Reduktion von Integrationsproblemen für Software im Automobil durch frühzeitige Erkennung und Vermeidung von Architekturfehlern

Herbert Reiter

Institut für Informatik
der Technischen Universität München

Reduktion von Integrationsproblemen für Software im Automobil durch frühzeitige Erkennung und Vermeidung von Architekturfehlern

Herbert Reiter

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Uwe Baumgarten
Prüfer der Dissertation:	
1.	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2.	Univ.-Prof. Dr. Jochen Ludewig, Universität Stuttgart

Die Dissertation wurde am 12.07.2010 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.11.2010 angenommen.

Kurzfassung

Ein wesentlicher Anteil der Funktionalität eines Automobils wird mittlerweile über den Einsatz von Elektrik, Elektronik und der sie steuernden Software realisiert. Vor der Integration müssen die einzelnen Komponenten umfangreiche Modultests bestanden haben. Dennoch zeigen sich im Verbund Fehler, die auf Schwächen in der Architekturbeschreibung und der Integration hindeuten.

Diese Arbeit definiert zunächst die Integrationsproblematik sowie zentrale Begriffe formal. Anschließend werden die Ergebnisse der im Rahmen dieser Arbeit durchgeführten empirischen Untersuchung bei der Robert Bosch GmbH vorgestellt. Zur praxistauglichen Erkennung und Vermeidung der Integrationsproblematik werden analytische sowie konstruktive Qualitätssicherungsmaßnahmen vorgeschlagen und bewertet.

Abstract

In recent years many key functions in automobiles have come to be performed by electrical and electronic components and by the software on which they operate. Before integration phase each component has to pass comprehensive module tests. However, interconnected components lead to errors that indicate an insufficient architecture description and integration.

This thesis offers formal definitions of key terms and of the problems connected with integration and then presents results obtained from an empirical investigation at Robert Bosch GmbH. In order to detect and avoid the integration problem in practice, analytical as well as constructive quality assurance methods are suggested and evaluated.

Danksagung

Mein herzlicher Dank gilt meinem Doktorvater Prof. Broy für die stets konstruktive Rückmeldung und zahlreiche wertvolle Anregungen und Denkanstöße sowie seinen Lehrstuhlmitarbeitern für die vielseitigen Diskussionen. Ebenso möchte ich mich bei Prof. Ludewig für die Übernahme des Zweitgutachtens und seine hilfreichen Anmerkungen bedanken.

Ganz besonders bedanke ich mich bei der Robert Bosch GmbH, die es mir ermöglicht hat, diese Dissertation im industriellen Umfeld anzufertigen, und insbesondere bei Andreas Thums für die Betreuung.

Des Weiteren bedanke ich mich bei meinen Korrekturlesern Iris Reiter, Ingeborg Rentsch und Torsten Schütze, bei letzterem auch für die vielen Tipps im Umgang mit LaTeX.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Mit fortschreitender Technologie werden an das Automobil immer höhere Anforderungen gestellt. Bis 2010 soll die Zahl der Verkehrstoten in der EU halbiert[1] werden, die CO_2-Diskussion fordert umweltschonendere Technologie und nicht zuletzt verlangt der Markt nach Innovation.

Um die ständig wachsenden Anforderungen erfüllen zu können, wird die technische Ausstattung der Automobile stetig verbessert: Das Antiblockiersystem (ABS) und das Elektronische Stabilitäts-Programm (ESP) sorgen für die größtmögliche Stabilität in schwierigen Fahrsituationen; Luftmassenmesser und Lambdasonden, welche die Zusammensetzung der Ansaug- bzw. Abgasluft messen, sorgen für eine präzise Regelung des Luft-Kraftstoff-Gemisches und erlauben so die Einhaltung der Abgasnormen; Komfortsysteme wie Adaptive Cruise Control (ACC), Klimaanlage und Multimediasysteme helfen, den Fahrer zu entlasten und den Insassen eine möglichst angenehme Fahrt zu bereiten.

Der intensive Einsatz von Elektrik und Elektronik im Automobil hat dazu geführt, dass in einem Kleinwagen mittlerweile ca. 20 Steuergeräte verbaut sind und in der Oberklasse bis zu 70. Der Mehrwert neuer Funktionalitäten liegt zunehmend im intelligenten Zusammenspiel, also der Vernetzung elektronischer Komponenten. Laut einer Studie der Boston Consulting Group [ZH04] werden 70 % der künftigen Innovationen allein durch Software realisiert. Eine Mercer-Studie [Mer06] schätzt den Wert der elektrischen und elektronischen Bauteile samt Software weltweit betrachtet bereits auf einen Anteil von 20 % am Gesamtwert eines Automobils. Bis 2015 soll dieser Anteil auf über 30 % steigen. Die stark ansteigende Zahl an Funktionalitäten im Automobil sorgt für eine zunehmende Komplexität, selbst wenn die Anzahl der Steuergeräte künftig gleichbleibend oder gar rückläufig sein sollte.

Die starke Vernetzung der Steuergeräte birgt auch Risiken. Kleine Änderungen oder lokale Fehler können sich dadurch systemweit auswirken und eigentlich unabhängige Funktionen sich gegenseitig beeinflussen (Feature Interaction). Gerade Zeitanforderungen werden hier oft zum Problem.

[1] Die Europäische Kommission hat eine Empfehlung an die EU herausgegeben, in den Jahren 2001 bis 2010 eine Halbierung der Zahl der Verkehrstoten zu erreichen. Im Jahr 2005 gab es 44900 Verkehrstote in der EU, davon 5361 in Deutschland (Quelle: Eurostat).

Das Handelsblatt [Han03] berichtete schon vor einigen Jahren, dass Fehler in der Elektronik für 55 % der Ausfälle im Automobil verantwortlich sind. Die Hauptursache wird im mangelnden Zusammenspiel der Subsysteme verschiedener Zulieferer gesehen. Zudem ist in einem undurchschaubaren Systemverbund extrem schwer festzustellen, wo der Fehler liegt, und am Einzelgerät oft kein Fehler zu entdecken. Laut Boston Consulting Group [ZH04] sind 50 % aller Fahrzeugrückrufe allein auf Softwarefehler zurückzuführen.

Bei der mittlerweile erreichten Vielzahl an Funktionalität im Automobil stoßen die bisherigen Qualitätssicherungsverfahren, die im Wesentlichen auf Reviews und intensiven Tests beruhen, an ihre Grenzen. Durch die gemeinsam genutzten Ressourcen genügt es nicht mehr, die Funktionen einzeln zu testen. Die Fehlersuche wird zusätzlich durch die Nebenläufigkeit der Subsysteme im verteilten System erschwert, da zeitlich bedingte Fehler meist nur schwer reproduzierbar sind. Ein unzureichendes Design macht die Integration der einzelnen Steuergeräte zu einem vernetzten Steuergeräteverbund zu einer zeit- und kostenintensiven Aufgabe [Bro06]. Noch verstärkt wird dies durch die im Automobilbereich auftretende Vielzahl an Funktions- und Ausstattungsvarianten.

Fehlerhaftes Zusammenspiel kann vielfältige Ursache haben und reicht von inkonsistenter Datendarstellung über unterschiedliche Zeitabstimmung bis hin zu unberücksichtigten Störeinflüssen zwischen zwei oder mehreren Subsystemen.

Bei heute eingesetzten Entwicklungsprozessen werden viele dieser Fehler erst bei der Integration des Systems entdeckt. Dabei sind die Fehler größtenteils nicht erst bei der Integration, d. h. in späten Entwicklungsphasen, entstanden. Auch die Überprüfung der einzelnen Steuergeräte gegenüber ihrer Spezifikation wird in der Regel ohne nennenswerten Fehlerbefund abgeschlossen. Dies lässt den Schluss zu, dass die Fehlerursachen schon in früheren Entwicklungsphasen zu suchen sind. Neuartige Vorgehensweisen sind notwendig, um das erfolgreiche Zusammenspiel der Komponenten von Anfang an sicherzustellen und damit die steigende Komplexität im Automobil zu beherrschen.

1.2 Eigener Lösungsansatz im Überblick

Diese Arbeit zeigt auf, welche typischen Integrationsprobleme aktuell in der Praxis vorkommen und schlägt entsprechende Abhilfemaßnahmen vor, die eine Erkennung bzw. Vermeidung der Fehlerursachen schon in frühen Entwicklungsphasen ermöglichen.

Der Aufbau der Arbeit ist zweigeteilt. Im theoretischen Teil wird die Systemintegration auf formaler Basis untersucht. Zwei Integrationsebenen stehen dabei im Vordergrund:

- Der *Steuergeräteverbund* als Integration mehrerer, nebenläufiger Steuergeräte, die über einen Datenbus kommunizieren.

- Ein *Steuergerät* als Integration von Softwaretasks, Betriebssystem und Steuergeräte-Hardware. Die Tasks sind wiederum (weitgehend) nebenläufig und kommunizieren über den gemeinsamen Arbeitsspeicher.

Im empirischen Teil der Arbeit wird der Stand der Praxis in der Entwicklung eingebetteter Systeme im Automobilbereich untersucht. Speziell werden Software-Entwicklungsprozesse und die Beschreibung von Architekturen im Hinblick auf integrationsrelevante Aspekte hin betrachtet.

Nach einer Einführung in die Thematik der Systemintegration (Kapitel 2) wird als Grundlage für die weitere Arbeit ein formales Systemmodell eingeführt (Kapitel 3). Mit dessen Hilfe lassen sich das Problem der Integration und zentrale Begriffe in dieser Arbeit formal charakterisieren sowie in wichtige Fehlerklassen unterteilen (Kapitel 4).

Eine Analyse empirischer Fehlerdaten zeigt, welche Integrationsprobleme aktuell in der Praxis auftreten und greift die zuvor eingeführte Fehlerklassifikation auf, um die große Spannweite der Fehlerursachen darzustellen (Kapitel 5).

Im Hinblick auf die Erkennung und Vermeidung von Integrationsproblemen werden im nächsten Schritt Anforderungen an den Entwicklungsprozess aufgestellt, die hinreichend für die Erstellung integrierbarer Systeme sind. Dabei wird neben der Neuentwicklung von Systemen auch die im Automobilbereich dominierende, änderungsgetriebene Entwicklung berücksichtigt, bei der wichtig ist, dass nach jeder Änderung wieder ein integrierbarer Systemzustand erreicht wird (Kapitel 6).

Zur Erfüllung dieser Prozessanforderungen in der Praxis werden ausgewählte Maßnahmen aus den Bereichen Prozesse, Spezifikationssprachen und Werkzeuge untersucht. Neben dem Nutzen, also der potenziellen Erkennung oder Vermeidung von Integrationsproblemen, ist auch der damit einhergehende Aufwand für den praxistauglichen Einsatz einer Maßnahme von entscheidender Bedeutung (Kapitel 7).

Das Vorgehen, Integrationsprobleme schon in frühen Entwicklungsphasen zu erkennen und zu vermeiden, verspricht eine hohe Produktqualität per Konstruktion. Doch nicht immer ist dieser Ansatz dem herkömmlichen Vorgehen, intensiv zu testen und gefundene Fehler zu beseitigen, überlegen. Letztlich entscheiden ökonomische Aspekte, welche Investition in Fehlervermeidungsmaßnahmen langfristig optimal ist (Kapitel 8).

2 Systemintegration im Automobil

2.1 Umfeld

Viele Faktoren sind dafür verantwortlich, dass Subsysteme später problemlos integriert werden können und das dadurch entstehende System einwandfrei funktioniert. Neben technischen Faktoren wie richtig gesetzten Bohrlöchern, niedrigem Schwingungsverhalten und passenden Schraubengewinde sind auch elektrische und elektronische Faktoren wie geringe Schwankungen der Versorgungsspannung, richtige Polung der Anschlüsse und ein passendes Kommunikationsprotokoll von entscheidender Bedeutung. Eine Schlüsselrolle spielt dabei die Software, in der immer mehr Teile der Systemfunktionalität realisiert sind. Für die Untersuchung von Fehlern bezüglich funktionaler Anforderungen in der Systemintegration sind vor allem eingebettete Systeme und die darin enthaltene Software zu betrachten. Im Hinblick auf die Vermeidung von Integrationsproblemen kommt den frühen Entwicklungsphasen eine besondere Bedeutung zu. Dieser Abschnitt erläutert das Umfeld dieser Arbeit.

2.1.1 Eingebettete Systeme und deren Zusammenspiel

Definition 2.1 (Eingebettete Systeme)
Eingebettete Systeme sind Software-/Hardwaresysteme, die als Teil eines größeren Systems für den Benutzer verborgen arbeiten, aber für die Funktions- und Leistungsfähigkeit des Gesamtsystems entscheidend sind. Ihre Aufgabe besteht darin, eine Anzahl technischer Geräte und physikalischer Prozesse der Umgebung zu überwachen und/oder gemäß einer definierten Funktionalität über Sensoren und Aktoren zu steuern und zu regeln [FGP04]. □

Die im Automobil eingesetzten eingebetteten Systeme werden auch als *Electronic Control Unit* (ECU) oder *Steuergerät* bezeichnet. Sie zeichnen sich durch folgende Eigenschaften aus (in Anlehnung an [Jaz98]):

echtzeitfähig Für die korrekte Funktionalität ist nicht nur die Ausgabe des korrekten Ergebnisses erforderlich, sondern zusätzlich muss die Reaktion innerhalb vorgegebener Zeitschranken erfolgen. Gerade bei Regelungsaufgaben wie dem Antiblockiersystem (ABS) ist diese Eigenschaft essenziell.

nebenläufig Jedes Steuergerät besitzt einen eigenen Prozessor und hat einen eigenen, internen Rechentakt. Die eingebetteten Systeme arbeiten daher nebenläufig, die Kommunikation untereinander erfolgt asynchron.

Massenprodukt Automobile werden in der Regel in hoher Stückzahl hergestellt. Daher sind geringer Produktionsaufwand und niedrige Materialkosten sehr wichtig. Für die eingebetteten Systeme bedeutet das, dass sie möglichst geringe Stückkosten haben müssen.

beschränkte Ressourcen Aufgrund möglichst niedriger Stückkosten sind die durch die Hardware bereitgestellten Ressourcen sehr beschränkt. Das bedeutet insbesondere einen kleinen Arbeitsspeicher, wenig Rechenleistung und eine geringe Bandbreite zur Kommunikation zwischen Steuergeräten.

wartungsfrei Die eingebetteten Systeme im Automobil sind auf wartungsfreien[1] Betrieb ausgelegt. Nur in Ausnahmefällen bedarf es eines manuellen Eingriffs.

Funktionalität in Software Die wesentliche Verhaltenslogik eines eingebetteten Systems ist in Software implementiert. Sie wird im Flash- oder ROM-Speicher abgelegt. Software ist leicht austauschbar und verursacht aufgrund ihres immateriellen Charakters praktisch keine Vervielfältigungskosten.

vorab festgelegte Funktionalität und Umgebung Die Funktionalität des eingebetteten Systems und die Umgebung, in die es eingebettet ist, sind bereits zur Entwurfszeit festgelegt. Nur in Ausnahmefällen wird die technische Architektur eines Fahrzeugs im Feld nachträglich verändert. Selbst der nachträglich Einbau von Zubehör ist bereits zur Entwurfszeit eingeplant. Darüber hinausgehende Eingriffe sind von Herstellerseite nicht erwünscht und können zum Erlöschen der Betriebserlaubnis und des Gewährleistungsanspruchs führen.

einfache Datenstrukturen und Kommunikationsabläufe Die Kommunikation zwischen eingebetteten Systemen erfolgt typischerweise mittels relativ einfacher Datenstrukturen. Diese bestehen aus primitiven Datentypen (z. B. uint8, int16, bool) und zusammengesetzten Datentypen. Komplexe Datenstrukturen wie Ströme oder serialisierte Objekte, die sich über mehrere Nachrichtenpakete erstrecken, treten typischerweise nur im Multimediaverbund auf. Die Kommunikation zwischen eingebetteten Systemen verwendet schlanke Protokolle, auf Verbindungsorientierung und Sessions wird verzichtet. Die Komplexität liegt v. a. in der Algorithmik der eingebetteten Systeme.

[1] Anmerkung zum Begriff *Wartung*: Die Norm VDI 2896 [VDI94] definiert Wartung als „Maßnahmen zur Bewahrung des Sollzustandes von technischen Mitteln eines Systems“. Hierbei wird davon ausgegangen, dass das System bereits vor der Wartung korrekt funktioniert hat. Im Unterschied dazu wird bei Software unter Wartung die Änderung der Software zur Korrektur von Fehlern, Funktionserweiterung oder Anpassung verstanden (ISO/IEC 12207 [ISO95]).

Die Realisierung vieler Funktionen mittels elektronischer Steuergeräte ermöglicht es, diese auf einfache Weise untereinander zu vernetzen. Dadurch wird ein Potential geschaffen, das ganz neue, bisher unbekannte Funktionalitäten erlaubt. Dies wird in der Praxis immer häufiger genutzt. Der Mehrwert neuer Funktionalitäten liegt zunehmend im intelligenten Zusammenspiel mehrerer Subsysteme. Eine Vernetzung findet dabei sowohl zwischen den Steuergeräten als auch zwischen den Softwaremodulen und Tasks innerhalb eines Steuergeräts statt.

Ein typisches Beispiel für eine Verbundfunktionalität ist Adaptive Cruise Control (ACC) – ein Tempomat mit Abstandsregelung zum vorausfahrenden Fahrzeug. Es verbindet einen Abstandssensor mit dem Bremssystem (üblicherweise angesteuert durch das Elektronische Stabilitäts-Programm (ESP)), der Motorsteuerung, der Getriebesteuerung und den Anzeige- und Bedienelementen. Die Beschleunigung des Fahrzeugs erfolgt über eine Erhöhung des Motormoments, die Verzögerung des Fahrzeugs über das Schleppmoment des Motors und bei Bedarf zusätzlich über einen Bremseingriff. Das Getriebesteuergerät gibt Aufschluss über das momentane Übersetzungsverhältnis zwischen Motor- und Raddrehzahl und damit über die Beschleunigungswirkung einer Motormomentveränderung. Abbildung 2.1 zeigt den ACC-Verbund schematisch. Weitere Informationen sind in [Bos02, Bos07] zu finden.

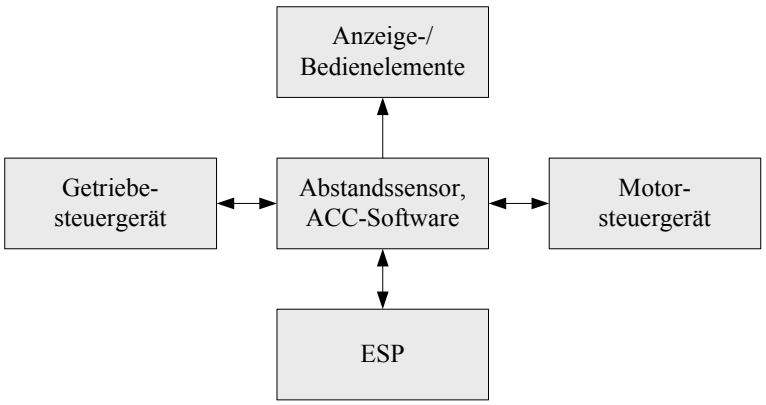


Abbildung 2.1: ACC als Verbund mehrerer Subsysteme. Die Pfeile geben den Informationsfluss bei einem aktiven ACC-System an.

Aber nicht nur Funktionen werden gemeinsam genutzt, sondern auch Ressourcen wie Rechenzeit, Arbeitsspeicher und Kommunikationsverbindungen. Heutige Bussysteme ersetzen eine Vielzahl an Verbindungskabeln und helfen so, den Kabelbaum im Automobil in Volumen und Gewicht enorm zu reduzieren.

2.1.2 Softwarebedingte Fehler

Der Großteil der Funktionalität eines Steuergeräts ist in Software implementiert. Tritt ein Fehler auf, ist die Ursache daher meistens auch in der Software zu suchen bzw. kann durch Änderung der Software behoben werden. In dieser Arbeit werden daher vorwiegend softwarebedingte Fehler untersucht, d. h.

- Fehler in einer Software-Komponente,
- Interaktionsfehler zwischen Software-Komponenten,
- Interaktionsfehler zwischen Software und Hardware sowie
- Verhaltensfehler in der Hardware, soweit sie das Verhalten des eingebetteten Systems beeinflussen.

Der Begriff des Fehlers wird in Abschnitt 2.2 genauer beschrieben.

2.1.3 Frühe Entwicklungsphasen

Um Fehler aus der Sicht des Entwicklungsprozesses zu behandeln, zu beheben oder zu vermeiden, kommen grundsätzlich folgende Möglichkeiten in Betracht.

Fehlervermeidung während der Entwicklung Während der Entwicklung werden Vorkehrungen getroffen, so dass Fehler gar nicht erst entstehen können.

Fehlerbehebung während der Entwicklung Im Entwicklungsprozess wird darauf geachtet, dass entstandene Fehler möglichst schnell gefunden und behoben werden.

Autonome Fehlerbehandlung zur Laufzeit Die zur Laufzeit unvermeidlichen physikalischen Störeinflüsse auf das System (beispielsweise elektromagnetische Einstrahlung, Erschütterung und Wärme) können auch bei einem funktional fehlerfreien System zu einem falschen Systemverhalten führen. Mittels Fehlertoleranzmechanismen (siehe [TS03, Ech90]) lassen sich solche Fehler abmildern und manchmal sogar vollständig kompensieren. Neben physikalischen Störeinflüssen können damit teilweise auch Fehler, die trotz sorgfältiger Entwicklung im Produkt vorhanden sind, behandelt werden.

Fehlerbehebung zur Laufzeit Hier wird ein System durch Eingriff von außen (z. B. durch die Werkstatt) verändert. Damit lassen sich während der Laufzeit defekt gewordene Subsysteme und auch solche, die bereits seit der Entwicklung fehlerhaft sind, durch fehlerfreie ersetzen oder reparieren.

Bezüglich der Erkennung und Vermeidung von Fehlern ergibt sich daraus eine Vielzahl an methodischen Ansatzpunkten. Häufig sind Produktfehler auf ein unzureichendes Design zurückzuführen, das die Architektur und die Interaktion zwischen den Subsystemen nicht ausreichend präzise beschreibt. Diese Fehler hätten also schon in frühen Entwicklungsphasen, d. h. bei der Erstellung der Systemspezifikation und des Designs, gefunden oder gar vermieden werden können. Daher untersucht diese Arbeit bezüglich der Erkennung und Vermeidung von Fehlern speziell die frühen Entwicklungsphasen. Der Schwerpunkt liegt dabei auf der Fehlervermeidung während der Entwicklung. Prinzipiell gehört dazu auch die autonome Fehlerbehandlung zur Laufzeit, da es sich hierbei um Mechanismen im Produkt handelt, die ebenfalls schon in frühen Entwicklungsphasen zu berücksichtigen sind. Auf physikalische Störeinflüsse und deren Beherrschung wird in dieser Arbeit ebenfalls eingegangen, sie spielen aber nur eine untergeordnete Rolle und werden wie die Fehlertoleranzmechanismen nur am Rande behandelt.

2.2 Der Fehler-Begriff

Da in dieser Arbeit häufig von Fehlern gesprochen wird, soll der Begriff zunächst genauer definiert werden. Eine formale Definition wird später in Abschnitt 4.2.1 gegeben.

In der Literatur ist der Begriff des Fehlers nicht einheitlich festgelegt und wird in unterschiedlichen Bedeutungen verwendet. In deutschen Texten findet man Begriffe wie *Fehler*, *Ausfall*, *Versagen* und *Fehlerursache*, während in der englischsprachigen Literatur Begriffe wie *error*, *fault*, *bug* und *failure* zu lesen sind. Diese Begriffe lassen sich nicht unmittelbar einander zuordnen. Beispielsweise werden *error* und *bug* meist als *Fehler* übersetzt, wohingegen *failure* im Deutschen in *Ausfall* und *Versagen* unterschieden wird [Lap92, IEE90]. Der Fehler-Begriff in dieser Arbeit richtet sich nach [Lap92].

Allen im Folgenden verwendeten Fehler-Begriffen liegt folgende Definition des Begriffs *Fehler* zu Grunde:

Definition 2.2 (Fehler)
Ein *Fehler* ist eine Abweichung zwischen Soll und Ist. □

Wesentlich ist dabei, dass ein Fehler nur dann festgestellt werden kann, wenn es eine – als richtig angesehene – Soll-Vorgabe gibt. Nur wenn Soll und Ist nicht übereinstimmen, liegt ein Fehler vor. Eine Soll-Vorgabe muss nicht notwendigerweise schriftlich vorliegen. Es kann auch mehrere Soll-Vorgaben geben, wobei dann klar sein muss, auf welche sich der Fehler-Begriff bezieht. Abschnitt 2.4 vertieft diesen Aspekt.

Fehlerursache und -wirkung

Im Folgenden wird der Fehler-Begriff weiter differenziert in *Fehlerursache*, *Fehlzustand* und *Versagen*. Dadurch lässt sich besser verdeutlichen, auf welche Eigenschaft eines Systems man sich konkret bezieht.

Definition 2.3 (Fehlerursache (engl. fault))
Eine *Fehlerursache* ist eine Abweichung des tatsächlichen vom beabsichtigten Aufbau eines Produkts. □

Definition 2.4 (Fehlzustand (engl. error))
Ein *Fehlzustand* oder auch *fehlerhafter Zustand* liegt vor, wenn ein Produkt in einen nicht beabsichtigten internen Zustand übergeht. □

Definition 2.5 (Versagen (engl. failure))
Ein *Versagen* oder auch *fehlerhaftes Verhalten* ist ein von der Spezifikation abweichendes, nach außen sichtbares Verhalten eines Produkts. □

Ein Versagen bezieht sich auf das Eingabe-Ausgabe-Verhalten eines Produkts. Verhält sich das Produkt nicht gemäß einer bestimmten Spezifikation, so liegt ein Versagen vor. Eine *Spezifikation* eines Produkts ist dabei eine Beschreibung der erwarteten Funktionen und/oder Leistungen und die Bedingungen, unter denen sie zu erbringen sind [Lap92].

Damit ein Versagen auftreten kann, muss ein Fehlzustand vorliegen, d. h. das Produkt befindet sich in einem nicht beabsichtigten internen Zustand. Führt der Fehlzustand zu einer falschen Ausgabe und wird so für die Umgebung des Produkts sichtbar, so spricht man von einem Versagen. Das Vorliegen eines Fehlzustands muss aber nicht in jedem Fall zu einem Versagen führen. Durch den Einsatz von Fehlertoleranzverfahren wird versucht, einen Fehlzustand rechtzeitig zu erkennen und in einen beabsichtigten internen Zustand überzuführen, um dadurch ein Versagen zu vermeiden.

Ebenso verhält es sich zwischen Fehlerursache und Fehlzustand. Damit ein Fehlzustand auftreten kann, muss eine Fehlerursache, also eine Abweichung im Aufbau des Produkts, vorliegen. Führt eine Fehlerursache zu einem Fehlzustand, spricht man von einer *aktivierten* Fehlerursache. Das Vorhandensein einer Fehlerursache muss aber nicht in jedem Fall zu einem Fehlzustand führen.

Beispiel 2.1 Betrachtet man ein Softwaremodul, so wäre eine Codezeile, in der ein Operator `+` mit `-` vertauscht wurde, eine *Fehlerursache*. Kommt das Modul zur Ausführung, kann diese Codezeile zu einem falschen Zwischenergebnis führen, also einer falschen Variablenbelegung, einem *Fehlzustand*. Wenn das falsche Zwischenergebnis zur Ausgabe eines Ergebnisses verwendet wird und so als falsche Antwort an die Umgebung des Moduls sichtbar wird, liegt ein *Versagen* des Moduls vor. □

Fehlerursache, Fehlzustand und Versagen beschreiben eine nach beiden Seiten offene *Fehlerfortpflanzungskette* [Lap92, ALRL04]. Jede Fehlerursache kann als Auswirkung und damit Versagen einer zeitlich davor liegenden Ursache aufgefasst werden. Ebenso kann jedes Versagen die Ursache für ein weiteres Versagen (also eines Folgefehlers) sein. Beispielsweise kann das Versagen einer Komponente als Ursache für das Versagen des gesamten Systems gesehen werden.

$$\cdots \rightarrow \text{Fehlerursache} \xrightarrow{\text{erzeugt}} \text{Fehlzustand} \xrightarrow{\text{wird sichtbar als}} \text{Versagen} \xrightarrow{\text{fungiert als}} \rightarrow \text{Fehlerursache} \rightarrow \cdots$$

Häufig unterscheidet man zusätzlich zwischen *Versagen* und *Ausfall*, siehe [Gra07]. Dabei ist Ausfall ein Spezialfall von Versagen: Von einem Ausfall eines Produkts spricht man, wenn es (vorübergehend) die Fähigkeit verliert, eine vorgegebene Funktionalität zu erbringen. Der Ausfall geht also immer mit einer Veränderung des Produkts einher, die entweder temporär (z. B. durch Überhitzung) oder permanent (z. B. durch Kabelbruch) vorliegt. Im Zusammenhang mit Software spricht man nur von Versagen, weil Software ein logisches Konstrukt ist und keinen physikalischen Einflüssen unterliegt, also insbesondere sich nicht abnutzt.

Weiter gehende Unterteilungen des Fehler-Begriffs finden sich beispielsweise in [Lap92, ALRL04, Mar03]. Der Begriff *Fehler* selbst wird künftig als Oberbegriff oder stellvertretend für eine der genannten Fehlerarten verwendet, wenn die Bedeutung aus dem Zusammenhang klar hervorgeht.

2.3 Grundidee der Integration

Dieser Abschnitt erläutert die Grundidee der Integration und definiert Fehler, die in diesem Zusammenhang auftreten können. Eine entsprechende formale Darstellung dazu wird später in den Abschnitten 4.1 und 4.2.2 angegeben.

2.3.1 Modulare Entwicklung

Das Automobil ist ein klassisches Beispiel für eine modulare Entwicklung. Die Systemspezifikation wird zunächst zerlegt in viele einzelne Module, die von verschiedenen Zulieferern entwickelt, gefertigt und geliefert werden. Bei der Integration werden die Module vom OEM schließlich zum Gesamtsystem zusammengebaut, d. h. integriert.

Dazu einige Begriffsdefinitionen:

Definition 2.6 (System)
Ein *System* ist eine Einheit, die mit ihrer Umgebung, d. h. anderen Systemen, kommuniziert. Ein System kann z. B. ein Stück Software, Hardware, eine Person oder die

physikalische Welt sein. Die gemeinsame Grenze zwischen einem System und seiner Umgebung wird *Systemgrenze* genannt.[2] □

In dieser Arbeit wird nicht zwischen System, Modul, Komponente und Produkt unterschieden. Der Begriff System dient hier als gemeinsamer Oberbegriff.

Definition 2.7 (Subsystem)
Ein *Subsystem* ist ein System, welches Bestandteil eines (übergeordneten) Systems ist. Der Begriff Subsystem wird verwendet, um auf eine ist-Teil-von-Beziehung zwischen einem System und seinem übergeordneten System hinzuweisen. □

Ein Subsystem kann entweder wiederum hierarchisch zusammengesetzt sein oder eine nicht mehr sinnvoll weiter zerlegbare Einheit darstellen.

Definition 2.8 (Architektur)
Eine *Architektur* eines Systems ist gegeben durch eine Menge von Subsystemen zusammen mit den Kommunikationsverbindungen zwischen den Subsystemen. □

Definition 2.9 (Integration)
Integration (oder *Systemintegration*) ist das Zusammenfügen mehrerer realisierter Subsysteme zu einem System. □

Wendet man die Idee der Integration auf Spezifikationen an und vereint die Subsystemspezifikationen, um damit beispielsweise Eigenschaften des gesamten Systems vorherzusagen, so spricht man von *virtueller Integration*. Dieses Vorgehen findet bei der modellbasierten Entwicklung Anwendung.

Abbildung 2.2 stellt die modulare Entwicklung eines Systems schematisch dar. Ausgangspunkt ist eine Systemspezifikation S. Diese wird zerlegt in mehrere Subsystemspezifikationen S_i zusammen mit einer Architekturbeschreibung A, die das Zusammenspiel der Subsysteme beschreibt. Die einzelnen Subsysteme werden dann unabhängig voneinander erstellt und es entsteht jeweils eine Realisierung R_i. Anschließend werden die Subsysteme gemäß Architekturbeschreibung A zum System R integriert.

Die modulare Entwicklung wird in der Regel dazu genutzt, ein komplexes System in mehrere weniger komplexe Subsysteme – hier Module genannt – zu zerlegen und diese arbeitsteilig zu entwickeln und zu produzieren.

Ist ein zu realisierendes Subsystem S_i recht komplex, kann die Idee der modularen Entwicklung selbstverständlich erneut angewendet und S_i in weitere Subsysteme zerlegt werden.

[2] Der hier verwendete Systembegriff entspricht dem in [ALRL04].

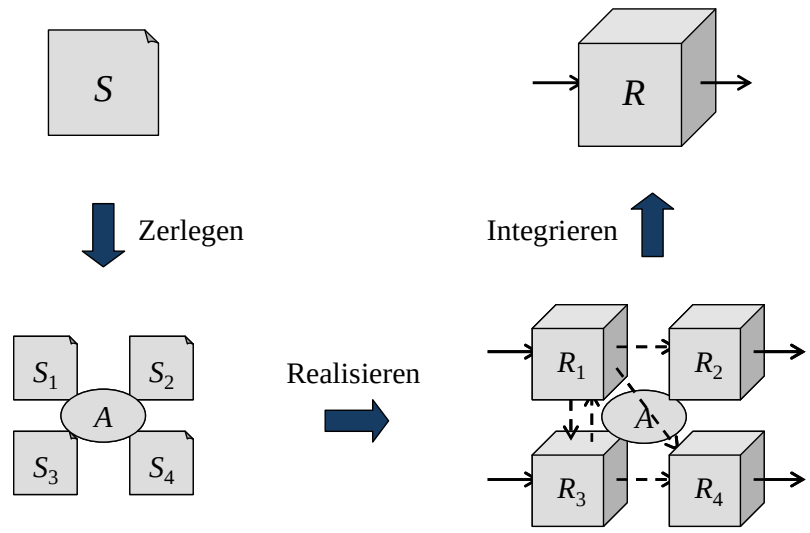


Abbildung 2.2: Schematische Darstellung der modularen Entwicklung

2.3.2 Architektur- und Integrationsfehler

Das Ziel der Integration ist, das fehlerfreie Zusammenspiel der Subsysteme zu gewährleisten. Im Vordergrund steht dabei ein fehlerfreies Zusammenspiel aus Architektursicht und weniger aufgrund der Korrektheit der einzelnen, beteiligten Subsysteme. Unter einem Integrationsproblem wird allgemein ein Fehler, der beim Zusammenspiel der Subsysteme beobachtet wird, verstanden (siehe Kapitel 1). Im Hinblick auf die Fehlerursache sollen im Folgenden die Begriffe *Architekturfehler* und *Integrationsfehler* unterschieden werden.

Definition 2.10 (Architekturfehler)
Ein *Architekturfehler* ist eine fehlerhafte Zerlegung einer Systemspezifikation, d. h. die durch die Zerlegung entstandene Architektur stimmt nicht mit der Systemspezifikation überein. □

Definition 2.11 (Integrationsfehler)
Ein *Integrationsfehler* ist eine fehlerhafte Integration eines Systems, d. h. das durch die Integration entstandene System stimmt nicht mit der Architekturbeschreibung überein. □

Gemäß Abbildung 2.2 liegt ein Architekturfehler vor, wenn beim Zerlegen der Systemspezifikation ein Fehler entsteht, und ein Integrationsfehler, wenn beim Integrieren der Subsystemrealisierungen zum System ein Fehler entsteht.

Der Grundstein für die Integration wird bereits zu dem Zeitpunkt gelegt, zu dem die Subsysteme des Systems identifiziert werden. Teilweise sind einige Subsysteme

bereits in den Anforderungen festgelegt, andere entstehen erst mit den Architekturentscheidungen in der Designphase und später. Ein typisches Beispiel für einen Architekturfehler ist ein Kommunikationsproblem zwischen zwei Subsystemen aufgrund unterschiedlicher Bitdarstellung der ausgetauschten Information.

Auch während der Integration, also des Zusammensetzens des Systems aus seinen Einzelbestandteilen, können Fehler entstehen. Beispielsweise können Subsysteme vergessen, falsche Subsystemversionen verwendet oder Verbindungskabel zwischen Subsystemen falsch angeschlossen werden.

Nicht betrachtet werden Fehler bei der Realisierung einzelner Subsysteme, da diese nicht explizit die Architektur betreffen.

Diese Arbeit konzentriert sich auf die Erkennung und Vermeidung von Architekturfehlern, da bereits in frühen Entwicklungsphasen ein großer Einfluss auf die spätere Integration besteht und hier gemachte Fehler in nachfolgenden Entwicklungsphasen nur noch schwer auszugleichen sind.

2.4 Fehlerlokalisierung im Entwicklungsprozess

Im zeitlichen Verlauf des Entwicklungsprozesses können Fehler in verschiedenen Entwicklungsphasen entstehen. Abhängig von der Sicht des Betrachters kann ein Fehler unterschiedlich ausgelegt werden, wenn der zugehörige Referenzpunkt, d. h. das *Soll* gemäß Definition 2.2 des Fehlerbegriffs, nicht einheitlich ist. Dieser Abschnitt erklärt, woher Fehler kommen, welche Fehler es in den unterschiedlichen Entwicklungsphasen geben kann und wie man mit unvollständigen Spezifikationen umgehen kann.

2.4.1 Tatsächliche und ideale Artefakte

Während der Entwicklung eines Produkts werden mehrere Zwischenprodukte, hier als *tatsächliche Artefakte* bezeichnet, erstellt und weiter verwendet. Betrachtet am Modell der modularen Entwicklung gehören dazu die tatsächliche Spezifikation, die tatsächliche Architektur, die tatsächliche Realisierung der einzelnen Subsysteme und das tatsächliche System.

Die Feststellung, ob ein Artefakt fehlerfrei ist, erfordert einen Vergleich zwischen einem tatsächlichen Artefakt und einem Referenzartefakt, hier als *ideales Artefakt* bezeichnet. Ideale Artefakte sind die ideale Spezifikation, die ideale Architektur, ideale Subsystemrealisierungen und das ideale System. Ein Fehler liegt vor, wenn ein tatsächliches Artefakt von einem als korrekt angesehenen idealen Artefakt abweicht, d. h. die Verfeinerungsbeziehung nicht gilt. Konkret spricht man von einem *Spezifikationsfehler*, wenn die tatsächliche Spezifikation von der idealen abweicht. Analoges gilt für *Architekturfehler*, *Subsystemfehler* und *Systemfehler*.

Die idealen Artefakte sind in der Regel nicht eindeutig vorgegeben, sondern müssen ausgewählt werden, um eine Abweichung zwischen einem konkreten Produkt und

der entsprechenden Idealvorstellung ausdrücken zu können. Zwischen den idealen Artefakten, die man in einem konkreten Fall ausgewählt hat, wird man sinnvollerweise eine gewisse Kompatibilität verlangen, insbesondere die Korrektheitsbeziehungen gemäß der modularen Entwicklung aus Abschnitt 4.1. Somit ist die Wahlfreiheit der idealen Artefakte auf die möglichen Verfeinerungsbeziehungen, ausgehend von der idealen Spezifikation, eingeschränkt. Die idealen Artefakte sollen im Folgenden automatisch alle geforderten Eigenschaften erfüllen, was durch das Adjektiv *ideal* bereits suggeriert wird.

Neben der Abweichung von den idealen Artefakten können auch die Übergänge *zwischen* den tatsächlichen Artefakten fehlerbehaftet sein. Die entsprechenden Fehlerbegriffe sind *Zerlegungsfehler*, *Realisierungsfehler* und *Integrationsfehler*. Abbildung 2.3 zeigt eine Übersicht über die genannten Artefakte und die zugehörigen Fehlerbegriffe. Entsprechend dem üblichen Sprachgebrauch wird im Folgenden nicht unterschieden zwischen Architekturfehler und Zerlegungsfehler. Falls für das Verständnis erforderlich, wird im Einzelfall angegeben, welchem Artefakt die tatsächliche Architektur gegenübergestellt wird.

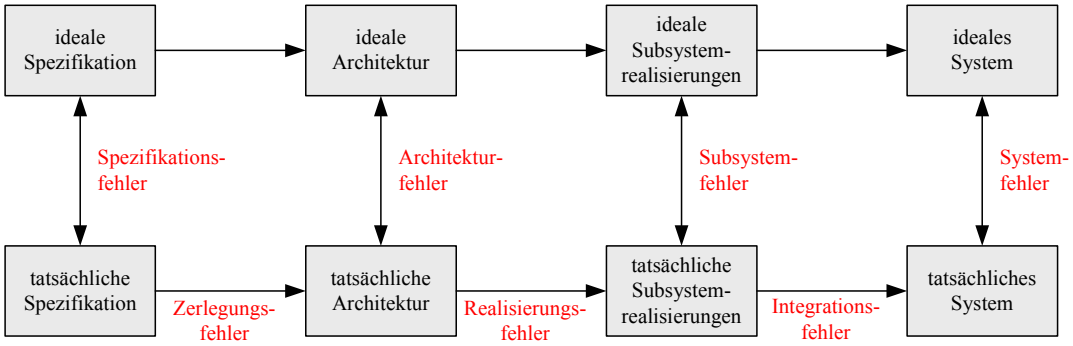


Abbildung 2.3: Ideale und tatsächliche Artefakte im Entwicklungsprozess mit Einordnung der Fehlerbegriffe

2.4.2 Mentale Artefakte

Obwohl die idealen Artefakte oft nicht explizit bekannt sind, glauben die Entwickler mehr oder weniger genau zu wissen, was zu entwickeln ist und wie das Ergebnis am Schluss auszusehen hat. Der Grund dafür sind gedankliche Modelle in den Köpfen der Entwickler, sog. *mentale Artefakte*. Sie treten an die Stelle der idealen Artefakte und bilden für die Entwickler die Grundlage für die Feststellung, ob ein Fehler vorliegt oder nicht (Abbildung 2.4).

Jeder Entwickler verfügt über ein eigenes mentales Modell. Die mentalen Artefakte stimmen in der Praxis nie exakt mit den fiktiven idealen Artefakten überein, sie

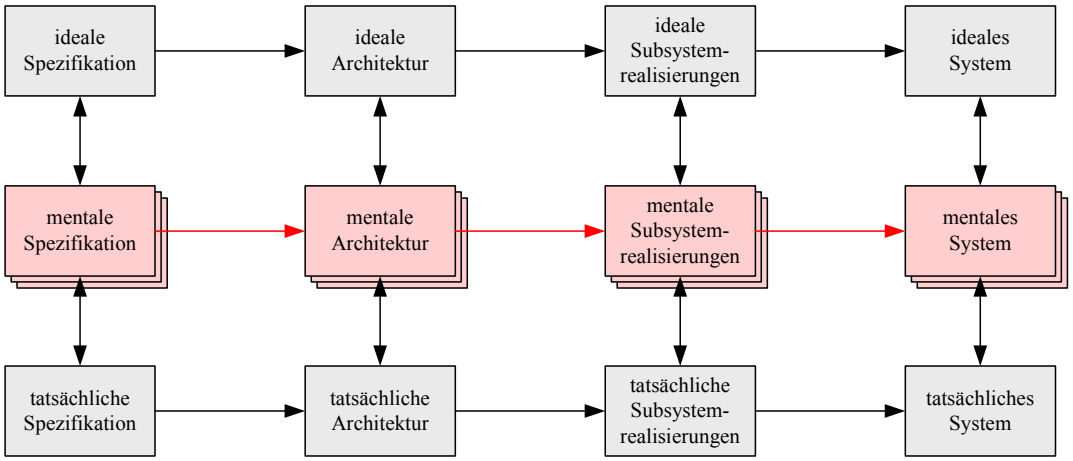


Abbildung 2.4: Einfluss der mentalen Artefakte der Entwickler

spiegeln eher die subjektive Einschätzung des Entwicklers wider. Diese ist geprägt von persönlicher Erfahrung, eigener Intuition und implizit getroffenen Annahmen („nach bestem Wissen und Gewissen“). Eine Gefahr ist dabei, dass mehrere Entwickler, die häufig noch dazu in ganz unterschiedlichen Domänen mit unterschiedlichem Begriffsverständnis arbeiten, zu deutlich von einander abweichenden Vorstellungen über das ideale System kommen können, d. h. sich deren mentale Artefakte stark unterscheiden. Nach arbeitsteiliger Entwicklung der Subsysteme stellt man dann bei der Integration fest, dass die realisierten Subsysteme gar nicht zusammenpassen. Aber auch wenn die mentalen Artefakte mehrerer Entwickler weitgehend gleich sein sollten, lässt sich daraus nichts über die Übereinstimmung mit den idealen Artefakten ableiten.

Die mentalen Artefakte bilden sich oft auf der Grundlage der vorhandenen Spezifikationen. Die Gefahr für große Abweichungen zwischen den mentalen Artefakten ist besonders dann recht hoch, wenn die Spezifikationen unvollständig sind, denn in diesem Fall gibt es einen größeren Interpretationsspielraum. Abhilfe schaffen kann nur eine präzisere Spezifikation oder zumindest eine Abstimmung unter den Entwicklern.

Die mentalen Artefakte können das Ergebnis eines Entwicklungsprozesses in der Praxis sehr stark beeinflussen. Für die objektive Bestimmung von Fehlern ist jedoch immer das jeweilige ideale Artefakt entscheidend. Daher wird bei der Fehlerklassifizierung in dieser Arbeit versucht, das ideale Artefakt zu schätzen.

3 Grundlagen: Formales Systemmodell

In diesem Kapitel wird ein formales Systemmodell vorgestellt, mit dessen Hilfe sich die Struktur und das Verhalten von Netzwerken eingebetteter Systeme beschreiben lassen. Es stellt die Grundlage für die nachfolgenden Kapitel dar und ermöglicht insbesondere eine formale Definition des Fehlerbegriffs sowie die Formulierung formaler Korrektheitseigenschaften von Entwicklungsprozessen.

3.1 Überblick

Das im Folgenden betrachtete Systemmodell basiert auf der Methodik FOCUS [BS01]. Es weicht nur an wenigen Stellen davon ab, was dann jeweils im Detail beschrieben wird. Dieses Kapitel geht im Wesentlichen nur auf die in dieser Arbeit verwendeten Bestandteile des Systemmodells ein.

FOCUS wurde als Ansatz gewählt, weil es die folgenden Merkmale zur Beschreibung von Systemen besonders gut unterstützt:

- Vollständige, funktionale Beschreibung von digitalen Systemen,
- Darstellung des Zeitverhaltens, insbesondere der Nebenläufigkeit,
- modulare Entwicklung,
- klare Semantik von Systemen,
- zustandslose Beschreibung von Systemen (Black-Box-Sicht) sowie
- Nichtdeterminismus.

Nach einer Einführung in die Repräsentation von Daten und Zeit mittels Strömen im folgenden Abschnitt stellt Abschnitt 3.3 die Verhaltensbeschreibung mittels Relationen zwischen Ein- und Ausgabeströmen vor. In Abschnitt 3.4 wird die Komposition mehrerer Subsysteme zu einem System beschrieben. Abschnitt 3.5 geht schließlich auf die Grenzen des Systemmodells ein.

3.2 Datenströme und Operatoren auf Strömen

Die Kommunikation auf einem Kanal zwischen zwei Kommunikationspartnern erfolgt unidirektional über eine Folge von Zeichen aus einem bestimmten Zeichenvorrat. Diese Zeichenfolge wird Strom genannt. Ein Strom stellt die gesamte Kommunikationshistorie zwischen zwei Kommunikationspartnern dar und kann endlich oder unendlich lang sein. Darüber hinaus erlaubt es FOCUS, über das spezielle Symbol $\surd$ explizit Zeitticks zu modellieren.

3.2.1 Formale Darstellung von Strömen

Sei $\mathbb{N}_+ \stackrel{\text{def}}{=} \mathbb{N} \setminus \{0\}$ die Menge aller positiven natürlichen Zahlen. $[\,1 \ldots n\,]$ sei die Menge aller natürlichen Zahlen zwischen 1 und n für $n \geq 1$ und die leere Menge für $n = 0$.

Definition 3.1 (Strom)
Sei M eine beliebige, nichtleere Menge (Alphabet). Ein *endlicher Strom* über M ist eine Abbildung $[\,1 \ldots n\,] \to M$, d. h. eine endliche Folge in M. Ein *unendlicher Strom* über M ist eine Abbildung $\mathbb{N}_+ \to M$, d. h. eine unendliche Folge in M. Für die Darstellung eines Stroms s wird folgende Notation verwendet:

$$s = \langle m_1, m_2, m_3, \ldots \rangle$$

Der leere Strom wird mit $\langle\rangle$ dargestellt. □

$s(1) = m_1$ ist das erste Element, $s(2) = m_2$ das zweite Element und $s(i) = m_i$ das i-te Element des Stroms. Alle Elemente m_i werden auch als *Nachrichten* bezeichnet und sind aus dem Alphabet M des Stroms.

Definition 3.2 (Menge aller Ströme)
Sei M eine beliebige, nichtleere Menge. Die Menge M^* aller endlichen Ströme, die Menge M^∞ aller unendlichen Ströme und die Menge M^ω aller Ströme werden definiert als

$$M^* \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} \{s \mid s : [\,1 \ldots n\,] \to M\},$$

$$M^\infty \stackrel{\text{def}}{=} \mathbb{N}_+ \to M,$$

$$M^\omega \stackrel{\text{def}}{=} M^* \cup M^\infty.$$

□

3.2.2 Darstellung der Zeit

Die eingebetteten Systeme eines Automobils bilden ein verteiltes, nebenläufiges System. Da allein aus dem Inhalt und der Position der ausgetauschten Nachrichten in den Strömen noch kein Schluss auf die zeitliche Beziehung zweier Nachrichten in verschiedenen Strömen möglich ist, wird ein systemweiter Zeittakt eingeführt. FOCUS bietet hierzu zeitbehaftete Ströme (engl. timed streams) an.

Definition 3.3 (Zeitbehafteter Strom)
Sei M eine beliebige, nichtleere Menge. Sei $\surd \notin M$ das Symbol, das einen Zeittick repräsentiert. Die Menge $M^{\underline{*}}$ aller endlichen, zeitbehafteten Ströme über M, die Menge $M^{\underline{\infty}}$ aller unendlichen, zeitbehafteten Ströme über M und die Menge $M^{\underline{\omega}}$ aller zeitbehafteten Ströme über M werden definiert als

$$M^{\underline{*}} \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} \{s \mid s : [1 \ldots n] \to M \cup \{\surd\}\},$$

$$M^{\underline{\infty}} \stackrel{\text{def}}{=} \{s \mid s : \mathbb{N}_+ \to M \cup \{\surd\} \text{ und } \forall j \in \mathbb{N}_+ \ \exists k \in \mathbb{N}_+ : k \geq j \wedge s(k) = \surd\},$$

$$M^{\underline{\omega}} \stackrel{\text{def}}{=} M^{\underline{*}} \cup M^{\underline{\infty}}.$$

□

Jeder Strom aus $M^{\underline{\infty}}$ enthält unendlich viele Zeitticks. Zwischen jeweils zwei benachbarten Zeitticks im Strom befinden sich höchstens endlich viele Zeichen aus M. Die Zeitticks unterteilen den Strom also in eine unendliche Folge von endlichen Teilströmen. Die Teilströme können auch leer sein.

Beispiel 3.1 Beispiel für einen endlichen, zeitbehafteten Strom:

$$s = \langle m_1, \surd, m_2, m_3, \surd, \surd, m_4 \rangle$$

□

Im Folgenden wird *Strom* als Oberbegriff sowohl für einen zeitlosen als auch für einen zeitbehafteten Strom verwendet. Wenn nichts anderes angegeben ist, sei M stets eine beliebige, nichtleere Menge.

3.2.3 Operatoren auf Strömen

Viele Stromoperatoren sind sowohl auf zeitlose als auch auf zeitbehaftete Ströme anwendbar. Es genügt, diese nur auf zeitlosen Strömen zu definieren, da sie damit automatisch auch auf zeitbehaftete Ströme anwendbar sind: Jeder zeitbehaftete Strom ist auch ein zeitloser Strom, wenn man das Symbol $\surd$ in das Alphabet des zeitlosen Stroms aufnimmt, d. h. $M^{\underline{\omega}} \subseteq (M \cup \{\surd\})^{\omega}$.

Definition 3.4 (Länge eines Stroms)
$\#s$ bezeichnet die Länge des Stroms s. Es gilt:

$$\# : M^{\omega} \to \mathbb{N}_{\infty}$$

$$\#s \stackrel{\text{def}}{=} \begin{cases} n & \text{falls } s : [\,1 \ldots n\,] \to M \text{ für ein } n \in \mathbb{N} \\ \infty & \text{falls } s \in M^{\infty} \end{cases}$$

□

Definition 3.5 (Konkatenation von Strömen)
$r \frown s$ bezeichnet die Konkatenation der Ströme r und s. Es gilt:

$$\frown : M^{\omega} \times M^{\omega} \to M^{\omega}$$

und für alle $i \in \mathbb{N}_{+}$ ist

$$(r \frown s)(i) \stackrel{\text{def}}{=} \begin{cases} r(i) & \text{falls } 1 \leq i \leq \#r \\ s(i - \#r) & \text{falls } \#r < i \leq \#r + \#s \end{cases}$$

□

Wichtige Eigenschaften der Konkatenation sind:

- $\#r = \infty \quad \Rightarrow \quad r \frown s = r$
- $\#(r \frown s) = \#r + \#s$
- $\langle\rangle$ ist neutrales Element: $\quad s \frown \langle\rangle = s, \quad \langle\rangle \frown s = s$
- Assoziativität: $\quad (r \frown s) \frown t = r \frown (s \frown t)$

Definition 3.6 (Potenzierung eines Stroms)
s^n bezeichnet den Strom, welcher durch n-malige Konkatenation von s entsteht. Es gilt:

$\widehat{\ } : M^{\omega} \times \mathbb{N}_{\infty} \to M^{\omega}$, $\quad$ wobei anstatt $s \widehat{\ } n$ immer s^n geschrieben wird

$$s^n \stackrel{\text{def}}{=} \begin{cases} \langle\rangle & \text{falls } n = 0 \\ s \frown s^{(n-1)} & \text{falls } n \geq 1 \end{cases}$$

□

Beispiel 3.2 $\langle m, \surd \rangle^4 = \langle m, \surd, m, \surd, m, \surd, m, \surd \rangle$ □

Definition 3.7 (Präfix, Anfangsstrom)
Ein Strom r ist ein Präfix (Anfangsstrom) eines Stroms s, kurz $r \sqsubseteq s$, wenn r zu s verlängert werden kann. Für alle $r, s \in M^{\omega}$ gilt:

$$r \sqsubseteq s \quad \Leftrightarrow_{\mathsf{def}} \quad \exists\, t \in M^{\omega} : r \frown t = s$$

□

Beispiel 3.3 $\langle m_1, m_2 \rangle \sqsubseteq \langle m_1, m_2, m_3, m_4 \rangle$ □

Die Präfixrelation $\sqsubseteq$ ist eine Halbordnung auf M^{ω}, d. h. eine reflexive, antisymmetrische und transitive Relation.

Definition 3.8 (Filter)
$D \circledS s$ bezeichnet einen Teilstrom (substream) von s, welcher entsteht, wenn alle Nachrichten aus s entfernt werden, die nicht in D enthalten sind. $\circledS$ wird als Filter-Operator bezeichnet. Es gilt:

$$\circledS : \mathcal{P}(M) \times M^{\omega} \to M^{\omega}$$

$$D \circledS s \overset{\mathsf{def}}{=} \begin{cases} \langle\rangle & \text{falls } s = \langle\rangle \\ \langle m \rangle \frown (D \circledS r) & \text{falls } s = \langle m \rangle \frown r \text{ und } m \in D \\ D \circledS r & \text{falls } s = \langle m \rangle \frown r \text{ und } m \notin D \end{cases}$$

□

Definition 3.9 (Abschneiden eines Stroms)
$s|_i$ bezeichnet einen Strom, welcher nach dem i-ten Eintrag abgeschnitten ist. Es gilt:

$$| : M^{\omega} \times \mathbb{N}_{\infty} \to M^{\omega}$$

$$s|_i \overset{\mathsf{def}}{=} \begin{cases} r & \text{falls } i \leq \#s, \text{ wobei } r \sqsubseteq s \text{ mit } \#r = i \\ s & \text{falls } i > \#s \end{cases}$$

□

Definition 3.10 (Zeitbehaftetes Abschneiden eines Stroms)
$s{\downarrow_i}$ bezeichnet einen zeitbehafteten Strom, welcher direkt nach dem i-ten Zeittick abgeschnitten ist. Es gilt:

$$\downarrow : M^{\underline{\omega}} \times \mathbb{N}_{\infty} \to M^{\underline{\omega}}$$

$$s{\downarrow_i} \overset{\mathsf{def}}{=} \begin{cases} \langle\rangle & \text{falls } i = 0 \\ s & \text{falls } i > \#(\{\surd\} \circledS s) \text{ oder } i = \infty \\ r & \text{sonst, wobei } r \sqsubseteq s \text{ mit } \#(\{\surd\} \circledS r) = i \text{ und } r(\#r) = \surd \end{cases}$$

□

Die ursprüngliche Definition des ↓-Operators in FOCUS erstreckt sich nur auf unendliche Ströme. Die hier vorgestellte Definition erweitert den Definitionsbereich des Operators auf endliche Ströme.

Beispiel 3.4 Vergleich der Operatoren zum zeitlosen bzw. zeitbehafteten Abschneiden von Strömen:

$$\langle m_1, \surd, m_2, \surd, m_3, \surd, m_4\rangle|_3 = \langle m_1, \surd, m_2\rangle$$

$$\langle m_1, \surd, m_2, \surd, m_3, \surd, m_4\rangle\downarrow_3 = \langle m_1, \surd, m_2, \surd, m_3, \surd\rangle$$

□

Definition 3.11 (Zeitabstraktion)
$\overline{s}$ bezeichnet einen zeitlosen Strom, welcher durch Entfernen aller Zeitticks aus dem zeitbehafteten Strom s entsteht. Es gilt:

$$^{-} : M^{\underline{\omega}} \rightarrow M^{\omega}$$

$$\overline{s} \stackrel{\text{def}}{=} M \circledS s$$

□

Beispiel 3.5 $\overline{\langle m_1, m_2, \surd, m_3, \surd, m_4\rangle} = \langle m_1, m_2, m_3, m_4\rangle$ □

3.3 Verhaltensbeschreibung mittels Stromrelationen

Aus Informationsverarbeitungssicht ist ein Systemverhalten eine Zuordnung von Eingaben zu Ausgaben. FOCUS verwendet ebenfalls diese Sicht und beschreibt das Verhalten eines Systems als Relation zwischen Eingabeströmen und Ausgabeströmen. Bei deterministischen Systemen ist diese Relation eine stromverarbeitende Funktion.

3.3.1 Systemverhalten

Definition 3.12 (Kanal)
Ein *Kanal* ist ein expliziter Kommunikationsweg, den ein System zur Kommunikation mit seiner Umgebung bereitstellt. Jeder Kanal hat einen Namen, ein Alphabet und ist unidirektional. Man unterscheidet zwischen Ein- und Ausgabekanälen. □

Damit zwei Systeme miteinander kommunizieren können, muss ein Ausgabekanal des einen Systems mit einem Eingabekanal des anderen Systems verbunden sein und die beiden Alphabete müssen gleich sein.

Definition 3.13 (Schnittstelle)
Die *Schnittstelle* eines Systems ist die Menge aller Kanäle des System. Die Namen der Kanäle eines Systems müssen dabei paarweise verschieden sein. □

Beispiel 3.6 Betrachtet werden soll ein System *Merge*, das zwei Eingabeströme zu einem Ausgabestrom verarbeitet, indem die Nachrichten zwischen jeweils zwei Zeitticks in den Eingabeströmen konkateniert und einen Zeittakt später ausgegeben werden. *Merge* hat die Eingabekanäle i_1 und i_2 sowie den Ausgabekanal o. Die Alphabete der Eingabekanäle sind zwei beliebige, nichtleere Mengen M_1 und M_2, das Alphabet des Ausgabekanals ist $M_1 \cup M_2$. Abbildung 3.1 stellt die Schnittstelle grafisch dar. □

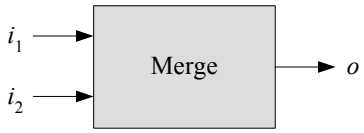


Abbildung 3.1: Schnittstelle des Systems *Merge*

Definition 3.14 (Verhaltensrelation, Systemverhalten)
Seien $I_1, \ldots, I_n$ die Alphabete der Eingabekanäle und $O_1, \ldots, O_m$ die Alphabete der Ausgabekanäle eines Systems S. Zur Abkürzung werden

$$I_S{}^{\omega} \stackrel{\text{def}}{=} I_1{}^{\omega} \times \cdots \times I_n{}^{\omega}$$

$$O_S{}^{\omega} \stackrel{\text{def}}{=} O_1{}^{\omega} \times \cdots \times O_m{}^{\omega}$$

eingeführt. Das Verhalten des Systems S ist eine Relation

$$\mathcal{R}_S \subseteq I_S{}^{\omega} \times O_S{}^{\omega}$$

$\mathcal{R}_S$ nennt man auch *Verhaltensrelation* oder *Systemverhalten*. □

Im Folgenden werden häufig zeitbehaftete Systeme mit unendlichen Strömen betrachtet. Die Verhaltensrelation hat dann die Eigenschaft

$$\mathcal{R}_S \subseteq I_S{}^{\infty} \times O_S{}^{\infty}$$

Beispiel 3.7 Die formale Verhaltensbeschreibung von *Merge* sieht in FOCUS-Notation wie folgt aus.

Merge — timed

in $i_1 : M_1;\quad i_2 : M_2$

out $o : M_1 \cup M_2$

$o = concat(i_1, i_2)$
wobei $concat \in M_1 \overset{\infty}{} \times M_2 \overset{\infty}{} \rightarrow (M_1 \cup M_2) \overset{\infty}{}$ so dass
$\forall\, s_1 \in M_1{}^*, t_1 \in M_1 \overset{\infty}{}, s_2 \in M_2{}^*, t_2 \in M_2 \overset{\infty}{} :$
$concat(s_1 \frown \langle\surd\rangle \frown t_1, s_2 \frown \langle\surd\rangle \frown t_2) = \langle\surd\rangle \frown s_1 \frown s_2 \frown concat(t_1, t_2)$

Im oberen Teil der Spezifikation werden die Ein- und Ausgabekanäle i_1, i_2 und o definiert. Dabei wird nur das Alphabet des Kanals angegeben. Da es sich um ein zeitbehaftetes System handelt – festgelegt durch das Schlüsselwort *timed* rechts oben –, ergeben sich die Mengen aller möglichen Ströme der Kanäle zu $M_1 \overset{\infty}{}$, $M_2 \overset{\infty}{}$ bzw. $(M_1 \cup M_2) \overset{\infty}{}$. □

Man beachte, dass bei einem zeitbehafteten System die Ströme stets unendlich lang sind. Das liegt daran, dass die Zeit niemals endet, unabhängig davon, ob das System Ausgaben produziert oder nicht. Das Systemmodell ordnet auch einem System, das von Anfang an ausgefallen ist und deshalb keinerlei Ausgabe produziert, einen definierten Ausgabestrom zu, nämlich den Strom $\langle\surd\rangle^\infty$, bestehend nur aus Zeitticks.

Die Modellierung mittels zeitbehafteter Ströme erweist sich bei eingebetteten Systemen im Automobil als gut geeignet. Mit Hilfe der Zeitticks kann trotz Nebenläufigkeit ein zeitlicher Bezug zwischen Strömen verschiedener Kanäle hergestellt werden. Eingebettete Systeme im Automobil sehen ebenfalls kein zeitliches Ende ihrer Funktionalität vor (abgesehen von technischen Einschränkungen wie Materialermüdung oder die Verschrottung des Automobils) und sind daher mit unendlichen Strömen zu modellieren.

In Abbildung 3.2 ist anhand Beispielströmen dargestellt, wie das System *Merge* arbeitet.

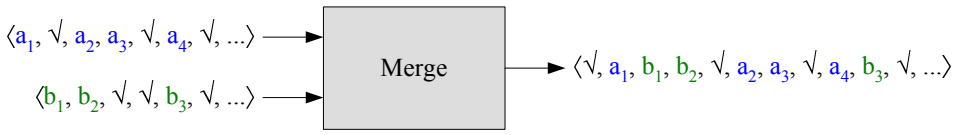


Abbildung 3.2: Verhalten von *Merge* mit Beispielströmen

3.3.2 Black-Box- und Glass-Box-Sicht

Die Verhaltensbeschreibung eines Systems mittels einer Verhaltensrelation benötigt keinerlei Wissen über die internen Vorgänge des Systems. Man spricht deshalb von *Black-Box-Sicht* oder auch *Schnittstellenverhalten*. Da die Berechnung der Ausgabe möglicherweise von der gesamten bisherigen Eingabehistorie abhängt, setzt die Verhaltensrelation $\mathcal{R}_S$ nicht nur einzelne Eingabenachrichten mit einzelnen Ausgabenachrichten in Beziehung, sondern ganze Eingabeströme mit Ausgabeströmen.

Die *Glass-Box-Sicht* (auch White-Box-Sicht) hingegen führt einen Systemzustand ein, der die gesamte bisherige Eingabehistorie repräsentiert. Für die Berechnung der nächsten Ausgabe wird nur die aktuelle Eingabe und der Systemzustand benötigt. Explizites Wissen über die gesamte Eingabehistorie ist nicht mehr erforderlich.

Beispiel 3.8 Das System *Merge* sieht aus Glass-Box-Sicht folgendermaßen aus:

Merge — timed

in $i_1 : M_1;\ \ i_2 : M_2$

out $o : M_1 \cup M_2$

$o = concat[\langle\rangle, \langle\rangle](i_1, i_2)$

wobei *concat* so dass

$\forall\, b_1 \in M_1{}^{*},\ b_2 \in M_2{}^{*},\ s_1 \in M_1,\ s_2 \in M_2,\ t_1 \in M_1{}^{\infty},\ t_2 \in M_2{}^{\infty} :$

$concat[b_1, b_2](\langle s_1\rangle \frown t_1, t_2) = concat[b_1 \frown \langle s_1\rangle, b_2](t_1, t_2)$

$concat[b_1, b_2](t_1, \langle s_2\rangle \frown t_2) = concat[b_2, b_2 \frown \langle s_2\rangle](t_1, t_2)$

$concat[b_1, b_2](\langle\surd\rangle \frown t_1, \langle\surd\rangle \frown t_2) = \langle\surd\rangle \frown b_1 \frown b_2 \frown concat[\langle\rangle, \langle\rangle](t_1, t_2)$

In dieser Darstellung liest *Merge* sukzessive immer nur eine Nachricht aus einem Eingabestrom ein. Zur Zwischenspeicherung der Nachrichten werden zwei interne Puffer verwendet: b_1 puffert die Nachrichten aus Eingabestrom i_1 und b_2 die Nachrichten aus i_2. Sobald der nächste Zeittick an beiden Eingängen anliegt, werden der konkatenierte Pufferinhalt $b_1 \frown b_2$ ausgegeben und die Puffer geleert. Die beiden Puffer bilden zusammen den Systemzustand. Die Glass-Box-Darstellung mit der schrittweisen Verarbeitung der Eingabe ist näher an einer möglichen Implementierung, da sie bereits den konkreten Berechnungsalgorithmus verdeutlicht. □

Im Folgenden wird ausschließlich die Black-Box-Sicht verwendet, weil bei der Systembeschreibung möglichst auf Implementierungsdetails verzichtet werden soll.

3.3.3 Realisierbarkeit eines Systems

Die oben vorgestellte Verhaltensbeschreibung von Systemen erfolgt mittels einer Verhaltensrelation. Dabei lassen sich prinzipiell auch Systeme beschreiben, die keine Entsprechung in der Praxis haben. Ein Beispiel dafür wäre ein System, das in die Zukunft sehen kann, indem es stets Nachrichten ausgibt, die erst einige Zeitticks später auf einem Eingabekanal empfangen werden.

Daher fordert das Systemmodell von einem System zusätzlich, dass es *kausal* ist, d. h. die Ausgabe darf nur von den bereits empfangenen Eingaben abhängen. Dabei wird unterschieden zwischen schwacher und starker Kausalität. Im Gegensatz zur starken Kausalität darf bei der schwachen Kausalität zusätzlich noch die Eingabe im aktuellen Zeitintervall für die Ausgabe berücksichtigt werden.

Definition 3.15 (x-Schnitt einer Relation)
Seien A und B beliebige Mengen und $R \subseteq A \times B$ eine Relation. Der *x-Schnitt* von R, kurz $R(x)$, für ein $x \in A$ ist definiert als

$$R(x) \stackrel{\mathsf{def}}{=} \{y \mid (x, y) \in R\}\,.$$

□

Definition 3.16 (Schwache und starke Kausalität)
Ein System S mit Verhaltensrelation $\mathcal{R}_S \subseteq {I_S}^{\infty} \times {O_S}^{\infty}$ ist *schwach* bzw. *stark kausal*, wenn gilt:

$$\forall\, r, s \in {I_S}^{\infty}\ \forall\, i \in \mathbb{N} :\ r{\downarrow_i} = s{\downarrow_i} \ \Rightarrow\ \mathcal{R}_S(r){\downarrow_i} = \mathcal{R}_S(s){\downarrow_i} \quad \text{bzw.}$$

$$\forall\, r, s \in {I_S}^{\infty}\ \forall\, i \in \mathbb{N} :\ r{\downarrow_i} = s{\downarrow_i} \ \Rightarrow\ \mathcal{R}_S(r){\downarrow_{i+1}} = \mathcal{R}_S(s){\downarrow_{i+1}}$$

Dabei bedeutet das zeitbehaftete Abschneiden $A{\downarrow_i}$ einer Menge A an Strömen, dass jeder Strom einzeln abgeschnitten wird, d. h. $A{\downarrow_i} = \{s{\downarrow_i} \mid s \in A\}$. □

Bei einem stark kausalen System darf jede Ausgabe nur von Eingaben abhängen, die mindestens ein Zeitintervall früher empfangen wurden. Dadurch wird die unweigerlich anfallende Verarbeitungszeit zum Berechnen der Ausgabe ausgedrückt. Bei einem schwach kausalen System hingegen darf zusätzlich die Eingabe im selben Zeitintervall wie die Ausgabe in die Berechnung der Ausgabe einfließen. Beschreibt man eingebettete Systeme in ausreichend kleinen Zeitintervallen, erhält man stets ein stark kausales System (siehe [Bro04]).

Eine Systemspezifikation beschreibt im Allgemeinen, mit welchen möglichen Ausgaben ein System auf bestimmte Eingaben reagieren darf und mit welchen nicht. Sie beschreibt also Einschränkungen des Systemverhaltens. Zum besseren Verständnis sollen folgende Extremfälle betrachtet werden.

Stark unterspezifiziertes System Enthält die Systemspezifikation keine Angaben zum Verhalten, d. h. gibt sie keinerlei Einschränkungen des Systemverhaltens vor, so liegt ein stark unterspezifiziertes System vor. Zu einem stark unterspezifizierten System gibt es stets viele verschiedene Realisierungen, die allesamt der Spezifikation genügen.

Deterministisches System Gibt es zu jeder Eingabe genau eine eindeutige, richtige Ausgabe, so handelt es sich um ein deterministisches System. Es gibt nur eine mögliche Realisierung, die sich im Sinne der Spezifikation korrekt verhält.

Widersprüchliche Spezifikation Wenn die Systemspezifikation zu mindestens einer Eingabe eine widersprüchliche Aussage bezüglich der zugehörigen Ausgabe macht, ist sie widersprüchlich. Es gibt dann kein realisiertes System, das sich bezüglich dieser Spezifikation stets korrekt verhalten kann, weil die Spezifikation bei mindestens einer Eingabe durch den Widerspruch keine korrekte Ausgabe zulässt.

Betrachtet man im ersten Fall die zugehörige Verhaltensrelation $\mathcal{R}_S$, so stellt man fest, dass es sich um die totale Relation handelt, also $\mathcal{R}_S = I_S{}^{\overline{\infty}} \times O_S{}^{\overline{\infty}}$. Jedem Eingabestrom werden alle möglichen Ausgabeströme zugeordnet, es handelt sich um eine hochgradig *nichtdeterministische* Systembeschreibung. Systemspezifikationen sind typischerweise immer nichtdeterministisch und bieten so Freiheit bei der Erstellung der Realisierung.

Bei einem *deterministischen* System ist die zugehörige Verhaltensrelation eine Funktion, d. h. für jeden Eingabestrom $i \in I_S{}^{\overline{\infty}}$ ist der x-Schnitt einelementig, also $\mid \mathcal{R}_S(i) \mid = 1$. Jedes eingebettete System ist deterministisch und stellt eine Funktion dar.[1]

Eine widersprüchliche Spezifikation beschreibt im Unterschied zu den beiden vorhergehenden Punkten ein nicht realisierbares System. Dies ist für die Praxis kein brauchbarer Ausgangspunkt für eine Systementwicklung, daher wird folgende Systemeigenschaft eingeführt.

Definition 3.17 (Realisierbarkeit)
Ein System S ist *schwach realisierbar* bzw. *stark realisierbar*, wenn S schwach kausal bzw. stark kausal ist und es eine Funktion f gibt mit $f \subseteq \mathcal{R}_S$. Ein System wird als *realisierbar* bezeichnet, wenn es zumindest schwach realisierbar ist. □

Jedes stark realisierbare System ist auch schwach realisierbar. Eine realisierbare Spezifikation garantiert, dass es zu jedem Eingabestrom mindestens einen gültigen Ausgabestrom gibt. Sie ist damit widerspruchsfrei.

[1] Hierin geht ein, dass es in eingebetteten Systemen keinen echten Zufall gibt, sondern das Zustandekommen jeder Ausgabe theoretisch nachvollziehbar und reproduzierbar ist.

Aus der Definition folgt unmittelbar, dass jedes existierende, eingebettete System realisierbar ist. Das trifft selbst auf ein ausgefallenes System zu, welches nicht mehr reagiert, siehe Abschnitt 3.3.1.

Bei Systemspezifikationen soll ausgeschlossen sein, dass sie widersprüchlich oder nicht kausal sind. Daher wird im Folgenden stets gefordert, dass Systemspezifikationen realisierbar sind.

3.3.4 Verhaltensverfeinerung

Bei der Entwicklung eines Systems tritt eine Folge an Spezifikationsständen auf und mündet schließlich in eine Realisierung. Am Beispiel der modularen Entwicklung sind dies die Systemspezifikation, die Architekturbeschreibung, alle Subsystemrealisierungen zusammen und das System. Zur Beschreibung des korrekten Übergangs von einem Spezifikationsstand zum nächsten bzw. zur Realisierung wird der Begriff der Verhaltensverfeinerung eingeführt.

Definition 3.18 (Verhaltensverfeinerung)
Seien S_1 und S_2 beliebige Systeme. S_2 ist eine *Verhaltensverfeinerung* von S_1, kurz $S_1 \rightsquigarrow S_2$, wenn beide Systeme die gleiche Schnittstelle haben und jedes Verhalten von S_2 auch ein Verhalten von S_1 ist. Formal:

$$(S_1 \rightsquigarrow S_2) \quad \Leftrightarrow \quad (\mathcal{R}_{S_1} \supseteq \mathcal{R}_{S_2})$$

□

Definition 3.19 (Verhaltensäquivalenz)
Seien S_1 und S_2 beliebige Systeme. S_1 und S_2 sind verhaltensäquivalent, kurz $S_1 \leftrightsquigarrow S_2$, wenn $S_1 \rightsquigarrow S_2$ und $S_2 \rightsquigarrow S_1$. □

Bei realisierbaren Systemen bedeutet eine Verhaltensverfeinerung lediglich die Reduzierung des Nichtdeterminismus. Man kann also die Verhaltensverfeinerung verwenden, um Spezifikationsfreiräume zu nutzen.

Sind S, S', S'' aufeinander folgende Spezifikationsstände eines Systems und R die Realisierung am Ende der Entwicklung, dann sollte gelten:

$$S \rightsquigarrow S' \rightsquigarrow S'' \rightsquigarrow R$$

Da $\rightsquigarrow$ transitiv ist, wird dadurch sichergestellt, dass R ein korrektes Verhalten im Sinne der Spezifikation S aufweist, also $S \rightsquigarrow R$.

FOCUS kennt auch die Konzepte der *Schnittstellenverfeinerung* und *bedingten Verhaltens-* bzw. *Schnittstellenverfeinerung*, die jedoch in dieser Arbeit nicht benötigt und daher nicht weiter vorgestellt werden.

3.4 Komposition von Systemen

Definition 3.20 (Komposition von Systemen)
Seien S_1 und S_2 beliebige Systeme. Dann bezeichnet $S_1 \otimes S_2$ die *Komposition* von S_1 und S_2 und ist selbst wiederum ein System. Gleichnamige Aus- und Eingabekanäle werden durch lokale Kanten miteinander verbunden. Dabei wird vorausgesetzt, dass die Namen aller Ausgabekanäle beider Systeme verschieden und die Alphabete der verbundenen Kanäle gleich sind. Die Systemschnittstelle der Komposition besteht aus den nicht-gebundenen Kanälen beider Systeme. □

Der Kompositionsoperator $\otimes$ ist kommutativ. Sind die Namen aller Eingabekanäle verschieden, ist der Operator auch assoziativ. Bei der Komposition mehrerer Systeme kann dann auf die Klammerung verzichtet werden. Eine formale Definition der Komposition ist in FOCUS zu finden.

Beispiel 3.9 Abbildung 3.3 zeigt die Komposition dreier Subsysteme S_1, S_2 und S_3. Das dadurch entstehende System S hat die Eingabekanäle a und b sowie die Ausgabekanäle g und h. Alle anderen Kanäle der Subsysteme werden lokal gebunden und sind nicht nach außen sichtbar. □

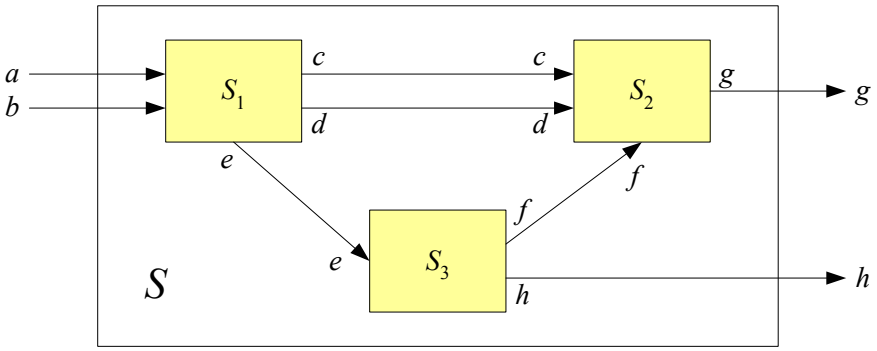


Abbildung 3.3: Komposition von drei Subsystemen

Definition 3.21 (Verhalten einer Komposition)
Seien S_1 und S_2 beliebige Systeme. Das *Verhalten* der Komposition $S = S_1 \otimes S_2$ wird definiert durch

$$\mathcal{R}_S \stackrel{\text{def}}{=} \{(i,o) \in I_S{}^{\infty} \times O_S{}^{\infty} \mid \exists\, l \in L^{\infty} : ((\Pi_{i_{S_1}}.i, \Pi_{i_{S_1}}.l), (\Pi_{o_{S_1}}.o, \Pi_{o_{S_1}}.l)) \in \mathcal{R}_{S_1} \wedge ((\Pi_{i_{S_2}}.i, \Pi_{i_{S_2}}.l), (\Pi_{o_{S_2}}.o, \Pi_{o_{S_2}}.l)) \in \mathcal{R}_{S_2} \}$$

wobei i_{S_1}, o_{S_1}, i_{S_2}, o_{S_2} die Namen der Ein- bzw. Ausgabekanäle von S_1 bzw. S_2 sind und $\Pi_A.t$ die Projektion auf die Einträge mit Namen A des Tupels t. Der Bezeichner

l steht für die Ströme der lokalen Kanten und L^{∞} ist die Menge aller Ströme der lokalen Kanten. □

Beispiel 3.10 Seien M_a, M_b, M_c, M_d, M_e, M_f, M_g und M_h die Alphabete der Kanäle aus obigem Beispiel. Dann ergibt sich:

$$I_S{}^{\infty} = M_a{}^{\infty} \times M_b{}^{\infty}$$

$$O_S{}^{\infty} = M_g{}^{\infty} \times M_h{}^{\infty}$$

$$L^{\infty} = M_c{}^{\infty} \times M_d{}^{\infty} \times M_e{}^{\infty} \times M_f{}^{\infty}$$

$$\mathcal{R}_S = \{\ ((a,b),(g,h)) \in I_S{}^{\infty} \times O_S{}^{\infty}\ \mid\ \exists (c,d,e,f) \in L^{\infty} : (a,b,c,d,e) \in \mathcal{R}_{S_1} \wedge\ (c,d,f,g) \in \mathcal{R}_{S_2}\ \wedge\ (e,f,h) \in \mathcal{R}_{S_3}\ \}$$

□

Der Kompositionsoperator $\otimes$ ist außerdem monoton bezüglich der Verhaltensverfeinerung:

$$S_1 \rightsquigarrow S_1' \ \wedge\ S_2 \rightsquigarrow S_2' \quad \Rightarrow \quad (S_1 \otimes S_2) \rightsquigarrow (S_1' \otimes S_2')$$

Die Komposition definiert ein System bottom-up, d. h. das Verhalten der Komposition ist vollständig durch das Verhalten ihrer Subsysteme festgelegt.

3.5 Grenzen des Systemmodells

Das in diesem Kapitel vorgestellte Systemmodell beschränkt sich auf eine rein funktionale Beschreibung von Systemen. Nicht darstellbar sind hingegen nicht-funktionale Anforderungen wie Bedienbarkeit oder Zuverlässigkeit. Zur Beschreibung der Zuverlässigkeit fehlen dem Systemmodell beispielsweise die Angabe eines Operationsprofils, d. h. die relativen Aufrufhäufigkeiten der einzelnen Systemfunktionen sowie die Versagenswahrscheinlichkeiten der Systemfunktionen.

Ein weiterer Punkt ist die Zeitdarstellung in FOCUS. Die Beschreibung der zeitlichen Abläufe erfolgt anhand diskreter Zeittakte, die den kontinuierlichen Zeitverlauf in Intervallabschnitte zerlegen. Mit dieser Zeitabstraktion ist es in FOCUS nicht direkt möglich, beliebige reelle Zeitabstände zu messen. Eine Erweiterung auf zeitkontinuierliche Ströme sowie diskrete Ströme auf Basis einer kontinuierlichen Zeit ist in [Bro01] beschrieben. Damit lässt sich auch der Bezug zur realen Zeit herstellen, um beispielsweise einen Timeout von 100 ms zu modellieren.

Für das in dieser Arbeit benötigte Systemmodell ist FOCUS aber eine ideale Basis und hat sich als ausreichend mächtig und äußerst kompakt erwiesen.

4 Integration aus formaler Sicht

In diesem Kapitel wird die Integration aus formaler Sicht beschrieben. Wichtige Eigenschaften und Begriffe werden mit Hilfe des eingeführten Systemmodells charakterisiert.

4.1 Phasen der modularen Entwicklung

Gemäß Abschnitt 2.3.1 besteht die modulare Entwicklung im Wesentlichen aus drei Phasen: Zerlegen in Subsysteme, Realisieren der Subsysteme, Integrieren zum System. Formal stellen sich diese drei Phasen wie folgt dar.

Phase 1: Zerlegen in Subsysteme Eine gegebene Systemspezifikation S wird zerlegt in mehrere Subsystemspezifikationen $S_1, \ldots, S_n$. Die Zerlegung ist korrekt, wenn die Subsystemspezifikationen zusammen ein Systemverhalten beschreiben, das der ursprünglichen Systemspezifikation entspricht, also

$$S \rightsquigarrow (S_1 \otimes \cdots \otimes S_n) \tag{4.1}$$

Fehler in dieser Phase werden als *Zerlegungsfehler* bezeichnet.

Phase 2: Realisieren der Subsysteme Die einzelnen Subsysteme werden in dieser Phase unabhängig voneinander realisiert, und aus jeder Subsystemspezifikation S_i entsteht eine Realisierung R_i. Die Realisierungen sind korrekt, wenn jedes einzelne realisierte Subsystem seiner Spezifikation entspricht, also

$$\forall\, i \in [\,1 \ldots n\,] : S_i \rightsquigarrow R_i \tag{4.2}$$

Fehler in dieser Phase werden als *Realisierungsfehler* bezeichnet.

Phase 3: Integrieren zum System Schließlich werden die Subsysteme R_i zum System R integriert. Seien $R'_1, \ldots, R'_m$ die Subsysteme, aus denen R besteht, d. h. $R = R'_1 \otimes \cdots \otimes R'_m$. Die Integration ist korrekt, wenn alle n Subsysteme integriert werden, also

$$n = m \quad \text{und} \quad \forall\, i \in [\,1 \ldots n\,] : R_i = R'_i \tag{4.3}$$

Fehler in dieser Phase werden als *Integrationsfehler* bezeichnet.

Die Integration geht in der Praxis über die Komposition im Systemmodell hinaus. Beispielsweise kann ein Subsystem vergessen oder eine falsche Version integriert worden sein, was durch den Kompositionsoperator allein nicht darstellbar ist. Die Komposition stellt nur die ideale Integration dar.

Sind alle drei Phasen fehlerfrei, so entsteht ein korrektes Produkt bezüglich der Systemspezifikation, wie folgende Überlegung zeigt: Aus (4.2) und der Monotonie des Kompositionsoperators $\otimes$ bezüglich der Verhaltensverfeinerung (Abschnitt 3.4) folgt

$$(S_1 \otimes \cdots \otimes S_n) \rightsquigarrow (R_1 \otimes \cdots \otimes R_n).$$

Zusammen mit (4.1) und (4.3) ergibt sich

$$S \rightsquigarrow (S_1 \otimes \cdots \otimes S_n) \rightsquigarrow (R_1 \otimes \cdots \otimes R_n) = (R'_1 \otimes \cdots \otimes R'_m) = R.$$

Da die Verhaltensverfeinerung transitiv ist (Abschnitt 3.3.4), erhält man

$$S \rightsquigarrow R,$$

also ein korrektes Produkt.

Im Unterschied zur Beschreibung in Abschnitt 2.3.1 wird in der formalen Sicht auf die explizite Angabe der Architektur verzichtet, weil diese im Systemmodell implizit über die Namensgleichheit von Kanälen und die Auflistung der Subsysteme gegeben ist.

4.2 Formalisierung wichtiger Begriffe

4.2.1 Fehler-Begriffe

Der Fehler-Begriff wurde bereits in Abschnitt 2.2 informell eingeführt. Hier folgt die formale Definition. Sie entspricht der in [Bre01].

Definition 4.1 (Versagen)
Ein *Versagen* eines Systems R gegenüber seiner Spezifikation S ist ein fehlerhaftes Verhalten $(i, o) \in \mathcal{R}_R \setminus \mathcal{R}_S$. Die *Menge aller Versagen* ist $\mathcal{R}_R \setminus \mathcal{R}_S$. □

Diese Definition ergibt sich unmittelbar aus der informellen Definition 2.5. $\mathcal{R}_R$ stellt das tatsächliche Verhalten des (realisierten) Systems R dar und $\mathcal{R}_S$ das von einer Spezifikation S vorgeschriebene ideale Verhalten. Ein Versagen ist ein abweichendes Verhalten des Systems von der Spezifikation und damit ein Ein-/Ausgabe-Paar in $\mathcal{R}_R \setminus \mathcal{R}_S$.

Weist ein System kein Versagen auf, ist die Menge aller Versagen leer. Es gilt dann $S \rightsquigarrow R$ (siehe Verhaltensverfeinerung in Abschnitt 3.3.4).

Voraussetzung für ein Versagen ist immer eine Fehlerursache, d. h. eine Abweichung des tatsächlichen vom beabsichtigten Aufbau eines Systems. Diese Abweichung im Aufbau eines Systems wird formal als *Modifikation* des Systemverhaltens beschrieben, siehe nachfolgende Definition.

Definition 4.2 (Modifikation)
Sei S ein System mit der Menge aller möglichen Eingabeströme I_S^{∞} und der Menge aller möglichen Ausgabeströme O_S^{∞}. Eine *Modifikation* $\mathcal{M}$ eines Systems S ist ein Paar

$$\mathcal{M} \stackrel{\text{def}}{=} (E, F)$$

mit $E \subseteq I_S^{\infty} \times O_S^{\infty}$ und $F \subseteq I_S^{\infty} \times O_S^{\infty}$. Das dadurch modifizierte System wird mit

$$S \triangle \mathcal{M}$$

bezeichnet und hat die Verhaltensrelation

$$\mathcal{R}_{S \triangle \mathcal{M}} \stackrel{\text{def}}{=} (\mathcal{R}_S \setminus E) \cup F.$$

□

Eine Modifikation $\mathcal{M}$ beschreibt also eine Verhaltensänderung eines Systems S. Aus der ursprünglichen Verhaltensrelation $\mathcal{R}_S$ werden die Ein-/Ausgabetupel E entfernt und die Ein-/Ausgabetupel F hinzugefügt. Neutrale Modifikationen, die keine Änderung bewirken, sind u. a. alle (A, A) mit $A \subseteq \mathcal{R}_S$, insbesondere $(\varnothing, \varnothing)$. Die Ausdruckskraft der hier eingeführten Modifikation ist groß genug, um damit alle möglichen Verhaltensänderungen bei gleich bleibender Schnittstelle auszudrücken. Sinnvollerweise wählt man E und F disjunkt. Abbildung 4.1 stellt die Modifikation des Verhaltens graphisch dar.

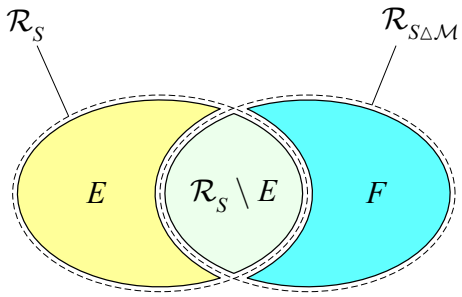


Abbildung 4.1: Modifikation eines Systems

Mit Hilfe der Modifikation wird der Begriff Fehlerursache nun wie folgt formal definiert.

Definition 4.3 (Fehlerursache)
Liegt ein Versagen gemäß Definition 4.1 vor, so ist die *Fehlerursache* eine Modifikation $\mathcal{M}$, die das fehlerhafte System $S \triangle \mathcal{M}$ bezüglich seiner Spezifikation S beschreibt. □

Die als Fehlerursache bezeichnete Modifikation $\mathcal{M}$ ist nicht notwendigerweise eindeutig. Im Allgemeinen gibt es mehrere unterschiedliche Modifikationen, die eine unerlaubte Abweichung eines Systems von seiner Spezifikation ausdrücken können. Eine solche Modifikation ist stets $(\mathcal{R}_S, \mathcal{R}_{S \triangle \mathcal{M}})$.

Eine Fehlerursache umfasst dabei in der Regel gleich mehrere Ein- und Ausgabetupel, um die sich die Spezifikation und das fehlerhafte System unterscheiden. Dies entspricht auch der Praxis, wo ein Fehler in einer Programmzeile meistens bei mehreren unterschiedlichen Eingaben zu einer falschen Ausgabe führt.

Auf die formale Definition von Fehlzustand wird hier verzichtet, da im Folgenden nur die Black-Box-Sicht verwendet wird und die Beschreibung innerer Zustände nicht nötig ist.

4.2.2 Architektur- und Integrationsfehler

In Abschnitt 2.3 wurde bereits die Grundidee der Integration vorgestellt. Diese wird nun mit Hilfe des Systemmodells formal gefasst.

Definition 4.4 (Integration)
Gegeben seien beliebige Systeme $S_1, \ldots, S_n$. Die *Integration* dieser Systeme zu einem System S ist die Komposition $S = (S_1 \otimes \cdots \otimes S_n)$. □

Die informelle Definition 2.9 des Begriffs Integration bezieht sich explizit auf realisierte Systeme und unterscheidet sich damit von der *virtuellen Integration* bei Spezifikationen. Die formale Definition 4.4 der Integration abstrahiert hiervon und führt beides auf die Komposition zurück.

Aus den informellen Definitionen der Begriffe Architekturfehler und Integrationsfehler in Abschnitt 2.3.2 ergeben sich die folgenden, formalen Definitionen.

Definition 4.5 (Architekturfehler)
Gegeben sei eine Systemspezifikation S sowie eine Architektur mit den Subsystemspezifikationen $S_1, \ldots, S_n$. Ein *Architekturfehler* liegt vor, wenn

$$S \not\rightarrow (S_1 \otimes \cdots \otimes S_n) \quad \text{oder} \quad \text{ein } S_i \text{ nicht realisierbar ist.}$$

□

Definition 4.6 (Integrationsfehler)
Gegeben seien die realisierten Subsysteme $R_1, \ldots, R_n$ sowie ein integriertes System $R = (R'_1 \otimes \cdots \otimes R'_m)$. Ein *Integrationsfehler* liegt vor, wenn

$$n \neq m \quad \text{oder} \quad \exists\, i \in [\,1 \ldots n\,] : R_i \neq R'_i$$

□

Ein Architekturfehler liegt vor, wenn die Phase der Zerlegung eines Systems in Subsysteme (siehe Abschnitt 4.1) fehlerhaft ist oder die erhaltene Architektur nicht realisierbar ist. Die Realisierbarkeit der Subsystemspezifikationen (siehe Abschnitt 3.3.3) wird als eine elementare Eigenschaft einer Architektur angesehen, weil eine Architektur eine sinnvolle Basis für eine spätere Realisierung sein soll.

Ein Integrationsfehler liegt vor, wenn die Korrektheitseigenschaft der Integration eines modularen Systems verletzt ist.

4.2.3 Verbundfehler

Liegt ein Architekturfehler vor, will man den Fehler lokalisieren. Im einfachsten Fall ist der Fehler auf ein einzelnes, fehlerhaftes Subsystem zurückzuführen. Ist allerdings kein einzelnes Subsystem eindeutig als fehlerhaft identifizierbar, liegt die Fehlerursache im Zusammenspiel mehrerer Subsysteme. Ein solcher Architekturfehler wird im Folgenden *Verbundfehler* genannt.

Definition 4.7 (Fehlerkorrektur)
Gegeben seien eine Systemspezifikation S und die Subsysteme $S_1, \ldots, S_n$ mit

$$S \not\rightsquigarrow (S_1 \otimes \ldots \otimes S_n).$$

Eine *Fehlerkorrektur* von $S_1, \ldots, S_n$ bezüglich S ist ein n-Tupel von Modifikationen

$$(\mathcal{M}_1, \ldots, \mathcal{M}_n),$$

so dass gilt:

$$S \rightsquigarrow ((S_1 \triangle \mathcal{M}_1) \otimes \ldots \otimes (S_n \triangle \mathcal{M}_n))$$

□

Eine Fehlerkorrektur ist also eine Modifikation der Subsysteme, die einen Architekturfehler behebt. Abbildung 4.2 veranschaulicht den Sachverhalt.

Definition 4.8 (Verbundfehler)
Seien S und $S_1, \ldots, S_n$ wie oben. Ein *Verbundfehler* liegt vor, wenn es zu jedem Subsystem S_i, $i \in [\,1 \ldots n\,]$, eine Fehlerkorrektur $(\mathcal{M}_1, \ldots, \mathcal{M}_n)$ gibt, die das Subsystem S_i unverändert lässt, d. h. $S_i \nsim (S_i \triangle \mathcal{M}_i)$. □

Liegt ein Verbundfehler vor, kann man immer eine Fehlerkorrektur finden, die unter Umgehung eines beliebigen Subsystems den Fehler behebt. Der Verbundfehler entsteht damit erst durch Kombination mehrerer Subsysteme. Jedes Subsystem einzeln betrachtet ist hingegen fehlerfrei.

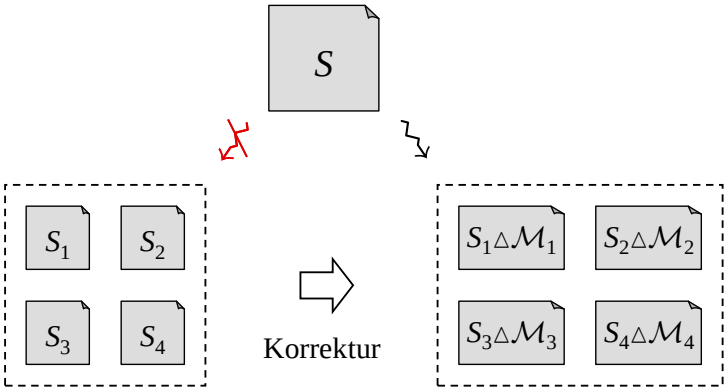


Abbildung 4.2: Veranschaulichung des Begriffs der Fehlerkorrektur

Im Unterschied zu Abschnitt 2.4 kommt der Begriff des Verbundfehlers ohne die Betrachtung der jeweiligen idealen Artefakte aus. Es wird lediglich die Zerlegung einer Systemspezifikation betrachtet. Wäre die ideale Architektur bekannt, gäbe es keine Verbundfehler, da die fehlerhaften Subsysteme dann eindeutig als Abweichung von den idealen Subsystemen identifizierbar wären.

Ein wesentliches Merkmal des Verbundfehlers ist, dass keines der beteiligten Subsysteme einzeln als fehlerhaft nachgewiesen werden kann, sondern nur mehrere oder alle zusammen als fehlerhaft gelten. Für die Praxis bedeutet das, dass Verbundfehler nicht mittels Modultests nachgewiesen werden können. Erst eine gesamtheitliche Sicht kann diese Architekturfehler aufdecken.

4.3 Klassifikation von Architekturfehlern

Ein Architekturfehler umfasst alle Arten von Fehlern bei der Zerlegung eines Systems. Die möglichen Fehlerursachen dafür können vielfältig sein und werden in diesem Abschnitt nach unterschiedlichen Gesichtspunkten klassifiziert. Diese Klassifikationen[1] sollen zum einen dazu dienen, die Integrationsproblematik systematisch zu untersuchen und damit besser zu verstehen, zum anderen werden sie später zur Einordnung von empirischen Fehlerdaten verwendet.

4.3.1 Abstraktionsebenen der Informationsdarstellung

Die Kommunikation zwischen den Steuergeräten spielt sich auf unterschiedlichen Ebenen ab. *Physikalisch* betrachtet werden Spannungsverläufe verwendet, um Informa-

[1] Der Begriff *Klassifizierung* bezeichnet das Erstellen einer Unterteilung, während eine *Klassifikation* das Ergebnis einer Klassifizierung ist.

tionen zwischen Sender und Empfänger auszutauschen und Signale durch Rechennetze in einen Prozessor zu schicken. Abstrahiert man von den Spannungsverläufen und idealisiert diese zu zeitlichen Folgen von 0- und 1-Bits, so gelangt man zu einer Verhaltensbeschreibung in *Maschinendarstellung*. Der Datenaustausch erfolgt anhand von Bitsequenzen, und die Datenverarbeitung in einem Prozessor bildet in jedem Schritt ein Eingabetupel von Bits auf ein Ausgabetupel von Bits ab. Interpretiert man Bittupel als Zahlen oder Zeichen, abstrahiert man wiederum von der Maschinendarstellung und gelangt zur *Semantikdarstellung*, also der Bedeutung der Bitmuster.

Im Folgenden werden die Abstraktionsebenen einzeln genauer beschrieben.

Semantikdarstellung Die Semantikdarstellung beschreibt die von einem System verarbeitete Information, ohne dabei auf maschinenspezifische Details einzugehen. Die Information wird selbstbeschreibend dargestellt mit Hilfe mathematischer Zahlenmengen (z. B. $\mathbb{N}$, $\mathbb{R}$) und abstrakter Aussagen. Die Kommunikation zwischen zwei Systemen benötigt keine Zeit, es wird lediglich die zeitliche Reihenfolge von Nachrichten beschrieben. Nachrichten benötigen keinen Speicherplatz und können beliebig groß sein.

Maschinendarstellung In der Maschinendarstellung werden die Informationen als digitale Bitmuster repräsentiert. Aufgrund von endlichem Speicher erfolgt eine Quantisierung auf endlich viele Bits. Zudem wird von einer konkreten Hardware ausgegangen, wodurch die Betrachtung der Berechnungs- und Kommunikationsdauer möglich wird. Das bringt Nebenläufigkeit mit sich und erfordert ggf. die Synchronisation von Tasks. Seiteneffekte, die durch gemeinsam genutzte Ressourcen (Prozessor, Arbeitsspeicher, Kommunikation) entstehen, sind zu berücksichtigen. Allerdings wird die Hardware der Maschine als perfekt angesehen, d. h. sie unterliegt keinen physikalischen Störungen.

Physikalische Darstellung Die physikalische Darstellung betrachtet die dargestellte Information als analoge Spannungsverläufe. Wichtige Aspekte in dieser Darstellung sind Signalverzerrungen und -störungen. Hier wird das reale Verhalten der Hardware beschrieben. Darüber hinaus wird hier die Spannungsversorgung beschrieben; beispielsweise ist der verwendete Massepunkt in der Karosserie wichtig, um das passende Spannungspotential zu erreichen.

Die Unterscheidung mehrerer Abstraktionsebenen begründet sich auf unterschiedliche Sichten auf ein und denselben Vorgang. Die Ebenen sind eng miteinander verbunden und (im Allgemeinen nicht eindeutig) ineinander überführbar. Analog zum ISO-OSI-Schichtenmodell [ISO94, Tan03] werden in den Ebenen unterschiedliche Aspekte der Datendarstellung und des Zeitverhaltens beschrieben. Im Unterschied dazu bauen hier die Ebenen aber nicht wie beim ISO-OSI-Schichtenmodell aufeinander auf, sondern stellen die unterschiedlichen Grade der Abstraktion dar. Die

drei genannten Abstraktionsebenen werden zur Klassifikation von Architekturfehlern verwendet.

Beispiel 4.1 Betrachtet wird das System *Bremssteuergerät* aus Abbildung 4.3. Es nimmt einen Verzögerungswunsch aus dem Bereich $[0\,\frac{\mathrm{m}}{\mathrm{s}^2};\ 10\,\frac{\mathrm{m}}{\mathrm{s}^2}]$ entgegen und übersetzt diesen linear in eine Aktorspannung im Bereich $[0\,\mathrm{V};\ 40\,\mathrm{V}]$, die an die Bremsaktorik weitergeleitet wird. Die Ein- und Ausgabesignale werden im Folgenden jeweils gemäß den einzelnen Abstraktionsebenen dargestellt.

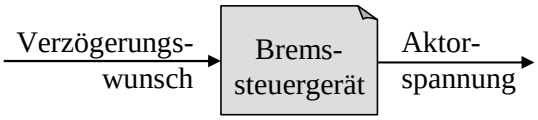


Abbildung 4.3: Das System *Bremssteuergerät*

In der *Semantikdarstellung* wird nur die Bedeutung der Information beschrieben. Die Stromtypen lauten also

$$I_1 = \left\{\text{Verzögerungswunsch von } x\,\tfrac{\mathrm{m}}{\mathrm{s}^2} \mid x \in [0;\ 10]\right\}$$

$$O_1 = \{\text{Aktoransteuerung von } y \mid y \in [0;\ 1]\}$$

In der *Maschinendarstellung* sollen die Eingabewerte als 8-Bit-Werte kodiert werden.

$$I_2 = \mathbb{B}_8$$

Der Übergang zwischen der Semantikdarstellung und der Maschinendarstellung wird mittels einer Repräsentations- bzw. einer (dazu inversen) Interpretationsrelation dargestellt. Die Beschreibung dieses Zusammenhangs ist wichtig, da die Bitdarstellung allein keine Semantik hat. Für die Repräsentationsrelation Rep_{12}^I gilt:

$$Rep_{12}^I : I_1{}^{\infty} \to I_2{}^{\infty}$$

$$\forall\, i \in I_1{}^{\infty}\ \forall\, k \in \mathbb{N}_+ :$$

$$Rep_{12}^I(i)(k) = \begin{cases} Bitrep_8(\lfloor \frac{255}{10} x \rfloor) & \text{falls } i(k) = \text{Verzögerungswunsch von } x\,\frac{\mathrm{m}}{\mathrm{s}^2} \\ \surd & \text{falls } i(k) = \surd \end{cases}$$

wobei

$$Bitrep_8(n) = \begin{cases} (0,0,0,0,0,0,0,0) & \text{falls } n \leq 0 \\ (1,1,1,1,1,1,1,1) & \text{falls } n \geq 2^8 - 1 \\ (b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0) & \text{sonst, mit } n = \sum_{i=0}^{7} 2^i b_i \end{cases}$$

Der Term $\left\lfloor \frac{255}{10} x \right\rfloor$ bildet das reelle Intervall $[0;\ 10]$ auf 256 äquidistante Punkte mit der Schrittweite $\frac{10}{255} \approx 0{,}039$ ab. Die Ausgabewerte O_1 werden als analoge Spannung übertragen, daher entfällt hier die digitale Maschinendarstellung.

Die *physikalische Darstellung* beschreibt die Signale als analoge Spannungsverläufe. Für die Ein- und Ausgabewerte werden folgende Spannungswerte verwendet:

$$I_3 = \{0\,\mathrm{V},\ 2\,\mathrm{V}\}$$

$$O_3 = [0\,\mathrm{V};\ 40\,\mathrm{V}]$$

Läuft die Kommunikation wie hier beim Eingabestrom digital ab, so bietet es sich an, auf einen passenden Kommunikationsstandard zurückzugreifen. In diesem Beispiel soll Controller Area Network (CAN) zum Einsatz kommen. Die Datenübertragung erfolgt durch zwei Kabel, genannt CAN_H und CAN_L, wobei die Bitinformation anhand der Potentialdifferenz $V_{\mathrm{diff}} = V_{\mathrm{CAN\_H}} - V_{\mathrm{CAN\_L}}$ wie folgt kodiert wird:

V_{diff}	Bus-Zustand	Bit-Bedeutung
0 V	dominant	0
2 V	rezessiv	1

Die Repräsentationsrelation Rep^I_{23}, die den Übergang von der Maschinendarstellung zur physikalischen Darstellung beschreibt, wird durch die CAN-Spezifikation in ISO 11898 [ISO03] festgelegt. Die physikalische Darstellung der Ausgabedaten erfolgt mittels der Repräsentationsabbildung Rep^O_{13}:

$$Rep^O_{13} : O_1{}^{\infty} \to O_3{}^{\infty}$$

$$\forall\, o \in O_1{}^{\infty}\ \forall\, k \in \mathbb{N}_+ :$$

$$Rep^O_{13}(o)(k) = \begin{cases} 40y & \text{falls } o(k) = \text{Aktoransteuerung von } y \\ \surd & \text{falls } o(k) = \surd \end{cases}$$

Die Repräsentationsabbildung Rep^O_{13} bildet das Intervall $[0;\ 1]$ der Semantikdarstellung linear auf das Intervall $[0\,\mathrm{V};\ 40\,\mathrm{V}]$ der physikalischen Darstellung ab. □

Es sei darauf hingewiesen, dass dieses Beispiel der Einfachheit halber auf jegliche Störungsabsicherung verzichtet und eine idealisierte Spannungsumsetzung beschreibt. Für die Datendarstellung in der physikalischen Darstellung wurde hier ebenfalls die Notation des in Kapitel 3 eingeführten Systemmodells verwendet, das allerdings für analoge Spannungsverläufe nicht entwickelt wurde. Streng genommen müssten die Spannungsverläufe als zeitkontinuierliche Funktionen beschrieben werden, was hier zur Vereinfachung entfällt.

Modellierung von Störungen

Bei der physikalischen Darstellung wird das reale Verhalten der Hardware beschrieben. Dazu zählen insbesondere auch Störeinflüsse wie elektromagnetische Strahlung, elektrische und magnetische Felder, Temperatur, Erschütterung und Spannungsschwankungen in der Energieversorgung. Da sich diese Einflüsse auf die Ausgabe eines Systems auswirken können, sind sie als implizite Eingaben zu berücksichtigen.

Eine Möglichkeit, diese Störungen im Systemmodell zu berücksichtigen, ist die Beschreibung durch nichtdeterministisches Verhalten. Einem konkreten Eingabestrom werden dann mehrere alternative Ausgabeströme zugeordnet, die sich nur durch *Rauschen* voneinander unterscheiden.

Bei kleinen Störeinflüssen ist diese Vorgehensweise durchaus angebracht. Sollen allerdings größere Störungen betrachtet werden, z. B. der vorübergehende Ausfall eines Steuergerätes durch Unterspannung im Bordnetz, ist eine Beschreibung durch nichtdeterministisches Verhalten nicht mehr brauchbar. In diesem Fall empfiehlt es sich, die Störeinflüsse explizit als zusätzliche Eingangskanäle zu modellieren. Abhängig vom Störeingang kann dann ein entsprechendes Störverhalten des Systems beschrieben werden.

Sollen Störeinflüsse auf dem Übertragungsweg zwischen zwei Systemen, z. B. durch magnetische Felder bei Metallverbindungen, modelliert werden, ist die Kommunikationsverbindung durch ein neues Subsystem zu ersetzen. Dann kann wie schon beschrieben jedes beliebige Störverhalten modelliert werden.

4.3.2 Interaktionseigenschaften

Der Informationsaustausch bei der Interaktion zwischen mehreren (nebenläufigen) Subsystemen erfordert die Einhaltung eines gemeinsamen Protokolls. Wesentliche Eigenschaften hierbei sind ein gemeinsames Verständnis über die Datendarstellung und ein abgestimmtes Zeitverhalten. Nichteinhalten dieser Eigenschaften führt zu einem *Datenfehler* bzw. *Zeitfehler*.

Definition 4.9 (Datenfehler)

Ein *Datenfehler* liegt vor, wenn zwei Systeme einen ausgetauschten Wert unterschiedlich interpretieren, d. h. als unterschiedliche Informationen auffassen. □

Beispiel 4.2

- Die Zahl 4 wird interpretiert als 4 °C bzw. 4 °F (unterschiedliche physikalische Einheiten).
- Eine Bit- oder Byte-Sequenz wird interpretiert als Little-Endian- bzw. Big-Endian-Darstellung.

- Ein gesendeter Wert ist nicht im Wertebereich des Empfängers, er wird einmal als gültiger und einmal als ungültiger Wert interpretiert.
- Die Prüfsumme eines Datenpakets wird von Sender und Empfänger unterschiedlich berechnet und das Datenpaket vom Empfänger daher als verfälscht interpretiert.
- Das Vorzeichen einer Zahl wird als *links* bzw. als *rechts* interpretiert.
- Die vom Empfänger erwartete Genauigkeit eines übertragenen Wertes ist höher als die vom Sender bereitgestellte.

□

Definition 4.10 (Zeitfehler)
Ein *Zeitfehler* liegt vor, wenn die tatsächliche zeitliche Abfolge von gesendeten bzw. empfangenen Nachrichten von der zeitlichen Sollabfolge abweicht. □

Zeitfehler offenbaren sich insbesondere dann, wenn zwei miteinander kommunizierende Subsysteme die zeitliche Abfolge von Nachrichten unterschiedlich interpretieren, wie in folgendem Beispiel dargestellt.

Beispiel 4.3

- Die Reihenfolge zweier Nachrichten ist anders als vom Empfänger erwartet. Im Extremfall kann dies zu einem Deadlock führen, beispielsweise wenn zwei Systeme gegenseitig auf eine Nachricht warten.
- Der zeitliche Abstand zweier Nachrichten wird unterschiedlich interpretiert z. B. als Zeitüberschreitung (Timeout) bzw. als keine Zeitüberschreitung.
- Das Alter einer Information, d. h. die vergangene Zeit seit Entstehen der Information, ist größer als erwartet.

□

Daten- und Zeitfehler schließen sich nicht aus, sie können auch kombiniert auftreten. Zudem sind sie nicht immer klar voneinander unterscheidbar, z. B. wenn ein Zeitstempel als Wert übertragen wird (siehe *The Dual Role of Time* in [Kop97]).

5 Stand der Praxis

Dieses Kapitel soll helfen, die Integrationsproblematik besser zu verstehen, und eine Grundlage für die Bewertung der Praxistauglichkeit von Maßnahmen in Kapitel 7 bilden.

Im Folgenden wird der aktuelle Stand der Praxis in der Softwareentwicklung für eingebettete Systeme im Automobil untersucht. Dabei soll beantwortet werden, welche Merkmale die Entwicklungsprozesse hinsichtlich der Systemintegration aufweisen und welche durch Architekturfehler verursachten Fehler in späten Entwicklungsphasen sowie nach der Auslieferung tatsächlich auftreten. Die Untersuchung erfolgt am Beispiel des Unternehmensbereichs Kraftfahrzeugtechnik der Robert Bosch GmbH.

5.1 Architekturbeschreibung

Betrachtet man die Software-Entwicklungsprozesse vor dem Hintergrund der Systemintegration, spielt die Zerlegung des Systems in seine Subsysteme und deren Zusammenspiel eine ganz besondere Rolle. Aus diesem Grund wird die im Entwicklungsprozess verwendete Architekturbeschreibung hier separat untersucht. Alle anderen Aspekte der Entwicklungsprozesse sind Gegenstand des nächsten Abschnitts.

In die empirische Untersuchung gehen die von den größten Automobil-Geschäftsbereichen erstellten Prozessbeschreibungen, Dokumentvorlagen und Checklisten, eingesetzte Softwarewerkzeuge und Dokumentationen ausgewählter, aktueller Produkte ein. Es wurde darauf geachtet, den verbreiteten Entwicklungsstand zu dokumentieren und weniger davon abweichende Sonderfälle in einzelnen Abteilungen. Häufig sind die ermittelten Ergebnisse unterschiedlicher Geschäftsbereiche sehr ähnlich. Daher beziehen sich die Angaben im Folgenden gleichermaßen auf alle genannten Geschäftsbereiche. In Ausnahmefällen wird explizit darauf hingewiesen.

Zudem erfolgen die Betrachtungen aus der Sicht auf einen Zulieferer und weniger auf einen Automobilhersteller. Das bedeutet, dass die hier betrachteten Systemarchitekturen nicht ein ganzes Automobil umfassen, sondern einen Teilverbund aus Steuergeräten oder auch nur ein einzelnes Steuergerät mit seinen Hardware- und Softwarekomponenten.

5.1.1 Typische Bestandteile der Architekturbeschreibung

Bei der Untersuchung von Architekturbeschreibungen aus der Praxis stellt sich zunächst die Frage: Welche Dokumente gelten als Bestandteil der Architekturbeschreibung? Prinzipiell kommen alle Dokumente in Frage, welche die einzelnen Subsysteme des Systems beschreiben und etwas über deren Schnittstellen, Verhalten und Zusammenspiel aussagen. Diese Eigenschaft würde aber beispielsweise auch der Quellcode der Softwareimplementierung aufweisen, der aber die Realisierung selbst darstellt und nicht eine Beschreibung der Realisierung. Deshalb wurden sinnvollerweise nur Dokumente berücksichtigt, die zeitlich *vor* der Realisierung des Produkts entstehen und nicht selbst Bestandteil der Realisierung sind.

Die untersuchten Architekturbeschreibungen enthielten deutlich detailliertere Angaben zur Systemschnittstelle und zum Systemverhalten – als System wird hier ein Steuergerät inklusive der daran angeschlossenen Sensoren und Aktoren gesehen – als bezüglich der inneren Struktur. Daher ist es sinnvoll, diese beiden Teile nicht zu vermischen, sondern separat voneinander zu betrachten.

Beschreibung des Systemverhaltens

Die Beschreibung des Systemverhaltens eines Steuergerätes besteht in der Praxis typischerweise aus einer CAN-Matrix und einer textuellen Verhaltensbeschreibung. Die CAN-Matrix beschreibt den Ausschnitt des Datenverkehrs eines CAN-Busses (siehe [ISO03]), bei dem das betroffene Steuergerät als Sender oder als Empfänger von Nachrichtenpaketen auftritt. Sie enthält üblicherweise folgende Informationen über den Aufbau eines Nachrichtenpakets:

- Name und eindeutige Identifikationsnummer des Pakets
- Länge des Pakets in Byte
- im Paket enthaltene Signale, jeweils mit
  - Bezeichner
  - zugeordneten Bits
  - Bedeutung spezieller Bitbelegungen, Fehlerwerte
  - Wertebereich mit physikalischer Einheit
  - Umrechnung zwischen Maschinendarstellung (z. B. uint) und interpretierter Darstellung
  - Genauigkeit (absolut und/oder relativ)
- Sendeart: zyklisch oder sporadisch
- Zykluszeit in ms

- maximal erlaubte Verzögerung (in ms) bei Eingabewerten
- maximale Zeit bis zur Umsetzung von Ausgabewerten
- sendendes Steuergerät
- empfangende Steuergeräte

Die textuelle Verhaltensbeschreibung erläutert die Bedeutung und Belegung der Signale in den Nachrichtenpaketen.

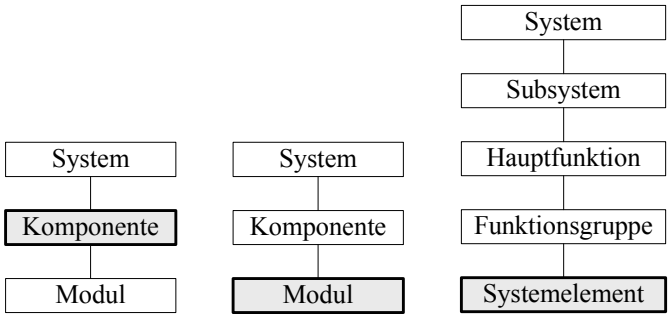
Beschreibung der inneren Struktur

Zur Beschreibung der inneren Struktur eines Systems gehört die Liste aller enthaltenen Subsysteme sowie das Zusammenspiel der Subsysteme.

In der Praxis erfolgt die Beschreibung der Subsysteme durchgängig auf einer einheitlichen Abstraktionsebene. Bei großen Systemen werden die vorhandenen Subsysteme durch logische Strukturebenen unterteilt, die selbst aber nur informell beschrieben sind und deren Schnittstellen nicht zusätzlich explizit angegeben werden. Abbildung 5.1 zeigt die unterschiedlichen Strukturebenen einiger Geschäftsbereiche. Hervorgehoben ist diejenige Ebene, auf der die detaillierteste Schnittstellenbeschreibung erfolgt. Geschäftsbereich 1 beschreibt die Schnittstellen auf Komponentenebene, bei der je Komponente in etwa zwischen fünf und 50 Ausgabekanäle bereitgestellt werden. Geschäftsbereich 2 verwendet zur Schnittstellenbeschreibung Module, die ein oder zwei Ausgabekanäle haben. Bei Geschäftsbereich 3 enthält das System eine größere Anzahl an Subsystemen – hier Systemelemente genannt – mit Schnittstellenbeschreibungen und hat dementsprechend viele Strukturebenen. Zu den Systemelementen gehören Softwaremodule, Steuergerätehardware, Sensoren und Steller samt Verbindungen zum Steuergerät. Der von Geschäftsbereich 3 verwendete Begriff *Subsystem* bezeichnet – im Unterschied zum Begriffsverständnis in dieser Arbeit – nur die Bestandteile einer bestimmten Strukturebene.

Die Schnittstellenbeschreibung eines Subsystems enthält in der Regel folgende Angaben:

- Bezeichner des Subsystems
- kurze, textuelle Beschreibung der Funktionalität
- zu den Schnittstellenvariablen
  - Name der Variablen
  - importierter oder exportierter Wert
  - Datentyp in Maschinendarstellung (z. B. uint16)
  - Wertebereich (minimaler und maximaler Wert)



(a) GB 1 (b) GB 2 (c) GB 3

Abbildung 5.1: Unterschiedliche Strukturebenen verschiedener Geschäftsbereiche. Hervorgehoben ist die Strukturebene mit der detailliertesten Schnittstellenbeschreibung.

  - physikalische Einheit der Werte

Bei einem Geschäftsbereich enthält die Schnittstellenbeschreibung zusätzlich folgende Angaben:

- zu den Schnittstellenvariablen
  - abstrakter Datentyp (z. B. $\mathbb{R}$), der zeitdiskret oder -kontinuierlich sein kann
  - Umrechnungsfunktion zwischen dem Datentyp in Maschinendarstellung und dem abstrakten Datentyp
- Vorgaben zur Aufrufreihenfolge der Softwaremodule in den Tasks

Die Kommunikationsstruktur der Architektur ergibt sich durchwegs implizit über Namensgleichheit der Ein- und Ausgabekanäle der Schnittstellen.

Verteilte Dokumentation
Die Beschreibung der Architektur eines Systems ist in der Praxis keineswegs in einem Dokument zu finden. Üblicherweise verteilt sich die Architektur über mehrere Dokumente, die häufig auch unterschiedliche Dateiformate aufweisen.

Beispiel 5.1

- Verhaltensbeschreibung einzelner Subsysteme in jeweils einem Word-Dokument
- Aufrufparameter der Komponentenschnittstelle in einer Excel-Tabelle
- Aufrufreihenfolgen in Sequenzdiagrammen in proprietären Softwarewerkzeugen

- Schnittstellensignale in einer XML-Datei (MSR-Format)
- Architekturvorgaben in Systemanforderungen in proprietären Softwarewerkzeugen
- Ersatzreaktion auf erkannte Fehler in einem Word-Dokument

□

5.1.2 Bewertung der Architekturbeschreibungen

In den verschiedenen Bosch-Geschäftsbereichen hat sich eine im Wesentlichen einheitliche, systematische Vorgehensweise etabliert. Das Verständnis des Schnittstellen-Begriffs ist praktisch überall gleich. Das lässt sich aus den weitgehend gleichen Schnittstellenaspekten in den untersuchten Architekturbeschreibungen ableiten.

Die übliche Beschreibung der Systemschnittstelle deckt die Maschinendarstellung hinsichtlich der Datendarstellung (siehe Abschnitt 4.3) gut ab, d. h. viele Datenfehler lassen sich damit vermeiden. Durch die übliche CAN-Anbindung der Steuergeräte ist über das CAN-Protokoll auch die physikalische Darstellung der Daten eindeutig festgelegt. Diese beinhaltet aber nicht die physikalischen Größen, die durch Sensoren und Aktoren am Steuergerät gemessenen bzw. beeinflusst werden. Die Semantikdarstellung erfolgt in der Regel über eine informelle Beschreibung der Daten.

Das Zeitverhalten an der Systemschnittstelle wird üblicherweise über die Zykluszeit von CAN-Paketen beschrieben. Teilweise wird darüber hinaus die maximal erlaubte Verzögerung, die dem Alter einer Information entspricht, und die maximale Umsetzungszeit für Aktoren angegeben.

Der momentane Stand der Praxis stellt eine gute Grundlage dar, um wesentliche Schnittstelleninkonsistenzen und damit Architekturfehler schon früh erkennen zu können. Vor dem theoretischen Hintergrund in Kapitel 4 sind noch weitere Schnittstelleneigenschaften denkbar, mit deren Hilfe sich das Potential zur Fehlererkennung und -vermeidung weiter steigern lässt.

5.2 Software-Entwicklungsprozesse

Die Software-Entwicklung in der Praxis nimmt das V-Modell als Vorbild. Die Prozessbeschreibungen richten sich nach den Phasen der Systementwicklung. Darüber hinaus werden auch Querschnittsprozesse definiert wie das Problem- und Änderungsmanagement und das Konfigurationsmanagement. Abbildung 5.2 zeigt einen Ausschnitt der inkrementellen Systementwicklung im V-Modell XT [RB07].

Bei der folgenden Darstellung der in der Praxis angewendeten Software-Entwicklungsprozesse wird der Fokus auf die Systemintegration gelegt. Für die Integration interessante Aspekte werden dabei näher betrachtet.

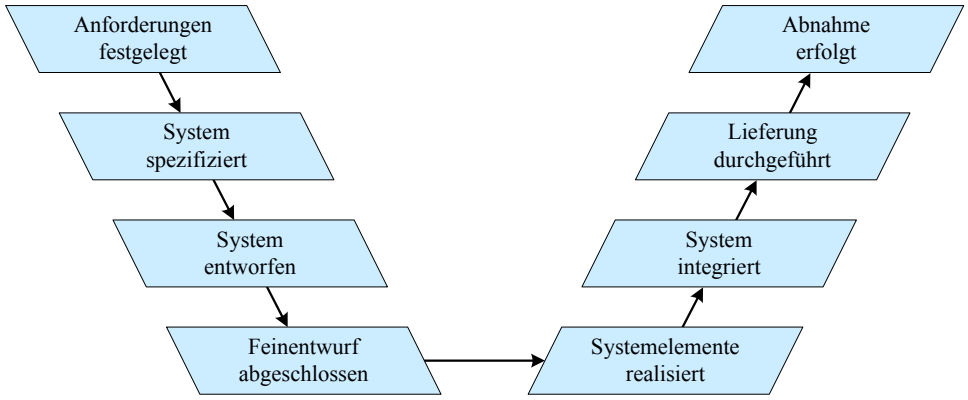


Abbildung 5.2: Ausschnitt aus der inkrementellen Systementwicklung im V-Modell XT

Grundsätzlich muss unterschieden werden zwischen der Entwicklung großer, komplexer Systeme und kleiner, vergleichsweise einfacher Systeme. Große Systeme bestehen aus zahlreichen Subsystemen, die in mehrere Ebenen hierarchisch gegliedert sind, und werden von mehreren Entwicklerteams arbeitsteilig entwickelt. Deshalb sind hier auch deutlich umfangreichere Prozesse im Einsatz als bei der Entwicklung kleinerer Systeme, die von einem einzelnen Entwicklerteam erstellt werden.

5.2.1 Integrationslastige Entwicklung

Trotz des in Abschnitt 5.1 beschriebenen Spezifikationsaufwands lassen sich bei Änderungen nicht immer alle Schnittstellendetails sofort klären. Beispielsweise entstehen durch Weglassen von Subsystemen mit nicht benötigter Funktionalität oder Wechsel zu einer neueren Subsystemversion offene Schnittstellen, also Eingabekanäle, zu denen kein entsprechender Ausgabekanal mehr vorhanden ist. Der verbleibende Abstimmungsaufwand zwischen den Entwicklern der Subsysteme verlagert sich auf diese Weise sehr häufig in die Integrationsphase. Dazu gehören die Klärung offener interner Schnittstellen und damit verbunden das Schließen der Schnittstellenlücken, die Festlegung von Systemkonstanten (Applikation) und der Aufrufreihenfolge von Softwaremodulen in den Tasks sowie die Behebung aller gefundenen Fehler.

Die Fehlersuche selbst erfolgt über diverse Reviews und Modul-, Integrations- und Systemtests. Die nur rudimentär beschriebenen internen Schnittstellen verleihen den Modultests, also den Tests der einzelnen Subsysteme, allerdings nur eine recht geringe Aussagekraft. Viele Fehler können dadurch erst in den Integrations- und Systemtests gefunden werden, was zum erhöhten Aufwand in der Integrationsphase beiträgt. Derzeit werden schätzungsweise 40 % des Entwicklungsaufwands eines Produkts für die Integration benötigt.

Die integrationslastige Entwicklung hat den Vorteil, dass der Spezifikationsaufwand

relativ gering ist und keine separat ausgebildeten Experten für Spezifikationsmethoden und -sprachen benötigt werden. Dem gegenüber steht aber der Nachteil des hohen Integrationsaufwands. Nach [LL06] ist es aber nicht nur im Automobilbereich ein übliches Vorgehen, die Integration zur Anpassung der Schnittstellen zu nutzen, um ein lauffähiges System zu erhalten.

5.2.2 Entwicklung durch Änderung – das Altmeisterprinzip

Große Systeme werden praktisch nie komplett neu entwickelt, sondern bauen immer auf einem bestehenden System auf – meist einer Vorgängerversion oder einem ähnlichen System, auch „Altmeister“ genannt. Dieses System wird dann weiter angepasst, bis alle Anforderungen realisiert sind. Dadurch wird bei größeren Systemen ein hoher Wiederverwendungsgrad erreicht.

Die Weiterentwicklung erfolgt iterativ in mehreren Durchläufen, parallel zur Hardwareentwicklung. Dabei entsteht nach jeder Iteration eine neue Version, die als *Releasestand* bezeichnet wird. Vor der endgültigen Version werden auch einige vorherige Releasestände an den Kunden ausgeliefert und von ihm getestet. Auf diese Weise ist der Kunde über eine Feedbackschleife in die Entwicklung eingebunden und kann seine Anforderungen an das zu erstellende System weiter konkretisieren.

Änderungswünsche kommen nicht nur vom Kunden. Die Software wird streng unterteilt in den kundenspezifischen und den Plattform-Anteil. Letzterer ist ein kundenunabhängiger Softwarebestandteil und bei einem Großteil der Kunden gleich. Die Änderungen dafür entstehen intern im Rahmen der permanenten Weiterentwicklung und Wartung.

Nicht jeder Änderungswunsch wird auch tatsächlich umgesetzt. Die Kontrolle der Änderungen wird im *Änderungsmanagement* geregelt. Dieses ist querschnittlich zu den Entwicklungsphasen und hat im Wesentlichen die Aufgaben

- Änderungsbewertung hinsichtlich Machbarkeit, geschätztem Realisierungsaufwand und -termin und Auswirkungen auf das bestehende System, mit dem Ergebnis der Zustimmung oder Ablehnung des Änderungswunsches
- Gewährleistung der tatsächlichen Umsetzung akzeptierter Änderungen durch Verfolgung (engl. tracking) der Änderungsaufträge
- Beibehalten der Kompatibilität der Schnittstellen

Im Hinblick auf die Systemintegration ist besonders die letzte Aufgabe wesentlich. Die Prozessbeschreibungen geben allerdings keine genaueren Angaben dazu, welche Maßnahmen zur Erfüllung dieser Aufgabe ergriffen werden sollen. In der Praxis werden Werkzeuge zur Prüfung der Schnittstellenkonsistenz eingesetzt, die aber nur jene Schnittstelleneigenschaften prüfen können, die auch in der Architekturbeschreibung angegeben sind, siehe Abschnitt 5.1. Änderungen im Zeitverhalten oder kleinere

Anpassungen in der Berechnungsvorschrift eines Wertes werden üblicherweise nicht als explizite Änderungen betrachtet und damit nicht dem Änderungsmanagement unterworfen.

Die Annahme eines Änderungswunschs wird gegen Ende der Entwicklung, d. h. in den letzten Iterationen, erschwert, indem ein sog. Freeze-Status gesetzt wird. Ein Freeze betrifft sowohl die internen Schnittstellen als auch die Systemschnittstelle. Man unterscheidet üblicherweise zwischen

- Schnittstellen-Freeze: keine Änderungen mehr an den Schnittstellen
- Logic-Freeze: keine Änderungen mehr in der Ablauflogik der Software
- Parameter-Freeze: keine Änderungen mehr an den Systemkonstanten

Nach einem Freeze dürfen entsprechende Änderungen nur noch mit besonderer Genehmigung eingebracht werden. Typischerweise ist dann nur noch die Behebung von Fehlern erlaubt, aber nicht mehr das Einbringen neuer Eigenschaften. In einigen Prozessbeschreibungen spricht man vor einem Freeze von *Anforderungsmanagement* und nach einem Freeze von *Änderungsmanagement*. In dieser Arbeit wird diese sprachliche Unterscheidung nicht gemacht und unter Änderungsmanagement beides verstanden.

Die iterative Weiterentwicklung eines bestehenden Systems relativiert die vorhandenen Defizite in der Architekturbeschreibung zum Teil. Durch die wiederkehrende Integration mit jeder Iteration entsteht ein Kontrollinstrument, das eventuell entstandene Architekturfehler schon vor der Fertigstellung des Systems entdecken kann. Zudem sorgt der „Altmeister“ als funktionsfähiges System dafür, dass unverändert übernommene Subsysteme tendenziell problemlos integriert werden können.

5.2.3 Durchgängigkeit der Entwicklung

Selbst bei einer vorbildlichen Architekturbeschreibung mit konsistenten Schnittstellen ergibt sich nur dann eine positive Auswirkung auf die Integration, wenn die Eigenschaften der Architektur in der Realisierung wiederzufinden sind, d. h. die Durchgängigkeit der Entwicklung gewährleistet ist. Das bedeutet insbesondere, dass die Anforderungen beim Übergang von einer Entwicklungsphase zur nächsten unverändert aufrechterhalten werden.

In der Praxis wird dazu eine durchgängige Werkzeugkette gebildet, die zumindest Teile der Artefakte aus frühen Entwicklungsphasen schrittweise in fertige Produktbestandteile transformiert. Einen wesentlichen Beitrag dazu leistet die automatische Generierung von Schnittstellencode aus der Architekturbeschreibung. Die in Abschnitt 5.1 beschriebenen Schnittstellenangaben erlauben die Generierung von kompilierbarem Programmcode, der sämtliche Schnittstellenvariablen deklariert, und die Generierung der Aufruflisten der Softwaremodule in den Steuergeräte-Tasks.

Die Codegenerierung kann natürlich nur soweit gehen, wie das aufgrund der unvollständigen Architekturbeschreibungen überhaupt möglich ist. Das macht manuelle Codeergänzungen und vor allem die Programmierung der Anwendungslogik derzeit noch unersetzbar.

5.2.4 Abbildung der Organisationsstruktur auf die Systemarchitektur

Die Entwicklung größerer Systeme erfolgt arbeitsteilig in mehreren Teams. Dabei ist zu beobachten, dass die Teamaufteilung und die Subsystemunterteilung des Systems einander sehr ähnlich sind.

Diese Beobachtung ist aber keineswegs untypisch. Sie bestätigt vielmehr *Conways Gesetz* [Con68], nach dem die Architektur eines Systems die Struktur der Organisation widerspiegelt.

Trotz dieser Ähnlichkeit kann die organisatorische Aufstellung das vorhandene Defizit in der Architekturbeschreibung nicht effektiv ausgleichen. Die Entwicklerteams arbeiten daher (oft unbewusst) mit mentalen Artefakten (siehe Abschnitt 2.4.2) und verlagern die erforderliche Schnittstellenabstimmung in späte Entwicklungsphasen. Die Ursache für dieses Problem ist aber weniger in den Entwicklerteams zu suchen, sondern vielmehr im Fehlen eines Verbundverantwortlichen, der speziell das Zusammenspiel der Subsysteme – sowohl in der Architekturbeschreibung als auch später in der Realisierung – im Blickfeld hat und dafür verantwortlich ist. Diese Rolle darf nicht gleichgesetzt werden mit der des Projektleiters, der prinzipiell auch für die Systemfunktionalität, und damit implizit für das Zusammenspiel der Subsysteme, verantwortlich ist. Durch seine Controlling-Aufgaben und die Verwaltung seiner Mitarbeiter hat er eine zu große Distanz zu Schnittstellendetails.

5.3 Empirische Untersuchung von Architekturfehlern

In den vorhergehenden Abschnitten wurde das praktische Vorgehen bei der Softwareentwicklung vorgestellt. Welche Architekturfehler dabei nicht oder nicht immer rechtzeitig gefunden und behoben werden konnten, zeigt die empirische Untersuchung in diesem Abschnitt.

Fehler, die erst in späten Entwicklungsphasen oder im Feld auftreten, werden in der Regel durch ein Fehlverhalten (Versagen) des Systems entdeckt. Abhängig von den Eingabedaten kann sich derselbe Fehler auf recht unterschiedliche Weise (und teilweise auch gar nicht) bemerkbar machen. Aus diesem Grund erfolgen die empirischen Untersuchungen anhand der Fehlerursachen.

Die ermittelten Fehler werden zudem der theoretischen Klassifikation in Abschnitt 4.3 gegenübergestellt und erlauben so eine Plausibilisierung der theoretischen Überlegungen.

5.3.1 Erläuterung und Klassifizierung der ermittelten Fehlerdaten

Aus Gründen der Übersichtlichkeit und der Geheimhaltung wurden die Fehlerdaten auf die wesentlichen Merkmale der Fehlerursache abstrahiert und zu Fehlerklassen mit gleicher Fehlerursache zusammengefasst. Das Ziel ist, einen Überblick über aktuell aufgetretene Architekturfehler zu erhalten, ohne dabei einen Rückschluss auf einzelne betroffene Produkte, Kunden oder Geschäftsbereiche ableiten zu können. Die erhaltenen Fehlerklassen werden im Folgenden einzelnen beschrieben. Tabelle 5.1 gibt einen Überblick über die Fehlerklassen sowie die Einordnung bezüglich der Klassifikationen in Kapitel 4.3.

	Datenfehler	Zeitfehler
Semantik-darstellung	• Tacho- vs. Fahrzeuggeschwindigkeit • Falsche Konfigurationsdaten	• unerwartete Nachrichtenreihenfolge empfangen • Modulaufruf im falschen Task-Zeitraster • falsche Aufrufreihenfolge von Modulen
Maschinen-darstellung	• unterschiedliche Quantisierung • unterschiedliche physikalische Einheiten • falscher Wertebereich • falsch berechnete Prüfsumme • Bit doppelt belegt	• Informationsalter größer als erwartet • Lesen nicht-initialisierter Speicherbereiche • fehlende Tasksynchronisation • Deadlock • Timeout nicht mit Worst-Case-Zeitverhalten abgestimmt • zu große Zeitverzögerung beim Weiterleiten einer Botschaft
Physikalische Darstellung	• Verzerrung von Eingabedaten	• Hardwareausfall nicht berücksichtigt • Hardwaredefekt durch zu viele Schreibzyklen

Tabelle 5.1: Einordnung in Fehlerklassen

Tacho- vs. Fahrzeuggeschwindigkeit Ein sendendes Steuergerät übermittelt die Tachogeschwindigkeit, während das empfangende Steuergerät die Fahrzeugge-

schwindigkeit erwartet hat. Da beide Werte leicht voneinander abweichen, hat dies zu einem falschen Regelungsverhalten geführt.

falsche Konfigurationsdaten Die Konfigurationsdaten eines Fahrzeugs enthalten Angaben zu Fahrzeugeigenschaften und -ausstattung, z. B. Motorleistung, Achsabstand und Klimaanlage. Wenn die Konfigurationsdaten nicht zum tatsächlichen Fahrzeug passen, werden falsche Berechnungsmodelle angewandt, und es kann zu einem unerwünschten Systemverhalten kommen.

unerwartete Nachrichtenreihenfolge empfangen Steuergeräte kommunizieren über den Austausch von Nachrichtenpaketen. Trifft das empfangende Steuergerät bestimmte Annahmen über die Reihenfolge der Pakete, die von den sendenden Steuergeräten nicht erfüllt werden, kann das zu einem fehlerhaften Gesamtverhalten führen.

Modulaufruf im falschen Task-Zeitraster Die Berechnungen in einem Steuergerät erfolgen in regelmäßigen Zeitabständen. Die Modulaufrufe sind beispielsweise unterteilt in 10 ms-, 20 ms-, 100 ms- und 200 ms-Tasks und werden mit der jeweiligen Periodendauer aufgerufen. Im konkreten Fehlerfall erfolgte ein Modulaufruf fälschlicherweise im 200 ms-Task anstatt im 100 ms-Task, was zu einem falschen Berechnungsergebnis führte.

falsche Aufrufreihenfolge von Modulen Die Berechnung von Werten erfolgt im Steuergerät häufig in mehreren Schritten, indem nacheinander bestimmte Softwaremodule aufgerufen werden. Eine falsche Implementierung der Aufrufreihenfolge (engl. sequencing) führt möglicherweise zu einem falschen Berechnungsergebnis.

unterschiedliche Quantisierung Unter Quantisierung versteht man die Darstellung einer (physikalischen) Größe in einem Zahlensystem, in dem sie nur diskrete Werte annehmen kann. Typische Beispiele für aufgetretene, unterschiedliche Quantisierungen sind: 1 °C vs. 0,75 °C Schrittweite, 5/3 vs. 3/5 Schrittweite.

unterschiedliche physikalische Einheiten In der Maschinendarstellung wird die physikalische Einheit eines Zahlenwerts in der Regel nicht mit übertragen, da diese ohnehin konstant ist und so nur unnötigen Overhead erzeugen würde. Gehen Sender und Empfänger aber von unterschiedlichen physikalischen Einheiten aus, wird der übertragene Wert unterschiedlich interpretiert. Beispiele aus der Praxis sind: °C vs. K, hPa vs. Pa, Nm vs. %.

falscher Wertebereich Der Sender eines Wertes verwendet einen Wertebereich, der nicht vollständig innerhalb des Wertebereichs des Empfängers liegt. Dies ist beispielsweise der Fall, wenn der Empfänger keine negativen Werte erwartet, der Sender u. U. aber welche sendet.

falsch berechnete Prüfsumme Durch ein geändertes Berechnungsverfahren für eine Prüfsumme hat der Empfänger die erhaltenen Daten als fehlerhaft interpretiert. Dieser Fehler ist eigentlich ein Spezialfall eines *falschen Wertebereichs*, wenn man Nutzdaten und Prüfsumme zu einem Tupel zusammenfasst.

Bit doppelt belegt Abhängig von der Fahrzeugausstattung war ein bestimmtes Bit in einem CAN-Paket unterschiedlich verwendet. Bei einer bestimmten Ausstattungsvariante kam es zu einer Doppelbelegung des Bits. Sendendes und empfangendes Steuergerät haben es mit völlig unterschiedlicher Bedeutung interpretiert.

Informationsalter größer als erwartet Das Informationsalter ist die verstrichene Zeit seit dem Entstehen einer Information z. B. durch Messen mit einem Sensor. Das Informationsalter steigt mit der Verarbeitungszeit in einem Steuergerät sowie auch durch Glättung einer Zeitreihe mit Hilfe des gleitenden Durchschnitts, da sich der Wert älteren Werten angleicht und damit zeitlich zurückliegt. Empfängt ein Steuergerät eine Information, die älter ist als erwartet, so wirkt sich das negativ auf die Korrektheit der Berechnungsergebnisse aus. Besonders deutlich wird der Effekt sichtbar, wenn mehrere Regelkreise Informationen ohne abgestimmtes Informationsalter austauschen und dadurch zu schwingen beginnen.

Lesen nicht-initialisierter Speicherbereiche Ist zum Zeitpunkt des Lesezugriffs auf einen Speicherbereich dieser noch nicht initialisiert worden, so werden in der Regel – abhängig von der Hardware und den Umgebungsbedingungen – zufällige Werte gelesen. Dies führt fast immer zu falschen Berechnungsergebnissen.

fehlende Tasksynchronisation Greifen mehrere, nebenläufige Tasks auf gemeinsame Daten oder Betriebsmittel zu, muss die zeitliche Reihenfolge der Zugriffe u. U. koordiniert werden (Synchronisation). Fehlt die Synchronisation, kann das zu unvorhergesehenen Datenflüssen, inkonsistenten Datensätzen und schließlich zu falschen Berechnungsergebnissen führen. Der Grund für eine fehlende Tasksynchronisation ist häufig die Unkenntnis über die Nutzung der Variablen.

Deadlock Stehen zwei Systeme in gegenseitiger Wartebeziehung, so liegt ein Deadlock vor. Dies ist beispielsweise der Fall, wenn zwei Steuergeräte nach dem Einschalten auf das Setzen eines OK-Flags jeweils beim anderen Steuergerät warten.

Timeout nicht mit Worst-Case-Zeitverhalten abgestimmt Um den Ausfall eines Steuergeräts zu erkennen, wird mit Hilfe eines Watchdogs eine maximal zu wartende Zeit ohne Lebenszeichen überwacht und bei Zeitüberschreitung (timeout) ein bestimmtes Signal gesendet. Typische Szenarien sind die gegenseitige Überwachung von Steuergeräten mit Protokollierung von Fehlern oder die Selbstüberwachung eines Steuergeräts mit einem Hardware-Watchdog, der bei

Zeitüberschreitung einen Reset auslöst. Wird die maximale Wartezeit zu klein gewählt, d. h. kleiner als die im fehlerfreien Betrieb mögliche Zeit bis zum nächsten Lebenszeichen, dann wird ein scheinbar vorliegender Fehler erkannt. Bei der Wahl der maximalen Wartezeit müssen Hardware-Toleranzen und ein möglicher Jitter des Sendezeitpunkts berücksichtigt werden.

zu große Zeitverzögerung beim Weiterleiten einer Botschaft Einige Nachrichten, die ein Steuergerät erreichen, sind von diesem einfach nur weiterzuleiten z. B. an einen Aktor, der am Steuergerät angeschlossen ist. Erfolgt hierbei eine zu große Zeitverzögerung, z. B. weil erst noch auf den Beginn des nächsten Taskzyklus gewartet wird, kann sich das negativ im Gesamtsystem bemerkbar machen. Beispielsweise macht sich dies besonders deutlich bemerkbar, wenn mehrere Aktoren (z. B. Blinkleuchten) zeitgleich angesteuert werden sollen, aufgrund unterschiedlicher Nachrichtenverzögerungen aber zeitversetzt reagieren.

Verzerrung von Eingabedaten Die Übertragung digitaler Daten erfolgt üblicherweise über Spannungsverläufe in metallischen Leitern. Durch Einwirkung von elektromagnetischer Strahlung kann der Spannungsverlauf derart verzerrt werden, dass man bei der Interpretation auf einen falschen Wert kommt. Ist die Kommunikation nicht ausreichend gegen Bitfehler abgesichert, bleibt der Fehler unerkannt und kann zu falschen Ausgaben führen.

Hardwareausfall nicht berücksichtigt Wird der Ausfall eines oder mehrerer Steuergeräte in der Architektur nicht ausreichend berücksichtigt, so ist das Verhalten des Gesamtsystems im Fehlerfall möglicherweise nicht vorhersagbar.

Hardwaredefekt durch zu viele Schreibzyklen Der nichtflüchtige Speicherbaustein (EEPROM, Electrically Erasable Programmable Read-Only Memory) in Steuergeräten ist nur für eine bestimmte Maximalzahl an Schreibzyklen, typischerweise 100 000, ausgelegt. Wird diese Anzahl überschritten, kann der Speicher defekt werden. Diese Begrenzung muss von der Software berücksichtigt werden, um eine ausreichende Lebensdauer des Steuergeräts sicherzustellen.

5.3.2 Ursachen für die Entstehung von Architekturfehlern

Bei den im vorherigen Abschnitt vorgestellten Fehlerdaten ist jeweils die Fehlerursache angegeben. Es zeigt sich ein großes Spektrum an Fehlerquellen. Geht man der Frage nach, wie es überhaupt zu diesen Fehlern kommen konnte, stößt man interessanterweise immer wieder auf dieselben Grundprobleme. Insgesamt lässt sich die Entstehung von Architekturfehlern auf drei Hauptursachen zurückführen, die allesamt Schwachpunkte der eingesetzten Entwicklungsprozesse darstellen.

Unzureichende Spezifikation der Architektur

Die Architektur eines Systems beschreibt die Schnittstellen aller Subsysteme sowie deren Zusammenspiel. Damit legt die Architektur die Zerlegung des Systems in seine Bestandteile fest.

Wie in Abschnitt 2.4.2 bereits erwähnt, wird die fehlende Information über die Architektur durch implizite Annahmen in den mentalen Modellen der Entwickler „vervollständigt“. Diese architekturrelevanten Annahmen führen nicht zuletzt zu den Fehlern im Zusammenspiel von Komponenten, die im vorherigen Abschnitt beschrieben worden sind.

Enthält eine Architektur nur eine unzureichende Beschreibung der Subsysteme, erschwert das die Fehlerlokalisierung oder macht sie evtl. sogar unmöglich. Denn in der Praxis sind die Subsystemspezifikationen ausschlaggebend für den Korrektheitsnachweis von Subsystemrealisierungen. Bei einer rudimentären Spezifikation wird das Subsystem recht schnell als spezifikationskonform eingestuft. Im Extremfall kann das dazu führen, dass *alle* Subsysteme für korrekt befunden werden und dennoch das integrierte System fehlerhaft ist. Dies ist ein klares Indiz für eine unzureichende Architekturbeschreibung.

Einseitige Schnittstellenänderungen

Die Entwicklung von neuen Systemen basiert zum Großteil auf der Wiederverwendung und Anpassung ähnlicher bereits bestehender Systeme (siehe Abschnitt 5.2). Das Einbringen neuer Funktionalität erfordert in der Regel die Änderung von Subsystemschnittstellen. Geschieht das nur einseitig, d. h. nur an einem Subsystem, ohne davon betroffene Nachbarsubsysteme ebenfalls anzupassen, hinterlässt das möglicherweise eine inkonsistente Architektur, bei der das Zusammenspiel der Subsysteme gestört ist.

Komponentenorientierte Unternehmensstruktur

Wie in Abschnitt 5.2 bereits erwähnt, folgt die Einteilung der Entwicklerteams weitgehend der Subsystemstruktur der zu entwickelnden Produkte. Diese Form der Unternehmensstruktur stärkt die arbeitsteilige Entwicklung der Subsysteme, sorgt aber nicht automatisch für ein reibungsloses Zusammenspiel nach der Integration der realisierten Teilsysteme. Letzteres erfordert die Rolle eines Verbundverantwortlichen, der das funktionale Zusammenspiel der Subsysteme vom Anfang des Entwicklungsprozesses an überwacht und sicherstellt. Zu den Aufgaben des Projektverantwortlichen gehört dies in der Regel nicht.

5.3.3 Erkenntnisse aus den empirischen Fehlerdaten

Die gefundenen Fehlerdaten zeigen, dass die in der Praxis auftretenden Architekturfehler sehr vielfältig sind und auf allen Abstraktionsebenen auftreten (siehe Tabelle 5.1). Obwohl nur Fehlerdaten ausgewertet wurden, die erst recht spät im Entwicklungsprozess gefunden wurden, finden sich darin dennoch auch vergleichsweise banale Fehler

wieder wie unterschiedliche physikalische Einheiten oder ein doppelt belegtes Bit.

Andererseits ist auffällig, dass keine Fehler aufgrund von Ressourcenbeschränkungen durch eingebettete Systeme enthalten sind. Denn ob die Ressourcen ausreichend sind, kann aus Sicht der einzelnen Subsysteme im Allgemeinen gar nicht entschieden werden. Erst die ganzheitliche Betrachtung aller Subsysteme erlaubt eine Abschätzung des gesamten Ressourcenverbrauchs. Im Einzelnen ist zu prüfen:

- Reicht der Speicher/Heap/Stack für alle Komponenten aus?
- Reicht die verfügbare Rechenzeit aus? Ist ein Scheduling aller Tasks unter Einhaltung aller Zeitvorgaben möglich?
- Können alle Daten in der erforderlichen Zeit übertragen werden? Reicht die Bandbreite und Übertragungsgeschwindigkeit aus?

Aus den empirischen Fehlerdaten ergeben sich jedoch keine deutlichen Hinweise auf Architekturfehler bezüglich der gemeinsamen Nutzung von Ressourcen. Dies lässt den Schluss zu, dass die Ressourcenbeschränkung in der Praxis gut beherrscht wird.

5.4 Zusammenfassung

In diesem Kapitel wurde der Stand der Praxis in der Softwareentwicklung im Automobilbereich beschrieben. Es wurde aufgezeigt, welche integrationsrelevanten Aspekte in den Entwicklungsprozessen enthalten und welche Defizite zu erkennen sind. Dabei wurde vertieft auf die gängige Praxis bei der Architekturbeschreibung eingegangen.

Wesentliche Beobachtungen sind die Konzentration auf syntaktische Schnittstelleneigenschaften in der Architekturbeschreibung sowie die integrationslastige, iterative Entwicklung von Software. Die Untersuchung empirischer Fehlerdaten belegt die Vielfalt der möglichen Fehlerquellen, die zuvor in Kapitel 4 formal beschrieben wurden.

Die Erkenntnisse in diesem Kapitel sind eine wichtige und hilfreiche Grundlage, um gezielt Maßnahmen zur effektiven Erkennung und Vermeidung von Architekturfehlern ergreifen zu können (siehe Kapitel 7).

6 Anforderungen an den Entwicklungsprozess zur Vermeidung von Architekturfehlern

In Kapitel 5 wurde der Stand der Praxis in der Softwareentwicklung für eingebettete Systeme im Automobilbereich untersucht. Dabei wurde gezeigt, dass heutige Software-Entwicklungsprozesse hinsichtlich der rechtzeitigen Erkennung und Vermeidung von Architekturfehlern noch verbessert werden können.

Dieses Kapitel untersucht zunächst aus formaler Sicht, welche Eigenschaften ein Entwicklungsprozess haben muss, um Architekturfehler vollständig vermeiden zu können. Konkrete Maßnahmen zur Erfüllung dieser Eigenschaften werden anschließend in Kapitel 7 vorgestellt.

6.1 Durchgängigkeit des Entwicklungsprozesses

Ein (theoretisch) fehlerfreies Produkt lässt sich entwickeln, indem die Durchgängigkeit der spezifizierten (Sub-)Systemeigenschaften sichergestellt wird, d. h. der Übergang zwischen jeweils zwei aufeinander folgenden Entwicklungsphasen eine Verfeinerung des Artefakts bedeutet. Unter Verfeinerung wird die im Systemmodell vorgestellte Verhaltensverfeinerung verstanden (siehe Abschnitt 3.3.4).

In der Praxis ist diese Durchgängigkeit oft nicht gegeben. Architekturbeschreibungen enthalten in der Regel nur unzureichende Angaben zum Schnittstellenverhalten. Wird beispielsweise keine Aussage über die maximalen Antwortzeiten der Subsysteme gemacht, sind beliebig lange Antwortzeiten der Subsystemrealisierungen akzeptabel. Dadurch kann aber eine sinnvolle Systemfunktionalität im Echtzeitumfeld nicht gewährleistet werden. Viele Schnittstelleneigenschaften bleiben auf diese Weise offen und erlauben prinzipiell auch Realisierungen, die offensichtlich nicht der Echtzeitanforderung in der Systemspezifikation entsprechen. Aber auch inkonsistente Schnittstelleneigenschaften und fehlerhafte Transformationen der Entwicklungsartefakte können zu einem fehlerhaften System führen.

Die heute recht intensiv durchgeführten Reviews und Tests sind ein sehr wichtiges Mittel zur Überprüfung der Produktqualität. Als analytische Qualitätssicherungsmaßnahmen sind sie aber nicht zur systematischen Vermeidung konzeptioneller Fehler ausreichend.

6.1.1 Hinreichende Architekturbeschreibung

Gegeben sei eine Systemspezifikation S und eine Architektur mit den Subsystemspezifikationen $S_1, \ldots, S_n$. Eine Spezifikation beschreibt, welche Eigenschaften ein System haben muss, bildet also Vorgaben und damit Einschränkungen des möglichen Verhaltens. Sind Subsysteme nur unzureichend genau beschrieben, so können daraus, wie oben bereits geschildert, fehlerhafte Realisierung abgeleitet werden. Formal gesehen stellt eine derartige Architekturbeschreibung eine inkorrekte Verfeinerung der Systemspezifikation dar, d. h.

$$S \not\rightsquigarrow (S_1 \otimes \cdots \otimes S_n)$$

Größere Systeme werden häufig arbeitsteilig entwickelt. Die Entwicklerteams sind nicht selten auch räumlich voneinander getrennt, und es findet nur ein begrenzter Austausch zwischen ihnen statt. Besonders bei unzureichend beschriebenen inneren Strukturen der zu entwickelnden Subsysteme (siehe Abschnitt 5.1) tritt dann der Effekt der implizit getroffenen Annahmen auf. Architekturrelevante Annahmen können leicht Probleme bei der Integration verursachen. Durch eine hinreichende Architekturbeschreibung kann das verhindert werden. Sie legt das Zusammenspiel der Subsysteme derart fest, dass man bei fehlerfreier Realisierung der Subsysteme gemäß ihrer Spezifikation und anschließender Integration ein korrektes System erhält. Eine Architekturbeschreibung wird als hinreichend bezeichnet, wenn gilt:

$$S \rightsquigarrow (S_1 \otimes \cdots \otimes S_n)$$

Um diese Eigenschaft zu erfüllen, ist es nicht erforderlich, alle denkbaren Eigenschaften der Subsystemschnittstellen festzulegen. Es müssen nur diejenigen Eigenschaften definiert werden, die zur Einhaltung der Systemanforderungen notwendig sind. Macht die Systemspezifikation beispielsweise keine Aussage über maximale Antwortzeiten, dann sind in den Subsystemspezifikationen ebenfalls keine Antwortzeiten erforderlich. Der notwendige Spezifikationsumfang hängt also direkt von der Systemspezifikation ab.

6.1.2 Konsistente Schnittstellen in der Architekturbeschreibung

Die Forderung nach einer korrekten Verfeinerung der Systemspezifikation reicht alleine noch nicht aus. Um eine fehlerfreie Realisierung der Subsysteme zu ermöglichen, sind konsistente und damit widerspruchsfreie Schnittstellenbeschreibungen eine Grundvoraussetzung.

Formal drückt sich dies in der Realisierbarkeit der Subsystemspezifikationen aus. Zwar macht sich eine fehlende Realisierbarkeit spätestens beim Versuch, die Spezifikation umzusetzen, bemerkbar, allerdings wird bereits von einer sinnvollen Architekturbeschreibung verlangt, dass sie eine vernünftige Basis für die weitere Entwicklung ist.

Es muss also gelten:

$\forall\, i \in [\,1 \ldots n\,] : S_i$ ist realisierbar

6.1.3 Korrekte Verfeinerungsschritte über den Entwicklungsprozess hinweg

Die bisher genannten beiden Eigenschaften beziehen sich nur auf die Herleitung der Architekturbeschreibung. Die Durchgängigkeit des Entwicklungsprozesses erfordert jedoch die Korrektheit *aller* Übergänge aufeinander folgender Entwicklungsphasen. Gemäß der modularen Entwicklung müssen also folgende Eigenschaften erfüllt sein (siehe Abschnitt 4.1):

- $S \rightsquigarrow (S_1 \otimes \cdots \otimes S_n)$
- $\forall\, i \in [\,1 \ldots n\,] : S_i \rightsquigarrow R_i$
- $n = m \quad \text{und} \quad \forall\, i \in [\,1 \ldots n\,] : R_i = R'_i \quad \text{sowie} \quad R = (R'_1 \otimes \cdots \otimes R'_m)$

Natürlich kann jedes Subsystem S_i wiederum in mehrere Subsysteme zerlegt und modular entwickelt werden. Auf diese Weise werden je nach Bedarf mehrere Architekturebenen erzeugt.

6.2 Konsistenzsicherung bei Änderung eines bestehenden Systems

Der vorherige Abschnitt beschreibt Eigenschaften, die aus der Sicht der einmaligen Erstellung eines Systems zu einem korrekten System führen. Wie in Kapitel 5 dargestellt, trifft man in der Praxis häufig eine iterative Entwicklung an, also ein wiederholtes Durchlaufen der Entwicklungsphasen.

Natürlich lassen sich auch hier die Eigenschaften aus dem vorherigen Abschnitt anwenden. Aufgrund der oft nur lokalen Änderungen an einem System ist aber evtl. eine optimierte Vorgehensweise günstiger, für die zur Sicherstellung der Korrektheit des Systems nicht in jedem Fall die gesamten Artefakte betrachtet werden müssen.

Ausgangspunkt ist ein bestehendes System zusammen mit seinen Entwicklungsartefakten. Diese müssen korrekt zueinander sein, also die Eigenschaften aus dem vorherigen Abschnitt erfüllen. Das Ziel ist, nach jeder Änderung wieder die dort beschriebenen Korrektheitseigenschaften zu erfüllen. Insgesamt erhält man so ein System ohne Architekturfehler.

6.2.1 Feingranulare Änderungserfassung

Eine wichtige Voraussetzung für die Konsistenzsicherung ist die Erfassung *aller* Änderungen, auch wenn sie noch so unscheinbar zu sein scheinen. Dazu gehören nicht nur Änderungen z. B. in der Datendarstellung einer Schnittstelle, sondern auch Änderungen im Zeitverhalten, in der Berechnungsvorschrift für die Bestimmung der Ausgabe in Abhängigkeit von den Eingaben des Subsystems sowie Änderungen in der Kommunikationsstruktur.

6.2.2 Propagieren von Änderungen

Wurde eine Änderung an der Schnittstelle bzw. dem Schnittstellenverhalten eines Subsystems durchgeführt, ist anschließend dafür zu sorgen, dass alle betroffenen Nachbarsubsysteme ebenfalls angepasst werden, um dadurch die Konsistenz der Schnittstellen und die Korrektheit des Systems wieder herzustellen.

Bei den Nachbarsystemen wiederholt sich dieser Vorgang, und es wird wiederum entschieden, welche weiteren Nachbarsysteme von der Änderung betroffen sind. Dadurch wird eine Änderung potenziell durch die gesamte Architektur propagiert und die Auswirkung für jedes Subsystem geprüft. Entscheidend für die Effizienz dieses Vorgehens ist nun, dass man erkennt, welche Nachbarsubsysteme nicht von der Änderung betroffen sind und sich die Änderungswelle deshalb dort nicht fortsetzt.

Hat eine Änderung keine Auswirkung auf die Systemschnittstelle, d. h. alle Subsysteme zusammen bilden eine Verfeinerung des Systems, so kann diese Änderung zwar mehrere Subsysteme betreffen, wird aber von diesen komplett aufgefangen. Formal wird dies durch den Begriff der Umgebungskompatibilität wie folgt ausgedrückt:

Definition 6.1 (Umgebungskompatibilität einer Änderung)
Gegeben sei die Architektur eines Systems mit Subsystemen $S_1, \ldots, S_n$. Eine Änderung an Subsystemen einer Menge $M \subseteq \{S_1, \ldots, S_n\}$ ist *umgebungskompatibel*, wenn gilt:

$$\bigotimes M \rightsquigarrow \bigotimes M'$$

wobei M' die Subsysteme aus M nach der Änderung enthält und $\bigotimes M$ abkürzend für die Komposition aller Subsysteme in M steht, d. h.

$$\bigotimes M \stackrel{\mathsf{def}}{=} M_1 \otimes \cdots \otimes M_k \quad \text{mit} \quad M = \{M_1, \ldots, M_k\}$$

□

Beispiel 6.1 Typische Beispiele für Änderungen, die umgebungskompatibel sein können, sind:

- Änderung des Kommunikationsprotokolls zwischen zwei Subsystemen (siehe Abbildung 6.1).

- Änderungen an den Schnittstellen innerhalb einer Menge von Subsystemen.
- Änderung der Genauigkeit eines Ausgabewerts innerhalb des Toleranzbereichs der empfangenden Subsysteme.
- Änderung des Antwortzeitverhaltens eines Subsystems, das aber immer noch innerhalb des erlaubten Empfangszeitjitters der Nachbarsubsysteme liegt.

□

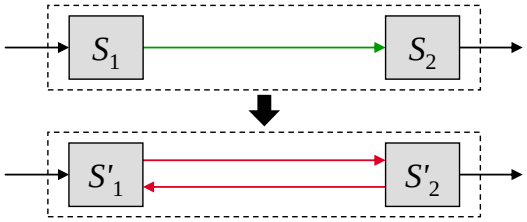


Abbildung 6.1: Änderung des Kommunikationsprotokolls zwischen den Subsystemen in $M = \{S_1, S_2\}$. Wenn $(S_1 \otimes S_2) \rightsquigarrow (S'_1 \otimes S'_2)$ gilt, ist die Änderung umgebungskompatibel und hat damit keine Auswirkung auf das restliche System, insbesondere nicht auf das Systemverhalten. Mit $M' = \{S'_1, S'_2\}$ gilt also $\bigotimes M \rightsquigarrow \bigotimes M'$.

Bestehen die Subsysteme eines Systems wiederum aus einer Zerlegung in Subsysteme, genügt es, die Konsistenzsicherung nur jeweils auf die einzelnen Zerlegungen anzuwenden.

Beispiel 6.2 Hat ein System einen hierarchischen Aufbau wie in Abbildung 6.2, sind lediglich folgende Konsistenzen zu sichern:

$S \rightsquigarrow (S_1 \otimes S_2 \otimes S_3)$

$S_1 \rightsquigarrow (S_{11} \otimes S_{12} \otimes S_{13})$

$S_2 \rightsquigarrow (S_{21} \otimes S_{22} \otimes S_{23} \otimes S_{24})$

$S_3 \rightsquigarrow (S_{31} \otimes S_{32})$

Aus der Monotonie des Kompositionsoperators (siehe Abschnitt 3.4) ergibt sich unmittelbar die Korrektheit der gesamten Zerlegung. □

Die oben eingeführte Umgebungskompatibilität ist auch auf nicht benachbarte Subsysteme in einem hierarchisch aufgebauten System anwendbar. Die Zerlegungsstruktur bleibt dabei erhalten, wie folgendes Beispiel illustriert.

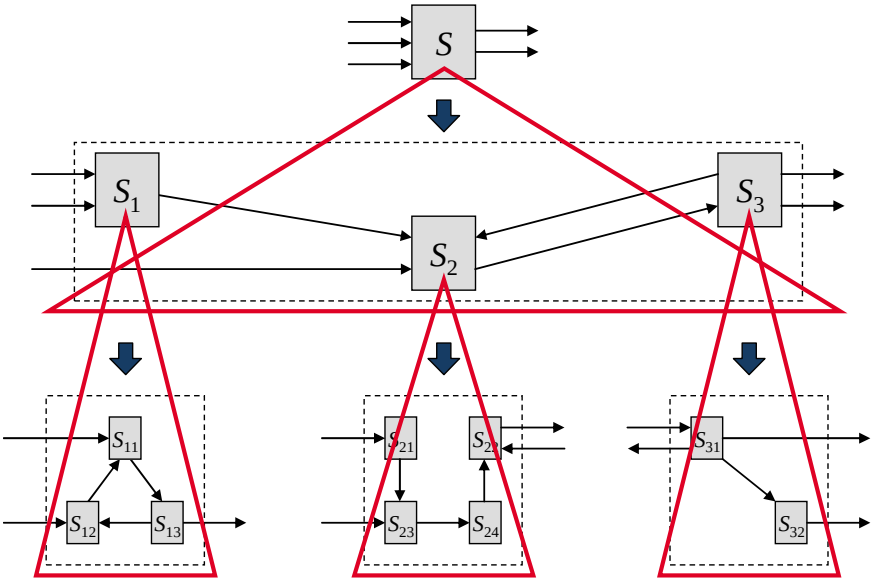


Abbildung 6.2: Konsistenzsicherung bei einem hierarchisch aufgebauten System. Die zu sichernden Zerlegungen sind durch Dreiecke gekennzeichnet.

Beispiel 6.3 Ausgangspunkt sei wieder der hierarchische Aufbau in Abbildung 6.2. Beispielsweise werde die Kommunikation zwischen den Subsystemen S_{13} und S_{23}, die sich sogar in verschiedenen Zerlegungsteilbäumen befinden, umgebungskompatibel geändert. Für die geänderten Subsysteme S'_{13} und S'_{23} gilt aufgrund der Umgebungskompatibilität

$$(S_{13} \otimes S_{23}) \rightsquigarrow (S'_{13} \otimes S'_{23}).$$

Mit

$$S'_1 \stackrel{\text{def}}{=} S_{11} \otimes S_{12} \otimes S'_{13},$$

$$S'_2 \stackrel{\text{def}}{=} S_{21} \otimes S_{22} \otimes S'_{23} \otimes S_{24}$$

ergibt sich aufgrund der Kommutativität und Monotonie des Kompositionsoperators

$$S \rightsquigarrow (S_1 \otimes S_2 \otimes S_3) \rightsquigarrow (S'_1 \otimes S'_2 \otimes S_3),$$

also die Korrektheit der Zerlegung unter Beibehaltung der Zerlegungsstruktur. □

7 Praxisorientierte Maßnahmen zur Erkennung und Vermeidung von Architekturfehlern

In diesem Kapitel werden konkrete Maßnahmen vorgeschlagen, die zur Erfüllung der in Kapitel 6 beschriebenen Anforderungen an den Entwicklungsprozess beitragen und damit Architekturfehler vermeiden. Dabei wird besonderes Augenmerk auf die Praxistauglichkeit der Maßnahmen gelegt, zu deren Bewertung die Fehlerdaten aus Kapitel 5 herangezogen werden.

Ziel dieses Kapitels ist die Bewertung von existierenden Maßnahmen und nicht die Herleitung neuartiger Maßnahmen.

7.1 Arten von Maßnahmen

Im Folgenden soll ein kurzer Überblick über die unterschiedlichen Arten möglicher Maßnahmen gegeben werden (siehe auch [SZRS06]). Dieser dient der Einordnung der später vorgestellten Maßnahmen.

Maßnahmen zur Fehlererkennung und -vermeidung sind Bestandteil der Qualitätssicherung. Balzert [Bal97] unterteilt diese in analytische und konstruktive Qualitätssicherungsmaßnahmen[1]:

Analytische Qualitätssicherungsmaßnahmen sind Maßnahmen zur Überprüfung der Arbeitsprodukte auf Qualität, also diagnostische Maßnahmen. Sie sind nicht darauf ausgelegt, Qualität in das Produkt zu bringen, sondern messen das vorhandene Qualitätsniveau. Diese Maßnahmen werden weiter unterteilt in *analysierende Verfahren* wie statische Analyse, Programmverifikation und Review sowie *testende Verfahren* wie dynamischer Test und Simulation.

Konstruktive Qualitätssicherungsmaßnahmen sind Maßnahmen, die dafür sorgen, dass das entstehende Produkt a priori bestimmte Eigenschaften besitzt. Hier unterscheidet man zwischen *technischen Maßnahmen*, also Methoden, Sprachen

[1] Balzert [Bal97] verwendet dafür den Begriff *Qualitätsmanagementmaßnahmen*. In dieser Arbeit wird jedoch der Begriff *Qualitätssicherungsmaßnahmen* vorgezogen, da dieser das Ziel der Maßnahmen besser ausdrückt, nämlich die Sicherung eines bestimmten Qualitätsniveaus.

und Werkzeugen, und *organisatorischen Maßnahmen* wie Prozessen (CMMI, Verfahrensanweisungen, ...), Weiterbildung und Zertifizierungen.

7.2 Analytische Maßnahmen

Analytische Maßnahmen messen das vorhandene Qualitätsniveau. Damit sind sie in erster Linie ein beobachtendes Entwicklungsinstrument, das zur Erkennung diverser Fehlerarten eingesetzt werden kann. Die Fehlererkennung allein führt zwar nicht unmittelbar zu einer hohen Produktqualität, sie kann jedoch als Indikator für mögliche bevorstehende Qualitätsprobleme verwendet werden und durch geeignete Folgemaßnahmen zu einer Qualitätsverbesserung führen.

Im Rahmen dieser Arbeit steht die frühe Erkennung von Architekturfehlern im Vordergrund.

7.2.1 Reviews

Die Bezeichnung *Review* wird hier als Oberbegriff für manuelle Prüfmethoden verwendet. Dabei werden schriftliche Dokumente begutachtet mit dem Ziel, Fehler zu finden. Kein Bestandteil des Reviews ist das Finden der Ursache, eines Schuldigen oder das Identifizieren eines Lösungsansatzes.

Im Wesentlichen werden folgende Review-Arten unterschieden:

Inspektion Eine formalisierte Prüfmethode, um Fehler in einem schriftlichen Dokument anhand von Referenzunterlagen zu finden. Zu den Teilnehmern gehören ein Moderator, der Autor, Gutachter, ein Protokollant sowie optional ein Vorleser. Während der Inspektion ist der Autor passiv, die Prüfer sind aktiv. Eine genaue Beschreibung der Inspektion ist in [Fag76] enthalten.

(technisches) Review Eine semiformale Prüfmethode, um Stärken und Schwächen eines schriftlichen Dokuments anhand von Referenzunterlagen zu finden. Das technische Review ist weniger formal als die Inspektion. Die Rollenbesetzung ist wie bei der Inspektion, allerdings ohne Vorleser. Für weitere Details siehe [FLS06].

Walkthrough Eine informelle Prüfmethode, um Fehler in einem schriftlichen Dokument zu finden. Der Autor selbst stellt das Dokument einer Runde von Gutachtern vor.

Reviews zeichnen sich durch viele Vorteile aus (siehe [Ben05]): Sie sind in allen Entwicklungsphasen einsetzbar, insbesondere schon in frühen Entwicklungsphasen. Es können beliebige Dokumente geprüft werden. Es wird ein breites Fehlerspektrum abgedeckt. Im Vergleich zu anderen Qualitätssicherungsmaßnahmen in diesem Kapitel

sind Reviews relativ einfach, schnell und kurzfristig durchführbar und haben zudem ein gutes Kosten-Nutzen-Verhältnis. Empirische Ergebnisse belegen, dass durch Inspektion 50 % bis 75 % aller Entwurfsfehler gefunden werden können [Bal97].

Erkennbare Architekturfehler

Durch das breite Fehlerspektrum können mit Reviews prinzipiell alle in Kapitel 5 ermittelten Fehlerarten gefunden werden. Eine Garantie gibt es dafür allerdings nicht. Da es sich um eine manuelle Methode handelt, ist der Erfolg von der Erfahrung der Gutachter abhängig. Weitere Informationen zu Reviews finden sich in [GGF93, Lai02].

7.2.2 Architekturverifikation

Die Architekturverifikation geht über die Fehlersuche, wie sie beim Review stattfindet, hinaus und hat zum Ziel, eine Architektur als korrekt gegenüber der Systemspezifikation nachzuweisen oder dabei vorhandene Fehler aufzudecken. Grundsätzlich betrachtet die Architekturverifikation mehrere Bereiche der Produktqualität wie Funktionalität, Zuverlässigkeit und Wartbarkeit. Im Hinblick auf die Systemintegration wird an dieser Stelle aber nur auf die funktionale Korrektheit näher eingegangen.

Methoden zur Architekturbewertung, bei denen eine Architektur hinsichtlich mehrerer Qualitätseigenschaften qualitativ oder quantitativ bewertet wird, setzen eine einigermaßen detaillierte Architekturbeschreibung voraus, die in der Praxis meistens nicht gegeben ist.

Alternativ kann die Architektur der Realisierung untersucht werden, da diese konkret gegeben ist, auch wenn sie möglicherweise erst durch Reengineering gewonnen werden muss. Mögliche Ansätze zur Identifizierung potenzieller Architekturfehler sind:

- *Model Checking*, um das zeitliche Verhalten und die Interaktion mehrerer Subsysteme, ausgedrückt durch endliche Zustandsautomaten, auf unerlaubte Zustandskombinationen und Deadlocks zu untersuchen.
- *Datenflussanalyse* mehrerer Softwaremodule, um nach unerwünschten Zugriffsmustern auf gemeinsame Speicherbereiche zu suchen wie das Lesen nicht-initialisierter Werte (Lesen vor Schreiben) oder das Überschreiben ungelesener Werte (Schreiben nach Schreiben). Dieser Ansatz ist gerade bei nebenläufigen Vorgängen, wie sie mit Software-Tasks in einem Steuergerät gegeben sind, von besonderer Relevanz.

Verlagert sich die Architekturverifikation aufgrund einer unzureichenden Architekturbeschreibung auf die Realisierung, findet die Fehlersuche erst relativ spät und nicht mehr in den frühen Entwicklungsphasen statt. Besonders problematisch ist dabei, dass anhand der Realisierung nicht alle Architekturfehler gezielt aufgedeckt werden können. Ein Beispiel dafür ist eine uneinheitliche Semantik ausgetauschter Daten wie etwa unterschiedliche physikalische Einheiten, da diese in der Software bzw. Hardware

nicht explizit wiederzufinden ist und auch nicht mit den Daten übertragen wird (da sie konstant ist und damit aus Performanzgründen wegoptimiert werden kann).

Erkennbare Architekturfehler
Die Idee der Architekturverifikation ist in der Praxis aufgrund der meist unzureichend beschriebenen Architekturen nicht effektiv einsetzbar. In Kombination mit einer detaillierten Architekturbeschreibung wird dieser Aspekt aber in Abschnitt 7.4.1 noch einmal aufgegriffen.

7.2.3 Tests

Beim Test wird ein System oder ein Teil davon unter definierten Umgebungsbedingungen und mit bestimmten Eingabedaten ausgeführt. Untersucht wird, ob die Ausgabe des Systems korrekt ist. Grundsätzlich werden folgende Testarten unterschieden:

Modultest Beim Modultest wird eine Entwicklungseinheit, typischerweise ein Subsystem, isoliert getestet. Als Vergleichsbasis dient die zugehörige Subsystemspezifikation.

Integrationstest Der Integrationstest erfolgt in mehreren Stufen. Einige oder alle Subsysteme werden einem Integrationsplan folgend integriert und fehlende Subsysteme durch Schnittstellen-Stubs ersetzt. Bei diesen Testläufen wird insbesondere das Zusammenspiel der Subsysteme getestet.

Systemtest Der Systemtest untersucht das vollständig integrierte System. Als Vergleichsbasis dient die Systemspezifikation.

Tests sind eine unerlässliche Maßnahme zur Qualitätssicherung. Hinsichtlich der Erkennung und Vermeidung von Architekturfehlern haben Tests den Nachteil, dass sie relativ spät im Entwicklungsprozess stattfinden und dadurch keinesfalls Fehler bereits in frühen Entwicklungsphasen entdecken können. Wie in Abschnitt 4.2.3 bereits erläutert, sind Modultests zudem gar nicht in der Lage, Architekturfehler aufgrund von Unterspezifikation zu finden, was die Fehlersuche durch Tests weiter nach hinten verlagert. Auch durch eine zyklische Produktentwicklung (siehe Abschnitt 5.2) ergibt sich nicht in jedem Fall ein Beschleunigungseffekt, weil die Tests nicht nach jedem Zyklus in vollem Umfang durchgeführt werden. Ein weiterer Nachteil von Tests ist, dass sie nur eine stichprobenartige Untersuchung darstellen, bei der viele Fehler unentdeckt bleiben können.

Erkennbare Architekturfehler
Mit Tests können prinzipiell alle in Kapitel 5 genannten Fehlerarten erkannt werden. Gerade bei Zeitfehlern ist aber eine geschickte Wahl der Testdaten erforderlich, um beispielsweise selten auftretende Konstellationen nachstellen zu können. Zur Suche

von Architekturfehlern in frühen Entwicklungsphasen haben sich Reviews aber als effektiver als Tests erwiesen [RAT+06].

7.3 Vermeidung von Seiteneffekten in der Architektur

In diesem Kapitel werden konstruktive Maßnahmen vorgestellt.

Werden Ressourcen in einem System von mehreren Subsystemen gemeinsam genutzt, kann es zu Seiteneffekten kommen, also unerwünschter gegenseitiger Beeinflussung der Subsysteme (feature interaction). Als Ressourcen kommen dabei Prozessorzeit, Arbeitsspeicher und Kommunikationsmedien in Betracht.

In diesem Abschnitt werden Eigenschaften einer Architektur vorgestellt, mit deren Hilfe sich bestimmte Architekturfehler vermeiden lassen, ohne dazu den genauen Anwendungsfall kennen zu müssen. Damit geht einher, dass Ressourcen möglicherweise nicht optimal ausgenutzt werden. Andererseits hat dies den Vorteil, dass bestimmte, umfangreiche Nachweise über die Fehlerfreiheit dafür entfallen.

7.3.1 Zeitgesteuerte Kommunikation

Das derzeit im Automobil dominierende Kommunikationsmedium zwischen Steuergeräten ist der CAN-Bus, siehe [ISO03]. Jeder Kommunikationspartner kann zu jeder Zeit versuchen, ein Nachrichtenpaket zu senden. Man spricht hier von einer *ereignisgesteuerten* Kommunikation. Als Ereignis kann das Vorliegen eines Berechnungsergebnisses, bei zyklischen Nachrichten ein lokaler Taktgeber oder ein externes Ereignis wie das Öffnen einer Tür dienen.

Problematisch beim Einsatz einer ereignisgesteuerten Kommunikation ist, dass bereits eine Änderung bei einem Busteilnehmer oder auch das Hinzufügen oder Entfernen eines Busteilnehmers das zeitliche Gefüge der Nachrichtenpakete verändern kann und sich so potenziell auf alle Kommunikationspartner auswirken kann. Hinzu kommt noch, dass mit steigender Busauslastung die Worst-Case-Dauer für den Datenaustausch zunimmt, was für Echtzeitsysteme nur begrenzt akzeptabel ist.

Eine Alternative, die diese Probleme nicht aufweist, ist die *zeitgesteuerte* Kommunikation. Hier werden feste Zeitintervalle vorgegeben, in denen jeweils nur ein Kommunikationspartner ein Paket senden darf. Dieser zeitgesteuerte Zugriff auf ein gemeinsames Kommunikationsmedium wird Time Division Multiple Access (TDMA) genannt. Er führt zur zeitlichen Entkopplung der Kommunikationspartner und bietet folgende Vorteile (siehe auch [EBK03, NSSLW05]):

- Eine Änderung bei einem Busteilnehmer, das Hinzufügen oder das Entfernen eines Busteilnehmers wirken sich zeitlich nicht auf die anderen Kommunikationspartner aus. Das zeitliche Verhalten der Busteilnehmer ist statisch vorhersagbar.

- Jeder Busteilnehmer hat garantierte Buszugriffszeiten und kann nicht durch höherpriore Nachrichten verzögert werden. Die Buszugriffszeiten sind unabhängig von der Busauslastung – aufgrund der Zeitanforderungen eine für Echtzeitsysteme sehr wichtige Eigenschaft.

- Die Busteilnehmer können unabhängig voneinander getestet werden. Dies verringert gerade bei vielen Ausstattungsvarianten die Anzahl der Testfälle enorm.

- Die zeitgesteuerte Kommunikation weist im Unterschied zur ereignisgesteuerten Kommunikation eine geringere Komplexität auf. Ein konservatives Design vereinfacht Korrektheitsnachweise, zumal formale Methoden nicht gut skalieren [Ern03].

In der Praxis gibt es bereits mehrere Bussysteme mit zeitgesteuerter Kommunikation, z. B. FlexRay [Fle05] und TT-CAN [ISO03]. FlexRay und TT-CAN erlauben sogar Zeitintervalle, innerhalb derer wahlfreier Zugriff von allen Busteilnehmern erfolgen kann. Das ist gerade bei sporadischen und selten auftretenden Nachrichtenpaketen ohne Echtzeitanforderung wie bei der Diagnose sinnvoll, um nicht für jede Paketart ein eigenes Zeitintervall reservieren zu müssen und dadurch die Busauslastung unnötig zu verringern. Es ist jedoch nicht ratsam, dieses Feature zu missbrauchen, nur um die höhere Übertragungsgeschwindigkeit bei FlexRay mit bis zu 10 $\frac{\text{Mbit}}{\text{s}}$ gegenüber CAN mit bis zu 1 $\frac{\text{Mbit}}{\text{s}}$ (High-speed CAN) zu nutzen, da hierdurch der Vorteil der Zeitsteuerung kompromittiert wird.

Der Einsatz einer zeitgesteuerten Kommunikation hat allerdings seine Grenzen. Ist eine schnelle Reaktion auf ein unregelmäßiges Ereignis erforderlich und die Wartezeit bis zum nächsten Zeitintervall zu lang, stellt die Zeitsteuerung keine adäquate Lösung dar. Ein Beispiel dafür wäre der Austausch von Nachrichten abhängig von der aktuellen Drehzahl des Motors; bei höherer Drehzahl werden die Nachrichten in kürzeren Zeitabständen gesendet, bei niedrigerer Drehzahl in längeren Zeitabständen.

Vermeidbare Architekturfehler

Mit Hilfe der zeitgesteuerten Kommunikation lassen sich Fehler aufgrund einer „unerwarteten Nachrichtenreihenfolge“ (siehe Kapitel 5) vollständig vermeiden. Durch die zeitlich fest vorgegebene Sendereihenfolge der Nachrichtenpakete erhalten die Empfänger die Nachrichten ebenfalls stets in derselben Reihenfolge.

7.3.2 Statisches Scheduling

Zur effizienten Nutzung der Rechenzeit des Prozessors führt das Betriebssystem auf dem Steuergerät ein Scheduling durch, um die Rechenzeit geeignet auf die Softwaretasks zu verteilen. Mitunter wird das zeitliche Gefüge der Taskausführung aber sehr komplex. Unberücksichtigte Unterbrechungsanforderungen durch externe Ereignisse

und Taskunterbrechungen durch das Scheduling können zu unvorhergesehenen Datenflüssen zwischen den Tasks führen und damit zu einem falschen Berechnungsergebnis. Die empirischen Fehlerdaten in Kapitel 5 belegen dies: Es werden Fehler aufgrund „fehlender Tasksynchronisation“ beobachtet.

Abhilfe lässt sich einerseits durch systematische Tasksynchronisation erreichen, wodurch aber die Komplexität der Abläufe sowie das Deadlock-Risiko weiter steigen. Eine andere Möglichkeit besteht darin, analog zur oben beschriebenen zeitlichen Entkopplung von Steuergeräten auch die Software-Tasks innerhalb der Steuergeräte zeitlich zu entkoppeln. Konkret bedeutet das, dass eine zyklische Abfolge von Zeitintervallen vorgesehen wird und die Zeitintervalle bereits zur Entwurfszeit den einzelnen Tasks zugewiesen werden. Es erfolgt also ein statisches Scheduling. Für mögliche Unterbrechungsanforderungen durch externe Signale sind entsprechende Reserven in jedem Zyklus und evtl. ein eigener Speicherbereich zur Wahrung der Datenkonsistenz zu reservieren. Wichtig ist, dass die Rechenzeiten nicht variieren und die Tasks sich so nicht gegenseitig zeitlich beeinflussen können.

Vermeidbare Architekturfehler
Vorteile des statischen Schedulings sind:

- Die Taskreihenfolgen und deren Ausführungszeiten sind vollständig vorhersagbar,
- auf Tasksynchronisation kann verzichtet werden, d. h. Fehler aufgrund „fehlender Tasksynchronisation“ lassen sich vermeiden, und
- die Komplexität des Kontroll- und Datenflusses in der Steuergerätesoftware wird verringert.

7.3.3 Statische Speicherallokation

Eine weitere Maßnahme zur Vermeidung von Seiteneffekten in der Architektur ist die statische Vergabe des vorhandenen Arbeitsspeichers und des maximal erlaubten Stack-Verbrauchs pro Softwaremodul. Der Stack-Verbrauch ist in der Praxis schwer abzuschätzen, wenn beispielsweise Software von Drittanbietern eingebunden werden soll und der Quelltext nicht zur Verfügung steht.

Die in Kapitel 5 ermittelten Fehlerdaten deuten aber nicht auf ein offenes Problem in der Praxis bezüglich der Speicherverteilung hin.

7.4 Umfassende Schnittstellenbeschreibung

Die meisten Architekturfehler sind auf lückenhafte Architekturbeschreibungen zurückzuführen. Deshalb sind konstruktive Maßnahmen erforderlich, die zu möglichst vollständigen Architekturbeschreibungen führen. Dieser Abschnitt stellt einige derartige Ansätze vor.

Die Darstellung erfolgt aus der Sicht der Neuentwicklung eines Systems, d. h. es wird von einer schrittweisen Entwicklung des Systems ausgegangen und nicht von der Anpassung eines bestehenden Systems. Das bedeutet aber nicht, dass die nachfolgend beschriebenen Maßnahmen nicht dazu geeignet wären, in die Weiterentwicklung eines bereits bestehenden System eingebracht zu werden. Mit steigenden Zuverlässigkeits- und Sicherheitsanforderungen kann das durchaus sinnvoll sein.

7.4.1 Architekturbeschreibungssprachen (ADL)

Eine Architekturbeschreibungssprache (engl. architecture description language, ADL) ist eine Spezifikationssprache zur Beschreibung von Architekturen softwareintensiver Systeme, also insbesondere der inneren Struktur eines Systems und des Zusammenspiels der Subsysteme.

Neben der Identifizierung der Subsysteme und der expliziten Beschreibung aller Schnittstellen bieten ADLs typischerweise Sprachelemente zur Beschreibung von Konnektoren zwischen Aus- und Eingabekanälen. Dazu gehören physikalische Eigenschaften wie Bandbreite, Latenzzeit und Genauigkeit ebenso wie das verwendete Protokoll zum Datenaustausch inklusive Fehlertoleranzmechanismen. ADLs bieten also sowohl Unterstützung zur Beschreibung von funktionalen als auch von nichtfunktionalen Systemeigenschaften. Einen Überblick über ADLs bietet [MT00].

Am Beispiel der Sprache *Architecture Analysis & Design Language* (AADL) [SAE04, FGH06] wird im Folgenden genauer untersucht, welche Architekturfehler mit einer ADL erkannt und vermieden werden können.

Beispiel 7.1 Betrachtet werden soll ein vereinfachtes Brake-by-Wire-System, das die vom Fahrer ausgeübte Pedalkraft in ein entsprechendes Bremsmoment an den Rädern übersetzt. Das System besteht aus den drei Subsystemen Pedal, Bremssteuergerät und Bremsaktorik, die seriell miteinander verbunden sind. Abbildung 7.1 zeigt einen grafischen Überblick über die Architektur und Listing 7.1 einen Ausschnitt der entsprechenden textuellen Darstellung in AADL. Das vollständige Listing ist in Anhang B angegeben.

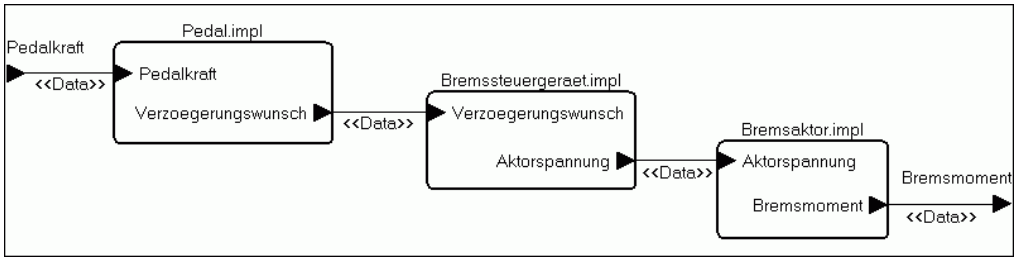


Abbildung 7.1: Architektur des Brake-by-Wire-Systems in grafischer Notation

```
1  system implementation Bremssystem.impl
2    subcomponents
3      Pedal.impl: system Pedal;
4      Bremssteuergeraet.impl: system Bremssteuergeraet;
5      Bremsaktor.impl: system Bremsaktor;
6    connections
7      c1: data port Pedalkraft -> Pedal.impl.Pedalkraft;
8      c2: data port Pedal.impl.Verzoegerungswunsch
9          -> Bremssteuergeraet.impl.Verzoegerungswunsch;
10     c3: data port Bremssteuergeraet.impl.Aktorspannung
11         -> Bremsaktor.impl.Aktorspannung;
12     c4: data port Bremsaktor.impl.Bremsmoment
13         -> Bremsmoment;
14 end Bremssystem.impl;
```

Listing 7.1: Beschreibung der Kommunikationsstruktur in AADL

In der Beschreibung des Systems *Bremssystem* werden im Abschnitt `subcomponents` die drei Subsysteme aufgelistet und im Abschnitt `connections` die Verbindungen zwischen den Subsystemen sowie zur Systemschnittstelle hin. AADL unterscheidet streng zwischen einem abstrakten Typ und seiner Implementierung. Letztere ist im Listing am (willkürlich) gewählten Zusatz `.impl` erkennbar. Die Subsysteme selbst werden an einer anderen Stelle im AADL-Dokument beschrieben, wo auch sie als System definiert werden. AADL-Spezifikationen sind also modular aufgebaut.

Die gezeigte Systembeschreibung enthält aber nur die Struktur der internen Kommunikation. Eine genauere Beschreibung der ausgetauschten Nachrichten enthält die abstrakte Typdefinition. Listing 7.2 zeigt einen entsprechenden Ausschnitt. Darin werden die Datentypen `PedalkraftData` und `VerzoegerungswunschData` definiert, jeweils mit Angabe der oberen und unteren Grenze des Wertebereichs, der physikalischen Einheit und des Signaltyps bzw. der Bedeutung des Vorzeichens. AADL kann bezüglich der definierbaren Eigenschaften beliebig erweitert werden. Die hier verwendeten Präfixe `semantic` und `machine` stehen für die Semantikdarstellung und Maschinendarstellung der Daten gemäß Abschnitt 4.3.1. Das abstrakte System `Pedal` hat den Eingabekanal `Pedalkraft` und den Ausgabekanal `Verzoegerungswunsch` mit den jeweils zuvor definierten Nachrichtentypen.

Auch das Zeitverhalten kann mittels AADL beschrieben werden. Dazu werden die möglichen Datenflüsse explizit spezifiziert und mit Zeitanforderungen versehen, siehe Listing 7.3. Das abstrakte System `Bremssystem` definiert zunächst nur die Systemschnittstelle in Black-Box-Sicht und verlangt, dass der Datenfluss zwischen Ein- und Ausgabe maximal 50 ms benötigt. Die Systemimplementierung `Bremssystem.impl` beschreibt den inneren Aufbau des Systems, also die bereits bekannten Subsysteme,

```
1  data PedalkraftData
2    properties
3      semantic::min_value => 0.0;
4      semantic::max_value => 500.0;
5      semantic::physical_unit => "N";
6      machine::signal_type => analogue;
7  end PedalkraftData;
8
9  data VerzoegerungswunschData
10   properties
11     semantic::min_value => 0.0;
12     semantic::max_value => 10.0;
13     semantic::physical_unit => "m/s^2";
14     machine::leading_sign_meaning
15       => "entgegen Fahrtrichtung";
16 end VerzoegerungswunschData;
17
18 system Pedal
19   features
20     Pedalkraft: in data port PedalkraftData;
21     Verzoegerungswunsch: out data port
22       VerzoegerungswunschData;
23 end Pedal;
```

Listing 7.2: Beschreibung der Nachrichtentypen in AADL

die Verbindungen dazwischen mit jeweils praktisch verzögerungsfreier Datenübertragung sowie den internen Datenfluss. Aufgrund des modularen Aufbaus von AADL-Spezifikationen werden die Datenflüsse innerhalb der Subsysteme nur referenziert. Das genaue Zeitverhalten ist dann in den Spezifikationen der einzelnen Subsysteme zu finden.

```
1  system Bremssystem
2    features
3      Pedalkraft: in data port PedalkraftData;
4      Bremsmoment: out data port BremsmomentData;
5    flows
6      f: flow path Pedalkraft -> Bremsmoment
7         { Expected_Latency => 50 Ms; };
8  end Bremssystem;
9
10 system implementation Bremssystem.impl
11   subcomponents
12     Pedal.impl: system Pedal;
13     Bremssteuergeraet.impl: system Bremssteuergeraet;
14     Bremsaktor.impl: system Bremsaktor;
15   connections
16     c1: data port Pedalkraft -> Pedal.impl.Pedalkraft
17         { Latency => 0 Ms; };
18     c2: data port Pedal.impl.Verzoegerungswunsch
19         -> Bremssteuergeraet.impl.Verzoegerungswunsch
20         { Latency => 0 Ms; };
21     c3: data port Bremssteuergeraet.impl.Aktorspannung
22         -> Bremsaktor.impl.Aktorspannung
23         { Latency => 0 Ms; };
24     c4: data port Bremsaktor.impl.Bremsmoment -> Bremsmoment
25         { Latency => 0 Ms; };
26   flows
27     f: flow path Pedalkraft -> c1 -> Pedal.impl.flow1
28        -> c2 -> Bremssteuergeraet.impl.flow2
29        -> c3 -> Bremsaktor.impl.flow3
30        -> c4 -> Bremsmoment;
31 end Bremssystem.impl;
```

Listing 7.3: Beschreibung des Zeitverhaltens in AADL

□

Mit AADL lassen sich vielfältige funktionale und nicht-funktionale Eigenschaften einer Architektur beschreiben. Der Sprachumfang geht deutlich über das oben gezeigte Beispiel hinaus und erlaubt u. a. auch die Spezifikation von Software-Threads, Speicher

und Prozessoren. Die Sprache kann um die Beschreibung beliebiger Eigenschaften, wie im obigen Beispiel bereits demonstriert, erweitert werden.

Vermeidbare Architekturfehler

Aufgrund der strengen Typisierung mittels abstrakter Typen lassen sich prinzipiell alle Datenfehler in der Semantikdarstellung in Kapitel 5 vermeiden. Führt man beispielsweise zwei Datentypen *Tachogeschwindigkeit* und *Fahrzeuggeschwindigkeit* ein, werden diese von AADL als zwei unterschiedliche Typen gesehen, auch wenn sie in allen zusätzlich angegebenen Eigenschaften übereinstimmen sollten. Dadurch lassen sich Typfehler in der Semantikdarstellung sofort erkennen. Datenfehler in der Maschinendarstellung lassen sich nur teilweise vermeiden. Fehler wie *unterschiedliche physikalische Einheiten*, *falscher Wertebereich* und *Bit doppelt belegt* sind leicht vermeidbar, allerdings bietet AADL im Sprachumfang keine gute Unterstützung für die Beschreibung einer Quantisierung oder der Kennzeichnung von Werten mit einer Sonderbedeutung wie *ungültig* oder *nicht verfügbar*. Zur Vermeidung von Datenfehlern in der physikalischen Darstellung sieht AADL keine vordefinierten Sprachelemente vor, diese ist nur über selbst zu definierende Zusatzeigenschaften erreichbar.

Bei der Vermeidung der in Kapitel 5 genannten Zeitfehler brilliert AADL geradezu. Diese lassen sich prinzipiell vollständig vermeiden. AADL bietet dazu die explizite Spezifikation von Datenflüssen inklusive Zeitverbrauch, Beschreibung zyklisch ablaufender Software-Threads, Kommunikation über separate Kanäle sowie die Beschreibung von Betriebsmodi und ihrer Übergänge. Letztere eignen sich besonders zur Beschreibung von Ausfallszenarien und den dadurch geänderten Datenflüssen. Tabelle 7.1 fasst die vermeidbaren Fehler zusammen.

	Datenfehler	Zeitfehler
Semantikdarstellung	++	++
Maschinendarstellung	teilweise	++
Physikalische Darstellung	−−	++

Tabelle 7.1: Mit AADL vermeidbare Fehlerklassen

Trotz der Vielfalt an Architekturfehlern, die mit AADL vermeidbar sind, ist die Effektivität an mehrere Voraussetzungen gebunden. Erstens fordert die Sprache keinen Mindestumfang an zu beschreibenden Eigenschaften. Es ist durchaus erlaubt, keinerlei Eigenschaften anzugeben. Auch Datentypen müssen außer ihrem Namen keine weiteren Angaben enthalten. Im Hinblick auf eine umfassende Schnittstellenbeschreibung hilft der Einsatz von AADL allein nicht automatisch weiter. Erst in Verbindung mit Vorgaben bezüglich Modellierungsgrundsätzen kann AADL seine Vorteile voll entfalten. Zweitens ist AADL ohne adäquaten Werkzeugeinsatz nicht praktikabel einsetzbar. Unbedingt erforderlich sind ein grafisches Modellierungswerk-

zeug, ein Konsistenzprüfer für Schnittstellen und weitere Modelleigenschaften sowie ein Codegenerator, um den Schnittstellencode für die Implementierung automatisch zu erzeugen und die Durchgängigkeit im Entwicklungsprozess zu unterstützen. Das frei verfügbare Werkzeug OSATE [Osa07] erlaubt erste Versuche mit AADL, ist aber für den Einsatz im großen Stil noch zu instabil, zu wenig dokumentiert und bietet zu wenig Funktionalität für den Praxiseinsatz.

7.4.2 Spezifikationssprachen zur Schnittstellenbeschreibung

Im Unterschied zu ADLs im vorherigen Abschnitt haben Spezifikationssprachen zur Schnittstellenbeschreibung die Verhaltensbeschreibung eines einzelnen (Sub-)Systems aus Black-Box-Sicht zum Ziel.

Damit eine Spezifikationssprache zur Schnittstellenbeschreibung in der Praxis einen ausreichenden Nutzen bringen kann, müssen mindestens folgende Voraussetzungen erfüllt sein, die im Folgenden näher erläutert werden.

- Ausreichender Sprachumfang
- Abstraktion von der Realisierung
- Große Verbreitung der Sprache
- Gute Werkzeugunterstützung
- Einbindung in den Entwicklungsprozess

Mit der Spezifikationssprache müssen alle wichtigen Schnittstelleneigenschaften beschrieben werden können. Die Ausdrucksmächtigkeit sollte mindestens alle Angaben, die in der *Vorlage zur Schnittstellenspezifikation* in Anhang A genannt sind, umfassen. Diese Vorlage eignet sich gut als Maßstab zur Einschätzung von Spezifikationssprachen bezüglich der Vermeidung von Architekturfehlern, weil die darin referenzierten Schnittstelleneigenschaften aus den in Kapitel 5 identifizierten Spezifikationslücken abgeleitet worden sind. In der Regel fokussiert eine Spezifikationssprache bestimmte Aspekte, zu deren Beschreibung sie besonders gut geeignet ist, und kann daher möglicherweise nicht alle geforderten Eigenschaften auf einmal abdecken. In diesem Fall ist auch die Kombination mehrerer unterschiedlicher Spezifikationssprachen denkbar, was aber den erforderlichen Aufwand erhöht. Ein wichtiges Merkmal einer Sprache ist eine möglichst eindeutige Semantik, um unterschiedliche Interpretationen zu vermeiden.

Eine Spezifikation sollte zudem eine Abstraktion von der Realisierung darstellen, die sich auf wesentliche Schnittstelleneigenschaften beschränkt und unwesentliche Implementierungs- und Realisierungsdetails nicht vorwegnimmt.

Des Weiteren ist es förderlich, wenn die Spezifikationssprache eine große Verbreitung hat, d. h. auch von Entwicklern eingesetzt wird, die an Nachbarsubsystemen

arbeiten. Im Umfeld der Automobilindustrie bedeutet das, dass alle Zulieferer möglichst dieselbe Sprache zur Schnittstellenbeschreibung verwenden wie die Hersteller. Das erleichtert die Konsistenzprüfung zwischen zusammengehörenden Schnittstellen in einer Architektur bzw. macht bestimmte Analysen überhaupt erst möglich. Ein hoher Verbreitungsgrad einer Spezifikationssprache setzt meist die Standardisierung und eine allgemeine Akzeptanz der Sprache voraus. Bekannte Beispiele hierfür sind Manufacturer Supplier Relationship (MSR) [MSR07] und Automotive Open System Architecture (Autosar) [Aut07].

Die Vorteile einer Spezifikationssprache lassen sich aber erst dann vernünftig nutzen, wenn der Einsatz mit geeigneter Software-Werkzeugunterstützung erfolgt. Die meisten Spezifikationssprachen haben eine kompakte, textuelle Notation, die erst erlernt werden muss. Hier bietet ein Werkzeug mit einer grafischen Darstellung der Spezifikation oder zumindest einer von der Sprachsyntax abstrahierten Darstellung der am häufigsten benötigten Schnittstelleneigenschaften wertvolle Dienste. Das spart nicht nur Zeit bei der Beschreibung von (Sub-)Systemen, sondern ermöglicht zusätzlich weitergehende Konsistenzprüfungen der Schnittstellen in der Architektur sowie die automatische Generierung von Schnittstellencode. Die Sprache selbst ist dann evtl. für den Anwender nicht mehr direkt sichtbar, bildet aber das formale Rückgrat und definiert die Semantik der erstellten Spezifikation. Die Einführung eines Software-Werkzeugs muss allerdings immer im Gesamtkontext des Unternehmens betrachtet werden (siehe [LL06], Kap. 15).

Doch das Bereitstellen einer Spezifikationssprache samt Werkzeugunterstützung nützt allein wenig, wenn der genaue Einsatz dieser Hilfsmitteln nicht durch den Entwicklungsprozess geregelt ist. Die in Kapitel 5 entdeckten Schwachstellen in den Entwicklungsprozessen belegen dies anschaulich. Es genügt nicht, wenn der Entwicklungsprozess empfiehlt oder vorschreibt, *dass* ein bestimmtes Werkzeug eingesetzt wird. Vielmehr muss der Entwicklungsprozess vorschreiben, wie und in welchem Umfang die vorgesehenen Hilfsmittel einzusetzen sind. Im Falle einer Spezifikationssprache sollte explizit vorgegeben sein, welche Schnittstelleneigenschaften mindestens zu beschreiben sind. Dazu kann beispielsweise die Vorlage in Anhang A herangezogen werden. Teile eines Entwicklungsprozesses können dabei durchaus in Werkzeuge gegossen sein, indem das Werkzeug z. B. unvollständige Angaben zurückweist.

Beispiel 7.2 Als konkrete Beispiele sollen *MSR* und *Autosar* im Folgenden etwas näher betrachtet werden. MSR ist eine standardisierte Spezifikationssprache auf Basis von XML, die zum Austausch von Anforderungen, Spezifikationen und Dokumentationen zwischen Automobilherstellern und Zulieferern entwickelt wurde. Autosar ist das Resultat einer Bestrebung mehrerer Automobilhersteller und Zulieferer, die Schnittstellen von Software-Komponenten im Automobil zu standardisieren. Vergleicht man den Sprachumfang von MSR (MSRSW V2.3.0) mit Autosar (V3.0) anhand der Vorlage in Anhang A, ergeben sich kaum Unterschiede, wie Tabelle 7.2 zeigt.

Schnittstelleneigenschaft	MSR	Autosar
Name des Signals	✓	✓
Name des zugehörigen (Sub-)Systems	✓	✓
Signalrichtung (Eingabe/Ausgabe)	✓	✓
informelle Bedeutung	✓	✓
physikalische Einheit	✓	✓
Grundmenge des Wertebereichs	nur implizit	✓
Grenzen des Wertebereichs	✓	✓
Signaltyp (analog/digital)	nur digital	nur digital
Anzahl Bits	✓	✓
Semantik der Bitbelegungen		
Datentyp der Maschine	✓	✓
Bytereihenfolge	✓	✓
Bitreihenfolge innerhalb eines Byte	✓	✓
Bedeutung des Vorzeichens	–	–
Umrechnung Semantik-/Maschinendarstellung	✓	✓
Genauigkeit (absolut bzw. relativ)	–	–
Bedeutung der Fehlerwerte	–	–
Zeitverhalten		
minimales und maximales Informationsalter	–	–
Aktualisierungszeitpunkte		
Synchronisierungsereignis	–	–
Sendefrequenz	–	–
maximaler Zeit-Jitter	–	–
Zeitverhalten (kontinuierlich/diskret)	–	–
Signal-zu-Rausch-Abstand (in dB)	–	–
Entprellzeit bei Signalflanke	–	–

Tabelle 7.2: Mit MSR und Autosar beschreibbare Schnittstelleneigenschaften

Die Untersuchung von Autosar bezieht sich nur auf die Schnittstellenbeschreibung der Softwarekomponenten (Software Component Template). Die große Ähnlichkeit zu MSR ist zunächst etwas überraschend, da MSR auf einem XML-Schema basiert, die Architekturbeschreibung in Autosar jedoch auf einem UML-Metamodell. Allerdings stand MSR Pate bei der Entwicklung von Autosar – viele Sprachelemente von MSR sind in Autosar wiederzufinden.

Betrachtet man die oben genannten Voraussetzung für den Einsatz einer Spezifikationssprache, ergibt sich folgendes Bild.

Beide Sprachen haben eine ausgeprägte Ausdrucksmächtigkeit bei der Beschreibung syntaktischer Schnittstelleneigenschaften, sowohl in der Semantikdarstellung als auch in der Maschinendarstellung. Eine große Schwachstelle ist hingegen die Beschreibung des Zeitverhaltens an der Schnittstelle. Aus der Beschreibung von Software-Tasks ließe sich zwar bei beiden Sprachen implizit z. B. die Zykluszeit eines periodisch zu sendenden Signals ableiten, das wäre aber keine Schnittstellenbeschreibung im Black-Box-Sinn. Die Beschreibung der physikalischen Darstellung ist ebenfalls nicht möglich. Der Sprachumfang von MSR und Autosar ist damit als nicht ausreichend einzustufen.

Gut erfüllt ist hingegen die Abstraktion von der Realisierung bei der Beschreibung des Schnittstellenverhaltens. Beide Sprachen unterscheiden zwischen Semantikdarstellung und Maschinendarstellung und können die Konvertierung von Nachrichten zwischen den beiden Darstellungen angeben. Auch die große Verbreitung der Sprachen, begründet durch die Beteiligung mehrerer Automobilhersteller und Zulieferer bei deren Entwicklung, ist ein großer Vorteil für den Praxiseinsatz.

Für den Einsatz von MSR gibt es bereits eine Reihe von Software-Werkzeugen, von XML-Editoren bis hin zur Überprüfung der Schnittstellenkonsistenz. Autosar ist noch in der Entwicklung, so dass es für die Frage nach der Werkzeugunterstützung noch zu früh ist. □

Neben MSR und Autosar erfüllen viele der in der Literatur zu findenden Spezifikationssprachen nicht alle der oben angegebenen Voraussetzungen (auch ohne Berücksichtigung der Einbindung in den Entwicklungsprozess). Hier besteht noch deutlicher Entwicklungsbedarf.

Vermeidbare Architekturfehler

Mit Spezifikationssprachen zur Schnittstellenbeschreibung lassen sich prinzipiell alle der in Kapitel 5 genannten Architekturfehler dauerhaft vermeiden. Dazu ist aber eine Kombination mehrerer Spezifikationssprachen erforderlich, da keine der betrachteten Sprachen alleine die oben genannten Voraussetzungen erfüllt. Der Einsatz von Spezifikationssprachen ist in jedem Fall mit einem deutlich höheren Aufwand für die Einführung und die Anwendung der Sprache verbunden als beispielsweise beim Review.

7.5 Wiederherstellen konsistenter Schnittstellen

Gerade bei größeren Softwarebestandteilen in eingebetteten Systemen findet die Softwareentwicklung meistens durch Anpassung bereits bestehender Software statt und weniger durch Neuentwicklung. Dieser Abschnitt betrachtet konstruktive Maßnahmen zur Fehlervermeidung, wenn bestehende Systeme an neue Anforderungen angepasst werden.

Jede Änderung in einem System beeinflusst im Allgemeinen das Schnittstellenverhalten eines Subsystems. Ist die veränderte Schnittstelle bzw. das veränderte Verhalten nicht mehr mit den Nachbarsubsystemen konsistent, sind auch diese anzupassen. Das Ziel ist es, wieder konsistente Schnittstellen zu erhalten. Dies erfolgt unter der Voraussetzung, dass die Schnittstellen bereits vorher konsistent waren.

7.5.1 Kennzeichnen und Propagieren von Schnittstellenänderungen

Ausgehend von einer Änderung im System erfolgt die Wiederherstellung konsistenter Schnittstellen iterativ, indem schrittweise die Änderungen bezüglich einer Schnittstelle identifiziert, die Änderungen aller betroffenen Nachbarschnittstellen propagiert und dort die Änderungen eingepflegt werden. Im Extremfall kann sich eine Änderung in einem Subsystem auf alle anderen Subsysteme auswirken. Auf die Fortpflanzung einer Änderung wurde bereits in Abschnitt 6.2 näher eingegangen.

Kennzeichnung von Änderungen Zunächst sind alle Änderungen bezüglich einer Schnittstelle zu identifizieren. Dazu gehören jegliche Änderungen des Schnittstellenverhaltens, also sowohl Änderungen in der Datendarstellung als auch Änderungen im Zeitverhalten. Dabei kommt es besonders darauf an, dass auch kleinste Änderungen berücksichtigt werden.

Wurde die veränderte Schnittstelle mit Hilfe einer Spezifikationssprache beschrieben, lassen sich Änderungen direkt an der Spezifikation ablesen. Dieser Idealfall ist in der Praxis aber nicht immer gegeben. Dann empfiehlt sich die Kontrolle von Änderungen mit Hilfe der Schnittstellenvorlage in Anhang A.

Die identifizierten Schnittstellenänderungen können ins bestehende Konfigurationsmanagement eingetragen werden. Eine Kernaufgabe des Konfigurationsmanagements ist die Konfigurationskontrolle, d. h. sicherzustellen, dass die Bestandteile einer Konfiguration konsistent sind (siehe [LL06], Kap. 21). Die Änderung kann beispielsweise mittels eines SpecChanged-Flags für die jeweilige Schnittstelleneigenschaft gekennzeichnet werden.

Propagieren der Änderungen Mit Hilfe der eingetragenen SpecChanged-Flags und der Kommunikationsstruktur der Architektur kann auf einfache Weise ermittelt

werden, welche Nachbarsubsysteme betroffen sind, und vor allem, welche Schnittstelleneigenschaften im Einzelnen auf Anpassung zu untersuchen sind. Das Ziel dieses Schritts ist herauszufinden, ob auch das jeweilige Nachbarsubsystem zu ändern ist.

Über die Verantwortlichkeiten der Entwickler ergibt sich dadurch auch, welchen Entwickler eine Änderung betrifft und wie groß die Auswirkung auf sein Subsystem ist.

Betroffene Subsysteme anpassen Alle betroffenen Subsysteme sind anschließend anzupassen, so dass die Schnittstellenkonsistenz wieder hergestellt wird. Dabei ist natürlich nicht ausgeschlossen, dass durch diese Änderung weitere Schnittstellenkanäle beeinflusst werden, worauf die genannten Schritte erneut durchlaufen werden müssen.

Wird das Schnittstellenverhalten eines (Sub-)Systems geändert, so ist dabei zu beachten, dass die Änderungen stets auf allen Modellebenen konsistent zu erfolgen haben, d. h. eventuelle Anforderungen an die Schnittstelle im Lasten- bzw. Pflichtenheft, die Architekturbeschreibung sowie eine bereits vorhandene Implementierung müssen stets konform zueinander sein. Dieses Vorgehen wird als *roundtrip engineering* bezeichnet.

Vermeidbare Architekturfehler

Mit dem Kennzeichnen und Propagieren von Schnittstellenänderungen lassen sich prinzipiell alle der in Kapitel 5 genannten Fehlerarten vermeiden. Der Erfolg hängt aber entscheidend davon ab, wie effektiv die Schnittstellenänderungen erkannt werden. Wird dazu ein Review eingesetzt, ist die Erfahrung der Gutachter ausschlaggebend. Beim Einsatz einer Spezifikationssprache zur Schnittstellenbeschreibung lassen sich je nach Werkzeugunterstützung durch statische Analyse oder Simulation die geänderten Schnittstelleneigenschaften bestimmen.

7.5.2 Wiederverwenden konsistenter Systemkonfigurationen

Um zu vermeiden, dass die Schnittstellenkonsistenz einer Menge von Subsystemen unnötigerweise mehrfach nachgewiesen wird, kann man konsistente Konfigurationen festhalten.

Der Ansatz, konsistente Konfigurationen für das gesamte System festzuhalten, wird von Ehlers [Ehl03] verfolgt. Hier wird eine Datenbank aufgebaut, in der alle Software- und Hardware-Versionen einer als konsistent nachgewiesenen Fahrzeugkonfiguration abgelegt werden. Wird nun eine Komponente durch eine andere Version ersetzt, so ergibt sich aus der Datenbank sofort, welche anderen Komponenten ebenfalls zu ersetzen sind, damit wieder eine bekannte, konsistente Konfiguration erreicht wird. Damit dieser Ansatz von praktischem Nutzen ist, müssen bekannte Konfigurationen

häufig auftreten. Bei der Weiterentwicklung eines bestehenden Systems entstehen aber nur neue Komponentenversionen, über die in der Datenbank noch keine Aussage enthalten sein kann. In diesem Fall kann der beschriebene Ansatz in dieser Form keine Einsparung bringen.

Auf der Ebene von Teilsystemen hingegen findet obiger Ansatz in ähnlicher Weise schon heute praktische Anwendung. In einem Bosch-Geschäftsbereich wird die Software eines Steuergeräts beispielsweise in sogenannte Hauptfunktionen (engl. main functions) unterteilt (siehe Kapitel 5). Jede Hauptfunktion stellt dabei eine Softwareeinheit dar, die in sich konsistent ist. Durch diese quasi vorintegrierten Softwarekomponenten erreicht man eine Einsparung bei der Integration.

Die Ausweitung des Ansatzes der vorintegrierten Subsysteme auf größere Systembestandteile – auch als *Plattform* bezeichnet – verspricht nicht in jedem Fall einen Gewinn. Durch die im Automobilbereich recht ausgeprägte Variantenvielfalt explodiert die Anzahl der Kombinationsmöglichkeiten. Dadurch ist die Pflege einer konsistenten Plattform nur im Low-Cost-Bereich realistisch und auch zu empfehlen.

Vermeidbare Architekturfehler

Die Wiederverwendung konsistenter Systemkonfigurationen zielt nicht auf die unmittelbare Vermeidung bestimmter Architekturfehler ab, sondern hilft allgemein, den Aufwand für die Integration zu senken, weil vorintegrierte Softwareeinheiten tendenziell häufiger wiederverwendet werden. Dadurch verringert sich auch die Anzahl der auftretenden Architekturfehler.

7.6 Weitere Unterstützung durch den Entwicklungsprozess

7.6.1 Rolle des Systemintegrators

Die Untersuchung aktueller Entwicklungsprozesse hat u. a. ergeben, dass bei der arbeitsteiligen Entwicklung größerer Systeme die Subsystemunterteilung des Systems der Teamaufteilung der Entwickler folgt (siehe Abschnitt 5.2.4). Viele Subsystemschnittstellen erfordern somit ein häufiges Abstimmen zwischen Entwicklerteams, damit beispielsweise Schnittstellenänderungen an einem Subsystem an die Entwickler der betroffenen Nachbarsubsysteme kommuniziert werden. Eine solche Vorkehrung sehen die heute eingesetzten Entwicklungsprozesse nicht explizit vor.

Dieses Problem kann beseitigt werden, indem ein *Verbundverantwortlicher* berufen wird, der speziell das Zusammenspiel der Subsysteme verantwortet und dieses bereits bei der Erstellung der Architekturbeschreibung und auch der Realisierung überwacht und ggf. beeinflusst. Das V-Modell XT [RB07] ordnet diese Aufgabe dem *Systemintegrator* zu.

Vermeidbare Architekturfehler
Durch einen Systemintegrator lassen sich – analog zum Review – prinzipiell alle der in Kapitel 5 genannten Architekturfehler vermeiden. Der Erfolg ist aber nicht garantiert und hängt stark von der Erfahrung des Systemintegrators ab. Im Unterschied zum Review stellt die Installation eines Systemintegrators eine zeitlich durchgängige, entwicklungsbegleitende Maßnahme dar, mit der Fehler nicht nur frühzeitig erkannt, sondern sogar vermieden werden können.

7.6.2 Erweiterung des Schnittstellen-Begriffs

Wie in den Abschnitten 5.1.2 und 5.2 bereits ausführlich beschrieben, umfasst der bei den untersuchten Bosch-Geschäftsbereichen verwendete Schnittstellen-Begriff im Wesentlichen nur syntaktische Schnittstelleneigenschaften. Dem durch den Echtzeitcharakter eingebetteter Systeme ebenso wichtigen Zeitverhalten wird hier fast gar nicht Rechnung getragen. Dadurch greifen etliche Prozessziele wie das *Beibehalten der Kompatibilität der Schnittstellen* im Änderungsmanagement (siehe Abschnitt 5.2.2) nicht im erforderlichen Umfang.

Daher wird empfohlen, den Schnittstellen-Begriff zu erweitern, so dass auch das Zeitverhalten mit einbezogen wird. Eine konkrete Auflistung aller Schnittstelleneigenschaften liefert die Vorlage zur Schnittstellenspezifikation in Anhang A.

Vermeidbare Architekturfehler
Die Erweiterung des Schnittstellen-Begriffs allein wird in der Praxis kaum unmittelbar Fehler vermeiden. Allerdings wird dadurch die Notwendigkeit für zusätzlichen Handlungsbedarf aus den Prozessvorgaben heraus unterstrichen. Man macht sich hier die Tatsache zu Nutze, dass die Prozessbeschreibungen bereits in der heutigen Form auch auf Zeitanforderungen anwendbar sind.

7.7 Zusammenfassung

Dieses Kapitel schlägt unterschiedliche Maßnahmen zur Erkennung und Vermeidung von Architekturfehlern vor und bewertet diese. Sie werden zum besseren Verständnis getrennt in analytische und in konstruktive Maßnahmen, während in der Praxis immer eine Kombination davon zum Einsatz kommen wird.

Abhängig von der geforderten Produktqualität können unterschiedlich teure Maßnahmen eingesetzt werden. Diese reichen von Reviews mit vergleichsweise geringem Aufwand bis hin zu Spezifikationssprachen samt unterstützenden Werkzeugen mit vergleichsweise hohem Aufwand für Einführung und Durchführung.

Durch die bereits recht umfangreichen Prozessbeschreibungen (siehe Kapitel 5) genügt oft schon die Optimierung bestehender Maßnahmen wie der Reviews und

des Änderungsmanagements, um einen deutlichen Effekt in der Erkennung und Vermeidung von Architekturfehlern zu erzielen. Mit steigenden Qualitätsanforderungen, insbesondere der funktionalen Sicherheit (safety), wird jedoch eine umfassende Architekturbeschreibung mittels entsprechender Spezifikationssprachen unumgänglich werden.

Mit den vorgestellten Maßnahmen lassen sich alle in Kapitel 5 genannten Architekturfehler vermeiden. Nicht immer ist allerdings der dafür erforderliche Kosteneinsatz zu rechtfertigen. Deshalb ist eine sinnvolle Balance zwischen Kosten und erreichbarer Produktqualität anzustreben. Dieses Thema wird in Kapitel 8 näher behandelt.

8 Kostenoptimaler Einsatz von Qualitätssicherungsmaßnahmen

In Kapitel 7 wurden unterschiedliche Maßnahmen vorgestellt, mit deren Hilfe Fehler frühzeitig erkannt und vermieden werden können, um dadurch eingebettete Systeme mit hoher Qualität herzustellen. Prinzipiell stehen zwei grundsätzliche Herangehensweisen zur Verfügung. Zum einen kann man mit systematischen Tests und Reviews die in der Entwicklung entstandenen Fehler aufspüren und beseitigen, zum anderen kann man Fehlervermeidungsmaßnahmen einsetzen, die das Entstehen von Fehlern von Anfang an verhindern und so per Konstruktion zu einem Produkt mit hoher Qualität führen.

Keine der beiden Herangehensweisen ist in allen Fällen die bessere Wahl. Vielmehr wird in der Praxis eine geeignete Kombination davon die optimale Lösung sein. Im industriellen Umfeld entscheidet letzten Endes die Wirtschaftlichkeit über die Auswahl der Maßnahmen. Dieses Kapitel untersucht die Auswirkungen von Fehlervermeidungsmaßnahmen auf die Produktkosten und zeigt auf, wie sich die richtige Balance finden lässt.

8.1 Qualität, Preis und Kosten

Definition 8.1 (Qualität)
Qualität ist der Grad, in dem ein Satz inhärenter Merkmale die vorgegebenen Anforderungen erfüllt. [DIN05] □

Die zu erfüllenden Anforderungen können vielfältig sein. Typischerweise gehören dazu die Zuverlässigkeit, Sicherheit und Wartbarkeit des Produkts.

Qualität und Preis

Der Zusammenhang zwischen der Qualität eines Produkts und dessen Verkaufspreis ist keinesfalls klar gegeben. Die Vermutung, dass sich der Preis unmittelbar nach der Qualität richtet, ist im Allgemeinen nicht richtig. Im Consumer-Bereich konnte empirisch nur eine leicht positive Korrelation zwischen Qualität und Preis nachgewiesen werden. Teilweise wird vom Kunden ein höherer Preis oder ein größerer Marktanteil des Verkäufers mit höherer Qualität assoziiert. Hier spielt die Marktpsychologie eine große Rolle. Im industriellen Umfeld basieren Kaufentscheidungen auf deutlich besseren

technischen und wirtschaftlichen Informationen und werden daher viel rationaler getroffen [JG99].

Qualität und Kosten

Ein klareres Bild zeichnet sich ab, wenn man Qualität und die damit verbundenen Kosten vergleicht. Betrachtet werden dabei die *Qualitätskosten*, also der Aufwand für die Sicherung und Steigerung der Qualität sowie Aufwand aufgrund mangelnder Qualität [LL06]. Darin nicht enthalten sind die Netto-Herstellungskosten und Wartungskosten (ohne Qualitätskosten). Die Kosten lassen sich wie folgt unterteilen [JG99, Wag07].

Konformanzkosten Unter *Konformanzkosten* (conformance costs) fallen alle Kosten, die zur Erstellung eines spezifikationskonformen Produkts, das alle Qualitätsanforderungen erfüllt, anfallen. Diese setzen sich aus Vermeidungs- und Prüfkosten zusammen.

Vermeidungskosten (prevention costs) sind Kosten zur Vermeidung von Fehlern wie Mitarbeiterschulungen, Anforderungsmanagement, Projektmanagement, Toolunterstützung, Richtlinien und weitere konstruktive Qualitätssicherungsmaßnahmen (siehe Abschnitt 7.1).

Prüfkosten (appraisal costs) entstehen beim Nachweis der Konformanz des Produkts durch analytische Qualitätssicherungsmaßnahmen wie diverse Testarten und Reviewverfahren.

Fehlerkosten Als *Fehlerkosten* (nonconformance costs) werden alle Kosten bezeichnet, die aufgrund von Abweichungen des Produkts von der Produktspezifikation entstehen. Dazu zählen die Suche nach der Fehlerursache und die anschließende Behebung des Fehlers, weitere Nacharbeiten aufgrund nichterfüllter Qualitätsanforderungen wie das Umgestalten der Architektur bei zu langen Reaktionszeiten, Anpassungen durch geänderte Anforderungen sowie Folgekosten. Die Fehlerkosten werden unterteilt in interne und externe Kosten.

Interne Fehlerkosten (internal failure costs) sind diejenigen Fehlerkosten, die während der Entwicklung, also noch vor Auslieferung an den Kunden, entstehen.

Externe Fehlerkosten (external failure costs) hingegen sind Fehlerkosten, die nach der Auslieferung an den Kunden entstehen. Mit enthalten sind Kosten für

- die Behebung von Fehlern, die beim Kunden auftreten, Rückrufaktionen bei fehlerhafter Serienproduktion und sonstige Gewährleistungsansprüche des Kunden,
- Schadensersatz und Strafzahlungen aufgrund der gesetzlichen Produkthaftung, wenn der *Stand der Technik* nicht eingehalten wurde, und

- langfristige Auswirkungen aufgrund unzureichender Produktqualität z. B. Imageschaden, Umsatzrückgang und Verlust von Marktanteilen.

Abbildung 8.1 gibt einen Überblick über die Zusammensetzung der Qualitätskosten.

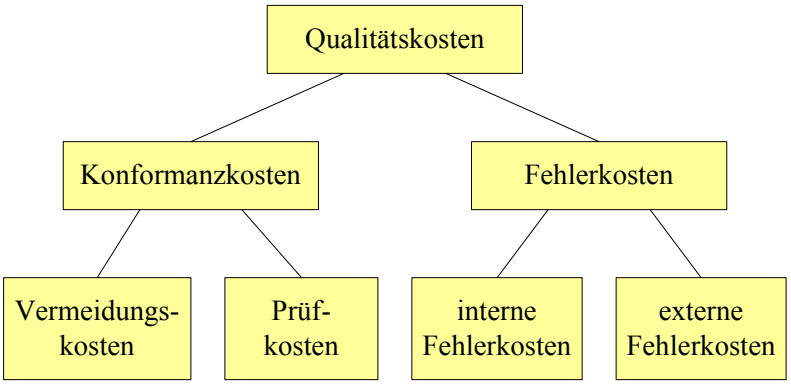


Abbildung 8.1: Zusammensetzung der Qualitätskosten

8.2 Qualitative Bewertung von Fehlervermeidungsmaßnahmen

Mit Hilfe der in Kapitel 7 vorgestellten Fehlervermeidungsmaßnahmen können viele der vorkommenden Feldfehler vermieden werden. Laut Boehm [BB01] entstehen 40 – 50 % des Projektaufwands in Softwareprojekten allein durch vermeidbare Nacharbeiten. Der Einsatz von Fehlervermeidung erfordert allerdings auch eine entsprechende Investition. In diesem Abschnitt wird aus Sicht des gesamten Produktlebenszyklus untersucht, welche qualitativen Merkmale bei der Abwägung von Fehlervermeidungsmaßnahmen eine wichtige Rolle spielen.

8.2.1 Vorteile von Fehlervermeidungsmaßnahmen

Qualität per Konstruktion Im Unterschied zum Einsatz von Tests und Reviews werden bei der Fehlervermeidung Vorkehrungen getroffen, um Fehler gar nicht erst entstehen zu lassen. Bei konsequentem Einsatz von Fehlervermeidungsmaßnahmen erhält man ein Produkt, das im Idealfall nicht mehr nachgebessert werden muss und eine höhere Qualität aufweist. Qualität entsteht also systematisch per Konstruktion und nicht durch Fehlersuche und häufige Nachbesserungen.

geringer Integrationsaufwand Werden mehrere Subsysteme eines zu realisierenden Systems arbeitsteilig realisiert, stellt sich oft erst in der Integrationsphase

heraus, ob die Schnittstellen zueinander passen und das integrierte System richtig funktioniert. Bei der Fehlervermeidung wird die Integrierbarkeit schon in frühen Entwicklungsphasen berücksichtigt und die spätere Integrierbarkeit sichergestellt. Dadurch kommt es nicht zu unerwartet großen Problemen und damit verbundenen Lieferverzögerungen und Qualitätsproblemen.

hohe Wiederverwendbarkeit der Subsysteme Aufgrund der detaillierten Schnittstellenbeschreibung ist das Verhalten jedes Subsystems genau bekannt. Dies erleichtert die Wiederverwendung der Subsysteme.

geringere Fehlerkosten Bei der Entwicklung modularer Systeme werden viele Fehler erst in der Integrationsphase erkannt. Durch den relativ späten Zeitpunkt innerhalb des Entwicklungsprozesses mit relativ aufwändiger Fehlersuche und -behebung sind notwendige Korrekturen vergleichsweise teuer. Die Kosten pro Fehler sind noch deutlich höher, nachdem das Produkt an den Kunden ausgeliefert wurde – nach Boehm [BB01] oft 100-mal teurer als in der Anforderungs- und Designphase. Im Automobilbereich verstärkt sich das Problem zusätzlich, weil die entwickelten Produkte in der Regel in großer Stückzahl hergestellt werden und bei einem Fehler jedes Stück nachgebessert oder ausgetauscht werden muss. Durch die Vermeidung von Fehlern können also enorme Fehlerkosten eingespart werden.

frühe Klärung der Anforderungen Da gerade zur Vermeidung von Integrationsproblemen schon in frühen Entwicklungsphasen die Schnittstellen und das Verhalten der Subsysteme genau festzulegen sind, führt dies zur schnelleren Klärung und Festlegung der Systemanforderungen. Das vermeidet häufiges Ändern und Ergänzen von Anforderungen in späteren Entwicklungsphasen.

effektivere Modultests Ist das Schnittstellenverhalten der Subsysteme nicht detailliert genug beschrieben, können viele Fehler mittels Modultests der einzelnen Subsysteme gar nicht gefunden werden. Solche Fehler werden dann frühestens bei der Integration gefunden. Erst durch eine detaillierte Schnittstellenbeschreibung in einer konsistenten Architektur können viele Fehler schon mit Modultests gefunden werden.

8.2.2 Nachteile von Fehlervermeidungsmaßnahmen

hoher Spezifikationsaufwand in frühen Entwicklungsphasen Um die spätere Integrierbarkeit sicherzustellen, ist genaue Kenntnis über das Schnittstellenverhalten aller Subsysteme erforderlich. Dazu ist ein höherer Spezifikationsaufwand für die Schnittstellen und die Verhaltensbeschreibung einzuplanen.

aufwändiges Änderungsmanagement Bei der Entwicklung eingebetteter Systeme entstehen neue Anforderungen oft noch während der Entwicklungszeit. Das kann Änderungen in Schnittstellen notwendig machen. Um die Integrierbarkeit der Subsysteme nicht zu gefährden, muss jede Schnittstellenänderung an einem Subsystem mit allen betroffenen Nachbarsubsystemen abgeglichen und ggf. dort nachgezogen werden. Diese feingranulare Kontrolle aller Schnittstellenänderungen macht das Änderungsmanagement zu einem aufwändigen Querschnittsprozess in der Produktentwicklung.

frontlastiger Entwicklungsprozess Der intensive Einsatz von Fehlervermeidungsmaßnahmen verlangt eine höhere Investition in frühe Entwicklungsphasen, es entsteht ein frontlastiger (front-loaded) Entwicklungsprozess. Bei unveränderter Personalplanung verschieben sich die nachfolgenden Entwicklungsphasen zeitlich nach hinten. Erste Produktmuster stehen erst später zur Verfügung, und es steigt bei unvollständiger Projektplanung die Gefahr, dass Deadlines nicht eingehalten werden [Kno93].

8.3 Quantitative Auswirkung von Fehlervermeidungsinvestitionen

Im vorherigen Abschnitt wurden die grundsätzlichen qualitativen Auswirkungen von Fehlervermeidungsmaßnahmen erläutert. Ohne Einsatz von Fehlervermeidung ist die Entwicklung zwar anfänglich deutlich billiger, geht aber das Risiko ein, dass bei zu vielen verbleibenden Fehlern die Integration sehr teuer wird und hohe Folgekosten durch Feldfehler entstehen. Andererseits kann ein überdimensionierter Einsatz von Fehlervermeidungsmaßnahmen zu Overdesign, also unnötig hoher Qualität, führen [JG99]. In der Praxis wird man in der Regel eine geeignete Kombination davon anwenden. Die Frage ist nun, welches Verhältnis aus Kostensicht optimal ist.

Im Folgenden werden die quantitativen Auswirkungen von Fehlervermeidungsinvestitionen auf die Qualitätskosten untersucht. Aufgrund des Fehlens entsprechender empirischer Daten erfolgt die Untersuchung mittels theoretischer Überlegungen.

Zunächst wird der quantitative Zusammenhang zwischen Fehlervermeidungsinvestitionen und den in Abschnitt 8.1 genannten Kostenarten *Vermeidungskosten*, *Prüfkosten*, *interne Fehlerkosten* sowie *externe Fehlerkosten* hergestellt. Anschließend erfolgt eine Gegenüberstellung der Fehlervermeidungsinvestitionen mit den Gesamtkosten.

8.3.1 Vermeidungskosten

Bei den zu optimierenden Fehlervermeidungsinvestitionen handelt es sich genau um Vermeidungskosten. Daher ist in diesem Fall der Zusammenhang trivial, siehe

Abbildung 8.2.

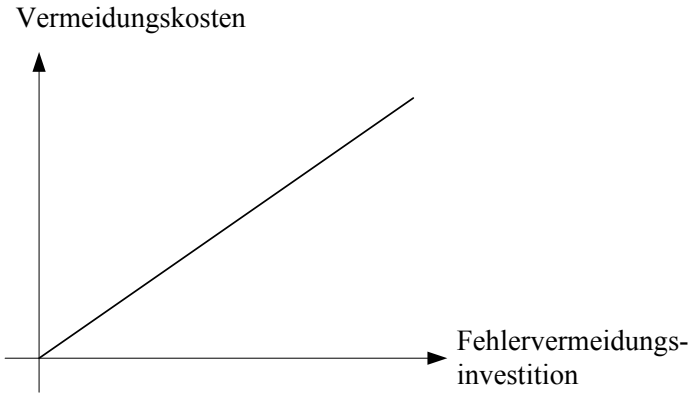


Abbildung 8.2: Auswirkung der Fehlervermeidungsinvestition auf Vermeidungskosten

Enthalten sind hier alle Konformanzkosten, die zur Vermeidung von Fehlern getätigt werden. Im Hinblick auf die Vermeidung von Integrationsproblemen gehören hierzu insbesondere der erhöhte Spezifikationsaufwand und ein umfassendes Änderungsmanagement zur Konsistenzsicherung bei Schnittstellenänderungen.

8.3.2 Prüfkosten

Prüfkosten sind Kosten, die beim Nachweis der Konformanz entstehen. Hierzu gehören analytische Qualitätssicherungsmaßnahmen wie Tests und Reviews.

Die Prüfkosten hängen grundsätzlich vom Umfang der nachzuweisenden Konformanz ab. Dieser ergibt sich u. a. aus der Komplexität des Produkts und dem angestrebten Sicherheitsniveau. Darüber hinaus besteht aber auch eine Abhängigkeit von der Fehlervermeidungsinvestition. Bei der Fehlervermeidung versucht man bereits durch ein geeignetes Design, bestimmte Produkteigenschaften sicherzustellen. Kann der Entwicklungsprozess sicherstellen, dass sich bestimmte Eigenschaften, die im Design nachgewiesen sind, auch später im Produkt wiederfinden, lassen sich dadurch einige Tests einsparen. Mit zunehmender Investition in die Fehlervermeidung nehmen also die Prüfkosten tendenziell ab. Aber trotz intensivem Einsatz von Fehlervermeidungsmaßnahmen lassen sich die Prüfkosten nie vollständig eliminieren, weil beispielsweise einige Nachweisverfahren gesetzlich vorgeschrieben sind oder der Kunde einen Abnahmetest verlangt. Die Kurve in Abbildung 8.3 konvergiert also von oben gegen einen Mindestaufwand an Prüfkosten.

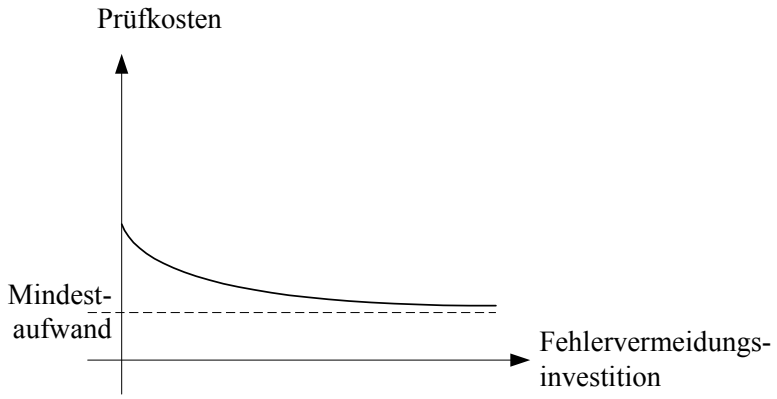


Abbildung 8.3: Auswirkung der Fehlervermeidungsinvestition auf Prüfkosten

8.3.3 Interne Fehlerkosten

Werden Abweichungen von der Spezifikation – also Fehler – noch vor der Auslieferung an den Kunden entdeckt und behoben, so rechnet man die dadurch entstandenen Kosten zu den internen Fehlerkosten. Hierzu zählt jeglicher Aufwand, der durch Fehler *zusätzlich* entstanden ist.

Die internen Fehlerkosten verlaufen gegenläufig zu den Fehlervermeidungsinvestitionen. Bereits einfache Vermeidungsmaßnahmen können schon eine deutliche Abnahme der Anzahl der Fehler bewirken und so die Fehlerkosten deutlich senken. Bei zunehmendem Aufwand für Fehlervermeidung wird die Steigerung des Effekts der Fehlervermeidung immer kleiner, weil durch die Fehlervermeidungsmaßnahmen selbst wiederum Fehler entstehen können. Theoretisch gesehen können mittels Fehlervermeidungsmaßnahmen alle Fehler vermieden werden, der Aufwand dazu ist aber extrem hoch. Abbildung 8.4 zeigt den geschätzten Verlauf der internen Fehlerkosten in Abhängigkeit von den Fehlervermeidungsinvestitionen.

In konkreten Projekten ist es durchaus möglich, dass sich die Fehlervermeidungsinvestitionen bereits über die dadurch eingesparten internen Fehlerkosten lohnen. Gerade bei der Vermeidung kostenintensiver Fehler wie beispielsweise Integrationsprobleme, die erst spät im Entwicklungsprozess sichtbar werden und größere Nacharbeiten erfordern können, kann das zutreffen. Im Allgemeinen kann diese Aussage aber erst nach der Berücksichtigung aller entstehenden Kosten getroffen werden.

8.3.4 Externe Fehlerkosten

Externe Fehlerkosten sind alle Fehlerkosten, die nach Auslieferung des Produkts an den Kunden entstehen. Diese setzen sich zusammen aus

- der *Behebung von Fehlern*, die beim Kunden auftreten, Rückrufaktionen bei

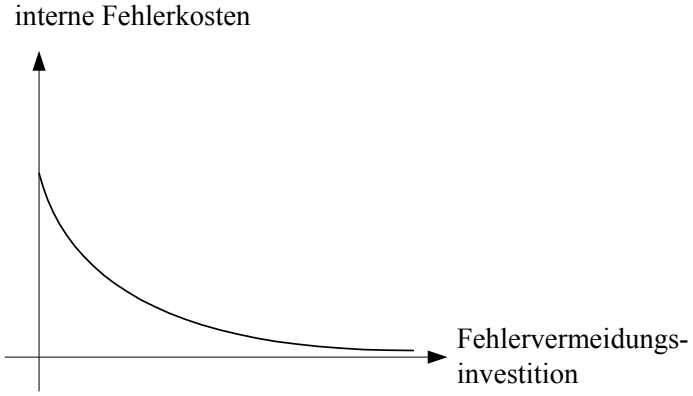


Abbildung 8.4: Auswirkung der Fehlervermeidungsinvestition auf interne Fehlerkosten

fehlerhafter Serienproduktion und sonstigen Gewährleistungsansprüchen des Kunden. Fehler, die erst nach der Auslieferung an den Kunden entdeckt und behoben werden, sind wesentlich teurer als vor der Auslieferung. Aufgrund der Serienproduktion im Automobilbereich ist immer gleich eine größere Stückzahl betroffen und zur Fehlerbehebung u. U. eine Rückrufaktion notwendig. Mit zunehmender Fehlervermeidung ist die Anzahl der verbleibenden Fehler im Produkt rückläufig, damit werden die Fehlerbehebungskosten geringer.

- Schadensersatz und Strafzahlungen aufgrund der gesetzlichen *Produkthaftung*. Die Produkthaftung hängt entscheidend davon ab, ob der Stand der Technik eingehalten wurde oder nicht. Darüber hinaus kann es weitere Qualitätsforderungen seitens des Kunden geben. Diese Qualitätsanforderungen werden unter dem Begriff *Zielqualität* zusammengefasst. Wird die Zielqualität unterschritten, ist man einem unmittelbaren, sehr hohen Haftungsrisiko ausgesetzt. Mit dem Erreichen der Zielqualität sinkt das Haftungsrisiko stark ab und ist dann nur noch sehr gering. Da davon auszugehen ist, dass mit zunehmender Fehlervermeidungsinvestition die tatsächliche Produktqualität im Allgemeinen steigt, verläuft die entsprechende Kurve in Abbildung 8.5 S-förmig.

- *langfristigen Auswirkungen* aufgrund unzureichender Produktqualität wie etwa Imageschaden, Umsatzrückgang und Verlust von Marktanteilen. Dieser Kostenpunkt ist strategisch gesehen sehr wichtig. Wie in Abschnitt 8.1 dargestellt, spielt die Marktpsychologie eine größere Rolle bei der Preisbildung als die unmittelbare Produktqualität. Dennoch kann eine schlechte Qualität langfristig zu einem Imageschaden und nachhaltigen Umsatzeinbußen führen. Bei höherer Investition in Fehlervermeidung steigt tendenziell die Produktqualität und senkt das langfristige Kostenrisiko.

Abbildung 8.5 zeigt den tendenziellen Verlauf obiger Kostenbestandteile und deren Summe.

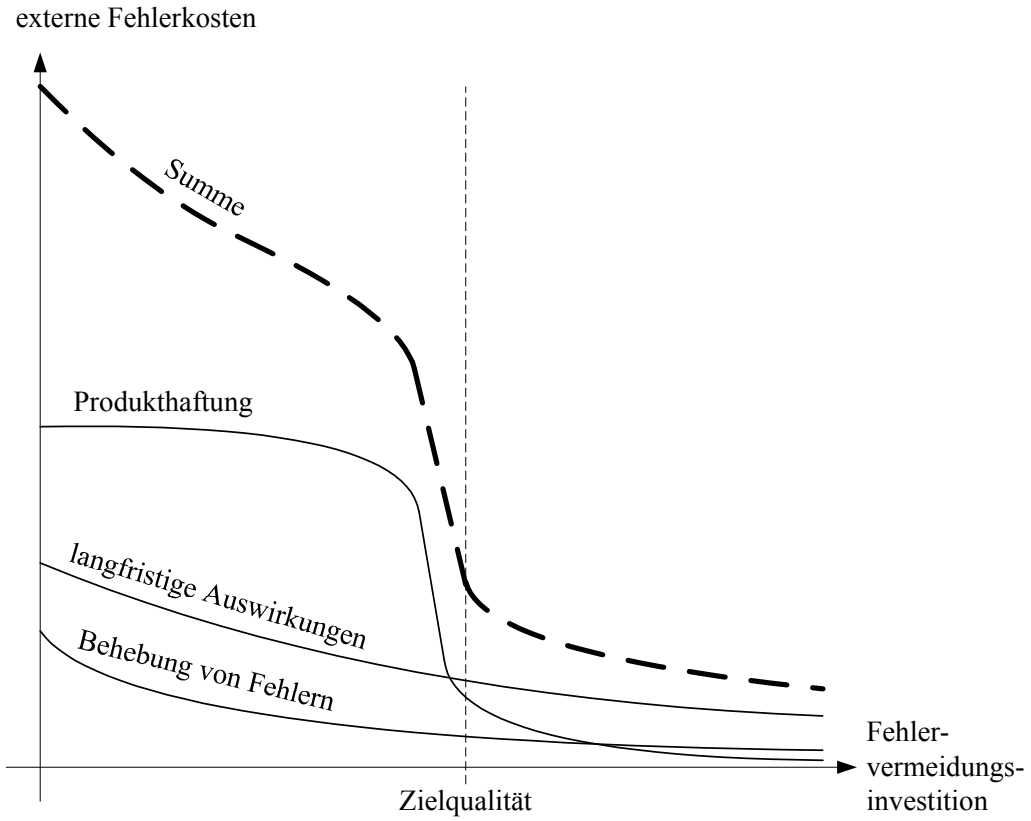


Abbildung 8.5: Auswirkung der Fehlervermeidungsinvestition auf externe Fehlerkosten

8.3.5 Optimierung der Gesamtkosten

Legt man die Kosten übereinander, ergibt sich in Abhängigkeit von den Fehlervermeidungsinvestitionen das Bild in Abbildung 8.6. Wie man leicht sieht, liegt der optimale Aufwand für Fehlervermeidungsmaßnahmen im Bereich der Zielqualität. Das Ergebnis ist nicht verwunderlich, allerdings wird hier dargestellt, welche Kosten zu berücksichtigen sind, um den kostenoptimalen Einsatz von Fehlervermeidungsmaßnahmen zu ermitteln. Hierin spiegeln sich die qualitativen Abwägungen in Abschnitt 8.2 wider.

Die Ermittlung der optimalen Fehlervermeidungsinvestitionen ist also ein Optimierungsproblem. Bei zu geringer Fehlervermeidungsinvestition schlagen besonders die externen Fehlerkosten, also v. a. Nachbesserung aufgrund einer zu geringen Qualität sowie die Produkthaftung, stark zu Buche, und zwar deutlich stärker als die zunächst

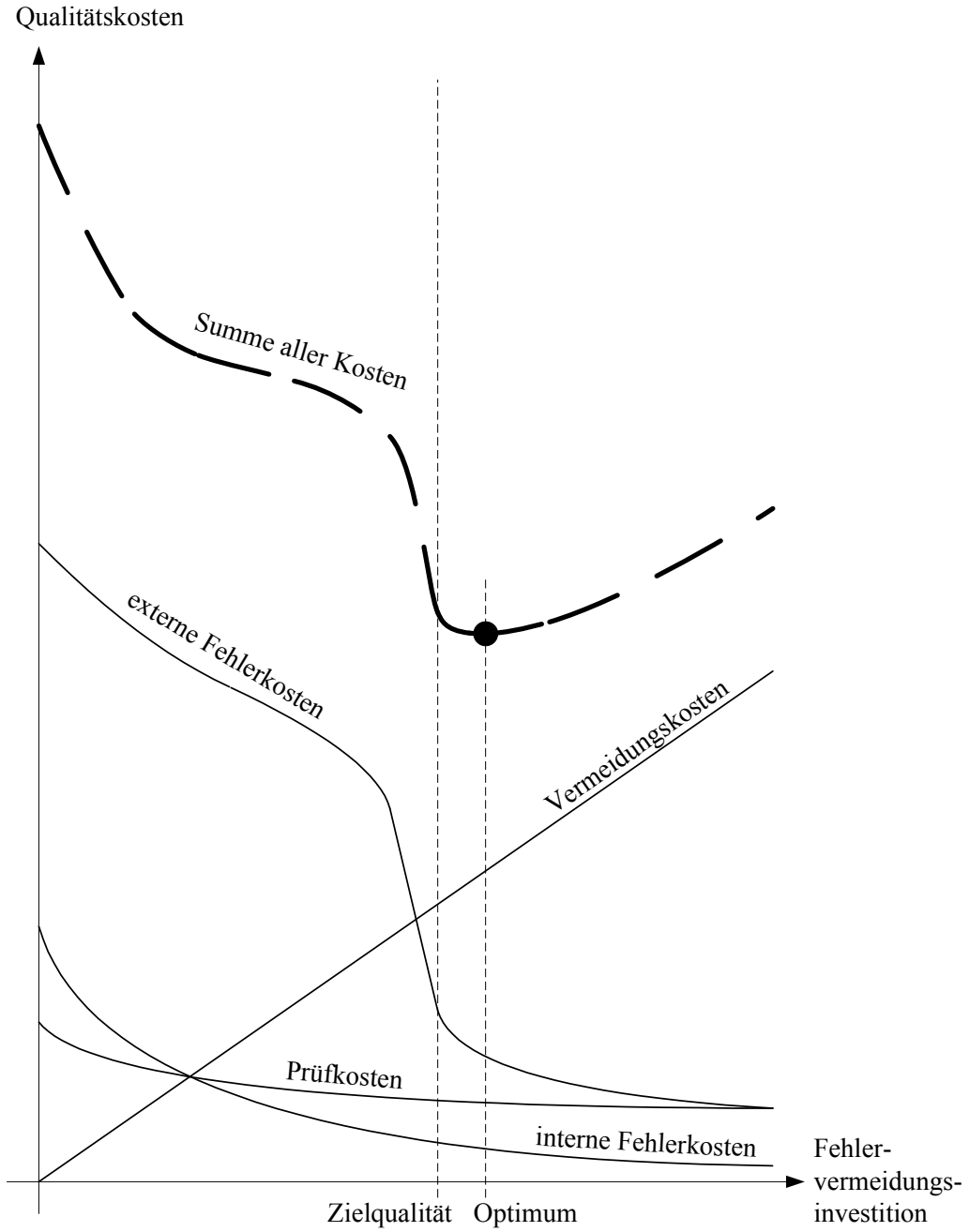


Abbildung 8.6: Auswirkung der Fehlervermeidungsinvestition auf die Qualitätskosten

eingesparten Kosten für die Fehlervermeidung. Auch eine zu hohe Investition in die Fehlervermeidung ist wirtschaftlich gesehen nicht optimal. Die dadurch erzielbaren Einsparungen sind kleiner als die zusätzliche Investition.

8.3.6 Berücksichtigung ethischer Fragestellungen

Neben den bekannten Annehmlichkeiten birgt das Automobil auch unmittelbare Gefahren für die Gesundheit von Mensch und Tier. Eine Fehlauslösung eines Airbags oder ein Versagen des Bremssystems können beispielsweise tödliche Folgen haben. Wie geht man mit diesen Risiken um? Welches Risiko ist man bereit zu tragen?

Geht man davon aus, dass ein bestimmtes ideales System (gemäß Abschnitt 2.4) praktisch kein Risiko für Lebewesen mit sich bringt, so ist man aus Sicht der Qualitätssicherung natürlich daran interessiert, ein möglichst fehlerfreies tatsächliches System zu erhalten. Ist ein tatsächliches System fehlerfrei, verhält es sich genau wie das ideale System und ist dementsprechend risikolos. Enthält das tatsächliche System hingegen Fehler, ist die Risikoabschätzung des idealen Systems nicht mehr darauf übertragbar. Das Risiko kann dann deutlich höher sein, wie die oben genannten Beispiele des Airbags und des Bremssystems zeigen.

Die in den externen Fehlerkosten (siehe Abschnitt 8.3.4) enthaltenen Kosten für die Produkthaftung berücksichtigen diesen Aspekt zumindest teilweise und sorgen für eine höhere Fehlervermeidungsinvestition. Doch der ethische Aspekt ist damit keinesfalls ausreichend berücksichtigt.

Ein weiterer Punkt ist, dass sich fehlerfreie tatsächliche Systeme in der Regel nicht realisieren oder nicht über einen längeren Zeitraum aufrechterhalten lassen. Grund dafür sind begrenzte finanzielle Mittel für die Fehlervermeidung sowie physikalisch bedingte Alterungseffekte. Trotz aller Anstrengungen bleibt immer ein gewisses Restrisiko bestehen.

Eine Lösung wäre, auf Airbags zu verzichten, um dem Risiko einer Fehlauslösung zu entgehen, oder auf das Automobil zu verzichten, um das Risiko eines Bremsversagens auszuschließen. Doch damit würde man jegliche Neuerungen blockieren bzw. Risiken nur verlagern, wie am Beispiel des Airbags leicht zu erkennen ist. Eine andere Lösung ist, mit jeder Entscheidung für eine bestimmte Funktionalität im Automobil sich auch deren Risiken bewusst zu sein und deren Konsequenzen zu akzeptieren – selbstverständlich unter Einsatz aller zumutbarer Risikominimierung. Aber auch dem Fahrer eines Automobils muss bewusst sein, dass dessen Benutzung ein Risiko für ihn selbst sowie andere darstellt. Ihm wird eine für ihn überschaubare Eigenverantwortung zugemutet.

8.4 Bedeutung für die Entwicklung von Subsystemen

Im vorherigen Abschnitt wurde aufgezeigt, welche Kosten bei der Bestimmung der optimalen Fehlervermeidungsinvestition zu berücksichtigen sind und wo das Optimum liegt. Diese Aussagen gelten zunächst nur für das zu entwickelnde System als Ganzes. Doch wie kann daraus abgeleitet werden, wie die einzelnen Subsysteme zu entwickeln sind?

Betrachtet man die Qualitätsmerkmale *Zuverlässigkeit* und *funktionale Sicherheit*, so bezieht sich die Systemspezifikation bezüglich dieser Eigenschaften zunächst nur auf das Black-Box-Verhalten des Systems. Dabei kann das System mehrere Funktionalitäten mit unterschiedlich hohen Qualitätsanforderungen zu erbringen haben.

Trennt die Architektur diese Funktionalitäten strukturell, wie in Abbildung 8.7 dargestellt, so lassen sich Seiteneffekte zwischen diesen Teilen ausschließen und die unterschiedlichen Qualitätsanforderungen der einzelnen Funktionalitäten können unmittelbar auf die jeweiligen Teile der Architektur abgebildet werden.

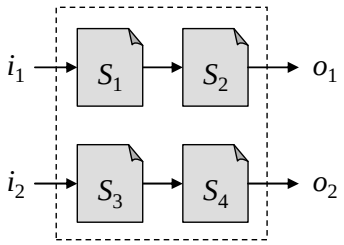


Abbildung 8.7: Architektur mit zwei unabhängigen Bestandteilen

Sind jedoch alle Subsysteme in einer Architektur miteinander verbunden, können sich diese gegenseitig beeinflussen und evtl. enthaltene Fehler im gesamten System propagieren. Ohne weitere Zusicherungen bestimmt das qualitativ schwächste Subsystem die Qualität des ganzen Systems.

Für die Entwicklung der einzelnen Subsysteme folgt daraus, dass jedes Subsystem mit mindestens der Qualitätsvorgabe zu entwickeln ist, die für die qualitativ anspruchsvollste Funktionalität gefordert wird, an der es beteiligt ist. Miteinander verbundene Subsysteme unterliegen damit stets denselben Qualitätsansprüchen. Ausnahmen davon sind nur zulässig, wenn eine Beeinflussung trotz struktureller Verbindung im Einzelfall garantiert ausgeschlossen werden kann. Beispielsweise führt ein Autoradio, das sich 10 Minuten nach dem Einschalten automatisch wieder ausschaltet, nicht notwendigerweise zu einem fehlerhaften Verhalten des Fahrzeugantriebs, der Nachweis ist in der Praxis aber oft nur schwer zu erbringen. Fehlermöglichkeits- und Einflussanalysen (FMEA) für Software sind noch nicht ausreichend verstanden [Lyu07] und Fehlerauswirkungsanalysen mittels Model Checking sind recht implementierungsnah und für komplexe Software derzeit nicht praktikabel.

Im Automobil sind alle Steuergeräte miteinander verbunden. Zum einen sind mittlerweile alle Subnetze mittels Gateways verbunden und können so potenziell beliebige Daten austauschen. Zum anderen hat die gesamte Elektrik/Elektronik nur eine gemeinsame Energieversorgung, die wiederum zu Seiteneffekten führen kann und so eine strukturelle Verbindung aller Steuergeräte darstellt. Ein übermäßiger Stromverbrauch im Autoradio kann zum Abschalten anderer Subsysteme führen und sich damit potentiell auch auf den Fahrzeugantrieb auswirken.

9 Schluss

9.1 Zusammenfassung

Während man sich in den ersten Jahrzehnten der Automobilentwicklung auf die Verbesserung von Mechanik und Material konzentrierte, ist mittlerweile der intensive Einsatz von Elektrik und Elektronik in den Mittelpunkt gerückt. Die Vernetzung der elektronischen Komponenten hat zu neuartigen Funktionen geführt, wobei das korrekte Zusammenspiel der Komponenten eine zentrale Herausforderung darstellt.

Das Ziel dieser Arbeit ist die Bewertung von Maßnahmen zur Erkennung und Vermeidung von softwarebedingten Integrationsproblemen, deren Ursachen in frühen Entwicklungsphasen liegen. Das Integrationsproblem wird dabei sowohl auf formaler Ebene behandelt als auch im Praxisumfeld untersucht. Anhand eines formalen Systemmodells auf der Basis von FOCUS werden zunächst die Integrationsproblematik sowie wichtige Begriffe definiert, um ein einheitliches Begriffsverständnis zu erlangen. Je nach Entwicklungsphase, in der ein Integrationsproblem entsteht, wird zwischen Architekturfehler und Integrationsfehler unterschieden. Der Begriff des Verbundfehlers macht dabei deutlich, dass es Integrationsprobleme gibt, deren Fehlerursachen sich nicht eindeutig einem einzelnen Subsystem zuordnen lassen, sondern erst durch die Kombination mehrerer Subsysteme entstehen.

Die Untersuchung des Stands der Praxis zeigt auf, dass heutige Architekturbeschreibungen nicht hinreichend vollständig sind. Die Angaben zum Schnittstellenverhalten der Subsysteme konzentrieren sich vorwiegend auf syntaktische Merkmale der Schnittstelle, die das Datenformat beschreiben, gehen jedoch nicht intensiv genug auf das – bei Echtzeitsystemen genauso wichtige – Zeitverhalten ein. Dieses Schnittstellenverständnis führt dazu, dass die in den Software-Entwicklungsprozessen verankerten Ziele zur Sicherstellung der Schnittstellenkonsistenz nicht immer die nötige Wirkung erreichen. Die Auswertung empirischer Fehlerdaten verdeutlicht die vorhandenen Schwächen aktueller Entwicklungsprozesse hinsichtlich der Systemintegration. Sie zeigt auch die Vielfalt möglicher Ursachen für Integrationsprobleme, sowohl in unterschiedlichen Abstraktionsebenen als auch in Daten- und Zeiteigenschaften.

Die Herangehensweise zur Behebung der beschriebenen Probleme erfolgt wiederum auf zwei Ebenen. Zunächst wird theoretisch untersucht, welche Anforderungen ein Entwicklungsprozess erfüllen muss, um Integrationsprobleme zu vermeiden. Anschließend werden verschiedenartige Ansätze hinsichtlich ihrer Praxistauglichkeit zur Erkennung und Vermeidung von Integrationsproblemen in frühen Entwicklungsphasen

bewertet. Analytische Maßnahmen wie Reviews, Ansätze zur Architekturverifikation und Tests messen das momentane Qualitätsniveau eines Produkts und eignen sich so zur Fehlererkennung. Konstruktive Maßnahmen wie die zeitliche Entkopplung von Subsystemen, der Einsatz von Spezifikationssprachen und die Wiederherstellung der Schnittstellenkonsistenz nach einer Änderung dienen der Vermeidung von Fehlern. Während beispielsweise ein Review vergleichsweise niedrige Investitionskosten erfordert und einen Großteil der vorhandenen Fehler aufdecken kann, ist bei hohen Qualitätsanforderungen eine umfassende Schnittstellenbeschreibung unter Umständen sogar Kosten sparender. Abschließende Überlegungen zum kostenoptimalen Einsatz von Qualitätssicherungsmaßnahmen verdeutlichen dies.

9.2 Ausblick

In dieser Arbeit wurden nur Maßnahmen zur Fehlererkennung und -vermeidung untersucht, die bereits zum Stand der Technik zählen. Von der Entwicklung neuartiger Ansätze wurde abgesehen, um den Umfang der Arbeit im Rahmen zu halten.

Ein viel versprechender und interessanter Ansatz wäre der Einsatz einer „Meta-Programmiersprache“, welche aus der Kombination einer Architekturbeschreibungssprache mit einer Programmiersprache wie C oder Ada entsteht und so die Architekturbeschreibung auf allen Abstraktionsebenen abbildet. Neben der bisherigen Programmierung der einzelnen Softwaremodule, die dann übersetzt und durch einen Linker gebunden werden, soll zusätzlich die Architektur auf einer abstrakteren Ebene beschrieben werden können. Die Schnittstellendetails aus der Architekturbeschreibung können dann für die Konsistenzprüfung sowie zur Generierung der Schnittstellenimplementierung herangezogen werden. Eine generierte Schnittstellenimplementierung kann z. B. eine automatische Datenkonvertierung zwischen unterschiedlichen Maschinenmodellen oder physikalischen Einheiten durchführen. Die Unterstützung der Top-Down-Entwicklung durch so eine Meta-Programmiersprache würde den Übergang zwischen Design und Implementierung erleichtern und für ein einheitliches Datenmodell in beiden Entwicklungsphasen sorgen.

Anhang A

Vorlage zur Schnittstellenspezifikation

Nachfolgend werden Eigenschaften aufgelistet, die eine Schnittstellenbeschreibung mindestens enthalten soll. Die Eigenschaften wurden aus den in Kapitel 5 identifizierten Spezifikationslücken abgeleitet und zu einer vollständigen Vorlage ergänzt.

Allgemeine Angaben:

- Name des Signals, z. B. Verzögerungswunsch
- Name des zugehörigen (Sub-)Systems, z. B. Bremssteuergerät
- Signalrichtung (Eingabe, Ausgabe), z. B. Eingabe

Angaben zur Semantikdarstellung:

- informelle Bedeutung, z. B. „aktueller Verzögerungswunsch des Fahrers bei bewegtem Fahrzeug“
- physikalische Einheit, z. B. $\frac{\mathrm{m}}{\mathrm{s}^2}$
- Grundmenge des Wertebereichs, z. B. $\mathbb{R}$
- Grenzen des Wertebereichs, z. B. $[0;\ 10]$

Angaben zur Maschinendarstellung:

- Signaltyp (analog, digital), z. B. digital
- Anzahl Bits, z. B. 8
- Semantik der Bitbelegungen
  - Datentyp der Maschine, z. B. signed integer
  - Bytereihenfolge, z. B. big endian
  - Bitreihenfolge innerhalb eines Byte, z. B. big endian

  - Bedeutung des Vorzeichens, z. B. „gegen die Fahrtrichtung“
  - Umrechnung der Semantikdarstellung in die Maschinendarstellung, z. B. $x \mapsto \left\lfloor \frac{255}{10} x \right\rfloor$
  - Genauigkeit (absolut bzw. relativ), z. B. $< 1\,\%$
  - Bedeutung der Fehlerwerte, z. B. -1 = nicht verfügbar
- Zeitverhalten
  - minimales und maximales Informationsalter, z. B. [0 ms; 10 ms]
  - Aktualisierungszeitpunkte
    * Synchronisierungsereignis, z. B. „sendendes Steuergerät“
    * Sendefrequenz, z. B. 10 ms
    * maximaler Zeit-Jitter, z. B. 100 ns

Angaben zur physikalischen Darstellung:

- Zeitverhalten (kontinuierlich, diskret), z. B. diskret
- Signal-zu-Rausch-Abstand (in dB), z. B. gemäß CAN-Spezifikation
- Entprellzeit bei Signalflanke, z. B. gemäß CAN-Spezifikation

Anhang B

Vollständiges Listing zum AADL-Beispiel

Nachfolgend ist das vollständige Listing in AADL zum Brake-by-Wire-Beispiel in Abschnitt 7.4.1 angegeben.

Listing B.1: Beschreibung der Systemarchitektur

```
1  data PedalkraftData
2    properties
3      semantic::min_value => 0.0;
4      semantic::max_value => 500.0;
5      semantic::physical_unit => "N";
6      machine::signal_type => analogue;
7  end PedalkraftData;
8
9  data VerzoegerungswunschData
10   properties
11     semantic::min_value => 0.0;
12     semantic::max_value => 10.0;
13     semantic::physical_unit => "m/s^2";
14     machine::leading_sign_meaning => "gegen die Fahrtrichtung";
15 end VerzoegerungswunschData;
16
17 data AktorspannungData
18   properties
19     semantic::min_value => 0.0;
20     semantic::max_value => 40.0;
21     semantic::physical_unit => "V";
22     machine::signal_type => analogue;
23     physical::time_behaviour => continuous;
24 end AktorspannungData;
25
26 data BremsmomentData
27   properties
28     semantic::min_value => 0.0;
29     semantic::max_value => 3000.0;
30     semantic::physical_unit => "Nm";
31     physical::signal_type => analogue;
```

```
32 end BremsmomentData;
33
34 system Bremssystem
35   features
36     Pedalkraft: in data port PedalkraftData;
37     Bremsmoment: out data port BremsmomentData;
38   flows
39     f: flow path Pedalkraft -> Bremsmoment
40        { Expected_Latency => 50 Ms; };
41 end Bremssystem;
42
43 system implementation Bremssystem.impl
44   subcomponents
45     Pedal.impl: system Pedal;
46     Bremssteuergeraet.impl: system Bremssteuergeraet;
47     Bremsaktor.impl: system Bremsaktor;
48   connections
49     c1: data port Pedalkraft -> Pedal.impl.Pedalkraft
50         { Latency => 0 Ms; };
51     c2: data port Pedal.impl.Verzoegerungswunsch
52         -> Bremssteuergeraet.impl.Verzoegerungswunsch
53         { Latency => 0 Ms; };
54     c3: data port Bremssteuergeraet.impl.Aktorspannung
55         -> Bremsaktor.impl.Aktorspannung
56         { Latency => 0 Ms; };
57     c4: data port Bremsaktor.impl.Bremsmoment -> Bremsmoment
58         { Latency => 0 Ms; };
59   flows
60     f: flow path Pedalkraft -> c1 -> Pedal.impl.flow1
61        -> c2 -> Bremssteuergeraet.impl.flow2
62        -> c3 -> Bremsaktor.impl.flow3
63        -> c4 -> Bremsmoment;
64 end Bremssystem.impl;
65
66 system Pedal
67   features
68     Pedalkraft: in data port PedalkraftData;
69     Verzoegerungswunsch: out data port VerzoegerungswunschData;
70   flows
71     flow1: flow path Pedalkraft -> Verzoegerungswunsch
72            { Latency => 10 Ms; };
73 end Pedal;
74
75 system Bremssteuergeraet
76   features
77     Verzoegerungswunsch: in data port VerzoegerungswunschData;
```

```
78      Aktorspannung: out data port AktorspannungData;
79    flows
80      flow2: flow path Verzoegerungswunsch -> Aktorspannung
81             { Latency => 10 Ms; };
82  end Bremssteuergeraet;
83
84  system Bremsaktor
85    features
86      Aktorspannung: in data port AktorspannungData;
87      Bremsmoment: out data port BremsmomentData;
88    flows
89      flow3: flow path Aktorspannung -> Bremsmoment
90             { Latency => 10 Ms; };
91  end Bremsaktor;
```

Listing B.2: Erweiterung von AADL um zusätzliche Eigenschaften

```
1   property set semantic is
2     basetype: enumeration (string, integer, real) => real
3               applies to (data);
4     min_value: aadlreal applies to (data);
5     max_value: aadlreal applies to (data);
6     physical_unit: aadlstring applies to (data);
7   end semantic;
8
9   property set machine is
10    signal_type: enumeration (digital, analogue) => digital
11                 applies to (data);
12    bit_size: aadlinteger => 8 applies to (data);
13    bit_interpretation: enumeration (string, uint, sint, float)
14                        => sint applies to (data);
15    bit_order: enumeration (little_endian, big_endian)
16               => big_endian applies to (data);
17    byte_order: enumeration (little_endian, big_endian)
18                => big_endian applies to (data);
19    accuracy_absolute: aadlreal applies to (data);
20    accuracy_relative: aadlreal applies to (data);
21    leading_sign_meaning: aadlstring applies to (data);
22    error_values: list of aadlreal applies to (data);
23    information_age: Time_Range applies to (port);
24    update_period: Time applies to (port);
25    update_jitter: Time applies to (port);
26  end machine;
27
28  property set physical is
29    representation_standard: enumeration (CAN, FlexRay,
30      internal_memory, other) applies to (data);
```

```
31    signal_to_noise_distance: aadlreal applies to (data);
32    debounce_time: Time applies to (data);
33    time_behaviour: enumeration (continuous, discrete) => discrete
34                    applies to (data);
35  end physical;
```

Literaturverzeichnis

[ALRL04] Avizienis, Algirdas, Jean-Claude Laprie, Brian Randell und Carl E. Landwehr: *Basic Concepts and Taxonomy of Dependable and Secure Computing.* IEEE Transactions on Dependable Secure Computing, 1(1):11–33, 2004.

[Aut07] Autosar Consortium. `http://www.autosar.org/`, zuletzt besucht am 15.10.2007.

[Bal97] Balzert, Helmut: *Lehrbuch der Software-Technik – Software-Management, Software-Qualitätssicherung und Unternehmensmodellierung*, Band 2. Spektrum Akademischer Verlag, 1997.

[BB01] Boehm, Barry und Victor R. Basili: *Software Defect Reduction Top 10 List.* IEEE Computer, 34(1):135–137, 2001.

[Ben05] Bender, Klaus: *Embedded Systems – qualitätsorientierte Entwicklung*, Band 1. Springer Verlag, 2005.

[Bos02] Robert Bosch GmbH und Horst Bauer (Herausgeber): *Adaptive Fahrgeschwindigkeitsregelung ACC.* BOSCH Gelbe Reihe. Christiani, Konstanz, 1. Auflage, 2002.

[Bos07] Robert Bosch GmbH (Herausgeber): *Kraftfahrtechnisches Taschenbuch.* Vieweg, 26. Auflage, 2007.

[Bre01] Breitling, Max: *Formale Fehlermodellierung für verteilte reaktive Systeme.* Dissertation, Technische Universität München, Fakultät für Informatik, 2001.

[Bro01] Broy, Manfred: *Refinement of time.* Theoretical Computer Science, 253(1):3–26, 2001.

[Bro04] Broy, Manfred: *Time, Abstraction, Causality and Modularity in Interactive Systems: Extended Abstract.* Electronic Notes in Theoretical Computer Science, 108:3–9, 2004.

[Bro06] Broy, Manfred: *Challenges in Automotive Software Engineering.* In: *ICSE '06: Proceeding of the 28th International Conference on Software Engineering*, Seiten 33–42, New York, NY, USA, 2006. ACM Press.

[BS01] Broy, Manfred und Ketil Stølen: *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement.* Springer-Verlag New York, Inc., 2001.

[Con68] Conway, Melvin E.: *How do Committees Invent?* Datamation, 14(4):28–31, April 1968.

[DIN05] DIN EN ISO 9000: *Qualitätsmanagementsysteme – Grundlagen und Begriffe (ISO 9000:2005)*, 2005.

[EBK03] Elmenreich, Wilfried, Günther Bauer und Hermann Kopetz: *The Time-Triggered Paradigm.* In: *Proceedings of the Workshop on Time-Triggered and Real-Time Communication*, Manno, Switzerland, Dec. 2003.

[Ech90] Echtle, Klaus: *Fehlertoleranzverfahren.* Springer-Verlag, 1990.

[Ehl03] Ehlers, Tiemo: *Verfahren zur Sicherstellung der Systemintegrität in Fahrzeugen mit vernetzten Steuergeräten.* Dissertation, Technische Universität Braunschweig, Institut für Elektrische Messtechnik und Grundlagen der Elektrotechnik, 2003.

[Ern03] Ernst, Rolf: *Putting it all Together.* Queue, 1(2):50–55, April 2003.

[Fag76] Fagan, Michael E.: *Design and code inspections to reduce errors in program development.* IBM Systems Journal, 15(3):182–211, 1976.

[FGH06] Feiler, Peter H., David P. Gluch und John J. Hudak: *The Architecture Analysis & Design Language (AADL): An Introduction.* Technischer Bericht, Carnegie Mellon University, Software Engineering Institute, CMU/SEI-2006-TN-011, Feb. 2006.

[FGP04] Fleischmann, Andreas, Eva Geisberger und Markus Pister: *Herausforderungen für das Requirements Engineering eingebetteter Systeme.* Technischer Bericht, Technische Universität München, Fakultät für Informatik, 2004.

[Fle05] *FlexRay Protocol and Physical Layer Specification V2.1 Rev. A.* FlexRay Consortium GbR, `http://www.flexray.com/`, 2005.

[FLS06] Frühauf, Karol, Jochen Ludewig und Helmut Sandmayr: *Software-Prüfung – Eine Anleitung zum Test und zur Inspektion.* Vdf Hochschulverlag, 6. Auflage, 2006.

[GGF93] Gilb, Tom, Dorothy Graham und Susannah Finzi: *Software Inspection*. Addison-Wesley, 1993.

[Gra07] Grams, Timm: *Tabelle der Begriffe: Englisch-Deutsch*. `http://www.fh-fulda.de/~grams/Reliability/R&S-Terms1.html`, zuletzt abgerufen am 26.09.2007.

[Han03] Handelsblatt: *Software-Absturz im Auto-Cockpit*, 30.09.2003. `http://www.handelsblatt.com/`.

[IEE90] IEEE Standards Board: *IEEE Standard Glossary of Software Engineering Terminology – IEEE Std. 610.12-1990*. IEEE, New York, NY, USA, 1990.

[ISO94] ISO/IEC 7498-1: *Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*, 1994.

[ISO95] ISO/IEC 12207: *Information technology – Software life cycle processes*, 1995.

[ISO03] ISO 11898: *Road vehicles – Controller area network (CAN)*, 2003.

[Jaz98] Jazdi, Nasser: *Komponentenbasierte Entwicklung Eingebetteter Systeme (KEES)*. Technischer Bericht, Universität Stuttgart, Institut für Automatisierungs- und Softwaretechnik, 1998.

[JG99] Juran, Joseph M. und A. Blanton Godfrey: *Juran's Quality Handbook*. McGraw-Hill, 5. Auflage, 1999.

[Kno93] Knox, Stephen T.: *Modeling the Cost of Software Quality*. Digital Technical Journal of Digital Equipment Corporation, 5(4):9–17, 1993.

[Kop97] Kopetz, Hermann: *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[Lai02] Laitenberger, Oliver: *A Survey of Software Inspection Technologies*. In: Chang, S. K. (Herausgeber): *Handbook on Software Engineering and Knowledge Engineering*, Seiten 517–555. World Scientific Publishing, Jan. 2002.

[Lap92] Laprie, Jean-Claude: *Dependability: Basic concepts and terminology in English, French, German, Italian, and Japanese (Dependable computing and fault-tolerant systems)*. Springer Verlag, 1992.

[LL06] Ludewig, Jochen und Horst Lichter: *Software Engineering.* Dpunkt Verlag, 1. Auflage, 2006.

[Lyu07] Lyu, Michael R.: *Software Reliability Engineering: A Roadmap.* In: *Future of Software Engineering, 2007. FOSE '07*, Seiten 153–170, Washington, DC, USA, 2007. IEEE Computer Society.

[Mar03] Mariani, Leonardo: *A Fault Taxonomy for Component-Based Software.* Electronic Notes in Theoretical Computer Science, 82(6):55–65, September 2003.

[Mer06] Mercer Management Consulting: *Elektronik setzt die Impulse im Auto: Mercer-Studie Autoelektronik*, 14.12.2006. `http://www.mercermc.de/`.

[MSR07] MSR-Arbeitsgruppe. `http://www.msr-wg.de/`, zuletzt besucht am 15.10.2007.

[MT00] Medvidovic, Nenad und Richard N. Taylor: *A Classification and Comparison Framework for Software Architecture Description Languages.* IEEE Transactions on Software Engineering, 26(1):70–93, Jan. 2000.

[NSSLW05] Navet, Nicolas, Yeqiong Song, Françoise Simonot-Lion und Cédric Wilwert: *Trends in automotive communication systems.* Proceedings of the IEEE, 93(6):1204–1223, June 2005.

[Osa07] *OSATE Webseite.* `http://la.sei.cmu.edu/aadl/currentsite/tool/osate-down.html`, zuletzt abgerufen am 09.10.2007.

[RAT+06] Runeson, Per, Carina Andersson, Thomas Thelin, Anneliese Andrews und Tomas Berling: *What Do We Know about Defect Detection Methods?* IEEE Software, Seiten 82–90, 2006.

[RB07] Rausch, Andreas und Manfred Broy: *Das V-Modell XT – Grundlagen, Erfahrungen und Werkzeuge.* Dpunkt Verlag, 1. Auflage, 2007.

[SAE04] Society of Automotive Engineers: *Architecture Analysis & Design Language (AADL).* Standard AS5506, Nov. 2004.

[SZRS06] Schneider, Jörn, Dirk Ziegenbein, Herbert Reiter und Vincent Schulte-Coerne: *Sichere und zuverlässige Software-Integration.* In: Plödereder, Erhard, Hubert B. Keller, Peter Dencker und Michael Tonndorf (Herausgeber): *Automotive – Safety & Security 2006 – Sicherheit und Zuverlässigkeit für automobile Informationstechnik*, Seiten 99–110. Shaker Verlag, Oktober 2006.

[Tan03] Tanenbaum, Andrew S.: *Computernetzwerke*. Pearson Studium, 4. Auflage, 2003.

[TS03] Tanenbaum, Andrew S. und Maarten van Steen: *Verteilte Systeme. Grundlagen und Paradigmen*. Pearson Studium, 2003.

[VDI94] *VDI 2896, Instandhaltungs-Controlling innerhalb der Anlagenwirtschaft*, 1994.

[Wag07] Wagner, Stefan: *Cost-Optimisation of Analytical Software Quality Assurance*. Dissertation, Technische Universität München, Fakultät für Informatik, 2007.

[ZH04] Zimmermann, Karsten W. und Hans-Michael Hauser: *The Growing Importance of Embedded Software: Managing Hybrid Hardware-Software Business*. The Boston Consulting Group, Inc., 2004.