ein Beispiel erläutert werden: für den Test eines Taschenrechners werde eine Kategorienklasse „Operationen“ aufgestellt, die die Elemente $\{a + b, a - b, a \cdot b, a \div b, \sqrt{a \cdot b}\}$ enthält. Eine weitere Klasse, „Operanden“ enthalte die Kategorien $\{< 0, 0, > 0\}$. Diese Klasse sei als zwei Instanzen, „a“ und „b“, vertreten. Davon ausgehend, dass in der Wurzelberechnung der Fehlerfall „negativer Radikand“ bereits berücksichtigt und getestet wurde, können im vorliegenden Fall alle Kombinationen von Operationen und Operanden, die im Radizieren eines negativen Operanden münden würden, ausgeschlossen werden.

Zur Berücksichtigung derartiger Randbedingungen können in CTE Regeln unter Verwendung logischer Ausdrücke formuliert werden, die dann bei der Testfallableitung angewendet werden. Für das obige Beispiel würde sich die folgende Abhängigkeitsregel ergeben:

`(a /<0 XOR b /<0) => NOT Operation/`$\sqrt{a \cdot b}$

(Lies: ist genau ein Operand kleiner Null, folgt daraus, dass die Operation nicht das Radizieren sein darf; der Schrägstrich / trennt Instanzenname und Äquivalenzklasse)

Beide Verfahren, also die geänderte Kombinationsstrategie und die Abhängigkeitsregeln, können parallel verwendet werden und sind geeignet, die Zahl der erzeugten Testfälle deutlich zu reduzieren, wie das folgende Beispiel zeigen wird. Jedoch muss sich der Anwender bewusst sein, dass vor allem durch den Ausschluss „unlogischer“ Kombinationen auch immer Negativtestfälle zur Überprüfung der korrekten Fehlerbehandlung des Prüflings ausgeschlossen werden und somit prinzipiell die Testdichte unzulässig reduziert werden kann.

Testfallableitung für Stellwerke mit Klassifikationsbäumen

Im Rahmen dieser Arbeit wurde die Klassifikationsbaummethode unter anderem auf ihre grundsätzliche Eignung zur Ableitung von Testfällen für Stellwerke untersucht. Für die Untersuchung wurden einerseits ein einzelnes Signal aus dem deutschen KS-System (KS = Kombinationssignal) modelliert und andererseits die einfache in Abbildung 6.14 gezeigte Bahnhofs- und Streckentopologie als Klassifikationsbaum hinterlegt.

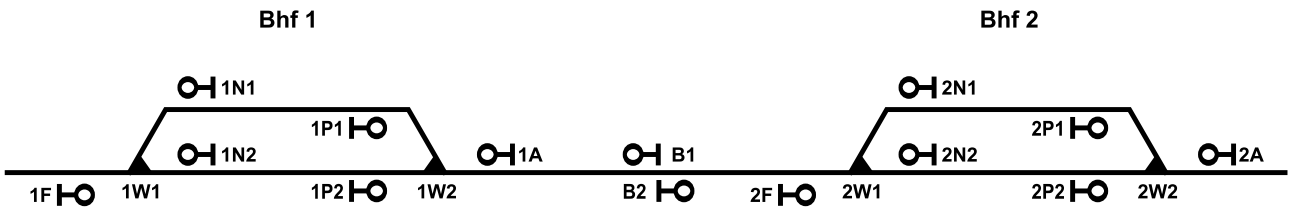

Abbildung 6.14: Einfache Testtopologie für die Ableitung von Stellwerkstests mit Hilfe von Klassifikationsbäumen

Für das alleinstehende KS-Signal wurden zwei mögliche Geschwindigkeits- und Geschwindigkeitsvoranzeiger angenommen (Zs3 und Zs3v). Ein möglicher Fahrtrichtungsanzeiger, der eher informativen denn sichernden Charakter hat, wurde nicht mit erfasst. Unter diesen Annahmen ergab sich der in Abbildung 6.15 gezeigte Klassifikationsbaum, aus dem sich ohne Anwendung einschränkender Abhängigkeitsregeln 2160 Parameterkombinationen ableiten lassen. Das entspricht dem Produkt der Elementanzahlen in den einzelnen Kategorien.

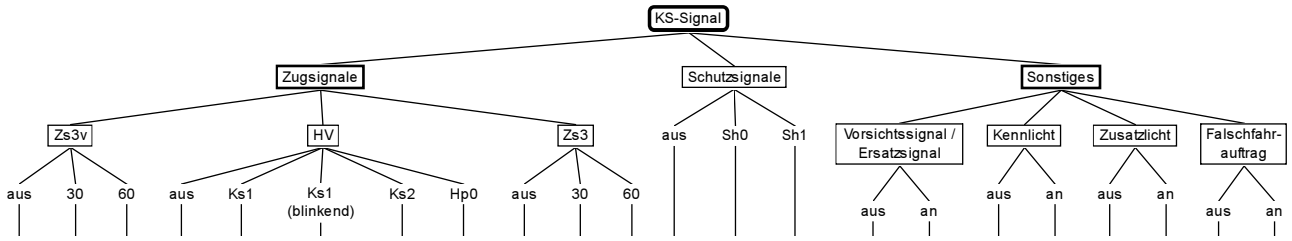

Abbildung 6.15: Klassifikationsbaum für ein alleinstehendes KS-Signal ohne Fahrtrichtungsanzeiger und mit zwei möglichen Geschwindigkeitssignalisierungen

Ein vollständiger Test eines einzelnen Signals in einem Stellwerk müsste also 2160 verschiedene Signalbegriffe umfassen. Dabei wird außer Acht gelassen, dass nicht alle Kombinationen wirklich sinnvoll sind. Beispiele für Randbedingungen an gültige Signalbegriffskombinationen:

- bei aktiviertem Kennlicht müssen alle anderen Anzeigen erloschen sein.
- bei aktiviertem Ersatz- oder Vorsichtssignal muss HP0 angezeigt werden
- bei SH0 muss HP0 angezeigt werden
- ...

Werden diese Abhängigkeiten in logischen Ausdrücken wie im obigen Taschenrechnerbeispiel erfasst und bei der Testfallerzeugung berücksichtigt, kann die Anzahl der resultierenden Testfälle von 2160 auf 182 reduziert werden. Durch Bildung aller Zweier- bzw. Dreierkombinationen anstelle der vollständigen Kombination aller Elemente kann die Zahl der Testfälle noch weiter auf 18 bzw. 43 gedrückt werden. Die Ausführung von 18 Testfällen für ein alleinstehendes Signal erscheint angemessen; durch die Ableitungsmethodik ist zudem sichergestellt, dass jeder Parameter jeden möglichen Wert annimmt und logisch sinnvolle Kombinationen von Parametern gebildet werden. Daher kann dies als sehr guter Kompromiss zwischen Aufwand und Testdichte angesehen werden.

Würden alle 14 Signale aus Abbildung 6.14 so detailliert erfasst wie das alleinstehende Signal aus Abbildung 6.15, ergeben sich rein rechnerisch $2160^{14} \approx 4,81 \cdot 10^{46}$ Testfälle. Ein System dieser Größe ist auch für den CTE nicht mehr handhabbar, selbst bei einer anschließenden Einschränkung der Testfälle durch Nebenbedingungen. Daher wurden für die Untersuchung des Bahnhofs die möglichen Signalbegriffe auf „Halt“ bzw. “Fahrt“ begrenzt. Vor dem Hintergrund, dass für dieses Testszenario weniger die Begriffe des Einzelsignals als vielmehr das Zusammenspiel verschiedener Signale und Weichen berücksichtigt werden soll, erscheint diese Vereinfachung legitim.

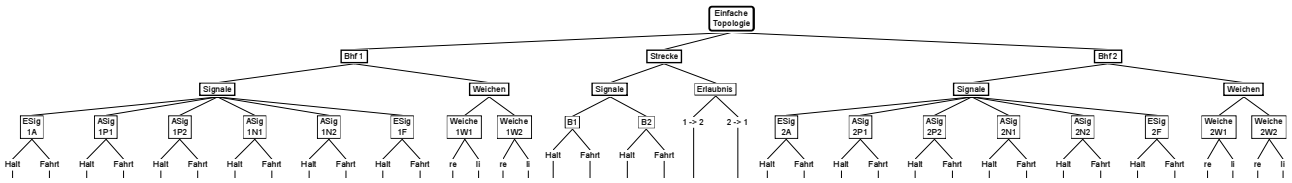

Abbildung 6.16: Klassifikationsbaum einer einfachen Topologie aus zwei Bahnhöfen mit eingleisiger Strecke, zwei Blocksignalen und Erlaubniswechsel

Das Ergebnis der Klassifikationsbaumerstellung zeigt Abbildung 6.16. Neben den eigentlichen Signalbegriffen wurden außerdem die Weichenlagen und die Erlaubnis – also die aktuell zulässige Fahrtrichtung – mit in den Baum aufgenommen.

	Alle Komb.	Mit Regeln	Regeln u. Zweierpaare	Regeln u. Dreierpaare
Einzelnes KS-Signal	2160	182	18	43
Testbahnhof	16384	336	240	–

Tabelle 6.1: Übersicht über die Anzahl der von CTE erzeugten Testfälle für die untersuchten Szenarien unter verschiedenen Konfigurationen

Die Abhängigkeitsregeln, die zur Erzeugung geeigneter Testfälle notwendig sind, entsprechen für diesen Anwendungsfall genau der in der Realität zugrunde liegenden Fahrstraßen- bzw. Blocklogik. Um beispielsweise das Signal `1P1` auf Fahrt stellen zu können, muss

- die Weiche `1W1` in Rechtslage verschlossen sein,
- das Signal `1P2` Halt zeigen
- das Signal `1A` Halt zeigen
- die Erlaubnis bei Bahnhof 1 liegen („1 → 2“)

Entsprechende Regeln wurden für alle Signale bzw. Weichen implementiert, wobei im gesamten Szenario keine Gleisfreimeldungen berücksichtigt werden (es wird immer davon ausgegangen, dass alle Gleise frei sind). Insgesamt wurden 18 Regeln im Modell hinterlegt, wobei jede Regel stets mehrere Parameter umfasst, wie das obige Beispiel zeigt.

Mit dem Ziel, alle Kombinationen von Signalbegriffen zu erfassen, werden ohne Berücksichtigung dieser Regeln $2^{14} = 16384$ Testfälle erzeugt. Unter Einbeziehung der Abhängigkeiten kann diese Zahl auf 336 verringert werden. Werden darüber hinaus nur Zweierkombinationen gebildet, kann die Zahl der resultierenden Fälle auf 240 verringert werden. Dreierkombinationen bringen keine weitere Reduzierung der Testfallanzahl, da die Signale mit „Halt“ und „Fahrt“ nur jeweils zwei verschiedene Werte annehmen können.

Die Tabelle 6.1 stellt alle ermittelten Testfallanzahlen nochmals in einer Übersicht dar. Es bleibt festzuhalten, dass durch Klassifikationsbäume und die entsprechende Unterstützung durch das Werkzeug CTE eine effektive Möglichkeit zur automatischen Testfallerzeugung gegeben ist, deren großer Vorteil vor allem durch die Berücksichtigung logischer Abhängigkeiten und die damit verbundene Testfallreduzierung ist. Dadurch wird eine der Herausforderungen beim Einsatz automatischer Testfallerzeugung – nämlich die anschließende Testfallauswahl – entscheidend vereinfacht. Aufgrund ihrer guten Eignung wurde die Klassifikationsbaummethode daher auch in der Fallstudie aus Kapitel 8 eingesetzt.

Die Grundlagen für diese Fallstudie werden auf den folgenden Seiten geschaffen. Sie zeigen, wie bisher in dieser Arbeit vorgestellten Modellierungs- und Beschreibungstechniken zu einer Methode vereinigt werden können, die Systementwurf, Systemtest, Testautomatisierung und Anforderungsmanagement unter dem Dach eines zentralen UML-Modells vereint.

7 Verknüpfung von System- und Testfallbeschreibungen in UML

Wie bereits in der Einleitung der Arbeit im Abschnitt 1.2 und besonders im Folgeabschnitt 1.3 dargelegt wurde, ist ein Kernproblem aktueller Leit- und Sicherungstechnikentwicklungen der hohe Testaufwand vor allem auf Integrations- und Systemebene, der aus

- *manueller Testausführung* resultiert, die wiederum aus
- *manueller Testfallbeschreibung* resultiert, die wiederum aus
- *manueller Testfallableitung* resultiert, die wiederum aus
- *informaler System- und Anforderungsbeschreibung* resultiert

Die Eigenschaften und Konsequenzen informaler Beschreibungsmittel für den Systementwurf wurden im Kapitel 3 genauer beleuchtet; besonders der in Abschnitt 3.2 erläuterte notwendige Sprachumfang für eine vollständige Systembeschreibung sei in diesem Zusammenhang nochmals erwähnt.

Der ungenügende Formalisierungsgrad in der Systembeschreibung behindert wiederum den effektiven Einsatz automatisierter Tests und Testfallableitungen, die ihrerseits Gegenstand von Kapitel 6 sind.

Daher soll auf den folgenden Seiten[1] ein Ansatz vorgestellt werden, der durch den konsequenten Einsatz der UML unter Verwendung geeigneter Profile zu einer konsistenten und homogenen System- und Testfallbeschreibung führt. Diese homogene Beschreibung kann ihrerseits eine geeignete Grundlage für weiterführende Techniken wie Codeerzeugung und Testautomatisierung darstellen. Dieses Kapitel beschränkt sich dabei auf die methodischen bzw. theoretischen Grundüberlegungen. Eine praktische Umsetzung der hier vorgestellten Ideen zeigt die Fallstudie im Folgekapitel.

7.1 Lösungsansatz

Bereits im Zusammenhang mit Abbildung 2.3 auf Seite 12 wurde kurz auf die zunehmende Formalisierung im Laufe eines Systementwurfs eingegangen. Abbildung 7.1 greift diesen Gedanken noch einmal auf und stellt ihn aus Datensicht dar: am Anfang der Entwicklung stehen informale und häufig grobe Anforderungsdokumente, die dann in überwiegend textuell geprägten Abläufen zunehmend konkretisiert werden, bis schließlich der Sprung von der textuellen (Prosa-)Ebene direkt auf die Codeebene erfolgt.

Vor allem in der IT-Domäne wird dieser Ablauf bereits heute durch den intensiven Einsatz von UML unterstützt (daher der zweite, parallele UML-Pfad in Abb. 7.1). Allerdings ist deren Anwendung auf die reine Systementwicklung beschränkt (beispielsweise für die Erstellung der Softwarearchitektur); der Einsatz der UML für Testzwecke hat nach wie vor keinerlei Praxisrelevanz.

Der hier vorgestellte Ansatz – dargestellt in Abbildung 7.2 – soll die Lücke zwischen informalen Kundenanforderungen und formalem Code / formalen Testskripten mit kombinierten UML-Modellen für System- und

[1] Die folgenden Abschnitte basieren auf bzw. verwenden Materialien aus [KL07a] und [KL07b]

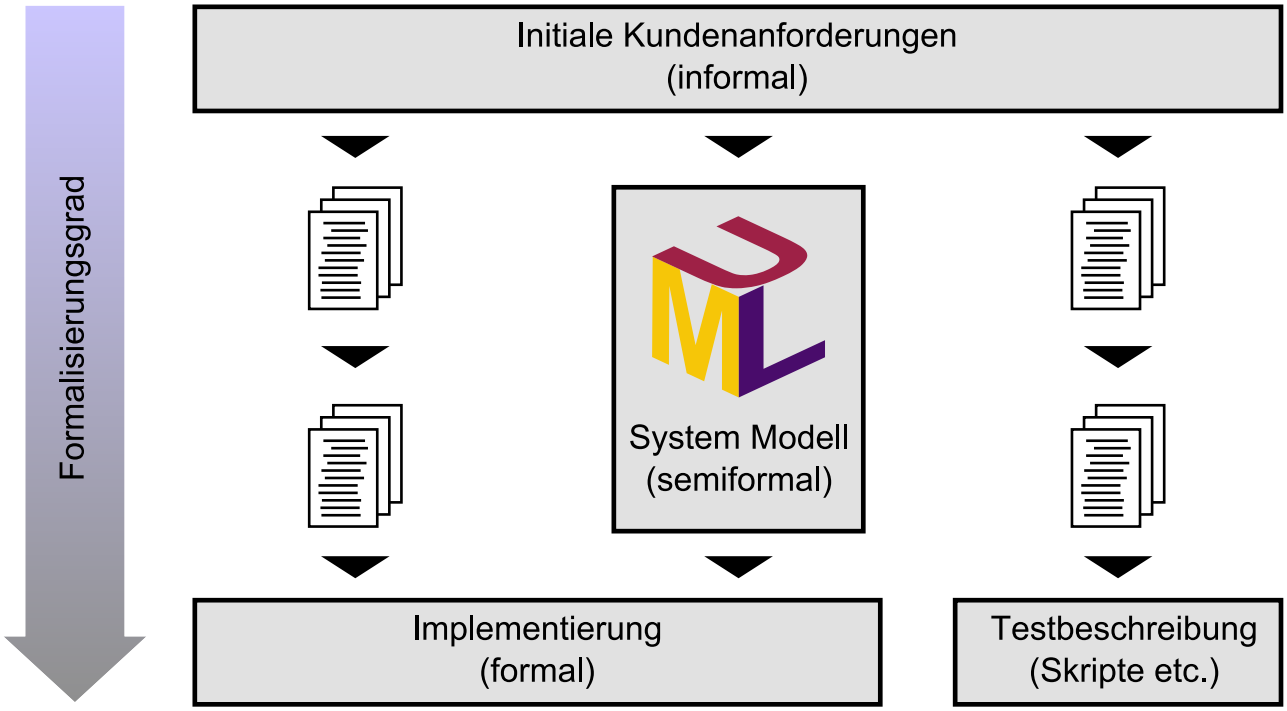

Abbildung 7.1: Verbreitete Arbeitsabläufe in der Softwareentwicklung: überwiegend textuelle Systementwicklung, mitunter Einsatz von UML zur Systembeschreibung, keine formalen Methoden in der Testbeschreibung

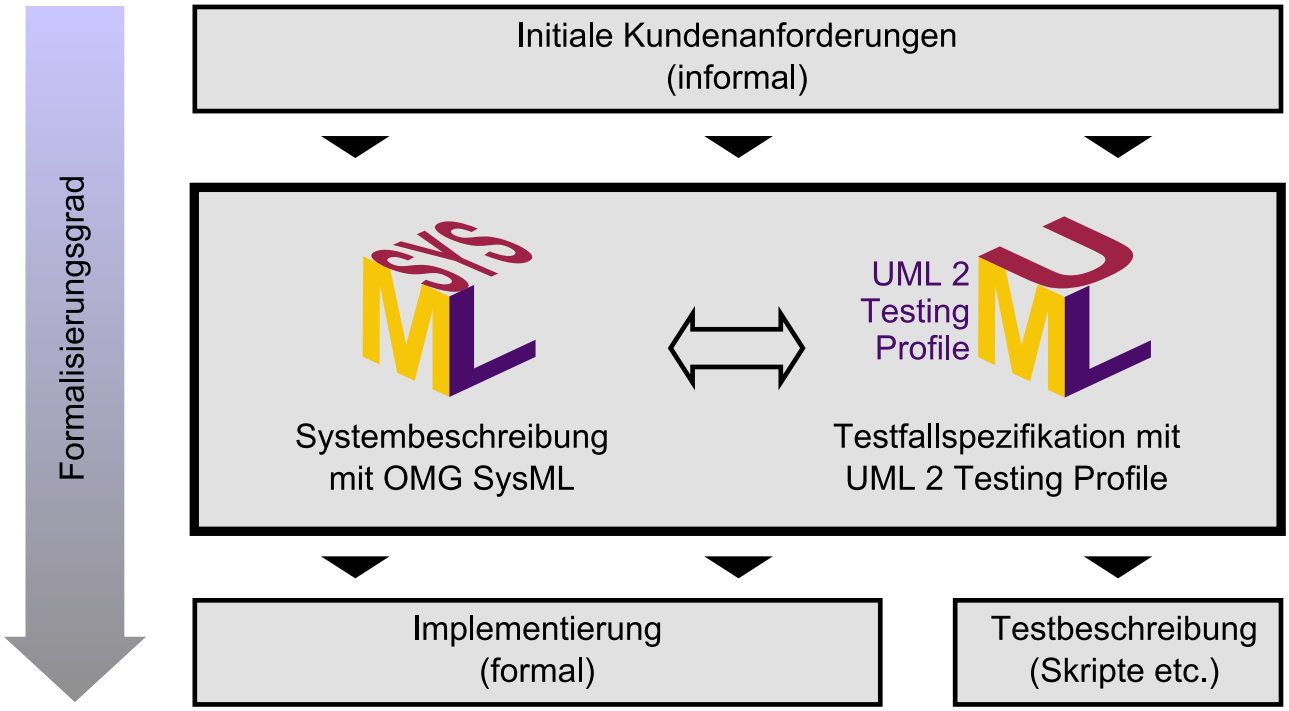

Abbildung 7.2: Ansatz eines verbesserten Arbeitsablaufs: UML ersetzt informale, textuelle Dokumente

Testbeschreibungen schließen. Während bisherige Verfahren vor allem Software- oder allenfalls Systemmodelle in UML abbilden, sollen hier gezielt auch Anforderungen und Testfälle mit Hilfe der UML erfasst und zu einem konsisten Gesamtmodell vereinigt werden.

Dadurch wird das Modell für Entwickler und Tester von einer frühen Entwicklungsstufe an zum zentralen Element des gesamten Prozesses, was eine Reihe von Vorteilen mit sich bringt:

- Anforderungen werden nur einmal zentral erfasst und können dann über das gesamte Modell bzw. über den gesamten Prozess konsistent referenziert werden.
- Es wird nur ein einziges Beschreibungsmittel eingesetzt und es entsteht ein einziges, homogenes Modell.
- Eine direkte Verfolgbarkeit von Anforderungen in horizontaler wie in vertikaler Richtung ist gegeben (d. h. von der Anforderung zur Implementierung und von der Anforderung auf den Testfall; „horizontal“ und „vertikal“ bezieht sich hier auf einen V-förmigen Entwicklungsprozess).
- Entwickler und Tester arbeiten enger zusammen, ohne dabei ihre Unabhängigkeit zu riskieren (wichtig für sicherheitsrelevante Systeme!).
- Viele textuelle Dokumente – beispielsweise Architektur-, Design- oder Testbeschreibungen – werden durch das Modell ersetzt.
- Ein formaleres, konsistentes Modell mit Entwicklungs- und Testspezifikationen vereinfacht Verifizierung und Validierung und ist eine tragfähige Grundlage für Zulassungsprozesse und Sicherheitsnachweise.

Viele der genannten Vorteile beziehen sich auf Anforderungen und ihre Nachverfolgbarkeit innerhalb des Modells. Daher soll in den folgenden Abschnitten zunächst auf den Umgang mit Anforderungen in kombinierten SysML-/U2TP-Projekten eingegangen werden. Anschließend werden Ansätze zur Codeerzeugung und Testskripterzeugung vorgestellt.

Insgesamt bestehen die wesentlichen methodischen Neuerungen, die mit dieser Arbeit eingeführt und auf den folgenden Seiten genauer beschrieben werden, damit aus:

- der Idee zur Kombination der an sich autarken UML-Profile SysML und U2TP zu einer einheitlichen und umfassenden UML-gestützten Systembeschreibung
- der Verfolgung von Anforderungen zwischen den verschiedenen Modellteilen und der Auswertung von Anforderungsbeziehungen z. B. für Aussagen über die Testabdeckung
- einem detaillierten Vorschlag für die Abbildung von U2TP auf TTCN-3, der über das generische Mapping der U2TP-Spezifikation [Obj05b] hinausgeht

7.2 Anforderungsverwaltung und -verfolgung

Am Ende von Kapitel 5 wurde bereits auf die Wichtigkeit der Anforderungsverwaltung und -verfolgung eingegangen. Ihr kommt – vor allem bei sicherheitsrelevanten Systemen – unter mehreren Aspekten besondere Bedeutung zu. Unter anderem muss das Anforderungsmanagement folgende Fragen beantworten können:

Produktqualität: Wurde alles wie vorgesehen bzw. wie zugesagt implementiert? Wie hoch ist die Testabdeckung?

Haftungsfragen: Sind *alle* Funktionen mit der *notwendigen* Sicherheit implementiert?

Zeit- und Kostenplanung: Wie weit ist die Entwicklung?

Die rein manuelle bzw. textuelle Verwaltung der Anforderungen ist bei größeren Projekten faktisch nicht mehr realisierbar. Zwar existieren zahlreiche Werkzeuge (z. B. DOORS von Telelogic) und noch mehr Individuallösungen (beispielsweise basierend auf Tabellenkalkulationen oder Textverarbeitungen), aber alle diese Systeme stehen vor der Hürde des Notationsbruchs beim Übergang auf die Implementierungsebene (Code). Konkret sei hier das Problem der Referenzierung zwischen Anforderungen und sich dynamisch wandelnden Codeteilen genannt. Diese Art der Referenzierung auf Implementierungsartefakte ist jedoch für ein durchgängiges Anforderungsmanagement zwingend erforderlich.

Ebenso ist mit gängigen Methoden die Referenzierung auf Testfälle schwierig. Liegen die Testfälle in informaler, textueller Form vor, sind sie zwar leicht zu referenzieren, aber ungeeignet als Grundlage für Testautomatisierung. Sind sie umgekehrt in formalisierter Form erfasst, stellt sich ebenso wie bei Programmcode die Herausforderung des Notationsbruchs.

Die folgenden Abschnitte werden erläutern, wie durch Abbildung von Anforderungen, Testfällen und Systemeigenschaften in UML die aufgezeigten Schwierigkeiten gelöst werden können. Zum einen lassen sich in einem kombinierten Anforderungs-, Testfall- und Systemmodell Referenzierung zwischen verschiedenen Modellteilen einfach realisieren und zum anderen ist ein solches Modell eine geeignete Grundlage für Testautomatisierung und Codeerzeugung. Darüber hinaus lassen sich die Anforderungsbeziehungen automatisiert auswerten, wie später in dieser Arbeit noch gezeigt wird.

Ausgangspunkt der folgenden Untersuchungen zur Anforderungsverfolgung ist ein klassischer V-förmiger Entwicklungsprozess, wie er typischerweise für die Entwicklung sicherheitsrelevanter Systeme eingesetzt wird. In einem solchen Prozess werden die Systemanforderungen zu Beginn erfasst (oben links in Abb. 7.3) und werden dann nach und nach in konkreten Designentscheidungen oder in der Implementierung umgesetzt. Anschließend werden die entwickelten Module integriert und getestet (rechte Hälfte von Abb. 7.3).

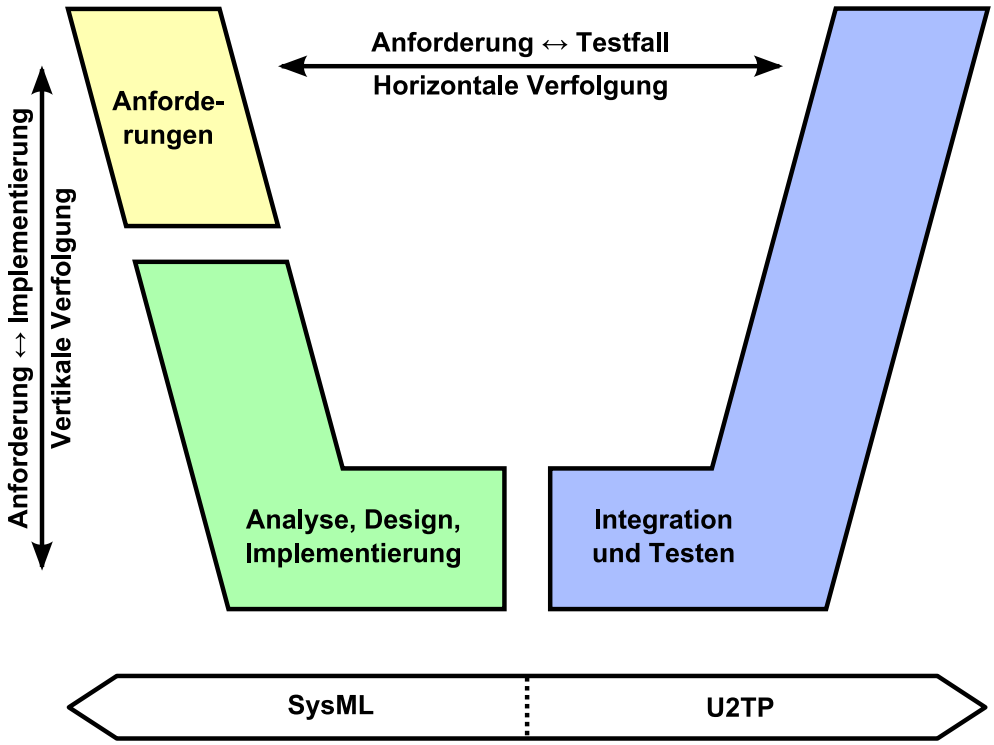

Abbildung 7.3: Komponenten des gemeinsamen SysML- / U2TP-Modells und die Rückverfolgung von Anforderungen

Um eine vollständige Implementierung und eine ausreichende Testabdeckung zu bestimmen, müssen Anforderungen in horizontaler wie in vertikaler Richtung nachverfolgbar sein. In vertikaler Richtung wird die zunehmende Konkretisierung bzw. Implementierung einer Anforderung verfolgt. Insbesondere ist hier die Verknüpfung mehrerer „Low-Level"-Anforderungen (beispielsweise einer Modulspezifikation) mit einer oder mehreren „High-Level"-Anforderungen zu berücksichtigen. Durch die horizontale Zuordnung soll sichergestellt sein, dass jede Anforderung auch getestet wird. Daher bildet sie (System-)Anforderungen auf einen oder mehrere Testfälle ab. Wie schon die Vertikalbeziehung so stellt auch die Horizontalbeziehung eine „many-to-many"-Abbildung dar, weil jede Anforderung mindestens einem Testfall zugeordnet werden muss und umgekehrt.

Für die Darstellung von Anforderungen und Anforderungsbeziehungen wird auf die Möglichkeiten des SysML-Profils zurückgegriffen. Die Darstellung basiert auf einer speziellen Variante von Klassendiagrammen, den sogenannten Anforderungsdiagrammen. Die Anforderung als solche wird darin durch einen speziellen Stereotypen für Klassen gekennzeichnet. Die Repräsentation von Anforderungen durch Klassen erlaubt auch die Darstellung verschachtelter Anforderungen durch Ausnutzung sogenannter „nested classes" und anderen Strukturierungsmöglichkeiten in Klassendiagrammen. Beziehungen zwischen Anforderungen und anderen Modellementen können als sogenannte „Dependency" abgelegt werden. Beide Aspekte, die Anforderungsstrukturierung und die Beziehungen zwischen Anforderungen, sollen in den folgenden Abschnitten genauer analysiert und veranschaulicht werden.

7.2.1 „Vertikales" Nachverfolgen von Anforderungen

Abbildung 7.4 zeigt ein Beispiel für ein Anforderungsdiagramm, das einige funktionale High-Level-Anforderungen und deren interne Beziehungen darstellt. Die Anforderung an der Spitze („Coupling to RailSiTe") wird durch verschachtelte Klassen auf Unteranforderungen heruntergebrochen. Andere Anforderungen aus der Abbildung wurden nicht durch Verfeinerung abgeleitet, sondern sind aus anderen, weiterführenden Überlegungen hervorgegangen. Dies wird im Diagramm durch die Stereotypen `≪DeriveReqt≫` zwischen den betroffenen Klassen kenntlich gemacht.

In der ursprünglichen SysML gemäß [Obj06a] wird nur ein einziger Stereotyp – `≪Requirement≫` – für alle Arten von Anforderungen vorgesehen. Während der Modellierung stellte sich jedoch heraus, dass eine weitere Unterteilung in `≪Requirement_SRS≫` (für Systemanforderungen), `≪Requirement_FRS≫` (für funktionale Anforderungen) und `≪Requirement_RAMS≫` (für nicht-funktionale RAMS(S)-Anforderungen) sinnvoll und nützlich ist. Diese verfeinerte Typisierung von Anforderungen erhöht die Lesbarkeit der Diagramme und erleichtert das Erkennen der logischen Beziehungen zwischen Anforderungen. Durch diese „neuen" Stereotypen verwendet das Diagramm aus Abbildung 7.4 eine leicht abgeänderte SysML-Notation.

Ein Beispiel für die Abbildung funktionaler Anforderungen auf Systemanforderungen ist in Abbildung 7.5 dargestellt. Der SRS-Stereotyp macht die neue Anforderungsklasse kenntlich, während die Beziehung zwischen den Anforderungen durch den Einsatz einer UML-Abhängigkeit (*dependency relationship*) hergestellt wird, die mit `≪Satisfy≫` stereotypiert ist.

Außer der reinen Darstellung von Beziehungen zwischen Anforderungen ist auch die Verbindung von Anforderungs- und Implementierungselementen möglich, wie Abbildung 7.6 zeigt. Anstelle der Verwendung einer stereotypierten Abhängigkeit wurde ein *SatisfyCallout* in das Diagramm aufgenommen. Dieser wird durch einen Textrahmen ähnlich einer Kommentarbox dargestellt, der das Schlüsselwort `Satisfies` zusammen mit einem Verweis auf die Anforderungen enthält. Dieser Rahmen wird mit allen Elementen verbunden, die zur Umsetzung der Anforderung beitragen. Darüber hinaus zeigt Abbildung 7.6 die SysML-konforme Darstellung innerer Strukturen und Datenflüsse des modellierten Systems. Dieser WhiteBox-Blick wird durch die *Internal Block Diagrams* der SysML realisiert, die auf UML-Kompositionsstrukturdiagrammen beruhen.

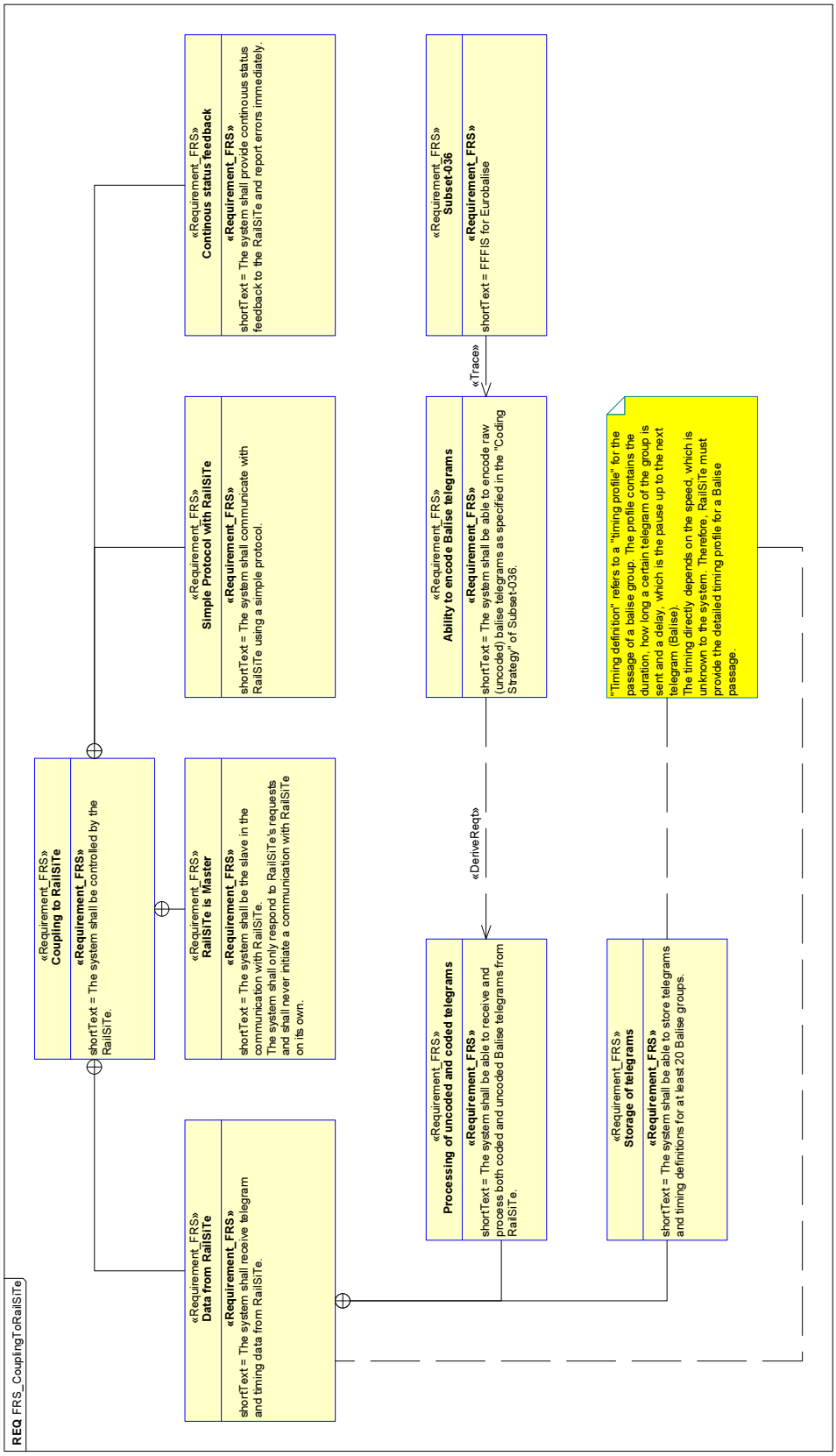

Abbildung 7.4: Beispiel für ein SysML-Anforderungsdiagramm mit funktionalen, verschachtelten Anforderungen

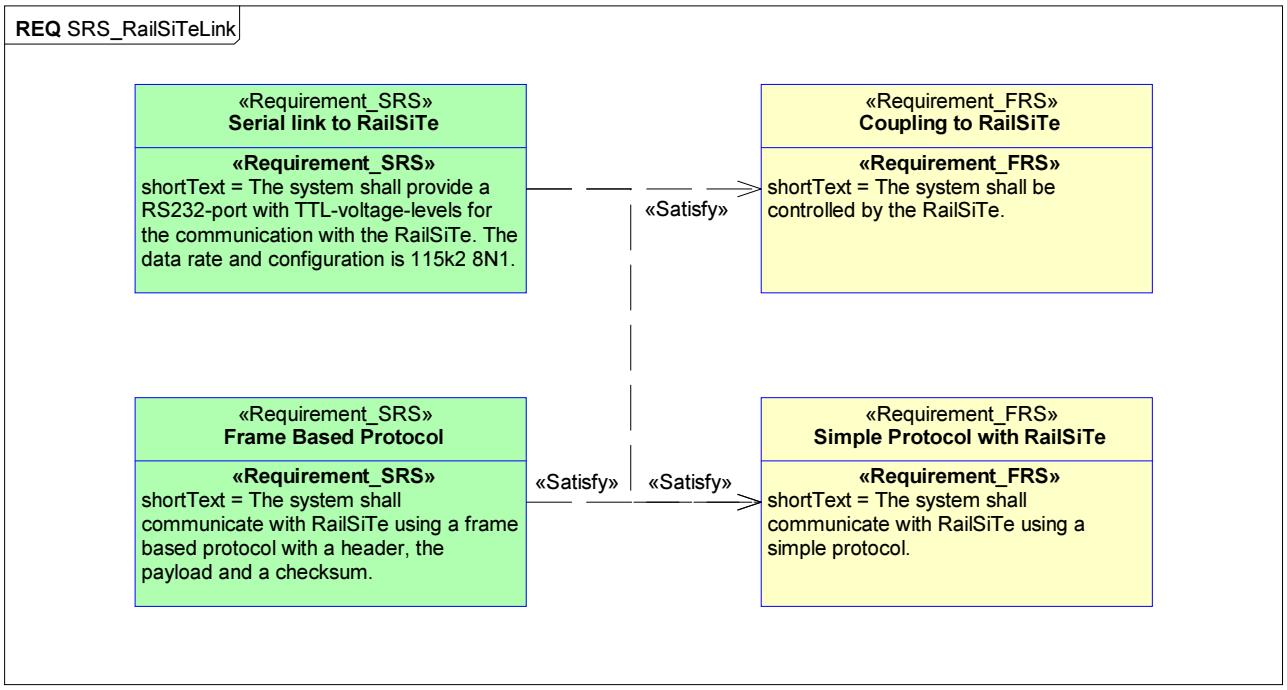

Abbildung 7.5: Beispiel für die Zuordnung zwischen funktionalen Anforderungen und Systemanforderungen

7.2.2 „Horizontales“ Nachverfolgen von Anforderungen

Bisher wurden nur die Beziehungen zwischen Anforderungen, anderen Anforderungen und Designelementen berücksichtigt (Vertikalbeziehungen in Abbildung 7.3). Damit das UML-Modell als vollständige und konsistente Basis eines Entwicklungsprozesses verwendet werden kann – und genau das ist das Ziel dieser Arbeit – muss auch eine Beziehung zwischen Anforderungen und Testfällen ermöglicht werden (Horizontalbeziehung in Abbildung 7.3).

Die U2TP-Notation eines Testfalls ist eine Methode, die einer als Testkontext stereotypierten Klasse zugeordnet ist (siehe auch Abb. 6.6 auf Seite 77). Das Verhalten des Testfalls bzw. der Methode wird durch weitere Diagramme, wie z. B. Sequenzdiagramme, konkretisiert.

Für die Beziehung zwischen einer Anforderung (Klasse) und einem Testfall (Methode) stellt SysML den *VerifyCallout* zur Verfügung. Seine grafische Darstellung ähnelt der des SatisfyCallouts in Abbildung 7.6. Exemplarisch zeigt Abbildung 7.7 die Verknüpfung des VerifyCallouts mit den relevanten Elementen und der Referenz auf den Testfall.

Eine alternative Darstellung wäre ein mit `«verify»` stereotypierter Abhängigkeitspfeil (*dependency*), aber da Abhängigkeiten nicht direkt mit Methoden, sondern nur mit Klassen verbunden werden können, ist diese Art der Darstellung hier nicht anwendbar.

Aufgrund des Aufbaus des U2TP ist der Verweis auf die Testfallmethode ausreichend, um alle testrelevanten Parameter zu spezifizieren: das Verhalten wird durch das zur Methode gehörige Sequenzdiagramm bestimmt und die Methode selbst gehört zu einem Testkontext, der aus den notwendigen Testkomponenten, Datenpools (also den Testvektoren), dem Arbiter und dem Scheduler zusammengesetzt ist.

Diese indirekte Verknüpfung zu den Eigenschaften der Testumgebung schafft eine semantisch sehr viel stärkere Beziehung zwischen Anforderung und Testfall als das in „traditionellen“ Anforderungsverfolgungen der Fall ist. Denn letztere beruhen – wenn sie denn überhaupt eingesetzt werden! – auf einem Verweis auf eine Tabelle mit Testschritten. Eine detaillierte Verhaltens- und Strukturinformation der Testanordnung wird daher mit solch traditionellen Methoden nicht erreicht. Entsprechend weit interpretierbar, unvollständig und

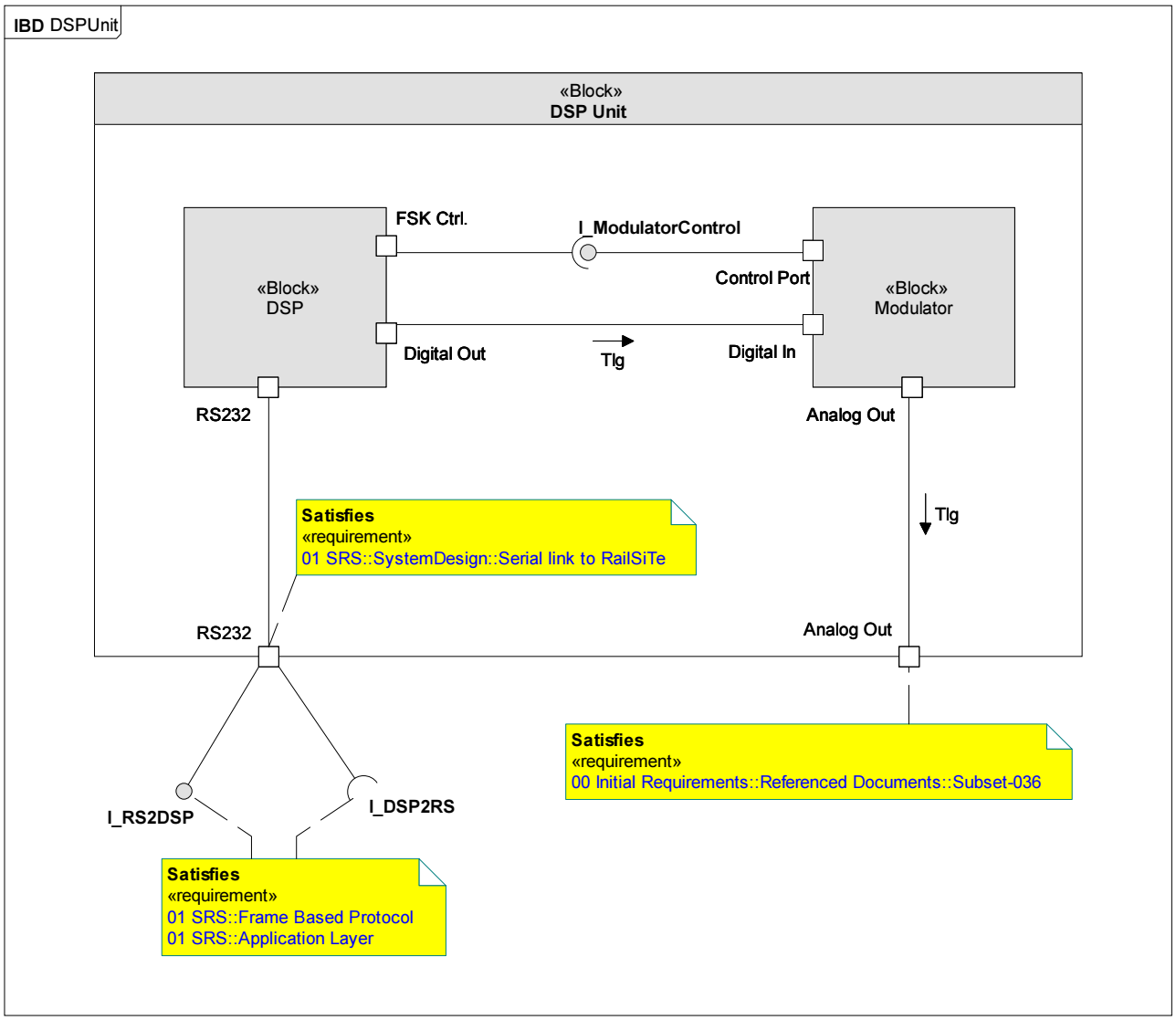

Abbildung 7.6: „Internal Block Diagram“ der SysML mit WhiteBox-Sicht auf das System und einer Verknüpfung zwischen Anforderungs- und Implementierungselementen

schlecht reproduzierbar sind die daraus resultierenden Testfälle.

7.2.3 Auswertung der Anforderungsbeziehungen

Neben der reinen Darstellung der Anforderungen und ihrer Beziehungen ist für eine effektive Unterstützung des Entwicklungsprozesses auch die automatische Analyse von Anforderungsbeziehungen entscheidend. Sie ermöglicht, beispielsweise nicht implementierte oder nicht getestete Anforderungen zu identifizieren oder die Auswirkung von Anforderungsänderungen während der Designphase abzuschätzen. Daher muss auch die Zuordnung zwischen Anforderung, Design / Implementierung und Testfall bidirektional sein.

Eine solche Analyse sollte die folgenden Aspekte berücksichtigen und aus einem UML-Modell herausfiltern:

- alle Anforderungen
- alle Ziele einer `≪Satisfy≫`-Beziehung
- alle Ziele einer `≪Verify≫`-Beziehung

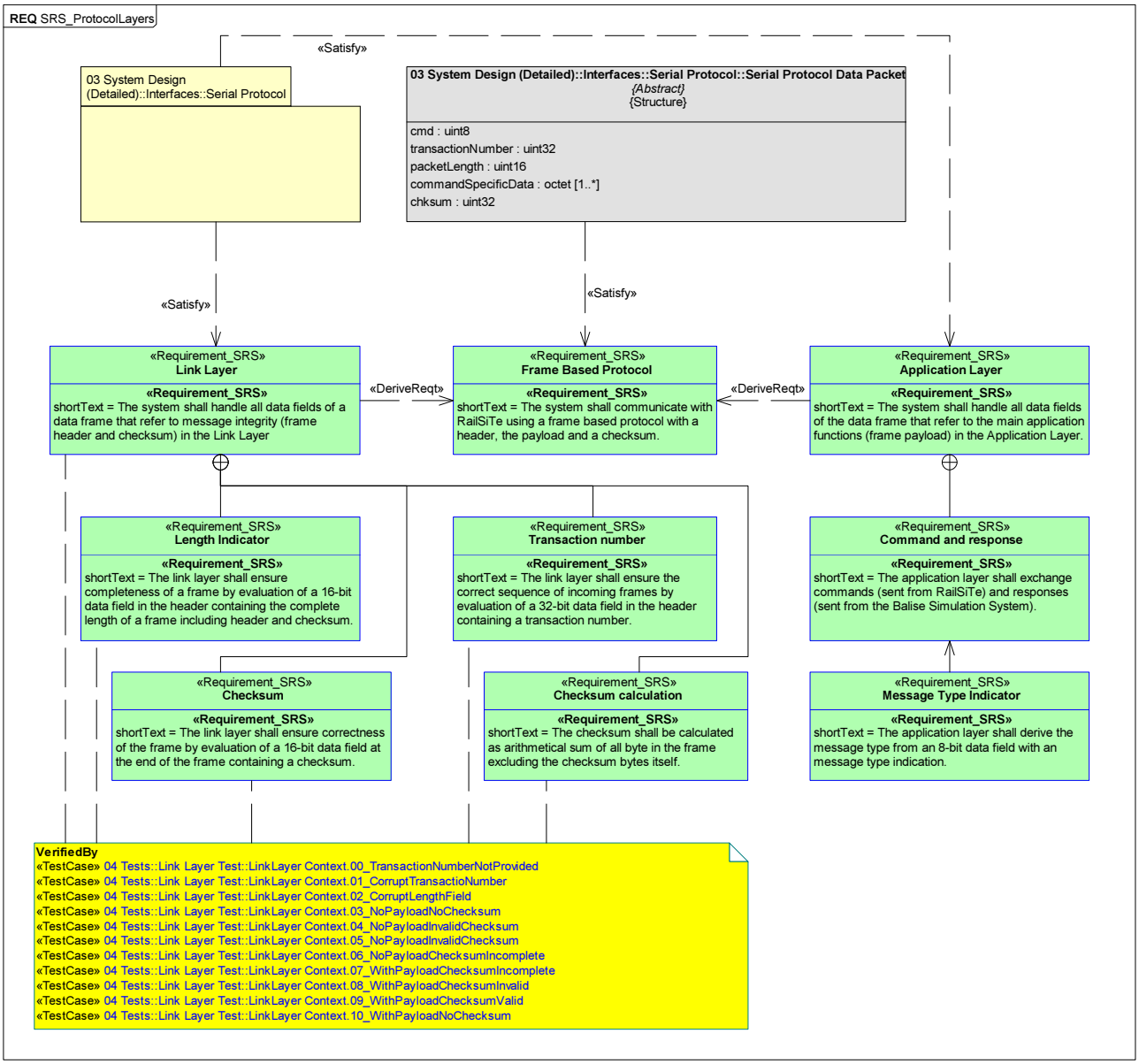

Abbildung 7.7: Beispiel einer UML-Verknüpfung zwischen SysML-Anforderung und U2TP-Testfall

- alle Zuordnungen der Form: Anforderung → `≪Satisfy≫`- / `≪Verify≫`-Ziel
- alle Zuordnungen der Form: `≪Satisfy≫`- / `≪Verify≫`-Ziel → Anforderung
- alle Anforderungen ohne `≪Satisfy≫`-Abhängigkeit
- alle Anforderungen ohne `≪Verify≫`-Abhängigkeit
- alle Testfälle, die keiner Anforderung zugeordnet sind

Die Realisierung einer solchen Analyse ist auf vielfache Weise denkbar. Ein weitgehend werkzeugunabhängiger Ansatz wäre der XMI-Export des UML-Modells und die nachgelagerte Analyse der XMI-Datei. Dieser Weg hat jedoch den entscheidenden Nachteil, dass die zuvor erwähnten Callouts, die ebenfalls auszuwertende Informationen enthalten, nicht nach XMI exportiert werden, da sie aus Metamodellsicht lediglich Kommentarfelder darstellen. Genauer gesagt bedient sich SysML der Notation von Kommentarfeldern und ergänzt sie

mit einer neuen Semantik (Callouts). Selbstverständlich ist einem normalen UML-Werkzeug diese erweiterte Semantik nicht bewusst, so dass es die Callouts weiterhin wie Kommentare behandelt.

Eine bessere Möglichkeit, die Analysefunktionen umzusetzen, ist daher der werkzeugspezifische Direktzugriff auf Datenbanken und Dateien des Modelers. Auf diese Weise wird der Zugriff auf alle Modellelemente einschließlich Kommentarfeldern gewährleistet. Die Fallstudie im folgenden Kapitel zeigt, wie ein solcher Zugriff für das CASE-Tool Artisan Studio realisiert wurde. Abgesehen vom grundsätzlichen Modellzugriff wurden natürlich auch die erwähnten Analysefunktionen realisiert.

Ergebnis der Anforderungsverwaltung in UML

Die letzten Abschnitte haben gezeigt, dass Anforderungen und Anforderungsbeziehungen auf geeignete Weise in einem UML-Modell abgelegt werden können. Das Modell ist nicht nur in der Lage, die Struktur von Anforderungen zu erfassen, sondern auch, Referenzen zu Implementierungselementen und Testfällen abzubilden. Durch die automatische Analysierbarkeit dieser Referenzen wird dem Entwickler ein wichtiges Werkzeug an die Hand gegeben, um den Implementierungsgrad, die Testabdeckung und die Modellkonsistenz zu bestimmen.

Die Besonderheit bei der Verwendung der UML ist, dass diese Vorteile ohne den eingangs des Kapitels hervorgehobenen Notationsbruch zwischen Anforderungs-, System- und Testfalldarstellung erreicht werden. Verglichen mit informalem Text, durch dessen durchgängige Verwendung ebenfalls den Notationsbruch vermieden werden kann, verfügt die UML aber über den Vorzug der automatischen Codeerzeugung, auf den die folgenden Seiten im Einzelnen eingehen werden.

7.3 Codeerzeugung aus dem UML-Modell

Prinzipiell besteht die Möglichkeit, aus UML-Modellen ausführbaren Code zu erzeugen. Die Möglichkeiten der Codeerzeugung sind jedoch stark vom verwendeten UML-Werkzeug abhängig und sind daher nicht einheitlich zu beschreiben.

Die meisten CASE-Tools unterstützen zumindest die Erzeugung von Klassendeklarationen und Funktionsköpfen aus den im Modell hinterlegten Klassendiagrammen. Dies schließt häufig die Berücksichtigung von Assoziationen und Rollen in Form von Variablen (oder Zeigern) im erzeugten Code mit ein.

Abgesehen von der Übersetzung rein statischer Struktur- und Klasseninformationen in Code unterstützen manche Programme auch die Umwandlung bestimmter Verhaltensdiagramme in Quelltext. In diesem Zusammenhang sind vor allem Zustandsdiagramme zu nennen, aus denen Code für das Verhalten einer Klasse abgeleitet werden kann. Sequenz- oder Aktivitätsdiagramme werden aufgrund ihrer Komplexität bzw. breiten semantischen Interpretierbarkeit von keinem Tool zur Codeerzeugung herangezogen.

Die pure „Automatenlogik“, die in einem Zustandsdiagramm hinterlegt ist (also das reine Fortschalten der Zustände unter bestimmen Bedingungen oder Ereignissen) ist jedoch nicht ausreichend, um ein wirklich sinnvolles Programm zu erzeugen. Vielmehr müssen die Zustände bzw. Transitionen noch mit Aktionen hinterlegt werden, die für Ausgaben und Reaktionen des Programms sorgen. Diese Aktionen werden dann – je nach Modellierung – beim Schalten einer Transition, beim Betreten, beim Verharren oder beim Verlassen eines Zustandes ausgeführt. Die UML definiert zu diesem Zweck auf Metaebene [Obj07b] ein komplexes Konstrukt aus verhaltensbeschreibenden Klassen (*Behavior*, *BehavioralFeature*, *Actions*, *Activities* etc.) mit einer zugehörigen Semantik aus Lese- und Schreibzugriffen, Signalerzeugung, Signalempfang und weiteren grundlegenden Verhaltensmustern. In der Praxis erfolgt die Verhaltensspezifikation im UML-Zustandsdiagramm jedoch durch die direkte Eingabe von Code bzw. Funktionsaufrufen für die entsprechende Transition bzw. den entsprechenden Zustand. Inwieweit dieser direkt eingegebene Code dann auf das UML-Metamodell abgebildet wird ist wiederum werkzeugabhängig und auch weitestgehend vor dem Nutzer verborgen.

Als Zielsprachen für die Quelltextgenerierung bzw. die direkte Codeannotation an Verhaltenselementen kommen üblicherweise C/C++ oder Java zum Einsatz. Einige UML-Werkzeuge bieten die Möglichkeit, durch Konfiguration bestimmter Vorlagen (engl. *Templates*) die Ausgabe grundsätzlich an beliebige Programmiersprachen anzupassen. De facto wird dies jedoch häufiger eingesetzt, um innerhalb der gleichen Sprachfamilie Anpassungen an bestimmte Compiler oder Laufzeitumgebungen vorzunehmen.

Einschränkend muss erwähnt werden, dass durch das direkte Einfügen von Code in das UML-Modell dessen Analysierbarkeit (beispielsweise für Testfallableitungen) eingeschränkt wird. Andererseits würde es einen erheblichen Aufwand bedeuten, jedes elementare Programmkonstrukt, wie beispielsweise eine `if-then`-Anweisung, formal in UML zu modellieren. Die offensichtliche Lösung dieses Problems ist, die Modellierung an einer bestimmten Granularitätsstufe abzubrechen und auf die direkte Eingabe von Code überzugehen. Es sollte also nur soviel wie nötig modelliert werden und nicht soviel wie möglich. Wo die Schwelle zur direkten Quelltexteingabe liegt, kann allerdings nicht allgemein festgelegt werden und muss von Fall zu Fall abgewogen werden.

7.4 TTCN-3-Erzeugung aus dem UML-Modell

In Abschnitt 4.6 wurde bereits auf die Ähnlichkeit zwischen U2TP und TTCN-3 hingewiesen. Die grundsätzliche Abbildung von U2TP-Elementen auf TTCN-3-Konstrukte wird bereits in der U2TP-Spezifikation in tabellarischer Form und anhand eines Beispiels erläutert [Obj05b]. Allerdings enthält die U2TP-Spezifikation keine direkten Transformationsvorschriften für spezifische UML-Verhaltensdiagramme, sondern spricht nur allgemein von der Abbildung von „Stimuli“ auf TTCN-3-Nachrichten.

Einen Ansatz für eine mögliche Transformation von MSCs nach TTCN-3 stellt Ebner vor [Ebn04, Ebn05]. Aufgrund der strukturellen Ähnlichkeit zwischen MSCs und UML-Sequenzdiagrammen lässt sich dieses Mapping größtenteils auch für UML-Sequenzdiagramme verwenden. Die grundsätzliche Verwendung von Sequenzdiagrammen für Testfallbeschreibungen – allerdings ohne Bezug zu TTCN-3 oder U2TP – wird darüber hinaus von Rumpe ausführlich behandelt [Rum05].

Trotz der bestehenden wissenschaftlichen Vorarbeit hat eine von Janke [Jan07] durchgeführte Recherche allerdings keine (kommerziell) verfügbaren Werkzeuge ergeben, die die Abbildung von U2TP nach TTCN-3 realisieren.

Basierend auf Ebners Anregungen wurde daher für die Fallstudie ein TTCN-3-Generator erstellt, der aus U2TP-Modellen TTCN-3-Code erzeugt, der wiederum auf der von Janke [Jan07] erstellten Testumgebung ausführbar ist. Dadurch wird die Lücke zwischen Testfallmodell und Testausführung bzw. -automatisierung geschlossen.

Wie Abbildung 7.8 zeigt, beruht die Transformation auf Informationen aus Klassen-, Struktur- und Sequenzdiagrammen. Alle Daten werden direkt dem Modell entnommen, externe Dateien sind nicht erforderlich. Der zum Testfall aus Abbildung 7.8 gehörige TTCN-3-Code ist in Abbildung 7.9 wiedergegeben. Die Beziehung zwischen Modell- und Codeelementen ist aus Tabelle 7.1 ersichtlich.

Um sowohl eine möglichst schnelle Modellierung als auch eine unkomplizierte TTCN-3-Erzeugung zu gewährleisten, wurde das Datenpool-Konzept vom U2TP etwas vereinfacht umgesetzt. Die Testdaten werden zwar noch immer in Datenpools gesammelt, allerdings erfolgt der Zugriff nicht über Funktionen verschiedener Klassen dieses Pools. Vielmehr wurden für jeden Datensatz (Stimulus oder Prüflingsantwort) eine eigene Klasse angelegt und die zugehörigen Daten als Vorgabewert eines Attributs `data` abgespeichert. Eine solche Klasse wurde zusätzlich mit dem Stereotyp `≪TestVector≫` gekennzeichnet, der nicht Teil der U2TP-Spezifikation ist.

Im Modell sind einige grundlegende Konsistenzkriterien einzuhalten, um sicherzustellen, dass ein Testfall bzw. eine Testfallbeschreibung auch in TTCN-3-Code übersetzt werden kann. Zu diesen Kriterien gehört:

Nummer (Abb. 7.8)	UML-/U2TP-Element	TTCN-3	Zeile (Abb. 7.9)
1	Klasse (≪`TestContext`≫)	Modul	1
2a, 2b	Port, Porttyp, Lebenslinie	Port und phys. Schnittstelle	5 – 8
3	Assoziation zwischen Ports	Kommunikationsbeziehung	–
4a	Komponenten (≪`TestComponent`≫)	Komponente	15 – 19
4b	Prüfling (≪`SUT`≫)	Prüfling	–
5a, 5b	Parameter bzw. Konstanten	–	37
6	Klassenoperation (≪`TestCase`≫)	Testfall	24, 25
7	Zeitbedingung	Timer	36, 37, 47
8	Kommentar zum Sequenzschritt	Kommentar	34, 42
9	Event im Sequenzdiagramm	`send`- oder `receive`-Ereignis	40, 50

Tabelle 7.1: Die wichtigsten Transformationsbeziehungen zwischen U2TP (UML) und TTCN-3 (in Anlehnung an [Ebn04])

- Eine durch ≪`TestCase`≫ stereotypierte Methode muss zu einer ≪`TestContext`≫-Klasse gehören
- Jeder Testfall benötigt mindestens ein Verhaltensdiagramm (hier: ein Sequenzdiagramm), das dessen dynamisches Verhalten festlegt. Liegen mehrere Diagramme pro Testfall vor, werden diese in alphabetischer Reihenfolge ihrer Namen aneinandergereiht.
- Alle Lebenslinien eines Sequenzdiagramms müssen Ports sein. Dies sind in der Regel die Ports der beteiligten Testkomponenten oder des Prüflings.
- Jeder Port darf nur einmal pro Diagramm verwendet werden.
- Da auf nebenläufiges TTCN-3 verzichtet wird, darf nur eine Testkomponente pro Testfall verwendet werden.
- Jeder Port einer Testkomponente muss einen eigenen Typ aufweisen, der sich von denen der anderen Ports unterscheidet. Diese Einschränkung ist notwendig, weil aus dem Typ der Name der physikalisch zu verwendenden Schnittstelle abgeleitet wird und jede Schnittstelle natürlich nur einmal verwendet werden darf.
- Alle ausgetauschten Nachrichten müssen auf gültige Elemente eines Datenpools verweisen.
- Nachrichten dürfen nur entlang der im Strukturdiagramm des Testkontextes festgelegten Kommunikationsbeziehungen ausgetauscht werden.

Die praktische Umsetzung der hier nur grundsätzlich beschriebenen TTCN-3-Erzeugung wird im folgenden Kapitel im Rahmen der Fallstudie genauer erläutert. Die Umsetzung basiert im Wesentlichen auf der Abbildung der statischen Testfallinformationen in entsprechende Typdeklarationen im Kopf der TTCN-3-Datei (Nr. 1 bis 6 in Tabelle 7.1) und der anschließenden sequentiellen Auflistung aller Einzelschritte im `testcase`-Teil der Datei. Die Schritte werden dazu direkt aus den Events der Sequenzdiagramme erzeugt.

7.5 Zusammenfassende Darstellung der gezeigten Verfahren

Die zu Beginn des Kapitels beschriebene inhomogene Darstellung von Anforderungen, Systemeigenschaften, Implementierung und Testfällen hat verschiedene negative Konsequenzen. Als Beispiele seien die schlechte Abbildung von Anforderungsbeziehungen, die für Testautomatisierung ungeeigneten Testfallbeschreibungen und kaum realisierbare Analysen wie Testfall- oder Codeabdeckung genannt.

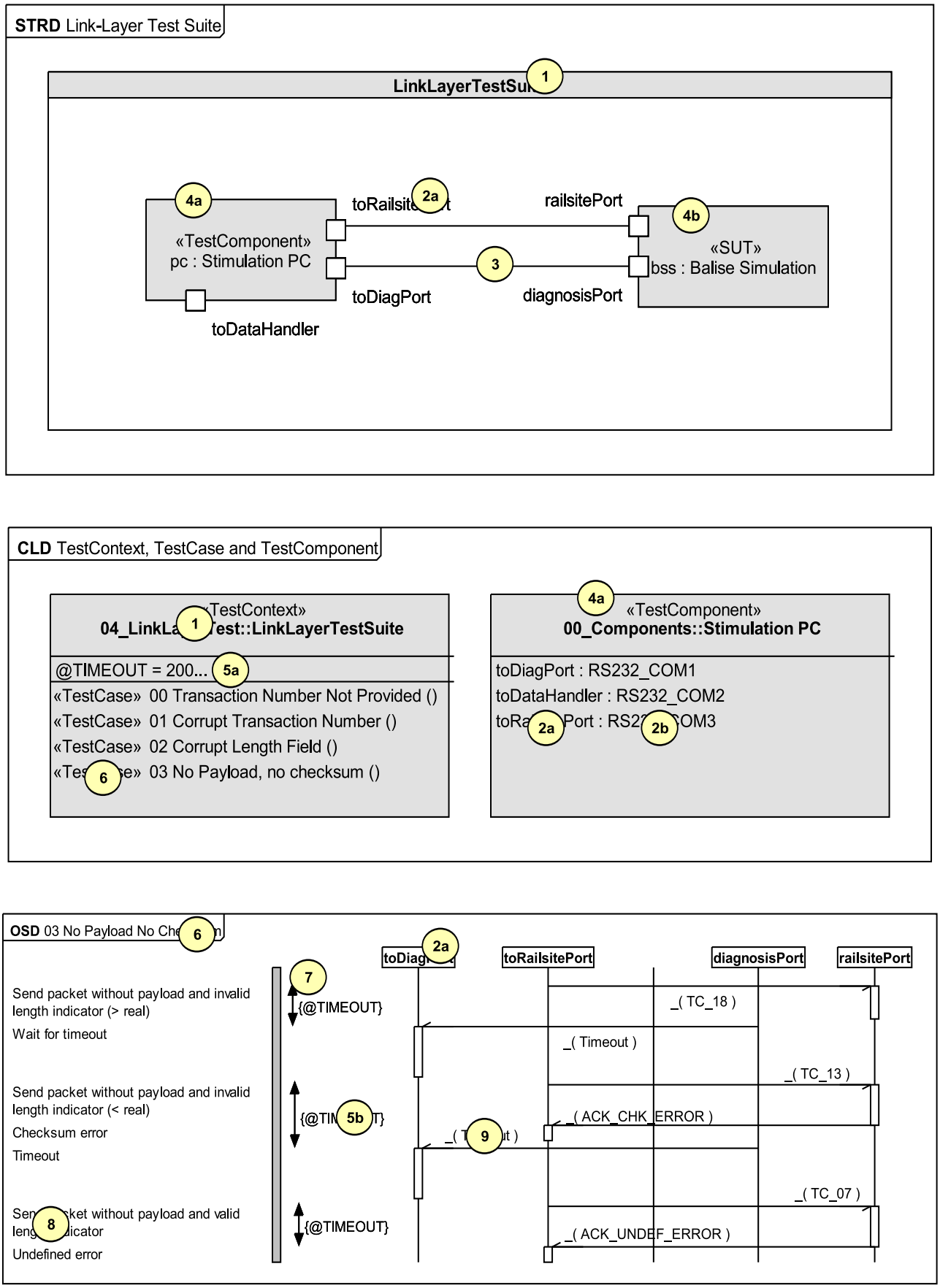

Abbildung 7.8: Auszug aus einem U2TP-Modell zur Erzeugung von TTCN-3-Testskripten

```
module LinkLayerTestSuite {   // Siehe Tab. 7.1, Nr. 1
  // Port definitions

  // Ports for component 'Stimulation PC' (vgl. Tab. 7.1, Nr. 2)
  type port RS232_COM2 message {
    out charstring;
    in charstring;
  }

  // [....] Weitere Portdefinitionen aus Platzgründen gekürzt

  // Component definitions

  // component 'Stimulation PC' (vgl. Tab. 7.1, Nr. 4)
  type component Stimulation_PC {
    port RS232_COM2 toDataHandler;
    port RS232_COM1 toDiagPort;
    port RS232_COM3 toRailsitePort;
  }

  // Test case definition

  // Test case for component 'Stimulation PC' (vgl. Tab. 7.1, Nr. 6)
  testcase Stimulation_PC__03_No_Payload_no_checksum() \
  runs on Stimulation_PC system mySystem {

    // Mapping of component ports to test system ports
    map(self:toDataHandler, system:dummyPort00);
    map(self:toDiagPort, system:dummyPort01);
    map(self:toRailsitePort, system:dummyPort02);

    // [....] Vorangegangene Testschritte aus Platzgründen gekürzt

    // Step: 'Send packet without payload and valid length indicator'
    //
    timer t01D14D78;         // vgl. Tab. 7.1, Nr. 7
    t01D14D78.start(200);    // vgl. Tab. 7.1, Nr. 5
    //
    // toRailsitePort ---> railsitePort
    toRailsitePort.send(hexstring:'F4493734420700'H)  // vgl. Tab. 7.1, Nr. 9

    // Step: 'Undefined error'
    //
    //
    // railsitePort ---> toRailsitePort
    alt {
      [] t01D14D78.timeout {
        verdict.set(fail);
      }
      [] toRailsitePort.receive(hexstring:'010203040D0001000018000000'H) {
        verdict.set(pass);
      }
    }

  }

  // Control part
  control {
    var verdicttype v;
    v := execute(Stimulation_PC__03_No_Payload,_no_checksum());
  }
}
```

Abbildung 7.9: Aus dem in Abbildung 7.8 gezeigten UML-Modell erzeugter TTCN-3-Code (Auszug: einige Portdefinitionen, Testschritte und Kommentarzeilen aus Platzgründen gekürzt; Quelltext um Referenzen auf Tabelle 7.1 ergänzt)

In diesem Kapitel wurde gezeigt, wie diese Probleme durch den neuen Ansatz eines homogenen, konsistenten UML-Modells gelöst werden können. Durch Kombination geeigneter UML-Profile erlaubt es die Erfassung von Anforderungen, Systemstruktur, Systemverhalten und Testfällen in einem einzigen Modell:

- Anforderungen werden ähnlich wie Klassen modelliert, was eine detaillierte Strukturierung und Hierarchisierung der Anforderungen ermöglicht.
- Anforderungen können auf andere Anforderungen, Implementierungselemente oder Testfälle referenzieren.
- Die Anforderungsbeziehungen können automatisiert nach verschiedenen Kriterien ausgewertet werden. So können unter anderen Aussagen zur Modellkonsistenz, zum Implementierungsgrad oder zur Testabdeckung abgeleitet werden.
- Das entstandene UML-Modell läßt die Erzeugung von ausführbarem Systemcode zu.
- Das UML-Modell erlaubt die direkte Erzeugung von TTCN-3-Skripten für die automatische Durchführung von Testfällen.

Das folgende Kapitel wird die Anwendung der auf den letzten Seiten eingeführten Konzepte im Rahmen einer Fallstudie zeigen. Dies betrifft zu aller erst den Aufbau einer geeigneten Toolkette, mit deren Hilfe dann die Kombination von SysML- und U2TP-Modellteilen, die Code- und Testfallerzeugung und schließlich das Anforderungsmanagement exemplarisch vorgeführt werden.

8 Fallstudie

Im vorangegangenen Kapitel wurde ein Verfahren zur Kopplung von System- und Testmodellen auf Basis von UML erläutert. In den folgenden Abschnitten soll diese Methode einer praktischen Anwendung unterzogen werden.

Dazu werden zunächst die verwendeten Werkzeuge zur Umsetzung des im letzten Kapitel beschriebenen Verfahrens vorgestellt. Dabei werden auch die Gründe dargelegt, die zur Wahl des jeweiligen Werkzeugs geführt haben. Anschließend wird ausführlich auf den wichtigen Punkt der Werkzeugkopplung eingegangen, weil erst dadurch eine einsatzfähige Toolkette entsteht.

Diese Kette wird dann eingesetzt, um das Beispielsystem zu modellieren. Zunächst wird das zu modellierende System überblicksartig beschrieben und dann das entstandene Modell eingehend erläutert. Dabei werden auch die Ergebnisse der Codeerzeugung, Testfallerzeugung und Testskripterzeugung vorgestellt. Den Abschluss des Kapitels bildet eine Bewertung des Verfahrens.

8.1 Die Werkzeugkopplung

Die auf den letzten Seiten erläuterte methodische Verknüpfung und Auswertung der UML-Modelle muss für den praktischen Einsatz natürlich durch entsprechende Werkzeuge unterstützt werden (BMW-Prinzip, S. 49).

Im Einzelnen kamen dazu für diese Fallstudie folgende Programme zum Einsatz:

Artisan Studio: UML-CASE-Tool

Texas Instruments Code Composer Studio (CCS): Entwicklungsumgebung für den eingesetzten digitalen Signalprozessor

Classification Tree Editor (CTE): Programm zur Erstellung von Klassifikationsbäumen mit anschließender Testfallerzeugung

TTCN-3-Testumgebung: DLR-Entwicklung, entstanden im Rahmen der Arbeit von Janke und genau auf die Bedürfnisse dieser Fallstudie zugeschnitten [Jan07]

Python: objekt-orientierte OpenSource-Skriptsprache[1]

Die Entscheidung zugunsten der Verwendung von Artisan Studio wurde von mehreren Faktoren beeinflusst. Zum einen ist das Programm vollständig konform zur UML 2.0 und unterstützt alle relevanten Diagrammarten. Zum anderen erlaubt es eine intensive modellinterne Verknüpfung von Elementen auf Metaebene, die wesentlich zur Konsistenz des Gesamtmodells beiträgt. Darüber hinaus verfügt Artisan Studio über einen sehr flexiblen Codegenerator, der auch Zustandsdiagramme übersetzen kann. Am wichtigsten ist aber der direkte Zugriff auf das Modell über eine wohldefinierte Schnittstelle (*OLE Automation Interface*, OLE: Object Linking and Embedding), den Artisan Studio bietet. Erst diese Schnittstelle ermöglichte eine wirkungsvolle Analyse der Anforderungsbeziehungen und einen Export der U2TP-Testfälle als TTCN-3-Code (s. auch Abschnitte 7.2.3 und 7.4).

[1] `http://www.python.org`

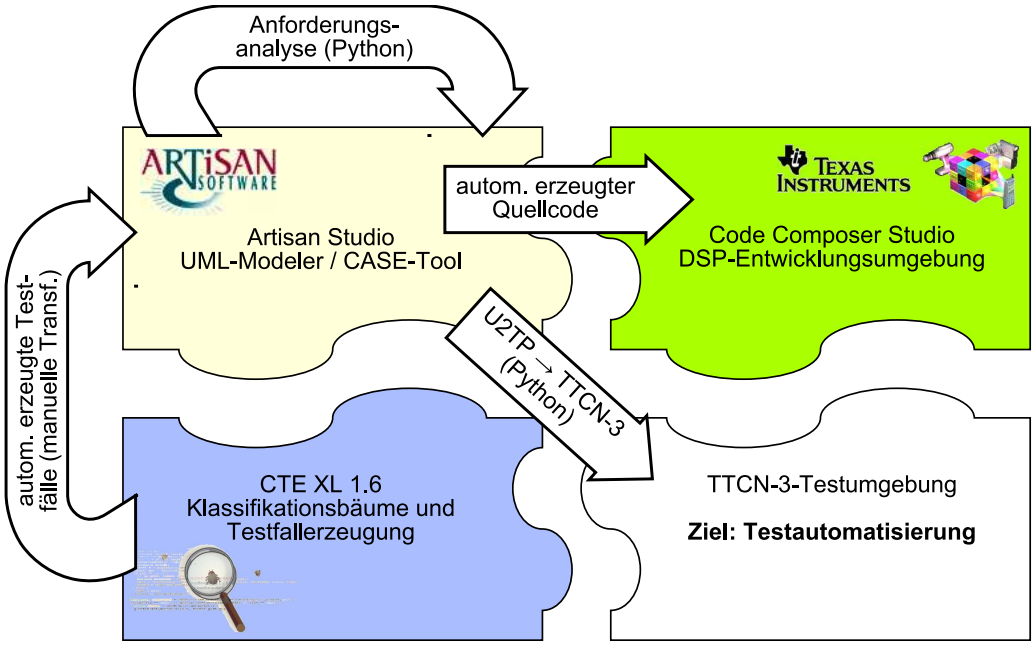

Abbildung 8.1: Zusammenspiel und ausgetauschte Informationen der an der Fallstudie beteiligten Programme

Die Verwendung des CCS ergab sich direkt aus der Entscheidung für den DSP von Texas Instruments, der bestimmte technische Anforderungen zugrunde lagen (Rechenleistung, Anzahl und Art der Hardwareschnittstellen, Übertragungsgeschwindigkeiten an den Schnittstellen etc.).

Das Programm CTE ist ein kostenlos verfügbares Werkzeug, das von den maßgeblichen Entwicklern der Klassifikationsbaummethode erstellt wurde. Es ist eines der Aktuellsten unter den verfügbaren Tools und bietet durch die automatische Erzeugung von Testfällen und der Berücksichtigung logischer Randbedingungen vielseitige Fähigkeiten, die für den hier vorgesehenen Anwendungsfall voll ausreichend sind.

Die Skriptsprache Python hat eine Reihe von Eigenschaften, die sie für zahlreiche Einsatzzwecke qualifiziert:

- vollständige Objektorientierung
- einfach zu erlernen, sehr gut lesbarer Quellcode
- umfangreiche Bibliotheken bereits in der Grundinstallation (z. B. XML-Parser, Netzwerkzugriff, HTTP-Server, Prozessmanagement und reguläre Ausdrücke); mächtige Datei- und Textbearbeitungsmöglichkeiten schon im Sprachkern enthalten
- Plattformunabhängigkeit
- gute Erweiterbarkeit durch C/C++
- umfangreiche Dokumentation, hervorragende Unterstützung durch die OpenSource-Community

Mit Hilfe eines Erweiterungspakets ist Python in der Lage, sowohl als Client wie als Server das OLE Automation Interface zu bedienen. Dadurch wird der Zugriff auf das Artisan Studio Metamodell entschieden vereinfacht, da so für eine einzelne Anfrage an das UML-Modell nur ein Python-Befehl statt ca. 10 C-Befehlen notwendig ist.

8.1.1 Das Zusammenspiel der Programme

Die Abbildung 8.1 zeigt skizzenhaft, wie die genannten Programme miteinander interagieren. Der wichtigste Punkt ist Artisan Studio, in dem die Anforderungen, das Systemmodell und die Tests zentral hinterlegt sind.

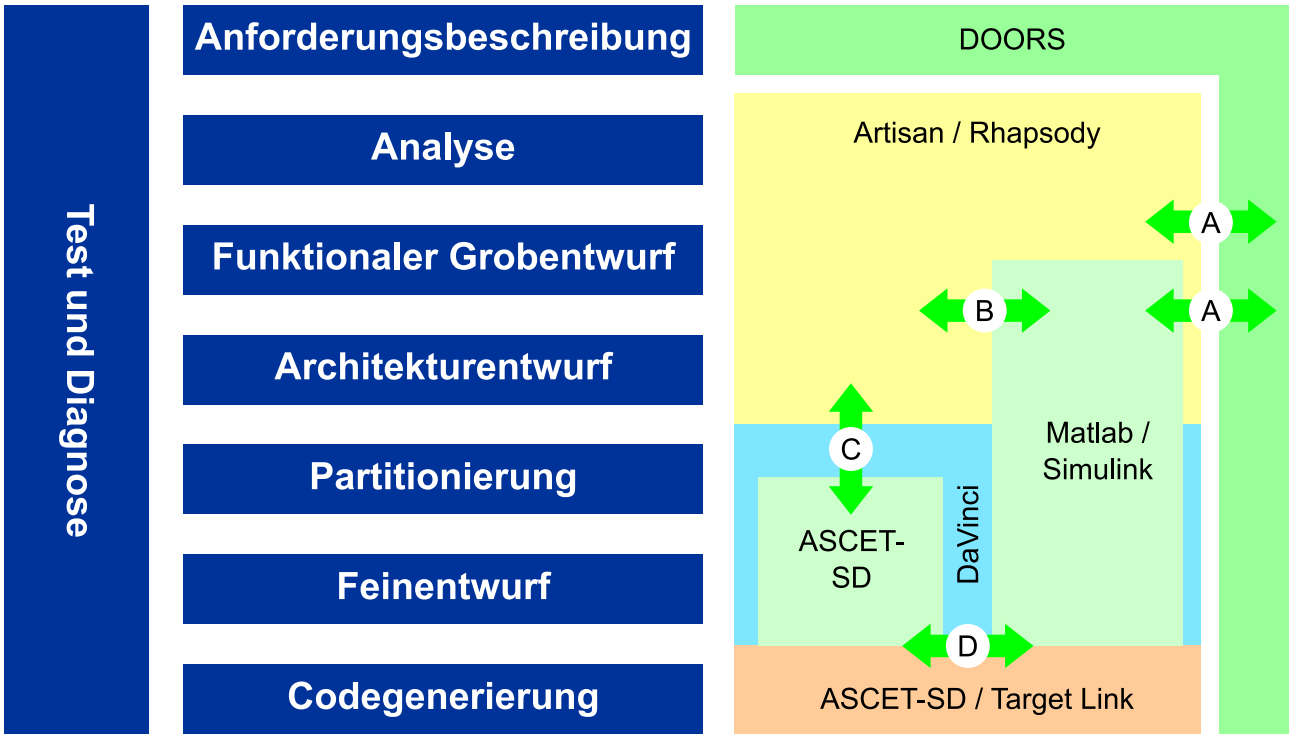

Abbildung 8.2: Werkzeugkopplung im STEP-X-Projekt (nach [HH03b]); Erläuterung der einzelnen Werkzeugbeziehungen: A = Modellverlinkung, B = Kosimulation, C = Modelltransformation, D = Codeintegration.

Ein Gutteil dieser Tests wurde durch die in CTE hinterlegten Klassifikationsbäume automatisch erzeugt und dann manuell in das UML-Modell übernommen.

Mittels des bereits in Abschnitt 7.2.3 beschriebenen Python-Skriptes kann anschließend das gesamte UML-Modell auf Konsistenz hinsichtlich der Umsetzung und Nachverfolgbarkeit der Anforderungen überprüft werden.

Das UML-Modell ist auch Basis für die Ableitung weiterer Informationen. Zum einen wird mit dem in Artisan Studio enthaltenen Codegenerator C++-Code erzeugt, der dann im CCS kompiliert und an den Zielrechner übertragen wird. Zum anderen liest das in Abschnitt 7.4 erläuterte Programm die als U2TP-Modelle hinterlegten Testfälle aus und konvertiert sie in TTCN-3-Code, der von der TTCN-3-Testumgebung ausgeführt werden kann.

Insgesamt weist die hier vorgestellte Werkzeugkopplung weniger Elemente als bei vergleichbaren modellbasierten Entwicklungsprozessen auf. Der Hauptgrund ist die einheitliche Speicherung aller relevanten Informationen in einem einzigen UML-Modell, was Transformationen von Modellinhalten in andere Beschreibungsmittel bzw. andere Werkzeuge erspart. Als Vergleich ist in Abbildung 8.2 die Werkzeugkopplung aus dem in der Einleitung erwähnten STEP-X-Projekt (s. Abschnitt 1.3 und 1.4) aufgeführt. Sie enthält erheblich mehr Schnittstellen und Referenzierungen zwischen verschiedenen Tools. Ursache dafür ist beispielsweise die Umsetzung des Anforderungsmanagements in einem separaten Programm (DOORS) anstelle der direkten Verwendung des UML-Modells wie in dieser Arbeit. Ebenso erzeugt der Übergang von UML auf ASCET weiteren Schnittstellenaufwand, der in dm hier vorgestellten Lösungsansatz vermieden wird, weil der Zielcode direkt aus dem UML-Modell erzeugt wird.

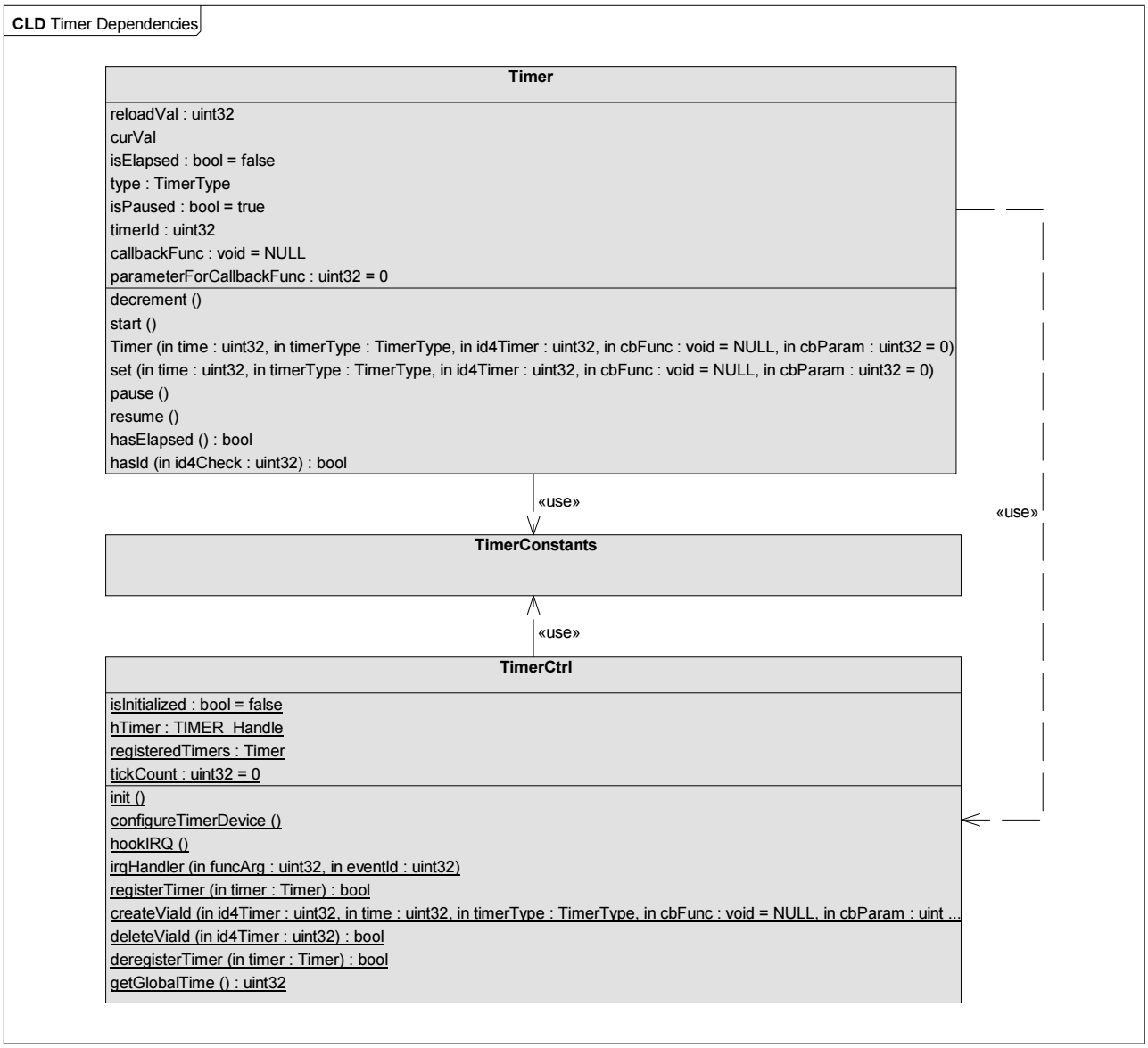

Abbildung 8.3: Diagramm der zur Kopplung von Artisan Studio und CCS notwendigen Timerklassen

8.1.2 Anpassung zwischen Artisan Studio und CCS

Für eine funktionierende Übernahme des vom CASE-Tool erzeugten Codes in das CCS bedarf es einiger Anpassungen, die aber größtenteils im Modell selber durch zusätzliche Klassen vorgenommen werden. Die Bearbeitung externer Ressourcen ist nur an einer Stelle notwendig.

Die Anpassungen beziehen sich auf die korrekte Ausführung von Zustandsmaschinen, für die der von Artisan Studio erzeugte Code zum einen auf Mutex-Funktionen und zum anderen auf Timer zugreifen muss. Mutexes werden gebraucht, um den konkurrierenden Zugriff auf eine Zustandsmaschine zu verhindern (beispielsweise das Auslösen von Transitionen während vorhergehende Übergänge noch abgearbeitet werden), während Timer mit Hilfe von Callback-Funktionen zeitgesteuerte Zustandsübergänge realisieren.

Für viele Zielplattformen – wie beispielsweise Windows – stehen derartige Funktionen bereits in einer Bibliothek des Betriebssystems oder der Laufzeitumgebung zur Verfügung. Das trifft auf den eingesetzten DSP jedoch nicht zu. Daher müssen die benötigten Routinen manuell ergänzt werden.

Prinzipiell stehen dafür zwei Vorgehensweisen zur Verfügung: entweder die Implementierung der Funktio-

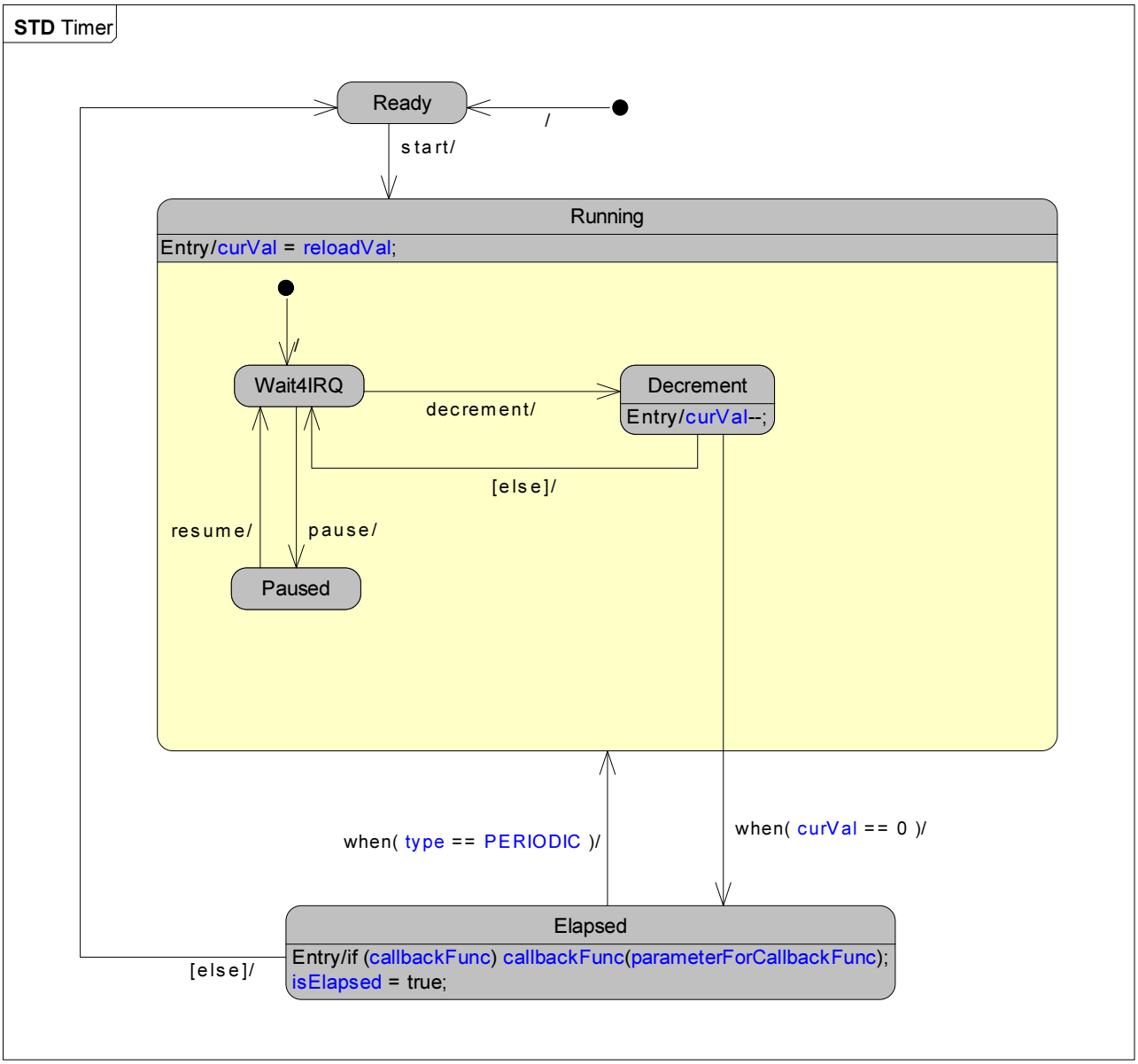

Abbildung 8.4: Zustandsdiagramm der zum logischen Timer gehörenden Klasse

nen in externen Bibliotheken außerhalb des Modells oder die Aufnahme der Funktionen in das Modell und damit die Erzeugung des benötigten Codes durch den Codegenerator selbst. Für die vorliegende Fallstudie wurde die zweite Variante gewählt, um möglichst wenig Abhängigkeiten des implizit im Modell vorhandenen Codes von externen Bibliotheken zu erreichen.

Für eine effektive Ausnutzung der physikalisch auf dem DSP vorhandenen Timer wurde die Möglichkeit geschaffen, mehrere logische Timer in Artisan Studio auf einen einzigen physikalischen Timer im DSP abzubilden. Entsprechend entstand eine Klasse für die Steuerung des physikalischen Timers und die Verwaltung der logischen Timer (`TimerCtrl` in Abb. 8.3) sowie eine Klasse für die logischen Timer (`Timer`). Der Code von `TimerCtrl` wurde aufgrund seiner großen Hardwarenähe (Interruptprogrammierung) direkt im Modell hinterlegt. Das Verhalten von `Timer` wurde dagegen überwiegend als Zustandsdiagramm erfasst, dass in Abbildung 8.4 wiedergegeben ist. Der zu `Timer` gehörige Code wurde somit zusammen mit dem restlichen Programm aus dem Modell erzeugt.

Für die Mutexes wurde ebenfalls eine Klasse in Modell hinterlegt, die jedoch wesentlich einfacher gehalten

ist als die Timer. Sie besteht lediglich aus einem einzigem Attribut, das anzeigt, ob der Mutex gesperrt ist oder nicht. Durch zwei entsprechende Methoden (`acquire` und `release`) kann auf den Mutex zugegriffen werden. Vor dem Sperren des Mutex müssen allerdings alle Interrupts deaktiviert werden, um konkurrierenden Zugriff auf das Attribut zu vermeiden. Aufgrund dieser hardwarenahen Implementierung wurde auch für diese Klasse der Code direkt im Modell hinterlegt und nicht aus Zustandsdiagrammen generiert[2].

Die Verwendung von Timern und Mutexes durch den von Artisan Studio erzeugten Code erfolgt mittels generischer Namen und Parameter. Durch Anpassung einer Headerdatei können diese generischen Namen als Makros auf die tatsächlich implementierten Funktionen (hier die entsprechenden Klassen bzw. ihre Instanzen aus dem Modell) abgebildet werden. Durch diesen sehr einfachen und effektiven Mechanismus war die Anpassung von Artisan Studio an die Entwicklungsumgebung CCS mit wenigen Handgriffen möglich.

8.1.3 Interaktion der Pythonskripte mit dem UML-Modell

Die in Artisan Studio enthaltenen Modelle können auf drei verschiedenen Wegen zugänglich gemacht werden:

1. Export der Modelldaten in einem proprietären Dateiformat, das zum Austausch mit anderen Nutzern von Artisan Studio vorgesehen ist

2. Export als XMI-Datei [Obj05a]

3. direkter Zugriff auf das Metamodell über eine Softwareschnittstelle

Für den Zugriff mit einem externen Skript scheidet die erste Variante aus, da das Dateiformat nicht frei zugänglich ist. Die zweite Variante, XMI, wurde in ersten Versionen des Skriptes zur Analyse der Anforderungsbeziehungen verwendet. Leider zeigten sich schnell Unzulänglichkeiten, wie beispielsweise der nicht vorhandene Zugriff auf Kommentarfelder (für Callouts) und auf Stereotypen. Daher kam nur die dritte Möglichkeit in Betracht, die aus Softwaresicht nicht nur eine sehr elegante Lösung darstellt, sondern auch Vollzugriff auf *alle* Modellelemente und sogar eine Fernsteuerung von Artisan Studio ermöglicht.

Die Schnittstelle basiert auf COM (Component Object Model), das wiederum auf OLE aufbaut. Über COM ist es möglich, Softwareobjekte verschiedener Klassen über eine einheitliche Schnittstelle anzusprechen. Die Interaktion mit den Objekten erfolgt dynamisch und auch der genaue Aufbau der Schnittstelle muss erst zur Laufzeit feststehen. COM erlaubt so den sprachunabhängigen und hoch dynamischen Zugriff auf Objekte anderer Applikationen. Detaillierte Informationen zu COM können einschlägigen Fachbüchern entnommen werden (beispielsweise [Box97]).

Im Falle von Artisan Studio bietet das CASE-Tool die Objekte an, es agiert als COM-Server[3]. Die verschiedenen Pythonskripte verbinden sich mit dem Server und greifen auf die angebotenen Objekte zu.

Jedes Objekt entspricht – grob gesagt – einem Element im UML-Modell und verfügt über Attribute (Name, Typ, ID) und Verknüpfungen zu anderen Objekten (zum zugehörigen UML-Paket, zur Elternklasse und viele mehr). Dadurch ergibt sich ein vielfach vermaschtes Netz aus Objekten, über das, ausgehend von einem Wurzelobjekt, durch das gesamte Modell navigiert werden kann.

Über Python ist es auf sehr einfache Weise möglich, auf diese Objekte zuzugreifen. Das COM-Objekt wird durch ein Objekt in Python repräsentiert, über das auf die Attribute und Verknüpfungen des COM-Objektes direkt zugegriffen werden kann.

[2] Die Modellierung als Zustandsdiagramm wäre zwar elegant und übersichtlich (nur zwei Zustände), schließt sich jedoch logisch aus: Mutexes werden für die Implementierung von Zustandsmaschinen gebraucht, daher können Mutexes nicht in Zustandsmaschinen implementiert werden...

[3] genauer gesagt ist es nicht der Modeler als solches, sondern die Komponente *Enabler*, eine Art Datenbank, in der das Modell abgelegt ist und von mehreren Benutzern parallel verwendet werden kann. Auch der Modeler greift nur als Client auf diese Datenbank zurück

Abbildung 8.5 zeigt dazu exemplarisch den Zugriff auf Verknüpfungen über die `item`-Methode (Zeilen 5 und 6) und die Abfrage von Attributen wie `Class Name` und `Full Name` (Zeile 9). Die für einen bestimmten Objekttyp verfügbaren Attribute und Verknüpfungen sind in [Art06] detailliert beschrieben.

```
1  from win32com.client import Dispatch
2  from sys import argv
3
4  allProjects = Dispatch("OMTE.Projects")
5  model = allProjects.Item("Project", argv[1])
6  dictionary = model.item("Dictionary")
7
8  for element in dictionary:
9    print "%s (%s)" % (element("Full Name"), element("Class Name"))
10
```

Abbildung 8.5: Einfaches Pythonskript zur Ausgabe aller Elemente mit Namen und Typ, die sich in einem UML-Modell befinden

Um die Pythonapplikation für die eigentliche Modellanalyse von der COM-Ebene zu entkoppeln, wurde eine Abstraktionsschicht eingezogen, die alle COM-Objekte auf eine zugehörige Handlerklasse abbildet. Einen Ausschnitt aus der resultierenden Klassenhierarchie zeigt Abbildung 8.6. Das Basisobjekt (`baseOleHandler`) ist prinzipiell auf alle COM-Objekte anwendbar. Für spezielle Typen existieren jedoch spezielle Handlerklassen, um auf die individuellen Eigenschaften eines solchen Typs (beispielsweise eine Klasse oder einer Abhängigkeit) einfach zugreifen zu können. Alle Objekte werden über die Funktion `convCOM2Py` instantiiert, die je nach Typ des COM-Objektes entweder eine Instanz von `baseOleHandler` oder von einer abgeleiteten Spezialklasse erzeugt.

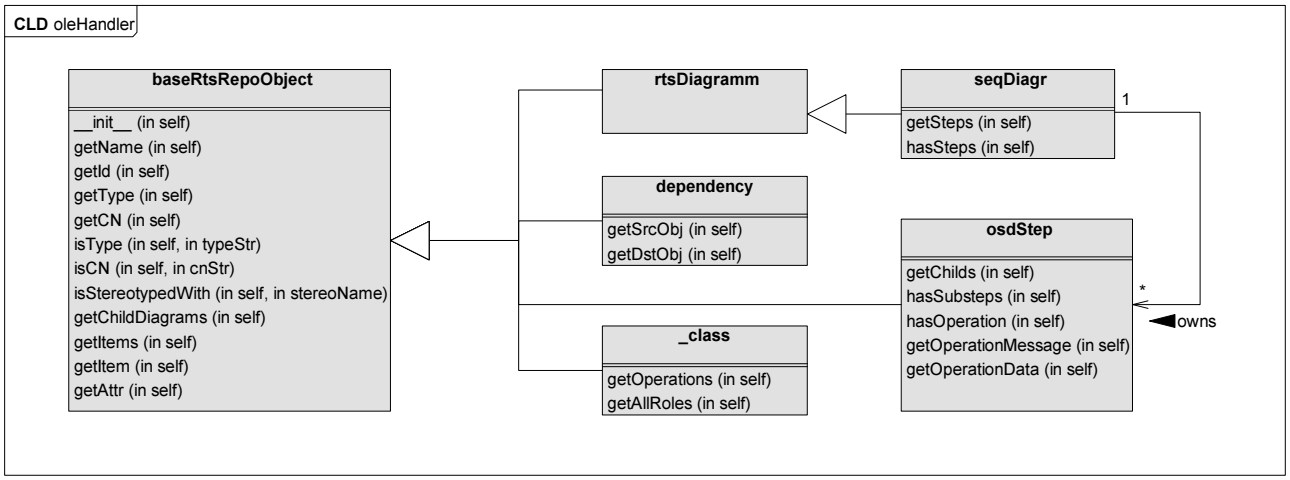

Abbildung 8.6: Ausschnitt aus der Klassenhierarchie zur Kapselung der COM-Objekte in speziellen Handlerklassen

Basierend auf dieser Abstraktionsebene konnten dann die eigentlichen Applikationen für die Konvertierung der U2TP-Testbeschreibungen in TTCN-3-Code und für die Analyse der Anforderungsbeziehungen entwickelt werden. Selbstverständlich lässt sich diese Zwischenebene auch für andere Projekte außerhalb von SysML und U2TP einsetzen. Das „Wissen“ über die beiden UML-Profile ist erst in den höheren Softwareebenen implementiert.

8.1.4 Skripte zur TTCN-3-Erzeugung und zum Anforderungsmanagement

Die zuvor erwähnte höhere Ebene wird vom sogenannten `profileHandler` gebildet, der unmittelbar auf dem `oleHandler` aufbaut und ebenso wie dieser aus einer generischen Basisklasse und davon abgeleiteten Spezialklassen besteht. Diese Spezialklassen kapseln die Eigenschaften verschiedener SysML- oder U2TP-

Klasse	Eigenschaften und Aufgaben
`profileHandler`	Initialisierung des Modellzugriff durch Erzeugung einer `oleHandler`-Instanz, zentrale Instantiierung neuer Profilelemente (s. folgende Klassen), Pflege einer globalen Liste instantiierter Elemente
`profileElementBaseClass`	Basisklasse aller Profilelemente. Ermöglicht Zugriff auf allgemeine Eigenschaften wie beispielsweise den Objektnamen
`TestContext`	Zugriff auf Komponenten, Testfälle und Parameter eines Testkontextes
`Requirement`	Zugriff auf untergeordnete Anforderungen, Analyse der `≪Satisfy≫`- und `≪Verify≫`-Beziehungen
`Port`	Zugriff auf den Porttyp
`TestComponent`	Zugriff auf alle Ports der Testkomponente
`TestCase`	Zugriff auf zugehörige Testschritte und auf zugehörigen Testkontext, auf SysML-Verknüpfungen zu Anforderungen, auf alle im Testfall verwendeten Ports, Komponenten und SUTs, Konsistenzcheck für TTCN-3-Export
`TestVector`	Zugriff und Konvertierung von Stimuli aus Datenpools, insbesondere Übersetzung von Zeichenketten, Binär-, Dezimal- und Hexadezimalwerten in TTCN-3-Hexstrings
`TestStep`	Repräsentiert einen einzelnen Testschritt, erlaubt den den Zugriff auf die Ports der Sender und Empfänger der zugehörigen Nachricht, ermittelt evtl. gesetzte Timer
`Timer`	Repräsentiert einen Timer bzw. einen Timeout, ermöglicht Zugriff auf den zugehörigen Start- und Stoptestschritt des Timers, generiert für jeden Timer einen eindeutigen Variablennamen

Tabelle 8.1: Klassen der `profileHandler`-Ebene zur Interpretation der UML-Modelle unter SysML- und U2TP-Aspekten

Elemente wie beispielsweise Anforderungen oder Testfälle unter Rückgriff auf die zugehörigen Elemente des zugrunde liegenden `oleHandler` (z. B. eine Klasse samt Methoden und Stereotypen). Tabelle 8.1 fasst die verschiedenen Klassen dieser Softwareebene kurz zusammen.

Daneben existiert eine Klasse `ttcn3gen`, die mit einer `TestCase`-Instanz initialisiert wird und diesen Testfall in ein TTCN3-Skript wandelt. Der Zugriff auf das Modell erfolgt dabei über die Objekte des oben erwähnten `profileHandler`.

Der Algorithmus für die TTCN-3-Erzeugung arbeitet wie folgt:

1. Überprüfe die Konsistenz des zu konvertierenden Testfalls (vgl. Abschnitt 7.4) und breche ggf. mit Fehlermeldung ab.

2. Erzeuge einleitende Kommentarzeilen (Datum, Uhrzeit, Modellname, Testkontextname, Testfallname, Prüflingsname, verwendete Testkomponenten)

3. Erzeuge für jeden Port einer Testkomponente einen TTCN-3-Typ, dessen Name auf dem Porttyp im U2TP-Modell beruht (Zeilen 5 – 7 in Abb. 7.9)

4. Erzeuge für jede Komponente einen TTCN-3-Typ, dessen Name sich aus dem Komponentennamen ergibt. Dieser Typ beinhaltet eine Deklaration der Komponentenports mit ihren Namen (aus dem UML-Modell) und ihrem im Schritt zuvor definierten Typ (Zeilen 15 – 18 in Abb. 7.9).

5. Eröffne einen Testfallabschnitt und beginne mit einer Zuordnung der „internen“ Komponentenports zu den „externen“ Testsystemschnittstellen (wird hier nicht verwendet, ist aber für eine korrekte Portverwendung notwendig. Daher Zuordnung zu „Dummyports“).

6. Erzeuge eine Liste aller Testschritte. Iteriere über diese Liste und erzeuge für jeden Schritt einen `send`-, `receive`- oder Timeraufruf. Wenn nötig, kombiniere mehrere dieser Aufrufe in einem `alt`-Konstrukt.
7. Schließe den Testfallabschnitt.
8. Erzeuge den „Control"-Abschnitt, der aus der Ausführung des zuvor definierten Testfalls besteht.

Eine der wichtigsten Aktivitäten ist Schritt 6 („Bestimme Liste aller Testschritte"). Hier muss nicht nur die mögliche Aneinanderreihung mehrerer Sequenzdiagramme für einen einzelnen Testfall berücksichtigt werden, sondern auch eine korrekte Verarbeitung sogenannter „Interaction Frames". Diese speziellen Schritte können Referenzen auf andere Sequenzdiagramme enthalten, deren Schritte an der jeweiligen Stelle einzufügen sind. Abbildung 8.7 zeigt ein Beispiel eines solchen Diagramms. Es besteht aus sechs referenzierten Diagrammen.

Ein besonderer Vorteil der Interaction Frames: mit ihnen lassen sich Bibliotheken häufig verwendeter Testschritte aufbauen, wobei die Wiederverwendung der Bausteine dank Drag & Drop denkbar einfach ist. Modellintern wird auf die Bibliothekselemente nur referenziert (es wird *keine* Kopie angelegt), so dass bei Änderung in der Testschrittbibliothek die auf sie verweisenden Diagramme nicht zusätzlich angepasst werden müssen.

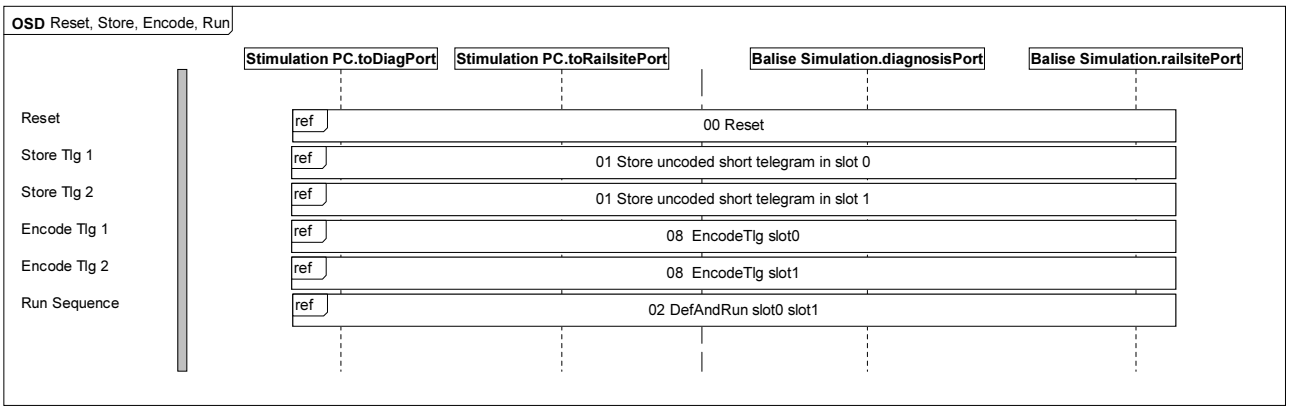

Abbildung 8.7: Sequenzdiagramm eines Testablauf mit Referenzen auf andere Sequenzdiagramme

Bei der Erstellung der TTCN-3-Testfallschrittliste ist also dieser Referenzierungsmechanismus auf andere Diagramme zu beachten. Wichtig ist, dass diese eingebundenen Diagramme ihrerseits wieder Referenzen enthalten können usw. Daher ist der gesamte Algorithmus zur Testschrittbestimmung rekursiv auszulegen. Darüber hinaus sind noch zwei Kriterien zu überprüfen:

- das referenzierte Diagramm muss ein Sequenzdiagramm sein, das zu einem `≪TestCase≫` gehört, der seinerseits einem `≪TestContext≫` zugeordnet ist.
- zur Vermeidung zirkulärer Abhängigkeiten darf kein Diagramm referenziert werden, das bereits zu einem früheren Zeitpunkt der Testschrittbestimmung besucht wurde. Bei Verletzung dieses Kriteriums würde die Testschrittbestimmung nicht terminieren.

Auf TTCN-3-Ebene löst der TTCN-3-Generator die Referenzierung von Diagrammen durch direktes Einkopieren der zugehörigen Schritte in den Testablauf auf. Die TTCN-3-Datei enthält daher keine Aufrufe anderer Module, sondern besteht nur aus einem einzigen sequentiellen Ablauf aller Schritte.

Grafische Benutzeroberfläche

Für die benutzerfreundliche Interaktion mit den vom `profileHandler` und vom TTCN-3-Generator zur Verfügung gestellten Funktionen wurde eine grafische Benutzerschnittstelle geschaffen, die in Abbildung 8.8

gezeigt ist. Mit Hilfe dieser Oberfläche lassen sich alle oder nur Teile der gültigen Testfälle eines Modells in TTCN-3-Dateien umwandeln und speichern. Darüber hinaus wird zu ungültigen Testfällen eine Fehlermeldung ausgegeben.

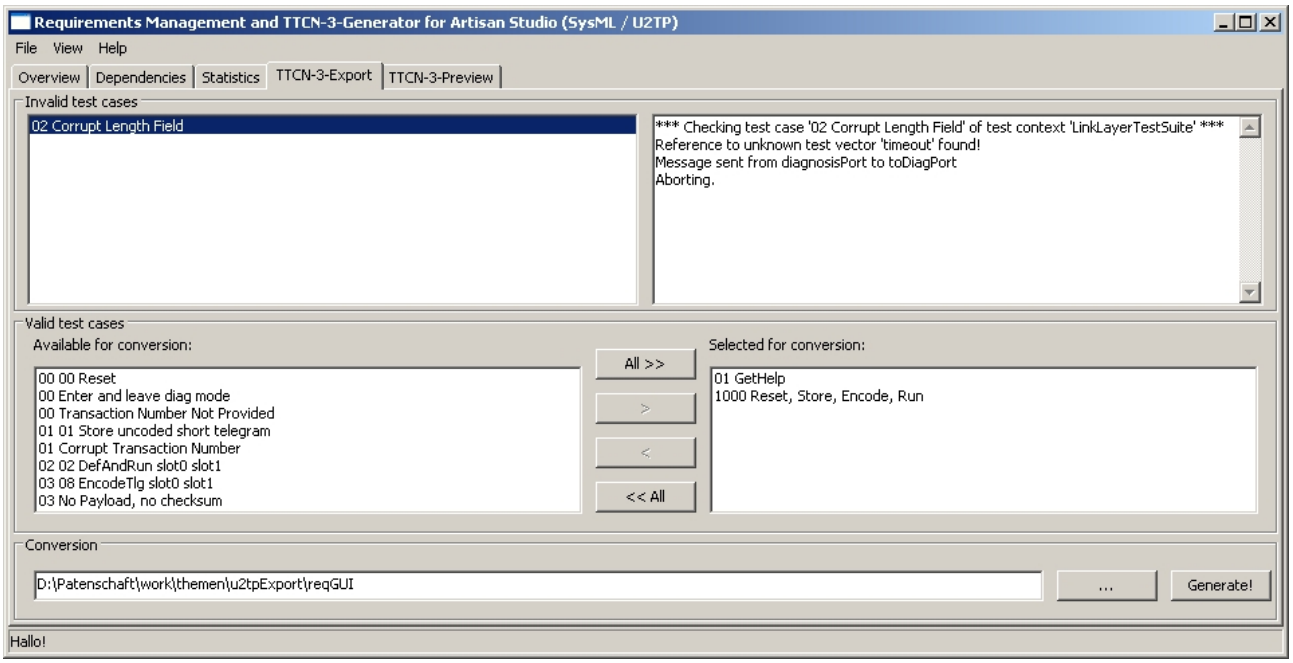

Abbildung 8.8: Grafische Oberfläche zur Erzeugung von TTCN-3-Skripten aus den Testfällen eines UML-Modells

In gleicher Weise kann mit Hilfe des gezeigten Programms auf die Anforderungen in einem UML-Modell zugegriffen werden. Abbildung 8.9 zeigt, wie die `≪Satisfy≫`- und `≪Verify≫`-Beziehungen zwischen Anforderungen, Implementierung und Testfällen ausgewertet und dargestellt werden. Die Ansicht kann so verändert werden, dass nur Anforderungen gezeigt werden, die keinen Testfall bzw. keiner Implementierung zugeordnet sind. Auf diese Weise kann sehr einfach die Konsistenz und Vollständigkeit des Modells überprüft werden. Der untere Teil der Abbildung zeigt darüber hinaus eine Übersicht aller Testkontexte und der jeweils zugehörigen Testfälle.

Für eine möglichst einfache Integration mit den zuvor beschriebenen Handlerklassen wurde auch die grafische Benutzeroberfläche in Python geschrieben. Dazu wurde auf „WxWidgets“[4] zurückgegriffen. Dabei handelt es sich um eine C++-Bibliothek zur Erstellung plattformunabhängiger grafischer Applikationen, die über sogenannte „Bindings“ allerdings auch von anderen Sprachen wie beispielsweise Python genutzt werden kann.

Mit Hilfe von Python und der auf WxWidgets basierenden grafischen Benutzeroberfläche wird die Werkzeugkopplung vervollständigt. Durch die beschriebenen Anpassungen, Skripte und Verfahren ist es möglich, die zentral in einem UML-Modell abgelegten Testfall- und Systembeschreibungen in C-Code und TTCN-3-Skripte zu überführen und zwischen allen Modellteilen ein nachverfolgbares und analysierbares Anforderungsmanagement aufzubauen. Die folgenden Abschnitte zeigen nun die Anwendung dieser Werkzeugkette auf ein Beispielsystem aus dem DLR-Bahnsimulationslabor RailSiTe.

[4] www.wxwidgets.org

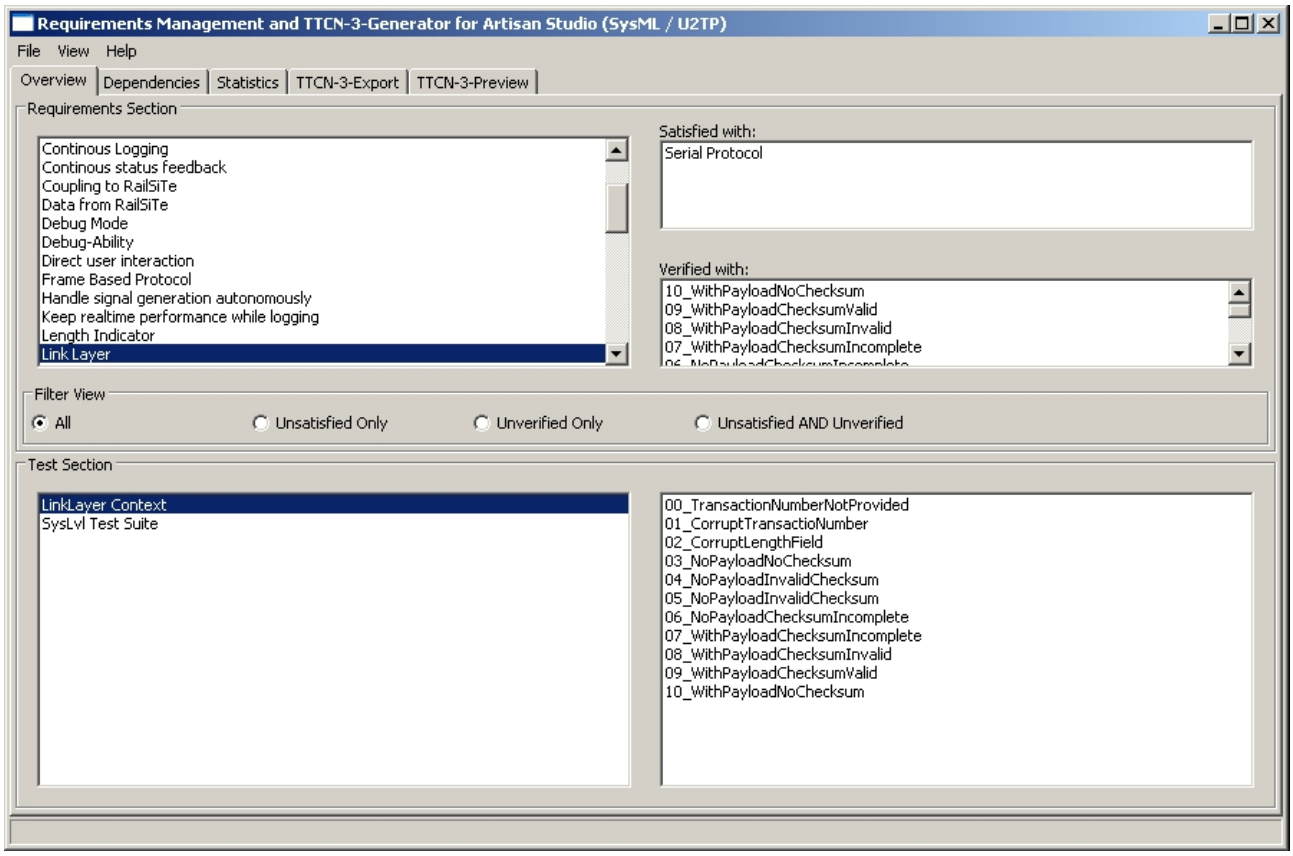

Abbildung 8.9: Grafische Oberfläche zur Analyse der Anforderungsbeziehungen innerhalb eines UML-Modells

8.2 Kurzbeschreibung des modellierten Systems

Das untersuchte System besteht aus einer Anlage zur Erzeugung von Transpondersignalen, die Teil des in Abschnitt 6.2 beschriebenen Bahnlabors RailSiTe ist. Durch diese Laborkomponente kann einem realen Fahrzeugrechner im Rahmen eines HiL-Tests die Vorbeifahrt an einer sogenannten Balise (passiver Transponder) vorgetäuscht werden.

Dies geschieht, indem die in der Realität von der Balise abgestrahlten Hochfrequenzsignale durch das Simulationssystem erzeugt und über eine Antenne ausgesendet werden (Abb. 8.10). Bei der Signalerzeugung müssen die zu übertragenden Daten und das von der (virtuellen) Zuggeschwindigkeit abhängige zeitliche Profil der Übertragung berücksichtigt werden. Die Balisensimulation ist dazu über eine RS232-Verbindung mit dem Laborkern verbunden und bezieht von dort alle notwendigen Daten. Die Kommunikation erfolgt über ein verbindungsloses, auf einzelnen Nachrichtentelegrammen basierendes Protokoll [Wil06].

Die Echtzeiterzeugung des von der Antenne zu sendenden Datenstroms erfolgt autark durch das Balisensimulationssystem. Dafür ist es, wie in Abbildung 8.10 dargestellt, mit einem digitalen Signalprozessor (DSP) ausgestattet, der die zu sendenden Daten aufbereitet und unter Berücksichtigung des korrekten zeitlichen Profils (Sendedauer eines Telegramms, Pausendauer zwischen zwei Telegrammen) an einen Modulator übergibt. Dieser wandelt die digitalen Daten mittels Frequenzumtastung in ein Analogsignal, das dann über eine Antenne abgestrahlt wird.

Für diese Balisensimulation wurden eine Reihe von Anforderungen, Implementierungsklassen sowie Integrations- und Systemtests mit Hilfe der UML erfasst, wobei hier vor allem auf das Kommunikationsprotokoll eingegangen werden soll. Das Protokoll umfasst zwei Ebenen:

Sicherungsschicht: Diese Ebene stellt durch Längeninformationen, Sequenznummern und eine Prüfsumme

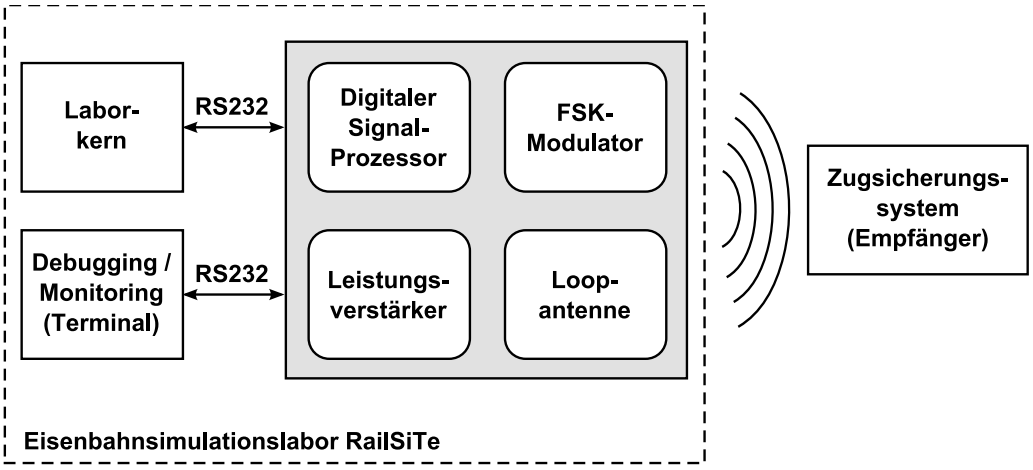

Abbildung 8.10: Vereinfachtes Blockdiagramm des in der Fallstudie untersuchten Systems (grau hinterlegt)

den korrekten und vollständigen Empfang von Datenpaketen sicher.

Applikationsschicht: Die Applikationsdaten, also die Nutzdaten der Sicherungsschicht, stellen die eigentlichen PDUs (Protocol Data Units) dar, über die das RailSiTe und die Balisensimulation miteinander kommunizieren. Über sie werden beispielsweise Timing- oder Balisentelegrammdaten vom Labor an die Balisensimulation übertragen.

Die Modellierung der Balisensimulation erfolgte zu einem Zeitpunkt, als bereits eine herkömmlich entwickelte Software auf dem System im Einsatz war. Eine detaillierte Beschreibung der Hard- und Softwarekomponenten sowie des Kommunikationsprotokolls kann der Arbeit von Willms entnommen werden [Wil06].

8.3 Das Modell

Wie jedes andere UML-Modell ist auch die Beschreibung der Balisensimulation hierarchisch in Pakete strukturiert. In der obersten Hierarchieebene befinden sich die folgenden Pakete:

Initial Requirements: die initialen Anforderungen, die überwiegend funktionaler Natur sind, aber darüber hinaus auch nicht-funktionale Aspekte wie beispielsweise RAMS(S)-Anforderungen enthalten können. Dieses Paket ist weitestgehend implementierungsneutral und somit am ehesten mit einem Lastenheft vergleichbar.

SRS: die aus den Inhalten des vorgenannten Pakets abgeleitete Systemspezifikation. Dieses Paket enthält konkrete Lösungsansätze für das System und entspricht somit in etwa einem Pflichtenheft.

Systemarchitektur: die Systemarchitektur beschreibt die wichtigsten Systemkomponenten und Schnittstellen sowie die an diesen Schnittstellen auszutauschenden Daten. Die Architekturbeschreibung ist aber noch unabhängig von der später verwendeten (Klassen-)Struktur der Software. Ebenso wird die Verteilung der Funktionen auf die verschiedenen Hardwarekomponenten nicht in allen Details berücksichtigt.

Systemdesign: hier wird die tatsächliche Implementierung des Systems beschrieben. Dies umfasst vor allem die Implementierung der DSP-Software.

Tests: enthält U2TP-Testfallspezifikationen für verschiedene Testebenen.

SysML-, U2TP- und C++-Profil: diese drei Pakete sind zur Definition der Stereotypen, Tags und Interfaceklassen der UML-Profile in die Modellhierarchie eingebunden. Auf sie ist nur Lesezugriff möglich und sie entstammen einer Bibliothek, sind also kein echter Teil der Modellierung. Durch ihre Einbindung werden die von ihnen definierten Elemente im Namensraum des restlichen Modells sichtbar.

Entsprechend dieser Modellstruktur werden in den folgenden Abschnitten die Anforderungs-, System- und Testmodellierung detaillierter beschrieben.

8.3.1 Die Anforderungsmodellierung

Die Anforderungen wurden den SysML-Vorgaben gemäß erfasst, wobei – wie in Abschnitt 7.2.1 beschrieben – der sehr generische Stereotyp `≪Requirement≫` durch die Unterkategorien FRS, SRS, RAMSS ergänzt wurde.

In Anlehnung daran ist das Paket der initialen Anforderungen weiter unterteilt in funktionale und nichtfunktionale Anforderungen. Diese beiden Untergruppen werden durch Pakete mit referenzierten Dokumenten und einer Definition des Systemkontextes (als Anwendungsfalldiagramm) ergänzt.

Das SRS-Paket ist in weitere Unterpakete für verschiedene Systemkomponenten wie Kommunikationsprotokolle oder den FSK-Modulator unterteilt. Zusätzlich werden an relevanten Stellen die zugrundeliegenden initialen Anforderungen referenziert, um von Beginn an eine Anforderungsverfolgung zu ermöglichen.

Dabei zeigt sich bereits ein Vorteil der UML-Modellierung: eine Anforderung – also eine UML-Klasse – kann auf beliebig vielen Diagrammen verwendet werden aber repräsentiert doch immer das gleiche Element. Dadurch wird eine Änderung an der Klasse bzw. der Anforderung unmittelbar auf allen Diagrammen wirksam und die Integrität des Modells ist sichergestellt.

SysML ermöglicht die Erfassung zahlreicher zusätzlicher Informationen als zum `≪Requirement≫`-Stereotyp gehörige *Tagged Values* (sozusagen Attribute des Stereotyps). Diese Werte beziehen sich vor allem auf eine logische Verknüpfung von Anforderungen und auf die Erfassung grundsätzlicher Daten wie eine ID oder einen Anforderungstext.

In der Fallstudie wurde von der vergleichsweise aufwändigen und im Diagramm unübersichtlichen Methode der Datenspeicherung in den Tags abgesehen. Die Verknüpfungsinformationen werden ausschließlich über Callouts oder explizite Abhängigkeitsbeziehungen hinterlegt. Die manuelle Vergabe einer ID ist ebenfalls nicht notwendig, da Artisan Studio bereits für jedes Objekt eine eindeutige Kennzeichnung erzeugt. Diese für den Einsatz im Klartext nur bedingt sinnvoll zu verwendende Ziffernfolge kann für den praktischen Einsatz auch durch das direkte Verknüpfen von Objekten via *Drag and Drop* oder durch den voll qualifizierenden Namen der Anforderung ersetzt werden.

Der einzige konsequent verwendete Tagged Value ist `text` (SysML) bzw. `shortText` und `longText` (Fallstudie). Die Aufteilung wurde gewählt, um kürzere Anforderungsbeschreibungen direkt im Diagramm darzustellen, während längere Texte in einer externen Datei untergebracht sind, auf die `longText` verweist.

Die Anforderungstexte orientieren sich durchgängig an der von Rupp empfohlenen Sprachschablone [RS04]. Die gesteigerte sprachliche Exaktheit wird allerdings durch einen eher monotonen Wortlaut erkauft, der längeres Lesen der Anforderungen sehr anstrengend macht. Insgesamt wurden auf diese Weise 178 Anforderungen im SRS-Paket erfasst.

8.3.2 Die Systemmodellierung

Die Systemmodellierung beinhaltet, wie oben bereits angedeutet, die Systemarchitektur und das System- bzw. Softwaredesign. Kernaspekt der Architekturmodellierung ist das Zusammenspiel der DSP-Komponenten, die Interaktion zwischen DSP und FSK-Modulator und die Beschreibung der externen Schnittstellen.

Das System wurde dazu unter verschiedenen Aspekten (engl. *Views*) modelliert, was im Folgenden am Beispiel der seriellen Schnittstelle zum RailSiTe verdeutlicht werden soll.

Aus *funktionaler Sicht* stellt die serielle Verbindung zum RailSiTe Befehle wie beispielsweise das Speichern eines Balisentelegramms, das Speichern einer Sequenzdefinition oder das Kodieren eines Telegramms zur Verfügung. In der „Gegenrichtung“ erwartet der DSP Funktionen wie z. B. die Rückmeldung des Kommunikations- und Systemstatus (Abb. 8.11)

Aus *Sicht der Sicherungsschicht* werden an der seriellen Schnittstelle Funktionen wie das Versenden oder Empfangen eines Datenpakets und dessen Prüfsummenkontrolle realisiert.

Aus *physikalischer Sicht* kann die Schnittstelle einen wie auch immer gearteten Datenblock versenden bzw. empfangen und es können Kommunikationsparameter eingestellt werden.

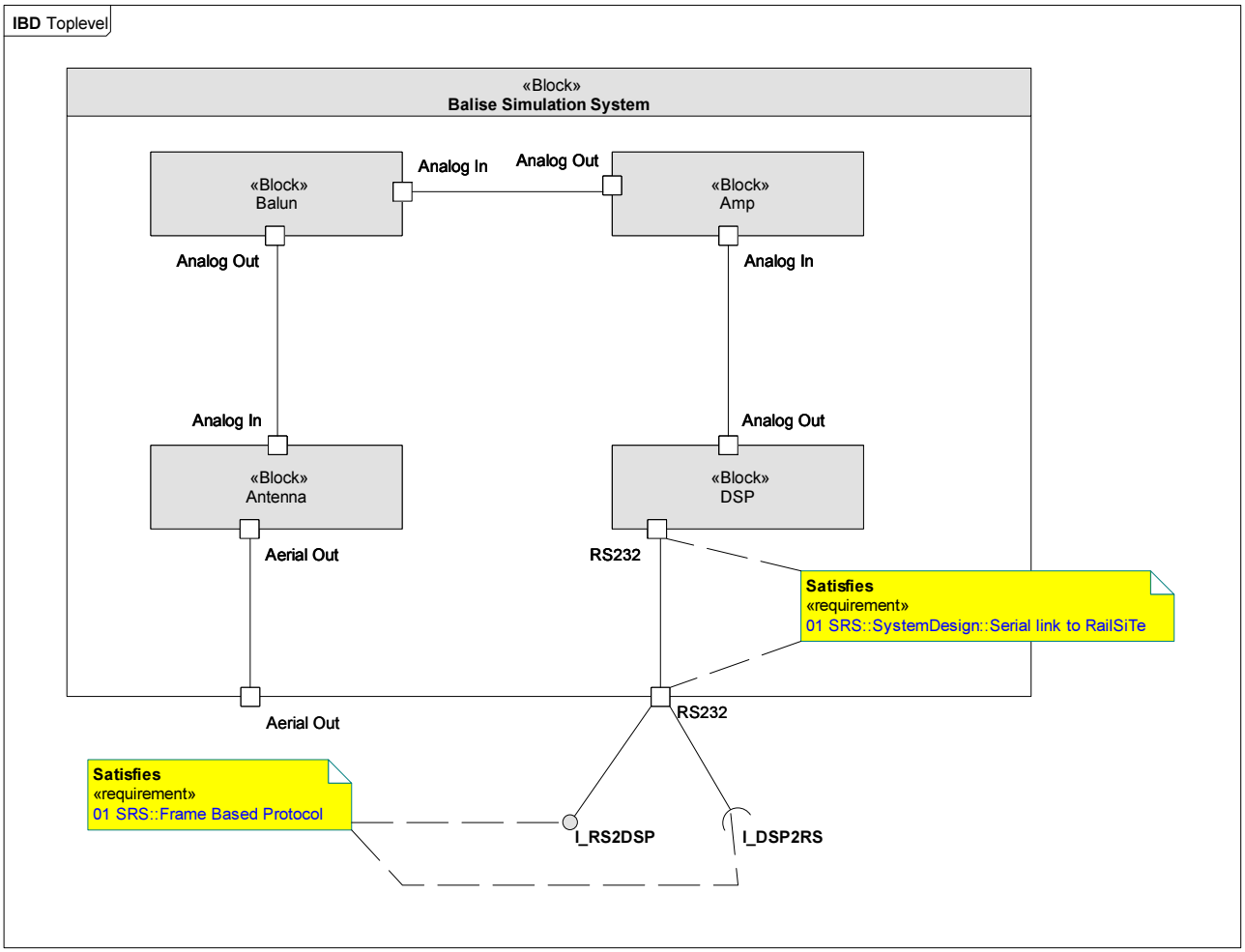

Abbildung 8.11: Strukturdiagramm der obersten Ebene der Systemarchitektur. Neben der Schnittstellendefinition auf Applikationsebene (erkennbar an den „Ball-and-Socket“-Symbolen) ist auch die Verknüpfung zu Systemanforderungen durch sogenannte „Satisfy-Callouts“ erkennbar.

Insgesamt korrespondiert diese „Multi-View-Modellierung“ mit den Ebenen des Kommunikationsprotokolls zwischen Balisensimulation und RailSiTe. Aus diesen Grund werden auch nicht alle Teile des Systems auf allen Ebenen neu oder wiederholt modelliert, sondern nur die relevanten Teile. Der interne Telegrammspeicher wird beispielsweise nur auf der obersten, der funktionalen Ebene, modelliert und stellt dort Funktionen zum Speichern, Abrufen und zur Typprüfung eines Balisentelegramms zur Verfügung. Zum einen wird der Telegrammspeicher erst auf der obersten Systemebene benötigt und zum anderen war die Modellierung weiterer Ebenen innerhalb des Telegrammspeichers nicht notwendig.

Größere, komplexere Systeme werden bei Multi-View-Modellierung nicht nur auf verschiedenen funktionalen Ebenen beschrieben, sondern auch aus der Sicht verschiedener Nutzergruppen oder Beteiligter am Entwicklungsprozess (sogenannte Stakeholder). Dadurch wird eine möglichst umfassende Betrachtung des Systems erreicht und die Wahrscheinlichkeit von Entwicklungsfehlern reduziert. Allerdings sieht sich diese Methode auch dem nicht ganz ungerechtfertigten Vorwurf ausgesetzt, dass prinzipiell derselbe Inhalt mehrfach beschrieben wird und dadurch zusätzlicher Aufwand entsteht. Darüber hinaus besteht bei der Mehrfachmodellierung von Inhalten natürlich auch immer ein Risiko der Inkonsistenz innerhalb des Modells.

Das Systemdesign des Fallstudienmodells ist geprägt durch die Klassenmodellierung der DSP-Software. In ihr werden die verwendeten Softwareklassen und deren Beziehungen beschrieben. Für die Erfassung des Systemverhaltens und zur Codererzeugung wurden die Klassen, bei denen es sinnvoll war, durch Zustandsautomaten hinterlegt. Darüber hinaus wurden Methoden und Zustandstransitionen direkt mit Quelltext versehen, der unverändert in die erzeugten Quelldateien übernommen wird.

Abbildung 8.12 zeigt beispielsweise, wie die Umschaltung zwischen beiden DSP-Betriebsarten „RailSiTe“ und „Interaktiver Debugmodus“ mit Hilfe von Zustandsmaschinen realisiert wurde. Die verschiedenen FSMs innerhalb des Systems kommunizieren dabei über Methodenaufrufe, die ihrerseits Transitionsübergänge in der zur FSM zugehörigen Klasse auslösen. Die Übergabe von Daten an Folgezustände muss dabei allerdings manuell erfolgen, also beispielsweise über einen vorhergehenden händisch eingefügten Methodenaufruf mit entsprechenden Parametern. Die direkte Übergabe von Parametern im Transitionstrigger ist leider nicht möglich, was als Nachteil bei der Modellierung angesehen werden muss.

Durch das direkte Hinterlegen des manuell hinzugefügten Codes in den zugehörigen FSMs und Methoden konnte die gesamte Codebasis im Modell gespeichert werden. Eine Nachbearbeitung des aus dem Modell erzeugten Codes war somit nicht notwendig und die generierten Quelldateien konnten direkt durch die DSP-Entwicklungsumgebung übersetzt und auf die Zielplattform übertragen werden.

Die in Artisan Studio realisierten Funktionen umfassen die grundlegende Systeminitialisierung einschließlich der IRQ-Konfiguration und der Implementierung der zugehörigen IRQ-Handler in (statischen) Klassenmethoden, die in Abbildung 8.12 gezeigte Betriebsartenumschaltung und die Umsetzung des Kommunikationsprotokolls zwischen RailSiTe und DSP bis einschließlich der Sicherungsebene. Der so realisierte Funktionsumfang war ausreichend, um die grundsätzliche Eignung der Methode experimentell zu beweisen.

8.3.3 Die Testmodellierung

Die Erfassung der Tests umfasste zwei Schritte: zum einem die Testfallableitung (überwiegend mittels Klassifikationsbäumen und CTE XL, s. Abschnitt 6.4.2) und zum anderen die Modellierung der erzeugten Tests in UML unter Anwendung des U2TP. Hier soll exemplarisch der im Rahmen der Fallstudie entstandene Klassifikationsbaum für die Testfallableitung auf Sicherungsebene des DSP-RailSiTe-Protokolls beschrieben werden.

Wie Abbildung 8.15 auf der gegenüberliegenden Seite zeigt, wurden den unabhängig voneinander verwendbaren Parametern des Protokollrahmens eigene Zweige im Klassifikationsbaum zugewiesen, deren Blätter wiederum den einzelnen Wertekategorien des Parameters entsprechen. Bei der Definition der Kategorien wurde auch der Fall berücksichtigt, dass das entsprechende Datenfeld nicht oder nicht vollständig im Datenpaket enthalten ist. Ein solcher Fehler würde zu einer Verschiebung und damit zu einer Fehlinterpretation der nachfolgenden Datenfelder führen und muss ebenfalls getestet werden.

Besonderes Augenmerk wurde auf die Länge der Nutzdaten gerichtet. Hier unterscheidet der Klassifikationsbaum zwischen der *tatsächlichen* Länge („Payload Size“) und der im Protokollrahmen *angekündigten* Länge („Length Indicator“). Differieren diese Angaben, werden entweder weniger Daten gesendet als erwartet, was zu einem Timeout führen muss, oder überzählige Daten werden als Folgepaket interpretiert, das entsprechend behandelt werden muss.

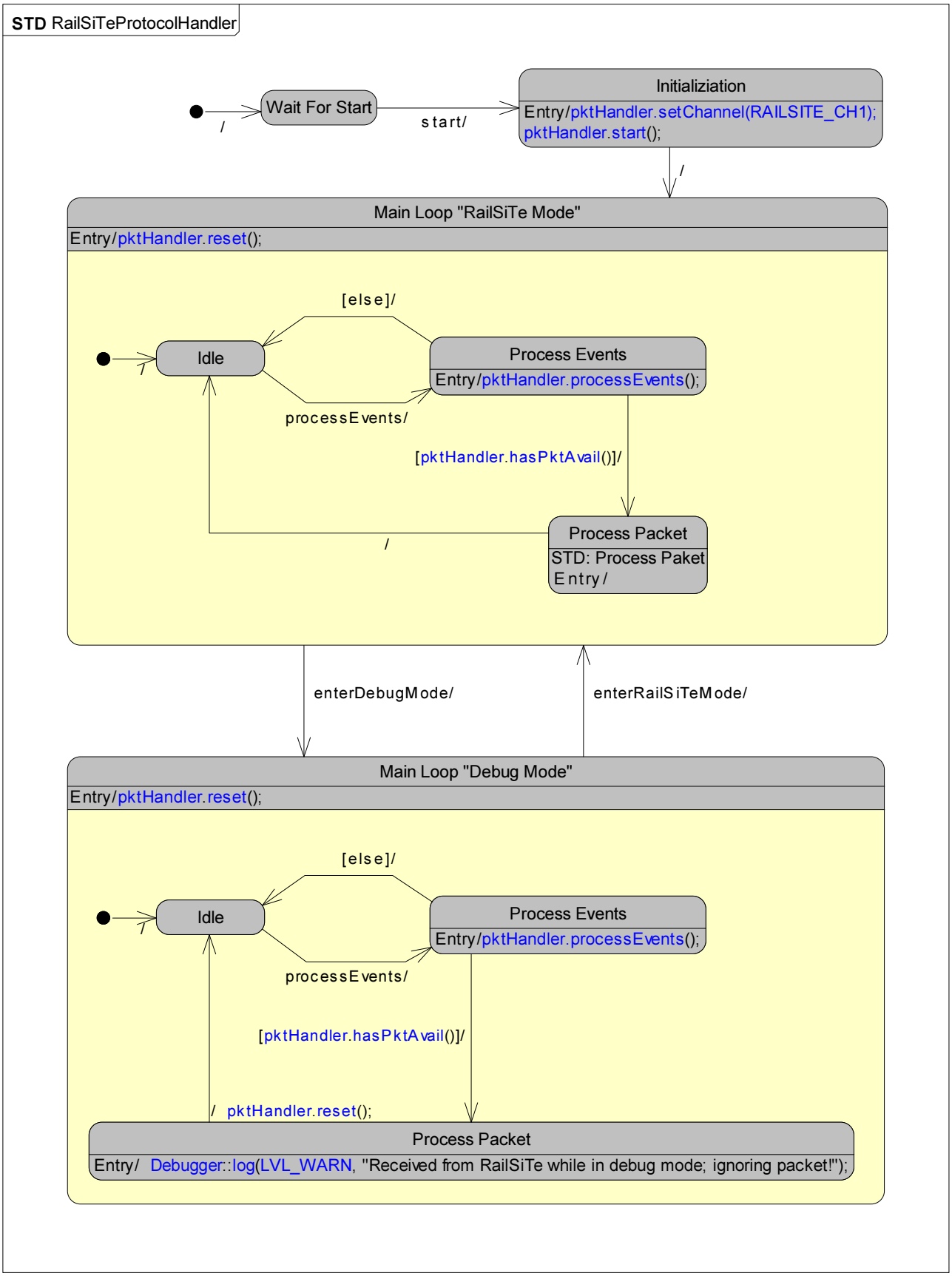

Abbildung 8.12: Modellierung der Umschaltung zwischen RailSiTe-Modus und interaktivem Modus in der Zustandsmaschine des Protokollhandlers (oberste Ebene einer hierarchisierten FSM

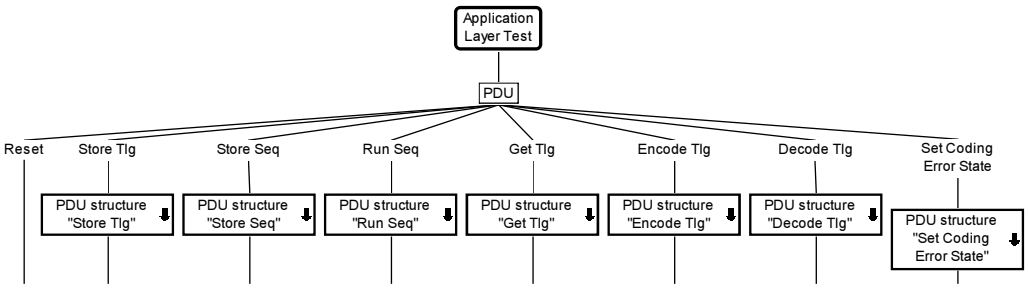

Abbildung 8.13: Klassifikationsbaum für die Applikationsebene des Kommunikationsprotokolls. Der Baum weist Verzweigungen zu Unterbäumen auf.

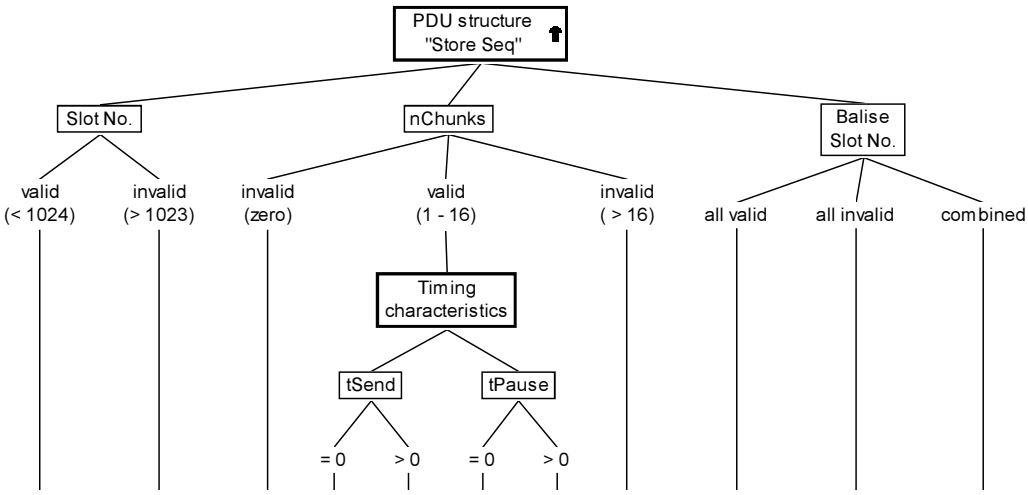

Abbildung 8.14: Beispiel eines Unterbaums zu Abbildung 8.13; dargestellt sind die Parameter zum Speichern einer Balisentelegrammsequenz

Zusätzlich fließt in Form des Feldes „Overall Size“ noch WhiteBox-Wissen in den Klassifikationsbaum ein: der Protokollinterpreter verwendet einen internen Datenpuffer, der für die anwendungsüblichen Datenpakete mehr als ausreichend bemessen ist, aber ein Paket mit theoretischer Maximallänge nicht aufnehmen kann. Die für den Fall der Pufferüberschreitung notwendige Fehlerbehandlung (Abweisen des Paketes) muss ebenfalls getestet werden und geht daher mit in den Klassifikationsbaum ein.

Die logischen Nebenbedingungen, die für diesen Klassifikationsbaum definiert wurden, spiegeln die Struktur des Datenpakets wieder. Wird beispielsweise für die Transaktionsnummer, die eines der ersten Felder im Datenpaket ist, der Wert „incomplete“ oder „not provided“ gewählt – was einem unzulässig verkürztem Datenpaket entspräche –, dürfen alle nachfolgenden Datenfelder, wie z. B. die Nutzdaten, nur die Länge Null beziehungsweise „not provided“ aufweisen. Durch derartige Nebenbedingungen kann, wie bereits in Abschnitt 6.4.2 erläutert, die Anzahl der erzeugten Testfälle sinnvoll limitiert werden.

Dieses Verfahren beschränkte sich nicht nur auf die Behandlung der Sicherungsschicht des Protokolls, sondern es wurde auch auf die Nutzdaten (also die Applikationsebene) angewendet. Da diese Ebene natürlich erheblich komplexer ist als die Sicherungsebene, kamen dabei erweiterte Funktionen von CTE XL zum Einsatz, wie beispielsweise die Aufteilung eines Klassifikationsbaums in separate Unterbäume. Exemplarisch zeigt dies Abbildung 8.13 mit dem zugehörigen Unterbaum zum Speichern von Sequenz- und Timingdaten in Abbildung 8.14. Ohne Berücksichtigung der Fehlermodi für die Signalerzeugung wurden auf diese Weise 140 Testfälle für die Applikationsebene der Balisensimulation abgeleitet, wobei dem schon erste Vereinfachungen zugrunde liegen (beispielsweise nur Sequenzen mit Null, weniger oder mehr als 16 Balisentelegrammen, siehe Abb. 8.14; denkbar wären auch einzelne Testfälle für ein, zwei, drei Telegramme usw.). Die weitere Be-

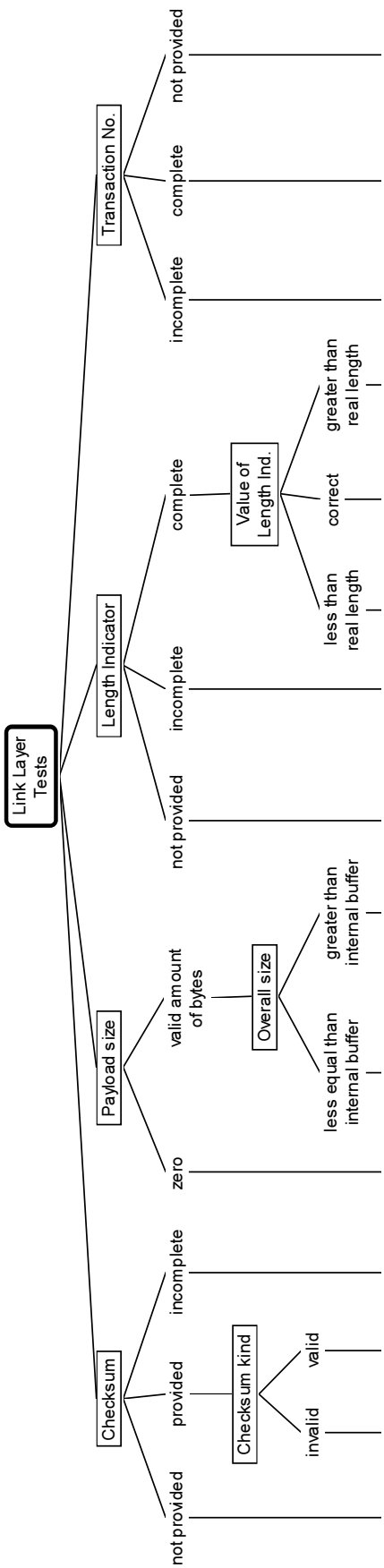

Abbildung 8.15: Klassifikationsbaum für die Testfallableitung auf Ebene der Sicherungsschicht der DSP-RailSiTe-Kommunikation

schreibung des Modellierungsverfahrens soll sich aber aus Konsistenzgründen auf die Sicherungsschicht beschränken. Die Methode ist unverändert auch auf die Applikationsschicht der Software anwendbar.

Die erzeugten Testfälle für die Sicherungsebene wurden thematisch sortiert und zu entsprechenden Testsequenzen kombiniert (Sequenzen mit allen Testfällen ungültiger Checksumme, Sequenzen mit allen Testfällen falscher Längeninformation, usw.). Ebenso wurde jedem Testpaket das von der Balisensimulation zu erwartende Antwortpaket zugeordnet: kein Paket (Timeout), Fehler oder Bestätigung.

Die Übernahme der Sequenzen in das UML-Modell erfolgte manuell, wobei zunächst für jeden Testfall ein zugehöriges Datentelegramm entworfen und in einem Datenpool gespeichert wurde (Abb. 8.16, Ausschnitt ohne Antwortpakete von der Balisensimulation).

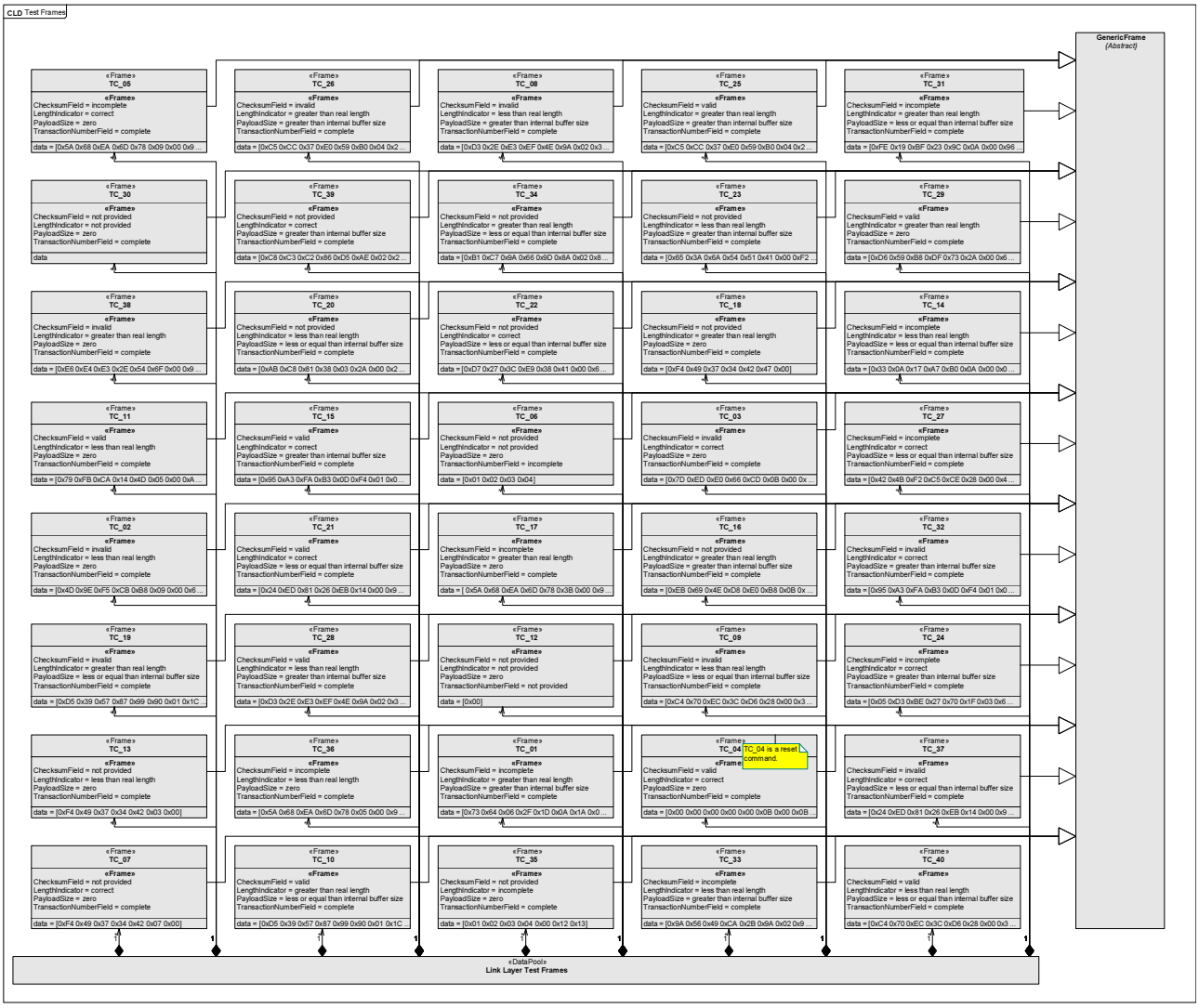

Abbildung 8.16: UML-Klassendiagramm des U2TP-Datenpools für die Testtelegramme der Sicherungsschicht des DSP-RailSiTe-Protokolls (Ausschnitt; der Schwerpunkt der Abbildung liegt auf der Struktur des Modells und weniger auf den Beschriftungen der Klassen)

Die Erzeugung der Testdaten erfolgte teilautomatisiert. Ein Skript erzeugte nach Vorgabe der Randbedingungen des Testfalls (Gültigkeit der Checksumme, etc) entsprechende Datenpakete aus Zufallszahlen und gab sie in byteweiser hexadezimaler Schreibweise am Bildschirm aus (sogenannter *Hexdump*). Das Skript musste jedoch für jedes Paket neu gestartet und manuell parametriert werden. Darüber hinaus stellte das Skript sicher, dass bei Paketen mit Überlänge die Abschnitte der Nutzdaten, in denen aufgrund der fehlerhaften Längeninformation vom Empfänger die Checksumme erwartet wird, keine plausiblen Daten stehen, die ins-

gesamt zu einem gültigen Telegramm führen würden. Ebenso wurde durch das Skript dafür gesorgt, dass der überschüssige Teil eines überlangen Paketes nicht wieder ein gültiges Paket darstellt.

Die Testsequenzen wurden als UML-Sequenzdiagramm abgelegt und entsprechen vom Aufbau her denen in Abbildung 8.17. Die Ports der Testkomponente und des SUT bilden die Lebenslinien und die auszutauschenden Daten stellen die Nachrichtenströme zwischen diesen Lebenslinien dar. Die Konfiguration der Testanordnung wurde in einem Strukturdiagramm hinterlegt, aus dem auch die Zuordnungen der im TTCN-3-Skript verwendeten Portnamen auf physikalische Ports sowie die grundsätzlich möglichen Kommunikationsbeziehungen hervorgehen (vgl. Abb. 7.8 auf S. 103).

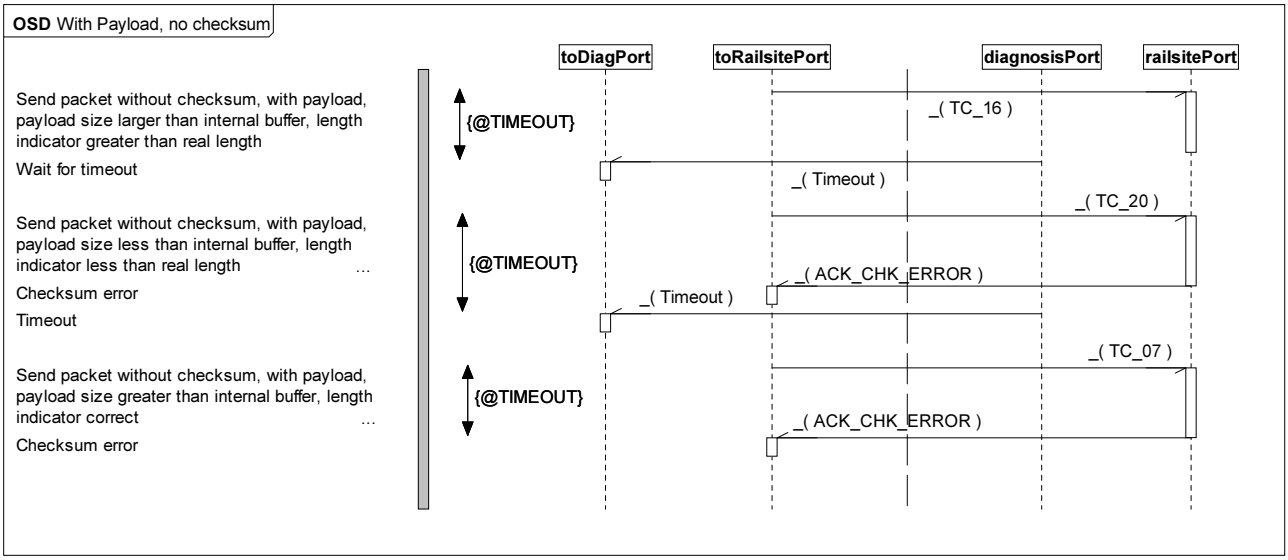

Abbildung 8.17: Ausschnitt eines der Fallstudie entnommenen Sequenzdiagramms zur Testfallbeschreibung (hier: korrekte Erkennung von Telegrammen ohne Prüfsumme)

Von den sehr differenzierten Möglichkeiten des U2TPs, zu einer Testbewertung zu kommen (verschiedene Arbiter, Defaults, etc), wurde bei der Modellierung kein Gebrauch gemacht; das Ergebnis eines Testfalls ist stets „pass“ oder „fail“ und sollte ein Testfall negativ bewertet werden, so gilt das auch für die gesamte Testsequenz[5].

8.3.4 Vom Testmodell auf die Testanlage

Für den Übergang von den Testfallmodellen zur automatischen Testausführung auf der Testanlage wurden basierend auf den im vorangegangenen Arbeitsschritt entstandenen UML-(Sequenz-)Diagrammen und dem in Abschnitt 7.4 erläuterten TTCN-3-Generators Testskripte erzeugt.

Diese Skripte wurden in der von Janke [Jan07] erstellten Testumgebung zur Ausführung gebracht. Die Testumgebung kann dazu über eine grafische Oberfläche bedient und die Ausführung eines bestimmten Testfalls angestoßen werden. Alternativ kann die auszuführende TTCN-3-Datei wird als Kommandozeilenparameter übergeben werden und die Testausführung läuft ohne jede weitere Benutzerinteraktion ab.

Die letztgenannte Variante – also die automatische Ausführung einer TTCN-3-Datei – stellt die Grundlage für die Testautomatisierung dar. Dank dieser Schnittstelle können skriptgesteuert beliebig viele Testfälle zur Ausführung gebracht werden und das Testergebnis, das von der Testumgebung auf der Standardausgabe ausgegeben wird, kann automatisch protokolliert werden.

[5] Zum einen erfordert die Art der Testfälle kein differenziertes Bewertungsschema und zum anderen wird dies in der derzeitigen Version der Testumgebung auch nicht unterstützt

8.4 Ergebnisse und Bewertung der Fallstudie

Die Fallstudie hat gezeigt, dass die durchgängige System- und Testfallmodellierung mit UML bzw. den Profilen SysML und U2TP möglich ist. Es wurde erreicht, dass das Modell die gemeinsame, konsistente Basis für die Anforderungserfassung, die Systemarchitektur, den Systementwurf einschließlich Codeerzeugung, die Testfallverwaltung und die automatisierte Testausführung bildet. Dank des strukturierten modellbasierten Ansatzes wurde implizit eine gute Dokumentation des Systems erzeugt und durch die Nachverfolgung der zugrundeliegenden Anforderungen eine hohe Produktqualität sichergestellt.

Ein methodischer Nachteil ergibt sich aus der manuellen Überführung der mit CTE XL erzeugten Testfälle in das Modell; dieser Schritt ist mit einem hohen Aufwand verbunden, der außerdem an einigen Stellen sehr monotone Arbeitsschritte erfordert (beispielsweise die Übernahme der Testvektoren in den U2TP-Datenpool). Eine Automatisierung dieses Vorgangs ist allerdings nur bedingt denkbar, da zwischen den Ergebnissen von CTE XL (Testfälle) und der schlussendlichen Modellierung (Test Context, Testsequenzen) noch Informationen in Form von Domänen- bzw. Kontextwissen hinzugefügt werden müssen.

Die eingangs von Kapitel 7 prognostizierten Vorteile konnten durch die Fallstudie bestätigt werden. Die Erfassung der Anforderungen als Teil des UML-Modells erwies sich aufgrund der sich daraus ergebenden Möglichkeiten der Konsistenzprüfung bzw. Nachverfolgbarkeit als positiv. Die Verwendung der UML als einheitliches Beschreibungsmittel und die damit verbundene Darstellungen von Anforderungen, System und Tests ohne Notationsbruch ermöglichte effizientes und angenehmes Arbeiten. Das daraus resultierende homogene Modell vereinigt alle Informationen über das System und den Entwicklungsvorgang, so dass das Modell viele sonst notwendige (textuelle) Dokumentationen überflüssig macht.

Im Sinne des in Abschnitt 4.4 erläuterten BMW-Prinzips müssen jedoch nicht nur das Beschreibungsmittel und die Methode geeignet sein, sondern auch eine adäquate Werkzeugunterstützung gegeben sein. Zentrales Werkzeug – neben CTE XL – ist natürlich das CASE-Tool, also im vorliegenden Fall Artisan Studio.

Bei der Durchführung der Fallstudie zeigte sich, dass werkzeugseitig einige Abläufe effizienter gestaltet werden sollten, um vor allem immer wiederkehrende Aktivitäten zu beschleunigen. Als Beispiel sei hier das Anlegen von Klassen beim Einpflegen der Anforderungen (Erstellen einer neuen Klassen, Editieren des Namens, Zuweisung eines Stereotypen, Festlegung bestimmter Darstellungsparameter) oder das Einpflegen der `≪satisfy≫`- und `≪verify≫`-Abhängigkeiten (Erstellen einer Dependency, Anpassung der Dependency-Optionen, Zuweisung eines Stereotypen) genannt.

In neueren Versionen von Artisan Studio, die erst nach Anfertigen großer Teile der Fallstudie verfügbar wurden, sind unter Verwendung der neuen Technik des „Ergonomic Profiling“ verbesserte Anpassungsmöglichkeiten des Werkzeugs an Profile geschaffen worden. Beispielsweise ist das Anlegen neuer Anforderungen oder die Erzeugung von Callouts jetzt erheblich effizienter gelöst. Insgesamt wird SysML durch eine Vielzahl neuer Symbole, Verknüpfungen und Menüpunkte nun intensiv unterstützt. Diese Weiterentwicklung des Programms zeigt, dass die Möglichkeiten der neuen „Sprache“ SysML auch von den Werkzeugentwicklern positiv beurteilt und in die eigenen Programme aufgenommen werden. Im Falle von Artisan Studio muss noch erwähnt werden, dass das „Ergonomic Profiling“ nicht SysML-spezifisch ist, sondern auch anderen Profilen wie beispielsweise U2TP offen steht.

Insgesamt hat die Fallstudie gezeigt, dass die durchgängige Anforderungs-, System- und Testfallbeschreibung in einem einzigen homogenen UML-Modell ein wirkungsvolles Verfahren ist, um die Entwicklung *wartbarer* und strukturierter Soft- und Hardware zu vereinfachen. Eine methodisch bessere Ankopplung der Testfallerzeugung muss Gegenstand zukünftiger Arbeiten werden, während eine Verbesserung der Werkzeugunterstützung, die ebenfalls für einige Probleme während der Fallstudie sorgte, schon heute absehbar ist.

9 Zusammenfassung und Ausblick

Es wurde auf Basis methodischer Betrachtungen gezeigt, wie durch den Einsatz von UML und geeigneter UML-Profile die Systementwicklung in den Bereichen Anforderungsmanagement, Codeerzeugung, Testfallbeschreibung und Testautomatisierung entscheidend verbessert werden kann. Dies wurde anhand einer Fallstudie empirisch belegt.

Ausgangspunkt dieser Untersuchungen war die Fragestellung, wie der Entwurf komplexer Eisenbahnleit- und -sicherungstechniksysteme durch moderne modellbasierte Entwicklungsmethoden unterstützt werden kann. Dazu wurde zunächst erarbeitet, welche Eigenschaften Modelle im Allgemeinen aufweisen und wie im Besonderen technische bzw. betriebliche Modelle in der Domäne Eisenbahn beschaffen sein müssen.

Darauf aufbauend wurden verschiedene formale und semiformale Beschreibungsmittel vorgestellt, mit denen sich die zuvor untersuchten Modelle realisieren lassen. Aufgrund der Flexibilität, der Verbreitung und der jüngsten Entwicklungen der Sprache konzentrierten sich die weiteren Ausführungen auf die UML, die zusammen mit ihren Profilen SysML und U2TP ausführlich behandelt wurde.

Neben den technischen Fragestellungen der Modellierung und der Wahl des Beschreibungsmittels wurde auf organisatorischer Seite auch die Bedeutung des Entwicklungsprozesses für den Entwicklungserfolg betrachtet. Dazu wurden gängige Vorgehensmodelle vorgestellt und eine inkrementelle Herangehensweise innerhalb des V-Modells als am erfolgversprechendsten beurteilt. Leichtgewichtigere, agile Methoden bieten zwar grundsätzlich Vorteile gegenüber traditionellen Prozessformen, sind aber im Umfeld der Entwicklung sicherheitsrelevanter Systeme aufgrund der zu erfüllenden Normen und Richtlinien als problematisch einzustufen.

Mit Hinblick auf die Sicherheitsrelevanz der zu betrachtenden Systeme wurde das zentrale Entwicklungsthema „Testen“ in einem eigenen Kapitel detailliert untersucht. Neben der grundsätzlichen Testmethodik steht dabei die Modellierung von Testfällen und Testumgebungen mit der UML im Mittelpunkt. Ebenso wurden gängige Verfahren zur automatischen Testfallerzeugung vorgestellt und zum Teil anhand von Praxisbeispielen verdeutlicht.

In Vorbereitung auf die Fallstudie wurde dann die Verknüpfung von Anforderungs-, Design- und Testfallinformationen in einem einzigen homogenen UML-Modell hergeleitet. Mit Blick auf das Requirements Engineering ist auch die Ableitung von Aussagen über den momentanen Testfallabdeckungs- und Implementierungsgrad durch Auswertung der Beziehungen zwischen verschiedenen Modellelementen behandelt worden. Darüber hinaus wurde gezeigt, wie eine Überführung der UML-Testsequenzen in die Testfallbeschreibungssprache TTCN-3 prinzipiell realisiert werden kann.

Die so geschaffenen methodisch-theoretischen Grundlagen wurden in einer Fallstudie einer praktischen Untersuchung unterzogen. Dazu wurde ein System, das im Simulationslabor RailSiTe eingesetzt wird und der Erzeugung von Transpondersignalen dient, gemäß dem zuvor abgeleiteten Verfahren nachmodelliert. Voraussetzung dafür war eine geeignete Kopplung von Modellierungs-, Codeerzeugungs-, Testfallableitungs- und Modellanalysewerkzeugen, die auf Basis des CASE-Tools Artisan Studio realisiert wurde. Die Kopplung umfasste auch die automatisierte Auswertung von Anforderungsbeziehungen und die Implementierung eines exemplarischen TTCN-3-Generators, der UML-Sequenzdiagramme in ausführbaren TTCN-3-Code transformiert. Insgesamt wurden durch die Werkzeugkopplung und das erzeugte Modell folgende Eigenschaften erfüllt:

- Erfassung von funktionalen und nicht-funktionalen Anforderungen, der Systemarchitektur, der Systemimplementierung und zugehöriger Testfälle in einem einzigen UML-Modell
- Die Erzeugung von C++-Code aus dem Modell und die direkte Übersetzung des Codes in der Entwicklungsumgebung für die Zielplattform (Digitaler Signalprozessor von Texas Instruments)
- Die automatische Generierung von TTCN-3-Code aus im Modell hinterlegten UML-Sequenzdiagrammen und dessen direkte Ausführung in einer dafür vorgesehenen Testumgebung [Jan07]
- Die toolbasierte Auswertung von Beziehungen zwischen Anforderungen, Implementierung und Testfall direkt aus dem UML-Modell einschließlich statistischer Ausgaben und der gezielten Suche nach nicht getesteten oder nicht implementierten Anforderungen

Die Evaluation der Fallstudie zeigte, dass sich die theoretisch vorhergesagte Kopplung von Testfall- und Systemmodell und die mit ihr verbundenen Vorteile auch praktisch realisieren lassen. Die Modellierung an sich war jedoch mitunter etwas mühsam, was aber nicht in einer grundsätzlichen Schwäche der vorgestellten Methode sondern im Status des verwendeten CASE-Tools begründet lag. Durch mittlerweile erfolgte Aktualisierungen des Werkzeugs konnten diese Probleme größtenteils behoben werden.

Ausblick

Im Rahmen dieser Arbeit konnte nur die grundsätzliche Eignung und Realisierbarkeit der vorgestellten Methode gezeigt werden. Eine Weiterentwicklung der hier gezeigten Ansätze ist daher auf vielfältige Weise möglich. In erster Linie ist in diesem Zusammenhang die automatisierte Übernahme der mittels CTE XL erzeugten Testfälle in das UML-Modell zu nennen, da eine manuelle Übernahme (wie in der Fallstudie) im Allgemeinen sehr aufwendig und auch fehleranfällig ist. Im Kontext der Testautomatisierung ist eine Erweiterung des TTCN-3-Generators um komplexere Konstrukte wie alternative Abläufe, Schleifen etc. denkbar. Gleiches gilt für die TTCN-3-Testumgebung, die bislang ebenfalls nur einen grundlegenden Befehlsumfang unterstützt.

Abseits dieser eher technischen Fragestellungen ist noch zu untersuchen, wie die hier beschriebenen Ansätze am besten und nutzbringendsten in einen existierenden Entwicklungsprozess eingegliedert werden können, wobei auf den Ergebnissen von Uhtenwoldt [Uht06] aufgebaut werden kann.

Die Eingliederung neuer Methoden und Werkzeuge in einen realen Entwicklungsprozess war auch eine der Fragestellungen, die im Rahmen einer anderen Fallstudie behandelt wurden. Gegenstand der Untersuchung war das Werkzeug SCADE (s. Abschnitt 3.3.2 auf S. 38) und dessen Eignung für die Entwicklung von Eisenbahnleit- und -sicherungstechnik. Im Verlauf dieser Fallstudie, die von der Siemens AG initiiert wurde, entstand ein Modell eines einfachen, INDUSI-ähnlichen Zugsicherungssystems. Unter anderem wurde aus dem Modell Code erzeugt und auf der Zielplattform, die in realen Fahrzeugen zum Einsatz kommt, zur Ausführung gebracht. Darüber hinaus wurden auch die Möglichkeiten des Model Checkers und Simulators von SCADE erprobt und der Einfluss des Einsatzes von SCADE auf das entwicklungsbegleitende Testen analysiert.

Als Vorteile, die der Einsatz von SCADE in der Entwicklung mit sich bringt, zeigte sich vor allem:

- erhöhte Typsicherheit (Schutz vor Variablenüberläufen, etc.) durch strenge Typisierung von SCADE
- Ersatz von Modultests durch formale Verifikation und Modellsimulation; durch Einsatz qualifizierter Codegeneratoren und Compiler ist die Gleichwertigkeit von Modell und (Binär-)Code gewährleistet
- Unterstützung der Anforderungsabdeckungsanalyse durch die sogenannte „Model Test Coverage“, die eine Beziehung zwischen Anforderung und Testfall herstellt.

Alle diese Vorteile werden dadurch erkauft, dass Systemverhalten und Systemanforderungen zu einem möglichst frühen Entwicklungszeitpunkt formalisiert als SCADE-Modell abgelegt werden. Somit wird der (Test-)Aufwand, zumindest beim erstmaligen Einsatz von SCADE, eher verlagert als reduziert.

Die möglichst frühe formale Erfassung von Anforderungen und Systemsollverhalten ist im Sinne eines durchgängigen Requirements Engineering und eines stringenten V-Prozesses sicher zu begrüßen. Allerdings muss auch berücksichtigt werden, dass naturgemäß in einem frühen Entwicklungsstadium noch keine detaillierte bzw. formalisierte Spezifikation des Gesamtsystems gegeben werden kann. Hier muss zunächst „unscharf“ formuliert werden, weil die notwendige Schärfe erst im Laufe des Entwicklungsprozesses erarbeitet wird. SCADE ist daher auf den unteren Ebenen des V-Modells am effektivsten einzusetzen und sollte mit anderen, vorgelagerten Verfahren – beispielsweise einer semiformalen UML-Modellierung – kombiniert werden.

Hierin liegt eine weitere Chance für den Einsatz der in dieser Arbeit eingeführten Methode. UML erlaubt die oben geforderte „unscharfe“ Modellierung, während SCADE unbestreitbar bessere Ergebnisse hinsichtlich Codeerzeugung und formaler Verifikation erbringt als die UML. Die Nutzung der hier vorgestellten Verfahren zur UML-basierten Anforderungsverwaltung, Systemanalyse und Testfallbeschreibung kombiniert mit dem Einsatz von SCADE zur konkreten Implementierung verspricht somit eine wirksame Kombination aus Beschreibungsmitteln, Methoden und Werkzeugen.

In weiterführenden Arbeiten sollte daher untersucht werden, ob die Vorteile dieses Ansatzes mögliche Nachteile wie etwa den Notationsbruch zwischen UML und SCADE oder die Zunahme der Werkzeug- und Methodenvielfalt rechtfertigen. In jedem Fall zeigt die Möglichkeit solch kombinierter Ansätze die Flexibilität der in dieser Dissertation vorgestellten Methoden. Es ist daher zu hoffen, dass durch die kombinierten, UML-basierten Testfall- und Systemmodelle die Lücke zwischen informalen Kundenanforderungen und formaler Implementierung künftig effektiv geschlossen werden kann.

A Abkürzungsverzeichnis

AEG – Allgemeines Eisenbahngesetz

API – Application Programming Interface

BAV – Bundesamt für Verkehr (Schweiz)

BNF – Backus-Naur-Form bzw. Backus-Normalform

BOStrab – Verordnung über den Bau und Betrieb der Straßenbahnen

CAT – Customer Acceptance Test

CCS – Code Composer Studio (Entwicklungsumgebung für digitale Signalprozessoren von Texas Instruments)

CENELEC – Comité Européen de Normalisation Electrotechnique (in der Eisenbahndomäne werden die Normen [DIN99], [DIN01] und [DIN03] häufig auch zusammenfassend als „CENELEC-Normen“ bezeichnet)

COM – Component Object Model

CTE – Classification Tree Editor, Werkzeug zur Testfallentwicklung mit Klassifikationsbäumen

DB – Deutsche Bahn

DFD – Datenflussdiagramm, auch engl.: Data Flow Diagram

DIN – Deutsche Industrienorm

DLR – Deutsches Zentrum für Luft- und Raumfahrt e. V.

DMI – Driver Machine Interface

DSL – Domain Specific Language (Domänenspezifische Sprache)

DSP – Digitaler Signalprozessor

EA – Endlicher Automat

EBA – Eisenbahnbundesamt

EBG – Eisenbahngesetz (Schweiz)

EBV – Eisenbahnverordnung (Schweiz)

EBO – Eisenbahnbetriebsordnung

EMV – Elektromagnetische Verträglichkeit

EN – Europäische Norm

ESO – Eisenbahnsignalordnung

ESTW – Elektronisches Stellwerk

ETSI – European Telecommunication Standards Institute

EU – Europäische Union

EVC – European Vital Computer (ETCS Fahrzeugrechner)

FAT – Factory Acceptance Test

(FF)FIS – (Form Fit) Functional Interface Specification (Schnittstellenspezifikation, evtl. einschließlich physikalischer / mechanischer Aspekte)

FRS – Functional Requirements Specification (Funktionale Anforderungsspezifikation)

FSM – Finite State Machine (Endlicher Zustandsautomat)

FÜM – Festlegeüberwachungsmelder (Teil der Fahrstraßenausleuchtung in einem ESTW)

HiL – Hardware-in-the-Loop

INDUSI – Induktive Zugsicherung (Standardzugsicherungssystem in Deutschland)

ITU – International Telecommunication Union

JRU – Juridical Recoder Unit

KoRil – Konzernrichtlinie (der DB AG)

KS – Kombinationssignal

LST – Leit- und Sicherungstechnik

LZB – Linienförmige Zugbeeinflussung

MDA – Model Driven Architecture

MDD – Model Driven Development (Modelbasierte Entwicklung)

MFA – Modulare Führerraumanzeige (Anzeigeeinheit auf Loks)

MiL – Model-in-the-Loop

MOF – MetaObject-Facility

MOSES – Methodology for Object-Oriented Software Engineering of Systems

MSC – Message Sequence Charts

MTC – Model Test Coverage, eine Test- und Verifizierungskomponente von SCADE

OBU – Onboard Unit (Teil des ETCS)

OOA – Object Oriented Analysis

OOD – Object Oriented Design

OCL – Object Constraint Language, Teil der UML-Spezifikation

OLE – Object Linking and Embedding

OMG – Object Management Group, Herausgeber der Spezifikationen für UML, SysML und U2TP

OML – Object Modeling Language

OMT – Object Modeling Technique (ein UML-Vorläufer)

OPEN – Object-Oriented Process, Environment and Notation

OOSE – Object-Oriented Software Engineering (ein UML-Vorläufer)

PBerfG – Personenbeförderungsgesetz

PDU – Protocol Data Unit

PIM – Platform Independent Model, Teil des MDA-Ansatzes

PSM – Platform Specific Model, Teil des MDA-Ansatzes

PZB – Punktförmige Zugbeeinflussung

RAD – Rapid Application Development

RailSiTe – Rail Simulation and Testing

RAMS(S) – Reliability, Availability, Maintainability, Safety, Security

RBC – Radio Block Center (dt.: Funkstreckenzentrale), ETCS-Komponente

RDD – Responsibility Driven Design

RE – Requirements Engineering

RM – Requirements Management

RP – Rapid Prototyping

SA – Strukturierte Analyse (nach Tom DeMarco)

SDL – Specification and Description Language

Sifa – Sicherheitsfahrschaltung (Totmannschaltung)

SiL – Software-in-the-Loop

SIL – Safety Integrity Level

SOMA – Semantic Object Modeling Approach

SRS – System Requirements Specification (Systemanforderungsspezifikation)

SSV – Signalisationsverordnung (Schweiz)

STEP-X – Strukturierter Entwicklungsprozess am Beispiel X-by-Wire

STM – Specific Transmission Module (Teil des ETCS)

SUT – System Under Test (Prüfling)

SysML – Systems Modelig Language

TC – Test Case (Testfall)

TS – Test Sequenz / Test Sequence

TTCN-3 – Testing and Test Control Notation, Version 3

U2TP – UML 2 Testing Profile

UML – Unified Modeling Language

VAL – Véhicule Automatique Léger

XMI – XML Metadata Interchange (XML-basiertes Austauschformat für UML-Modelle)

XML – eXtended Markup Language

XP – eXtreme Programming (agile Softwareentwicklungsmethode)

ZFM – Zielfestlegemelder (Teil der Fahrstraßenausleuchtung in einem ESTW)

ZSG – Zugsteuergerät

B Literaturverzeichnis

[Abe90] ABEL, DIRK: *Petrinetze für Ingenieure: Modellbildung und Analyse diskret gesteuerter Systeme*. Doktorarbeit, Rheinisch-Westfälische Technische Hochschule Aachen, 1990.

[Abr96] ABRIAL, JEAN-RAYMOND: *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[Art06] ARTISAN SOFTWARE: *ARTiSAN Studio Automation Interface, User's Guide, Version 6.1*, 2006.

[Bac59] BACKUS, JOHN W.: *The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference.* In: *IFIP Congress*, Seiten 125–131, 1959.

[Bal01] BALZERT, HELMUT: *Lehrbuch der Software-Technik, Band 1: Software-Entwicklung*. Spektrum Akademischer Verlag, Heidelberg, Berlin, Oxford, 2. Auflage, 2001.

[Bau02] BAUR, KARL GERHARD: *Die Baureihen 145, 146 und 185*. EK-Verlag, Freiburg, 2002.

[Bec99] BECK, KENT: *Embracing Change with Extreme Programming.* IEEE Computer, 32(10):70–77, 1999.

[Bec00] BECK, KENT: *Extreme Programming: Das Manifest*. Addison-Wesly, 2000.

[Bec02] BECK, KENT: *Test Driven Development By Example*. Addison-Wesley, 2002.

[Ber00] BERGLEHNER, RANDOLF: *Anforderungsspezifikation für die Hp-Schnittstelle*. In: SCHNIEDER, E. (Herausgeber): *Fortschritt-Berichte VDI: FORMS 2000 - Formale Techniken für die Eisenbahnsicherung*, Nummer 441 in *Reihe 12*, Seiten 103 – 123, Düsseldorf, 2000. VDI Verlag GmbH.

[Bla04] BLANCHARD, BENJAMIN S.: *System Engineering Management*. John Wiley & Sons Inc., 3. Auflage, 2004.

[Boe88] BOEHM, BARRY W.: *A Spiral Model of Software Development and Enhancement.* IEEE Computer, 21(5):61–72, 1988.

[Bowa] BOWEN, JONATHAN: *The B Method*. http://vl.fmnet.info/b, Stand: 2007-01-29.

[Bowb] BOWEN, JONATHAN: *Formal Methods*. http://vl.fmnet.info, Stand: 2007-01-29.

[Bowc] BOWEN, JONATHAN: *The Z Notation*. http://vl.zuser.org, Stand 2007-01-29.

[Box97] BOX, DON: *Essential COM*. Addison-Wesley Professional, 1. Auflage, 1997.

[Bun06] BUNDESAMT FÜR VERKEHR: *Online-Bibliothek des Bundesamtes für Verkehr*. http://www.bav.admin.ch/dokumentation/, 2006. Stand: 17.07.2006.

[CCM+99] CASAZZA, ANTONIO, DARIO COMINI, ANGELO MORZENTI, MATTEO PRADELLA, PIERLUIGI SAN PIETRO und FABIO SCHREIBER: *Specification and Test Case Generation for the Safety Kernel of the Naples Subway*. In: LARSEN, P. G. und J. S. PETERSEN (Herausgeber): *Proceedings of the FMERail Nr. 3*, www.ifad.dk/Projects/fmerail/fmerail3.htm, Tagungsort: St. Pölten, 1999. IFAD.

[CGP00] CLARKE, EDMUND M., ORNA GRUMBERG und DORON A. PELED: *Model Checking*. MIT Press, Cambridge, Massachusetts, 2000.

[Coc01] COCKBURN, ALISTAIR: *Writing Effective Use Cases*. Addison-Wesley Longman, Amsterdam, 2001.

[CPHP87] CASPI, PAUL, DANIEL PILAUD, NICOLAS HALBWACHS und JOHN PLAICE: *LUSTRE: a declarative language for programming synchronous systems*. In: *14th ACM Symposium on Principles of Programming Languages, POPL '87*, 1987.

[DeM79] DEMARCO, TOM: *Structured Analysis and System Specification*. Prentice Hall, Englewood Cliffs, New Jersey, 1979.

[DHM98] DRÖSCHEL, WOLFGANG, WALTER HEUSER und RAINER MIDDERHOFF (Herausgeber): *Inkrementele und objektorientierte Vorgehensweisen mit dem V-Modell 97*. Oldenbourg Verlag, München, 1998.

[DIN87] DIN 69901:1987: *Projektwirtschaft; Projektmanagement; Begriffe*. Beuth Verlag GmbH, 1987.

[DIN97] DIN 69905:1997-05: *Projektwirtschaft - Projektabwicklung - Begriffe*. Beuth Verlag GmbH, 1997.

[DIN99] DIN EN 50126:1999: *Bahnanwendungen - Spezifikation und Nachweis der Zuverlässigkeit, Verfügbarkeit, Instandhaltbarkeit, Sicherheit (RAMS); Deutsche Fassung EN 50126:1999*. Beuth Verlag GmbH, 1999.

[DIN01] DIN EN 50128:2001: *Bahnanwendungen - Telekommunikationstechnik, Signaltechnik und Datenverarbeitungssysteme - Software für Eisenbahnsteuerungs- und Überwachungssysteme; Deutsche Fassung EN 50128:2001*. Beuth Verlag GmbH, 2001.

[DIN03] DIN EN 50129:2003: *Bahnanwendungen – Telekommunikationstechnik, Signaltechnik und Datenverarbeitungssysteme – Sicherheitsrelevante elektronische Systeme für Signaltechnik; Deutsche Fassung EN 50129:2003*. Beuth Verlag GmbH, 2003.

[DIN05] DIN EN ISO 9000: *Qualitätsmanagementsysteme – Grundlagen und Begriffe (ISO 9000:2005); Dreisprachige Fassung EN ISO 9000:2005*. Beuth Verlag GmbH, Berlin, 2005.

[Ebn04] EBNER, MICHAEL: *UML-based Test Specification for Communication Systems: A Methodology for the use of MSC and IDL in Testing*. Doktorarbeit, Georg-August-Universität Göttingen, 2004.

[Ebn05] EBNER, MICHAEL: *TTCN-3 Test Case Generation from Message Sequence Charts*. Technischer Bericht IFI-TB-2005-02, Institut für Informatik, Georg-August-Universität Göttingen, Germany, 2005. Presented at ISSRE04 Workshop on Integrated-reliability with Telecommunications and UML Languages (ISSRE04:WITUL), IRISA, Rennes, France, 2. November 2004.

[ED07] ESSAMÉ, DIDIER und DANIEL DOLLÉ: *B in Large-Scale Projects: The Canarsie Line CBTC Experience*. In: JULLIAND, JACQUES und OLGA KOUCHNARENKO [JK06], Seiten 252–254.

[Ein03] EINER, STEFAN: *Petrinetzbasierte Spezifikation und Analyse operationaler Prozesse am Beispiel der Eisenbahnsicherung*. In: *Fortschritt-Berichte*, Nummer 373 in *20*. VDI-Verlag, Düsseldorf, 2003.

[EMzHL07] EBRECHT, LARS, MICHAEL MEYER ZU HÖRSTE und KARSTEN LEMMER: *The Basic Concept for the Formal Test Description - Horizontal Composition and Vertical Differentiation of the Atomic Element*. In: SCHNIEDER, ECKEHARD; TARNAI, GÉZA (Herausgeber): *FORMS/FORMAT 2007. 6th Symposium. Formal Methods for Automation and Safety in Railway and Automotive Systems*, Seiten 447 – 457. GZVB, 2007.

[ENL04] EN-NOUAARY, ABDESLAM und GANG LIU: *Timed Test Cases Generation Based on MSC-2000 Test Purposes*. Technischer Bericht, Department of Electrical and Computer Engineering, Concordia University, Montréal, Québec, Canada, 2004. Vorgestellt beim ISSRE04 Workshop on Integrated-reliability with Telecommunications and UML Languages (ISSRE04:WITUL), IRISA, Rennes, Frankreich, 2. November 2004.

[ESB05] EBRECHT, LARS, MARKUS SCHACHER und CHRISTIAN BÜHLER: *Test Specification in XML the most important Element for Test Automation.* In: *ARTiSAN Benutzerforum D.A.CH, Lindau, 7.-8.4.2005*, 2005.

[ESS94] ERDMANN, L., A. G. SCHIELKE und E. SCHNIEDER: *Referenzmodell zur Strukturierung von Leitsystemen*. AT - Automatisierungstechnik, 94(5):187 – 197, 1994.

[ETS05] ETSI ES 201 873-1: *The Testing and Test Control Notation Version 3*. 2005. Version 3.1.1.

[Eur02a] EUROPÄISCHE KOMMISSION: *TSI Fahrzeuge*. Amtsblatt der Europäischen Union, (245):402–506, 2002. Entscheidung Nr. 2002/735/EG.

[Eur02b] EUROPÄISCHE KOMMISSION: *TSI Infrastruktur*. Amtsblatt der Europäischen Union, (245):143–279, 2002. Entscheidung Nr. 2002/732/EG.

[Eur06] EUROPÄISCHE KOMMISSION: *TSI Zugsteuerung/Zugsicherung und Signalgebung*. Amtsblatt der Europäischen Union, (342):1–165, 2006. Entscheidung Nr. 2006/860/EG.

[EV03] EISEN, JENS und EGBERT VOIGT: *XML-basierte Prüfung der RBC-Streckenprojektierung*. Signal + Draht, 95(5):18 – 22, 2003.

[EzH04] EBRECHT, LARS und MICHAEL MEYER ZU HÖRSTE: *Formal Test Description – the central element for the automation of the process of testing*. In: SCHNIEDER, ECKEHARDT und GÉZA TARNAI (Herausgeber): *FORMS/FORMAT 2004 – Formal Methods for Automation and Safety in Railway and Automotive Systems / Proceedings of Symposium FORMS/FORMAT 2004*, Braunschweig, 2004. Technische Universität Braunschweig.

[Fay99] FAY, ALEXANDER: *Wissensbasierte Entscheidungsunterstützung für die Disposition im Schienenverkehr*. In: *Forschritt-Berichte*, Nummer 386 in *12*. VDI-Verlag, Düsseldorf, 1999.

[FBH05] FELDBRÜGGE, RAINER und BARBARA BRECHT-HADRASCHEK: *Prozessmanagement leicht gemacht*. Redline Wirtschaftsverlag, 1. Auflage, 2005.

[FHSG97] FIRESMITH, DONALD, BRIAN HENDERSON-SELLERS und IAN GRAHAM: *OPEN Modeling Language (OML) reference manual*. Cambridge Press / Sigs Books, New York, 1997.

[Fow05] FOWLER, MARTIN: *Refactoring – Oder: Wie Sie das Design vorhandener Software verbessern*. Addison-Wesley, 2005.

[GG93a] GRIMM, KLAUS und MATTHIAS GROCHTMANN: *Systematischer Software-Test mit der Klassifikationsbaum-Methode.* In: *GI Jahrestagung*, Seiten 253–259, 1993.

[GG93b] GROCHTMANN, MATTHIAS und KLAUS GRIMM: *Classification Trees for Partition Testing.* Softw. Test., Verif. Reliab., 3(2):63–82, 1993.

[GKJL06] GANTZ, OLIVER, VOLKER KNOLLMANN, KLAUS P. JASCHKE und KARSTEN LEMMER: *Visualisierung im Bahnlabor RailSiTe.* In: EDIS, ZILINSKA UNIVERZITA V ZILINE V (Herausgeber): *EURNEX - Zel 2006, 14th International Symposium*, Band 3, Seiten 162 – 168. EDIS, 2006.

[Gra00] GRAHAM, IAN: *Object-oriented Methods: Principles and Practice*. Addison Wesley, 3. Auflage, 2000.

[Göh98] GÖHNER, PETER: *Komponentenbasierte Entwicklung von Automatisierungssystemen*. In: *GMA-Kongreß ´98: Mess- und Automatisierungstechnik – Neue Entwicklungen, Technologie, Anwendungen*, Nummer 1397 in *VDI-Berichte*. VDI-Verlag, 1998.

[HCRP91] HALBWACHS, NICOLAS, PAUL CASPI, PASCAL RAYMOND und DANIEL PILAUD: *The synchronous dataflow programming language LUSTRE*. In: *Proceedings of the IEEE*, Band 79, Seiten 1305–1320, 1991.

[Hed03] HEDTSTÜCK, ULRICH: *Einführung in die theoretische Informatik : formale Sprachen und Automatentheorie*. Oldenbourg, München, 2. Auflage, 2003.

[HH03a] HARMS, MIRKO und MARC HORSTMANN: *Interlacing of Diagnosis and Test with Development of Embedded Systems*. In: *Software Intensive Embedded Systems – with Special Emphasis on Automotive*, 2003. Seminar.

[HH03b] HARMS, MIRKO und MARC HORSTMANN: *Strukturierter Entwicklungsprozess für eingebettete Systeme im Automobilbereich*. Automotive Electronics, 01, 2003.

[Hor05] HORSTMANN, MARC: *Verflechtung von Test und Entwurf für eine verlässliche Entwicklung eingebetteter Systeme im Automobilbereich*. Doktorarbeit, Technische Universität Braunschweig, 2005.

[HSE93] HENDERSON-SELLERS, BRIAN und JULIAN M. EDWARDS: *MOSES: an overview*. In: *TOOLS '93: Proceedings of the eleventh international conference on Technology of object-oriented languages and systems*, Seiten 561–571, Upper Saddle River, New Jersy, 1993. Prentice Hall.

[HSE94] HENDERSON-SELLERS, BRIAN und JULIAN M. EDWARDS: *BOOKTWO of object-oriented knowledge: the working object: object-oriented software engineering: methods and management*. Prentice Hall, Upper Saddle River, New Jersey, 1994.

[HSFG97] HENDERSON-SELLERS, BRIAN, DONALD FIRESMITH und IAN M. GRAHAM: *Methods Unification: The OPEN Methodology.* Journal of Object-Oriented Programming (JOOP), 10(2):41–43, 55, 1997.

[HVFR05] HARTMANN, JEAN, MARLON VIEIRA, HERBERT FOSTER und AXEL RUDER: *A UML-based approach to system testing*. Innovations in Systems and Software Engineering, 1(1/April 2005):12–24, 2005.

[HW03] HOGREFE, DIETER und ANTHONY WILES (Herausgeber): *Testing of Communicating Systems (TestCom), 15th IFIP International Conference*, Band 2644 der Reihe *Lecture Notes in Computer Science*. Springer, 2003.

[Hür01] HÜRLIMANN, DANIEL: *Objektorientierte Modellierung von Infrastrukturelementen und Betriebsvorgängen im Eisenbahnwesen*. Doktorarbeit, ETH Zürich, Zürich, 2001. Diss. ETH Nr. 14281.

[IEC00] IEC 61508: *Functional safety of electrical/electronic/programmable electronic safety-related systems*. IEC, 2000. Dt. Fassung: DIN EN 61508 u. VDE 0803.

[ISO95] ISO/IEC 12207:1995: *Information technology – Software life cycle processes*. 1995.

[ISO02] ISO/IEC 13568:2002: *Information technology – Z formal specification notation – Syntax, type system and semantics*. ISO, 2002.

[ITU02] ITU: *ITU-T Z.100: Specification and Description Language (SDL)*. ITU, Genf, Schweiz, 2002. http://www.itu.int/ITU-T/studygroups/com17/languages/Z100.pdf, Stand: 2007-01-29.

[ITU04] ITU: *ITU-T Z.120: Messages Sequence Charts (MSC)*. ITU, Genf, Schweiz, 2004. http://www.itu.int/ITU-T/studygroups/com17/languages/Z120.pdf, Stand: 2007-01-29.

[Jan07] JANKE, SEBASTIAN: *Entwicklung einer Testumgebung für die Testfallbeschreibungssprache TTCN-3 am Beispiel eines digitalen Signalprozessors*. Diplomarbeit, Technische Universität Braunschweig, Braunschweig, 2007.

[JCJ92] JACOBSON, IVAR, MAGNUS CHRISTERSON und PATRIK JONSSON: *Object- Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley, Amsterdam, 1992.

[Jen97] JENSEN, KURT: *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use, Volume 2: Analysis Methods*. Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, 2. Auflage, 1997.

[JK06] JULLIAND, JACQUES und OLGA KOUCHNARENKO (Herausgeber): *B 2007: Formal Specification and Development in B, 7th International Conference of B Users, Besançon, France, January 17-19, 2007, Proceedings*, Band 4355 der Reihe *Lecture Notes in Computer Science*. Springer, 2006.

[JS01] JANSEN, LARS und ECKEHARD SCHNIEDER: *Begriffsmodelle der Automatisierungstechnik - Basis effizienten Enginerings*. In: SCHNIEDER, E. (Herausgeber): *Engineering komplexer Automatisierungssysteme*, Seiten 1 – 27, Braunschweig, 2001. Institut für Regelungs- und Automatisierungstechnik der Technischen Universität Braunschweig, Braunschweig.

[Kec06] KECHER, CHRISTOPH: *UML 2.0 : Das umfassende Handbuch*. Galileo Press, Bonn, 2. Auflage, 2006.

[KGL07] KNOLLMANN, VOLKER, OLIVER GANTZ und KARSTEN LEMMER: *Untersuchung von ETCS-Betriebsszenarien durch Co-Simulation von BEST und RailSiTe*. In: *Best User's Conference 2007*. Funkwerk IT, 2007.

[KL06] KNOLLMANN, VOLKER und KARSTEN LEMMER: *Graphische Testfallbeschreibung mit UML – Expertenwissen in Bildern?* In: *VDE-Kongress 2006 – Innovations for Europe*. VDE-Verlag, 2006.

[KL07a] KNOLLMANN, VOLKER und KARSTEN LEMMER: *Combined UML-based System- and Test Models as Backbone for the Development Process*. In: *6th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT)*, 2007.

[KL07b] KNOLLMANN, VOLKER und KARSTEN LEMMER: *Durchgängige Anforderungs-, System- und Testfallbeschreibung mit UML*. In: *41. Regelungstechnisches Kolloquium Boppard*, 2007. http://www.iitb.fraunhofer.de/servlet/is/9951/Knollmann_Boppard07.pdf, Stand: 2007-04-25.

[Kla79] KLAUS, GEORG (Herausgeber): *Wörterbuch der Kybernetik*, Band 1. Fischer, 1979.

[KOJL05] KNOLLMANN, VOLKER, MIROSLAV OBRENOVIC, BÄRBEL JÄGER und KARSTEN LEMMER: *Optimized Migration Process in Rail Traffic Using Simulation Support*. In: *1st International Seminar on Railway Operations Modelling and Analysis, Delft (Niederlande), 8.-10.06.2005*, 2005.

[Koo97] KOORDINIERUNGS- UND BERATUNGSSTELLE DER BUNDESREGIERUNG FÜR INFORMATIONSTECHNIK IN DER BUNDESVERWALTUNG IM BUNDESMINISTERIUM DES INNERN (KBST): *V-Modell 97*. 1997. http://www.v-modell.iabg.de, Stand: 2007-05-25.

[Koo05] KOORDINIERUNGS- UND BERATUNGSSTELLE DER BUNDESREGIERUNG FÜR INFORMATIONSTECHNIK IN DER BUNDESVERWALTUNG IM BUNDESMINISTERIUM DES INNERN (KBST): *V-Modell XT*. 2005. Version 1.2.0, http://www.v-modell-xt.de, Stand: 2007-05-25.

[Lig90] LIGGESMEYER, PETER: *Modultest und Modulverifikation: State of the Art*, Band 4 der Reihe *Angewandte Informatik*. BI-Wissenschaftsverlag, Mannheim, Wien, Zürich, 1990.

[Lig02] LIGGESMEYER, PETER: *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, Heidelberg, 2002.

[LN04] LAPLANTE, PHILLIP A. und COLIN J. NEILL: *Opinion: The Demise of the Waterfall Model Is Imminent.* ACM Queue, 1(10):10–15, 2004.

[LW00] LEHMANN, ECKARD und JOACHIM WEGENER: *Test Case Design by Means of the CTE XL*. In: *Proceedings of the 8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000)*, Kopenhagen, Dänemark, 2000.

[May05] MAYDL, WALTER: *Komponentenbasierte Softwareentwicklung für datenflußorientierte eingebettete Systeme*. Doktorarbeit, Universität Passau, 2005.

[ME99] MÜLLER-ETTRICH, GUNTER: *Objektorientierte Prozeßmodelle: UML einsetzen mit OOTC, V-Modell, Objectory*. Addison-Wesly-Longman, Bonn, 1. Auflage, 1999.

[Mey04] MEYER ZU HÖRSTE, MICHAEL: *Methodische Analyse und generische Modellierung von Eisenbahnleit- und -sicherungssystemen*. In: SCHNIEDER, ECKEHARD (Herausgeber): *Fortschritt-Berichte VDI*, Nummer 571 in *Reihe 12*. VDI Verlag GmbH, 2004.

[Mey05] MEYER ZU HÖRSTE, MICHAEL: *Defining engineering design requirements to address operational scenarios for ERTMS infrastructure systems*. In: IQ, TRANSPORT (Herausgeber): *EUROPEAN SIGNALLING 2005*, 2005.

[MP03] MUTZ, MARTIN und STEFFEN PIETSCH: *Verbesserung des modellbasierten Softwareentwurfs im Automobilbereich durch einen Design-Regel-Checker*. In: *Elektronik im Automobil*, Tagung im Haus der Technik, Stuttgart, 2003.

[MSBT04] MYERS, GLENFORD J., COREY SANDLER, TOM BADGETT und TODD M. THOMAS: *The Art of Software Testing*. John Wiley & Sons, 2. Auflage, 2004.

[MSS00] MEYER ZU HÖRSTE, MICHAEL, ECKEHARD SCHNIEDER und HANS-MARTIN SCHULZ: *Test-case Generation from Petri-net Specifications of a Train Control System*. In: MELLITT, B., R. J. HILL, J. ALLAN, G. SCIUTTO und C. A. BREBBIA (Herausgeber): *Computers in Railways, COMPRAIL VII*, Seiten 117 – 126, Boston, Southampton, 2000. WIT Press. Tagungsort: Bologna (I).

[Mut03] MUTZ, MARTIN: *Ein Regel-Checker zur Verbesserung des modellbasierten Softwareentwurfs*. In: *Entwurf komplexer Automatisierungssysteme (EKA)*, Seiten 411–423, Braunschweig, 2003. Toolpräsentation.

[NL03] NEILL, COLIN J. und PHILLIP A. LAPLANTE: *Requirements Engineering: The State of the Practice.* IEEE Software, 20(6):40–45, 2003.

[OB88] OSTRAND, THOMAS J. und MARC J. BALCER: *The Category-Partition Method for Specifying and Generating Functional Tests.* Commun. ACM, 31(6):676–686, 1988.

[Obj05a] OBJECT MANAGEMENT GROUP: *MOF 2.0/XMI Mapping Specification, V2.1.* 2005. http://www.omg.org/cgi-bin/doc?formal/2005-09-01, Stand: 2006-08-01.

[Obj05b] OBJECT MANAGEMENT GROUP: *UML Testing Profile*. 2005. Version 1.0. http://www.omg.org/cgi-bin/apps/doc?formal/05-07-07.pdf, Stand: 2006-08-01.

[Obj05c] OBJECT MANAGEMENT GROUP: *Unified Modeling Language: Superstructure*. 2005. http://www.omg.org/cgi-bin/doc?formal/2005-07-04.

[Obj06a] OBJECT MANAGEMENT GROUP: *OMG Systems Modeling Language (OMG SysML) Specification*. 2006. Final Adopted Specification, http://www.omg.org/cgi-bin/apps/doc?ptc/06-05-04.pdf, Stand: 2006-08-01.

[Obj06b] OBJECT MANAGEMENT GROUP: *Unified Modeling Language: Diagram Interchage, Version 1.0*. 2006. http://www.omg.org/cgi-bin/doc?formal/2006-04-04.

[Obj06c] OBJECT MANAGEMENT GROUP: *Unified Modeling Language: Object Constraint Language, Version 2.0*. 2006. http://www.omg.org/cgi-bin/doc?formal/2006-05-01.

[Obj07a] OBJECT MANAGEMENT GROUP: *Unified Modeling Language: Infrastructure, Version 2.1.1*. 2007. http://www.omg.org/cgi-bin/doc?formal/2007-02-04.

[Obj07b] OBJECT MANAGEMENT GROUP: *Unified Modeling Language: Superstructure, Version 2.1.1*. 2007. http://www.omg.org/cgi-bin/doc?formal/2007-02-03.

[OKJL05] OBRENOVIC, MIROSLAV, VOLKER KNOLLMANN, BÄRBEL JÄGER und KARSTEN LEMMER: *Methodik für die Entwicklung, Bewertung und simulative Validierung von Migrationsstrategien*. In: EDIS, ZILINSKÁ UNIVERZITA V ZILINE V (Herausgeber): *ZEL 2005, Zilina (Slowakische Republik), 24.-25.05.2005*, Band 12, Seiten 194 – 201. EDIS Zilina, 2005.

[Pac00] PACHL, JÖRN: *Systemtechnik des Schienenverkehrs*. B. G. Teubner, Stuttgart, Leipzig, 2 Auflage, 2000.

[Pet62] PETRI, CARL ADAM: *Kommunikation mit Automaten*. Doktorarbeit, Universität Bonn, Institut für Instrumentelle Mathematik, Bonn, 1962.

[RBP+91] RUMBAUGH, JAMES, MICHAEL BLAHA, WILLIAM PREMERLANI, FREDERICK EDDY und WILLIAM LORENSEN: *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[RG92] RUBIN, KENNETH S. und ADELE GOLDBERG: *Object Behavior Analysis*. Communications of the Association for Computing Machinery (ACM), 35(9):48–62, 1992.

[RHQ+05] RUPP, CHRIS, JÜRGEN HAHN, STEFAN QUEINS, MARIO JECKLE und BARBARA ZENGLER: *UML 2 glasklar*. Hanser, München, 2005.

[Roy87] ROYCE, WINSTON W.: *Managing the Development of Large Software Systems: Concepts and Techniques.* In: *ICSE*, Seiten 328–339, 1987. Nachdruck. Originalartikel erschienen in IEEE WESCON, August 1970, S. 1 – 9.

[RS04] RUPP, CHRIS und SOPHIST GROUP: *Requirements-Engineering und -Management*. Hanser, 3 Auflage, 2004.

[Rum05] RUMPE, BERNHARD: *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring*. Springer Verlag, 2005.

[SA92] SCHNIEDER, ECKEHARD und DIRK ABEL: *Petrinetze in der Automatisierungstechnik*. Oldenbourg, 1992.

[Sch92] SCHNIEDER, ECKEHARD: *Prozeßinformatik*. Vieweg, Braunschweig, 2. Auflage, 1992.

[Sch99] SCHNIEDER, ECKEHARD: *Methoden der Automatisierung*. Vieweg, Braunschweig, 1999.

[Sch02] SCHNIEDER, ECKEHARD: *Integration heterogener Modellwelten der Automatisierungstechnik*. In: NAGL, M. und B. WESTFECHTEL (Herausgeber): *Workshop vom 20. - 22. März 2002: Modelle, Werkzeuge und Infrastrukturen zur Unterstützung von Entwicklungsprozessen*, Seiten 23 – 41, Aachen, 2002. Deutsche Forschungsgemeinschaft (DFG), Wiley VCH.

[SD04] SCHIEBER, REINHARD und FRANK DERICHSWEILER: *Testautomatisierung in der Automobilindustrie*. OBJEKTspektrum, 4:63–67, 2004.

[SDGR03] SCHIEFERDECKER, INA, ZHEN RU DAI, JENS GRABOWSKI und AXEL RENNOCH: *The UML 2.0 Testing Profile and its Relation to TTCN-3*. In: HOGREFE, DIETER und ANTHONY WILES [HW03], Seiten 79–94.

[SG03] SCHIEFERDECKER, INA und JENS GRABOWSKI: *The Graphical Format of TTCN-3 in the Context of MSC and UML*. In: SHERRATT, EDEL (Herausgeber): *Telecommunications and beyond: The Broader Applicability of SDL and MSC, Third International Workshop, SAM 2002, Aberystwyth, UK, June 24-26, 2002. Revised Papers*, Band 2599 der Reihe *Lecture Notes in Computer Science*, Seiten 233–252. Springer, 2003.

[SJMS97] SCHULZ, HANS-MARTIN, LARS JANSEN, MICHAEL MEYER ZU HÖRSTE und ECKEHARD SCHNIEDER: *Studie: Methodische Erstellung von Testfällen auf der Basis formaler Spezifikationen*. Technischer Bericht, Institut für Regelungs- und Automatisierungstechnik, Technische Universität Braunschweig, 1997.

[SMPS98] SCHULZ, HANS-MARTIN, MICHAEL MEYER ZU HÖRSTE, BERNHARD PTOK und ECKEHARD SCHNIEDER: *Modelling and Simulation of the new ERTMS/ETCS for Test Case Generation*. In: MELLITT, B., R. J. HILL, J. ALLAN, G. SCIUTTO und C. A. BREBBIA (Herausgeber): *Computers in Railways Comprail VI*, Seiten 649 – 658, Boston, Southampton, 1998. WIT Press. Tagungsort: Lissabon (Pt).

[SS99] SCHOLZE-STUBENRECHT, WERNER (Herausgeber): *Das große Wörterbuch der deutschen Sprache*. Dudenverlag, 3. Auflage, 1999. In zehn Bänden.

[ST03] SÁGHI, BALÁZS und GÉZA TARNAI: *Introduction of formal methods in railway signalling*. In: SKYVA, L. (Herausgeber): *Zel 2003*, Band 1, Seiten 151 – 157, Zilina (SK), 2003. Universität Zilina. Band 1, Tagungsort Zilina (SK).

[ST07] SCHNIEDER, ECKEHARD und GÉZA TARNAI (Herausgeber): *FORMS/FORMAT 2007 - Formal Methods for Automation and Safety in Railway and Automotive Systems, Proceedings of Symposium, 25th and 26th January 2007*. GZVB e. V., Braunschweig, 2007.

[Sta73] STACHOWIAK, HERBERT: *Allgemeine Modelltheorie*. Springer, Wien, 1973.

[Stö05] STÖGER, ROMAN: *Geschäftsprozesse erarbeiten - gestalten - nutzen*. Schäffer-Poeschel, 2005.

[TE04] TROG, CARSTEN und LARS-HENRIK ERIKSSON: *Spezifikation von Stellwerkslogik mit formalen Methoden*. Signal + Draht, Seiten 18 – 21, 2004.

[Tur36] TURING, ALAN: *On Computable Numbers, With an Application to the Entscheidungsproblem*. In: *Proceedings of the London Mathematical Society*, Band 42 der Reihe 2, 1936.

[Uht06] UHTENWOLDT, JAN: *Analyse des PEACC+-Prozesses hinsichtlich der Anwendbarkeit modellbasierter Entwicklungsmethoden*. Projektarbeit, TU Braunschweig, 2006.

[UNI05a] UNISIG: *Subset 076 – Test Cases Related to Features (Version 2.2.2), Subset 076 – Test Sequences (Version 2.0.0)*. European Railway Agency (ERA), Brüssel, 2005. http://www.era.eu.int, Stand: 2007-09-06.

[UNI05b] UNISIG: *Subset-094: Functional Requirements for an On-board Reference Test Facility*. European Railway Agency (ERA), Brüssel, 2005. Version 2.0.0, http://www.era.eu.int, Stand: 2007-09-06.

[VDI91] VDI/VDE 3694: *Lastenheft/Pflichtenheft für den Einsatz von Automatisierungssystemen*. Beuth Verlag GmbH, 1991.

[Ver00] VERSTEEGEN, GERHARD (Herausgeber): *Das V-Modell in der Praxis: Grundlagen, Erfahrungen, Werkzeuge*. dpunkt-Verlag, Heidelberg, 1. Auflage, 2000.

[Vie05] VIEIRA, MARLON: *Invited Presentation II: Experiences in Applying Model Based System Testing Generation.* In: BRIAND, LIONEL C. und CLAY WILLIAMS (Herausgeber): *MoDELS*, Band 3713 der Reihe *Lecture Notes in Computer Science*, Seite 430. Springer, 2005.

[WBWW90] WIRFS-BROCK, REBECCA, BRIAN WILKERSON und LAUREN RUTH WIENER: *Designing object-oriented software*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[WDT+05] WILLCOCK, COLIN, THOMAS DEISS, STEPHAN TOBIES, STEFAN KEIL und FEDERICO ENGLER: *An Introduction to TTCN-3*. Wiley, 1. Auflage, 2005.

[WE07] WULF, ALEXANDER und CARLA EICKMANN: *Zwischenbericht zum Projekt „Betriebsführung Regionalstrecken“: Arbeitspaket 1.2 – „Funktionalitäten“*. DLR, Institut für Verkehrsführung und Fahrzeugsteuerung, bislang unveröffentlichter interner Bericht, 2007.

[Wel94] WELLER, EDWARD F.: *Using Metrics to Manage Software Projects.* IEEE Computer, 27(9):27–33, 1994.

[Wil06] WILLMS, MARTIN: *Entwicklung von Hardware- und Softwarekomponenten für ein DSP-System zur Anbindung eines Zugfahrzeugrechners an ein Bahnsimulationslabor über Eurobalisen*. Diplomarbeit, FH Braunschweig / Wolfenbüttel, 2006.

[WRL05] WOLF, HENNING, STEFAN ROOCK und MARTIN LIPPERT: *eXtreme Programming: eine Einführung mit Empfehlungen und Erfahrungen aus der Praxis*. dpunkt-Verlag, Heidelberg, 2. überarb. u. erw. Auflage, 2005.