Universität Ulm
Fakultät für Mathematik und Wirtschaftswissenschaften
Institut für Angewandte Informationsverarbeitung
Leiter: Prof. Dr. Franz Schweiggert

Zur Parallelisierung von Systemintegrationstests im E/E Umfeld

Dissertation
zur Erlangung des Doktorgrades

Doktor der Naturwissenschaften
(Dr. rer. nat.)

der Fakultät für Mathematik und Wirtschaftswissenschaften
der Universität Ulm

Matthias Kirchmayr
Schongau 2009

Amtierender Dekan:	Professor Dr. Frank Stehling
Gutachter:	1. Professor Dr. Franz Schweiggert
	2. Professor Dr. Helmuth Partsch
Tag der Promotion:	16.06.2009

Für Franz und Luise

Danksagung

Diese Arbeit entstand während meiner Zeit als Doktorand bei der Daimler AG in Sindelfingen im Bereich der PKW Elektronik Entwicklung. Die wissenschaftliche Betreuung wurde durch die Fakultät für Mathematik und Wirtschaftswissenschaften der Universität Ulm übernommen.

Mein Dank gilt in erster Linie Herrn Prof. Dr. Franz Schweiggert für die Betreuung meiner Arbeit. Neben den zahlreichen fachlichen Anregungen bin ich besonders für die motivierenden Gespräche in den schwierigen Phasen dieser Arbeit dankbar. Ebenso möchte ich Herrn Prof. Dr. Helmuth Partsch für die Übernahme des Zweitgutachtens danken.

Ferner möchte ich mich bei meinen Kollegen des HIL-Teams für die Unterstützung bedanken. Mein besonderer Dank gilt hierbei meinen Betreuer Dr. Hermann Schmid. Sein Rat und die vielen anregenden Diskussionen haben maßgeblich zum Gelingen meiner Arbeit beigetragen. Dr. Florian Kusche danke ich für seine Unterstützung bei den Problemen rund um die AL und RTAE. Für den nötigen Freiraum in der entscheidenden Phase meiner Arbeit danke ich den Herren Slanksy und Serries, die mich weitgehend vom Tagesgeschäft freigestellt haben.

Des Weiteren danke ich allen „Freiwilligen“, die sich dem Stil und der Rechtschreibung dieser Arbeit angenommen haben. Ihre Kommentare und Anregungen haben wesentlich zur Lesbarkeit dieser Arbeit beigetragen. Mein spezieller Dank gilt hierbei Helga Leimböck für Ihren rastlosen Einsatz.

Mein ganz besonderer Dank gilt meinen Eltern. Ihre Unterstützung gab mir stets einen starken Rückhalt, so dass ich mich auf das Wesentliche konzentrieren konnte. Ihr Glaube an mich war ausschlaggebend für das Erreichte.

Sindelfingen, im November 2008 Matthias Kirchmayr

Inhaltsverzeichnis

Kapitel 1

Einleitung

Moderne Kraftfahrzeuge zeichnen sich in zunehmendem Maße durch den Einsatz von Elektrik und Elektronik (E/E) aus. Hiermit können Funktionen implementiert werden, welche allein durch Mechanik nicht realisierbar wären. So hat beispielsweise der Umweltschutz stark durch die E/E profitiert, indem effizientere Motorsteuerungen die Abgaswerte verbessern [Rob07]. Derartige Innovationen werden auch vermehrt zu Vermarktungszwecken verwendet. Die E/E ist demnach ein wesentlicher Wettbewerbsfaktor für die Automobilhersteller geworden [Sch03]. Das Image speziell von Premium-Herstellern hängt somit auch in beträchtlichem Maße von den Innovationen auf dem Gebiet der E/E ab[1].

Auch wirtschaftlich stellt die Elektrik und Elektronik einen wichtigen Faktor dar. Nach [VDE07] wird der Anteil der E/E an der Wertschöpfung eines Automobils bis 2010 auf 40% ansteigen . Es ist daher nicht verwunderlich, dass die Neu- und Weiterentwicklung elektrisch/elektronischer Systeme im Fokus der Automobilhersteller steht. So werden 90% zukünftiger Innovationen im Fahrzeug in ihrer Funktionalität durch Elektronik geprägt sein [Rei06b].

Eine derart rasante Entwicklung hat zwangsläufig auch den Anstieg der Komplexität der elektrisch/elektronischen Systeme eines Fahrzeugs zur Folge. Um dennoch den eigenen Ansprüchen an die Qualität der Produkte gerecht zu werden, sind die Automobilhersteller zu umfangreichen Testaktivitäten gezwungen. Dies beinhaltet neben dem frühzeitigen Testbeginn auch den sy-

[1] so wirbt beispielsweise Audi mit dem Slogan „Vorsprung durch Technik“ und Mercedes-Benz versprach 2003 „Fortschritt beginnt mit einem Mercedes“

stematischen Einsatz reproduzierbarer Tests [HHS04]. Eine enge Integration in den Entwicklungsprozess soll dabei die optimale Unterstützung aller E/E-Entwicklungsphasen sicherstellen [Sch03].

1.1 Motivation

Auf abstrakter Ebene kann die Fahrzeugelektronik in verschiedene Systeme unterteilt werden, wie z.B. die Motorsteuerung oder die Außenbeleuchtung. Die konkrete Umsetzung der Funktionalität erfolgt durch weitgehend autonome Komponenten, sogenannte Steuergeräte (SG). Aus einer Kombination von Hard- und Software bestehend sind diese komplett in ein technisches Umfeld integriert. Daher werden die Steuergeräte auch als *eingebettete Systeme* bezeichnet. Die Funktion wird hierbei durch das Zusammenspiel von Hard- und Software erbracht. Eine Vernetzung der Steuergeräte ermöglicht die Kommunikation untereinander und somit die Verteilung einer Funktion auf mehrere Steuergeräte.

Beim Test eingebetteter Systeme liegt der Schwerpunkt auf der Prüfung der Software, durch welche der wesentliche Anteil der Gesamtfunktionalität erbracht wird [BB99]. Aufgrund der Komplexität der Software ist deren Entwicklung fehleranfällig. Diese Fehler führen bei eingebetteten Systemen meist zu hohen Kosten [PGW02]. Je später ein Fehler zudem gefunden wird, umso teurer ist dessen Beseitigung [BSS05]. Abbildung 1.1 (S. 3) zeigt die durchschnittlichen Fehlerbeseitigungskosten in Abhängigkeit vom Zeitpunkt der Fehlererkennung. Dieser Zusammenhang wird durch [Sch07] untermauert. Eine frühe Fehlererkennung rechtfertig daher fast jeden Testaufwand [FLS02]. Der Testprozess muss somit in den Entwicklungsprozess integriert werden.

Der Entwicklungsprozess kann in mehrere Phasen unterteilt werden, welche sich auch am V-Modell orientieren (vgl. 1.2, S. 3). Zu Beginn der Entwicklung werden die allgemeinen Leistungsmerkmale für die neue Baureihe festgelegt. Aus diesen Systemanforderungen folgt die Systemspezifikation. Da die Implementierung der Systeme durch das Zusammenspiel einzelner Steuergeräte erfolgt, muss die Aufteilung der Aufgaben definiert werden. Als Resultat hieraus entsteht die Komponentenspezifikation. Darauf basierend erstellen die Zulieferer das Softwaredesign sowie die Steuergeräte-Software. Zu jeder Phase der Entwicklung existiert eine zugehörige Testphase. Auf-

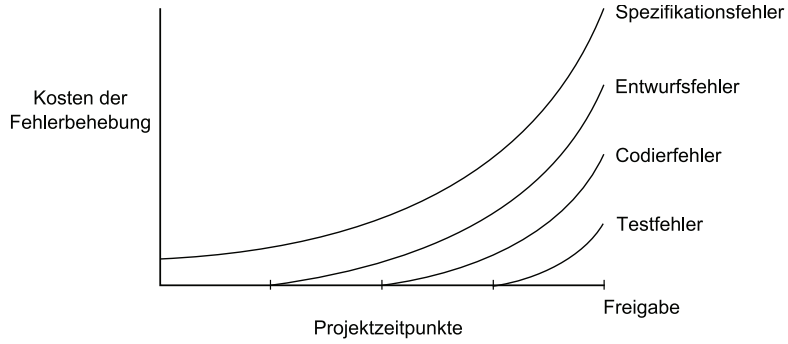

Abbildung 1.1: Kosten der späten Fehlerbehebung nach [Krü90]

bauend auf dem Test der Softwaremodule wird die Komponente als Ganzes geprüft. Während der Modultest meist vom Zulieferer übernommen wird, ist die Zuordnung des Komponententests im Allgemeinen nicht möglich. Je nach Vereinbarung wird der Test vom Zulieferer, vom Fahrzeughersteller oder auch von Beiden durchgeführt. Bei der Systemintegration wird das Verhalten der Steuergeräte im Verbund getestet. Der Fokus liegt hierbei auf Funktionen, welche durch den Beitrag mehrerer Steuergeräte erbracht werden. Die Fahrzeugerprobung dient schließlich der Kontrolle des realen Fahrzeugs. Sowohl die Systemintegration, als auch die Fahrzeugerprobung wird ausschließlich vom Fahrzeughersteller durchgeführt.

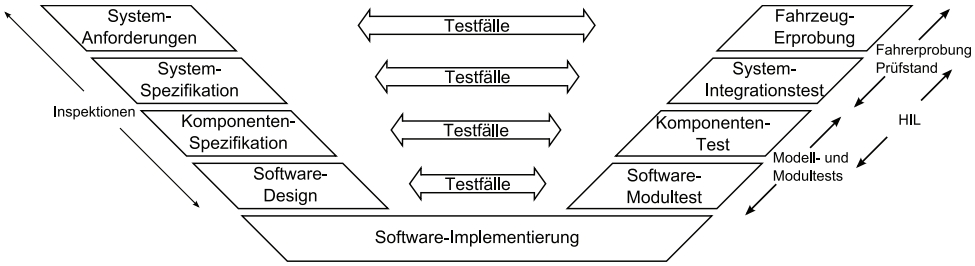

Abbildung 1.2: V-Modell in der E/E-Entwicklung

Die wachsende Bedeutung der Fahrzeugelektronik führt zu immer komplexeren Systemen und Komponenten. So hat sich die Anzahl der Steuergeräte der aktuellen S-Klasse der Daimler AG im Vergleich zum 1998 erschiene-

nen Vorgänger um ca. 25% erhöht. Gleichzeitig werden auch die einzelnen Komponenten immer leistungsfähiger. Hatte beispielsweise eine Motorsteuerung 1995 noch eine Programmlänge von etwa 100 KB, so sind es zehn Jahre später bereits 2000 KB. Abbildung 1.3 (S. 4) zeigt die Entwicklung der Programmlänge von Motorsteuerungen in den letzten Jahrzehnten.

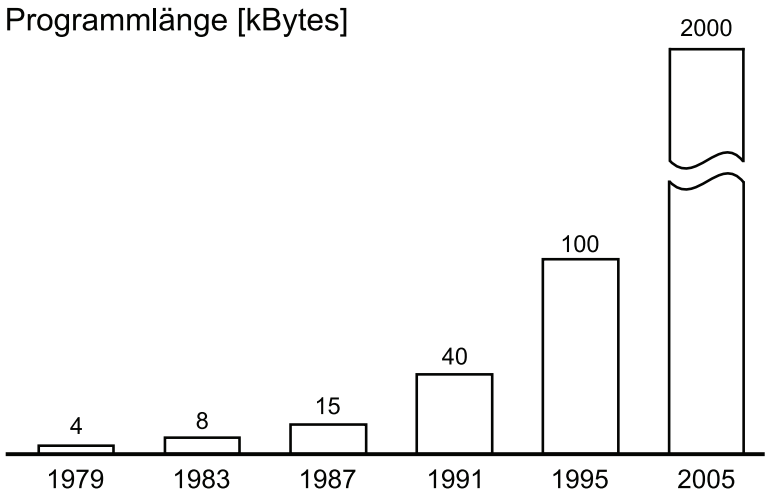

Abbildung 1.3: Entwicklung Programmlänge von Motorsteuerungen nach [Kus06]

Die Kombination aus immer zahlreicheren Komponenten sowie die Steigerung ihrer jeweiligen Komplexität stellt große Herausforderungen an den Entwicklungs- und Testprozess. Die Notwendigkeit einer engen Abstimmung dieser beiden Prozesse ist offenkundig. In dieser Arbeit liegt dabei der Fokus auf der Integrationsphase. Ein wichtiger Aspekt dieser Phase sind die Regressionstests. Diese erlauben zu jedem neuen Release die exakt gleichen Prüfschritte durchzuführen. Damit wird sichergestellt, dass einmal gefundene und behobene Fehler in einem zukünftigen Release nicht wieder auftreten oder zumindest erneut gefunden werden. Daraus resultiert jedoch auch, dass die Testsuite über die Projektdauer kontinuierlich wächst und somit die Ausführdauer stetig zunimmt. Die im Zuge dieser Arbeit betrachteten Hardware-in-the-Loop (HIL) Testsysteme müssen sich somit der Herausforderung ständig zunehmder Testumfänge stellen. Um dies auch zu zukünftig bewältigen zu können, müssen neue Methoden zur Steigerung der für Tests

zur Verfügung stehenden Zeit angewandt werden. Eine Verkürzung der Kernentwicklungszeit des Fahrzeugs auf zwei Jahre [Sch03] wird diese Situation weiter verschärfen.

1.2 Problembeschreibung

Die Problematik der steigenden Komplexität auf der einen und der verkürzten Entwicklungszeit auf der anderen Seite stellt besonders die Integrationstests vor enorme Herausfoderungen. In dieser Arbeit wird speziell auf die Situation bei den Hardware-in-the-Loop Prüfständen zum Systemintegrationstest der PKW-Entwicklung der Daimler AG eingegangen.

Hardware-in-the-Loop (HIL)
HIL bezeichnet ein Verfahren, bei welchem reale elektronische Komponenten in einer simulierten Umgebung getestet werden. Das Testsystem besteht dabei im Wesentlichen aus zwei Teilen. Zum einen aus der zu testenden Komponente, zum anderen aus einem Simulationsrechner. Der Simulationsrechner berechnet in Echtzeit die Umgebung des Testobjekts, so dass dieses keinen Unterschied zwischen dem Laboraufbau und einem realen Fahrzeug erkennt. Um dies erreichen zu können, müssen die Aus- und Eingänge der Komponente mit den Ein- und Ausgängen des Simulationsrechners verbunden werden. Die hierdurch entstehende Regelschleife zeichnet Hardware-in-the-Loop Testsysteme aus. HIL-Testsysteme werden sowohl zum Test einzelner Komponenten (Komponenten-HIL), als auch für Systemintegrationstests (System-HIL) verwendet. Ein detailierte Beschreibung von Hardware-in-the-Loop Testsystemen folgt im Kapitel 2.3.2 (S. 31).

Die Aufgabe der Systemintegrationstests besteht in der Absicherung der Steuergeräte im Systemverbund. Da bereits der Tausch einer Komponente Einfluss auf das gesamte System hat, ist es wichtig die Testsuite komplett über eine unveränderte Konfiguration[2] durchzuführen. Damit dies sichergestellt wird, werden sogenannte Releasetermine definiert, zu welchen Ände-

[2]Eine Konfiguration bezieht sich in diesem Zusammenhang auf die Kombination der Steuergeräte. Sowohl Änderungen der Hardware als auch der Software führen zu einen neuen Steuergerätestand und somit zu einer neuen Konfiguration

rungen eines Steuergeräts einfließen können. Dies bedeutet aber gleichzeitig, dass die Zeit zum Test eines Release begrenzt ist. Trotz des hohen Automatisierungsgrades der Tests und der damit gewonnenen Zeit nachts und am Wochenende wird es aufgrund des steigenden Testumfangs immer anspruchsvoller die Testsuite in der geforderten Zeit durchzuführen. Um dies auch zukünftig gewährleisten zu können müssen neue Methoden zur Optimierung der Testaktivitäten angewandt werden.
Ein möglicher Weg ist die Parallelisierung von Testprogrammen. Gegenüber der konventionellen Vorgehensweise der sequentiellen Ausführung kann somit die Ressource *Zeit* in die Ressource *Rechenleistung* umgewandelt werden. Dabei ist jedoch zu beachten, dass keine Wechselwirkungen zwischen den Testprogrammen bestehen, welche das Testergebnis wertlos machen. Die Definition allgemeiner Ursachen, sowie das Erkennen konkreter Wechselwirkungen stehen daher im Mittelpunkt dieser Arbeit.

1.3 Ziel der Arbeit

Elektronische Systeme moderner Kraftfahrzeuge stellen hohe Ansprüche an den Entwicklungs- und Testprozess. Dabei wird der Testumfang nicht mehr durch die Technik, sondern vielmehr durch die zur Verfügung stehende Zeit begrenzt. Für die Arbeit ist das folgende Problem somit der zentrale Aspekt.

> Wie kann unter Einbehaltung beziehungsweise Steigerung der Qualität elektronischer Systeme die Testausführungszeit minimiert werden?

Gleichzeitig führt der technische Fortschritt auch zu immer leistungsfähigeren Prüfsystemen. So stehen der zeitlichen Vollauslastung der Testsysteme freie Ressourcen bei der Rechenleistung gegenüber. Ziel muss es daher sein, die Rechenlast zugunsten verkürzter Ausführungszeiten zu erhöhen. Eine Parallelisierung mehrerer Testprogramme kann diese Verschiebung der Ressourcen bewirken.

> Ziel dieser Arbeit ist ein Konzept, welches eine Abfolge von Testprogrammen festlegt, bei der die Testprogramme auch parallel ausgeführt werden können und somit die Gesamtausführungszeit minimiert wird. Dabei muss sichergestellt sein, dass keine ungewollten Wechselwirkungen zwischen den Testprogrammen bestehen.

1.4 Aufbau der Arbeit

Kapitel 2 *„Grundlagen“* beschreibt das Umfeld, in welchem sich diese Arbeit bewegt. Es beginnt mit einer Einführung in die klassischen Methoden der Testfallermittlung sowie der Beschreibung von Testsystemen. Auf Basis einer allgemeinen Definition des Begriff des *Systems* wird dessen Relevanz für die PKW Entwicklung diskutiert. Der Abschluss des Kapitels widmet sich dann dem Systemintegrationstest elektrisch/elektronischer Komponenten.

In Kapitel 3 *„Stand der Technik“* werden aktuelle Entwicklungen zur Parallelisierung sequentieller Abläufe betrachtet und auf ihre Eignung für den Systemintegrationstest bewertet.

Kapitel 4 *„Fahrzeugfunktionen“* analysiert die einzelnen Bestandteile einer Funktionsbeschreibung und zeigt die Defizite aktueller Testprogramme, welche eine parallele Ausführung verhindern. Aus diesen Ergebnissen werden Foderungen an zukünftige Testprogramme abgeleitet. Diese schaffen die Grundlage für eine automatisierte Konflikterkennung und somit für die Parallelisierung der Testprogramme.

Ein Überblick über den Prozess der parallelen Testausführung gibt Kapitel 5 *„Ablauf der parallelen Testausführung“*. Von der Auswahl der Testprogramme über die Konflikterkennung bis hin zur Auswertung der Ergebnisse werden hier die einzelnen Phasen beschrieben.

Kapitel 6 *„Konflikterkennung“* beschreibt das Konzept, welches die Identifizierung von Wechselwirkungen zwischen Testprogrammen ermöglicht. So können beispielsweise Konflikte zwischen Funktionen bestehen, falls sie auf die gleichen Ressourcen zugreifen. In diesem Fall wären die Testergebnisse bei einer parallelen Ausführung nicht aussagekräftig. Auf Basis einer Analyse möglicher Konfliktursachen werden in diesem Kapitel Regeln definiert, welche Wechselwirkungen zwischen Funktionen im Vorfeld erkennen lassen.

In Kapitel 7 *„Optimierung“* wird ein Verfahren zur Bestimmung einer zeitlich optimalen Testsuite vorgestellt. Dieses Verfahren basiert auf den Erkenntnissen aus Kapitel 6 und soll einen möglichst großen Nutzen einer Parallelisierung von Testprogrammen gewährleisten.

Kapitel 8 *„Zusammenfassung und Ausblick“* gibt einen kurzen Abriß über die zentralen Aspekte dieser Arbeit und fasst deren Ergebnisse zusammen. Ferner werden Ideen zur weiteren Nutzensteigerung des Verfahrens angesprochen, welche als Ansätze für weitere Forschungsaktivitäten herangezogen werden können.

Kapitel 2

Grundlagen

Ziel dieses Kapitels ist die Beschreibung des Umfelds der Arbeit. Beginnend mit einer allgemeinen Einführung in die Testmethodik, wird im weiteren Verlauf der zentrale Begriff des *Systems* definiert. Dieser wird für den Anwendungsfall der Kraftfahrzeugelektronik konkretisiert. Eine Übersicht unterschiedlicher Testsysteme sowie die Organisation der Testdurchführung an den Hardware-in-the-Loop Prüfständen zum Systemintegrationstest der Daimler AG bilden den Abschluss des Kapitels.

2.1 Testmethodik und Systemtheorie

Immer komplexere technische Produkte bedingen nicht nur einen steigenden Testaufwand. Ohne einen systematischen Ansatz und effiziente Werkzeuge ist es schwierig die Komplexität zu beherrschen. Dies beginnt mit der Testfallermittlung, welche eine möglichst große Testabdeckung bei minimalem Aufwand ermöglichen soll.

2.1.1 Methoden der Testfallermittlung

Der Begriff *Testen* bezeichnet Aktivitäten mit dem Ziel, Fehler im Testobjekt aufzudecken und somit das Risiko hoher Folgekosten zu verringern. Da ein vollständiger Test bereits relativ einfacher Prüfobjekte in einem immens hohen Aufwand resultiert, ist lediglich die Anwesenheit von Fehlern nachweisbar, nicht jedoch deren Abwesenheit [Sch02]. Daher soll durch den gezielten Einsatz unterschiedlicher Testmethoden, unter wirtschaftlich vertretbarem

Aufwand, die geforderte Qualität sichergestellt werden. Dabei wird zwischen zwei generellen Methoden unterschieden, dem *statischen* und dem *dynamischen Test*.

Statischer Test

Kennzeichnend für statische Tests ist, dass keine Ausführung des Testobjekts stattfindet. Sie eignen sich somit nicht nur für Programme, sondern sind auf alle Dokumentarten anwendbar [SL04]. Im Folgenden werden exemplarisch Vertreter von statischen Tests vorgestellt. Genaueres hierzu kann [Ins97] entnommen werden.

Review: Mit Review wird ein Analyse- und Bewertungsprozeß bezeichnet, bei welchem Ergebnisse einem Begutachterteam präsentiert und von diesem kommentiert oder genehmigt werden. Nach [SL04] können durch Reviews bis zu 70% der Fehler gefunden werden. Je nach Zielsetzung existieren Reviewarten, welche mehr oder weniger formal geregelt sind. Beispiele hierfür sind technische Reviews, Inspektionen oder Audits.

Statische Analyse: Bei der statischen Analyse wird das Testobjekt einer Reihe formaler Prüfungen unterzogen. Im Gegensatz zum Review kann dieser Vorgang automatisiert werden. So ist beispielsweise der Compiler einer Programmiersprache ein Vertreter der statischen Analyse, indem er etwa die Initialisierung einer Variablen prüft. Um die statische Analyse einsetzen zu können, muss das Testobjekt in einer formalen Struktur vorliegen. Weitere Beispiele für dieses Verfahren sind die Datenflußanalyse und die Kontrollflußanalyse [SL04].

Dynamischer Test

Beim dynamischen Test wird das Programm mit definierten Eingangsparametern zur Ausführung gebracht. Das Ergebnis wird mit den Sollwerten aus der Spezifikation verglichen. Eine Unterteilung bei der Vorgehensweise der Testfallermittlung führt zu drei Verfahren.

Blackbox-Verfahren: Bei diesem Verfahren wird das Testobjekt als Blackbox betrachtet, d.h. es sind keine Informationen über die interne Struktur verfügbar. Die Testfälle werden daher vollständig aus der Spezifikation gewonnen. Dabei stehen funktionale Tests im Mittelpunkt [SL04].

Ein typisches Vorgehen für das Blackboxverfahren ist die Bildung sogenannter *Äquivalenzklassen*. Hierbei werden alle Eingangsdaten, welche dasselbe Ergebnis hervorrufen, zu einer Klasse zusammengefasst. Somit kann der Testumfang auf einen Repräsentant jeder Äquivalenzklasse reduziert werden. Häufig wird zusätzlich eine *Grenzwertanalyse* durchgeführt. Da Fehler oft an den Grenzen einer Äquivalenzklasse auftreten, werden hierbei die Grenzwerte sowie deren „nähere“ Umgebung betrachtet.

Whitebox-Verfahren: Im Unterschied zum Blackbox-Verfahren basiert die Testfallermittlung beim Whitebox-Verfahren auf der internen Struktur des Programms. So kann die Abdeckung des Quellcodes bezüglich eines gegebenen Maßes gemessen und somit auch gefordert werden. Gängige Maße sind:

- Anweisungsüberdeckung: Ausführung von mind. x% aller Anweisungen (C0-Maß)
- Zweigüberdeckung: Durchlaufen von mind. x% aller Zweige (C1-Maß)
- Pfadüberdeckung: Betrachtung von mind. x% aller Pfade (C7-Maß)

Nähere Informationen, sowie weitere Überdeckungsmaße können beispielsweise unter [Sch02], [SL04] oder [Wik06] nachgelesen werden.

Modellbasierte Verfahren: Bei diesem Verfahren werden die Testfälle aus einem Modell abgeleitet. Dieses Modell beschreibt in vereinfachter Weise das Testobjekt [UL07] und beschränkt sich somit auf die wesentlichen Eigenschaften des Testobjekts. Aus diesem Modell werden nach vorgegebenen Kriterien die Testfälle generiert. So können sich die Testfälle beispielsweise auf bestimmte Bereiche des Modells beschränken.

Zusätzlich zu beiden methodischen Ansätzen sollte in der Praxis das „intuitive Testen“ nicht vernachlässigt werden. Hierbei wird die Erfahrung des Testers ausgenutzt, deren Einfluss auf die Güte der Tests nicht zu unterschätzen ist [Sch02].

2.1.2 Testsysteme

Um effizient testen zu können, wird neben der systematischen Testfallermittlung ein leistungsfähiges Testsystem benötigt. Wesentlicher Bestandteil ist dabei die Automatisierung der Testprogramme, welche ohne Rechnerunterstützung nicht möglich wäre. Die für diese Arbeit gültige Definition für Testsysteme stammt von [Har01]

Definition: Testsystem [Har01]
Ein Testsystem ist ein rechnergestütztes Werkzeug, welches den automatisierten Test von elektronischen Komponenten und Systemen eines Kraftfahrzeugs gestattet.

Abbildung 2.1 zeigt den schematischen Aufbau eines Testsystems. Über ein Testprogramm wird die Eingabe für das Testobjekt generiert. Eine Umgebungssimulation stellt sicher, dass alle relevanten Eigenschaften des Umfelds während des Tests bereitgestellt werden. Die Reaktionen des Testobjekts werden durch das Testsystem protokoliert, beispielsweise durch das Aufzeichnen von Messungen. Hierdurch wird im Nachgang eine Lokalisierung von Fehlern möglich. All diese Schritte werden durch das Testsystem automatisiert ausgeführt.

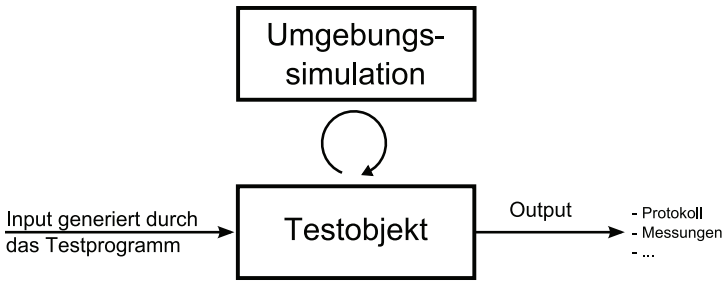

Abbildung 2.1: Testsystem

Die Automatisierbarkeit ist Voraussetzung für weitere Eigenschaften, welche effiziente Testsysteme erfüllen müssen.

Reproduzierbarkeit Die Aufgaben von Testsystemen enthalten meist die Durchführung von Regressionstests. Um vergleichbare Ergebnisse erzielen zu können, muss der Test exakt reproduzierbar sein. Im Umfeld von Echtzeitsystemen (vgl. 2.1.3, S. 18) stellt das auch Anforderungen an den zeitlichen Ablauf der Stimulationen.

Nebenläufigkeit Elektronische Komponenten eines Kraftfahrzeugs führen meist mehrere nebenläufige Prozesse aus. Es kann daher notwendig sein, mehrere dieser Prozesse gleichzeitig beobachten oder stimulieren zu können. Eine parallele Ausführung mehrerer Testsequenzen muss demnach vom Testsystem unterstützt werden.

Fehlerlokalisierung Wird ein Fehlverhalten des Testobjekts festgestellt, muss die Ursache hierfür ermittelt werden können. Die Nachvollziehbarkeit der einzelnen Testschritte sowie der Reaktionen des Testobjekts darauf muss daher gewährleistet sein.

Umgebungssimulation Interagiert das Testobjekt mit seiner Umgebung, so muss diese hinreichend simuliert werden. Bei reaktiven Systemen (vgl. 2.1.3, S. 19) reichen hierfür statische Methoden nicht aus. In diesem Fall müssen Modelle der Umgebung existieren. Näheres zu Modellen im Abschnitt 2.1.3 (S. 21).

Visualisierung und Steuerung Das Testsystem muss eine Schnittstelle zur Verfügung stellen, über welche der Testingenieur das System beobachten und steuern kann. Sie muss derart geschaffen sein, dass neben der automatisierten Testdurchführung auch eine manuelle Steuerung möglich ist [Kus06].

Dieser Arbeit liegen sogenannte Hardware-in-the-Loop Testsysteme zugrunde. Eine detaillierte Beschreibung dieser Systeme wird im Abschnitt 2.3.2 (S. 31) gegeben.

2.1.3 Systeme

Getrieben durch den starken Wettbewerbsdruck - speziell in der Automobilindustrie - hat sich eine rasante Weiter- und Neuentwicklung technischer

Produkte ergeben. Gleichzeitig steigen auch die Anforderungen an die Qualität der Produkte. Fehlverhalten wird deutlich kritischer beurteilt und nicht mehr ohne weiteres hingenommen [B+97]. Um der steigenden Komplexität, sowie den Ansprüchen an das Produkt bei immer kürzeren Innovationszeiten gerecht zu werden, ist eine ganzheitliche Herangehensweise erforderlich. Systems Engineering (SE) bietet hierfür einen interdisziplinären Ansatz, der den Entwicklungsprozeß vom Konzept über die Produktions- bis zur Betriebsphase unterstützt [INC04]. Von großer Bedeutung ist dabei der Gedanke, die Gesamtheit des darbietenden Problems durch Systeme abbilden zu können [B+97].

Allgemeiner Systembegriff

Für den Begriff *System* gibt es eine Vielzahl an Definitionen, die jeweils stark durch die Arbeits- bzw. Wissensgebiete geprägt sind, aus welchen sie hervorgehen. So unterschiedlich diese Gebiete auch sein mögen, so sind im Kern alle Definitionen gleich. Ein System besteht immer aus einer Anordnung von Elementen, welche in Beziehung zueinander stehen. Für die Arbeit ist folgende Definition gültig.

Definition: System [Sch02]
Ein System ist eine Anordnung von Gebilden (für die jeweilige Sicht abgrenzbare Elemente), die aufeinander durch Relationen einwirken und die durch eine Hüllfläche, die Systemgrenze, von ihrer Umgebung abgegrenzt sind.

Findet über die Systemgrenze hinweg ein Informationsaustausch statt, so handelt es sich um ein *offenes* System. Andernfalls heißt es *geschlossen*. Die Systemgrenze muss nicht physisch sichtbar sein und kann je nach Betrachtungsweise unterschiedlich verlaufen [B+97].

Systemkonzepte

Abhängig von der beabsichtigten Sicht auf ein System gibt es unterschiedliche Betrachtungsweisen, die die jeweils relevanten Eigenschften des Systems in den Vordergrund stellen.

- das **funktionale** Konzept stellt die Funktion des System in den Vordergrund, welche durch die Wandlung von Eingabeoperanden in Ausgabeoperanden beschrieben wird

- das **strukturelle** Konzept richtet den Fokus auf die Beschreibung der Elemente, deren Verhalten sowie auf die Schnittstellen zwischen den Elementen.

- das **hierarchische** Konzept beschreibt ein System durch Zerlegen in Teilsysteme.

Subsysteme und Supersysteme

Werden einzelne Elemente wiederum als System aufgefasst, indem Elemente auf tieferer Ebene gebildet und in Beziehung zueinander gesetzt werden, so entstehen Subsyteme (auch Teilsysteme). Dieser rekursive Vorgang kann solange wiederholt werden, bis eine weitere Unterteilung nicht mehr sinnvoll ist. Auf diese Art und Weise erhält man einen hierarchischen Systemaufbau (vgl. Abb. 2.2, S. 15). Charakteristisch für die Elementarstufe ist, dass keine weitere Verfeinerung erwünscht ist und die Elemente als Black-Boxes angesehen werden [B+97]. Aus der Existenz von Subsystemen folgt zwingend der Begriff des Supersystems. Dieses ergibt sich aus der Zusammenfassung mehrerer Systeme.

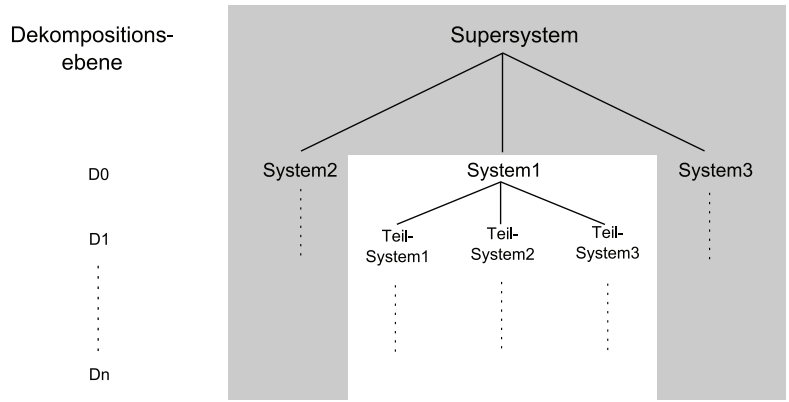

Abbildung 2.2: Systemhierarchie nach [Kus06]

Relationen (Schnittstellen)

Die Interaktion der einzelnen Elemente untereinander wird durch Relationen geregelt. Dabei können zwischen zwei Elementen auch mehrere solcher Relationen gleichzeitig definiert sein. Gleiches gilt für die Beziehung zwischen zwei Systemen. Die durch die Elemente und deren Beziehungen zueinander entstehende Ordnung wird auch Systemstruktur genannt [B+97].

Systemumgebung

Als Systemumgebung werden alle Systeme und Elemente bezeichnet, welche außerhalb der Systemgrenze liegen. Damit sinnvoll von einer Systemumgebung gesprochen werden kann, muss ein Mindestmaß an Interaktion über die Systemgrenze hinweg stattfinden. Der Verlauf der Systemgrenze kann meist jedoch nicht eindeutig bestimmt werden sondern variiert je nach Betrachtungsweise. Als Richtlinie dient dabei, dass innerhalb der Systemgrenze ein größeres Maß an Beziehungen besteht als über sie hinweg [Har64].

Funktionen

Der Begriff der *Funktion* wird in unterschiedlichsten Zusammenhängen verwendet. Eine allgemeine Definition ist daher nur auf einer stark abstrahierten Ebene möglich. Nach [Bro07] bezeichnet eine Funktion eine Tätigkeit oder Aufgabe. Dies gleicht jedoch eher einer Aufzählung von Synonymen als einer Definition. Für einzelne Anwendungsbereiche muss diese daher konkretisiert werden. Laut der mathematischen Definition ist eine Funktion beispielsweise eine Abbildung, welche einem Wert einer Ausgangsmenge eindeutig einen Wert einer Zielmenge zuweist [HSZ96]. Der Grundgedanke einer Transformation von Ein- in Ausgangsgrößen lässt sich auf Funktionen im Kraftfahrzeug übertragen. Aufgrund der Vielzahl unterschiedlicher Ausprägungen von Fahrzeugfunktionen muss die für diese Arbeit gültige Definition von Funktionen etwas allgemeiner gefasst werden.

Definition: Funktion
Eine Funktion ist eine Aufgabe, die ein System für seine Umwelt zu erfüllen hat. Diese hängt von äußeren Einflüssen ab und wird in der Regel durch ein Ereignis ausgelöst.

Für die Realisierung komplexer Funktionen werden, analog zu Systemen, Teilfunktionen definiert, welche für sich betrachtet eine geringere Komplexität aufweisen. Durch das Zusammenspiel der Teilfunktionen wird dabei die gewünschte Funktionalität bereitgestellt.

Die Verwendung von Systemen bildet einen sehr allgemeinen Ansatz, komplexe Sachverhalte auf das (subjektiv) Wesentliche zu reduzieren und übersichtlich darzustellen. Um dies im Einzelfall wirkungsvoll anwenden zu können, muss der Systembegriff jedoch weiter konkretisiert werden. Nachfolgend werden daher die wichtigsten Systeme der Kfz-Elektronik beschrieben.

Verteilte Systeme

In den achtziger Jahren entstand eine Reihe neuer Technologien, welche eine Abkehr von zentralisierten Computersystemen mit einfachen Terminals möglich machte [Web98]. An deren Stelle traten nun miteinander verbundene „Personal Computer“. Erreicht wurde dies zum einen durch technische Neuerungen bei der Produktion von Mikroprozessoren. Diese erlaubten die Herstellung großer Stückzahlen und damit die Senkung der Produktionskosten. Gleichzeitig stellten Hochleistungsnetze die Kommunikation zwischen den Computern sicher. Durch die Kopplung isolierter Computer entsteht ein neues Gesamtsystem, in welchem Ressourcen wie Hardware und Software, aber auch Daten und Dienste gemeinsam genutzt werden können. Derartige Systeme werden allgemein als verteilte Systeme bezeichnet.

Definition: Verteiltes System [BBK98]
Ein verteiltes System besteht aus Komponenten, die räumlich oder konzeptuell verteilt sind und koordiniert (gekoppelt oder vernetzt) zum Erreichen der Funktionalität des Gesamtsystems beitragen.

Echtzeitsysteme

Echtzeitsysteme müssen neben der Forderung nach korrekten Ergebnissen zusätzlich Zeitbedingungen erfüllen. Steht ein Ergebnis nicht innerhalb einer vorgegebenen Zeit zur Verfügung, so ist es schlimmstensfalls unbrauchbar [Kop97]. Echtzeitsysteme müssen somit zeitlich synchron zu ihrer Umgebung Vorgänge verarbeiten[KRW05].

Definition: Echtzeitsystem [Sax99]
Ein Echtzeitsystem ist ein System, bei dem die Richtigkeit der Systemantwort sowohl von einer korrekt durchgeführten Transformation der Eingangsgrößen in die Ausgangsgrößen, als auch von dem Zeitpunkt, zu dem die Ausgangsgrößen zur Verfügung gestellt werden, abhängt.

Aus der Definition lassen sich zwei Eigenchaften von Echtzeitsystemen ableiten. Zum einen müssen die Verarbeitungsergebnisse des Systems rechtzeitig vorliegen. Gemessen vom Zeitpunkt des Auftretens eines Ereignisses bis zur Verfügbarkeit der Ergebnisse darf eine definierte Zeitspanne nicht überschritten werden. Diese Eigenschaft wird *Rechtzeitigkeit* genannt. Zum anderen müssen mehrere Aufgaben parallel bearbeitet werden können, wobei wiederum für alle Aufgaben die Forderung nach Rechtzeitigkeit gilt. Dies wird unter dem Begriff der *Gleichzeitigkeit* geführt [WB05].

Abhängig von den Folgen, welche aus einer Verletzung der Zeitanforderungen entstehen, wird zwischen weichen und harten Echtzeitanforderungen unterschieden [Kop97].

Weiche Echtzeitanforderung
: Hat das Nichteinhalten der Zeitvorgaben keine katastrophalen Folgen, sondern sinkt lediglich der Nutzen der Ergebnisse, so werden weiche Echtzeitanforderungen an das System gestellt. Dennoch ist das System in der Lage, im Normalfall die Zeitvorgaben einzuhalten.

Harte Echtzeitanforderung
: Führt das Nichteinhalten der Zeitvorgaben zu katastrophalen Folgen

für das System oder die Umgebung, so werden harte Echtzeitanforderungen an das System gestellt.

Reaktive Systeme

Systeme, die in ständiger Interaktion mit ihrer Umgebung stehen, werden reaktive Systeme genannt. Sie reagieren kontinuierlich auf Eingaben, die ausschließlich aus deren Umgebung stammen [S+99]. Oft unterliegen reaktive Systeme Zeitvorgaben und sind somit auch Echtzeitsysteme.

Definition: Reaktives System [Geh00]
Ein reaktives System kommuniziert während seiner Ausführung fortlaufend mit seiner Umgebung, so daß ein permanenter Datenaustausch stattfindet.

Diskrete und kontinuierliche Systeme

Jedes System kann anhand seiner Zustände zu bestimmten Zeitpunkten beschrieben werden. Ist die Menge der Zustände abzählbar, handelt es sich um wertdiskrete Systeme. Wertkontinuierliche Systeme besitzen hingegen überabzählbare Zustandsmengen [Kie97]. Analog werden zeitdiskrete bzw. zeitkontinuierliche Systeme definiert. Hierfür wird die Menge der Änderungszeitpunkte betrachtet. Abbildung 2.3 (S. 20) stellt die vier möglichen Kombinationen dar.
Verarbeitet ein System sowohl diskrete als auch kontinuierliche Datenanteile oder interagiert es über kontinuierliche Zeiträume und zu diskreten Zeitpunkten mit seinem Umfeld, ist von hybriden Systemen die Rede [BBK98].

Eingebettete Systeme

Charakteristisch für eingebettete Systeme ist die vollständige Integration in ein technisches Umfeld. Die Schnittstellen zwischen System und Umfeld können zum Teil auch nichtelektronischer Art sein. In diesem Fall ermöglichen Sensoren und Aktoren die Interaktion mit der Umwelt, indem sie nichtelektronische physikalische Größen auf elektrische Signale abbilden und umgekehrt [Kus06]. Die enge Verbindung von System und Umfeld bedingt eine

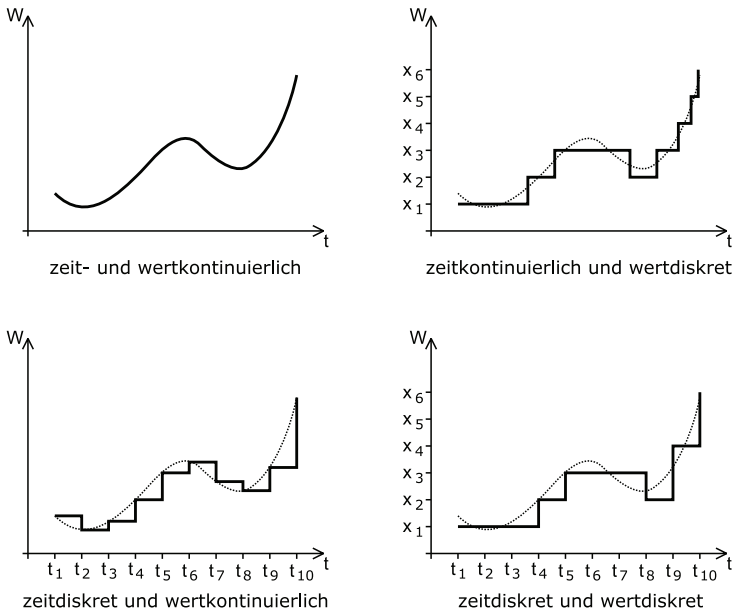

Abbildung 2.3: Diskrete und kontinuierliche Systeme [Kie97]

spezielle Anpassung von Hard- und Software auf den gegebenen Anwendungsfall [Mar07].

Definition: Eingebettetes System [BBK98]
Ein eingebettetes System ist eine Software-/Hardware-Einheit, die über Sensoren und Aktuatoren mit einem Gesamtsystem verbunden ist und darin Überwachungs-, Steuerungs- beziehungsweise Regelungsaufgaben übernimmt. In der Regel handelt es sich bei eingebetteten Systemen um reaktive, häufig auch um hybride verteilte Systeme mit Echtzeitanforderungen. Typischerweise sind solche Systeme dem menschlichen Benutzer nicht direkt sichtbar, er interagiert unbewusst mit dem eingebetteten System.

Für den Test eingebetteter Systeme ist eine Testumgebung notwendig, welche sowohl die Beobachtung als auch die Stimulation aller Schnittstellen zum Testobjekt ermöglicht. Hierbei muss die zunehmende Komplexität durch die meist verteilt realisierte Funktionalität sowie die Kommunikation beispielsweise über Bussysteme berücksichtigt werden. Dies macht den Test eingebet-

teter Systeme im Allgemeinen recht aufwendig.

Signale

Analog zu Systemen können auch Signale in kontinuierliche (analoge) und diskrete (digitale) Signale unterteilt werden.

Definition: Signal [Gra06]
Die Darstellung einer Mitteilung durch zeitliche Veränderungen einer physikalischen Größe heißt Signal.

Für diese Arbeit wird diese Definition erweitert. So können durch ein Signal auch logische oder abstrakte Größen dargestellt werden. Damit ein Mikroprozessor kontinuierliche Signale verarbeiten kann, müssen diese in digitale umgewandelt werden. In diesem Fall muss von der Hardware eine geeignete Quantisierung vorgenommen werden.

Systemmodell

Für den Test von Systemen muss deren Umwelt in ausreichendem Maße nachgebildet werden. Spätestens bei reaktiven Systemen reichen statische Methoden hierfür nicht mehr aus. Daher kommen Modelle zum Einsatz, welche die Umwelt simulieren. Wie bei Systemen, so gibt es auch für Modelle eine Vielzahl an Definitionen. Folgende ist für diese Arbeit relevant.

Definition: Systemmodell [Tab06]
Ein Systemmodell ist eine Abstraktion zu einem System (im Sinne des Systemgebildes), welche nur eine Menge ausgewählter, gerade interessierender Sachverhalte des betrachteten Systems aufweist.

Grundsätzlich weisen Modelle drei Merkmale auf [Sch02]

- Das **Abbildungsmerkmal** kennzeichnet das Modell als Abbildung eines (realen) Systems.
- Das **Verkürzungsmerkmal** ergibt sich aus der Reduktion der Komplexität des realen Systems. Es werden lediglich die Elemente und Rela-

tionen des realen Systems abgebildet, welche dem Modellierer relevant erscheinen.

- Das **pragmatische Merkmal** beschreibt das Ziel der Modellbildung. Folgende Modelle können hierbei unterschieden werden
  - **Beschreibungsmodelle:** Zweck ist die Informationsgewinnung über die Beschaffenheit des Systems durch Beschreibung des Systems und seiner Entscheidungssituationen.
  - **Erklärungsmodelle:** Ziel ist die Aussage über künftige Systemzustände, indem reale Erscheinungen des Systems erklärt werden.
  - **Entscheidungsmodelle:** Ausgehend von vorgegebenen Randbedingungen und Zielsetzungen sollen Handlungsmaßnahmen abgeleitet werden.

Da Modelle als Ersatz realer Systeme verwendet werden, lassen sich die Eigenschaften der beschriebenen Systeme größtenteils auf Modelle übertragen. Von besonderer Bedeutung sind in diesem Zusammenhang Echtzeitmodelle. Hierbei steht der Modellierer vor der zusätzlichen Aufgabe, das Modellverhalten derart zu optimieren, dass die Ausgaben rechtzeitig vorliegen. Im weiteren Verlauf sind Modelle immer Echtzeitmodelle.

2.2 Systeme im Automobil

Systeme beschreiben im Grundsatz die bereitzustellende Funktionialität eines Kraftfahrzeugs. Dabei kann es sich um elektronische geprägte oder mechanisch realisierte Systeme handeln. Auch eine Mischung aus beiden ist möglich. Systeme werden durch sukzessives Zerlegen in Teilsysteme konkreter definiert. Im Folgenden werden ausschließlich elektronisch geprägte Systeme betrachtet. Beispiele für solche Systeme sind etwa die Motorsteuerung oder die Außenbeleuchtung. Ein Teilsystem der Außenbeleuchtung ist beispielsweise die Blinkersteuerung. Die Realisierung der Systeme erfolgt durch Steuergeräte, welche untereinander vernetzt sind. Dies erlaubt die Verteilung einer Funktion auf mehrere Steuergeräte. Tabelle 2.1 (S. 23) zeigt beispielhaft den Zusammenhang von Systemen und Steuergeräten.

Die genaue Bezeichnung der in Tabelle 2.1 aufgeführten Steuergeräte findet sich auf Seite 106.

	EZS	MRSM	SAM_V	SAM_H	TSG_VL	TSG_VR	TSG_HL	TSG_HR
Außenbeleuchtung	X	X	X	X	X	X		
Fensterheber	X				X	X	X	X
Schliessung	X		X		X	X	X	X
Wischer	X	X	X	X				

Tabelle 2.1: Systeme und Steuergeräte

2.2.1 Steuergeräte

Steuergeräte (SG, auch ECU für Electronic Control Unit) sind die physikalische Umsetzung eines eingebetteten Systems eines Kraftfahrzeugs [BBK98]. Damit die hohen Ansprüche an die Systeme erfüllt werden können, ist ein hoch entwickeltes Steuer- und Regelungskonzept notwendig. Hierfür muss der Zustand des Umfelds erfasst, ausgewertet und gegebenfalls beeinflusst werden. Über Sensoren werden physikalische Größen wie Motordrehzahl gemessen und in elektrische Signale gewandelt. Die Verarbeitung erfolgt durch einen oder mehrere Mikroprozessoren. Dabei hat neben den Eingangssignalen auch der aktuelle Zustand, in welchen sich das Stuergerät, Einfluß auf die Werte der Ausgabesignale. Eine Ansteuerung der Aktoren durch das Steuergerät erfolgt nach entsprechender Aufbereitung der Ausgangssignale (vgl. Abbildung 2.4, S. 23).

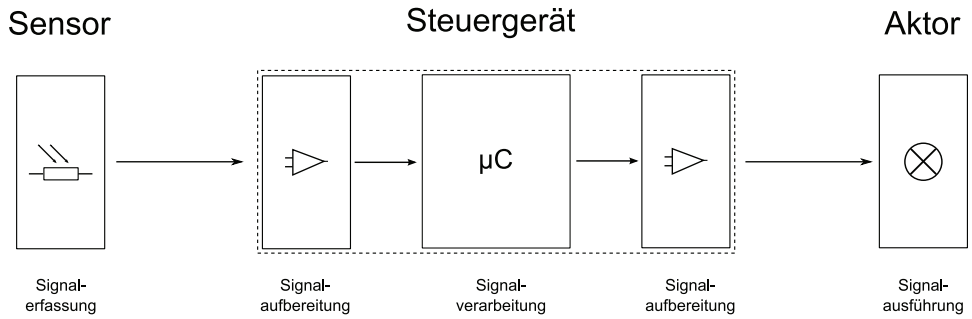

Abbildung 2.4: Schematischer Aufbau eines Steuergerätes mit Peripherie nach [Sch03]

In den letzten Jahren hat sich die Anzahl an verbauten Steuergeräten auf bis zu 60 bei Fahrzeugen der Premiumklasse gesteigert. Derzeit ist jedoch ein gegenläufiger Trend zur Reduzierung der Anzahl an Steuergeräten zu erkennen [Kus06]. Ziel ist die Kostensenkung durch Einsparung zusätzlicher Hardware. Des Weiteren ist ein Zuwachs sogenannter *intelligenter Sensoren* bzw. *Aktoren* zu verzeichnen. Hierbei werden bestimmte Funktionalitäten vom Steuergerät auf den Sensor bzw. Aktor übertragen.

2.2.2 Vernetzung

In den Anfängen des Einsatzes elektronischer Komponenten in Kraftfahrzeugen waren die Steuergeräte nicht untereinander vernetzt. Die Funktion war damit auf die unmittelbar angeschlossene Peripherie und das Steuergerät selbst beschränkt. Mit der Zunahme elektronischer Systeme war dieses Vorgehen jedoch weder technisch, noch wirtschaftlich sinnvoll. Durch die Vernetzung der Steuergeräte ist es erstmals möglich, neue E/E-Systeme auf bestehende Komponenten zu verteilen. Sofern keine weiteren Aktoren oder Sensoren für die Funktion notwendig sind, kann auf den Einsatz zusätzlicher Hardware verzichtet werden. Des Weiteren kann unter Umständen auch auf einzelne Komponenten verzichtet werden, indem deren Funktionalität auf andere Steuergeräte verteilt wird. Dies senkt die Kosten und den Platzbedarf für elektronische Komponenten. Nicht zuletzt reduziert sich auch das Gesamtgewicht des Fahrzeugs. Ein gerade bei der aktuellen Diskussion um spritsparende und umweltfreundliche Fahrzeuge attraktiver Punkt. Weitere Einsparungen können durch effiziente Verkabelungen von Sensoren und Aktoren erzielt werden. Es ist nicht notwendig sie mit dem Steuergerät zu verbinden, welches mit ihnen funktional in Verbindung stehen muss. Sie können vielmehr am räumlich nächstgelegenen Steuergerät angeschlossen werden. Durch die Vernetzung sind sie allen Teilnehmern verfügbar. So werden beispielsweise die Blinkleuchten im Außenspiegel durch das entsprechende Türsteuergerät angesteuert, obwohl die Blinklogik in einer anderen Komponente realisiert ist.

Je nach Anforderung an die Leistungsfähigkeit der Netzwerke werden verschiedene Varianten von Bussystemen eingesetzt. Dabei sind die wichtigsten Charakteristika die Übertragungsrate, Sicherheitsaspekte (z.B. Ausfallsicherheit), Priorisierung von Botschaften und Kosten. Im Folgenden werden die gängigen Bussysteme und ihr Einsatzgebiet beschrieben.

Local Interconnect Network (LIN)

Das Local Interconnect Network ist ein einfaches, kostengünstiges Bussystem mit einer Datenrate von maximal 20 kbit/s [LK]. LIN Busse dienen typischerweise der Kommunikation zwischen intelligenten Sensoren bzw. Aktoren und einem Steuergerät, welches über einen weiteren Bus (meist CAN) mit anderen Steuergeräten in Verbindung steht (vgl. Abbildung 2.5, S. 29).

Ein LIN Bus besteht aus einen Master und einem oder mehreren Slaves. Der Master hat die Kenntnis über die zeitliche Reihenfolge der zu übertragenden Daten. Entsprechend dieser Reihenfolge werden die Slaves abgefragt. Ein Slave sendet nur nach Aufforderung durch den Master. Nachrichtenkollisionen können aufgrund dieses Verfahrens nicht auftreten. Somit sind auch keine Mechanismen zur Auflösung von Buskollisionen notwendig.

Da die Nachrichten einem vorhersagbaren zeitlichen Muster folgen, liegt dem LIN ein deterministisches Zeitverhalten zugrunde. Der LIN Bus eignet sich daher nicht für ereignisgesteuerte Kommunikation.

Controller Area Network (CAN)

Der CAN-Bus ist ein asynchrones, serielles Bussystem, das in den achtziger Jahren von der Firma Bosch entwickelt wurde [E+94]. Die maximale Übertragungsrate beträgt 1 Mbit/s, typische Datenraten sind jedoch 100 kbit/s (Klasse B Bus) und 500 kbit/s (Klasse C Bus). Beim CAN Bus handelt es sich um ein sogenanntes Multi-Master-System, d.h. alle Teilnehmer sind gleichberechtigt.

Jeder CAN-Knoten kann mit dem Senden einer Nachricht beginnen, falls der Bus nicht belegt ist. Starten zwei Knoten gleichzeitig eine Datenübertragung, sorgt ein Arbitrierungsmechanismus dafür, dass sich stets der Knoten mit der höchstprioren Nachricht durchsetzt, welche durch deren *Identifier* angegeben wird. Die Datenübertragung erfolgt über zwei Signalpegel, einem dominanten und einem rezessiven. Beim Senden des Identifiers vergleicht jeder Knoten seinen Pegel mit dem des CAN-Bus. Sind sie unterschiedlich, stellt der Knoten das Senden seiner Nachricht ein und wechselt in den Empfangszustand. Dies kann nur auftreten, falls der Signalpegel des Knotens rezessiv, der des

Busses jedoch dominant ist. Das bedeutet, dass eine Nachricht mit höherer Priorität gesendet werden soll. Da der Knoten mit der niederprioren Nachricht erkennt, dass er nicht senden kann, geht diese Nachricht nicht verloren. Stattdessen wird der Knoten zu einem späteren Zeitpunkt erneut versuchen, diese Nachricht zu senden. Dieses Verfahren wird *CSMA/CA (Carrier Sense Multiple Access/Collision Avoidance)* genannt.

Bei konstant hoher Buslast ist es möglich, dass niederpriore Nachrichten nie, oder zumindest nicht innerhalb einer bestimmten Zeitspanne, gesendet werden. Es ist daher Sorge zu tragen, dass genug Reserven vorhanden sind. Um dies sicher zu stellen, werden die Steuergeräte eines Kfz auf mehrere CAN-Busse verteilt. Ausgewählte Steuergeräte übernehmen dabei die Rolle von Gateways, indem sie zwischen den Bussen die Daten austauschen (vgl. Abbildung 2.5, S. 29).

Der CAN-Bus ist heutzutage als Standard-Bussystem in der Fahrzeugelektronik anzusehen. Aufgrund der stetigen Neu- und Weiterentwicklung der Kfz-Elektronik und dem damit verbundenen Anstieg des Datenvolumen, reicht eine Aufteilung der Steuergeräte auf mehrere CAN-Busse jedoch nicht mehr aus. Zukünftig werden daher schnellere Netzwerke notwendig.

FlexRay

Mit FlexRay wurde ein Hochgeschwindigkeitsbus entwickelt, welcher den zunehmenden Ansprüchen der Fahrzeugelektronik gerecht werden soll. So erfüllt FlexRay neben einer deutlich höheren Datenübertragungsrate gegenüber CAN (2x10 Mbit/s[1]) auch harte Echtzeitanforderungen [Joc07].

Einsatzgebiete für FlexRay sind hauptsächlich sicherheitskritische Bereiche mit großem Datenvolumen, die beispielsweise in der Fahrdynamik auftreten.

Nach eigenen Angaben des FlexRay Konsortium soll sich FlexRay zukünftig als Standard für High-Speed Netze durchsetzen [Fle].

[1] zwei Kanäle, die entweder redundant oder zur Erhöhung der Datenrate verwendet werden können.

Media Oriented Systems Transport (MOST)

Aus der Forderung große Datenmengen übertragen zu können, wie sie beispielsweise beim Einsatz von Audio- oder Videoanwendungen auftreten, wurde Ende der Neunziger Jahre MOST entwickelt. Vorrangies Einsatzziel sind Multimedia- und Telematikanwendungen. Auf Basis eines optischen Mediums ist eine Datenübertragungsrate von bis 24 Mbit/s möglich [WWP04].

MOST ist ein synchrones Netzwerk, meist in einer Ringtopologie realisiert. Ein Knoten übernimmt die Rolle des *Timing-Masters*, indem er kontinuierlich Synchronisationsframes sendet. Die Datenübertragung erfolgt über zwei variable Bereiche für synchrone und asynchrone Kommunikation. Zusätzlich existiert ein Kontrollkanal zur Ansteuerung der synchronen und asynchronen Kanäle. Der synchrone Kanal dient der Übertragung kontinuierlicher Daten (etwa bei Audio oder Video), wohingegen der asynchrone Kanal zur schnellen Übertragung spontaner Daten (z.B. Informationen eines Navigationssystems) vorgesehen ist.

Detailierte Informationen zu MOST können unter [Mos] gefunden werden.

Zusammenfassung/ Vergleich

Die Entwicklung schneller und zuverlässiger Busse ermöglicht die Umsetzung neuer, datenintensiver Systeme. Sie bedeutet jedoch nicht das Ende der herkömmlichen Technologien. Je nach Einsatzgebiet haben die „alten“ Vertreter Vorteile, welche deren Verwendung rechtfertigen. Tabelle 2.2 (S. 28) fasst die Eigenschaften der vorgestellten Netzwerke zusammen.

2.2.3 E/E-Architektur

In der Automobilelektronik bezeichnet eine E/E-Architektur die Struktur eines elektrisch/elektronischen Systems [Rei06a]. Diese Struktur legt die notwendigen Steuergeräte sowie deren Vernetzung untereinander fest. Ferner spiegelt sich der Zweck des Systems in der Architektur wider. Wesentliche Bestandteile einer Architektur sind daher

- die zu realisierenden Funktionen

	LIN	CAN	FlexRay	MOST
Charakterisierung	Low-Cost-Bereich	weiche Echtzeitanforderung	harte Echtzeitanforderung	Telematik und Multimedia
typische Anwendung	Anbindung intelligenter Sensoren und Aktoren	Gesamtfahrzeug	sicherheitskritische Anwendungen (teilweise Überschneidung mit CAN), zukünftige Anwendungen (X-by-Wire)	Video-, Audiostreams, GPS, Telefon, ...
max. Datenrate	20 kbit/s	1 Mbit/s	2x10 Mbit/s	24 Mbit/s
relative Kosten pro Knoten	0,5	1	2,5	5

Tabelle 2.2: Fahrzeug-Bussysteme im Vergleich

- die eingesetzte Technik
- die Topologie des Systems

Insbesondere die Topologie der E/E des Fahrzeugs ist für den Systemintegrationstest von Bedeutung. Aus ihr wird die Art der Vernetzung, sowie der Anschluss der Aktoren und Sensoren entnommen. Abbildung 2.5 (S. 29) zeigt eine Beispieltopologie, wie sie in einer Baureihe der Daimler AG verbaut sein könnte.

2.3 Systemintegrationstest im Automobil

Der Systemintegrationstest ist die Aufgabe des OEM[2]. Nur er hat Zugang zu allen Komponenten und ist somit überhaupt erst in der Lage diesen durch-

[2]OEM (Original Equipment Manufacturer) bezeichnet den Hersteller von Produkten, welcher sie allerdings nicht selbst in den Markt bringt. U.a. in der Automobilindustrie steht OEM allerdings für das genaue Gegenteil. Hier werden fertige Produkte kombiniert und unter eigenem Namen vertrieben.

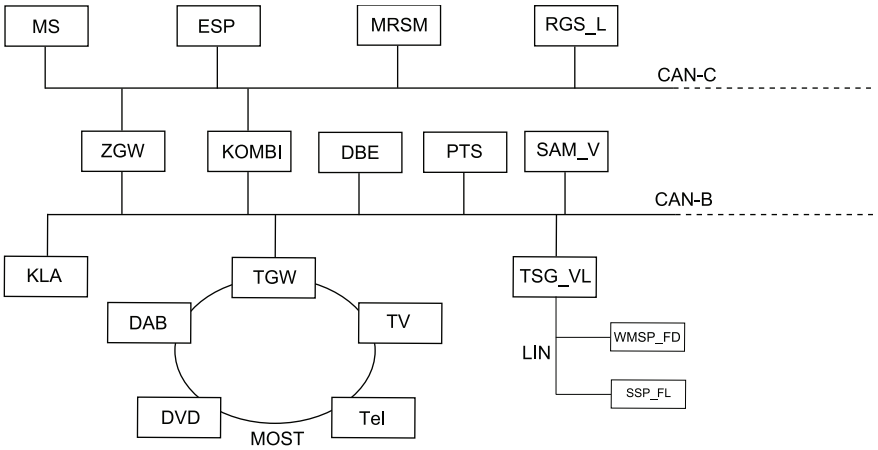

Abbildung 2.5: Beispieltopologie

zuführen. Je nach Projektphase und Zielsetzung kommen hierbei verschiedene Testsysteme zum Einsatz.

2.3.1 Klassifizierung von Testsystemen

Abhängig von der Entwicklungsphase haben Testsysteme unterschiedliche Ausprägungen. Diese wird zum einen durch die Form des Testobjekts, zum anderen durch die Art der Umgebung (Fahrzeug) beeinflusst. Beides kann entweder in realer Form vorliegen oder aber durch Modelle simuliert werden. Daraus resultieren vier Kategorien von Testsystemen (siehe Tabelle 2.3).

Umgebung / Steuergerät	simuliert	real
simuliert	Software-in-the-Loop Testsystem	Rapid Prototyping
real	Hardware-in-the-Loop Testsystem	Onboard Testsystem

Tabelle 2.3: Klassifizierung von Testsystemen nach [Har01]

Im Folgenden werden die Stärken und Schwächen der einzelnen Testsysteme charakterisiert. Vertiefende Beschreibungen können [Har01] entnommen werden.

Software-in-the-Loop (SIL)

Software-in-the-Loop wird hauptsächlich in frühen Entwicklungsphasen eingesetzt, in welchen lediglich der ausführbare Code, nicht aber die Hardware des Testobjekts zur Verfügung steht. Neben der Möglichkeit frühzeitig zu testen, kann mit SIL eine hohe Testtiefe erreicht werden, da der Algorithmus offen liegt, sowie alle internen Größen beobachtet werden können. Des Weiteren gestaltet sich der Aufwand für die Simulation der Umgebung relativ einfach. Der Nachteil dieses Verfahrens liegt in der niedrigen Realitätsnähe, da weder das Testobjekt noch die Umgebung in realer Form vorliegen.

Rapid Prototyping

Bei diesem Verfahren wird die Software mittels einer Prototypen-Hardware in eine reale Umgebung integriert. Da diese jedoch meist ebenfalls Gegenstand der Entwicklung ist, ist sie in frühen Projektphasen nicht immer verfügbar. Rapid Prototyping gewinnt gegebenüber Software-in-the-Loop auf Kosten höherer Komplexität an Realitätsnähe.

Hardware-in-the-Loop (HIL)

Existiert das Testobjekt bereits in realer Form, werden Hardware-in-the-Loop Testsysteme verwendet. Die Umgebung wird hingegen durch Modelle simuliert. Hierfür muss die digitale Rechnerwelt auf die Schnittstellen der Steuergeräte abgebildet werden. Dies ist mit einem großen Aufwand verbunden. HIL Testsysteme zeichnen sich durch eine Realitätsnähe aus, welche nur durch Onboard Testsysteme übertroffen wird. Hardware-in-the-Loop Testsysteme bilden die Grundlage für diese Arbeit, ihr genauer Aufbau wird daher in 2.3.2 (S. 31) beschrieben.

Onboard Testsystem

Mit der Existenz erster Prototypenfahrzeuge kommen Onboard Testsysteme zum Einsatz. Die Möglichkeiten sind gegenüber den vorherigen Verfahren, speziell HIL, eher begrenzt. Dies folgt zum einen aus der realen Umwelt, die nur eingeschränkt beeinflusst werden kann. Zum anderen gehen viele Stimulationen direkt vom Benutzer aus, wodurch Zeitvorgaben nicht immer eingehalten werden können. Onboard Testsysteme sind aber nicht zuletzt daher

unverzichtbar, da sie das Testobjekt in seiner endgültigen Form testen.

Im Sinne einer optimalen Qualitätssicherung sollte auf keines dieser Verfahren verzichtet werden. Vielmehr gilt es die jeweiligen Stärken zu kombinieren und damit einen, die gesamte Entwicklung begleitenden Testprozess zu installieren.

2.3.2 Hardware-in-the-Loop (HIL)

Beim Systemintegrationstest mittels Hardware-in-the-Loop wird das reale System (auch System under Test (SUT) genannt) in einer virtuellen Umgebung betrieben. Eine Verbindung der Ein- und Ausgänge des Systems mit den Aus- und Eingängen der Simulation stellt die notwendige Rückkopplung für den Betrieb des Systems sicher. Abbildung 2.6 (S. 31) zeigt den Aufbau eines Hardware-in-the-Loop Testsystems.

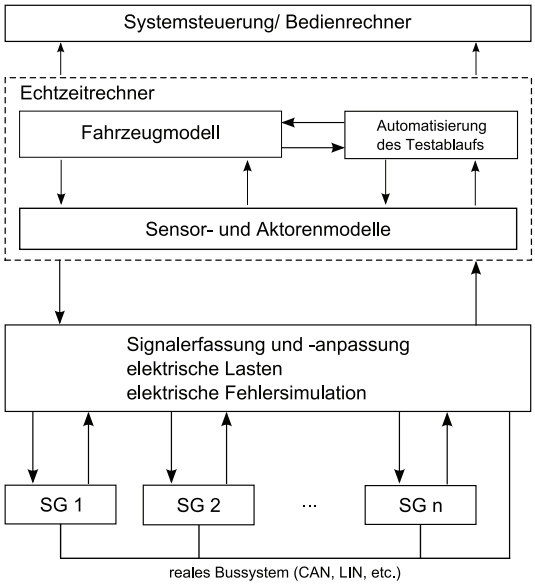

Abbildung 2.6: Aufbau eines HIL Testsystems [Sch03], [Hut06]

Kernstück dieser Testsysteme ist der Echtzeitrechner. Auf ihm befinden sich

die Modelle zur Simulation des technischen Umfelds des SUT. Zusätzlich werden hier die Modelle für die Sensoren und Aktoren realisiert, welche die Verbindung zwischen realem Steuergerät und dem nachgebildeten technischen Prozess herstellen [Hut06]. Die Berechnung der Modelle muss hierbei in Echtzeit erfolgen, da das SUT ansonsten in einen Notlauf wechseln würde und ein regulärer Test nicht mehr möglich wäre. Des Weiteren ermöglicht der Echtzeitrechner die Testautomatisierung und definiert die Schnittstelle zur Bedieneinheit.

Einen wesentlichen Bestandteil stellt natürlich auch das Testobjekt selbst dar. Es ist schließlich die Legitimation des Testsystems. Im vorliegenden Fall (vgl. Abbildung 2.6, S. 31) besteht es aus n Komponenten, die durch Vernetzung ein System bilden. Der logische Aufbau gleicht dabei der späteren Umsetzung im realen Fahrzeug. Zusätzlich werden alle Busse (CAN, Lin, etc.) sowie die restlichen Pins der Steuergeräte mit dem Echtzeitrechner verbunden. Zwischen realem System und dem Echtzeitrechner liegen dabei diverse Kopplungschichten.

Signalerfassung und -anpassung Die Aufgabe dieser Schicht besteht in der Anpassung der technischen Größen des Fahrzeugs auf die des Echtzeitrechners. Hierzu zählen beispielsweise Messverstärker sowie Schutzschaltungen [OS04].

Reale und simulierte elektrische Lasten Viele Steuergeräte kontrollieren anhand des fließenden Stroms die korrekte Funktion ihrer Aktoren. In diesem Fall müssen entweder die realen Bauteile oder Ersatzlasten, etwa in Form von Widerständen vorhanden sein.

Elektrische Fehlersimulation Moderne Steuergeräte besitzen meist Mechanismen, welche fehlerhafte Verbindungen mit anderen Steuergeräten oder Sensoren/Aktoren erkennen. Dazu zählen beispielsweise Leitungsunterbrechungen oder Kurzschlüsse. Dies hilft der Fachwerkstatt, im Störungsfall die Fehlerursache zu ermitteln. Zum Test dieser Funktion muss das Testsystem das Einspeisen derartiger Szenarien zulassen.

Zur Steuerung des Testsystems dient der sogenannte Bedienrechner. Mittels spezieller Software stellt er im Wesentlichen vier Funktionen zur Verfügung [Kus06].

- Laden von Simulationsmodellen
- Beobachten von Modellsignalen
- Manipulation von Modellsignalen
- Verwaltung von automatisierten Testabläufen

2.3.3 Real Time Automation Engine (RTAE)

Die RTAE bezeichnet eine Automatisierungs-Komponente, welche die Ausführung von Testprogrammen unter Echtzeit gestattet [Har01]. Sie wird synchron mit den Simulationsmodellen ausgeführt. Die RTAE stellt die gängigen Befehle anderer Programmiersprachen, wie etwa *if* oder *for* zur Verfügung. Zusätzlich bietet sie die Möglichkeit den Kontrollfluß in mehrere Zweige aufzuspalten, die fortan parallel ausgeführt werden. So kann beispielsweise die Reaktion mehrerer Signale auf eine Stimulation geprüft werden.

Eine komfortable Steuerung der RTAE ist durch die *Automation Library (AL)* realisiert worden. Mit Hilfe dieser Bibliothek können auf einfache Weise Testprogramme mit Echtzeitkomponenten erstellt und ausgeführt werden. Des Weiteren koordiniert sie den Austausch der Ergebnisse zwischen Echtzeit- und Bedienrechner, sowie deren Darstellung als Protokoll.

Eine ausführliche Beschreibung der *Real Time Automation Engine* sowie der *Automation Library* bietet [Kus06].

2.3.4 Organisation der Testdurchführung

Die Verteilung der Testaufgaben pro HIL Testsystem entfällt auf mehrere Tester. Jeder zeichnet sich dabei für mehrere Systeme verantwortlich. Daraus resultiert die Notwendigkeit, die verfügbare Testzeit auf die einzelnen Tester aufzuteilen. Priorisierungen auf einzelne Systeme bilden eher die Ausnahme, so dass eine gleichmäßige Vergabe der Testzeit auf jeden Tester als Standard anzusehen ist.

Grundsätzlich muss zwischen zwei Arten der Testsystemnutzung unterschieden werden. Zum einen gibt es Aufgaben, die nicht automatisiert durchgeführt werden können und der Tester somit vor Ort sein muss. Hierzu zählen die Inbetriebnahme neuer Testprogramme oder der Tausch von Steuergeräten. Diese Aufgaben werden während der normalen Arbeitszeit erledigt. Dem gegenüber steht der automatisierte Testlauf. Hierbei werden hauptsächlich nachts und an arbeitsfreien Tagen Testprogramme durchgeführt. Dabei erhält jeder Tester einen vordefinierten Zeitslot, welchen er mit seinen Testprogrammen belegt. Sobald er an der Reihe ist, werden diese Tests sequentiell durchgeführt. Nach Ablauf seiner Testzeit wird das laufende Programm abgebrochen und der nächste Tester ist an der Reihe.

Kapitel 3

Stand der Technik

Die automatische Parallelisierung sequentieller Aufgaben ist keine neue Idee. Bereits Ende der sechziger bzw. Anfang der siebtziger Jahre des letzten Jahrhunderts wurden Konzepte zur automatischen Parallelisierung sequentieller Programme entwickelt ([RG69a], [RG69b], [GR70]). Diese entstanden vor allem im Zuge der Verbreitung von Multiprozessor Computer Systemen. Dieses Kapitel gibt eine Übersicht über aktuelle Verfahren zur automatisierten Parallelisierung sequentieller Abläufe. Im Anschluss daran werden die Ursachen vorgestellt, weshalb sich diese Verfahren nicht zur Parallelisierung von Systemintegrationstests im E/E-Umfeld eignen.

3.1 Daten- und Kontrollflussanalysen

Dieses Verfahren konzentriert sich auf die Analyse von Datenabhängigkeiten innerhalb eines Programms sowie dessen Kontrollfluss. Es basiert dabei auf der Generierung und Auswertung unterschiedlicher Graphen, die den Input für den Parallelisierungsalgorithmus darstellen. Im Folgenden werden zwei Vertreter dieses Verfahrens vorgestellt.

Der erste Algorithmus wird durch [F+07] beschrieben. Ausgangspunkt hierfür ist ein beliebiges Programm, für welches ein modifizierter *System Dependence Graph (SGD)* erstellt wird. Darin werden die Abhängigkeiten der einzelnen Anweisungen untereinander dargestellt. Diese können entweder durch den Kontrollfluss oder den Datenfluss entstehen. Im nächsten Schritt wird der Graph in unabhängige Teilgraphen untergliedert, welche sich zur parallelen

Ausführung eignen. Dafür wird zunächst der Kontrollfluss analysiert und Anweisungen der gleichen Verzweigungsgbedingung werden in Gruppen zusammengefasst (vgl. Abbildung 3.1, S. 36). Diese sind, vom Kontrollfluss des Programms aus gesehen, unabhängig. Um eine parallele Ausführung zulassen zu können, müssen noch Datenabhängigkeiten ausgeschlossen werden. Hierfür werden wiederum einzelne Gruppen von Knoten gebildet, sogenannte *Cluster*. Ziel ist es, Abhängigkeiten auf ein Cluster zu beschränken. Die Datenabhängigkeit zweier Knoten kann aus dem SDG erkannt werden. Somit sind die Cluster untereinander bezüglich der Daten unabhängig. Eine Abhängigkeit im Kontrollfluss wurde durch den vorherigen Schritt bereits ausgeschlossen.

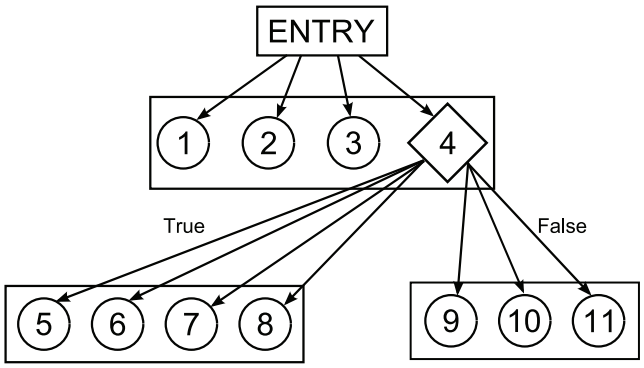

Abbildung 3.1: Gruppierung der Anweisungen anhand des Kontrollflusses

Aus diesen Erkenntnissen lässt sich wiederum ein Graph generieren. Die Knoten repräsentieren hierbei einen Block von Anweisungen. Abhängigkeiten werden durch Kanten zwischen zwei Knoten ausgedrückt. Somit lassen sich alle Bereiche, welche parallel ausgeführt werden dürfen, erkennen. Im abschließenden Schritt kann nun automatisiert neuer Code erzeugt werden, welcher die mögliche Parallelität ausnutzt.

Ein ähnlicher Ansatz kann auch bei [Eck86] gefunden werden. Hierbei werden allerdings im Unterschied zu [F+07] Schleifen als ein Block behandelt. Es ist daher nicht möglich, einzelne Durchläufe der Schleife zu parallelisieren.

Das zweite Verfahren nach [KK90] legt den Schwerpunkt auf die Erstellung maximaler paralleler Abläufe. Als Basis hierfür dient ein System $C = (J, <)$. J ist die Menge aller Anweisungen des Programms, die nicht weiter verfeinert werden können. Ein Element aus J wird demnach nur die Ein- und Ausgabedaten, sowie dessen Ausführungsdauer charakterisiert. Durch $<$ wird eine irreflexive[1], antisymmetrische[2] und transitive[3] Relation auf $J \times J$ definiert. Dabei bedeutet $T_i < T_j, \quad T_i, T_j \in J$, dass T_i beendet sein muss, bevor T_j gestartet werden kann. Die Definition von C bzw. von J und $<$ muss vom Anwender erstellt werden und kann nicht automatisiert erfasst werden.

Ausgehend von C werden Abhängigkeiten zwischen den Elementen aus J beschrieben. Zwei Elemente T_i, T_j beeinträchtigen sich nicht gegenseitig, wenn eine der folgenden Bedingungen erfüllt ist. D_T bezeichnet dabei den Speicherbereich auf den Element T lesend zugreift. Analog steht R_T für den Speicherbereich mit schreibenden Zugriff. Diese Bereiche müssen nicht disjunkt sein.

1. $T_i < T_j$
2. $T_j < T_i$
3. $(R_{T_i} \cap R_{T_j}) = (R_{T_i} \cap D_{T_j}) = (D_{T_i} \cap R_{T_j}) = \emptyset$

Ein System maximaler paralleler Abläufe wird dann dadurch definiert, dass durch das Entfernen der Kante (T_i, T_j) sich die Elemente T_i und T_j gegenseitig beeinträchtigen. Oder anders ausgedrückt:

$$(R_{T_i} \cap R_{T_j}) \cup (R_{T_i} \cap D_{T_j}) \cup (D_{T_i} \cap R_{T_j}) \neq \emptyset \tag{3.1}$$

Nach 3.1 können also Paare (T_i, T_j) bestimmt werden, deren Ausführung sequentiell durchgeführt werden muss. Alle anderen Paare (T_i, T_j) aus $<$ beeinträchtigen sich nicht gegenseitig und können somit parallel asugeführt werden.

---

[1] $R \subseteq M \times M : \forall x \in M : \neg xRx$
[2] $R \subseteq M \times M : \forall x, y \in M : xRy \wedge yRx \Rightarrow x = y$
[3] $R \subseteq M \times M : \forall x, y, z \in M : xRy \wedge yRz \Rightarrow xRz$

Es existieren noch zahlreiche weitere Verfahren, um sequentiellen Code zu parallelisieren (beispielsweise [Sch99], [Sch04], [C+05]). Sie unterscheiden sich im Grundsatz jedoch nicht wesentlich von den vorgestellten Methoden. So wird etwa bei [Sch04] ein zusätzlicher Kontrollmechanismus eingführt, welcher zur Laufzeit den Zugriff auf den Speicher überwacht. Sollten hierbei Störungen auftreten, können die entsprechenden Programmteile erneut parallel oder auch sequentiell gestartet werden.

3.2 Verwendung von Mustern

Ein anderer Ansatz zur Parallelisierung von sequentiellen Code stammt von [Keß94] und wurde für die Anwendungen der Numerik entwickelt. Dabei wird ausgenutzt, dass es viele Standardroutinen gibt, welche häufig verwendet werden und sich gut parallelisieren lassen. Hierzu zählen beispielsweise Matrixoperationen. Hierbei wird nicht versucht den existierenden Code zu parallelisieren, sondern vielmehr bestimmte Muster (z.B. Funktionsaufruf) zu erkennen und diese durch parallel ausführbaren Code zu ersetzen. Dieser Code ist in einer Bibliothek hinterlegt, welche auch das Aussehen der Muster definiert.

Das Verfahren von [Keß94] benötigt keine Analyse auf Abhängigkeiten innerhalb des Programms. Die Möglichkeiten der Parallelisierung beschränkt sich somit auf die in der Bibliothek definierten Standardfunktionen. Da diese jedoch nicht automatisiert sondern manuell erstellt werden, ist die Effizienz der Implementierung bezüglich der Parallelität maximal.

3.3 Abgrenzung

Die vorgestellten Ansätze bieten alle die Möglichkeit Teile eines sequentiellen Programms zu parallelisieren. Von entscheidender Rolle ist jeweils die vollständige Kenntnis des Quellcodes. Aus diesem wurden im ersten Verfahren die benötigten Ressourcen (Speicherbereiche) abgeleitet und existierende Abhängigkeiten erkannt. Das zweite Verfahren nutzt die Möglichkeit, auf parallel ausführbaren Code zurückgreifen zu können, um somit definierte Muster zu ersetzen.

Aufgrund der notwendigen Kenntnis des Programmcodes sind diese Methoden für die Parallelisierung von Systemintegrationstests jedoch ungeeignet. Hierbei müssen Funktionen der Steuergeräte parallelisiert werden, deren Quellcode nicht bekannt ist. Somit können keine Abhängigkeiten im Kontroll- oder Datenfluss erkannt werden. Eine Ersetzung von Mustern im Quellcode der Steuergeräte ist ebensowenig möglich. Eine Anwendung der Verfahren auf den bekannten Quellcode der Testprogramme löst das Problem auch nicht, da das Testprogramm lediglich Eingangsdaten für die Funktionen der Steuergeräte generieren. Die Schwierigkeit besteht nun darin, aus den Anweisungen des Testprogramms auf die Funktionalität des betroffenen Steuergerätecodes zu schliessen. Es müssen daher geeignete Methoden entwickelt werden, welche ausgehend von einem Testprogramm Rückschlüsse auf die benötigten Ressourcen zulassen. Derartige Methoden sind aktuell nicht verfügbar.

Kapitel 4

Fahrzeugfunktionen

In diesem Kapitel werden die Funktionen von Fahrzeugen genauer betrachtet. Zu Beginn wird die Zerlegung von Funktion in Teilfunktionen und die daraus resultierende Funktionshierarchie beschrieben. Anschließend wird die Form der Dokumentation und der Aufbau einer Funktionsbeschreibung diskutiert. Das Thema Funktionstest bildet das Ende dieses Kapitels. Darin findet sich eine Beschreibung aktueller Testprogramme, Erläuterungen zur eingesetzten Technik und Forderungen an zukünftige Testprogramme.

4.1 Funktionshierarchie

Die Spezifikation von Funktionen beginnt mit einer groben Beschreibung auf abstrahierten Niveau. Durch sukzessives Zerlegen in Teilfunktionen werden der Spezifikation weitere Details der Teilfuktionen hinzugefügt. Auf diese Weise entsteht eine Hierarchie, deren Basis hardwarenahe Treiber darstellen, die den Zugriff auf die Schnittstellen des Steuergeräts regeln [Har01]. Abbildung 4.1 (S. 42) zeigt exemplarisch den Aufbau einer Funktionshierarchie.

Wird eine Funktion durch mehr als ein Steuergerät umgesetzt, so wird der Informationsaustausch zwischen den Steuergeräten durch Bussysteme sichergestellt (vgl. Abschnitt 2.2.2, S. 24). Eine Kommunikationsmatrix (K-Matrix) enthält dabei alle Informationen, die für einen korrekten Nachrichtenaustausch notwendig sind [SZ06]. Dazu zählen unter anderem

- die ID

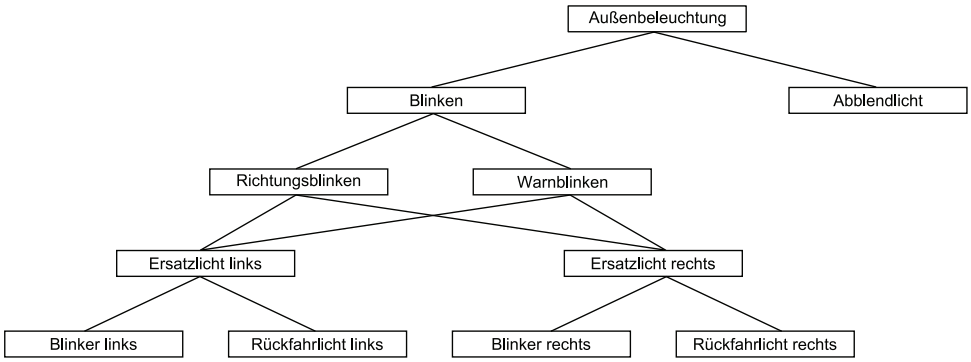

Abbildung 4.1: Funktionshierarchie nach [Har01]

- der Sender
- der Empfänger
- die Sendeart (zyklisch, sofort,...)
- der Inhalt

einer Nachricht. Die Kommunikationsmatrix ist somit ein fester Bestandteil der Funktionssoftware. Änderungen an einer Funktion können somit im schlimmsten Fall die Anpassung weiterer Steuergeräte notwendig machen.

Vor der Realisierung einer Funktion steht jedoch deren Spezifikation. Die daraus resultierenden Anforderungen werden im sogenannten Lastenheft festgehalten. Dabei existieren je nach Verwendungszweck unterschiedliche Varianten. Der folgende Abschnitt gibt einen Überblick über die gängigen Vertreter.

4.2 Lastenhefte

Gemäß [Deu97] beschreibt das Lastenheft die „vom Auftraggeber festgelegte Gesamtheit der Forderungen an die Lieferungen und Leistungen eines Auftragnehmers innerhalb eines Auftrages“. Dabei wird in der Regel das *was* und *wofür* beschrieben. Obwohl diese Anforderungen überprüfbar sein müssen, werden Lastenhefte meist in informeller Art verfasst, wodurch der Raum für

unterschiedliche Interpretationen entsteht [Sch03]. Dennoch bilden Lastenhefte die Grundlage für die Testfallerstellung. Aus dem Lastenheft resultiert das Pflichtenheft, in welchem das *wie* und *womit* festgelegt ist.

Lastenhefte lassen sich anhand ihrer Intention weiter untergliedern. Die Unterschiede bestehen dabei hauptsächlich im Detail der Informationen sowie in der Sichtweise auf diese.

Rahmenlastenheft

Das Rahmenlastenheft beschreibt die wichtigsten Merkmale einer Baureihe. Systeme und deren Funktionen werden ebenso wie die Topologie nur abstrakt beschrieben. Das Rahmenlastenheft bildet die Grundlage für die System- und Komponentenlastenhefte.

Systemlastenheft

Das Systemlastenheft spezifiziert alle Anforderungen an ein System. In erster Linie werden hierbei ausführliche Beschreibungen der Funktionen formuliert. Des Weiteren werden die beteiligten Steuergeräte sowie deren Anteil an der Gesamtfunktionalität festgelegt. Die Gesamtheit aller Systemlastenhefte bietet einen Zugang zum Fahrzeug aus Systemsicht.

Komponentenlastenheft

Da sich das Systemlastenheft nicht für die Vergabe einzelner Steuergeräte an den Zulieferer eignet, werden die Anforderungen an ein Steuergerät in einem Komponentenlastenheft formuliert. Darin ist für jede Komponente festgelegt, an welcher Funktion sie wie beteiligt ist. Sie bieten somit einen Zugang zum Fahrzeug aus Sicht der Komponenten. Das Komponentenlastenheft wird komplett an den Zulieferer zur Entwicklung des Steuergeräts übergeben.

Vertiefende Informationen sowie weitere Ausprägungen von Lastenheften können [Har01] entnommen werden.

4.3 Funktionsbeschreibung

Wesentlicher Bestandteil eines Systemlastenhefts ist die Funktionsbeschreibung. Sie beinhaltet detaillierte Informationen über das Verhalten der Funktionen und damit auch über das Verhalten des Systems. Auch wenn die Beschreibung zu meist in natürlicher Sprache verfasst ist, lassen sich in der Regel die folgenden Abschnitte zu jeder Funktion erkennen. In diesen werden die Rahmenbedingungen, unter welchen die Funktion ausgeführt werden kann sowie deren Funktionsweise beschrieben. Diese Bedingungen bestehen nur in Ausnahmefällen aus einer einzelnen Vorgabe. Meist werden die einzelnen Bestandteile mittels UND oder ODER zu einer Bedingung zusammengefasst.

4.3.1 Funktionsvoraussetzung

Die Funktionsvoraussetzung beschreibt eine Bedingung, die sowohl vor, als auch während der gesamten Ausführung wahr sein muss. Ist sie nicht mehr gegeben, so wird die Ausführung unter- oder abgebrochen. Abgebrochene Funktionen werden im Gegensatz zu den unterbrochenen nicht wieder fortgesetzt, nachdem die Funktionsvoraussetzung erneut gegeben ist. Beispiele für Funktionsvoraussetzungen sind etwa „Zündung AN“ oder ein verriegeltes Fahrzeug.

4.3.2 Auslöseereignis

Das Auslöseereignis startet die Funktionsausführung, sofern die Funktionsvoraussetzung gegeben ist. Eine Funktion kann entweder durch permanentes Anliegen (z.B. Schalter) oder durch einmaliges Eintreten (z.B. Taster) ausgelöst werden. Auch eine Sequenz mehrerer Einzelaktionen kann für den Start einer Ausführung notwendig sein. In diesem Fall wird die gesamte Sequenz als Auslöseereignis angesehen. Das Auslöseereignis muss nicht in allen Fällen durch den Fahrer herbeigeführt werden. So ist ein Auslöser für die Regenschließung[1] beispielsweise einsetzender Regen.

[1] bei abgestelltem Fahrzeug und geöffnetem Schiebe-Hebe-Dach wird das Dach unter anderem bei Regen in die Hubstellung gefahren, um den Innenraum des Fahrzeugs zu schützen [MB07a]

4.3.3 Funktionsverlauf

Im Funktionsverlauf ist das Verhalten der Funktion nach Eintreten des Auslöseereignisses definiert. Das beinhaltet neben der unmittelbaren Reaktion auf den Auslöser, etwa das Ansteuern eines Motors, auch den weiteren Verlauf sowie ein eventuelles Ende nach Ablauf eines Timers.

4.3.4 Abbruchbedingung

Die Abbruchbedingung bezeichnet alle Umstände, welche eine laufende Funktion ab- oder unterbrechen. Gelten diese Umstände jedoch bereits vor dem Beginn der Ausführung, so bleibt die Funktion hiervon unberührt. Durch diese Eigenschaft lässt sich die Abbruchbedingung eindeutig von der Funktionsvoraussetzung unterscheiden. Beispiel für eine Abbruchbedingung der Funktion „Intervallwischen“ ist das Öffnen der Fahrertür. Bei aktiver Funktion (Lenkstockschalter auf Intervall) bricht das Öffnen der Tür die Funktion ab [MB07a]. Es gibt keine weiteren Wischzyklen. Wird jedoch bei offener Tür das Intervallwischen aktiviert, so werden die Wischzyklen wie bei geschlossener Fahrertür durchgeführt.

4.3.5 Beispiel: „Intervallwischen“

Mit einem Beispiel anhand des Teilsystems *Intervallwischen* soll demonstriert werden, wie aus einer Funktionsbeschreibung Testfälle ermittelt werden. Die Umsetzung in Testprogramme wird im anschliessenden Kapitel „Anforderungen an ein Testprogramm“ demonstriert. Bei der folgenden Beschreibung handelt es sich um eine vereinfachte Version der Funktionsweise, wie sie aktuell in den Fahrzeugen der Mercedes Car Group umgesetzt ist. Für die reale Testfallermittlung müsste eine weitere Unterteilung des Teilsystems *Intervallwischen* in etwa „Intervallwischen aktivieren“, „Intervallwischen mit Regensensor“, etc. erstellt werden. In diesem Beispiel wird *Intervallwischen* jedoch der Einfachheit halber bereits als Funktion interpretiert.

Die Funktion „Intervallwischen“ wird über die Betätigung des Lenkstockschalters aktiviert. Steht der Zündschlüssel auf Position 3 (entspricht „Zündung AN“) wird der Wischermotor angesteuert und der Wischerarm bewegt sich. Bei verbautem Regensensor werden die Wischzyklen dem Niederschlag angepasst. Ist kein Regensensor vorhanden erfolgen die Zyklen in festen

Abständen. Um aussteigende Personen vor Spritzwasser zu schützen, werden die Wischbewegungen nach Öffnen einer Vordertür eingestellt.

Tabelle 4.1 (S. 46) zeigt die Unterteilung der Beschreibung in die einzelnen Funktionsabschnitte.

Abschnitt	**Bedingung**
Funktionsvoraussetzung	Schlüssel auf Position 3
Auslöseereignis	Lenkstockschalter auf Intervall
Funktionsverlauf	1. Wischermotor wird angesteuert UND Wischerarm bewegt sich 2. Regensensor steuert die Wischzyklen ODER Wischzyklen erfolgen in festen Abständen
Abbruchbedingung	Fahrertür öffnen ODER Beifahrertür öffnen

Tabelle 4.1: Funktionsabschnitte

Aus einer derartigen Einteilung können im nächsten Schritt die passenden Testfälle erarbeitet und daraus Testprogramme abgeleitet werden. Wie diese Testprogramme aussehen und welche Technik hierfür verwendet wird beschreiben die folgenden Abschnitte.

4.4 Funktionstest

Bei den für diese Arbeit relevanten Testskripten handelt es sich um imperative Programme. Sie werden in einem Visual Basic Dialekt, unter Verwendung eines COM-Objekts für die Generierung des Echtzeitcodes, geschrieben [Kus06]. Somit eignen sich die Programme für den Test des Systems unter Echtzeit.
Die Echtzeitkomponente ermöglicht auch die Parallelisierung einzelner Testsequenzen. Dies wäre mit den Mitteln, welche Visual Basic zur Verfügung

stellt, nicht machbar. Daraus folgt unmittelbar, dass nur Anweisungen, welche sich der Funktionalität der Echtzeitkomponente bedienen, parallel ausführbar sind. Derartige Anweisungen werden im weiteren Verlauf Echtzeitanweisungen bzw. Anweisungen mit Echtzeitanforderung genannt.
Da die Parallelisierung von Testsequenzen notwendig für die parallele Testausführung ist, wird eine bevorzugte Verwendung von Echtzeitanweisungen gefordert. In Ausnahmefällen kann von dieser Forderung abgesehen werden und auf die Möglichkeit der Parallelisierung verzichtet werden. So wird beispielsweise eine parallele Kodierung der Steuergeräte (vgl. 6.3, S. 76) nicht unterstützt. Somit ist eine entsprechende Programmierung auch nicht sinnvoll.

4.4.1 Analyse aktueller Testprogramme

Bei der Analyse von Testprogrammen muss zwischen inhaltlichen und formalen Eigenschaften unterschieden werden. Der Testinhalt wird hierbei durch die Funktionsbeschreibung bzw. durch das Lastenheft vorgegeben. Interessant an dieser Stelle ist jedoch die formale Umsetzung dieser Inhalte. Die aktuellen Testprogramme können durch folgende BNF Notation beschrieben werden (vgl. [Kus06]).

Testprogram	=	{Block};
Block	=	Sequence | Concurrency | Statement;
Sequence	=	'BeginSequence' {Block} 'EndSequence';
Concurreny	=	'BeginConcurrency' {Block} 'EndConcurrency';
Statement	=	Statement_PC | Statement_RT;

Folgendes Programm 4.1 (S. 48) zeigt ein vereinfachtes Beispiel für ein aktuelles Testprogramm. Es beinhaltet lediglich die für diese Arbeit relevanten Aspekte. Dadurch entfallen u.a. Anweisungen zum Aufzeichnen von Messwerten oder Angaben zur Organisation der Testdurchführung (etwa das Setzen von Attributen).
Erläuterungen zu Programm 4.1
Ziel dieses Programms ist der Test der Intervallwischfunktion. Hierfür werden zu Beginn die notwendigen Voraussetzungen geschaffen, indem zuerst die Stromversorgung der Steuergeräte global eingeschaltet wird (Zeile 1). Im

```
 1   Assign Batterie = 1
 2   Assign Zuendschluessel = 3
 3
 4   Assign Beifahrertuer = 0
 5   Assign Fahrertuer = 0
 6
 7   Assign Lenkstockschalter = 1
 8
 9   BeginConcurrency
10      VerifyTimeout Wischer.inBewegung == 1, 1
11      VerifyTimeout Wischer.Geschwindigkeit == 7, 0.5
12   EndConcurrency
13
14   Sleep 1
15
16   Assign Lenkstockschalter = 0
17
18   Assign Zuendschluessel = 0
19   Assign Batterie = 0
```

Programm 4.1: vereinfachtes Beispiel für ein aktuelles Testprogramm der Wischfunktion

nächsten Schritt (Zeile 2) wird der Zündschlüssel auf die Position 3 gestellt (entspricht „Zündung AN“). Zudem werden Fahrer- und Beifahrertür geschlossen (Zeilen 4 und 5). Inhaltlich ist hiermit die Funktionsvoraussetzung komplett. Aus dem Testprogramm ist dies aber nicht ablesbar. Die nächste Anweisung (7) repräsentiert das Auslöseereignis durch Aktivierung der Intervallwischfunktion. Diese wird im Folgenden anhand eines Statussignals (10) und der auf dem Bus angeforderten Wischgeschwindigkeit (11) kontrolliert. Der in Zeile 9 eingeleitete Block bewirkt hierbei, dass die beiden Kontrollen parallel ausgeführt werden. Durch die Anweisung *Sleep* wird lediglich eine Sekunde gewartet, bevor die Funktion deaktiviert wird (16). Am Ende des Programms wird noch der Zündschlüssel abgezogen (18) und die Stromversorgung ausgeschaltet (19).

Neben dem Programmcode können die Testprogramme beliebige Attribute besitzen. Diese werden in der Regel zur Organisation der Testdurchführung verwendet. Mit Ausnahme der folgenden sind sie jedoch für diese Arbeit ohne Bedeutung.

Getestete Steuergeräte dieses Attribut gibt an, welche Steuergeräte an der getesteten Funktion unmittelbar beteiligt sind. Zusätzlich kann die Version der Steuergeräte (bestehend aus Hard- und Softwarestand) bei der letzten Testausführung bestimmt werden. Aufgrund dieses Attributs können pro Release Auswertungen erstellt werden, welche eine Übersicht über die durchgeführten Umfänge geben. Nicht zuletzt kann hierdurch sichergestellt werden, dass kein Test vergessen wird.

Ausführungsdauer über dieses Attribut kann die Ausführungsdauer der einzelen, bereits durchgeführten Testläufe ausgelesen werden. Dies dient in Kapitel 4.1 als Grundlage zur zeitlichen Optimierung der Testsuite

Prozessorauslastung dieses Attribut enthält die maximale Auslastung des Echtzeitrechners während der Testausführung. Dies geht als Rahmenbedingung in die Generierung der Testsuite ein, da die durch die Programme verursachte Auslastung nicht die maximale Leistung des Echtzeitrechners übersteigen darf.

Der in Programm 4.1 beschriebene Aufbau in Verbindung mit Verwendung von Attributen genügt den aktuellen Anforderungen an ein Testprogramm.

Für den Vergleich zweier Testprogramme sind jedoch zusätzliche Informationen nötig. So ist neben dem jeweiligen *Statement* auch der Kontext (Testabschnitt aus 4.3) wichtig, in welchem es auftritt. Diese Zuordnung kann anhand aktueller Testprogramme jedoch nicht getroffen werden.

Eine weitere Einschränkung ergibt sich aus der Vermischung von Anweisungen mit Echtzeitanforderung und solchen ohne Echtzeitanforderung. Aufgrund der eingesetzten Technik ist ein Parallelisierung dieser Konstellation nicht möglich.

Aus diesen Gründen ist eine parallele Ausführung aktueller Testprogramme nicht sinnvoll. Für eine Parallelisierung ist demnach zuerst eine Änderung im Aufbau der Testprogramme notwendig. Im Folgenden werden die neuen Anforderungen konkretisiert und die Unterschiede anhand des Beispiels für den Test der Wischfunktion aufgezeigt.

4.4.2 Anforderungen an künftige Testprogramme

Aus den oben beschriebenen Einschränkungen folgen unmittelbar zwei Forderungen an die Teststruktur. Zum einen basiert Parallelität auf der Echtzeiteigenschaft der Anweisungen. Testschritte, welche nicht unter Echtzeitbedingungen ausgeführt werden, lassen sich daher auch nicht parallelisieren. Echtzeitcode ist somit Anweisungen ohne Echtzeiteigenschaft vorzuziehen. Eine Mischung beider Anweisungen ist nicht mehr zulässig. Zum anderen muss jede Aktion eines Testprogramms einem Testabschnitt aus 4.3 zugeordnet werden können. Hierfür bietet sich eine formale Aufteilung des Testprogramms in drei Abschnitte an: die Initialisierung, die Durchführung und der Abschluss. Neue Testprogramme lassen sich daher wie folgt beschreiben.

```
Testprogram  =  {Statement_PC} 'Init' {Block_RT} 'Run' {Block_RT}
                'Term' {Block_RT} {Statement_PC};
Block_RT     =  Sequence | Concurrency | Statement_RT;
Sequence     =  'BeginSequence' {Block_RT} 'EndSequence';
Concurreny   =  'BeginConcurrency' {Block_RT} 'EndConcurrency';
```

Programm 4.2 zeigt, wie sich die Änderungen auf das gerade vorgestellte Beispiel auswirken. Der Großteil des Codes bleibt hiervon unberührt. Lediglich

die Schlüsselworte *Init* (1), *Run* (6) und *Term* (13) wurden ergänzt. Durch sie wird der Beginn des Abschnitts definiert. Beendet wird er durch den Beginn des Nachfolgers bzw. durch das Ende des Testprogramms.

Hinweis: Der Fokus dieser Beispielprogramme liegt auf der Kennzeichnung der Initialisierung, der Durchführung und des Abschlusses, sowie der Zuordnung von Anweisungen zu einem dieser Abschnitte. Die Forderung nach Verwendung von Echtzeitanweisungen ist ohne größere Schwierigkeiten erfüllbar (etwa durch eines Vorgabe in den Programierrichtlinien) und wird hier daher nicht näher betrachtet.

```
 1   Init
 2      Assign Batterie = 1
 3      Assign Zuendschluessel = 3
 4      Assign Beifahrertuer = 0
 5      Assign Fahrertuer = 0
 6   Run
 7      Assign Lenkstockschalter = 1
 8      BeginConcurrency
 9         VerifyTimeout Wischer.inBewegung == 1, 1
10         VerifyTimeout Wischer.Geschwindigkeit == 7, 0.5
11      EndConcurrency
12      Sleep 1
13   Term
14      Assign Lenkstockschalter = 0
15      Assign Zuendschluessel = 0
16      Assign Batterie = 0
```

Programm 4.2: Umsetzung der neuen Anforderungen an ein Testprogramm

Die Initialisierung

Die Ergebnisse automatisierter Testprogramme zeichnen sich unter anderem dadurch aus, dass sie unter Verwendung der gleichen Testumgebung jederzeit reproduzierbar sind. Damit dies möglich ist, muss zu Beginn jedes Tests ein

wohl definierter Zustand hergestellt werden. Dazu gehören auch die für die Funktion spezifischen Vorbedingungen - die Funktionsvoraussetzung. Ausgehend von diesem Startszenario wird in der Durchführung der Testablauf gestartet und die einzelnen Prüfschritte durchgeführt. Die Besonderheit der Initialisierung besteht in der Unabhängigkeit von der Zeit. Dies betrifft sowohl die Schritte innerhalb der Initialisierung, als auch den Übergang zur Durchführung. Dieser Umstand ermöglicht das Eingreifen in den Testablauf durch eine übergeordnete Instanz, welche einzelne Testschritte synchronisieren kann. Dies ist im weiteren Verlauf der Arbeit entscheidend für den Vergleich zweier Testprogramme (siehe Kapitel 6).

Die Durchführung

Die Durchführung dient dem Test eines oder mehrerer im Lastenheft definierten Requirements. Ausgehend vom Szenario aus der Initialisierung werden Stimuli ausgelöst und das Verhalten des Systems mit den Sollwerten verglichen. Im Gegensatz zur Initialisierung gilt in der Durchführung eine Echtzeitanforderung. Aufgrund dieser zeitlichen Abhängigkeit von Stimulus und Reaktion ist eine Synchronisation analog zur Initialisierung nicht möglich.
In der Durchführung ist sowohl das Auslöseereignis, als auch der Funktionsverlauf enthalten. Obwohl sich die Abbruchbedingung unmittelbar auf die Durchführung auswirkt, kann sie in aller Regel nicht aus ihr gewonnen werden. Dies resultiert daraus, dass die notwendigen Informationen im Testprogramm nicht enthalten sind. Diese werden stattdessen in Form von Attributen bereitgestellt. Genaueres hierzu folgt in Kapitel 6.

Der Abschluss

Der Abschluss bietet dem Testprogramm die Möglichkeit, das System wieder in einen definierten Zustand zu überführen. Er hat dabei keinerlei Auswirkungen auf das Testergebnis und beeinflusst auch nicht eine parallele Ausführung mehrerer Testprogramme. Der Abschluss unterliegt keinen weiteren Anforderungen und ist optional. Aufgrund dieser Eigenschaften spielt er keine Rolle bei der Konflikterkennung.

Attribute

Bereits bei der Beschreibung der aktuellen Testprogramme wurde die Bedeutung von Attributen erwähnt. Für den Vergleich von Testprogrammen sind zwei Attribute zu berücksichtigen. Bei dem Ersten handelt es sich um die *getesteten Steuergeräte*. Zusätzlich zu der auf Seite 49 beschriebenen Funktion wird dieses Attribut auch beim Vergleich von Testprogrammen verwendet. Die genaue Verwendung wird im Kapitel 6 im Zusammenhang mit der Steuergerätekodierung gezeigt.
Mit Hilfe des zweiten Attributs wird die Abbruchbedingung erfasst. Hierfür wird für jedes Testprogramm die *getestete Funktion* angegeben. Wie anhand dieser die Abbruchbedingung berücksichtigt werden kann, wird in Abschnitt 6.2.1 (S. 65) beschrieben.

Kapitel 5

Ablauf der parallelen Testausführung

Bevor mehrere Testprogramme parallel ausgeführt werden können, muss eine Reihe an vorbereitenden Schritten durchgeführt werden. Durch diese wird eine reibungslose Ausführung der Testprogramme gewährleistet. Abhängig von der Anzahl auszuführender Testprogramme kann die Vorbereitungszeit nicht vernachlässigt werden. Da für diese jedoch kein Zugriff auf das Testsystem notwendig ist, geht sie nicht auf Kosten der Zeitersparnis, welche durch die Parallelität der Testprogramme gewonnen wird.

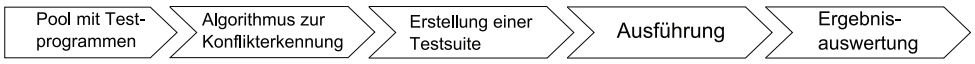

Abbildung 5.1: Schritte der parallelen Testausführung

Die Vorbereitung besteht im Wesentlichen aus drei Schritten (vgl. Abbildung 5.1). Im ersten Schritt muss eine Menge an Testprogrammen bestimmt werden, welche zur Ausführung gebracht werden sollen. Diese wird im weiteren Verlauf als *Testpool* bezeichnet. Aus diesem sollen nun möglichst viele Testprogramme innerhalb einer vorgegebenen Zeitspanne (etwa über Nacht) ausgeführt werden. Nachdem der Testpool definiert ist, werden im nächsten Schritt jeweils zwei Testprogramme auf gegenseitige Wechselwirkungen untersucht. Nach Betrachtung aller möglichen Kombinationen werden die Testprogramme gruppiert und zur Ausführung gebracht. Die Zeitersparnis hängt dabei maßgeblich von der Gruppierung ab, so dass der Einsatz von Optimie-

rungsverfahren empfehlenswert ist. Die Optimierung ist somit Bestandteil dieser Phase. Auf die Ausführung folgt im letzten Schritt schließlich die Auswertung der Ergebnisprotokolle. Anhand dieser muss im Fall einer Soll-Ist Abweichung die Fehlerursache lokalisiert werden.

5.1 Testpool

Eine automatisierte Testdurchführung beginnt mit der Auswahl der Testprogramme. Aus diesen setzt sich der Testpool zusammen. Bevor die Testprogramme bestimmt werden, wird typischerweise die Dauer des automatischen Testlaufs festgelegt. So wird etwa der Testpool für einen Nachttest[1] deutlich kleiner sein, als derjenige, der Testprogramme für ein ganzes Wochenende enthält. Die Auswahl der Testprogramme wird vom Tester, gegebenenfalls nach Vorgabe der Projektleitung, getroffen. Da in der Regel nicht alle auszuführenden Testprogramme am Stück laufen können, stellt ein Testpool ein Art Priorisierung dar. Dieser Umstand ist deshalb von Bedeutung, da innerhalb des Testpools die Priorisierung einzelner Testprogramme nur bedingt machbar ist. Sollte es notwendig sein, dass bestimmte Funktionen vorrangig getestet werden müssen, so ist eine Unterteilung des Testpools in disjunkte Teilmengen notwendig. In dieser befinden sich dann jeweils Testprogramme gleicher Priorität, so dass über die Priorisierung der Teilmengen der Testumfang gesteuert werden kann. Dies gilt sowohl für die sequentielle, als auch für die parallele Ausführung der Testprogramme. Folgendes Beispiel verdeutlicht die beschriebene Vorgehensweise zur Priorisierung.

Testprogramm	Tester	Priorität	Testpool
WischerAktivieren	X	2	B
FensterOeffnen	X	1	A
BlinkerLinks	Y	2	B
SitzheizungAktivieren	Y	1	A

Ohne Unterteilung der vier Testprogramme in zwei Testpools wäre es möglich, dass der hoch priorisierte Test „SitzheizungAktivieren“ nicht mehr zur Aus-

[1] Nachttest steht allgemein als Synonym für die Testprogramme, welche ohne Anwesenheit des Testers (in der Regel über Nacht) ausgeführt werden

führung kommt, obwohl bereits weniger dringende Testprogramme („WischerAktivieren“ und „BlinkerLinks“) ausgeführt wurden. Bedingt wird dies durch die sequentielle Abarbeitung der Testprogramme. Aber auch bei der parallelen Ausführung der Testprogramme ist keine Garantie möglich, dass dringende Testprogramme vorrangig durchgeführt werden. Vielmehr ist hier die Reihenfolge völlig zufällig. Die Einteilung der Testprogramme in Testpool A und Testpool B löst für beide Fälle das Problem. Dadurch dass alle Programme in einem Testpool die gleiche Priorität haben, spielt deren Reihenfolge keine Rolle mehr. Lediglich die Reihenfolge der Testpools ist von Bedeutung und diese kann beeinflußt werden.

Im Zusammenhang der Parallelisierung von Testprogrammen tritt jedoch ein weiteres Problem bei der Erstellung des Testpools zu Tage. Wie bereits beschrieben besteht der Standardfall aus einer gleichmäßigen Zuteilung der verfügbaren Testzeit auf alle Tester. Bei der sequentiellen Testausführung ist dies leicht umsetzbar. Die verfügbare Testzeit wird einfach durch die Anzahl der Tester geteilt. Anders sieht das bei der parallelen Testdurchführung aus. Hier hängt die verfügbare Zeit maßgeblich von den möglichen Kombinationen ab. Konkret bedeutet das, dass erst nach Erstellung der Testsuite klar ist, wieviel Zeit zusätzlich zur Testausführung eingeplant werden kann. Würde auf dieser Basis jedem Tester wiederum ein Zeitslot zugewiesen werden, wäre mit großer Wahrscheinlichkeit die berechnete Testsuite ungültig. Es müsste also eine neue Suite unter Berücksichtigung der individuellen Testzeit generiert werden. Daraus folgt aber eine neue verfügbare Testzeit, die individuellen Zeitslots wären ungültig und das Ganze ginge wieder von vorne los. Die gleichmäßige Aufteilung der Testzeit auf die Tester ist somit ein ungeeigneter Ansatz für die Parallelisierung von Testprogrammen.

5.2 Algorithmus zur Konflikterkennung

Die Konflikterkennung ist der zentrale Punkt bei der Parallelisierung von Testprogrammen. Das Ergebnis dieses Schritts ist eine Zusammenstellung, welche Testprogramme potentiell miteinander ausgeführt werden können. Das Ergebnis der Konflikterkennung lässt sich einer Konfliktmatrix beschreiben. Das Element der i-ten Zeile und der j-ten Spalte gibt das Verhältnis von Testprogramm i und j an. Weiterführende Informationen zur Konfliktmatrix finden sich in Kapitel 7.

Aufgrund der Komplexität der Testprogramme und nicht zuletzt durch deren Anzahl ist eine manuelle Bewertung nicht praktikabel. Somit muss ein Algorithmus definiert werden, durch welchen die Testprogramme automatisiert verglichen und Wechselwirkungen erkannt werden. Im folgenden Kapitel *„Konflikterkennung“* wird ein Algorithmus erarbeitet, welcher zwei Testprogramme auf Wechselwirkungen prüft. Das Ergebnis des Vergleichs gibt Aufschluss, ob ein Konflikt zwischen den beiden Testprogrammen besteht. Die Erstellung einer Testsuite, also welche Kombinationen der Programme in welcher Reihenfolge ausgeführt werden, wird im nächsten Schritt festgelegt.

5.3 Erstellung einer Testsuite

Dieser Schritt ist für die Erstellung einer Testsuite verantwortlich. Diese kann im einfachsten Fall aus einer zufälligen Kombination der Testprogramme bestehen, natürlich unter Berücksichtigung der Konfliktmatrix. Der Vorteil dieser Variante liegt in der sehr kurzen Berechnungszeit für die Testsuite. Allerdings werden hierbei meist Kombinationen erstellt, welche nur eine vergleichsweise geringe Zeitersparnis erreichen. Diese kann unter Einsatz von Optimierungsverfahren verbessert werden. Dies geht jedoch auf Kosten der Berechnungszeit. Es empfiehlt sich daher eine Zwischenlösung, welche in akzeptabler Zeit eine „gute“ Kombination berechnet. Die Quantifizierung von „gut“ muss dabei von Fall zu Fall festgelegt werden. Kapitel 7 stellt ein Verfahren vor, welches die Zeitersparnis maximiert. Dabei liegt der Fokus auf einer optimalen Lösung. An die notwendige Berechnungszeit werden keine Forderungen gestellt.

5.4 Ausführung von Testkombinationen

Nach den vorbereitenden Schritten folgt die Ausführung der Testprogramme. Diese wird durch eine übergeordnete Instanz initialisiert und gesteuert. Die Aufgaben dieser Instanz bestehen aus

- dem gleichzeitigen Start der Testprogramme: Parallel ausgeführte Testprogramme dürfen nicht versetzt gestartet werden.
- der Synchronisation der Testprogramme: Zu gewissen Zeitpunkten muss in den Ablauf der Testprogramme eingegriffen werden, um den reibungslosen Ablauf zu gewährleisten (vgl. 6.2.2, S. 68).

- Trennung der Ergebnisprotokolle: jedem Testprogramm muss das eigene Ergebnisprotokoll zugewiesen werden.

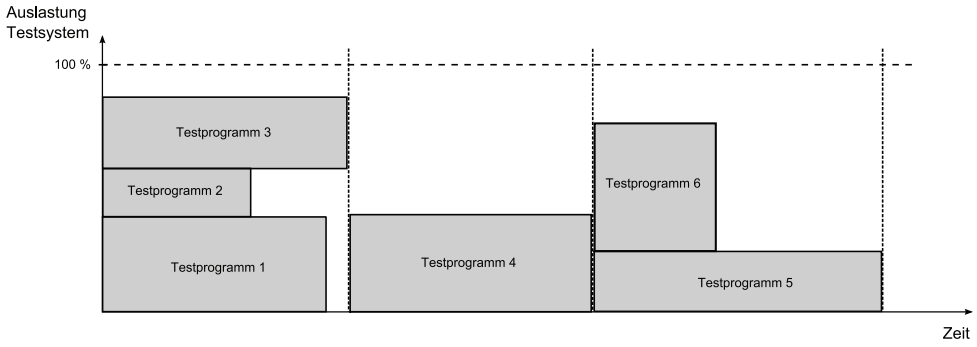

Abbildung 5.2: Parallele Ausführung von Testprogrammen

Abbildung 5.2 verdeutlicht den Ablauf der parallelen Ausführung von Testprogrammen. Ein Testprogramm wird hierbei durch zwei Größen charaktersiert. Zum einen durch die Dauer seiner Ausführung, zum anderen durch die maximale Auslastung des Echtzeitrechners bzw. des Testsystems. Beide Größen können aus vorherigen Ausführungen des Testprogramms gewonnen werden. Die Güte der Testsuite lässt sich demnach aus der Abbildung ablesen. Je kleiner die weißen Flächen sind, desto besser ist die Auslastung des Testsystems und umso größer die Zeitersparnis.

5.5 Auswertung der Ergebnisprotokolle

Das Ergebnisprotokoll gibt Aufschluß darüber, ob eine getestete Funktion unter den gegeben Rahmenbedingungen korrekt implementiert ist. In diesem Fall war der Test erfolgreich. Bei fehlgeschlagenen Tests ist hingegen mindestens eine Abweichung zwischen dem erwarteten und dem tatsächlichen Verhalten aufgetreten. Das Protokoll sollte demnach zusätzlich Detailinformationen enthalten, sodass eine Lokalisierung der Fehlerursache unterstützt wird. Das Beispiel 5.1 zeigt ein mögliches Ergebnisprotokoll des Programms 4.2 (S. 51).

```
 1   Assign Batterie = 1, 1.000
 2   Assign Zuendschluessel = 3, 1.000
 3   Assign Beifahrertuer = 0, 1.000
 4   Assign Fahrertuer = 0, 1.000
 5
 6   Assign Lenkstockschalter = 1, 1.000
 7   VerifyTimeout Wischer.inBewegung == 1,
 8          passed after 0.450, 1.450
 9   VerifyTimeout Wischer.Geschwindigkeit == 7, 0.5,
10          passed after 0.232, 1.232
11   Sleep 1
12
13   Assign Lenkstockschalter = 0, 2.450
14   Assign Zuendschluessel = 0, 2.450
15   Assign Batterie = 0, 2.450
```

Programm 5.1: Beispiel eines Ergebnisprotokolls

Jeder Schritt des Testprogramms spiegelt sich im Protokoll wider. Neben der Art der Aktion (Assign oder VerifyTimeout) wird zusätzlich ein Zeitstempel vermerkt. Dieser gibt relativ zum Testbeginn den Zeitpunkt der Aktion an. Im Falle eines VerifyTimeout (Zeile 7 bzw. 9) wird zusätzlich die Zeitspanne angegeben, nach welcher die Bedingung wahr wurde. Bei einem fehlgeschlagenen Test kann der Tester nun genau den Zeitpunkt bestimmen, zu welchem die Auswirkungen des Fehlers sichtbar wurden. Dadurch lässt sich die Ursache im Allgemeinen recht gut eingrenzen. Eine manuelle Nachstellung des Fehlers kann zusätzliche Informationen liefern.

Die Fehlersuche bei parallel ausgeführten Tests ist aufwendiger. Da das Testprogramm nicht alleine durchgeführt wurde, gibt es eine Reihe Aktionen aus anderen Testprogrammen, welche für das Fehlverhalten verantwortlich sein können. Die Information, welche Testprogramme gemeinsam ausgeführt wurden ist daher zwingend notwendig. Nur so kann der Tester sich ein Bild von potentiellen Einflüßen machen. Gibt das Ergebnisprotokoll keinen Aufschluß über mögliche Fehlerursachen muss der Tester versuchen, den Fehler manuell nachzustellen. Tritt er erneut auf, ist ein Einfluß von anderen Testpro-

grammen nicht gegeben. Kann der Fehler jedoch nicht nachgestellt werden handelt es sich entweder um einen sporadischen Fehler oder er ist durch eines der anderen Programme ausgelöst worden. Ist ein sporadischer Fehler auszuschließen, etwa durch erneutes Ausführen der Testkombination, muss der Fehler unter Berücksichtigung der anderen Programme gesucht werden. Hierfür ist es empfehlenswert dem Tester Hilfestellungen, beispielsweise in Form einer Tool-Unterstüzung, zur Verfügung zu stellen. Dies könnte von einer einfachen zeitlichen Gegenüberstellung aller beteiligten Testprogramme bis hin zu einer „lessions learnt“ Datenbank reichen. Die Entscheidung, welche Ursache dem Fehler tatsächlich zu Grunde liegt, muss aber vom Tester selbst getroffen werden.

Kapitel 6

Konflikterkennung

Vor der parallelen Ausführung der Testprogramme müssen diese auf Konflikte untereinander geprüft werden. Die Regeln dieser Prüfung stehen im Mittelpunkt dieses Kapitels. Hierfür werden die Ursachen für Konflikte kategorisiert und Indikatoren für ihr Auftreten erarbeitet. Als Resultat wird ein Algorithmus definiert, mit welchem automatisiert der Vergleich zweier Testprogramme durchgeführt werden kann. Beginnen wird das Kapitel jedoch mit der Definition der *Unabhängigkeit*, welcher im Zusammenhang der Konflikterkennung eine wesentliche Bedeutung zukommt.

6.1 Unabhängigkeit von Testprogrammen

Bevor Regeln für eine Prüfung definiert werden können, muss geklärt sein, was durch sie erkannt werden soll. Für die Parallelisierung steht die Erkennung von Konflikten im Vordergrund. Ein Konflikt ist immer dann gegeben, wenn Wechselwirkungen zwischen zwei Testprogrammen bestehen. In diesem Fall führen Aktionen des einen Testprogramms zu Reaktionen im anderen. Aufgrund dieser Einflüsse sind die beiden Programme nicht unabhängig.

Definition: Unabhängigkeit
Zwei Testprogramme heißen unabhängig, falls die Teilergebnisse der parallelen Ausführung denen der sequentiellen Ausführung entsprechen.

Ein Konflikt entsteht demnach genau dann, wenn zwei Testprogramme nicht unabhängig sind. Analog hierzu lässt sich Unabhängigkeit auch für mehrere

Programme definieren. Folglich heißen n Testprogramme unabhängig, falls gilt

$$(T_i, T_j) \text{ unabhängig } \forall i \neq j, \quad i, j \in \{1, .., n\}$$

Existiert ein (T_i, T_j), welches nicht unabhängig ist, so besteht ein Konflikt zwischen den n Testprogrammen.

Diese Definition zeigt bereits, dass die Konflikterkennung immer auf der Bewertung einer Testpaarung beruht. Dabei spielt es keine Rolle, wie viele Programme tatsächlich parallelisiert werden sollen. Zudem wird klar, dass die Unabhängigkeit als Relation symmetrisch und nicht transitiv ist.

Für diese Arbeit hat sich eine Unterteilung der Einflüsse in Kategorien als sinnvoll erwiesen. Ausschlaggebend hierfür war die Analyse bestehender Testprogramme. Diese lassen eine inhaltliche Einteilung in drei Abschnitte zu

- Stimulation und Überprüfung
- Kodierung
- Restbussimulation

Nachfolgend werden Kriterien aufgestellt, anhand welcher Konflikte zwischen zwei Testprogrammen erkannt werden können. Da zwischen den einzelnen Kategorien keine Abhängigkeiten bestehen, kann die Konflikterkennung auf jeweils eine Kategorie beschränkt werden. Zwei Testprogramme sind somit unabhängig, falls sie bezüglich aller Kategorien unabhängig sind. Sollen mehr als zwei Testprogramme gleichzeitig ausgeführt werden, verläuft die Konflikterkennung weitgehend analog. Hierfür müssen die Programme jedoch paarweise unabhängig sein. Die Unterschiede der Konflikterkennung werden an den entsprechenden Stellen erläutert.

6.2 Stimulation und Überprüfung

Die Konflikterkennung bei der Stimulation bildet die größte Herausforderung bei der automatisierten Testanalyse. Viele Funktionen sind auf mehrere Komponenten verteilt. Zusammen mit dem hohen Vernetzungsgrad der

Steuergeräte resultiert daraus eine Vielzahl möglicher Einflußfaktoren. Die Beherrschung dieser Komplexität ist Voraussetzung für eine zuverlässige Testanalyse.

Die Testanalyse basiert hierbei auf der Auswertung von Signalen. Mittels dieser kommunizieren die Steuergeräte mit ihrer Umwelt. Gleichzeitig werden die Steuergeräte durch die Signale stimuliert. Sie bilden daher die Grundlage zur Analyse der Testprogramme. Die Auswirkungen eines Signals auf die Unabhängigkeit der Testprogramme hängt dabei vom Zeitpunkt der Verwendung innerhalb des Testablaufs ab. Diese kann eindeutig einem der drei Abschnitte aus 4.4.2 (S. 50) zugeordnet werden.

6.2.1 Die Abbruchbedingung

Die Funktionen heutiger Automobile zeichnen sich unter anderem dadurch aus, auf sich ändernde Gegebenheiten reagieren zu können. Konkret bedeutet das, dass aktive Funktionen bei Bedarf ihr Verhalten ändern oder gar beendet bzw. unterbrochen werden können. Beispielsweise wird der Tempomat durch Betätigung des Bremspedals deaktiviert. Derartige Einflüsse bilden die Abbruchbedingung einer Funktion. Die Abbruchbedingung muss daher in die Analyse der Testprogramme mit einfließen. Sie ist jedoch nicht in den Testprogrammen enthalten und muss auf eine andere Art und Weise zur Verfügung stehen. Hierfür werden die Funktionen hierarchisch gegliedert. Abbildung 6.1 (S. 65) verdeutlicht das Prinzip anhand der Funktion „Wischen“. Im Gegensatz zur Funktionshierarchie aus Kapitel 4 (Abbildung 4.1, S. 42) handelt es sich hierbei um eine Baumstruktur. Dabei ist jeder Knoten eine

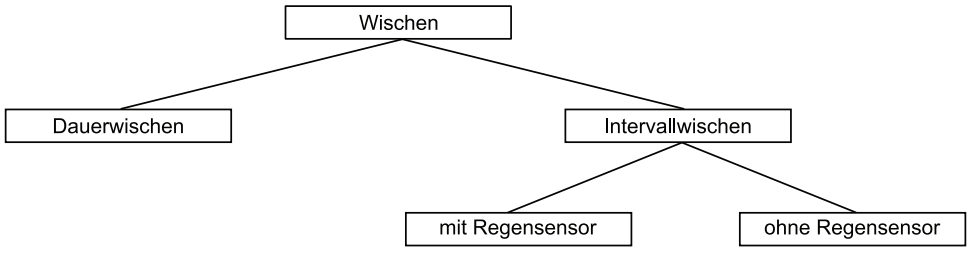

Abbildung 6.1: Exemplarischer Funktionsbaum

Spezialisierung der Funktion, welche durch den Elternknoten repräsentiert

wird. Durch diesen Aufbau kann die Abbruchbedingung auf alle Kindknoten vererbt werden. Jedem Knoten müssen nun lediglich die zusätzlichen Abbruchbedingungen hinzugefügt werden. Abbildung 6.2 (S. 66) verdeutlicht das Prinzip anhand des rechten Teilbaums aus Abb. 6.1. Die Baumstruk-

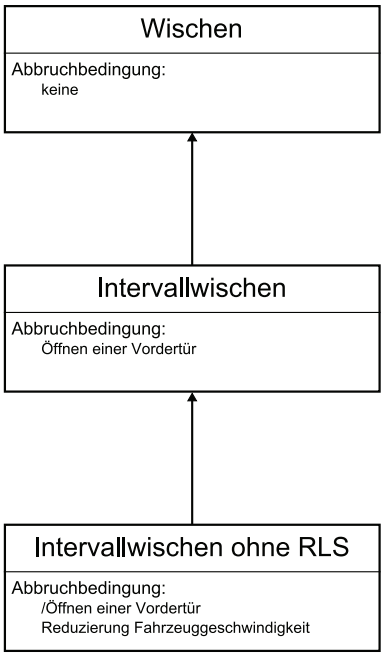

Abbildung 6.2: Vererbung Abbruchbedingung

tur kann leicht in eine maschinell verwertbare Form transformiert werden. Verweist ein Testprogramm nun auf die Funktion die getestet wird, so steht auch die Abbruchbedingung zur Verfügung. Somit stehen alle notwendigen Informationen bereit und zwei Testprogramme lassen sich auf Unabhängigkeit überprüfen.

Im Folgenden wird ein Algorithmus vorgestellt, anhand dessen ein Konflikt aufgrund von Stimulationen erkannt wird. Analog zur allgemeinen Definition kann Unabhängigkeit für die einzelnen Testabschnitte definiert werden. Zwei Testprogramme sind demnach genau dann Unabhängigkeit, wenn alle Testabschnitte unabhängig sind.

6.2.2 Konflikte in der Initialisierung

Konflikte in der Initialisierung entstehen aufgrund widersprüchlicher Startszenarien. Diese sind definiert durch die verwendeten Signale, deren Werte und Reihenfolge. Die Initialisierung kann daher beschrieben werden durch

$$I = \{(s, w) | s \in S, w \in \mathbb{R}^{n_s}, n_s \geq 1\},\ S\ =\ Menge\ aller\ Signale$$

Die n_s-dimensionalen Vektoren w repräsentieren dabei die Werte, welche dem Signal zugewiesen werden (bei schreibenden Zugriff) beziehungsweise welche es annehmen soll (lesender Zugriff). Das bedeutet für $a, b \in \mathbb{R}$

$$(s, a, b) \iff \begin{array}{l} s \text{ hat zum Zeitpunkt } t_x \text{ den Wert a} \\ s \text{ hat zum Zeitpunkt } t_y,\ t_x < t_y \text{ den Wert b} \end{array}$$

Diese Menge I wird für beide Testprogramme beziehungsweise deren Initialisierungen erstellt. Durch den Vergleich beider Mengen werden potentielle Konflikte anhand des folgenden Algorithmus erkannt.

Im ersten Schritt wird ermittelt, ob ein Signal s existiert, für welches gilt:

$$\exists w_1, w_2 : (s, w_1) \in I_{TestA}, (s, w_2) \in I_{TestB} \tag{6.1}$$

Existiert kein s, so dass (6.1) erfüllt ist, ist ein Konflikt in der Initialisierung ausgeschlossen. Beide Testprogramme benötigen in diesem Fall Startszenarien, welche keinerlei gemeinsame Ressourcen besitzen. Da die Funktionsvoraussetzung und damit auch die Initialisierung alle notwendigen Vorgaben enthält, können die Szenarien sich gegenseitig nicht beeinflussen. Existiert dagegen mindestens ein Signal, welches (6.1) erfüllt, liegt nicht zwingend ein Konflikt vor. Bei gleichwertiger Stimulation des Systems können die Testprogramme noch unabhängig sein.

Definition: gleichwertige Stimulationen

Seien $(s, w) \in I_{TestA},\ (s, v) \in I_{TestB}$

Seien n_1 die Dimension von w und n_2 die Dimension von v
o.B.d.A. sei $n_1 \geq n_2$

Dann heißt die Stimulation eines Signals s durch zwei Testprogramme gleichwertig, falls gilt

$$\forall i \in \{1, .., n_2\} : w_i = v_{i+n_1-n_2} \tag{6.2}$$

Damit eine gleichwertige Stimulation durch zwei Testprogramme nicht zu Konflikten führt, muss in den Testablauf eingegriffen werden. Dabei müssen gleiche Testanweisungen zeitlich synchronisiert werden. Dies kann bei der Verwendung mindestens zweier Signale durch beide Testprogramme zu Deadlocks und damit zu einem Konflikt führen. Existiert hingegen nur ein Signal, welches (6.1) erfüllt, sind Deadlocks ausgeschlossen und aus (6.2) folgt die Unabhängigkeit der Initialisierungen.

Synchronisation

Erst die Möglichkeit der Synchronisation kann bei gleichwertiger Stimulation eines Signals einen Konflikt verhindern. Dies hat jedoch einen Eingriff in den zeitlichen Ablauf des Testprogramms zur Folge. Damit dieser keinen Einfluß auf das Testszenario hat, darf der Zeitpunkt der Ausführung die Wirkung nicht beeinträchtigen. Die Initialisierung erfüllt diese Vorgabe, da sie keiner Echtzeitanforderung unterliegt. Beispiel 6.1 (S. 69) verdeutlicht die Vermeidung eines Konflikts durch Synchronisation zweier Testprogramme auf die gleiche Aktion.
Allerdings muss vor der Synchronisation sichergestellt sein, dass keine Deadlocks existieren. Diese können bei gleicher Stimulation entstehen, falls sie in unterschiedlicher Reihenfolge ausgeführt werden. Wird etwa Test B aus Bsp. 6.1 (S. 69) um die Anweisung „Fahrertür öffnen“ zum Zeitpunkt t_2 erweitert, so sind die Testprogramme nicht mehr unabhängig.

Um Deadlocks zwischen zwei Testprogrammen auschliessen zu können, muss (6.3) erfüllt sein. Für mehr als zwei Testprogramme ist dies jedoch nicht ausreichend. Hierfür müssen bei der Untersuchung auf Deadlocks die Signale

Zeitpunkt	Initialisierung Test A	Initialisierung Test B
t_1	$\vdots$	Zündung AN
t_2	Fahrertür öffnen (bei Zündung AUS)	$\vdots$
t_3	Zündung AN	$\vdots$
t_4	$\vdots$	$\vdots$

Die Ausführung der Initialisierung von Test B wird zum Zeitpunkt t_1 blockiert und erst wieder zu t_3 fortgesetzt. Dadurch wird ein Konflikt zum Zeitpunkt t_2 vermieden.

Bsp. 6.1: Auflösung eines Konflikts durch Synchronisation

aller Testprogramme betrachtet werden. Der paarweise Ausschluss von Deadlocks reicht nicht aus.

Sei $I_{TestA} = \{(i_1, v_1), .., (i_n, v_n)\}$ die Menge der Signal-Wert-Tupel, welche in der Initialiserung beider Testprogramme vorkommen, sortiert nach ihrem zeitlichen Auftreten in Testprogramm A. Analog sei $I_{TestB} = \{(j_1, w_1), .., (j_m, w_m)\}$ die Menge der Signal-Wert-Tupel, welche in der Initialiserung beider Testprogramme vorkommen, sortiert nach ihrem zeitlichen Auftreten in Testprogramm B.

$$\forall r \in \{1, .., n\},\ s \in \{1, .., m\} : i_r = j_s \Rightarrow i_t \neq j_u, t < r, s < u \tag{6.3}$$

Deadlockerkennung bei mehr als zwei Testprogrammen

Folgendes Beispiel 6.2 verdeutlicht die Problematik, wenn für mehr als zwei Testprogramme ein Konflitk aufgrund von Deadlocks erkannt werden soll. Zwischen jeweils zwei Testprogrammen besteht kein Konflikt aufgrund eines Deadlocks. Anders sieht das allerdings bei der parallelen Ausführung aller Testprogramme aus. In diesem Fall könnte Testprogramm A das Fahrzeug erst dann entriegeln, wenn Testprogramm C die Fahrertür geöffnet hat. Hierfür muss zuvor durch Programm B der Schlüssel auf die Position Zündung AN gestellt werden. Dies kann aber erst nach entriegeln des

Testprogramm A	Testprogramm B	Testprogramm C
Fahrzeug entriegeln	Zündung AN	Fahrertür AUF
Zündung AN	Fahrertür AUF	Fahrzeug entriegeln

Bsp. 6.2: Deadlock aufgrund mehrerer Testprogramme

Fahrzeugs durch Testprogramm A geschehen.

Soll zu einer aus zwei Testprogrammen (A und B) bestehenden Paarung ein weiteres Testprogramm (C) hinzugefügt werden, wird folgender Schritt durchgeführt. Zur Erkennung eines Deadlocks werden die Signale der Initialisierung aus Testprogramm A und B zu einer Menge zusammengefasst. Diese wird mit der Initialisierung von Programm C anhand (6.3) geprüft. Somit wird ein Deadlock erkannt. Die paarweise Unabhängigkeit nach den in diesen Kapiteln definierten Regeln gilt selbstverständlich aus Grundlage. Soll ein viertes Testprogramm hinzugefügt werden, müssen die Signale aus A, B und C zusammengefasst und mit denen des neuen Programms verglichen werden.

Zusätzlich zur Synchronisation innerhalb der Initialisierung müssen die einzelnen Abschnitte der Testprogramme ebenfalls synchronisiert werden. Das bedeutet, dass die Durchführung beider Testprogramme erst dann ausgeführt werden darf, wenn die Initialisierung beider Programme beendet ist. Gleiches gilt für den Übergang von Durchführung zu Abschluss. Da die Übergänge der Testabschnitte zeitlich keinen strikten Vorgaben folgen müssen, ist die Synchronisation problemlos möglich. Das Testergebnis bleibt davon unberührt.

6.2.3 Konflikte in der Durchführung

Konflikte in der Durchführung haben unmittelbaren Einfluß auf die getestete Funktion und damit auf das Testergebnis. Sie können dabei mehrere Ursachen haben:

1. Wegfall der Funktionsvoraussetzung: durch ein anderes Testprogramm wird die Funktionsvoraussetzung komplett oder teilweise entfernt. Die Funktion wird abgebrochen.

2. Rücknahme der Stimulation: muss die Stimulation permanent anliegen (z.B. Blinker) kann diese durch andere Testprogramme zurückgenommen werden. Die Funktion wird beendet und führt in dem Testprogramm zu einem Fehler, welcher das konstante Anliegen der Stimulation erwartet.

3. Eintreten der Abbruchbedingung: die Stimulation eines parallel ausgeführten Testprogramms enthält Teile der Abbruchbedingung.

Wie bereits bei der Konflikterkennung der Initialisierung (6.2.2, S. 67) werden die Konflikte anhand der verwendeten Signale erkannt. Für jedes Testprogramm werden diese wiederum zu einer Menge D zusammengefasst. Im Gegensatz zur Initialisierung spielen die Werte der Signale allerdings keine Rolle. Dies resultiert aus der Echtzeitanforderung an die Durchführung und der damit fehlenden Möglichkeit der Synchronisation. Identische Stimulationen können daher bereits zu einen Konflikt führen, wenn sie zu unterschiedlichen Zeitpunkten ausgeführt werden (vgl. Beispiel 6.1, S. 69). Somit gilt

$$D \subseteq S,\ S\ =\ Menge\ aller\ Signale$$

Nach 4.4.2 (S. 50) ist die Funktionsvoraussetzung in der Initialisierung enthalten. Demnach ist 1. gleichbedeutend damit, dass kein Signal in der Durchführung des einen Testprogramms, sowie in der Initialisierung des anderen enthalten ist. Zum Vergleich der beiden Mengen D und I wird die erste Projektion des kartesischen Produkts $I = S \times \mathbb{R}^n$ benötigt. Diese ist definiert durch

$$f : I \to S,\ mit\ f(s,w) = s,\ (s,w) \in I$$

Weiter sei S' das Bild von I unter f, d.h.

$$S' = \{s \in S | \exists i \in I : s = f(i)\}$$

Dann lässt sich 1. schreiben als

$$D_{TestA} \bigcap S'_{TestB} = \emptyset \tag{6.4}$$

$$\wedge$$

$$D_{TestB} \bigcap S'_{TestA} = \emptyset \tag{6.5}$$

Im Zusammenhang mit der Konflikterkennung wird der Begriff der Stimulation erweitert. Die Stimulation umfasst nicht nur das Auslösen einer Funktion, sondern auch die daraus resultierende Wirkkette. Am Beispiel der Wischeraktivierung heißt das, dass die Stimulation nicht nur das Betätigen des Lenkstockschalters bedeutet. Es beinhaltet auch das etwaige Routing der Anforderung über den Master bis hin zum Wischermotor. Dies ist notwendig, da nicht alle Konflikte bereits beim Auslösen der Funktion erkannt werden können. Vielmehr werden die Konflikte erst im Verlauf der Wirkkette deutlich. Daher kann 2. wie folgt formalisiert werden

$$D_{TestA} \bigcap D_{TestB} = \emptyset \tag{6.6}$$

Analog zum bisherigen Vorgehen werden die Signale der Abbruchbedingung zu einer Menge A zusammengefasst. Bei der Abbruchbedingung spielen die Werte der Signale ebenfalls keine Rolle. Die Menge A hat daher die gleiche Struktur wie D. Somit lässt sich 3. wie folgt darstellen

$$D_{TestA} \bigcap A_{TestB} = \emptyset \tag{6.7}$$

$$\wedge$$

$$D_{TestB} \bigcap A_{TestA} = \emptyset \tag{6.8}$$

Anhand der Gleichungen (6.4)-(6.8) werden alle Konflikte in der Durchführung erkannt. Einzig die beiden folgenden Spezialfälle lassen sich durch die bisherigen Regeln nur schwer bewerten. Aufgrund dieser Sonderstellung werden sie nachfolgend genauer untersucht.

6.2.4 Sonderfall: Signale mit großem Wertebereich

Die bislang aufgestellten Regeln eignen sich besonders für den Einsatz bei Signalen, die einen kleinen Wertebereich besitzen. Sie sind somit für die meisten Signale sinnvoll, da der Wertebereich in der Regel aus der Aufzählung möglicher Zustände besteht. Es gibt jedoch auch Signale, deren Wertebereich sehr groß ist. Dazu zählt beispielsweise die Bordspannung. Da nicht für jeden einzelnen Wert eine unterschiedliche Reaktion einer Funktion zu erwarten ist, wird der Wertebereich in Teilmengen geteilt. Somit gilt für den Wertebereich W eines Signals S bezüglich einer Funktion F:

$$W^F(S) = \bigcup_{i=1}^{N} W_i^F(S) \quad \text{, mit } W_i^F(S) \subset W^F(S)$$

Damit eine eindeutige Zuordnung eines Signalwerts in eine Teilmenge möglich ist, dürfen die Teilmengen keine gemeinsame Punkte enthalten. Dadurch gilt zusätzlich

$$W_1^F, .., W_N^F \text{ sind paarweise disjunkt}$$

Für eine Funktion ist es nun lediglich interessant, in welchen Bereich der Signalwert liegt, nicht jedoch der genaue Wert. Dies kann für die Konflikterkennung ausgenutzt werden. In diesem Fall ist es nicht notwendig, dass der Wert in beiden Programmen derselbe ist. Es ist ausreichend, wenn die Schnittmenge der relevanten Teilmengen nicht der leeren Menge entspricht. Sollten die Signalwerte der beiden Testprogramme innerhalb der Schnittmenge unterschiedlich sein, so können die Signale nach einem beliebigen Verfahren auf den gleichen Wert gesetzt werden.

Die Unterteilung des Wertebereichs kann entweder direkt im Testprogramm durch Angabe der relevanten Teilmengen oder durch eine Verknüpfung analog zur Abbruchbedingung erfolgen (vgl. 6.2.1, S. 65). Aufgrund der logischen Zuordnung dieser Aufteilung zu einer Funktion und nicht zum Testprogramm ist letztere Möglichkeit vorzuziehen.

Wichtigster Vertreter dieser Signale ist die Bordspannung. Sie verfügt über einen großen Wertebereich, der sich gut in wenige Teilmengen gliedern lässt. Des Weiteren ist die Spannung für jede Funktion von Bedeutung, so dass die Anwendung der Regeln aus 6.2.2 und 6.2.3 eine starke Einschränkung bedeuten würde. Im Folgenden wird anhand der Spannung das Verfahren exemplarisch an zwei Funktionen demonstriert.

Für die Funktionen „Wischer“ und „Fensterheber“ werden jeweils drei Teilmengen definiert sowie das Verhalten der Funktion darin festgelegt (vgl. Tabelle 6.1, S. 74). *Normalbetrieb* bedeutet in diesem Zusammenhang, dass die

Funktion ohne Einschränkungen ausgeführt werden kann. *Unter-* und *Überspannung* führen dagegen zu einem verminderten Funktionsumfang. Für die Funktion *Fensterheber* könnte das beispielsweise bedeuten, dass keine automatischen Fensterbewegungen möglich sind.

	Unterspannung	Normalbetrieb	Überspannung
Wischer	< 9V	9-15V	> 15V
Fensterheber	< 11V	11-16V	> 16V

Tabelle 6.1: Spannungsklassen

Soll nun das Testprogramm zur Funktion *Wischer* mit einer Spannung von 12V ausgeführt werden, das andere hingegen bei 13V, so entsteht dennoch kein Konflikt, da

$$W_2^{Wischer} \cap W_2^{Fensterheber} = [11, 15] \neq \emptyset$$

Es kann daher eine beliebige Spannung zwischen 11V und 15V während der Ausführung der Testprogramme am Testobjekt anliegen, ohne dass eine der beiden Funktionen ein anderes Sollverhalten hat. In diesen Beispiel wird die Spannung durch das Testprogramm bestimmt, welches sie zuletzt ändert (und dadurch die Einstellung des anderen überschreibt). Allgemein lautet die Regel zur Konflikterkennung für Signale mit großem Wertebereich dann wie folgt:

Sei S ein Signal, welches in beiden Testprogrammen verwendet wird. Seien s_A, s_B die Werte des Signals im Testprogramm zur Funktion A bzw. zur Funktion B. Ferner gelte $s_A \in W_i^A(S)$ und $s_B \in W_j^B(S)$. Dann sind die Testprogramme bezüglich des Signals S unabhängig, falls (6.9) gilt.

$$W_i^A(S) \cap W_j^B(S) \neq \emptyset \tag{6.9}$$

Diese Regel kann jedoch nur dann verwendet werden, wenn das Signal in mindestens einem der Testprogramme nur einmal verwendet wird. Werden die Signalwerte o.B.d.A[1] durch das Programm A mehrfach geändert, müssen

[1] ohne Beschränkung der Allgemeinheit

alle Werte des Signals aus Programm A in $W_i^B(S)$ enthalten sein. In diesem Fall wird das Signal komplett durch Programm A stimuliert. Die entsprechende Anweisung in Testprogramm B wird ignoriert.

Verändern beide Testprogramme mehrfach die Signalwerte, muss von einem Konflikt ausgegangen werden. In diesem Fall erwarten beide Programme ein genaues zeitliches Verhalten des Signals. Eine automatische Generierung dieses Verlaufs aus zwei unterschiedlichen Sequenzen ist nicht zu garantieren.

Ab wann ein Signal unter diese Kategorie fällt und nach Regel 6.9 behandelt werden kann, liegt im Ermessen der Testautoren. Diese Entscheidung gilt dann natürlich für alle Testprogramme einer Baureihe und ist nicht auf eine Teilmenge beschränkt. Prinzipiell sollte dieses Verfahren bei Signalen angewandt werden, deren Wertebereich sich gut in wenige Mengen teilen lässt. Außerdem sollte das Signal für eine ausreichend große Anzahl von Testprogrammen bzw. Funktionen relevant sein, so dass sich der Mehraufwand der Einteilung rechnet.

6.2.5 Sonderfall: Busruhe

Um den Stromverbrauch des Fahrzeugs möglichst gering zu halten, stellen einzelne Komponenten die Kommunikation auf den Bussystemen ein, sobald diese nicht mehr notwendig ist. Sie wechseln in einen passiven Zustand, in welchem sie lediglich die Kommunikation verfolgen. Beenden alle an einem Bus hängenden Steuergeräte die aktive Kommunikation, so spricht man von Busruhe. Hält die Busruhe über einen bestimmten Zeitraum an, wechseln die Steuergeräte in einen Ruhezustand, sofern nicht interne Funktionen aktiv sind. Nimmt ein Steuergerät die Kommunikation wieder auf, so wird der Bus geweckt. Alle Steuergeräte melden sich am Bus und beenden ihren Ruhezustand.
Im Zusammenhang mit Busruhe gibt es daher zwei wichtige Testszenarien

1. Übergang vom Normalbetrieb in Busruhe: beenden alle Steuergeräte die Kommunikation oder existieren Wachhalter?

2. Bus wecken: Wird der Bus geweckt und die angestoßene Funktion korrekt umgesetzt?

In Szenario 1 wird Busruhe als Teil des Funktionsverlaufs betrachtet. Um eine Aussage über mögliche Konflikte mit anderen Testprogrammen machen

zu können muss die Abbruchbedingung für Busruhe bekannt sein (vgl. 6.2.1, S. 65). Da jede Aktion, welche ein beliebiges Steuergerät zur Kommunikationsaufnahme veranlasst, die Busruhe stört, wird die Abbruchbedingung sehr mächtig. Gleichzeitig wird die Auswahl möglicher Partner zur parallelen Ausführung sehr klein. Der Aufwand für die Erstellung der Abbruchbedingung würde daher den Nutzen überschreiten.
Szenario 2 ist ebenso nicht zur Parallelisierung geeignet. In diesem Fall ist Busruhe eine Funktionsvoraussetzung. Die Aktion, welche zum Wecken des Busses führt, ist das Auslöseereignis. Um sicherzustellen, dass der Bus nicht durch die Durchführung anderer Testprogramme geweckt wird, müsste das Auslöseereignis (Bus wecken) in der Initialisierung stattfinden. Dies resultiert aus dem Verbot einer Synchronisation in der Durchführung. Das Auslöseereignis muss aber in der Durchführung enthalten sein (vgl. 4.4.2, S. 50).
Eine Parallelisierung von Testprogrammen der Funktion *Busruhe* ist daher unter Berücksichtigung der aufgestellten Rahmenbedingungen nicht möglich. Somit folgt im weiteren Verlauf der Arbeit aus Busruhe ein Konflikt.

6.3 Kodierung

Die Kodierung dient der Parametrisierung der Steuergeräte. Im Zuge der Verblockung einzelner Komponenten über verschiedene Fahrzeugvarianten und Baureihen, werden die Unterschiede der Funktionen über interne Parameter erreicht. Damit hat die Kodierung der Steuergeräte direkten Einfluß auf die Unabhängigkeit der Testprogramme.

Im Gegensatz zur bisherigen Konflikterkennung kann die Analyse von Signalen bei der Kodierung keine Konflikte aufdecken. Zum einen liegt das daran, dass nicht alle Kodierparameter durch ein Signal auf dem Bus repräsentiert werden. Zum anderen ist eine Aussage zur Unabhängigkeit anhand des Vergleichs zweier unterschiedlicher Signale nicht automatisiert treffbar. Die Konflikterkennung bei der Kodierung muss sich daher anderer Mittel bedienen. Diese hängen von der Domäne ab, zu welcher der Parameter gehört. Dabei handelt es sich um die *globale Variantenkodierung* und um die *lokale Variantenkodierung*.

6.3.1 Globale Variantenkodierung

Mittels der globalen Variantenkodierung werden Informationen bezüglich der Ausstattung des Fahrzeugs an die Steuergeräte weitergeleitet. Sie repräsentiert somit die reale Fahrzeugkonfiguration auf Softwareebene. Beispiele für derartige Informationen sind Sonderausstattungen (SA), wie etwa Regensensor oder Diebstahlwarnanlage. Aber auch Angaben zur Aufbauvariante sind in der globalen Variantenkodierung enthalten, beispielsweise Karosserieart oder Baureihe.

Konflikte können bei der Kodierung auf zwei Ursachen zurückgeführt werden. Zum einen können identische Kodierparameter durch zwei Testprogramme auf unterschiedliche Werte geändert werden. Dieser Fall kann leicht erkannt werden. Zum anderen kann die Kombination zweier oder mehrerer Parameter einen Konflikt verursachen. Beispielsweise ist Distronic[2] ohne Automatikgetriebe nicht verfügbar. Da beide durch einen entsprechenden Kodierparameter repräsentiert werden, würde die Kombination aus Beispiel 6.3 zu einem Konflikt führen.

Distronic	=	vorhanden	} *widersprüchliche Einstellung*
Automatikgetriebe	=	nicht vorhanden	

Bsp. 6.3: Konflikte in der Kodierung

Derartige Konflikte lassen sich jedoch ohne eine Datenbank nicht automatisiert erkennen. In dieser Datenbank müsste zu allen möglichen Kombinationen hinterlegt werden, ob aus dieser ein Konflikt resultiert. Tabelle 6.2 (S. 78) verdeutlicht den Aufwand in Abhängigkeit der Anzahl an Parametern. Hierbei wird unterstellt, dass jeder Parameter nur zwei Werte annehmen kann.

Bei bis zu 60 Kodierparametern ist dieser Aufwand nicht leistbar. Konflikte können jedoch auch nicht ohne manuelle Vorgaben erkannt werden. Als Grundlage für die Kodierung durch Testprogramme dienen daher sogenannte

[2] Abstandsregeltempomat: regelt die Geschwindigkeit und hält somit automatisch den Abstand zum vorausfahrenden Fahrzeug [MB07b]

Anzahl Parameter	minimale Anzahl Kombinationen (boolsche Parameter)
2	4
5	32
10	1024
50	1,1e+15

Tabelle 6.2: Kombinationen in Abhängigkeit der Anzahl an Parametern

Basiskonfigurationen. Diese werden von Hand erstellt und sind daher frei von Widersprüchen. Zusätzlich werden Änderungen durch die einzelnen Testprogramme erlaubt. Dies bietet die Möglichkeit ausgehend von einer geringen Anzahl an Basiskonfigurationen alle notwendigen Varianten zu erreichen. Die Änderungen unterliegen dabei folgenden Regeln

1. aus jeder Änderung von Parameterwerten gegenüber der Basiskonfiguration durch das Testprogramm resultiert eine neue Konfiguration. Diese darf keine Widersprüche enthalten.

2. jedes Testprogramm muss alle Parameterwerte kodieren, auch wenn sie dem Wert der Basiskonfiguration entsprechen.

Die Kodierung eines Testprogramms besteht also aus der Wahl der Basiskonfiguration, sowie aus den Änderungen einzelner Parameterwerte (siehe Bsp. 6.4, S. 79). Dabei muss sich das Testprogramm nicht auf eine Basiskonfiguration festlegen, sondern kann alle passenden Konfigurationen aufzählen.

Durch dieses Verfahren ist eine automatisierte Konflikterkennung möglich. Die Testprogramme sind bezüglich der Kodierung unabhängig, wenn gilt:

1. mindestens eine Basiskonfiguration ist für beide Testprogramme zulässig

2. der Wert eines Parameters wird nur von einem Testprogramm geändert

   ODER

   der Wert eines Parameters wird durch beide Testprogramme geändert und ist identisch

Testprogramm

⋮	
Auswahl Basiskonfiguration	A (u.a. Lichtsensor = vorhanden)
Änderungen	Regensensor = nicht vorhanden Klimaanlage = nicht vorhanden Lichtsensor = vorhanden ⋮
⋮	

Obwohl durch die Basiskonfiguration der Parameter *Lichtsensor* bereits auf den Wert *vorhanden* gesetzt wird, muss dieser durch das Testprogramm erneut angegeben werden. Hierdurch signalisiert das Testprogramm, dass dieser Parameter nicht verändert werden darf.

Bsp. 6.4: Kodierung durch Testprogramme

Die Wahl der Basiskonfigurationen ist für die Konflikterkennung von wesentlicher Bedeutung, da in diesen die Informationen über widersprüchliche Kombinationen enthalten sind. Gleichzeitig darf die Anzahl nicht zu groß werden. Mit zunehmender Anzahl an Basiskonfigurationen nimmt die Unabhängigkeit wegen Regel 1 ab, auch wenn die Kodierung nicht zu einem Konflikt führen würde. Um diesem Trend entgegenzuwirken sollte die Anzahl an Basiskonfigurationen auf ein Minimum reduziert werden. Dieses Minimum wird durch die Forderung definiert, dass ausgehend von den Basiskonfigurationen, durch Änderungen einzelner Parameter, alle möglichen Kombinationen erreicht werden können.
Die erlaubten Änderungen hängen dabei von dem Parameter ab. Parameter, welche sich auf die Ausstattung (insbesondere SA) beziehen, dürfen geändert werden. Allerdings nur unter der Voraussetzung, dass dadurch sich die Ausstattung verringert bzw. sich der Wert von „vorhanden“ nach „nicht vorhanden“ ändert. In Verbindung mit Forderung 1 auf Seite 78 kann somit Konfliktfreiheit laut obiger Regel garantiert werden. Für alle anderen Parameter sind Änderungen im Testprogramm nicht erlaubt und müssen über die Wahl der Basiskonfiguration berücksichtigt werden.

6.3.2 Lokale Variantenkodierung

Bei der lokalen Variantenkodierung werden interne Parameter eines Steuergeräts geändert. Die Auswirkung solcher Änderungen beschränken sich auf Funktionen, welche in diesem Steuergerät implementiert sind. Die Funktionsweise der anderen Steuergeräte ist dadurch nicht beeinträchtigt. Somit ist das kodierte Steuergerät als Ressource definiert, welche nicht von mehreren Testprogrammen verwendet werden darf. Ein Konflikt entsteht demnach, wenn Funktionen eines Steuergeräts geprüft und gleichzeitig Kodierparameter durch ein anderes Testprogramm geändert werden. Die Information, welche Steuergeräte an der geprüften Funktion beteiligt sind, ist im Testprogramm enthalten (vgl. 4.4.1, S. 49).

6.4 Restbussimulation

Die Restbussimulation beschreibt ein Verfahren, bei welchem das Verhalten fehlender Komponenten für das restliche System nachgebildet wird. Die hierfür bereitgestellten Modelle beschränken sich dabei zumeist auf die Simulation der Buskommunikation [Har01]. Nach [SS07] müssen diese Modelle bei der dynamischen Restbussimulation Echtzeitmodelle sein. Im Gegensatz zur statischen Restbussimulation werden dabei die Botschaften der nachgebildeten Komponente in Abhängigkeit der Umgebung generiert. Echtzeitmodelle werden meist mit CASE-Tools erstellt [Sch03].

Im Umfeld von Systemintegrationstests wird die Restbussimulation auch an Stelle realer Komponenten eingesetzt. Dies kann zum einen aus einem nicht ausreichenden Implementierungsstand der Steuergeräte-Software resultieren, meist in frühen Phasen eines Projekts. Zum anderen kann durch die Simulation bewusst falscher Botschaften das Verhalten des restlichen Systems geprüft werden.

Für die Konflikterkennung sind somit zwei Arten der Restbussimulation von Bedeutung

1. Simulation nicht real vorhandener Komponenten durch (Echtzeit-)Modelle

2. Simulation real vorhandener Komponenten durch das Testprogramm

6.4.1 Simulation nicht real vorhandener Komponenten

Konflikte zwischen zwei Testprogrammen entstehen immer dann, wenn durch die Deaktivierung einer Simulation Botschaften entfallen, welche von dem anderen Testprogramm erwartet werden. Daher muss bei der Simulation nicht real vorhandener Komponenten zwischen standardmäßig aktiven und deaktiven Modellen unterschieden werden. Das Beenden aktiver Modelle kann wie die Simulation real vorhandener Komponenten behandelt werden (vgl. 6.4.2, S. 81). In beiden Fällen erwarten die Testprogramme eine korrekte Buskommunikation der Komponente, ohne das hierfür besondere Einstellungen notwendig wären.
Anders verhält es sich bei standardmäßig deaktivierten Modellen. Diese werden durch das Testprogramm im Rahmen der Initialisierung aktiviert. Auch hierbei entstehen Konflikte nur, wenn durch ein Testprogramm die Simulation beendet wird, welche von dem anderen weiter erwartet wird. Voraussetzung hierfür ist allerdings, dass beide Testprogramme das gleiche Modell aktiviert haben. Somit würde das Deaktivieren der Simulation die Funktionsvoraussetzung des anderen Testprogramms beeinflussen. Da sowohl Aktivierung, als auch Deaktivierung von Modellen über Signale geschieht, wird ein derartiger Konflikt bereits durch (6.4) bzw. (6.5) aus 6.2.3 (S. 70) erkannt.

6.4.2 Simulation real vorhandener Komponenten

Die Simulation real vorhandener Komponenten zielt in der Regel auf die Manipulation weniger Botschaften beziehungsweise Signale ab. Dadurch kann der Beitrag des Steuergeräts auf die getestete Funktion beliebig gesteuert werden. Alle übrigen Botschaften, welche für die Durchführung anderer Funktionen nötig wären, werden jedoch nicht mehr auf dem Bus gesendet. Folgendes Beispiel 6.5 (S. 82) verdeutlicht einen Konflikt anhand der Simulation der Dachbedieneinheit. Diese sendet sowohl die Botschaft *DBE_SHD* (Schiebe-Hebedach), als auch *DBE_ILC* (Innenlicht).

Durch die Unterbrechung der Kommunikation zwischen Dachbedieneinheit und dem Bus wird unter anderem die Botschaft *DBE_ILC* nicht mehr empfangen. Testprogramm B führt somit zu keinem verwertbaren Ergebnis.

Testprogramm A	Testprogramm B
⋮	⋮
DBE vom CAN entfernen	⋮
Simulation der Botschaft *DBE_SHD*	Kontrolle der Botschaft *DBE_ILC*
⋮	⋮

Bsp. 6.5: Konflikt durch Restbussimulation

Beispiel 6.5 (S. 82) zeigt weiterhin die Probleme bei der Konflikterkennung auf. Beide Programme lassen sich zu ausführbaren Programmen erweitern, welche nach den bisherigen Regeln unabhängig wären. Der Konflikt kann erst dann erkannt werden, wenn der Einfluß der Kommunikationsunterbrechung auf die Botschaft *DBE_ILC*, und damit auch auf Testprogramm B transparent wird. Diese Verbindung kann aus der Kommunikationsmatrix (vgl. 4.1, S. 41) abgelesen werden. Wird eine Komponente im Laufe der Testausführung vom Bus getrennt, so muss die Menge D des Testprogramms um alle Signale, welche die Komponente auf dem Bus sendet, erweitert werden. Konflikte können somit nach den bisherigen Regeln, speziell (6.6) aus Abschnitt 6.2.3 (S. 70), erkannt werden.

6.5 Zusammenfassung

In diesem Kapitel wurden die Regeln für die Erkennung möglicher Konflikte zwischen zwei Testprogrammen definiert. Wichtigstes Hilfsmittel ist hierbei die Auswertung der vom Testprogramm verwendeten Signale. Diese werden für jedes Testprogramm, getrennt nach Initialisierung und Durchführung, in den Mengen I und D zusammengefasst. Die Menge D kann sich dabei noch aufgrund einer Restbussimulation erweitern. Grundsätzlich gilt, dass ein und dasselbe Signal nicht in beiden Programmen verwendet werden darf. Lediglich im Rahmen der Initialisierung können aufgrund von Synchronisation Ausnahmen zugelassen werden. Wichtiges Kritierum hierfür ist die Gleichwertigkeit der entsprechenden Signale.
Zusätzlich muss für jedes Testprogramm (ausgehend von der getesteten Funktion) die Menge A erstellt werden, welche die Abbruchbedingung der Funktion repräsentiert. Die Schnittmenge aus A und D muss ebenfalls leer sein.

Abschließend müssen Konflikte aufgrund der Kodierung der Steuergeräte erkannt werden. Die Erkennung basiert hierbei auf der Vorgabe weniger Basiskonfigurationen, welche auf klar definierte Weise durch das Testprogramm noch verändert werden kann. Dadurch bleibt die Vergleichbarkeit der Konfigurationen gewährleistet und ein Konflikt wird erkannt.

Kapitel 7

Optimierung

In diesem Kapitel wird ein Verfahren zur Erstellung einer optimalen Testsuite vorgestellt. Es basiert dabei auf den Ergebnissen aus Kapitel 6. Zunächst wird eine Konfliktmatrix aufgestellt. In dieser wird für je zwei Testprogramme angegeben, ob zwischen beiden ein Konflikt besteht oder ob eine parallele Ausführung möglich ist. Im weiteren Verlauf des Kapitels wird dann ein Algorithmus definiert, welcher aus einer gegebenen Anzahl an Testprogrammen eine zeitlich optimale Testsuite generiert. Die hierfür notwendigen Begriffe werden zu Beginn des Kapitels definiert.

7.1 Definitionen

Für eine optimale Auslastung des Testsystems und einer damit verbundenen maximalen zeitlichen Ersparnis sind neben den gegebenen Testprogrammen die Testsuite und Testslots von besonderen Interesse. Als Grundlage für deren Definitionen gilt dabei immer für die Menge aller Testprogramme folgende Bezeichnung

$$T = \{T_1, .., T_n\}$$

Definition: Testslot
Ein Testslot steht für die Gruppierung einer beliebigen Anzahl an Testprogrammen, welche parallel ausgeführt werden sollen. Die Testprogramme eines

Testslots sind paarweise unabhängig, d.h. zwischen zwei beliebigen Testprogrammen darf kein Konflitkt bestehen.

Da nicht zu jedem Testprogramm ein zweites, unabhängiges existieren muss, ist ein Testslot, bestehend aus lediglich einem Testprogramm zulässig. Der erste Testslot aus Abbildung 5.2 (S. 59) ist definiert durch

$$TS_1 = \{T_1, T_2, T_3\}$$

Eine Änderung der Testslots wirkt sich somit unmittelbar auf die Ausführdauer aus. Dagegen ist deren Reihenfolge bezüglich der erreichten Zeitersparnis ohne Bedeutung. Ist jedoch die Ausführdauer länger als die zur Verfügung stehende Testzeit, so darf die Abfolge der Testlots nicht unberücksichtigt bleiben. In diesem Fall erfolgt durch die Reihenfolge eine Priorisierung.

Definition: Testsuite
Eine Testsuite dient der Koordination der Testslots. Sie regelt zum einen deren zeitliche Abfolge. Zum anderen wird durch sie sichergestellt, dass alle Testprogramme für die Ausführung eingeplant sind. Somit gilt

$$\begin{aligned} TS &= \{TS_1, .., TS_m\},\ TS_i \text{ Testslot} \\ \bigcup_{i=1}^{m} TS_i &= T \end{aligned}$$

Zusätzlich kann durch eine Testsuite eine Mehrfache Ausführung eines Testprogramms unterbunden werden. In diesem Fall müssen die Testslots $TS_1, .., TS_n$ paarweise disjunkt sein. Abbildung 5.2 zeigt eine Testsuite. Nach obiger Notation wird diese wie in Beispiel 7.1 beschrieben

7.2 Konfliktmatrix

Eine Grundlage für die Erstellung einer optimalen Testsuite ist die Kenntnis über bestehenden Konflikte aller möglichen Testpaarungen. Da auch die

$$
\begin{array}{lcl}
T & = & \{T_1, T_2, T_3, T_4, T_5, T_6\} \\
TS & = & \{TS_1, TS_2, TS_3\} \\
TS_1 & = & \{T_1, T_2, T_3\} \\
TS_2 & = & \{T_4\} \\
TS_3 & = & \{T_5, T_6\}
\end{array}
$$

Bsp. 7.1: Beispiel einer Testsuite

Generierung von Testslots mit mehr als zwei Testprogrammen auf dem Vergleich jeweils zweier Programme zurückzugühren ist, eignet sich eine Matrix als Darstellung der notwendigen Informationen. In dieser stehen die Zeilen und Spalten jeweils für die Testprogramme. Der Eintrag der i-ten Zeile und der j-ten Spalte gibt dabei an, ob zwischen Testprogramm i und j ein Konflikt besteht (=1) oder ob sie unabhängig sind (=0).

$$
C = \begin{pmatrix}
0 & c_{12} & \cdots & c_{1n} \\
c_{21} & 0 & \cdots & c_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
c_{n1} & c_{n2} & \cdots & 0
\end{pmatrix}
$$

$$
\text{mit } c_{ij} = \begin{cases} 1 & \text{Konflikt zwischen Programm i und j} \\ 0 & \text{sonst} \end{cases}
$$

Die Diagonalelemente der Matrix geben hierbei das Verhältnis eines Testprogramms zu sich selber an. Da in diesem Fall kein Konflikt bestehen kann, sind die Diagonalelemete immer gleich 0. Zudem handelt es sich um eine symmetrische Matrix, da die Unabhängigkeit als Relation ebenfalls symmetrisch ist.

Eine Matrix mit oben beschriebenen Eigenschaften heißt Konfliktmatrix.

7.3 Bestimmung der Ausführzeit eines Testprogramms

Um eine zeitlich optimale Testsuite generieren zu können ist neben der Konfliktmatrix die Information über die Ausführungsdauer jedes Testprogramms grundlegend. Eine Methode einen verwertbaren Wert für die Dauer eines Testprogramms zu bestimmen ist ein Verfahren zur Ermittlung der *Worst Case Execution Time* (WCET). Hierbei kann beispielsweise der Programmfluss analysiert werden. Im Wesentlichen werden die Anweisungen des Testprogramms gezählt und mit ihrer Ausführdauer multipliziert. Über Pfadanalysen können die Auswirkungen von Schleifen, Verzweigungen, etc. berücksichtigt werden. Ziel ist die Schätzung einer Ausführdauer, welche stets größer als die reale Ausführdauer des Testprogramms ist.

Die Ausführdauer eines im Rahmen dieser Arbeit betrachteten Testprogramms lässt sich mit solchen Verfahren nicht ausreichend abschätzen. So kann etwa bei Schleifen oft nicht vorhergesagt werden, wie oft sie tatsächlich durchlaufen werden. Da sich die Ausführdauer des Testprogramms bei mehrfachen Ausführen typischerweise nicht wesentlich ändert, kann die Ausführdauer des letzten Laufs verwendet werden. Da jedes Testprogramm mindestens einmal im Vorfeld einer parallelen Ausführung gelaufen ist, beispielweise zur Inbetriebnahme des Tests, ist dieser Wert verfügbar. Im Folgenden wird die Ausführdauer des Testprogramms T_i mit t_i bezeichnet.

7.4 Kriterium für eine zeitlich optimale Testsuite

Die Ausführdauer einer Testsuite hängt zum einen von der Anzahl der Testslots, zum anderen von deren Länge ab. Die Länge eines Testslots wird durch das längste darin enthaltene Testprogramm festgelegt. Für die Ausführdauer einer Testsuite mit m Testslots ergibt sich somit

$$\text{Ausführdauer(TS)} = \sum_{i=1}^{m} \max_{T_j \in TS_i} t_j \tag{7.1}$$

Aufgrund möglicher Eingriffe in den Testablauf zwecks Synchronisation kann die tatsächliche Ausführdauer einer Testsuite von obiger Summe abweichen. Die Berücksichtigung der Synchronisation ist aber nicht möglich, so dass im weiteren Verlauf die Formel (7.1) als Grundlage zur Berechnung der Länge einer Testsuite verwendet wird.

Um eine Änderung der Ausführdauer zu erreichen, muss ein Testprogramm aus einem Testslot in einen anderen verschoben werden. Diese Verschiebung von T_i aus TS_ν nach TS_ω wirkt sich dann wie folgt auf die Ausführdauer der Testsuite aus.

$$\text{Verkürzung von } TS_\nu\text{: } \max(t_i - \max_{T_j \in\, TS_\nu \setminus \{T_i\}} t_j, 0)$$

$$\text{Verlängerung von } TS_\omega\text{: } \max(t_i - \max_{T_k \in\, TS_\omega} t_k, 0)$$

Hierbei muss sichergestellt sein, dass (T_i, T_k) unabhängig für alle $T_k \in TS_\omega$ sind. Somit lässt sich ein Kriterium für eine optimale Testsuite definieren.

Definition: optimale Testsuite
Eine Testuite heißt optimal, falls $\forall i, \nu, \omega$ gilt:

$$\max(t_i - \max_{T_j \in\, TS_\nu \setminus \{T_i\}} t_j, 0) \leq \max(t_i - \max_{T_k \in\, TS_\omega} t_k, 0) \tag{7.2}$$

$$(T_i, T_k) \text{ unabhängig für alle } T_k \in TS_\omega \tag{7.3}$$

Im folgenden Abschnitt wird ein Algorithmus definiert, anhand dessen eine optimale Testsuite bestimmt werden kann. Die Eingangsgrößen sind die Menge der Testprogramme (Testpool) sowie die Konfliktmatrix, die aus den Regeln aus Kapitel 6 resultiert.

7.5 Bestimmung einer optimalen Testsuite

Mit dem Testpool T und der Konfliktmatrix C stehen die notwendigen Eingangsgrößen für den Algorithmus zur Verfügung. Eine zusätzliche Bedingung

besteht darin, dass es keine Beschränkung der Anzahl parallel ausführbarer Testprogramme geben darf. Somit wird die Größe eines Testslots lediglich durch Konflikte der Testprogramme eingeschränkt. Dann läßt sich eine optimale Testsuite durch folgenden Algorithmus bestimmen.

1. $\nu = 1$
2. erstelle TS_ν
3. suche das Testprogramm $T_i \in T$ mit der längsten Ausführdauer
4. verschiebe T_i aus T nach TS_ν falls eine der folgenden Bedingungen erfüllt ist
   (a) $TS_\nu = \emptyset$
   (b) $c_{ij} = 0 \quad \forall T_j \in TS_\nu$ und keine Konflikte durch Deadlocks gemäß 6.2.2 entstehen
5. wiederhole die Schritte 3 und 4 $\forall T_j \in T' = T \setminus T_i$
6. falls $T \neq \emptyset$ erhöhe ν um 1 und gehe zu Schritt 2
7. $TS = \{TS_1, .., TS_\nu\}$

Die daraus resultierende Testsuite TS ist optimal.

7.5.1 Nachweis der Optimalität

Voraussetzung

$TS = TS_1, .., TS_m$ Testsuite bestimmt nach dem Algorithmus aus Abschnitt 7.5.

Behauptung

Die Testsuite TS ist optimal, d.h. $\forall i = 1, .., n, \ \nu, \omega = 1, .., m \ \ \nu \neq \omega$ gilt:

$$\max(t_i - \max_{T_j \in TS_\nu \setminus \{T_i\}} t_j, 0) \leq \max(t_i - \max_{T_k \in TS_\omega} t_k, 0) \tag{7.4}$$

Beweis

Annahme:
$\exists\ i, \nu, \omega$, so dass

$$\max(t_i - \max_{T_j \in TS_\nu \setminus \{T_i\}} t_j, 0) > \max(t_i - \max_{T_k \in TS_\omega} t_k, 0) \tag{7.5}$$

Sei $t_\nu = \max_{T_j \in TS_\nu \setminus \{T_i\}} t_j,\ \ t_\omega = \max_{T_k \in TS_\omega} t_k$

Fall 1: $t_i - t_\nu > 0$
$\Rightarrow t_\nu < t_i < t_\omega$
$\Rightarrow \omega < \nu$ nach Schritt 3 und 4
$\Rightarrow c_{il} = 1$ für ein $T_l \in TS_\omega$, ansonsten wäre nach Schritt 4 $T_i \in TS_\omega$
$\Rightarrow T_i$ kann nicht nach TS_ω verschoben werden

Fall 2: $t_i - t_\nu > t_i - t_\omega$
$\Rightarrow t_\nu < t_\omega$
$\Rightarrow \omega > \nu$ nach Schritt 3 und 4
$\Rightarrow c_{i\omega} = 1$ nach Schritt 4
$\Rightarrow T_i$ kann nicht nach TS_ω verschoben werden

Fall 3: $t_i < t_\nu$
$\Rightarrow$ die Ausführdauer von TS_ν wird nicht durch T_i festgelegt
$\Rightarrow$ Ausführdauer(TS_ν) = Ausführdauer$(TS_\nu \setminus \{T_i\})$
$\Rightarrow$ eine Verschiebung von T_i aus TS_ν nach TS_ω kann keine Verkürzung der Ausführdauer von TS bewirken

aus den Fällen 1-3 folgt, dass TS optimal ist

q.e.d.

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

Künftige Innovationen in der Automobilbranche werden maßgeblich durch vernetzte E/E Komponenten geprägt werden. In diesem Zusammenhang spielt der Test dieser Komponenten eine entscheidende Rolle. Nicht zuletzt durch immer kürzere Entwicklungszyklen der Steuergeräte ist bereits heute die Zeit der limitierende Faktor für die Testaktivitäten. Das in dieser Arbeit entwickelte Verfahren kann einen Beitrag zu effizienteren Ausnutzung der Zeit leisten und somit den Konflikt zwischen komplexen, qualitativ hochwertigen Produkten und einer verkürzten Entwicklungszeit entspannen.

Der Kernidee ist hierbei die Verkürzung der notwendigen Testausführungszeit durch das Ausführen mehrerer Testprogramme zur gleichen Zeit. Da nicht beliebige Tests parallel durchgeführt werden können, müssen Abhängigkeiten erkannt und somit Konflikte ausgeschlossen werden können. Ein automatisierter Vergleich zweier Testprogramme bildet hierfür die Grundlage. Anhand charakteristischer Eigenschaften der Testprogramme wurden Kriterien entwickelt, welche die Existenz von Konflikten aufzeigen bzw. unabhängige Testprogramme erkennen. Diese Kriterien gliedern sich grob in die Bereiche *Stimulation und Überprüfung, Kodierung* und *Restbussimulation*. Wichtigster Indikator für die Unabhängigkeit der Testprogramme ist hierbei die Verwendung unterschiedlicher Ressourcen. Diese werden in den meisten Fällen durch Signale repräsentiert, welche die Kommunikation zwischen den Steuergeräten regeln. Durch die Analyse der durch das Testpro-

gramm überwachten Kommunikation können Rückschlüße auf die Testinhalte gewonnen werden. Es gibt allerdings eine Reihe an Ressourcen, welche nicht exklusiv einem Testprogramm zugeordnet werden können, ohne damit die parallele Ausführung der Testprogramme praktisch auszuschließen. Dazu zählt beispielsweise die Spannungsversorgung des Testobjekts. Durch die Ausnutzung der Eigenschaften spezifischer Programmteile können daher trotz gemeinsamer Verwendung einer Ressource mehrere Testprogramme parallel ausgeführt werden. Dies erhöht die Zahl der möglichen Kombinationen von Testprogrammen und wirkt sich somit unmittelbar und positiv auf die gesamte Ausführdauer aus. Diese kann zusätzlich durch den Einsatz von Optimierungsverfahren verkürzt werden. Diese Arbeit enthält daher ein Verfahren, welches, unter Berücksichtigung der Konflikte, aus einer gegebenen Anzahl von Testprogrammen eine Testsuite mit minimaler Ausführdauer erzeugt.

Eine Parallelisierung der Testprogramme hat jedoch nicht nur Einfluß auf die Ausführdauer. Durch die Vermischung unterschiedlicher Startszenarien entstehen neue, realistischere Situationen. Somit wird implizit die Abwesenheit von Abhängigkeiten zwischen unabhängigen Funktionen geprüft. Da diese gelegentlich auftreten und keine logischen Zusammenhänge aufweisen, sind sie systemmatisch nicht auffindbar. Auch wenn ihre Anzahl gering ist und durch die Parallelsierung von Testprogrammen nur die Wahrscheinlichkeit für deren Erkennung gesteigert werden kann, ist die Erhöhung der Qualität der Testsuite ein nicht zu vernachlässigender Vorteil. Eine geeignte Unterstützung der Ergebnisprotokollierung und -auswertung ist jedoch eine notwendige Forderung, um diesen Vorteil auch praktisch umsetzen zu können. Da die Lokalisierung einer Fehlerursache speziell bei komplexeren Fehlern stark von der Erfahrung des Testers abhängt, kann die Unterstützung lediglich darin bestehen, die notwendigen Informationen aufbereitet in geeigneter Form zur Verfügung zu stellen.

8.2 Ausblick

In dieser Arbeit wurde gezeigt, dass eine Parallelisierung von Testprogrammen möglich ist, ohne zu starke Vorgaben für die Vorgehensweise bei der Testprogrammerstellung zu machen. Es wäre jedoch denkbar durch umfassendere Vorgaben den Grad der Parallelisierung von Testprogrammen zu stei-

gern. Hierzu zählt beispielsweise die Definition von Startszenarien analog zur Basiskonfiguration bei der Kodierung. Ein weiterer Ansatzpunkt zur Effizienzsteigerung sind Signale mit großem Wertebereich, der sich allerdings nicht in eine kleine Anzahl Untermengen gliedern lässt. Hierzu zählt etwa die Fahrzeuggeschwindigkeit. Diese ist jedoch gerade für Tests der Fahrdynamik von wesentlicher Bedeutung. Auch hier könnte die Lösung wiederum in der Vorgabe von verschiedenen Fahrsituationen bestehen. Für beide Ansätze gilt jedoch, dass die Bestimmung der Szenarien nicht trivial ist. So müssen sie doch einerseits in ihrer Anzahl begrenzt sein, so dass sich eine Steigerung der Effizienz zeigt. Andererseits darf darunter nicht die Qualität der Tests leiden. Es müssen also alle relevanten Situationen hergestellt werden können, wodurch die Anzahl der Szenarien ansteigt. Die entscheidende Frage ist demnach wie stark die Anzahl der betrachteten Situationen eingeschränkt werden darf, um die Ausführdauer der Testprogramme zu reduzieren.

Literaturverzeichnis

[B+97] Becker, Mario u. a. ; Daenzer, W. F. (Hrsg.) ; Huber, F. (Hrsg.): *Systems Engineering. Methodik und Praxis*. Verlag Industrielle Organisation, Zürich, 1997. – ISBN 3–85743–986–6

[BB99] Bergmann, Jochen ; Bender, Klaus: Funktionsprüfung eingebetteter Systeme der dezentralen Automatisierungstechnik. In: *at 7/99* (1999), S. 320ff

[BBK98] Broy, M. ; Beeck, M. von d. ; Krüger, I.: SOFTBED: Problemanalyse für das Großverbundprojekt „Systemtechnik Automobil - Software für eingebettete Systeme“. In: *Ausarbeitung für das BMBF*, 1998

[Bro07] Brockhaus: *Brockhaus in 15 Bänden*. `http://www.brockhaus-suche.de/suche/abstract.php?shortname=b15&artikel_id=20632100&verweis=1`, 2002-2007. – Online; Stand 7. Juni 2007

[BSS05] Bäro, Thomas ; Sax, Eric ; Schmerler, Stefan: Erhöhung der Testtiefe durch HiL-Testing. In: *Jahrestagung der ASIM/GI-Fachgruppe 4.5.5 „Simulation technischer Systeme“*, 2005, S. 4–13

[C+05] Cockx, Johan u. a.: *System and method for automatic parallelization of sequential code*. `http://www.freepatentsonline.com/20050188364.html`. Version: August 2005

[Deu97] Deutsches Institut für Normung: *DIN 69905: Projektwirtschaft - Projektabwicklung - Begriffe*, 1997

[E+94] Etschberger, Konrad u. a. ; Etschberger, Konrad (Hrsg.): *CAN Controller-Area-Network*. Carl Hanser Verlag München Wien, 1994. – ISBN 3–446–17596–2

[Eck86] Eckert, Claudia: Automatische Parallelisierung sequentieller FORTRAN-Programme fuer MIMD- Systeme. In: *Informatik-Berichte, Universitaet Bonn, Institut fuer Informatik* Bd. 54, 1986, S. 1–140

[F+07] Ferrandi, Fabrizio u. a.: Automatic Parallelization of Sequential Specifications for Symmetric MPSoCs. In: *IFIP International Federation for Information Processing - Embedded System Design: Topics, Techniques and Trends* Bd. 231. 2007. – ISBN 978–0–387–72257–3, S. 179–192

[Fle] FlexRay Consortium: *Website*. `www.flexray.com`, . – Online; Stand 25. September 2007

[FLS02] Frühauf, Karol ; Ludewig, Jochen ; Sandmayr, Helmut: *Software–Projektmanagement und –Qualitätssicherung*. Zürich: Vdf, Hochsch.-Verl. an der ETH, 2002. – ISBN 3–7281–2822–8

[Geh00] Gehrke, Thomas: *Dynamische Modelle für Reaktive Systeme mit Daten*, Technische Universität Braunschweig, Fachbereich für Mathematik und Informatik, Diss., 2000

[GR70] Gonzalez, M. J. ; Ramamoorthy, C. V.: Recognition and representation of parallel processable streams in computer programs-I (Sub-Task Parallelism). In: Hobbs, L.C. (Hrsg.): *Parallel Processor Systems, Technologies and Applications*, 1970, S. 335–373

[Gra06] Grass, Werner: *Vorlesungsskript: Schaltnetze und Schaltwerke (Rechensysteme A)*. Universität Passau, Wintersemester 2005/06

[Har64] Hartmann, Nicolai: *Der Aufbau der realen Welt*. de Gruyter, Berlin, 1964. – ISBN 978–3–11–000147–1

[Har01] Hartmann, Nico: *Automation des Tests eingebetteter Systeme am Beispiel der Kraftfahrzeugelektronik*, Universiät Karlsruhe, Diss., 2001

[HHS04] Honisch, Artur ; Hutter, Alexander ; Schmid, Hermann: Vollautomatisierter Test von Steuergeräte-Netzwerken. In: *Die neue Mercedes-Benz A-Klasse, Sonderausgabe von ATZ und MTZ* (Oktober 2004)

[HSZ96] Hackbusch, W. ; Schwarz, H. R. ; Zeidler, E.: *Teubner Taschenbuch der Mathematik*. B.G. Teubner Verlagsgesellschaft Leipzig, 1996. – ISBN 3–8154–2001–6

[Hut06] Hutter, Alexander: *Eine Systematik zur Erstellung virtueller Steuergeräte für Hardware-in-the-Loop-Integrationstests*, Technische Universität München, Lehrstuhl für Informationstechnik im Maschinenwesen, Diss., 2006

[INC04] INCOSE: *What is System Engineering?* `http://www.incose.org/practice/whatissystemseng.aspx`, 2004. – Online; Stand 24. Juli 2007

[Ins97] Institute of Electrical and Electronics Engineers: *IEEE Standard for Software Reviews. IEEE Std 1028-1997*. 1997

[Joc07] Jochim, Markus: Zeitig steuern - Sichere Datenübertragung im Automobil. In: *c't Magazin für Computertechnik* (Heft 2, 01.2007), S. 190–195

[Keß94] Kessler, Christoph W.: *Automatische Parallelisierung numerischer Programme durch Mustererkennung*, Universität des Saarlandes, Saarbrücken, Technische Fakultät, Diss., 1994

[Kie97] Kiencke, Uwe: *Ereignisdiskrete Systeme: Modellierung und Steuerung verteilter Systeme*. Oldenbourg Verlag, München, Wien, 1997. – ISBN 3–486–24150–8

[KK90] Kumar, Swarn P. ; Kowalik, Janusz S.: On the parallelization of sequential programs. In: *NATO ASI Series* Bd. F62, Springer-Verlag Berlin Heidelberg, 1990, S. 173–188

[Kop97] Kopetz, Hermann: *Real-Time Systems – Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997. – ISBN 0–7923–9894–7

[Krü90] Krüger, Michael: *Testen von Software als analytische Maßnahme der Software– Qualitätssicherung*, Universiät Ulm, Diss., 1990

[KRW05] Kiebusch, Sebastian ; Richter, Ernst ; Weiland, Jens: Metriken: Definition und Validierung. PESOA-Report No. 14/2005. (2005)

[Kus06] Kusche, Florian: *Eine Systematik zur E/E-Architekturunabhängigen Beschreibung von HIL-Echtzeit-Integrationstests*, Universiät Ulm, Fakultät für Mathematik und Wirtschaftswissenschaften, Diss., 2006

[LK] LIN-Konsortium: *LIN Specification Package*. `http://www.lin-subbus.org`, . – Online; Stand 12. September 2007

[Mar07] Marwedel, Peter: *Eingebettete Systeme*. Springer-Verlag Berlin Heidelberg, 2007. – ISBN 978–3–540–34048–5

[MB07a] Mercedes-Benz: *Betriebsanleitung C-Klasse*, 03/2007

[MB07b] Mercedes-Benz: *Betriebsanleitung CLK-Klasse*, 09/2007

[Mos] Most Cooperation: *Website*. `www.mostcooperation.com`, . – Online; Stand 25. September 2007

[OS04] Otterbach, Rainer ; Schütte, Frank: Effiziente Funktions- und Software-Entwicklung für mechatronische Systeme im Automobil. In: *Paderborner Workshop „Intelligente, mechatronische Systeme“*, 2004

[PGW02] Pitschinetz, Roman ; Grochtmann, Matthias ; Wegener, Joachim: *Test eingebetteter Systeme*. `http://www.systematic-testing.com/documents/overview_tes.pdf`, 2002. – Online; Stand 2. Juli 2007

[Rei06a] Reichart, Günter: Systemintegration – im Wechselspiel von Architektur, Technologie und Prozess. In: *10. Jahrestagung Eurofo-rum*, 2006

[Rei06b] Reimann, Reinhard: *Elektronik im Automobil – Segen oder Fluch?* `http://www.ba-mosbach.de/index.php?id=655`, 2006. – Online; Stand 21. Juni 2007

[RG69a] Ramamoorthy, C. V. ; Gonzalez, M. J.: Recognition and representation of parallel processable streams in computer programs-II (Task/process Parallelism). In: *Proceedings of the 1969 24th national conference*, 1969, S. 387–397

[RG69b] Ramamoorthy, C. V. ; Gonzalez, M. J.: A survey of techniques for recognizing parallel processable streams in computer programs. In: *Proc. AFIPS 1969 FJCC* Bd. 35, 1969, S. 1–15

[Rob07] Robert Bosch GmbH: *Lösungen für Umweltschutz und Verkehrssicherheit*. `http://www.bosch-presse.de/TBWebDB/de-DE/PressText.cfm?id=3314`, 2007. – [Online; Stand 17. August 2008]

[S+99] Schmid, Detlef u. a.: *Formale Verifikation eingebetteter System*. 1999

[Sax99] Sax, Eric: *Beitrag zur entwurfsbegleitenden Validierung und Verifikation elektronischer Mixed-Signal-Systeme*, Universität Karlsruhe, Diss., 1999

[Sch99] Schwartz, Naftali: *Automatic Parallelization: An Incremental, Optimistic, Practical Approach*, New York University, Department of Computer Science, Diss., 1999

[Sch02] Schweiggert, Franz: *Vorlesungsskript: Software Engineering Praxis (Imformationsverarbeitung für Aktuare)*. Universität Ulm, Sommersemester 2002

[Sch03] Schmid, Hermann: *Konzeption einer pragmatischen Testmethodik für den Test von eingebetteten Systemen*, Universiät Ulm, Fakultät für Mathematik und Wirtschaftswissenschaften, Diss., 2003

[Sch04] Schwartz, Leon: *Method for automatic parallelization of software*. `http://www.freepatentsonline.com/6708331.html`. Version: March 2004

[Sch07] Schweiggert, Franz: *Vorlesungsskript: Software- und System-Qualität*. Universität Ulm, 2007

[SL04] Spillner, Andreas ; Linz, Thilo: *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester*. dpunkt.verlag, Heidelberg, 2004. – ISBN 3–89864–256–9

[SS07] Stroop, Joachim ; Stolpe, Ralf: *FlexRay-Projekte leicht gemacht, Einsatz von modellbasierten Entwicklungswerkzeugen.* http://www.elektroniknet.de/home/automotive/technik-know-how/uebersicht/l/test-entwicklungstools/flexray-projekte-leicht-gemacht/, 2007. – [Online; Stand 24. April 2007]

[SZ06] Schäuffele, Jörg ; Zurawka, Thomas: *Automotive Software Engineering.* Wiesbaden : Vieweg, 2006. – ISBN 3–8348–0051–1

[Tab06] Tabeling, Peter: *Softwaresysteme und ihre Modellierung.* Springer-Verlag Berlin Heidelberg, 2006. – ISBN 978–3–540–25828–5

[UL07] Utting, Mark ; Legeard, Bruno: *Practical model-based testing : a tools approach.* Amsterdam : Elsevier, 2007. – ISBN 978–0–12–372501–1 ; 0–12–372501–1

[VDE07] VDE Verband der Elektrotechnik Elektronik Informationstechnik e.V.: *VDE: Mikroelektronik treibt die Automobiltechnik an.* http://www.vde.com/de/Verband/Pressecenter/Pressemeldungen/Fach-und-Wirtschaftspresse/Seiten/2007-50.aspx, 2007. – [Online; Stand 17. August 2008]

[WB05] Wörn, Heinz ; Brinkschulte, Uwe: *Echtzeitsysteme.* Springer-Verlag Berlin Heidelberg, 2005. – ISBN 978–3–540–20588–3

[Web98] Weber, Michael: *Verteilte Systeme.* Spektrum Akademischer Verlag, Heidelberg, 1998. – ISBN 3–8274–0221–2

[Wik06] Wikipedia: *Whiteboxtest — Wikipedia, Die freie Enzyklopädie.* http://de.wikipedia.org/w/index.php?title=Whiteboxtest&oldid=13493592, 2006. – [Online; Stand 3. September 2007]

[WWP04] Wolf, Marko ; Weimerskirch, André ; Paar, Christof: *Sicherheit in automobilen Bussystemen.* 2004

Abbildungsverzeichnis

Abkürzungsverzeichnis

Allgemeine Abkürzungen

AL	Automation Library
BNF	Backus-Naur-Form
CAN	Controller Area Network
CASE	Computer-Aided Software Engineering
CSMA/CA	Carrier Sense Multiple Access/Collision Avoidance
E/E	Elektrik/Elektronik
ECU	Electronic Control Unit
HIL	Hardware in the Loop
KB	Kilobyte
Kfz	Kraftfahrzeug
LIN	Local Interconnect Network
MB	Megabyte
MOST	Media Oriented Systems Transport
OEM	Original Equipment Manufacturer
PKW	Personenkraftwagen
RTAE	Real Time Automation Engine
SA	Sonderausstattung
SE	Systems Engineering
SG	Steuergerät
SGD	System dependence graph
SIL	Software-in-the-Loop
SUT	System under Test
WCET	Worst Case Execution Time

Steuergerätbezeichnungen

DAB ..... Digitalradio
DBE ..... Dachbedieneinheit
ESP ..... Elektronisches Stabilitätsprogramm
EZS ..... Elekronischer Zündstartschalter
KLA ..... Klimaanlage
KOMBI ..... Kombi Instrument
MRSM ..... Mantelrohrschaltermodul
MS ..... Motorsteuergerät
PTS ..... Parktroniksystem
RGS_L ..... Reversibler Gurtstraffer Links
SAM_H ..... Signalerfassungs- und Ansteuermodul hinten
SAM_V ..... Signalerfassungs- und Ansteuermodul vorne
SSP_FL ..... Schalterfeld Sitzverstellung / Zentralverrriegelungsinnentaster Türsteuergerät Vorne Links
Tel ..... Telefonmodul
TGW ..... Telematik Gateway
TSG_HL ..... Türsteuergerät hinten links
TSG_HR ..... Türsteuergerät hinten rechts
TSG_VL ..... Türsteuergerät vorne links
TSG_VR ..... Türsteuergerät vorne rechts
TV ..... TV-Tuner
WMSP_FD ..... Schalterfeld Fensterheber / Spiegelverstellung Türsteuergerät Fahrerseite
ZGW ..... Zentrales Gateway

Index