Diplomarbeit

Aufbau einer Modultestumgebung am Beispiel der Displayansteuerung für ein ACC-Fahrzeug

vorgelegt von
Amalinda Oertel
Januar 2006

Betreuer: Prof. Dr. Wolfgang Rosenstiel, Thomas Gorges

Arbeitsbereich Technische Informatik
Wilhelm-Schickard-Institut für Informatik
Fakultät für Informations- und Kognitionswissenschaften
Eberhard-Karls-Universität Tübingen

Erklärung zur Urheberschaft

Hiermit erkläre ich, dass ich den Inhalt dieser Diplomarbeit selbstständig verfasst und nur die angegebenen Quellen verwendet habe. Ferner wurde diese Arbeit noch keinem anderen Prüfungsamt vorgelegt.

Tübingen, 15.1.2006

Inhaltsverzeichnis

Kapitel 1

Wozu brauchen wir Modultest?

> There is no life today without software. The world would probably just collapse
>
> Frank Lanza

In immer mehr Bereichen unseres Lebens spielt Software eine große Rolle: in Bereichen wie der Raumfahrt, dem Flug- und Eisenbahnverkehr, militärischen Anwendungsgebieten, der Stromversorgung oder der Medizin wird immer mehr Software eingesetzt. Dies bringt immer einen großen Fortschritt, geht jedoch auch mit Problemen einher, denn die Folgen eines Programmfehlers können außerordentlich sein:

Ein Softwarefehler im Bereich der Medizin [LT93] kostete mindestens drei Menschen das Leben, andere Patienten wurden schwer verletzt. Therac-25 war ein Linearbeschleuniger zur Anwendung in der Strahlentherapie. Er wurde von 1982 bis 1985 in elf Exemplaren von der kanadischen Regierungsfirma Atomic Energy of Canada Limited (AECL) gebaut und in Kliniken in den USA und in Kanada installiert. Durch Softwarefehler und mangelnde Qualitätssicherung war ein schwerer Funktionsfehler möglich, der zu einer Überdosis führte. Diese Überdosis war in drei Fällen tödlich, andere Patienten erfuhren Lähmungen oder Organe mussten transplantiert werden [LT93].

1962 führte im Bereich der Raumfahrt ein fehlender Trennstrich in dem Steuerprogramm der Atlas-Agena zum Verlust der Venus-Sonde Mariner 1. Die Rakete im Wert von 18,5 Millionen US$ musste vier Minuten nach dem Start zerstört werden [Neu95].

1996 wurde der Prototyp der Ariane 5-Rakete der Europäischen Raumfahrtbehörde eine Minute nach dem Start zerstört, weil Programmcode, der von der Ariane 4 übernommen wurde und nur für einen (von der Ariane 4 nicht überschreitbaren) Bereich funktionierte, die Steuersysteme zum Erlie-

gen brachte, nachdem eben dieser Bereich von der Ariane 5 überschritten wurde [Rep96, Lan96].

Dies sind nur einige Beispiele für Softwarefehler mit gravierenden Auswirkungen, weitere Beispiele finden sich beispielsweise in [Neu95, Gle96]. Natürlich hat nicht jeder Softwarefehler so schlimme Auswirkungen - zum Glück, wenn man bedenkt, dass kommerzielle Software schätzungsweise zwischen drei und 25 Fehler pro 1000 Zeilen Programmtext aufweist [Gib94]. Doch gerade bei sehr kostenintensiven Anwendungen wie der Raumfahrt oder auch bei sicherheitskritischen Anwendungen aus der Medizin ist es wichtig, solche Fehler so gut wie möglich auszuschließen. Zwar ist es im Allgemeinen nicht möglich zu beweisen, dass eine Software fehlerfrei läuft. Aber durch hohen Testaufwand kann zumindest ein Teil der Fehler schon in der Entwicklung gefunden und behoben werden - und die Wahrscheinlichkeit einen derart schweren Softwarefehler übersehen zu haben wird ebenfalls stark verringert [Mye82].

Auch in der Automobilindustrie wird immer mehr Software eingesetzt, und die Elektronik nimmt einen immer größeren Anteil des Automobils ein. Elektronik, die fehleranfällig ist: Schon 1998 waren 45,2% aller Pannen auf Probleme mit der Elektronik und Elektrik zurückzuführen - bis zum Jahr 2010 soll der Anteil sogar auf 66% steigen [Hau03, Gre03]. Trotzdem sehen 60% der Autofahrer Vorteile durch die Anwendung von elektronischen Helfern[Spi04].

Ein Teil der Elektronik wird für Komfortsysteme entwickelt, wie beispielsweise Serviceanzeigen, Einparkhilfen oder Adaptive Cruise Control-(ACC). Komfortsysteme sind reine Luxusgegenstände, aber Bosch beispielsweise plant, das ACC-System langfristig mit sicherheitsrelevanten Funktionen auszurüsten. So soll das ACC-System in der letzten Ausbaustufe im Fall, dass ein Auffahrunfall unvermeidlich ist, eine Notbremsung auslösen, um so kostbare Sekunden zu gewinnen, die über einen tödlichen Ausgang des Unfalls entscheiden können. Auch andere Anbieter ähnlicher Systeme planen die Entwicklung solcher Systeme.

Doch wenn man dem Auto die Möglichkeit geben will, aus eigener Entscheidung eine Vollbremsung auszuführen, so bedarf es eines sehr großen Vertrauens in die Software. Ein Fall, in dem das Auto fehlerhaft auf der Autobahn eine Vollbremsung ausführt, könnte für den Fahrer des Wagens tödlich sein. Ein System, das entwickelt wurde, Unfälle so glimpflich wie möglich ausgehen zu lassen und eventuell Leben zu retten, könnte bei einem Auftreten eines Fehlers selbst zum Unfallverursacher werden. Nun kann die Möglichkeit eines Fehlers nie ganz ausgeschlossen werden, ein Restrisiko bleibt. Aber durch ein geschicktes Qualitätsmanagement, zu dem auch ausführliches und sinnvolles Testen gehört, kann das Risiko soweit erniedrigt werden, dass es hinnehmbar ist. Aus diesem Grund lohnt es sich, in diesen Bereich zu investieren.

Weiterhin kommt zu der moralischen Verantwortung, die ein Hersteller

sicherheitskritischer Anwendungen seinem Käufer gegenüber eingeht, auch ein großer Kostenfaktor. Sollte es tatsächlich zu einer Rückrufaktion kommen, so sind die dafür anfallenden Kosten immens. Selbst das Bekanntwerden geringerer Fehler kann zu einem großen Imageverlust der Firma führen und das Vertrauen in die Marke schwächen. Dies könnte sich langfristig negativ auf die Verkaufszahlen auswirken und somit die Firma schädigen.

Ausführliche Testreihen als Teil eines guten Qualitätsmanagements - beispielsweise nach der ISO-Norm 9126 [91201] - können dieses Risiko erniedrigen. Und sollte es doch zu einem Fehler kommen, so kann das Testmaterial als rechtliche Absicherung verwendet werden, das beweisen kann, dass die Firma eben nicht leichtfertig mit dem Risiko umgegangen ist, sondern sich verantwortungsvoll bemüht hat, die Software so gut es ging auf Korrektheit zu prüfen.

Ein verantwortungsvolles Testmanagement ist somit gerade für Hersteller sicherheitskritischer Anwendungen unabdingbar.

In dieser Arbeit wird gezeigt, dass nicht jeder Testfall gleich erfolgreich ist. Es gibt Methoden, die mit einer höheren Wahrscheinlichkeit Fehler aufdecken als andere. Bei einem guten Testmanagement muß daher zunächst geprüft werden, welche Methoden bei der speziellen Software am geeignetsten sind.

Der Modultest stellt eine Teststufe in einem Testprozess dar. Diese Testart kann bereits sehr früh im Entwicklungsprozess eingesetzt werden, da die Module einzeln getestet werden sollen, und somit nicht auf die Fertigstellung des gesamten Systems gewartet werden muß. Unter Modulen verstehen wir in dieser Arbeit c-Files, die wiederum aus verschiedenen Funktionen aufgebaut sein können. Eine Komponente hingegen besteht aus mehreren c-files. Modultest ist somit Softwaretest, bei dem die Funktionalität dieser c-Files im Vordergrund steht. Beim Komponententest dagegen werden mehrere c-files gemeinsam getestet, hier spielt also auch das Zusammenwirken der einzelnen Module eine Rolle.

Der Modultest kann als Fundament höherer Testmethoden fungieren. Es ist sinnvoll, einen Modultest durchzuführen, da die Behebung von Fehlern immer teurer wird je weiter das Projekt bereits fortgeschritten ist [Mye82]. Durch den Modultest können Fehler bereits in einem frühen Entwicklungsstadium gefunden werden, und somit noch verhältnismäßig kostengünstig behoben werden. Weiterhin ist es im Modultest einfacher Fehler zu finden, falls eine Fehlerwirkung auftritt: Da nur ein Modul getestet wird, kann sich der Fehler auch nur in diesem Modul finden, somit ist der Suchraum also stark eingeschränkt. Gerade im Vergleich zum Systemtest kann dies eine große Zeitersparnis bedeuten. Somit ist es kaum verwunderlich, dass der Modultest bei einer guten Qualitätssicherung vorausgesetzt wird.

Diese Diplomarbeit soll einen Anhaltspunkt geben, wie Module im Automobil Bereich sinnvoll getestet werden können, welche Methoden hier besonders vielversprechend sind und wie der Testprozess in das vorhandene

Qualitätsmanagement integriert werden kann. Die Arbeit entstand als Zusammenarbeit der Universität Tübingen mit der Abteilung Fahrerassistenzsysteme der Firma Bosch.

Unsere Aufgabe bestand darin, am Beispiel der Adaptive Cruise Control(ACC) Software eine Modultestumgebung mit dem Programm Rational Test RealTime aufzubauen und verschiedene Testmethoden zu evaluieren. Dabei verfolgten wir vier Ziele: Zunächst wurde geprüft, ob das Programm RTRT für den Modultest geeignet ist, welches Vorgehen sich damit anbietet und ob es gewisse Einschränkungen der Software nach sich zieht. Weiterhin sollte überprüft werden, ob es sich ebensogut für den Komponententest eignet und wo die Vor- und Nachteile der beiden Methoden liegen. Danach wurden verschiedene Testmethoden evaluiert, und geprüft, welche Methoden sich für die ACC-Software als besonders effizient erweisen. Das vierte Ziel bestand in der Entwicklung einer Softwaremetrik, welche sowohl für die Testplanung des Modultests wie auch als Testendekriterium eingesetzt werden kann, und die sich wieder besonders gut an die Ansprüche der ACC-Software anpasst. Die Ergebnisse unserer Arbeit finden sich in Kapitel 5, 6 und 7.

Wir haben im Rahmen dieser Diplomarbeit Programmtext der Module der ACC-Software, welche für die Displayansteuerung zuständig sind, mit verschiedenen Testmethoden getestet, um aus den Ergebnissen eine sinnvolle Teststrategie abzuleiten. Zunächst soll daher in Kapitel 2 das ACC-System genauer beschrieben werden - einerseits allgemein, wofür das System als Ganzes bisher eingesetzt wird, und welche Ausbaustufen noch geplant sind, andererseits speziell auf das Display bezogen, da wir Beispiele der Displayanzeigen verwenden. Auch die relevanten Aspekte der Softwareentwicklung werden in diesem Kapitel erläutert.

Danach folgt in Kapitel 3 eine kurze Zusammenfassung der Grundlagen des Softwaretestens. Es werden verschiedene Begriffe wie “Validierung”, “Verifikation”, “Fehler” und “Fehlerwirkung” [SL04] definiert, und Myers [Mye82] grundlegende Testprinzipien werden vorgestellt.

Aufbauend darauf wird in Kapitel 4 der Testprozess in den Softwarelebenszyklus eingegliedert. Hierfür wird zunächst das allgemeine V-Modell vorgestellt, gefolgt von einer Erläuterung der grundlegenden Teststufen sowie des fundamentalen Testprozesses. Danach werden die wichtigsten Testmethoden vorgestellt, sowie ein Vorgehen nach den Methoden an ausgewählten Beispielen verdeutlicht.

Die Evaluierung des Programms RTRT findet sich in Kapitel 5. Ein besonderes Augenmerk liegt hierbei darauf, zu prüfen welche Hilfen das Tool leistet, für welche Aufgaben es sich eignet und für welche Aufgaben es weniger geeignet ist. Weiterhin wird in Anhang A exemplarisch gezeigt, wie das Testvorgehen nach den verschiedenen Methoden mit dem Tool umgesetzt werden kann.

In Kapitel 6 wird unsere Studie bezüglich der Testmethoden vorgestellt:

Wir haben in dieser Arbeit verschiedene c-Funktionen nach unterschiedlichen Methoden getestet. Uns interessierte, welche Methoden welche Fehlerarten entdecken würden und welche Verfahren besonders effizient sind. Aus diesen Ergebnissen wurde dann eine Teststrategie abgeleitet. Diese wurde anhand von Tests dreier weiterer c-Funktionen überprüft und für gut befunden.

Der zweite Teil unserer Studie befasste sich mit dem Thema, nach welchen Kriterien Tests geplant und wann Tests beendet werden können. Hierzu entwickelten wir eine Testmetrik von Fenton [FN04] weiter und überprüften sie mit unseren Ergebnissen aus Kapitel 6. Im Vordergrund stand hier, eine Metrik zu entwickeln, die, angepasst auf die Beispielsoftware, gute Vorhersagen liefern kann, wie viele Fehler sich in einem Modul befinden. Anhand dieser Vorhersagen kann dann geplant werden, welche Module besonders intensiv getestet werden müssen, und auch das Testende kann in Abstimmung mit diesen Ergebnissen erfolgen. Eine genaue Beschreibung verschiedener Metriken, sowie unserer entwickelten Metrik und deren Überprüfung findet sich in Kapitel 7.

In Kapitel 8 und 9 werden die wichtigsten Ergebnisse zusammengefasst und es werden Empfehlungen ausgesprochen, wie ein erfolgreiches Testvorgehen aussehen sollte.

Kapitel 2

Vorstellung der ACC-Displayansteuerung

Die Displayansteuerungskomponente ist ein Teil des "Adaptive Cruise Control(ACC)" Projekts. Zunächst beschreiben wir das Gesamtprojekt, bevor wir auf den Aufbau und die Rahmendaten der Displayansteuerung eingehen.

2.1 Vorstellung von Adaptive Cruise Control

"Adaptive Cruise Control(ACC)" ist eine automatische Abstandsregelung, die die Geschwindigkeit des eigenen Fahrzeugs automatisch an den Verkehrsfluss anpasst und einen Sicherheitsabstand zum voraus fahrenden Fahrzeug hält.

Hierfür muss der Fahrer zunächst einen gewünschten Abstand zu voraus fahrenden Fahrzeugen sowie die gewünschte Geschwindigkeit definieren. ACC erfasst dann die aktuelle Verkehrssituation mit Hilfe von Radarstrahlen. Aus der Laufzeit der reflektierten Signale berechnet ACC die Entfernung, Richtung und Relativgeschwindigkeit der voraus fahrenden Fahrzeuge. Hinter einem langsamer fahrenden Auto bremst ACC automatisch ab und hält den vom Fahrer definierten Abstand. Sobald die Fahrspur wieder frei ist oder falls das voraus fahrende Fahrzeug wieder schneller wird, beschleunigt ACC auf die vom Fahrer eingestellte Geschwindigkeit und hält diese. Eine Anzeige im Cockpit informiert über die Einstellungen und den aktuellen Betriebszustand.

Das gegenwärtige ACC System: Anfang 2000 ging die erste ACC-Generation von Bosch in Serie. Diese Generation war vornehmlich für Autobahnfahrten bestimmt. Die heutige Version arbeitet in einem Geschwindigkeitsbereich zwischen 30km/h und 200km/h und ist bereits auf Landstraßen und Autobahnen einsetzbar. In der nächsten Stufe soll auch das Staufolgefahren mit ACC ermöglicht werden: Das Fahrzeug wird dann durch ACC

bis in den Stillstand abgebremst und bei freier Fahrt wieder beschleunigt.

Nach heutigen Stand ist das System ein reines Komfortsystem. Es soll den Fahrer bei Routinevorgängen entlasten; Gerade im dichten Autobahnverkehr ändert sich der Verkehrsfluß ständig. In dieser Situation muss der Fahrer dauerhaft hoch konzentriert fahren, um die Geschwindigkeit an den Verkehrsfluss anzupassen. ACC nimmt dem Fahrer diese Aufgabe ab, und entlastet ihn auf diese Weise.

Das ACC System der Zukunft: Auf lange Sicht soll das System auch als Sicherheitssystem Anwendung finden. Innerhalb des "Predictive Safety Systems" wird bereits an einer Fahrerunterstützung in unfallkritischen Situationen gearbeitet. Bei diesen Sicherheitssystemen wirken die aktiven Sicherheitssysteme, wie das Antiblockiersystem(ABS), das Elektronische Stabilitätsprogramm(EPS) sowie der Hydraulische Bremsassistent(HBA), mit ACC zusammen. Auf diese Weise soll eine schnellere Reaktion auf Gefahrensituationen vor dem Fahrzeug ermöglicht werden, um damit die Zahl der Verkehrsunfälle zu reduzieren und das Verletzungsrisiko zu vermindern.

Bosch plant die "Predictive Safety Systems" in drei Stufen auszubauen: Zunächst wird der "Predictive Brake Assist" auf den Markt kommen. Dieses Produkt wird aus Daten der Adaptive Cruise Control Systems unfallkritische Situationen erkennen, in denen eine bevorstehende Notbremsung wahrscheinlich ist. In solchen Fällen wird das System die Bremskreise vorbefüllen und die Bremsbeläge unmerklich an die Bremsscheiben anlegen. Auf diese Weise wird die volle Bremswirkung deutlich früher erreicht, und somit der Anhalteweg drastisch verkürzt.

In der nächsten Ausbaustufe, der "Predictive Collision Warning", wird der Fahrer der Fahrer durch einen kurzen, spürbaren Bremsruck oder ein kurzes Anziehen der Sicherheitsgurte in kritischen Situationen alarmiert. Die frühzeitige Warnung soll dem Fahrer ermöglichen, schneller auf die Gefahr zu reagieren.

In der dritten Ausbaustufe, dem "Predictive Emergency Braking", soll dann zusätzlich zu den ersten beiden Funktionen auch eine automatische Notbremsung zur Verfügung stehen. Diese soll dann ausgeführt werden, wenn ein Auffahrunfall nicht mehr vermieden werden kann. Auf diese Weise soll die Aufprallgeschwindigkeit so weit wie möglich reduziert und somit das Verletzungsrisiko vermindert werden.

Nachdem nun das Gesamtprojekt vorgestellt wurde, folgt nun eine Einordnung der Displayansteuerung in das Gesamtprojekt.

2.2 Vorstellung der Displayansteuerung

In diesem Abschnitt werden zunächst die wichtigsten ACC-Anzeigen des Displays kurz vorgestellt. Danach folgt eine Einordnung der Komponente

aus Sicht der Softwareentwicklung in das Gesamtprojekt.

2.2.1 Vorstellung der wichtigsten ACC-Anzeigen

Die optische Anzeige im Kombiinstrument ist in drei Anzeigebereiche unterteilt:

- **Primäranzeige:** Die Primäranzeigen werden im Ziffernblatt des Tachos dargestellt. Dies sind Anzeigen, die dem Fahrer ständig zu Verfügung stehen. Ein Beispiel hierfür ist das Signal, ob ein voraus fahrendes Fahrzeug als Zielfahrzeug erkannt wurde. Die Primäranzeige ist immer aktiv, sofern der ACC-Hauptschalter nicht gerastet ausgeschaltet ist oder sich das System im Standby Modus befindet.

- **Temporäre Primäranzeige(TPA):** In der Temporären Primäranzeige werden wichtige Informationen, die selten auftreten und daher nicht ständig angezeigt werden sollen, angezeigt. Ein Beispiel hierfür ist beispielsweise eine Meldung, falls ACC defekt sein sollte.

- **Zusatzanzeige:** In der Zusatzanzeige werden Informationen dargestellt, die die Systemfunktion näher erklären und somit zum Systemverständnis des Fahrers beitragen sollen. Diese Anzeige erfolgt nicht automatisch, sondern muss vom Fahrer aktiviert werden.

Im Folgenden sollen die wichtigsten Anzeigen kurz vorgestellt werden. Da wir in dieser Arbeit keine Beispiele aus der Zusatzanzeige oder der Temporären Primäranzeige verwenden, beschränken wir uns hier auf die Beschreibung der wichtigsten Primäranzeigen.

Die wichtigsten Anzeigen der Primäranzeige: Die vier wichtigsten Elemente der Primäranzeige sind die Anzeige der Wunschgeschwindigkeit sowie der ACC-Aktivanzeige, die Anzeige des “Relevantes-Objekt-erkannt”-Symbols und der Fahrerübernahmeaufforderung.

Die Wunschgeschwindigkeit wird vom Fahrer am Bedienhebel eingestellt und im Tachokranz mit einer LED angezeigt. Ist ACC aktiv, so leuchtet das ACC-Symbol grün. Wird, während ACC aktiv ist, ein voraus fahrendes Fahrzeug erkannt, so leuchtet zusätzlich das “Relevantes-Objekt-erkannt”-Symbol grün auf. Sollte der Fall eintreten, dass ein Hindernis auftaucht und die Verzögerungsleistung des ACC-Systems nicht mehr ausreicht, so wird das ACC-Symbol rot blinkend als “Fahrerübernahmeaufforderung” angesteuert. Dies soll den Fahrer dazu auffordern, die Fahrzeugführung wieder komplett zu übernehmen.

In diesem Abschnitt wurden die wichtigsten Primär-Anzeigen des Displays kurz vorgestellt, da einige davon als Beispiele zur Verdeutlichung von

Definitionen, Methoden oder Ergebnissen in den folgenden Kapiteln herangezogen werden. Im folgenden Abschnitt wird aus Sicht der Softwareentwicklung erläutert, wie sich die Komponente in das Gesamtprojekt eingliedert.

2.2.2 Relevante Aspekte der Softwareentwicklung

Im vorigen Abschnitt wurden die wichtigsten Anzeigen der Primäranzeige beschrieben, da diese in folgenden Beispielen wieder verwendet werden.

In diesem Teil werden allgemeine Hintergrundinformationen zu dem bestehenden Softwareentwicklungsprozess des ACC-Projekts gegeben, damit die Ergebnisse unserer Arbeit in diesen Kontext eingeordnet werden können. Hierfür sind einerseits die relevanten Informationen zum Programmtext selbst, und andererseits Informationen zu der ToolChain des Testprozesses von Interesse. Diese Informationen werden in den folgenden zwei Abschnitten behandelt.

Allgemeine Hintergrunddaten des Projekts Im Rahmen dieser Diplomarbeit haben wir Module der Displayansteuerung von ACC getestet, um auf Basis dieser Ergebnisse ein sinnvolles Testvorgehen abzuleiten. Um eine Bewertung der Ergebnisse auch für andere Bereiche zu ermöglichen, werden in diesem Abschnitt allgemeine Informationen zu dem Projekt bereitgestellt.

Testen ist solange ökonomisch sinnvoll, wie die Kosten für das Finden und Beseitigen der Fehler geringer ist als die Kosten, die mit dem Auftreten des Fehlers verbunden sind [PKS00]. Wie genau ein Produkt getestet werden muß, hängt somit auch von dessen Sicherheitsrisiko ab: Ein Textverarbeitungsprogramm kann viele Fehler enthalten, und trotzdem nützlich sein - ein medizinisches System hingegen sollte möglichst fehlerfrei funktionieren. Im heutigen Stand ist ACC ein reines Komfortsystem, so dass eine Fehlfunktion keine tödlichen Auswirkungen nach sich ziehen könnte. In der letzten Ausbaustufe zum predictive safety system könnte eine Fehlfunktion jedoch zu einer fälschlicherweise ausgeführten Vollbremsung führen, welche im schlimmsten Fall zu einer Massenkarambolage führen könnte. Wir haben daher schon jetzt höchste Sicherheitsansprüche an die Software gestellt.

Die Komponente, die für die Displayansteuerung zuständig ist, ist nicht so sicherheitskritisch: sie würde im schlimmsten Fall falsche Informationen liefern oder sich selbst abschalten und könnte den Fahrer so verunsichern. Daher hätte hier eine geringere Testüberdeckung ausgereicht. Da wir allerdings unsere Methodik anhand dieser Komponente entwickelt haben, behandelten wir die Displayansteuerung trotzdem wie einen höchst sicherheitskritischen Bestandteil der Software.

Unsere Methodik wurde aufgrund des Rahmenprojekts für den Zweck entwickelt, sehr hohen Sicherheitsansprüchen zu genügen. Bei der Übertragung dieses Vorgehens auf andere Systeme muß dies bedacht werden -

bei einem weniger sicherheitskritischen System kann eine andere Methodik ökonomisch sinnvoller sein.

Die ACC-Software ist in C geschrieben und wird mit dem Compiler Texas Instruments(TI) auf der Plattform ERCOSEK, einem dem Betriebssystem OSEK ähnlichen System, entwickelt. Da auf der Plattform keine "floating point" Operationen unterstützt werden, sind diese Funktionen in Software realisiert. Daraus folgt eine feste Bindung des Systems an einen 16Bit Prozessor.

Eine weitere Eigenschaft unserer Software, die die Übertragbarkeit unserer Studie auf andere C-Systeme beeinträchtigen könnte, ist, dass unser System ein Embedded-System ist. Für das ACC benötigt das Fahrzeug eine Sensor Control Unit(SCU), die sowohl das eigentliche Steuergerät wie auch den Radarsensor enthält. Die ACC-Software läuft auf dem Steuergerät.

RTRT bietet die Möglichkeit, sowohl auf dem Device selbst wie auch auf einem PC zu testen. Wir haben uns im Rahmen dieser Diplomarbeit nur mit zweiterer Möglichkeit beschäftigt.

Das ACC-Projekt als Ganzes umfasst etwa 300000 Zeilen Programmtext, davon nimmt die Displayansteuerung etwa 4000 Zeilen in Anspruch.

Wir denken, dass die entwickelte Testmethodik auch für andere C-Systeme von Interesse sein kann.

Eingliederung des Projekts in die Test-Toolchain In diesem Abschnitt wird beschrieben, in welcher Reihenfolge die Testaktivitäten bei ACC bisher verlaufen, und welche Werkzeuge hierfür verwendet werden. Ein besonderes Augenmerk liegt hierbei auf den Fehlerarten, die in den einzelnen Testphasen gefunden werden sollen.

Die Softwareentwicklung bei Bosch verläuft nach dem V-Modell. Dieses Modell wird in Kapitel 4.1 erläutert. In Abbildung 2.1 ist der gegenwärtige Ablauf der eingesetzten Tools in das V-Diagramm eingetragen. Direkt nach der Programmierung der Module beginnt der Modultest. In dieser Testphase werden drei Werkzeuge eingesetzt: "Polyspace" [Pol05], "QA-C" [QA 05] und "Rational Test RealTime".

Beschreibung der verwendeten Tools in der Modul-Testphase: In der Modul-Testphase werden die Module mit Polyspace und QA-C geprüft, darauf folgt ein Funktionstest mit RTRT.

Die ersten zwei Werkzeuge benötigen keinerlei Testcases sondern arbeiten ausschließlich durch Analyse des Programmtexts, RTRT hingegen benötigt die manuelle Erstellung von Testcases.

Das Programm Polyspace ist gegenwärtig eines der mächtigsten Werkzeuge auf dem Markt zur Auffindung von Laufzeit-Fehlern. Es ist ein statisches Programmtext-Analysetool, das basierend auf der Methode der "Abstrakten Interpretation" versucht, Laufzeitfehler in der zu untersuchenden

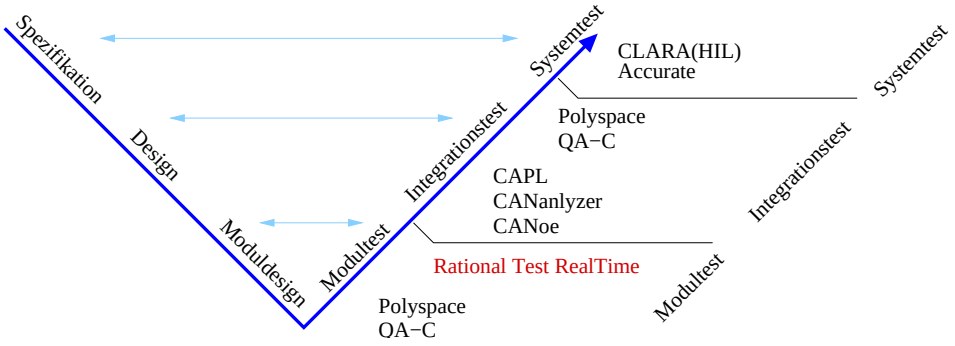

Abbildung 2.1: *Diese Abbildung veranschaulicht den Ablauf der Testaktivitäten im ACC-Entwicklungsprozess. Bevor mit dem Funktionstest mit RTRT begonnen wird, werden die Module mit QA-C und Polyspace überprüft. Für den Integrationstest werden CANalyser, CANoe und CAPL [Vec05] verwendet, gefolgt von einer weiteren Analyse durch Polyspace und QA-C. Der Systemtest erfolgt mit CLARA, einem Hardware-in-the-Loop Werkzeug sowie mit Accurate.*

Software zu finden.

In der ACC-Entwicklungsabteilung wird das Programm verwendet, um folgende Fehler zu finden:

- Out-of-Bound Zugriffe
- Arithmetische Ausnahmefehler (z.B. Division durch 0)
- Overflow/Underflow
- Nicht initialisierte Rückgabewerte
- Nicht initialisierte Variablen (lokale sowie globale)
- Nicht initialisierte Pointer
- Endlosschleifen
- Dereferenzierung durch Null- oder Out-of-Bounds-Pointers
- Dead Code

Während Polyspace zum Auffinden von Laufzeitfehlern eingesetzt wird, soll das Programm QA-C die Einhaltung von vordefinierten Coding-Rules überprüfen [QA 05]. Diese Regeln ergeben sich aus dem MISRA-C(Motor Industry Software Reliability Association) [MIS05] Standard, wurden aber für das Softwareprojekt erweitert. MISRA-C ist ein C-Programmierstandard

aus der Automobilindustrie. Durch die Vorgabe verschiedener Programmierregeln sollen gängige Softwarefehler vermieden werden.

Auch PolySpace bietet die Möglichkeit, den Programmtext auf die Einhaltung der MISRA-Standards zu überprüfen. Jedoch können hier die Kodierregeln noch nicht erweitert werden. Aus diesem Grund wird QA-C dazu verwendet, die Einhaltung der Coding-Rules zu überwachen.

Nachdem das Modul durch Polyspace auf Laufzeitfehler und durch QA-C auf die Einhaltung der Kodierregeln geprüft wurde, folgt nun eine Überprüfung der Funktion des Moduls. Hierzu wird das Programm Rational Test RealTime (RTRT) verwendet. Es analysiert das zu testende Modul und generiert einen passenden Testrahmen, der die benötigten Testtreiber und Platzhalter für die externen Funktionen bereitstellt. Um die Funktion des Moduls zu überprüfen, müssen in Testcases die Randbedingungen sowie die Eingaben und die erwarteten Ausgaben definiert werden. Das Tool führt dann den Programmtext unter diesen Bedingungen aus und überprüft, ob das erwartete Ergebnis dem tatsächlichen entspricht.

Auf diese Weise wird der Programmtext nach folgenden Fehlerarten geprüft:

- Funktionsfehler
- Fehlende Fehlerbehandlung
- Nicht mehr benötigter Programmtext
- Fehlende Requirements

Weiterhin bietet das Programm gute Werkzeuge zur Testprotokollierung sowie zur Testanalyse hinsichtlich der erreichten Programmtext-Überdeckung. Auch eine Laufzeitanalyse wird angeboten, die beispielsweise Memoryleaks aufdecken kann. Da solche Laufzeitfehler aber bereits durch Polyspace aufgefunden werden sollen, wird diese Funktion des Tools in diesem Rahmen nicht genutzt.

Auch Dead Programmtext kann sowohl von Polyspace gefunden werden wie auch mit Rational Test RealTime entdeckt werden. Da Dead Programmtext aber die Überdeckungsanalysen der Tests mit RTRT beeinträchtigt - eine 100%ige Programmtext-Abdeckung ist unmöglich, falls Dead Programmtext vorliegt - sollte dieser entfernt werden, bevor mit dem Testen mit RTRT begonnen wird.

Beschreibung der verwendeten Tools in der Integrations-Testphase: Im *Integrationstest* wird überprüft, ob die Komponenten wie im technischen Entwurf vorgesehen zusammenwirken. In dieser Testphase werden fünf verschiedene Werkzeuge eingesetzt: “CANalyser”, “CANoe” und “CAPL” [Vec05], sowie Polyspace [Pol05] und QA-C [QA 05].

CANalyser und CANoe bieten hierbei Möglichkeiten, Autofahrten zu simulieren und währenddessen Messungen der ACC-internen Signale vorzunehmen. Weiterhin können mit diesen Tools Parameter eingestellt werden, und dann das resultierende Fahrverhalten simuliert und analysiert werden.

Nachdem das Zusammenwirken der Komponenten mit CANalyser und CANoe geprüft wurden, folgt eine Laufzeitanalyse mit Polyspace sowie eine erneute Überprüfung der Kodierregeln mit QA-C, bevor mit dem Systemtest begonnen wird. In Abbildung 2.1 wurde der Einsatz dieser zwei Programme daher zwischen den Integrations- und den Systemtest eingetragen.

Beschreibung der verwendeten Tools in der System-Testphase: Nach Abschluss der Integrations-Testphase wird im *Systemtest* überprüft, ob das System als Ganzes die spezifizierten Anforderungen erfüllt. Hierfür werden zwei Tools eingesetzt: “Accurate” und “CLARA”.

Accurate ist ein Diagnose-Tool. Es verwendet die Simulationsfunktionen von CANalyser und CANoe. Beispielsweise wird damit eine Autofahrt simuliert, auf der einige Fehler hervorgerufen werden, um anschließend den Fehlerspeicher auszulesen und zu überprfen.

CLARA ist ein Hardware in the Loop(HIL) Tool. Dieses Programm liefert für den Testrahmen eine Simulation der Fahrumgebung(Straße, Steigung der Straße, Fahrerverhalten,...) sowie des Fahrzeugs selbst. In dieser Umgebung wird das erwartete Verhalten des ACC-Steuergeräts mit dem tatsächlich erhaltenen Verhalten verglichen. Hier wird also nach Funktionsfehlern des Gesamtsystems gesucht. Auch Flash- und Boot-Tests sind mit diesem Programm möglich, und auch Laufzeitverhalten - wie beispielsweise in welchen Taktzyklen das Fahrerübernahme-Symbol blinken soll - und Betriebssystem Einflüsse können mit diesem Tool überprüft werden.

Die HIL-Simulation ist jedoch nur eine Vereinfachung der Realität. Daher muss bei jedem Fehlverhalten geprüft werden, ob es sich um einem echten Fehler handelt, oder ob nur das der Simulation zugrunde liegende Modell zu stark vereinfacht oder sogar falsch war.

Anschließend können mit den Systemtests in echten Fahrzeugen begonnen werden.

Schlussfolgerungen für unser Projekt: RTRT wird in der Modul-Testphase angewendet. Da Laufzeitfehler - wie beispielsweise nicht initialisierte Variablen oder Speicherleaks - bereits durch Polyspace aufgedeckt werden sollen, wird RTRT in unserem Projekt ausschließlich zur Aufdeckung von Funktionsfehlern verwendet, obwohl auch dieses Programm Laufzeitfehler wie Speicherleaks entdecken könnte.

RTRT wird daher dazu verwendet, folgende Fehlerarten zu finden:

- Funktionsfehler

- Fehlende Fehlerbehandlung
- Nicht mehr benötigter Programmtext
- Fehlende Requirements
- Unerreichbarer Programmtext

Alle anderen Fehlerarten sollen bereits durch andere Werkzeuge abgedeckt werden.

Kapitel 3

Grundlagen des Softwaretestens

> Testing can only show the presence of errors, never their absence.
>
> E.W. Dijkstra [DDH72]

In diesem einleitenden Kapitel werden Grundbegriffe und grundlegende Verfahren des Softwaretestens erläutert. Um die Vorgehensweisen klarer werden zu lassen, werden sie bei Bedarf an einem Fallbeispiel des ACC-Projekts veranschaulicht.

3.1 Validierung versus Verifikation

Die Begriffe *Validierung* und *Verifikation* werden oft missbräuchlich verwendet. Aus diesem Grund sollen sie hier etwas genauer beleuchtet werden.

Ein Grund, weshalb die Begriffe oft falsch verstanden werden, liegt darin, dass die Begriffe überdefiniert wurden. Einerseits unterscheidet man zwischen Validierung und Verifikation in Bezug auf die Dokumente, gegen die geprüft werden soll: So soll bei der Validierung geprüft werden, ob das Testobjekt die Anforderungen der beabsichtigten Nutzung erfüllt, wohingegen bei der Verifikation geprüft wird, ob das Testobjekt die Vorgaben der Phaseneingangsdokumente erfüllt [SL04].

Verifikation ist hier also anders als Validierung auf einzelne Entwicklungsabschnitte bezogen. Untersucht wird, ob die Spezifikation richtig umgesetzt wird, unabhängig von dem beabsichtigten Zweck oder Nutzen des Produkts. Bei der Validierung hingegen wird die Spezifikation selbst in Frage gestellt. Hier soll die Frage beantwortet werden, ob das richtige System erstellt wurde, also eines, das den Kundenanforderungen genügt.

Methoden dieser Validierung sind Anforderungsüberprüfung, Designreviews sowie die Überprüfung der Reviews [Per03]. Diese Methoden können bisher nur von Menschen durchgeführt werden, da die Kundenanforderungen oft zu vage sind, um automatisiert geprüft werden zu können.

Gleichzeitig werden die Begriffe "Validierung" und "Verifikation" aber auch verwendet, um das Vorgehen zu unterscheiden: bei der Validierung wird stichprobenartig überprüft, ob das Testobjekt das gewünschte Verhalten zeigt, wohingegen bei der Verifikation formal bewiesen wird, ob eine Eigenschaft für alle möglichen Zustände des Systems erfüllt ist [Kro99]. Wir verwenden in dieser Arbeit diese zweite Bedeutung, aus diesem Grund soll dies im Folgenden kurz näher erläutert werden.

3.1.1 Validierung von Software

Das Prüfen, ob das Verhaltens der Implementierung der formalen Spezifikation entspricht, wird *Validierung* genannt [Kro99].

Die übliche Validierungsmethode in der Softwareentwicklung sind Tests. Hierbei wird die zu prüfende Software auf einem Rechner zur Ausführung gebracht und stichprobenartig geprüft, ob sie auf beliebige Eingaben die erwarteten Ausgaben zurückliefert. Allerdings müsste jede mögliche Eingabekombination getestet werden, wenn man die Korrektheit des Systems auf diese Weise nachweisen wollte. Da die Anzahl an Kombinationsmöglichkeiten der Eingabe exponentiell in der Anzahl der Eingangssignale wächst, muß aus praktischen Gründen in jedem komplexen System auf einen vollständigen Test verzichtet werden.

Testen ist in der Softwareentwicklung also semientscheidbar: Tests können nur zeigen, dass ein System fehlerhaft ist, aber nicht, dass es korrekt funktioniert [DDH72]. Trotzdem ist in der Softwareentwicklung die Validierung das meist verwendete Mittel zur Qualitätssicherung.

3.1.2 Verifikation von Software

Im Gegensatz zu der Validierung, bei der stichprobenartig geprüft wird, ob die Implementierung eines Systems der Spezifikation entspricht, liefert die *Software-Verifikation* den *Beweis*, dass Implementierung und Spezifikation übereinstimmen [HS01]. Hierfür muss gezeigt werden, dass sich Implementierung und Spezifikation zu jedem Zeitpunkt und für jede Eingabemöglichkeit decken.

Solche Beweise beruhen meist auf Methoden der formalen Semantik. Allerdings ist eine Verifikation von Software nicht in jedem Fall möglich, wie das Halteproblem und der Gödelsche Unvollständigkeitssatz zeigen.

Da Verifikationsbeweise meist sehr groß und oft nicht intuitiv sind, werden interaktive oder automatisierte Theorembeweiser eingesetzt. Erstere basieren auf symbolischer Deduktion, während letztere spezielle Datenstruk-

turen verwenden. Interaktive Theorembeweiser werden seit über 35 Jahren in Pilotprojekten der amerikanischen Regierung verwendet. Sie konnten sich aber nie richtig auf dem Markt durchsetzen [Kur97].

Zur automatisierten Verifikation werden oft Automatenmodelle eingesetzt. Hintergrund ist die Möglichkeit, formale Spezifikationen in äquivalente Automaten zu überführen. Dabei wird das Problem die Erfüllung einer Spezifikation zu prüfen, auf ein äquivalentes Problem der Analyse einer Eigenschaft des Automaten überführt. Automaten sind für die Analyse die geeignetere Repräsentation der Problemstellung, da hier gute Algorithmen bekannt sind.

Softwareverifikation wird bisher nur sehr selten von Firmen verwendet. Dies hat mehrere Gründe: Zunächst gibt es kommerzielle Verifikationstools erst seit wenigen Jahren. Weiterhin ist die Akzeptanz der Tools noch nicht sehr weit gediehen. Der Hauptgrund liegt aber vermutlich daran, dass sich viele Anwendungen noch nicht verifizieren lassen, da die Komplexität der entstehenden Modelle zu groß wird [Kur97].

Trotzdem hat sich in den letzten Jahren viel auf diesem Forschungsgebiet getan, und aufgrund der immer raffinierter werdenden Methoden sowie leistungsfähigerer Computer scheint es wahrscheinlich, dass die Verifikation von Software in einigen Jahren auch für Unternehmen interessant wird.

3.1.3 Veranschaulichung der Begriffe am Beispiel

Wie hängen die Begriffe nun zusammen? Bei der Validierung wird *stichprobenartig* geprüft, ob die Spezifikation erfüllt wird, wohingegen Softwareverifikation *beweist*, dass Implementierung und Spezifikation übereinstimmen.

im Folgenden werden die Unterschiede an einem Beispiel von [Gor89] veranschaulicht:

Beispiel 1. *Angenommen, ein Programmteil implementiert die Berechnung der Formel* $(x+1)^2$ *durch eine Berechnung der Teilergebnisse* x^2+2x+1. *Es muß also geprüft werden, ob* $(x+1)^2 = x^2+2x+1$.

Bei der Validierung würde nun stichprobenartig überprüft, ob die Ergebnisse der beiden Formeln übereinstimmen. Dies wird in Tabelle 3.1 veranschaulicht. Sollte mit diesem Verfahren bewiesen werden, dass die Formeln übereinstimmen, so müsste x alle möglichen Werte annehmen - im Normalfall ein unmögliches Unterfangen.

Soll also ein Beweis angestrebt werden, so muß ein anderes Verfahren gewählt werden: ein formaler Beweis. Innerhalb der Verifikation werden solche Beweise aufgestellt. Ein formaler Beweis für das Beispiel findet sich in Tabelle 3.2.

x	$(x+1)^2$	x^2+2x+1
0	1	1
1	4	4
2	9	9
3	16	16
5	36	36
10	121	121
...	...	...

Tabelle 3.1: *Bei der Validierung wird stichprobenartig überprüft, ob die Formeln übereinstimmen.*

	Umrechnung	Zugrunde liegendes Gesetz
1	$(x+1)^2 = (x+1)(x+1)$	Definition des Quadrats
2	$(x+1)(x+1) = (x+1)x+(x+1)1$	Distributionsgesetz
3	$(x+1)x+(x+1)1 = (x+1)x+x+1$	1 als neutrales Element
4	$(x+1)x+x+1 = x^2+x+x+1$	Distributionsgesetz
5	$x^2+x+x+1 = x^2+2x+1$	Definition von $2x$

Tabelle 3.2: *Bei der Verifikation wird durch einen formalen Beweis gezeigt, dass die Formeln übereinstimmen.*

3.2 Requirements

Die Entwicklung von Software ist ein komplexer Prozess der eine Vielzahl von Teildisziplinen der Informatik und des Projektmanagements umfasst. Die wesentlichen Gebiete dabei sind *Planung, Analyse, Entwurf, Implementierung, Test, Qualitätsmangement, Konfigurationsmanagement* und *Dokumentation*. Schon in der Planungsphase werden ein Lastenheft und ein Pflichtenheft erstellt, in dem die *Anforderungen* oder *Requirements*, die die Software erfüllen muss, beschrieben werden. Auf diesen Anforderungen baut der gesamte nachfolgende Prozess auf. Daher ist es sehr wichtig, genügend Projektzeit dafür einzuplanen: Jeder Fehler der hier gemacht wird, benötigt zur Korrektur ein Vielfaches der Zeit, die es kostet, eine umsichtige Anforderungsdefinition zu erstellen.

Definition 1. *Eine* Requirement-Spezifikation *ist eine präzise Beschreibung des Verhaltens und der Eigenschaften eines Systems. Sie soll so abstrakt wie möglich spezifizieren, was das System leisten soll und in welchem Umfeld es verwendet werden soll. Details zur Umsetzung werden vermieden [Kro99, RHS98].*

Sehr formale Requirement Spezifikationen sind in einer Sprache mit wohldefinierter Semantik verfasst - beispielsweise in mathematischen Formalismen - die es auch Computern ermöglicht, aus der Spezifikation logische Schlsse zu ziehen oder automatisiert Testskripts abzuleiten.

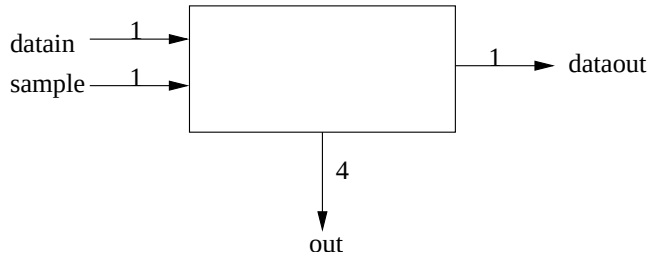

Abbildung 3.1: *In dieser Abbildung wird ein seriell-paralleler Konverter dargestellt. "Datain" erwartet einen Bitstream als Input. "Dataout" gibt diesen Bitstream wieder aus, allerdings um vier Taktzyklen verzögert.*

Man kann zwischen zwei Arten von Requirements unterscheiden: *funktionale* und *nichtfunktionale* oder *Quality-Requirements* [Cen02]. Funktionale Requirements spezifizieren, was das System machen soll, wohingegen nichtfunktionale die Eigenschaften des Systems spezifizieren. Beispiele für nichtfunktionale Forderungen sind Nutzerfreundlichkeit, Robustheit oder Performance des Systems.

Die Gratwanderung zwischen Abstraktion und ausreichender Spezifikation ist schwierig. Oft sind Requirements nicht ausreichend spezifiziert, sie bleiben vage, ungenau, unvollständig und schlecht strukturiert. Da dies nicht nur für die Design Spezifikation ein Problem darstellt, sondern später beim Testen ebenso hinderlich ist, soll diese Problematik an folgendem Beispiel von Gordon [Gor89] verdeutlicht werden. Er verwendet ein Beispiel aus der Hardware, aber es ist klar, dass in der Software dieselben Probleme auftreten können.

Beispiel 2. *Gegeben sei der Schaltkreis aus Abbildung 3.1 sowie folgende Spezifikation: "Datain" erwartet einen Bitstream als Input. "Dataout" gibt diesen Bitstream wieder aus, allerdings um vier Taktzyklen verzögert. Der Bus "out" ist vier Bits breit. Falls an "sample"* $false$ *anliegt, so sollen an out die letzten vier Bits des Inputs von datain weitergereicht werden. Andernfalls soll dort* $FFFF$ *anliegen.*

Obwohl diese Spezifikation auf den ersten Blick sehr ausführlich wirkt, stellt sie sich bei näherem Hinsehen als nicht ausreichend genau heraus. Zunächst ist sie ungenau: Beinhalten die "letzten vier Bits des Inputs von datain" das momentan eingelesene Bit? Weiterhin ist die Beschreibung unvollständig. Es wird nicht geklärt, was dataout im ersten, zweiten und dritten Taktzyklus ausgeben soll. Außerdem ist die Spezifikation in Alltagssprache verfasst, es wäre daher sehr schwierig sie automatisiert analysieren zu lassen.

Gordon erreicht über verschiedene Zwischenschritte die folgende formale Spezifikation, die sowohl vollständig als auch genau ist:

$$\forall t : time.dataout(t+4) = datain(t) \wedge$$
$$(out(t+4) = ifsample(t+4)$$
$$then[F, F, F, F]$$
$$else[datain(t), datain(t+1), datain(t+2), datain(t+3)])$$

Dieses Beispiel soll verdeutlichen, wie schwierig es ist, ausreichend spezifizierte formale Requirements zu definieren. Trotzdem lohnt der Aufwand, selbst wenn keine automatisierte Testskripterstellung geplant ist [AOS04, CCG+04]:

- Die Schwierigkeit des Formalisierens erfordert ein großes Verständnis des gewünschten Verhaltens, so dass auf diese Weise Designfehler bereits in dieser Phase aufgedeckt werden können.
- Weiterhin erfordert die Formalisierung eine hohe Abstraktionsstufe - aus diesem Grund wird, wie gewünscht, stärker versucht zu spezifizieren, wie sich das System verhalten soll; Wie genau dieses Verhalten erzielt werden soll gerät so in den Hintergrund. Somit wird stärker zwischen Design und Implementierung getrennt.
- Außerdem ist eine formale Beschreibung eine deutlich bessere Grundlage für das weitere Vorgehen: Sie stellt sicher, dass jeder der daran beteiligten Entwickler und Tester auf derselben Grundlage arbeitet. Schnittstellenfehler, die auf nicht spezifizierten Annahmen an die Module beruhen, können so vermieden werden.
- Zusätzlich kann der Testprozess so deutlich beschleunigt werden. Bei ungenauen Requirements muss ein Großteil der zur Verfügung stehenden Zeit dazu genutzt werden, herauszufinden, was genau denn getestet werden soll. Die Requirements müssen also nachträglich noch zusammen gesucht und genauer spezifiziert werden. Wenn dies vom Tester ausgeführt werden muss, so wird die sowieso meist knapp kalkulierte Testzeit auf Tätigkeiten verschwendet, die eigentlich am Anfang des Projekts hätten stattfinden müssen [SL04, Mye82].

Ausreichend spezifizierte Requirements bilden nicht nur die Grundlage für ein in sich schlüssiges Design sondern auch für die Testerstellung. Es ist daher darauf zu achten, dass die Anforderungen am Anfang des Projekts mit genügender Sorgfalt spezifiziert werden.

3.3 Fehlerbegriff und Fehlerursachen

Beim Testen wird die zu prüfende Software zur Ausführung gebracht und geprüft, ob sie sich den gestellten Anforderungen entsprechend verhält oder nicht. Es muss also im vorab geregelt sein, welches Verhalten korrekt und

erwartet ist. Ein *Fehler* ist dann die Nichterfüllung eines Requirements - das heißt das Ist-Verhalten, das während des Tests auftritt, unterscheidet sich in diesem Punkt vom Soll-Verhalten, das in der Spezifikation festgelegt wird. Häufig liegt ein Fehler darin begründet, dass Ausnahmesituationen nicht bedacht werden.

Im Unterschied dazu spricht man von einem *Mangel*, wenn eine gestellte Anforderung nicht angemessen erfüllt wird. *Fehlermaskierung* tritt auf, wenn ein Fehlerzustand durch einen anderen Defekt im Programm kompensiert wird - in diesem Fall macht sich der Fehler erst bemerkbar, nachdem der maskierende Fehler behoben wurde. Es gibt verschiedene Softwarefehlerarten, auf die unterschiedlich geprüft werden muss [Mye82]. Hier soll ein unvollständiger Überblick gegeben werden:

- Ein *Datenreferenzfehler* liegt beispielsweise vor, wenn eine Variable angesprochen wird, deren Wert zu dem Zeitpunkt im Programm noch nicht gesetzt oder initialisiert wurde oder falls bei einem Feldaufruf der Indexwert nicht in den definierten Grenzen der entsprechenden Dimension liegt.

- Ein *Datenvereinbarungsfehler* tritt auf, falls einer Variable nicht die korrekte Länge, Typ oder Speicherklasse zugeordnet wird.

- Man spricht von einem *Berechnungsfehler*, falls Fehler durch Berechnungen mit inkonsistenten Datentypen auftreten.

- Ein *Vergleichsfehler* tritt auf, falls die Vergleichsrelationen falsch angewendet werden, oder falls der Datentyp der zu vergleichenden Variablen nicht identisch ist.

- Ein *Steuerflußfehler* liegt vor, falls eine Schleife nie beendet werden kann oder falls sie zu oft bzw. zu selten ausgeführt wird(meist eine Iteration zuviel oder zuwenig).

- Man spricht von einem *Schnittstellenfehler*, falls die Anzahl der Parameter in einem Modul nicht mit der Anzahl der übergebenen Argumente übereinstimmt, oder deren Reihenfolge nicht korrekt ist. [1]

- Ein *Ein-/Ausgabefehler* kann auftreten, falls die Formatspezifikationen nicht zu den E/A-Anweisungen passt oder falls die Puffergrößen nicht für die Satzlänge ausreicht.

- Ein *Funktions-* oder *Designfehler* liegt vor, falls das Verhalten des Programms nicht dem in den Requirements geforderten Verhalten entspricht.

[1] Im automotive Bereich kann es auch schnell passieren, dass die entsprechenden Einheiten nicht übereinstimmen - also dass beispielsweise ein Modul im km/h und ein anderes in m/s rechnet, die Umrechnung aber vergessen wird.

Fehler wie Buffer Overflows, Null Pointer Referenzierungen oder nicht initialisierte Variablen können inzwischen verhältnismäßig zuverlässig durch Tools gefunden werden, die mit Hilfe einer statischen Analyse mögliche Pfade des Programms in Betracht ziehen [BPS00]. Solche Fehler können durch Tests nur sehr schwer gefunden werden. Funktionsfehler oder Designfehler hingegen können durch Tests verhältnismäßig leicht gefunden werden, wohingegen statische Analyse Tools in diesem Bereich versagen [NB05a]. Je nach Fehlerart macht es also Sinn, verschiedene Methoden anzuwenden. In dieser Arbeit soll das Testen im Vordergrund stehen. Aus diesem Grund wird ein besonderes Augenmerk auf Funktions- und Designfehler gelegt.

3.4 Testbegriff und Softwarequalität

Unter Softwaretest versteht man eine im Allgemeinen *stichprobenartige* Ausführung eines Testobjekts auf einem Computer, mit dem Ziel das Objekt zu überprüfen. Durch einen Vergleich zwischen dem Sollverhalten, das in den Requirements spezifiziert wird, und dem Ist-Verhalten, das während des Tests beobachtet werden kann, ist es dann möglich, zu bestimmen, ob das Objekt die geforderten Eigenschaften erfüllt.

Ziel des Testens ist es, Fehlerwirkungen gezielt und systematisch aufzudecken. Die Fehler danach zu beheben ist **nicht** Aufgabe des Testers sondern die des Softwareentwicklers. Der *Testprozess* gliedert sich in *Planung, Durchführung* und *Auswertung* der Tests. Ein *Testlauf* umfasst die Ausführung mehrerer *Testfälle*.

Der Testfall spezifiziert die festgelegten Randbedingungen - beispielsweise die Voraussetzungen zur Ausführung des Testobjekts, die Eingabewerte sowie die erwarteten Ausgaben.

Ein *guter Testfall* ist dadurch gekennzeichnet, dass er mit hoher Wahrscheinlichkeit einen bisher unbekannten Fehler findet. Beispielsweise ist es oft lohnend die Grenzwerte zu prüfen, also beispielsweise Testfälle zu entwickeln, die knapp inner- bzw. außerhalb des Wertebereichs der Variablen liegen. Ein *erfolgreicher Testfall* kennzeichnet sich dadurch aus, dass er einen bisher unbekannten Fehler entdeckt.

Das Testen von Software dient zum Auffinden von Fehlern. Durch deren anschließende Beseitigung soll die *Softwarequalität* gesteigert werden. Der Begriff umfasst allerdings mehr als nur die Behebung der beim Testen aufgefundenen Fehlern: Laut der ISO-Norm 9126 [91201] wird die Softwarequalität durch verschiedene Faktoren bestimmt. Diese Qualitätsmerkmale sind *Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit* und *Übertragbarkeit*.

- Das Merkmal *Funktionalität* beschreibt, ob die geforderten Fähigkeiten des Systems erfüllt wurden bzw. ob das spezifizierte Ein-/Ausgabeverhalten vorliegt. Weiterhin wird in diesem Bereich die Interoperabilität

getestet, also ob die Software fehlerfrei mit anderen Systemen (z.B. einem Betriebssystem) kommuniziert. Eine *ordnungsgemäße Funktionalität* liegt vor, wenn die Software anwendungsspezifische Normen und gesetzliche Bestimmungen erfüllt.

- *Zuverlässigkeit* beschreibt die Fähigkeit eines Systems ein Leistungsniveau unter festgelegten Bedingungen für einen zuvor definierten Zeitraum zu bewahren. Dies beinhaltet auch die Frage, wie häufig ein Fehlerzustand durch Versagen der Software überhaupt vorkommen kann, und ob und wie schnell das geforderte Leistungsniveau nach einem Versagen wieder erreicht werden kann.

- Die *Benutzbarkeit* spielt besonders bei interaktiven Softwaresystemen eine große Rolle. Hier steht im Vordergrund wieviel Aufwand für verschiedene Benutzergruppen notwendig ist, die Software zu verwenden. *Verständlichkeit, Erlernbarkeit* und *Bedienbarkeit* sind Teilaspekte der Benutzbarkeit.

- Die *Effizienz* misst die benötigte Zeit und den Verbrauch der Betriebsmittel für die Erfüllung der geforderten Aufgaben.

- Da Softwaresysteme oft über einen längeren Zeitraum hinweg weiterentwickelt werden, spielen auch die Kriterien *Änderbarkeit* und *Übertragbarkeit* eine große Rolle. Ein System ist dann gut änderbar, falls es leicht analysierbar, modifizierbar und prüfbar ist. Unter Übertragbarkeit wird *Anpassbarkeit, Installierbarkeit, Konformität* und *Austauschbarkeit* zusammengefasst.

Um die Gesamtqualität der Software beurteilen zu können, müssen beim Testen all diese Merkmale berücksichtigt werden. Welches Qualitätsniveau die Software dabei bezüglich jedes einzelnen Merkmals aufweisen soll muss in den Requirements spezifiziert werden. Diese Festlegung dient für den Test als Richtschnur zur Bestimmung der Intensität der Überprüfung der einzelnen Merkmale.

3.5 Grundlegende Testprinzipien

Oft werden Programme mit der Einstellung getestet, dass gezeigt werden soll, dass das zu prüfende Programm korrekt läuft. Tests, die mit dieser Einstellung geschrieben werden sind meist nicht besonders erfolgreich [Mye82]. Dies liegt an zwei Gründen: Erstens ist es aus praktischen Gründen unmöglich, dieses Ziel zu zeigen und zweitens führt eine solche Einstellung sehr wahrscheinlich zu einer falschen Herangehensweise.

Will man alle Fehler in einem Programm finden, so müßte jede mögliche Eingabebelegung getestet werden. Dies würde bereits bei kleinen Program-

men zu sehr vielen Testfällen führen, so dass es bei komplexen Programmen praktisch unmöglich ist, einen solchen Test durchzuführen.

Die große Anzahl an möglichen Testfällen führt bereits zu einem weiteren Testziel: da vollständiges Testen aufgrund oben genannter Komplexität nicht in Frage kommt, muss es ein Testziel sein, möglichst wirtschaftlich zu testen. Das bedeutet, dass man mit möglichst wenigen Testfällen möglichst viele Fehler entdecken will. Dies erfordert, dass der Tester in der Lage ist, das Programm zu analysieren und einige sinnvolle Annahmen über das Programm zu machen. Wirtschaftlich zu testen bedeutet aber auch, dass der Testaufwand in Relation zu dem mit dem Fehlerfall verbundenen Risiko und der Gefährdungsbewertung steht. Testen ist solange ökonomisch sinnvoll, wie die Kosten für das Finden und Beseitigen der Fehler geringer ist als die Kosten, die mit dem Auftreten des Fehlers verbunden sind [PKS00]. Eine Möglichkeit, zu entscheiden, ob das Testen beendet werden kann oder nicht, bieten Fehlermodelle, die die Anzahl der erwarteten Fehler der Software abschätzen. Dieses Prinzip wird in Kapitel 7 näher beleuchtet.

Das zweite Problem an der oben genannten Definition ist, dass die Einstellung falsch ist. Ein Programm wird getestet um dessen Qualität zu sichern und die Zuverlässigkeit zu prüfen. Statistisch enthält gute Software auf 1000 Zeilen Programmtext etwa 3 Fehler, die NASA erreichte durch großen Kostenaufwand einen Schnitt von etwas unter 1 Fehler, in der Industrie liegt der Standard bei 25 Fehlern auf 1000 Zeilen Programmtext [Gib94, Gle96, Neu95]. Es ist also davon auszugehen, dass sich in jeder komplexeren Software Fehler finden lassen. Um die Software zu verbessern müssen diese Fehler gesucht und danach entfernt werden. Und genau das sollte die Einstellung sein, mit der getestet wird:

Beim Testen versucht man zu zeigen, dass das Programm nicht korrekt ist [Mye82].

Laut Myers hat dieser Bedeutungsunterschied große Folgen. Da Menschen zielorientiert arbeiten kann die Einstellung, mit der getestet wird, den Testvorgang stark beeinflussen. Ist es das Ziel zu zeigen, dass die Software korrekt läuft, so werden unbewußt Testfälle gewählt, die nur mit sehr geringer Wahrscheinlichkeit fehlschlagen. Ist es dagegen das Ziel, nach Fehlern zu suchen, so werden eher die Randfälle abgesucht und somit steigt die Wahrscheinlichkeit auf Fehler zu stoßen.

Daher wollen wir im Folgenden auch von einem “erfolgreichen Testfall” sprechen, falls dieser einen Fehler aufdeckt, wohingegen beim unerfolgreichen Test das Programm wie erwartet abläuft. Dies mag auf den ersten Blick etwas ungewohnt erscheinen, aber macht Sinn: Wenn ein Kranker einen Arzt aufsucht, dann erwartet er eine Diagnose, an was für einer Krankheit er leidet und welche Medikamente er dagegen nehmen kann. Würde der Arzt ihn untersuchen ohne eine Krankheit festzustellen, dann wäre dem Kranken nicht geholfen, der Arztbesuch wäre dann nicht erfolgreich. Wie von dem Arzt erwartet wird, dass er diagnostizieren kann an welcher Krankheit ein

Patient leidet, so wird vom Tester erwartet, dass er die Fehler einer Software finden und genauso diagnostizieren kann.

Diese Diskussion kann zu drei wichtigen Folgerungen zusammengefasst werden [Mye82]:

Regel 1. *Beim Testen versucht man zu zeigen, dass das Programm nicht korrekt ist.*

Regel 2. *Man kann ein komplexes Programm nicht so testen, dass seine Fehlerfreiheit garantiert wird.*

Regel 3. *Ein fundamentaler Gesichtspunkt beim Testen ist die Wirtschaftlichkeit.*

Da das Testen wie oben angesprochen stark von der Psychologie beeinflusst wird, hat Myers weitere Testrichtlinien aufgestellt. Diese sind intuitiv einsichtig, werden aber allzu oft übersehen. Aus diesem Grund sollen sie hier noch einmal genannt werden:

Regel 4. *Ein notwendiger Bestandteil eines Testfalls ist die Definition der erwarteten Werte oder des Resultats.*

So offensichtlich dieses Prinzip klingt, die Nichtbeachtung dieser Richtlinie ist einer häufigsten Fehler beim Programmtesten [Mye82]. Nur wenn Erwartungen bestehen, wie sich das Programm verhalten soll, kann man diese mit dem tatsächlichen Verhalten vergleichen - bestehen keine Erwartungen, so besteht die Möglichkeit, dass ein Tester ein plausibles aber fehlerhaftes Verhalten als korrekt betrachtet.

Regel 5. *Ein Programmierer sollte nicht sein eigenes Programm testen.*

Diese Richtlinie soll nicht heißen, dass es für einen Programmierer unmöglich ist, sein eigenes Programm zu testen - sie soll vielmehr besagen, dass Tests, die von unabhängigen Testern durchgeführt werden, statistisch gesehen bedeutend erfolgreicher sind. Dieser Erfahrungswert kann durch zwei Argumente begründet werden: Zunächst ist Testen ein destruktiver Prozess. Ein Tester versucht nachzuweisen, dass das Programm fehlerhaft ist. Ein Programmierer, der sehr viel Zeit in das Design und die Umsetzung gesteckt hat, will aber vermutlich keine Fehler finden, sondern möchte zeigen, dass sein Programm korrekt funktioniert. Dass diese Einstellung zu schlechten Testfällen führen kann, haben wir bereits am Anfang dieses Kapitels besprochen. Der zweite Grund, weshalb ein Programmierer nicht seine eigene Software testen sollte, liegt darin, dass Fehler, die aufgrund eines Missverständnisses der Problemstellung seitens des Programmierers entstanden sind, nicht von ihm aufgespürt werden können. Der Programmierer wird als Tester die Aufgabe wieder so verstehen, wie er sie bei dem Design interpretierte. Auf diese Weise ist eine ganze Fehlerklasse nicht als Fehler auffindbar, falls der Programmierer selbst testen soll.

Regel 6. *Die Ergebnisse eines jeden Tests müssen gründlich überprüft werden.*

Myers hat zu diesem Thema einige Experimente durchgeführt, in denen er zeigte, dass viele Testpersonen nicht in der Lage sind, bestimmte Fehlerarten zu entdecken, obwohl deren Symptome im Ausgabelisting klar ersichtlich waren. Er zeigte, dass ein signifikanter Anteil an entdeckten Fehlern bereits in früheren Testfällen erkennbar waren, aber aufgrund nicht sorgfältiger Überprüfung der Ergebnisse übersehen wurden [Mye78].

Regel 7. *Testfälle müssen sowohl für ungültige wie auch für gültige Eingaben definiert werden.*

Obwohl man unwillkürlich dazu tendiert, die Tests auf zulässige Eingabedaten zu konzentrieren - Tests für ungültige Eingabedaten sind ebenso wichtig. In komplexen Softwaregebilden, kann es schnell geschehen, dass bei der Integration der Module ein Spezialfall vergessen wurde und ein Modul plötzlich mit ungültigen Eingabewerten arbeiten muss. Durch das Testen wird schnell offensichtlich, ob das Modul auch mit solchen Eingaben umgehen kann oder ob der Programmierer vergessen hat, solche Fälle abzudecken.

Regel 8. *Das Programm muss auch auf unerwünschte Nebeneffekte untersucht werden.*

Dies wird durch ein Beispiel schnell plausibel: Wenn die untersuchte Displayansteuerung zwar die Wunschgeschwindigkeit korrekt anzeigt, dabei aber eine Fehlermeldung ausgibt oder den Fehlerspeicher überschreibt, so ist die Funktion fehlerhaft - obwohl sie ihre Aufgaben korrekt erfüllt.

Regel 9. *Wegwerftestfälle sollten vermieden werden[SL04].*

Kapitel 4

Testen im Software Lebenszyklus

> Debugging is twice as hard as writing the Programmtext in the first place. Therefore, if you write the Programmtext as cleverly as possible, you are, by definition, not smart enough to debug it.
>
> Brian W. Kernighan

In diesem Kapitel soll anhand des allgemeinen V-Modells die Rolle des Testens im gesamten Softwarelebenszyklus veranschaulicht werden. Weiterhin soll gezeigt werden, welche Teststufen und Testmethoden im Entwicklungsverlauf zum Einsatz kommen.

4.1 Das allgemeine V-Modell

Zur Durchführung einer strukturierten und steuerbaren Softwareentwicklung werden Softwareentwicklungsmodelle eingesetzt. Es gibt viele verschiedene Modelle, beispielsweise das Wasserfallmodell [Boe73], das allgemeine V-Modell [Boe79] oder Extreme Programming. Allen Modellen gemein ist die Definition eines Vorgehens für eine systematischen, geordneten Projektabwicklung. Auf diese Weise soll eine für alle Beteiligten gemeinsame und verbindliche Sicht der auszuführenden Tätigkeiten und deren zeitlichen Abfolge in dem Projekt festgelegt werden. Meist werden Phasen festgelegt, die mit einem bestimmten Ergebnis abzuschließen sind. Ein *Phasenabschluß*, oder *Meilenstein*, ist dann erreicht, wenn die geforderten Dokumente fertig gestellt sind und den geforderten Qualitätskriterien genügen.

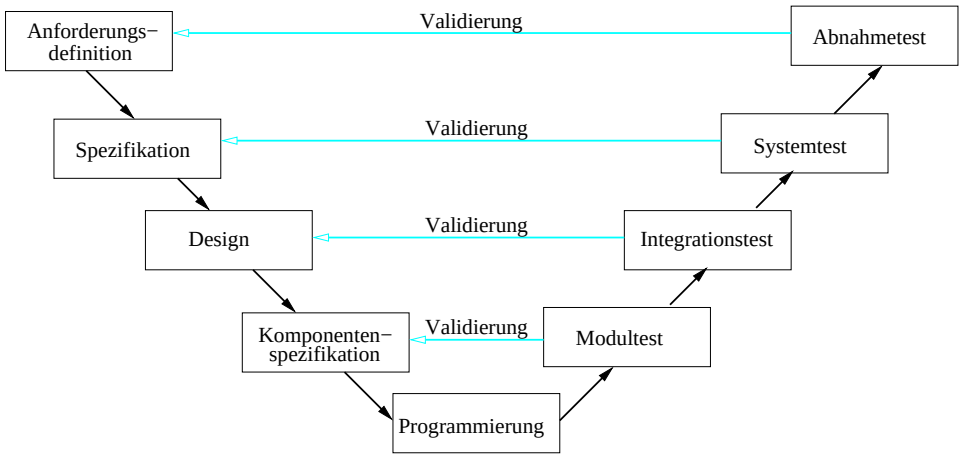

Abbildung 4.1: *Im allgemeinen V-Modell werden die Testaktivitäten erstmals als gleichwertig zu den Entwicklungsprozessen verstanden. Jedem Spezifikations- oder Konstruktionsschritt wird eine korrespondierende Teststufe zugeordnet.*

Das Testen findet sich in jedem dieser Vorgehensmodelle wieder, allerdings mit sehr unterschiedlicher Bedeutung und Umfang. Aus Sicht des Testens spielt das allgemeine V-Modell eine besondere Rolle. In diesem Modell werden die Testaktivitäten erstmals als gleichwertig zu den Entwicklungsarbeiten verstanden. Das Verständnis vom Softwaretest wurde damit nachhaltig beeinflusst [SL04]. Das allgemeine V-Modell wird in Abbildung 4.1 dargestellt.

Die Grundidee des allgemeinen V-Modells ist, dass Entwicklungsarbeiten und Testarbeiten zueinander korrespondierende, gleichberechtigte Tätigkeiten sind. Bildlich wird dies in den beiden Ästen des Vs dargestellt. Der linke Ast versinnbildlicht die immer detaillierter werdenden Entwicklungsschritte, in deren Verlauf das vom Kunden gewünschte System schrittweise entworfen und schließlich programmiert wird. Der rechte Ast dagegen zeigt die Integrations- und Testarbeiten. In diesem Ast werden elementare Softwarebausteine sukzessive zu größeren Teilsystemen zusammengefasst und auf Funktionalität geprüft.

Die Aktivitäten im linken Ast beginnen mit der *Anforderungsdefinition*. Hier sollen Zweck und gewünschte Leistungsmerkmale der zu erstellenden Software festgelegt werden. Dazu werden die Wünsche und Anforderungen des Auftraggeber oder des Nutzers gesammelt und spezifiziert. Sobald die Anforderungsdefinition feststeht folgt der *Funktionale Systementwurf*. Hier werden die Anforderungen auf Funktionen und Dialogabläufe des Systems abgebildet. im Folgenden *Technischen Systementwurf* wird die technische Realisierung des Systems, also die Schnittstellenbeschreibung sowie die Sy-

stemarchitektur entworfen. Ziel ist es, das System in möglichst unabhängige Teilsysteme zu zerlegen, so dass die Module idealerweise unabhängig voneinander entwickelt werden können. Danach müssen für jedes Modul in der *Komponentenspezifikation* dessen Aufgaben, das Verhalten sowie der innere Aufbau und die Schnittstellen zu anderen Teilsystemen definiert werden. Ist auch dieser Schritt erfolgt kann zur *Programmierung* übergegangen werden.

An der Spitze des Vs angelangt, folgen auf dem rechten Ast die verschiedenen Testphasen. Da Fehler leichter auf der Abstraktionsstufe gefunden werden, auf der sie entstanden sind [Mye82, SL04], ordnet der rechte Ast jedem Konstruktionsschritt einen entsprechenden Testschritt zu.

Der *Komponententest* prüft die Funktion jedes einzelnen Softwarebausteins für sich. Im *Integrationstest* wird dann überprüft, ob die Komponenten wie im technischen Entwurf vorgesehen zusammenwirken. Danach wird im *Systemtest* überprüft, ob das System als Ganzes die spezifizierten Anforderungen erfüllt. Die Entwicklung wird dann mit dem *Abnahmetest* abgeschlossen, in dem getestet wird, ob das System aus Kundensicht alle vertraglich vereinbarten Leistungsmerkmale erfüllt.

In dieser Arbeit steht der Modultest im Vordergrund, trotzdem werden im Folgenden auch die anderen Teststufen kurz erläutert.

4.2 Grundlegende Teststufen und -methoden

Die IEEE-Norm 1012-1986 strukturiert den Testprozess in vier Testaufgaben: In der ersten Teststufe, dem *Modultest*, werden die kleinsten Programmeinheiten auf Funktionalität getestet - isoliert von den anderen Softwarebausteinen. Durch die Isolierung werden Modul-externe Einflüsse beim Test ausgeschlossen, so dass sich die Ursache eines gefundenen Fehlers klar dem zu testenden Modul zuordnen lässt. Das zu prüfende Testobjekt kann auch aus mehreren Bausteinen zusammengesetzt sein. Es wird dann von *Komponententest* gesprochen, solange komponenteninterne Aspekte geprüft werden - erst wenn Wechselwirkungen mit Nachbarkomponenten im Vordergrund des Tests stehen, spricht man von *Integrationstest*.

Auf den Komponententest folgt der *Integrationstest*, innerhalb dessen die ersten Module zusammengefasst und getestet werden. Hierbei steht nicht wie im Komponententest die Funktionsfähigkeit der Module im Vordergrund, sondern es soll deren Zusammenwirken geprüft werden. Ziel des Integrationstests ist es, Fehlerzustände in Schnittstellen und sowie im Zusammenspiel zwischen integrierten Komponenten zu finden. Der Integrationstest sollte erst stattfinden, wenn alle beteiligten Komponenten schon auf Funktionalität geprüft wurden. Was für Fehler kann man also durch den Integrationstest noch finden? Betrachten wir unsere Displayansteuerung. Wenn die Information innerhalb der einen Komponente in km/h gerechnet wird, während die zweite Komponente in m/s rechnet, so wird ein Fehler auftreten, wenn

man die Informationen einfach von der einen an die andere Komponente schickt. Und das, obwohl beide Komponenten für sich fehlerfrei arbeiten. Solche Fehler sollen im Integrationstest aufgedeckt werden.

Als dritte Teststufe folgt der *Systemtest* dessen Ziel es ist, zu überprüfen ob die spezifizierten Anforderungen von dem Produkt erfüllt werden. Anders als bei den niederen Teststufen, in denen aus der Perspektive des Softwareherstellers auf die Erfüllung der Spezifikation geprüft wurde, wird hier aus der Perspektive des Kunden und des späteren Anwenders geprüft[1]. Die Tester müssen das Programm mit dessen ursprünglichen Leistungsbeschreibung vergleichen.

4.3 Der fundamentale Testprozess

Der fundamentale Testprozess gliedert sich in fünf Teilaufgaben:

- Testplanung
- Testspezifikation
- Testdurchführung
- Testprotokollierung
- Testauswertung

Der Prozess wird in Abbildung 4.2 dargestellt. im Folgenden wird er näher beschrieben.

4.3.1 Testplanung

Mit der Planung des Testprozesses sollte bereits zu Beginn des Softwareentwicklungsprojekts begonnen werden. Zunächst muss die Frage geklärt werden, was für Ressourcen - also wie viele und welche Mitarbeiter, wieviel Zeit und welche Arbeitsmittel - einzuplanen sind. Weiterhin muss in dieser Phase eine Teststrategie festgelegt werden. Die Teststrategie bestimmt, welche Systemteile wie stark getestet werden sollen, welche Testmethoden angewendet werden sollen sowie die Testendekriterien. Ziel der Teststrategie ist die optimale Verteilung der Tests auf die sicherheitskritischen Teile des Systems. Anhand von Risikoschätzungen müssen Prioritäten gesetzt werden, welche Systemteile wie stark getestet werden sollen. Für kritische Systemteile sollte hier mehr Zeit und auch eine genauere Code-Überdeckung eingeplant werden, als für weniger kritische Teile.

[1] Kunde und Anwender können zwei verschiedene Personengruppen oder Organisationen sein. Der Kunde hat das System bestellt und bezahlt, der Anwender hingegen nutzt es. In unserem Beispiel wäre beispielsweise BMW ein Kunde und BMW-Fahrer Anwender

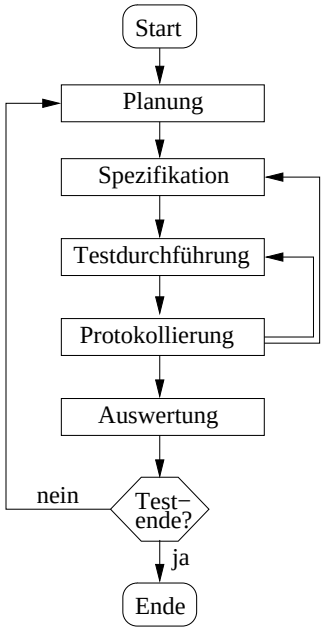

Abbildung 4.2: *Der fundamentale Testprozess gliedert sich in fünf Teilaufgaben: Testplanung, Spezifikation, Durchführung, Protokollierung und Auswertung. Wird bei der Protokollierung entdeckt, dass einige Testfälle fehlerhaft spezifiziert wurden, so muß die Spezifikation dieser Testfälle modifiziert werden und die Tests müssen erneut durchgeführt werden. Wenn durch die Auswertung der Tests ermittelt wird, dass das in der Testplanung spezifizierte Testendekriterium noch nicht erreicht wurde, muß die Testplanung modifiziert werden.*

Weiterhin ist eine Priorisierung der Tests nach Sicherheitsaspekten ratsam. Softwareprojekte werden oft unter großen Zeitdruck abgewickelt. Falls es dazu kommen sollte, dass aus Zeitgründen nicht alle Tests ausgeführt werden können, so stellt die Priorisierung sicher, dass zumindest die sicherheitskritischen Anwendungen bereits getestet wurden.

Weiterhin ist es sinnvoll, jeweils mit der Überprüfung der Hauptaufgaben des Testobjekts zu beginnen. Wenn bereits hier grobe Fehler gefunden werden sollten, dann können diese Fehler zuerst noch korrigiert werden, bevor mit dem weiteren Testen fortgefahren wird.

4.3.2 Testspezifikation

In der Testspezifikation werden die Testfälle gemäß der festgelegten Testmethodik gewählt. Hierfür werden zunächst logische Testfälle erstellt, also Testfälle, ohne Angaben von konkreten Werten. Erst danach werden diese Testfälle mit Werten belegt und so konkretisiert.

Für jeden Testfall müssen die Ausgangssituation, die Randbedingungen sowie das erwartete Verhalten spezifiziert werden. Weiterhin müssen sowohl Testfälle für gültige Eingabekombinationen wie auch für ungültige Eingaben erdacht werden.

4.3.3 Testdurchführung

Danach kann mit der Testdurchführung begonnen werden. Hierzu muss zunächst die festgelegte Testumgebung aufgebaut werden. Danach kann das Testobjekt in die Testumgebung integriert und dann getestet werden. Sollten schon am Anfang viele grobe Fehler aufgedeckt werden, muss überlegt werden, ob die Komponenten sofort korrigiert werden sollen und erst dann mit dem Testen fortgefahren werden soll.

4.3.4 Testprotokollierung

Die Testprotokollierung dient dem Zweck, die Testdurchführung auch für andere Personen, beispielsweise für den Kunden, nachvollziehbar zu machen. Weiterhin dienen die Protokolle als Absicherung, so dass das Unternehmen - für den Fall eines schweren Fehlers - beweisen kann, nicht leichtfertig mit dem Risiko eines Fehlers umgegangen zu sein, sondern dass es sich verantwortungsvoll bemüht hat, die Software so gut es ging auf Korrektheit zu prüfen.

Aus diesen Gründen muss die Durchführung der Tests exakt und vollständig protokolliert werden. In die Testprotokolle müssen neben dem Testobjekt auch eine Reihe weiterer Informationen aufgenommen werden: Wer hat wann getestet? Welcher Testrahmen wurde verwendet? Was für Eingabedateien wurden verwendet? Welche Ergebnisse wurden erwartet und welche wurden tatsächlich geliefert?

Die Daten sollten hierbei so verwaltet werden, das eine Wiederholung des Tests zu jedem späteren Zeitpunkt möglich ist - mit genau denselben Rahmenbedingungen und Testdaten.

4.3.5 Testauswertung

Danach kann zur Auswertung der Tests übergegangen werden. Unterscheiden sich erwartete Werte von den tatsächlich erhaltenen, so muss zunächst überprüft werden, ob tatsächlich ein Fehler in der Software vorliegt oder ob der Testfall selbst fehlerhaft war. Wurde tatsächlich ein Fehler aufgedeckt, so muss nun die Schwere des Fehlers bestimmt werden (siehe Tabelle 4.1). Nach Korrektur der Fehler müssen die Tests nochmals durchgeführt werden, um zu prüfen, ob der Fehler beseitigt wurde und ob durch die Beseitigung andere Fehler entstanden.

Am Ende einer Testreihe muss entschieden werden, ob das Testendekriterium erreicht wurde. Andernfalls müssen weitere Testfälle definiert werden, die den Test dem Testendekriterium näher bringen. Möglicherweise wird in diesem Stadium festgestellt, dass das Testendekriterium nicht oder nur mit zu hohem Aufwand erreichbar ist - beispielsweise wenn vollständige Überdeckung gefordert wird, das Programm aber unerreichbaren Programmtext enthält. In dem Fall sollte nachgeprüft werden, wie es zu dem unerreichbaren Programmtext kommen konnte, und entweder das Testendekriterium angepasst oder der Programmtext entfernt werden.

Klasse 1	Systemabsturz, ggf. mit Datenverlust; Das Testobjekt ist nicht einsetzbar
Klasse 2	Wesentliche Funktion ist fehlerhaft; Anforderungen falsch umgesetzt; Testobjekt ist nur mit großen Einschränkungen einsetzbar
Klasse 3	Funktionale Abweichung; Anforderungen fehlerhaft oder nur teilweise umgesetzt; System kann mit Einschränkungen genutzt werden
Klasse 4	Geringfügige Abweichung; System kann ohne Einschränkung genutzt werden; Fehler soll zum nächsten Release behoben werden
Klasse 5	Schönheitsfehler (beispielsweise Rechtschreibfehler oder Mangel im Maskenlayout); System ist ohne Einschränkung benutzbar

Tabelle 4.1: *Klassifikation der Schwere eines Fehlers*

4.4 Testfall Design

In diesem Kapitel wird detailliert beschrieben, wie man Software durch Ausführung der Testobjekte auf einem Rechner testen kann. Es werden verschiedene Vorgehensweisen zur Spezifikation der Testfälle vorgestellt und anhand von Beispielen verdeutlicht.

Hierbei wird ein besonderes Augenmerk auf Blackbox- und Whitebox-Verfahren gelegt. Whitebox-Verfahren eignen sich besonders gut für die unteren Teststufen, da die Orientierung am Programmtext für große Systeme kaum sinnvoll erscheint. Blackbox-Verfahren eignen sich hingegen für jede Teststufe. Auch jede Programmiermethode, die die Kodierung der Testfälle noch vor der Implementierung der Funktion fordert - wie beispielsweise Extreme Programming - beruht auf Blackbox-Verfahren.

4.4.1 Blackboxtest

Bei den *Blackbox-Verfahren* wird das zu testende Programmstück als schwarzer Kasten angesehen. Für diese Methode werden keinerlei Informationen über den ProgrammProgrammtext oder den inneren Aufbau der Funktionen benötigt, da nur die Funktion getestet werden soll. Aus diesem Grund wird das Blackbox-Verfahren auch *funktionales Testverfahren* genannt.

Auf welche Weise das Modul das gezeigte Verhalten implementiert interessiert hier nicht. Stattdessen beruhen die Überlegungen zu den Testfällen auf der Spezifikation.

Bei der gängigsten Methode des Blackbox-Verfahrens wird die Menge der möglichen Eingaben in *Äquivalenzklassen* unterteilt. Zu einer Äquivalenzklasse gehören all die Eingaben, von denen der Tester annimmt, dass das Testobjekt auf sie auf dieselbe Art reagiert. Da angenommen wird, dass sich das Testobjekt für alle Eingaben aus derselben Äquivalenzklasse auch gleich verhält, wird davon ausgegangen, dass es ausreicht, das Verhalten des Objekts nur mit einem Repräsentanten der Klasse zu testen. Wichtig ist hierbei, dass nicht nur Äquivalenzklassen für die gültigen sondern auch für die ungültigen Eingaben gebildet und getestet werden müssen.

Systematische Herleitung der Äquivalenzklassen

Um die Testfälle systematisch herzuleiten kann wie folgt vorgegangen werden. Für jede zu testende Eingabevariable wird der Definitionsbereich, also die Äquivalenzklasse aller zulässigen Werte, ermittelt. In diesem Definitionsbereich muss das zu testende Programmstück der Spezifikation gemäß reagieren. Zusätzlich muss sein Verhalten auch für unzulässige Werte, also Werte die sich außerhalb des Definitionsbereichs befinden, geprüft werden.

Im nächsten Schritt werden die Äquivalenzklassen verfeinert. Äquivalenzklassen, die Elemente enthalten, die laut Spezifikation unterschiedlich

verarbeitet werden müssen, werden bezüglich dieser Spezifikation in Unteräquivalenzklassen aufgeteilt. Diese Aufteilung erfolgt, bis sich alle unterschiedlichen Anforderungen mit den jeweiligen Äquivalenzklassen decken.

Danach wird für jede der Äquivalenzklassen ein Repräsentant für den Testfall ausgewählt sowie die zugehörigen erwarteten Ausgabedaten und gegebenenfalls die Vorbedingung für den Testlauf spezifiziert.

Überprüfung der Grenzen der Äquivalenzklassen

Die Äquivalenzklassenbildung und die Wahl der Repräsentanten ist sorgfältig vorzunehmen, da die Wahrscheinlichkeit, dass Fehler aufgedeckt werden, stark von dem jeweiligen Repräsentanten abhängen. Testfälle sind beispielsweise dann aussichtsreich, wenn sie die Grenzen der Äquivalenzklassen prüfen. Hier treten häufig Fehler auf, da aus umgangssprachlicher Formulierung nicht immer genau hervorgeht, welcher Grenzwert einer Äquivalenzklasse noch zuzuordnen ist. Eine umgangssprachliche Anforderung wie "bei weniger als 30km/h ... " in den Requirements kann mathematisch als $\leq 30km/h$ aber auch als $< 30km/h$ interpretiert werden. Wenn dann zwei der Designer das Requirement unterschiedlich deuten, so wird ein Fehler auftreten. Dieser Fehler kann durch einen zusätzlichen Testfall mit $x = 30km/h$ aufgedeckt werden.

Systematische Herleitung der Testfälle

Zur Spezifikation eines Testfalls muss jeder Eingabevariable ein Eingabewert zugeordnet werden. Hierfür muss entschieden werden, welche der verfügbaren Repräsentanten aus den ermittelten Äquivalenzklassen miteinander zu einem Eingabedatensatz zu kombinieren sind.

Folgende Regeln liefern eine sinnvolle Kombination [SL04]:

- Alle möglichen Kombinationen der Repräsentanten der **gültigen** Äquivalenzklassen sind miteinander zu kombinieren. Jede dieser Kombinationen bildet einen *gültigen Testfall*. Die Anzahl der gültigen Testfälle ergibt sich somit aus dem Produkt der Zahl der gültigen Äquivalenzklassen einer jeden Eingabevariable.

- Weiterhin sind Repräsentanten von **ungültigen** Äquivalenzklassen nur mit Repräsentanten gültiger Äquivalenzklassen zu kombinieren. Andernfalls kann die Fehlerwirkung nicht eindeutig zugeordnet werden oder es tritt sogar eine Fehlermaskierung auf.

Da im automotive Bereich ein Modul oft mehrere hundert Eingabevariablen besitzt, ergeben sich aufgrund der multiplikativen Kombination auch mit diesen Regeln noch viel zu viele Testfälle. Aus diesem Grund muss die Menge der gültigen Testfälle weiter reduziert werden. Möglichkeiten hierfür sind:

- Bevorzugung von Testfällen, die Grenzwerte enthalten.
- Zum Test nur benutzungsrelevante Testfälle heranziehen.
- Sicherstellung, dass jeder Repräsentant einer Äquivalenzklasse in mindestens einem Testfall vorkommt(Minimalkriterium).

Im automotive Bereich bietet es sich beispielsweise an, für jedes Signal nur die in den Requirements spezifizierten Parameter zu testen. Die nichtspezifizierten Eingabevariablen werden dann implizit mit 0 belegt. Allerdings muss man sich darüber im Klaren sein, dass auf diese Weise Randerscheinungen eventuell kaschiert werden.

In dem Beispiel der "Fahrerübernahmeanzeige" aus Abschnitt 4.4.3 und 4.4.4 tritt ein solches Problem auf. In diesem Signal ist eine weitere Funktionalität implementiert, die in den Requirements nicht spezifiziert wird. Diese Funktionalität wird in Abhängigkeit einer weiteren Eingabevariable angesteuert. Da diese Variable in den Requirements nicht erwähnt wurde, wurde diese Variable nicht explizit belegt, und daher von RTRT implizit mit 0 belegt. Aus diesem Grund wurde der Anweisungsbaum, der die zusätzliche Funktionalität implementierte, einfach übersehen.

In diesem Beispiel haben wir also Fehler übersehen, indem wir die Anzahl der Testfälle eingeschränkt haben. Dies ist aber leider ein Problem des Testens - absolute Sicherheit, keinen Fehler übersehen zu haben, erreicht man nur durch einen vollständigen Test. Sobald Eingabekombinationen nicht mehr getestet werden, kann nichts mehr über die Korrektheit des Systems gesagt werden, da nicht garantiert werden kann, dass nicht gerade diese Eingabe zu einem Fehler führt. Da die Zahl der Testfälle andernfalls aber unendlich groß werden kann, müssen solche Probleme toleriert werden.

4.4.2 Whiteboxtest

Whitebox-Testverfahren werden auch als *strukturelle Testverfahren* bezeichnet, da sie die Struktur des Testobjekts berücksichtigen. Die grundlegende Idee der Whitebox-Verfahren ist, dass alle Programmtext-teile eines Testobjekts mindestens einmal ausgeführt werden sollen. Daher muss der Programmtext vorliegen, so dass Testfälle entwickelt werden können, die sich an der Programmlogik orientieren. Das Erstellen der Testfälle richtet sich also nach dem Programmtext, die Bewertung, ob das Verhalten richtig ist, richtet sich aber weiterhin nach den Requirements.

Ziel des Whitebox-Verfahrens ist es, den Programmtext möglichst gut zu überdecken. Wann aber ist eine Überdeckung gut genug? Auch hier gibt es - wie bei Blackboxtest - viele unterschiedliche Verfahren und Metriken. Die gängigsten Testfallentwurfsmethoden sind die Folgenden:

- Anweisungsüberdeckung, Zweigüberdeckung

- Bedingungsüberdeckung
- Pfadüberdeckung

Weitere Überdeckungsarten, die von RTRT unterstützt werden, sind Funktionsaufruf- und Prozeduraufrufüberdeckungen, sowie eine Überdeckung von Befehlen wie “Goto” oder “Continue”. Diese Überdeckungsarten werden in Kapitel 5.1.2 kurz erläutert.

Für jede der Whitebox-Methoden muss zunächst der Kontrollflussgraph aus dem Programmtext des Testobjekts erstellt werden: Anweisungen werden als Knoten repräsentiert, die Reihenfolge, in der die Instruktionen abgearbeitet werden, als Kanten zwischen den zugehörigen Knoten. Anweisungssequenzen werden zu einem einzelnen Knoten zusammengefasst, da eine Ausführung der ersten Anweisung auch die Ausführung aller nachfolgen Anweisungen in der Sequenz bedingt. Anhand dieses Graphen können dann die Testfälle, die zu der gewünschten Überdeckung führen, gewählt werden. Die Requirements spezifizieren dabei das erwartete Verhalten. Mit Hilfe entsprechender Tools - wie beispielsweise mit RTRT - kann die Messung der Überdeckung erfolgen. Ist der in der Testplanung festgelegte Überdeckungsgrad erreicht, so wird der Test als ausreichend angesehen und kann beendet werden. Die verschiedenen Maße von Überdeckungsgraden werden jeweils an beispielhaften Kontrollflussgraphen verdeutlicht:

Anweisungsüberdeckung

In der *Anweisungsüberdeckung* stehen die einzelnen Anweisungen des Testobjekts, also die Knoten des Kontrollflussgraphen, im Vordergrund. Die Testfälle sollen eine zuvor festgelegte Mindestquote an Anweisungen, bzw. alle Anweisungen zur Ausführung bringen. Im Beispiel von Abbildung 4.3 könnten bereits alle Anweisungen überdeckt werden, indem man einen Testfall wählt, der den rot gefärbten Pfad zur Ausführung bringt. Die erreichte Anweisungsüberdeckung kann durch folgende Formel, dem “C0-Maß”, berechnet werden [SL04]:

$$\text{Anweisungsüberdeckung} = \left(\frac{\text{Anzahl der durchlaufenen Anweisungen}}{Gesamtzahl der Anweisungen}\right) \cdot 100\%.$$

Das C0-Maß ist nur ein schwaches Kriterium für die tatsächliche Güte der Überdeckung. In unserem Beispiel haben wir eine 100% Überdeckung durch einen einzigen Testfall erreicht, in anderen Programmen kann es sein, dass beispielsweise aufgrund vieler Ausnahmebehandlungen eine solche Überdeckung nur mit erheblichem Aufwand zu erreichen ist.

Außerdem kann es sein, dass auf diese Weise selbst bei 100%iger Überdeckung das Fehlen ganzer Anweisungsblöcke nicht erkannt wird - es werden ja nur die vorhandenen Anweisungen gezählt. Wenn beispielsweise beim Programmieren einer Bedingung zunächst nur erste Zweig codiert wurde während der else-Zweig auf später verschoben wurde, zu wird der leere else-Zweig in dieser Methode nicht ausgeführt. Falls der else-Zweig also unab-

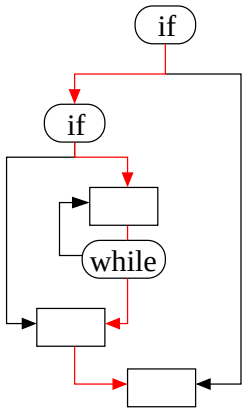

Abbildung 4.3: *Die Anweisungen dieses Kontrollflußgraphen werden durch einen Testfall, der den rot gefärbten Pfad ausführt, bereits vollständig überdeckt. In diesem Beispiel reicht also ein einziger Testfall aus, um eine vollständige Anweisungsüberdeckung zu erhalten.*

sichtlich leer blieb, so wird dieser Fehler beim Testen mit dieser Methode nicht aufgedeckt, obwohl der Fehler an sich sehr auffällig wäre. Das Erreichen einer 100%igen Anweisungsüberdeckung kann also ein trügerisches Gefühl der Sicherheit vermitteln.

Zweigüberdeckung

Ein weitergehendes Kriterium ist die *Zweigüberdeckung*. Hier stehen die Kanten des Kontrollflußgraphen im Vordergrund. Dabei spielt es keine Rolle, ob eine Kante zu einer weiteren Anweisung führt - es sollen so viele Kanten wie möglich abgedeckt werden, somit auch die der leeren else-Anweisung aus dem obigen Beispiel. Die Zweigüberdeckung bedingt, dass bei einer Verzweigung des Kontrollflusses alle Möglichkeiten und bei Schleifen auch die Umgebung bzw. ein Rücksprung zum Schleifenanfang zu berücksichtigen ist. Die farbig markierten Pfade in Abbildung 4.4 zeigen eine Möglichkeit für Testfälle, die zu einer vollständigen Zweigüberdeckung führen.

Der Überdeckungsgrad lässt sich mit folgender Formel, dem “C1-Maß”, berechnen: Zweigüberdeckung $= (\frac{\text{Anzahl der durchlaufenen Zweige}}{Gesamtzahl der Zweige}) \cdot 100\%$. In diesem Maß wird nur beachtet, ob ein Zweig durchlaufen wurde oder nicht - wie oft genau er durchlaufen wurde spielt hier keine Rolle. Dieses Kriterium ist schon deutlich aussagekräftiger als das Anweisungskriterium. Wie man sieht benötigt man hier bereits drei Testfälle um die vollständige Abdeckung zu erhalten, statt wie im vorigen Fall nur einen. Je nach erwartetem Risiko im Fehlerfall kann das Testendekriterium hier auf Werte zwischen 85% und 100% festgelegt werden, allerdings sollte eine Zweigüberdeckung von 100%

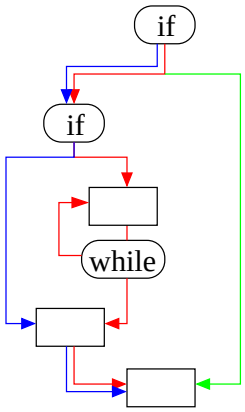

Abbildung 4.4: *Für eine vollständige Zweigüberdeckung werden bei diesem Kontrollflußgraphen bereits mindestens drei Testfälle benötigt, die die farbig markierten Pfade zur Ausführung bringen.*

angestrebt werden.

Bedingungsüberdeckung

Enthält ein Programm komplexe Bedingungen, dann sollten diese intensiv getestet werden, da gerade die Kombination logischer Ausdrücke fehleranfällig ist. Bei der Zweigüberdeckung wird jedoch nur der ermittelte Wahrheitswert einer Bedingung berücksichtigt, um so zu entscheiden, welcher Zweig als nächstes abgelaufen werden soll. Wie dieser Wert erreicht wurde ist hier nicht von Interesse. Setzt sich die Bedingung aber aus mehreren Teilbedingungen zusammen, so macht es Sinn, auch zu überprüfen, ob die Teilbedingungen richtig definiert wurden. Dies wird in der *Bedingungsüberdeckung* überprüft. Der Grad der Bedingungsüberdeckung kann noch weiter differenziert werden in *Einfache Bedingungsüberdeckung, Mehrfachbedingungsüberdeckung* und *Minimale Mehrfachbedingungsüberdeckung.*

Ziel der einfachen Bedingungsüberdeckung ist es, dass jede atomare Teilbedingung[2] im Test sowohl den Wert *true* als auch *false* annimmt. Allerdings wird nicht verlangt, dass bei der Auswertung der Gesamtbedingung unterschiedliche Wahrheitswerte erfolgen sollen. Es ist also möglich, dass so Zweige nicht abgedeckt werden. Dies kann am Beispiel von Abbildung 4.5 veranschaulicht werden. Das Beispiel kann bezüglich der einfachen Bedingungsüberdeckung durch die Belegungen $a = 1, b = 1, c = 0$ und $a = 0, b = 0, c = 1$ vollständig überdeckt werden. Die gesamte Bedingung würde allerdings in beiden Fällen mit *true* ausgewertet werden, es

[2]Eine Bedingung, die keine logischen Operatoren wie “AND”,“OR” oder “NOT” enthält, wird atomare Teilbedingung genannt.

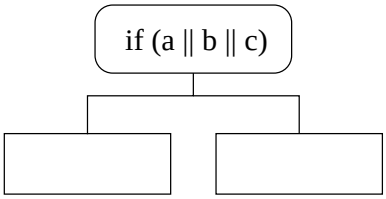

Abbildung 4.5: *In dieser zusammengesetzten Bedingung werden die drei atomaren Bedingungen a, b, c mit einem "oder" verknüpft. Diese Bedingung könnte mit zwei Testfällen bezüglich der einfachen Bedingungsüberdeckung vollständig überdeckt werden. Für die Mehrfachbedingungsüberdeckung würden acht Testfälle benötigt. Die minimale Mehrfachbedingungsüberdeckung befindet sich in der Mitte: hier würden vier Testfälle benötigt.*

würden also nicht beide Zweige ausgewertet. Daher ist die einfache Bedingungsüberdeckung ein schwächeres Kriterium als die Zweigüberdeckung.

Bei der Mehrfachbedingungsüberdeckung wird gefordert, dass auch Kombinationen der Wahrheitswerte der atomaren Teilbedingungen berücksichtigt werden. Hier sollen möglichst alle Variationen getestet werden, so dass sich auf diese Weise auch beide Wahrheitswerte für die Gesamtbedingung ergeben. Somit erfüllt diese Überdeckung auch die Kriterien der Zweig- und der Anweisungsüberdeckung. Sie ist sogar umfassender, da sie zusätzlich die Komplexität der zusammengesetzten Bedingungen prüft. Allerdings hat dies seinen Preis: Da die Anzahl an Kombinationsmöglichkeiten exponentiell in der Anzahl der atomaren Teilbedingungen steigt, wird ein solcher Test schnell zu aufwendig. In dem vorigen Beispiel müssten für eine vollständige Mehrfachbedingungsüberdeckung alle acht Kombinationsmöglichkeiten für die Belegungen der atomaren Bedingungen a, b, c geprüft werden.

Die minimale Mehrfachüberdeckung [Rie97] ist eine mögliche Lösung für das obige Problem. Hier müssen nicht alle Kombinationen geprüft werden, sondern nur diejenigen, bei denen Fehlerzustände nicht maskiert werden können. Dies sind dann genau die Kombinationen, bei denen die Änderung des Wahrheitswertes einer atomaren Bedingung den Wahrheitswert der Gesamtbedingung verändern kann. Im Beispiel von Abbildung 4.5 müssten die Belegungen $\{a = 1, b = 0, c = 0\}, \{a = 0, b = 1, c = 0\}, \{a = 0, b = 0, c = 1\}, \{a = 0, b = 0, c = 0\}$ gewählt werden, da bei diesen Belegungen die Änderung des Wahrheitswertes einer atomaren Bedingung zu einer Änderung der Auswertung der Gesamtbedingung führen kann. Bei allen anderen Belegungen müssten sich mindestens zwei atomare Bedingungen ändern, um den Wahrheitswert der Gesamtbedingung zu beeinflussen. Auf diese Weise kann die Anzahl an Testfällen erheblich verringert werden -

statt den $O(2^n)$ Testfällen[3] bei der Mehrfachbedingungsüberdeckung erhalten wir so zwischen $n + 1$ und $2n$ Testfälle. Die minimale Mehfachbedingungsüberdeckung ist das stärkste Kriterium, gleichauf mit der Mehrfachbedingungsüberdeckung. Allerdings benötigt sie deutlich weniger Testfälle, was sie sehr attraktiv macht. Der einzige Nachteil an dieser Methode ist, dass die Wahl der Eingabedaten um das geforderte Verhalten zu erhalten sehr aufwendig sein kann.

Allen Bedingungsüberdeckungen gemeinsam ist das Ziel eine 100%ige Überdeckung anzustreben.

Pfadüberdeckung

Ein Problem bei den Mehrfachbedingungsüberdeckungen bzw. der Zweigüberdeckung ist, dass jeder Zweig für sich betrachtet wird. Abhängigkeiten zwischen Zweigen werden hier nicht berücksichtigt. Bei der *Pfadüberdeckung* werden auch diese Abhängigkeiten berücksichtigt. Hier sollen alle Pfade abgedeckt werden, also alle möglichen Abfolgen von einzelnen Programmteilen des Programms. Sind Schleifen Teil des Programms, so zählt jede mögliche Anzahl an Schleifenwiederholungen als ein möglicher Pfad. Daher kann, sobald eine Schleife unbegrenzt oft aufgerufen werden kann, keine sinnvolle Messung der Überdeckung mehr erfolgen - es müssten ja unendlich viele Pfade abgedeckt werden. Aber auch sonst wird der Aufwand bei der Pfadüberdeckung schnell zu groß. Aus diesem Grund ist die Pfadüberdeckung eher als theoretisches Maß anzusehen. In dem Beispiel von Abbildung 4.6 sind nur die ersten vier Pfade eingezeichnet. Die Schleife würde hier zu unendlich vielen weiteren Pfaden führen. Wir haben in unserer Studie die Pfadüberdeckung vernachlässigt, da unsere Module eine zu komplexe Struktur aufweisen. Obwohl endlich viele Testfälle ausreichen würden, um eine vollständige Pfadüberdeckung zu erreichen, da unsere Software keine Schleifen enthält, wären extrem viele Testfälle notwendig gewesen, da die Funktionen sehr lang sind und mit vielen Verzweigungen arbeiten. Der Aufwand für eine vollständige Pfadüberdeckung erschien nicht gerechtfertigt, da in dieser Arbeit eine Teststrategie entwickelt werden soll, die nicht nur theoretisch einsetzbar ist.

4.4.3 Blackbox-Verfahren am Beispiel

Im Folgenden sollen die verschiedenen Verfahren am Beispiel des Signals “Fahrerübernahmeaufforderung” erläutert werden. Dieses Signal wird gesetzt, falls das ACC-System das Fahrzeug nicht mehr selbst ausreichend abbremsen kann. Das Signal soll den Fahrer dazu auffordern, die Fahrzeugführung wieder komplett zu übernehmen.

[3] n sei die Anzahl der atomaren Teilbedingungen

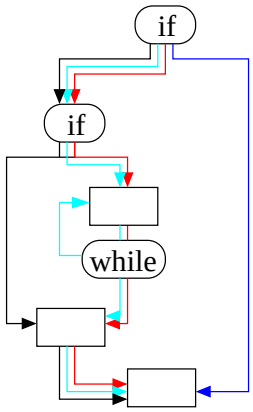

Abbildung 4.6: *In diesem Beispiel wurden die ersten vier Pfade eingezeichnet. Die Schleife könnte jedoch beliebig oft wiederholt werden, daher wären hier für eine vollständige Pfadüberdeckung unendlich viele Testfälle notwenig.*

Betrachten wir nun das folgende Requirement zum Signal "Fahrerübernahmeaufforderung":

Beispiel 3. Gegeben sei folgendes Requirement:
Es gibt drei Fälle in denen das Symbol beleuchtet werden soll:

1. *Die Verzögerung via ACC reicht nicht aus:*
   $GetTORactive() = T$ *(Anzeige nur für den Fall, dass der Fahrer nicht übertritt, also* $GetInternalState() \neq SuspendState$*)*

2. *PSS Latent Warnung liegt vor:*
   $GetLatentWarningFilt() = T$

3. *PSS Warnung liegt vor:*
   $GetPreWarningFilt() = T$

Die Anzeige soll abwechselnd im 400 ms Rhythmus blinken.

Hier fällt zunächst auf, dass das Requirement für einen Blackboxtest nicht genau genug ist. Fragen bleiben offen, wie beispielsweise: Wie heißt der Toggle-Timer? Wie heißt der Parameter, mit dem der Timer verglichen werden soll? Und in welcher Variable wird gesetzt, ob das Symbol leuchtet? Spätestens bei der Testfallspezifikation mit konkreten Werten müssen diese Namen bekannt sein. Aber beginnen wir zunächst mit der abstrakten Äquivalenzklasseneinteilung.

Die beteiligten Variablen sind: `GetTORactive()`,`GetInternalState()`,`GetLatentWarningFilt()`,`GetPreWarningFilt()`, `ToggleTimer`, `Parameter`; Zunächst muss der Definitionsbereich für jede Variable

ermittelt werden, also die Menge der zulässigen Eingabewerte. Dies wird in Tabelle 4.2 gezeigt.

$GetTORactive()$	T, F
$GetInternalState()$	$InitState, ReversibleFailureState, IrreversibleFailureState, RejectState, StandByState, ActiveControlState, SuspendState, \ldots$
$GetLatentWarningFilt()$	T, F
$GetPreWarningFilt()$	T, F
$ToggleTimer$	$0, \ldots, 255$
$SymbolLeuchtet?$	T, F

Tabelle 4.2: *Definitionsbereich der Variablen*

Danach müssen die Äquivalenzklassen gebildet werden. Die Klassen werden dann so lange unterteilt, bis sich alle unterschiedlichen Anforderungen mit den jeweiligen Äquivalenzklassen decken. Weiterhin muss dann für jede Klasse ein Repräsentant ausgewählt werden. Ein Beispiel, wie eine solche Unterteilung verlaufen kann, findet sich in Abbildung 4.7:

Die Klassen, bei denen "mindestens eine Fahrerübernahmebedingung" gilt, können weiter aufgeteilt werden in Klassen bei denen jeweils zumindest eine der Teilbedingungen ((`GetTORactive` $=$ T) $\wedge$ (`GetInternalState()` $\neq$ `SuspendState`)), (`GetLatentWarningFilt()` $=$ T) bzw. (`GetPreWarningFilt()` $=$ T) erfüllt wird. Im Prinzip gäbe es hier neun Kombinationsmöglichkeiten, allerdings könnte bei Eingaben, bei denen mehr als eine Bedingung erfüllt ist, eine Fehlermaskierung auftreten. Aus diesem Grund können wir uns auf die drei Kombinationen beschränken, bei denen immer genau eine Bedingung erfüllt ist. Die ersten vier Äquivalenzklassen werden also nochmals in insgesamt zwölf Klassen unterteilt.

Ebenso können die Klassen, in denen keine Fahrerübernahmebedingung vorliegt, weiter unterteilt werden. Hierbei kann man zwischen den Fällen, wo zwar `GetPreWarningFilt()`$= T$, aber `GetInternalState()`$=$ `SuspendState`, sowie denen, in denen das System zwar nicht im `SuspendState` ist, aber `GetTORactive` $= F$ ist, unterscheiden. Die vier Klassen werden also in acht unterteilt.

Auch die erhaltenen zwanzig Klassen könnten noch weiter unterteilt werden, da der `ToggleTimer` ebenfalls verschiedene Werte annehmen kann. Zumindest die Grenzen des Definitionsbereichs des Timers sollten zusätzlich geprüft werden, so dass wir durch die zusätzliche Unterteilung vierzig Testfälle erhalten. Die Unterteilung der ungültigen Äquivalenzklassen bleibt dem Leser überlassen.

Nach der Einteilung in Äquivalenzklassen müssen nur noch die Repräsentanten für die Testfälle gewählt werden, danach können die entsprechen-

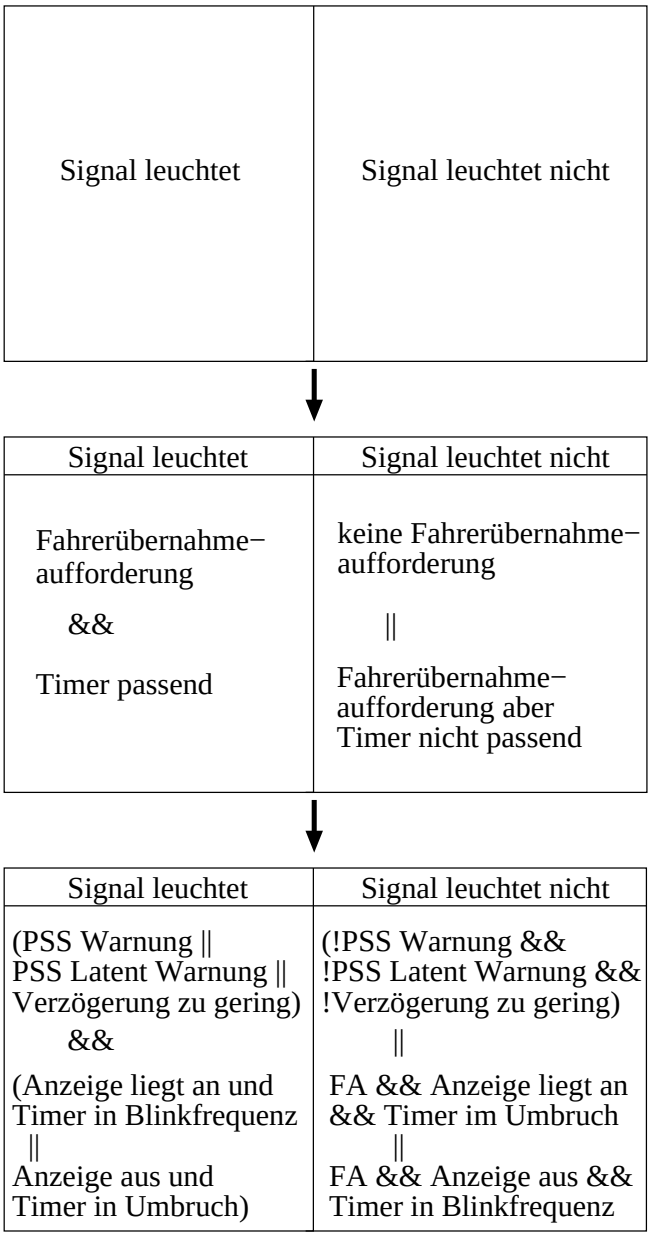

Abbildung 4.7: *In diesem Beispiel wird gezeigt, wie eine Einteilung in Äquivalenzklassen erfolgen kann. Zunächst werden nur zwei Klassen gebildet: eine, in der das Signal leuchtet und eine, in der es ausgeschaltet ist. Danach werden die Klassen weiter verfeinert. Leuchtet das Signal, so muß mindestens eine der Fahrerübernahmebedingungen gelten und der Timer muß sich in einer Blinkfrequenz befinden, oder gerade in eine solche überwechseln. Leuchtet es nicht, so gibt es dafür zunächst zwei Gründe: entweder es besteht keine Fahrerübernahmebedingung, oder aber es besteht zwar eine solche Bedingung, aber der Timer ist gerade in einer Nichtblinkphase. Die Klassen können noch weiter unterteilt werden.*

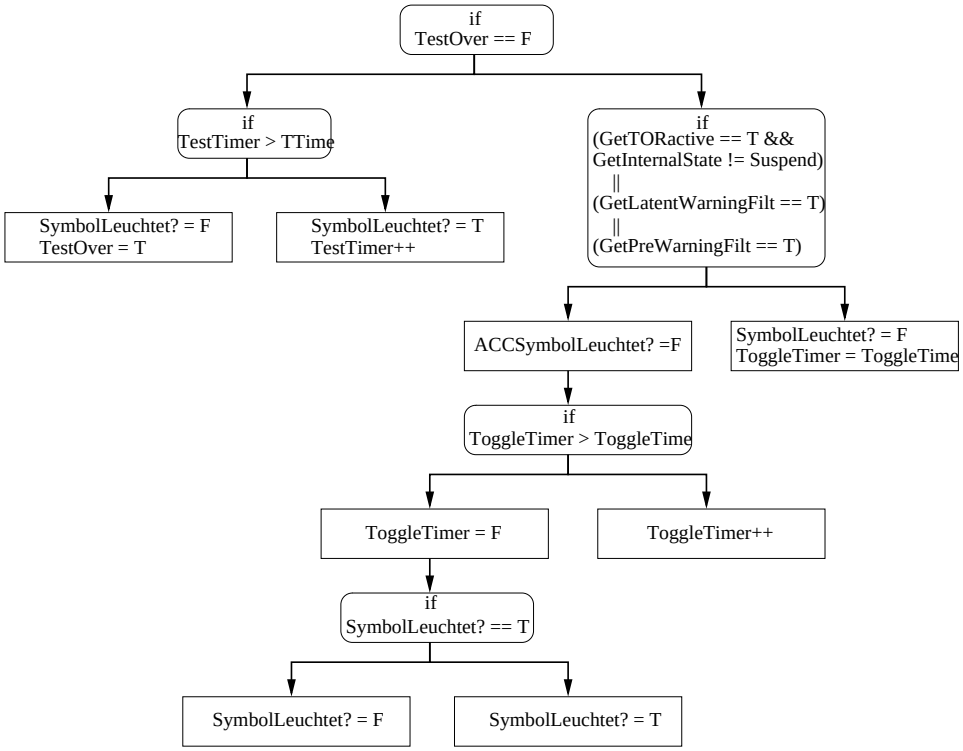

Abbildung 4.8: *Dieses Flussdiagramm wurde aus dem Programmtext für das Fahrerübernahmesignal abgeleitet. Beim Vergleich mit den Requirements fällt auf, dass keine Requirements für den linken Unterbaum bekannt sind.*

den Testfälle definiert werden. Wie die Umsetzung in der Skriptsprache von RTRT erfolgt, wird in Abschnitt A.2.1 dargestellt. Im folgenden Abschnitt wird gezeigt, wie dasselbe Requirement nach einer Whiteboxmethode getestet werden könnte.

4.4.4 Whitebox-Verfahren am Beispiel

Betrachten wir nun dasselbe Signal für das Whiteboxtesten. Zunächst wird aus dem Programmtext ein Flussdiagramm abgeleitet (vergleiche Abbildung 4.8).

Nun erkennt man, dass der ganze linke Zweig nicht durch die Requirements abgedeckt wird. Das gewünschte Verhalten in diesem Testzweig wird nirgends spezifiziert. Aus diesem Grund können für diesen Unterbaum noch keine Testfälle geschrieben werden. Zunächst muss beim Requirement-Manager geprüft werden, ob der Unterbaum seine Berechtigung hat und die Requirements vervollständigt werden müssen, oder ob der Unterbaum als Ganzes als Fehler gewertet werden muss.

Für den rechten Unterbaum liegen die Requirements bereits vor. Wir können also an diesem Baum die verschiedenen Whitebox-Testverfahren erläutern. Würden wir nur den ganz linken Pfad des Testbaums testen, so hätten wir bereits etwa 50% der Anweisungen überdeckt. Für die vollständige Überdeckung müssten noch die anderen Blätter getestet werden. In diesem Beispiel führen auch Zweig- und Pfadüberdeckung zu denselben Testfällen, da jeder Zweig zu einer separaten Anweisung führt und da das Programm keine Schleifen enthält.

Nur die Bedingungsüberdeckung führt zu anderen Testfällen. Bei der einfachen Bedingungsüberdeckung soll jede atomare Teilbedingung im Test sowohl den Wert *true* wie auch $false$ annehmen. Bei den letzten zwei Bedingungen werden so beide Zweige abgedeckt. Bei der ersten jedoch wäre es möglich, einen Test zu generieren, der nur den linken Zweig testet. Eine Fahrerübernahmebedingung liegt vor, falls (((`GetTORactive()=T`) $\wedge$ (`GetInternalState()` $\neq$ `SuspendState`)) $\vee$ (`GetLatentWarningFilt()` $= T$) $\vee$ (`GetPreWarningFilt()` $= T$)).

Wählt man also zwei Testfälle, so dass `GetLatentWarningFilt()` und `GetPreWarningFilt()` wechselseitig auf *true* bzw $false$ gesetzt werden(eines von beiden soll immer *true* sein) und die anderen Bedingungen einmal erfüllt und einmal unerfüllt sind, so wird beide Male die Gesamtbedingung als *true* ausgewertet und somit der linke Zweig gewählt. Dies zeigt, dass auch bei einer vollständigen einfachen Bedingungsüberdeckung nicht alle Anweisungen abgedeckt sein. Die einfache Bedingungsüberdeckung ist also ein schwächeres Kriterium als die Anweisungs- oder Zweigüberdeckung.

Bei der Mehrfachbedingungsüberdeckung hingegen sollen alle Kombinationen der möglichen Belegungen der atomaren Teilbedingungen geprüft werden. Die Bedingung besteht aus vier atomaren Teilbedingungen, die jeweils den Wert *true* oder $false$ annehmen können. Somit erhalten wir mit dieser Methode für die Bedingung bereits 2^4 Kombinationsmöglichkeiten.

Die minimale Mehrfachbedingungsüberdeckung beseitigt all die Kombinationen, wo eine fehlerhafte Auswertung der jeweiligen Teilbedingung maskiert wird, da sie keine Auswirkung auf das Gesamtergebnis hätte. In der obigen Bedingung erhielte man also drei Testfälle für den linken Zweig sowie zwei Testfälle für den rechten Zweig (siehe Tabelle 4.3). Somit hätten wir statt der 16 Kombinationen aus der Mehrfachbedingungsüberdeckung nur fünf Kombinationen zu testen.

In diesem Beispiel ist die minimale Mehrfachbedingungsüberdeckung das sinnvollste Kriterium. Die komplexe Bedingung wird genau getestet. Zweig- und Anweisungsüberdeckung - sowie in diesem Fall sogar die Pfadüberdeckung - werden durch dieses Verfahren subsumiert. Gleichzeitig benötigen wir dafür nur ca. 1/3 der Testfälle der normalen Mehrfachbedingungsüberdeckung. Die Umsetzung dieser Testfälle in der Skriptsprache von RTRT wird in A.2.2 dargestellt.

Linker Zweig	$GetTORactive() = T \wedge GetInternalState() \neq SuspendState \wedge Restfalse$	$\Rightarrow$ 1 Testfall
	$GetLatentWarningFilt() = T \wedge Restfalse$	$\Rightarrow$ 1 Testfall
	$GetPreWarningFilt() = T \wedge Restfalse$	$\Rightarrow$ 1 Testfall
Rechter Zweig	$(GetTORactive() = T \wedge GetInternalState() = SuspendState) \wedge Restfalse$	$\Rightarrow$ 1 Testfall
	$(GetTORactive() = F \wedge GetInternalState() \neq SuspendState) \wedge Restfalse$	$\Rightarrow$ 1 Testfall

Tabelle 4.3: *Testfälle zur minimalen Mehrfachüberdeckung*

4.4.5 Blackbox- versus Whitebox-Verfahren

Ein großer Vorteil der Blackbox-Verfahren ist, dass sich der Tester nur an den Requirements orientieren kann. Die Versuchung der Whiteboxtests, sich zu stark am Programmtext zu orientieren und vieles aus dem Programmtext als gegeben hinzunehmen ohne es noch sehr genau mit den Requirements abzugleichen, fehlt hier. Der Tester ist tatsächlich gezwungen, sich sehr genau mit dem Requirements auseinander zu setzen und sie gegebenenfalls zu hinterfragen, falls Unstimmigkeiten auftreten sollten. Unvollständigkeit oder Ungenauigkeiten der Requirements werden durch diese Methode schnell aufgedeckt - allerdings nur falls es sonst nicht möglich ist, überhaupt mit dem Testen zu beginnen. Scheinen die Requirements vollständig zu sein, und die entsprechenden Tests finden keinen Fehler, so kann es trotzdem passieren, dass ganze Anweisungsbäume so übersehen werden, denn beim Blackboxtesten kann nur das getestet werden, was in den Requirements spezifiziert wurde.

Ein weiterer Vorteil ist, dass hier die Funktionalität des Testobjekts im Vordergrund steht. Da die korrekte Funktion der Software höchste Priorität haben soll, sollte ein Blackbox-Verfahren im Testprozess keinesfalls fehlen.

Allerdings wird nur getestet, ob die Implementierung der Spezifikation entspricht. Ist die Spezifikation selbst bereits fehlerhaft, so wird das durch die Blackboxtests nicht erkannt. Allerdings ist das kein alleiniges Problem des Blackbox-Verfahrens, sondern ein generelles Problem, das beim Testen auftritt - also auch bei den Whitebox-Verfahren. Um solche Probleme zu finden, müssen Reviews durchgeführt werden.

Aber Blackboxtests haben noch drei weitere Nachteile: Sind die Requirements selbst nicht vollständig, so wird dies durch Blackboxtests nur in den seltensten Fällen erkannt. Ganze Anweisungsbäume können mit dieser

Methode übersehen werden, einfach, weil die entsprechenden Variablen, die an den Bedingungen zu diesen Bäumen teilhaben nicht in den Requirements spezifiziert wurden, und somit auch nicht belegt.

Der zweite Nachteil ist, dass, falls die Requirements nicht genau genug geschrieben wurden, auch die Tests vermutlich nicht die volle Funktionalität des Moduls prüfen werden. Bei der Bildung der Äquivalenzklassen versucht man ja, nur die Klassen genauer zu definieren, bei denen man dies für nötig hält. Generell möchte man aber, um Aufwand zu sparen, so wenig verschiedene Klassen wie möglich definieren.

Das größte Problem ist, dass durch Blackbox-Verfahren höchstens aus Zufall erkannt wird, falls das Testobjekt noch weitere, nicht vom Kunden geforderte, Funktionalität zur Verfügung stellt. Da die zusätzlichen Funktionen nicht in den Requirements auftauchen, werden auch keine Testfälle hierzu definiert, bzw. die Signale, die sich ebenfalls ändern, werden nicht geprüft. Da solche zusätzlichen Funktionen zu unerwünschtem Verhalten in seltenen Situationen oder zu Sicherheitsproblemen führen können, sollte nicht geforderte Funktionalität aus dem Programmtext entfernt werden. Da Blackbox-Verfahren diese nicht aufspüren kann, muss ein hierfür weiteres Verfahren angewendet werden.

Whitebox-Verfahren können diese Lücke füllen. Bereits beim Erstellen des Kontrollflussgraphen können, wie in unserem Beispiel gezeigt, ganze Anweisungsunterbäume auftauchen, deren Verhalten nirgends spezifiziert wird. Nicht geforderte Funktionalität kann auf diese Weise also leichter detektiert werden.

Ein weiterer Vorteil ist, dass auf diese Weise eine vollständige Programmtext-Überdeckung ermöglicht wird - welche Überdeckungsart gewählt wird, kann der Komplexität des Testobjekts angepasst werden. Die vollständige Programmtextüberdeckung kann bei Blackbox-Verfahren nicht gewährleistet werden.

Ein Nachteil der Whitebox-Verfahren ist, wie oben angesprochen, dass die Orientierung am Programmtext den Tester dazu verleiten kann, sich zuwenig auf die Requirements zu stützen. Oft finden sich im Programmtext noch Kommentare die ein eigentlich falsches Verhalten sinnvoll begründen - wenn der Tester dann nicht genau in den Requirements nachliest, kann es dazu kommen, dass er seine Testfälle selbst in Anlehnung an den Programmtext falsch spezifiziert. Der Fehler wird so natürlich nicht gefunden.

Außerdem verleitet der Programmtext dazu, sich bei Ungenauigkeiten der Requirements zunächst anzusehen, wie der Programmierer die Requirements gedeutet hat. Natürlich **sollte** der Tester sich selbst die Gedanken machen und gegebenenfalls beim Requirementmanager nachfragen, statt in den Programmtext zu sehen - allerdings, wenn der Programmtext sowieso vorliegt, so ist die Versuchung groß; Dann kann es schnell geschehen, dass der Tester sich die Lösung des Programmierers plausibel macht und selbst annimmt. Dass eine andere Interpretation der Requirements möglich gewe-

	Blackboxtest	Whiteboxtest
Vorteile	• Strenge Orientierung an den Requirements • Test der Funktionalität statt der Struktur	• 100% Programmtext Überdeckung möglich • Programmtext, zu dem es keine Requirements gibt, wird so erkannt
Nachteile	• Unvollständige Requirements können so kaum erkannt werden • Fehlende Genauigkeit der Requirements wird eventuell nicht erkannt • Ungewünschte Funktionalität kann übersehen werden	• Eventuell werden die Requirements vernachlässigt • Evt. zu starkes Vertrauen in den Programmtext

Tabelle 4.4: *Vor- und Nachteile von Blackbox- und Whiteboxtestverfahren*

sen wäre und dies an anderen Stellen im Programm zu Problemen führen kann, wird dann schnell übersehen.

4.4.6 Empfehlungen zum Testfalldesign

Beide Verfahren haben also unterschiedliche Vorteile und können die Nachteile gegenseitig kompensieren. Aus diesem Grund ist es ratsam, beide Verfahren anzuwenden. Sinnvollerweise sollte zunächst mit den Blackbox-Verfahren begonnen werden, da so die Requirements sehr genau geprüft und analysiert werden müssen, was sich später auch bei den Whiteboxtests positiv auswirken kann. Auch bei der Blackboxmethode sollten bereits Werkzeuge zur Messung der Programmtextüberdeckung verwendet werden. Danach können im Whiteboxtest gezielt die bisher nicht ausgeführten Teile des Testobjekts überprüft werden. Welche Art der Überdeckung als Kriterium für das Whiteboxtesten herangezogen werden soll, hängt sowohl von der Komplexität des Testobjekts wie auch von dessen Kritikalität für den Nutzer ab. Als minimales Kriterium sollte aber die Zweigüberdeckung gewählt werden - sobald das Testobjekt komplexe Bedingungen enthält ist zur minimalen Mehrfachüberdeckung zu raten. Weiterhin sollte darauf geachtet werden, dass, sofern Schleifen vorhanden, diese in den verschiedenen Testläufen auch mehrfach wiederholt werden.

4.5 Spezielle Anforderungen des Modultests

Beim Modultest soll überprüft werden, ob jeder Softwarebaustein für sich die Vorgaben der Spezifikation erfüllt. Soll dies am Rechner getestet werden, so muss die zu testende Komponente mit Eingabedaten versehen werden und zur Ausführung gebracht werden. Da aber nicht jede Komponente für sich bereits lauffähig ist, da beispielsweise andere Funktionen, die außerhalb der Komponente liegen, aufgerufen werden, muss das Testobjekt in einen *Testrahmen* eingebettet werden.

Ein *Testrahmen* ist eine Sammlung aller Programme, die notwendig sind, um Testfälle auszuführen, auszuwerten und Testprotokolle aufzuzeichnen. Zu diesem Zweck werden *Platzhalter* und *Testtreiber* benötigt. Die Platzhalter (engl. *stub*) simulieren das Ein-/Ausgabeverhalten der externen bzw. der noch nicht implementierten Programmteile. Statt eines Stubs können auch *dummys* oder *mocks* verwendet werden. Ein dummy bietet einen nahezu vollwertigen Ersatz für die tatsächliche Implementierung, ein mock bietet zusätzliche Funktionalität für Testzwecke.

Testtreiber sind Programme, die es ermöglichen, ein Testobjekt auszuführen, mit Testdaten zu versorgen und die Ausgabedaten des Objekts entgegen zu nehmen. Testtreiber und Platzhalter bilden gemeinsam den Testrahmen, der in Verbindung mit dem Testobjekt das ablauffähige Programm bildet.

Für unsere Studie verwendeten wir das Tool RTRT. Dieses Programm ist ein Testrahmengenerator, das heißt, es analysiert den Programmtext und erstellt dazu einen passenden Testrahmen. Der Tester muß sich daher nicht mit einer manuellen Erstellung eines Testrahmens beschäftigen sondern kann sich auf das Erstellen der Tests beschränken. Im folgenden Kapitel 5 wird dieses Programm genauer beschrieben.

Kapitel 5

Evaluierung von Rational Test RealTime

5.1 Vorstellung des Tools

In diesem Kapitel soll das Tool "Rational Test RealTime" vorgestellt werden. Die Software ermöglicht Modul- und Komponententests sowie Laufzeitanalysen für eingebettete und Echtzeit-orientierte Plattform-unabhängige Software. Ein beispielhaftes Test-Vorgehen mit dem Tool wird in Anhang A gezeigt.

5.1.1 Wie geht Rational Test RealTime vor?

Rational Test RealTime generiert den Testrahmen für das Testobjekt, das heißt, es stellt die Testtreiber sowie die Stubs für die externen Funktionen zur Verfügung und protokolliert die Testläufe. Damit die Tests Plattformunabhängig geschrieben werden können, verwendet das Tool einen "Target Deployment Port(TDP)", der den gewünschten Compiler, Linker, Debugger sowie die Zielarchitektur simuliert. Die Tests selbst sind unabhängig von dem TDP; Daher müssen diese nicht geändert werden, falls sich die Testumgebung(Compiler, Linker,...) ändern sollte. In Abbildung 5.1 wird ein solcher Testrahmen dargestellt.

Die Isolierung des zu testenden Moduls wird von RTRT automatisch vorgenommen. Bei der Erstellung eines neuen Projekts sucht die Software alle externen Schnittstellen des zu testenden Moduls und generiert mit Hilfe dieser Informationen Stubs und Testtreiber. Die externen Schnittstellen müssen jedoch in inkludierten Header-Files vorhanden sein, andernfalls treten Linker-Fehler auf.

Während der Erstellung des Testrahmens wird ein Testfile generiert, in dem die notwendigen Stubs deklariert werden sowie Basistests enthalten sind. Die Basistestscripts sind elementare Tests, die aus einem Aufruf der zu

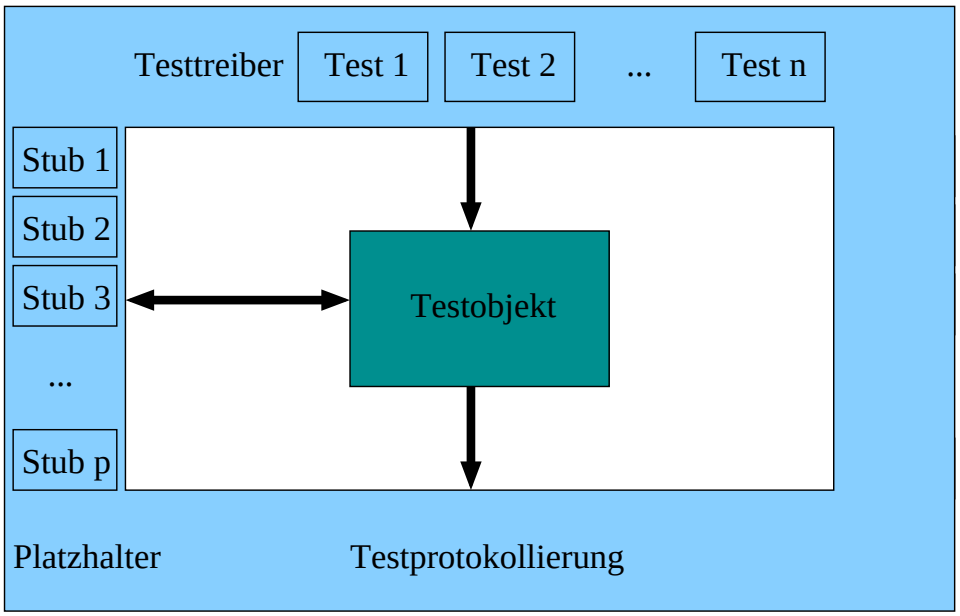

Abbildung 5.1: *Module müssen für sich nicht lauffähig sein: externe Funktionen müssen simuliert werden und es werden Testtreiber benötigt. Ein* Testrahmen *ist eine Sammlung aller Programme, die notwendig sind, um Testfälle auszuführen, auszuwerten und Testprotokolle aufzuzeichnen. RTRT generiert automatisch einen Testrahmen für den Modultest.*

testenden Funktionen bestehen. Ein erwartetes Ergebnis für die beteiligten Signale wird nicht definiert, daher können die Tests lediglich fehlschlagen, falls Stubs aufgerufen werden(Das Tool setzt die erwartete Anzahl an Aufrufen der Stubs defaultmäßig auf Null). Aus diesem Grund sind diese Tests nur als Ausgangsbasis für eine eigene Testerstellung zu verwenden.

Für jedes zu testende Modul wird bei der Testrahmengenerierung ein Testknoten erstellt. Dieser Knoten erhält als Kindknoten Tests, die die Funktionsweise dieses Moduls prüfen wollen. In Abhängigkeit von den ausgewählten Runtime-Analyse-Features wird der Programmtext instrumentiert, das heißt es wird weiterer Programmtext eingefügt, um die gewünschten Informationen berechnen zu können.

5.1.2 Welche Analysemöglichkeiten werden angeboten?

RTRT wurde speziell für Software-Entwickler entwickelt. Aus diesem Grund bietet die Software nicht nur Analysemöglichkeiten zur Testprotokollierung und Programmtext-Überdeckung sondern auch Features wie Memory Profiling, Runtime Tracing und Performance Profiling, die für einen Tester nur von untergeordneter Bedeutung sind. Im Folgenden sollen die Analysemöglichkeiten kurz vorgestellt werden.

Testprotokolle

Die Testprotokolle liefern eine hierarchische Zusammenfassung der Ergebnisse der Testdurchläufe. Ein Ausschnitt eines Testprotokolls findet sich in Abbildung 5.2. Bei jedem Testfall werden die spezifizierten Variablen mit Initialzustand, erwartetem Wert und erhaltenem Wert aufgelistet. Weiterhin werden, falls externe Funktionen aufgerufen werden, auch diese aufgelistet und gegebenenfalls getestet. Ein Testfall ist fehlgeschlagen, falls bei mindestens einer Variable der erwartete Wert nicht dem erhaltenen entspricht, oder falls eine externe Funktion anders als erwartet aufgerufen wurde.

Am Anfang der Protokolle werden die wichtigsten Informationen in einem Header zusammengefasst: welche Versionsnummer von Rational Test RealTime verwendet wurde, Pfad und Name des Projekts, die Gesamtzahl der Tests sowie die Gesamtzahl der fehlgeschlagenen Tests.

Programmtext Coverage

RTRT bietet drei verschiedene Aspekte zur Analyse der Programmtext-Überdeckung:

- eine allgemeine Übersicht, wieviel Prozent der gewählten Überdeckungsart über den gesamten Programmtext des Testmoduls erreicht wurde (vergleiche Abbildung 5.3);

1.3.338 - Test TPA_ACC_DEFECT-RESUME_2_1_1_1 (17/160)

1.3.338.1 - *Information*

Test Name	TPA_ACC_DEFECT-RESUME_2_1_1_1 (17/160)	Test Family	nominal
Status	Failed	Execution Time	111 micro sec.
Failed Variables	3		

1.3.338.2 - Element 1

1.3.338.2.1 - Variables

Variable	Status	Init Value	Expected Value	Obtained Value
((&sys_States_T20_packet_st)->pubFlagsAST_UN.w_ST .ActiveState_B)	Passed	1	?	1
((&sys_States_T20_packet_st)->State_UB)	Passed	8	?	8
((&sys_States_T20_packet_st)->OperationPrevent_UN.w_ST .FailOp_B)	Passed	1	1	1
((&sys_States_T20_packet_st)->switchOff_UN.w_ST.FailOff_B)	Passed	0	0	0

Abbildung 5.2: *In dieser Abbildung wird ein Ausschnitt aus einem Testprotokoll gezeigt. Bei jedem Testfall werden die spezifizierten Variablen mit Initialzustand, erwartetem Wert und erhaltenem Wert aufgelistet. Stimmen der erwartete und der erhaltene Wert überein, so wird die Zeile grün markiert, andernfalls rot. Ist ein Testfall fehlgeschlagen, so wird auch der dazugehörige Test als fehlgeschlagen markiert.*

- eine genauere Übersicht, welche Funktion welchen Überdeckungsgrad erreicht hat (vergleiche Abbildung 5.4);
- Programmtext, in dem markiert ist, welcher Programmtext bereits überdeckt wurde und welcher Programmtext noch nicht ausgeführt wurde (vergleiche Abbildung 5.5).

Es werden fünf verschiedene Überdeckungsarten angeboten, die ihrerseits noch weiter untergliedert werden können:

- Blocküberdeckung, die überprüft, ob alle Zeilen des Programmtexts überdeckt wurden;

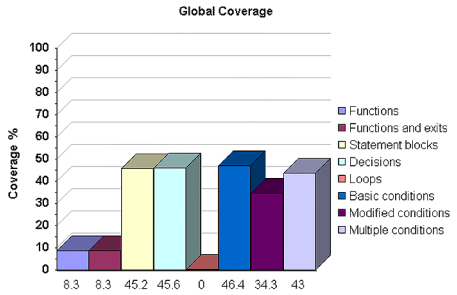

Abbildung 5.3: *Das Überdeckungsprotokoll bietet zunächst eine allgemeine Übersicht, welchen Überdeckungsgrad des Moduls die Tests erreicht haben.*

Display percentages and absolute values.

Item ▼	Functions	Functions and exits	Statement blocks	Decisions	Loops
DMA470IN.H	0.00 % 0/4	0.00 % 0/8	0.00 % 0/4	0.00 % 0/4	none
DMA_GetVersion_UL	0.00 % 0/1	0.00 % 0/2	0.00 % 0/1	0.00 % 0/1	none
DMA_Stop_V	0.00 % 0/1	0.00 % 0/2	0.00 % 0/1	0.00 % 0/1	none
DMA_Resume_V	0.00 % 0/1	0.00 % 0/2	0.00 % 0/1	0.00 % 0/1	none
DMA_GetActiveChannel_UL	0.00 % 0/1	0.00 % 0/2	0.00 % 0/1	0.00 % 0/1	none
E_INLINE.H	0.00 % 0/4	0.00 % 0/8	0.00 % 0/5	0.00 % 0/5	0.00 % 0/2
_suspendInterrupts	0.00 % 0/1	0.00 % 0/2	0.00 % 0/1	0.00 % 0/1	none
_resumeInterrupts	0.00 % 0/1	0.00 % 0/2	0.00 % 0/1	0.00 % 0/1	none
_returnWithoutResume	0.00 % 0/1	0.00 % 0/2	0.00 % 0/1	0.00 % 0/1	none
_countVal	0.00 % 0/1	0.00 % 0/2	0.00 % 0/2	0.00 % 0/2	0.00 % 0/2
DAG.C	25.00 % 1/4	25.00 % 2/8	46.70 % 133/285	47.00 % 135/287	none
dag_Init_V	0.00 % 0/1	0.00 % 0/2	0.00 % 0/1	0.00 % 0/1	none
dag_CalcAV_V	0.00 % 0/1	0.00 % 0/2	0.00 % 0/1	0.00 % 0/1	none
dag_CalcCV_V	0.00 % 0/1	0.00 % 0/2	0.00 % 0/1	0.00 % 0/1	none
dag_CalcMsgAcc2_V	100.00 % 1/1	100.00 % 2/2	47.20 % 133/282	47.50 % 135/284	none

Abbildung 5.4: *Das Überdeckungsprotokoll enthält weiterhin die hier dargestellte genaue Übersicht, welche Funktion des getesteten Moduls welchen Überdeckungsgrad - bezogen auf die jeweiligen gewählten Überdeckungsarten - erreicht hat.*

```
LOCFUNC VOID dag_CalcMsgAcc2_V(VOID
{
  /**/
  /* temporary variables */
  /**/
  ULONG t_temp_ul;
  UWORD t_temp_uw;
  UBYTE t_temp_ub;
  /* end temporary variables */

  /* Initialise temporary variables */
  t_temp_ul = (UL)0;
  t_temp_uw = (UW)0;
  t_temp_ub = (UB)0;


  /*****************************************************************************/
  /* TPA implementations start here                                            */
  /*****************************************************************************/

  /* If reversible Failure has gone while the according TPA is displayed    */
  /* and ACC is back in the AST_ACTIVE_CONTROL_STATE, than reset the        */
  /* TPA-timers and Display TPA 19 (XXX km/h)                               */
  /* TPAs represent indexes of column "TPA löschen, wenn ACC wieder aktiv"  */
  if(   (VDB_GetAstStateInternal_UB() == (UB)AST_ACTIVE_CONTROL_STATE)
    && (   (CIF_GetMsgIndexTPA_UB() == TPA_ACC_DEFECT)
        || (CIF_GetMsgIndexTPA_UB() == TPA_ACC_NOT_AVAILABLE)
        || (CIF_GetMsgIndexTPA_UB() == TPA_ESP_INTERVENTION)
        || (CIF_GetMsgIndexTPA_UB() == TPA_PARKING_BREAK)
        || (CIF_GetMsgIndexTPA_UB() == TPA_LOW_VELOCITY)
        || (CIF_GetMsgIndexTPA_UB() == TPA_NO_FORWARDGEAR)
        || (CIF_GetMsgIndexTPA_UB() == TPA_ACC_DEACTIVATION)
        || (CIF_GetMsgIndexTPA_UB() == TPA_VEHICLE_LOST)
        || (CIF_GetMsgIndexTPA_UB() == TPA_SHIFT_LEVER_POS)
        || (CIF_GetMsgIndexTPA_UB() == TPA_RPM_LIMIT)
        || (CIF_GetMsgIndexTPA_UB() == TPA_TAKE_OVER)
        || (CIF_GetMsgIndexTPA_UB() == TPA_VEHICLE_LOST)
    )  )
  {
```

Abbildung 5.5: *Zusätzlich gibt das Überdeckungsprotokoll die Möglichkeit, den SourceProgrammtext direkt zu analysieren. Hier wird der überdeckte Programmtext grün markiert, Bedingungen, die noch nicht vollständig überdeckt sind, werden gelb markiert, nicht besuchter Programmtext rot.*

- Funktionsaufrufüberdeckung, die überprüft, ob jede Funktion aufgerufen wurde;
- Bedingungsüberdeckung, die genauer prüft, wie genau eine Bedingung überdeckt wurde;
- Prozeduraufrufüberdeckung, die überprüft, ob jede Prozedur aufgerufen wurde;
- Überdeckung anderer Befehle wie Continue, Break, Goto,...

Hiervon können beliebig viele gleichzeitig ausgewählt und berechnet werden. In unserem Kontext sind die Block- und die Bedingungsüberdeckungen von Interesse. Die Funktions- und Prozeduraufrufüberdeckungen sind vornehmlich für den Integrationstest von Bedeutung, und die Überdeckung der Continue, Break, Goto Befehle benötigen wir nicht, da die Kodierregeln der hier betrachteten Software keine solchen Befehle zulassen. Aus diesem Grund können wir uns in diesem Kontext auf die Block- und Bedingungsüberdeckungen beschränken.

RTRT bietet drei verschiedene Blocküberdeckungen an: Anweisungsüberdeckung, Zweigüberdeckung und Schleifenüberdeckung. Ist die Schleifenüberdeckung gewählt, so sollen Schleifen null mal, einmal und mehrmals ausgeführt werden. In der Beispielsoftware wurden keine Schleifen verwendet - die getestete Komponente hat einen linearen Ablauf. Die Schleife wird durch einen Controler generiert: er ruft das Modul regelmäßig auf.

Die Funktionsaufruf-Überdeckung ist das schwächste Kriterium: hier wird nur festgestellt, ob jede Funktion des Testmoduls aufgerufen wurde.

Diese Überdeckung ist vornehmlich für den Integrationstest interessant, wo das Zusammenspiel verschiedener Module getestet werden soll. Für den Funktionstest ist diese Überdeckung aber kein starkes Aussagekriterium und wird daher in dieser Arbeit nicht weiter behandelt.

Im Gegensatz dazu sind die Bedingungs-Überdeckungsarten für den Modultest sehr interessant. RTRT bietet drei verschiedene Arten der Bedingungs-Überdeckungen an: die einfache Bedingungsüberdeckung, die Mehrfachbedingungsüberdeckung und die minimale Mehrfachbedingungsüberdeckung.

Eine genaue Beschreibung der verschiedenen Überdeckungsarten findet sich in Kapitel 4.4.2.

Memory Profiling

Dynamische Speicherverwaltung ist sehr fehleranfällig. Ein häufig auftretender Fehler ist, angeforderten Speicher nicht wieder frei zu geben oder versehentlich auf nicht angeforderten Speicher zu referenzieren. Diese Fehler können durch RTRT aufgedeckt werden.

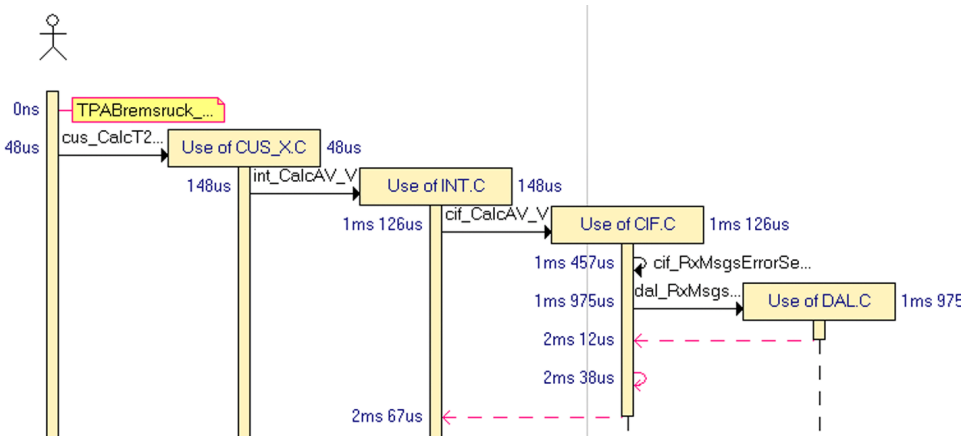

Abbildung 5.6: *In dem Runtime Tracing Protokoll werden die Funktions- oder Methodenaufrufsequenzen in einem UML-Diagramm graphisch dargestellt.*

Da allerdings in den Programmierrichtlinien des Beispielprojekts dynamische Speicheranforderung als nicht tolerabel definiert ist und die Einhaltung dieser Regel bereits von einem anderen Tool überprüft wird(vergleiche Kapitel 2.2.2), wird dieses Feature im Weiteren nicht näher betrachtet.

Performance Profiling

Für Echtzeit-Systeme ist Performanz ein sehr wichtiger Faktor. RTRT bietet die Möglichkeit, Informationen bereitzustellen, die dabei helfen, den Flaschenhals der zu testenden Funktion zu bestimmen. Beispielsweise werden Informationen gesammelt wie oft eine Funktion aufgerufen wird, wie lange sie ausgeführt wird, etc. Mit Hilfe dieser Informationen kann der Programmtext optimiert werden. In unserem Beispielprojekt wäre dieses Feature jedoch nur auf der Zielplattform von Interesse. RTRT bietet zwar die Möglichkeit, die Software auch auf der Zielplattform zu untersuchen, jedoch wurde diese Möglichkeit in dieser Arbeit nicht untersucht. Aus diesem Grund wurde auch dieses Feature nicht näher untersucht.

Runtime Tracing

Ist das Runtime Tracing Feature aktiviert so wird ein weiteres Fenster erzeugt, in dem Funktions- oder Methodenaufrufsequenzen in einem UML-Diagramm graphisch dargestellt werden. Dieses Sequenz Diagramm stellt alle Ereignisse dar, die während der Ausführung des Testobjekts auftreten. Ein Ausschnitt aus einem solchen Runtime Tracing Protokoll wird in Abbildung 5.6 gezeigt.

Die vertikalen Linien stellen die Lebenslinien von Source files oder In-

stanzen dar; die erste Lebenslinie, die zusätzlich durch ein Strichmännchen gekennzeichnet ist, repräsentiert die Welt, also das Betriebssystem.

Die Farben der Linien haben folgende Bedeutung:

- Grüne Linien stellen Aufrufe von Konstruktoren dar
- Schwarze Linien repräsentieren Methoden-Aufrufe
- Rote Linien sind Methoden-Rückgaben
- Blaue Linien zeigen Aufrufe von Destruktoren

Alle Linien sind mit einem Zeitstempel versehen und direkt mit den zugehörigen Programmstücken verlinkt.

Auf der linken Seite des UML-Diagramms befindet sich ein Balkendiagramm. Dieses soll anzeigen, wie gut der Programmtext in Abhängigkeit von der Ausführung überdeckt ist. Aus diesem Grund steigt der grüne Anteil im Laufe der Zeit. Allerdings ist dies nur ein sehr grobes Maß für die tatsächliche Programmtextüberdeckung.

Mit Hilfe des Runtime Tracing Features können unerwartete Ausnahmebehandlungen im Kontext betrachtet werden, wann diese auftreten. Falls sie durch fremden Programmtext hervorgerufen wurden, kann das mit diesem Tool schneller herausgefunden werden.

5.2 Komponententest mit RTRT

Wir haben RTRT vornehmlich für den Modultest verwendet. Ein Teil der Arbeit bestand aber auch darin, die Möglichkeiten des Tools für den Komponententest zu evaluieren. Die Ergebnisse fielen jedoch in unserem Fall negativ aus. Im Folgenden sollen die Vor- und Nachteile, die sich aus dem Komponententest mit Rational Test RealTime ergeben, erläutert werden.

Wir haben verschieden große Komponenten getestet. Der Aufwand fiel sehr unterschiedlich aus. Bei einigen kleineren Komponenten, mit nur drei bis vier Modulen, verlief die Generierung des Testbettes ebenso problemlos wie bei dem Modultest. Bei größeren häuften sich die Probleme. Das größte Problem ist, dass das Tool im Komponententest nicht mehr zuverlässig in der Lage ist, die benötigten Definitionen und Deklarationen der verwendeten Variablen und Funktionen zusammenzustellen. Oft müssen solche fehlenden Deklarationen nach Erstellung des Projekts vom Tester per Hand zusammengesucht werden - eine bei großen Komponenten sehr zeitraubende Tätigkeit. Ein weiteres Problem ist, dass manche der Definitionen in der Beispielsoftware nur in einer ".lib"-Datei vorliegen. Diese sind nicht immer mit der verwendeten Plattform kompatibel, und können in diesem Fall nicht integriert werden. In solchen Fällen mussten die Definitionen nachträglich nur für das Tool erstellt werden.

Neben dem größeren Initialaufwand ist auch der Testaufwand ungleich höher. Die Tests können nicht direkt vom Modultest übernommen werden, selbst wenn genau dasselbe Verhalten überprüft werden soll. Im Komponententest werden andere externe Funktionen verwendet, daher müssen die Platzhalter verändert werden, andernfalls würden alle Tests fehlschlagen, da andere Funktionen als erwartet aufgerufen würden. Selbst wenn die externen Funktionen nicht überprüft werden sollen, müssen diese in den Testfällen definiert werden. Im Modultest werden meist nur sehr wenig externe Funktionen aufgerufen - bei den Komponenten steigt die Anzahl jedoch rapide.

Wir wollten beim Komponententest die Komponente testen, in die die Displayfunktion integriert ist, und zwar so, dass gerade die CAN-Signale als Schnittstelle fungieren würden. Das Problem daran war, dass RTRT die Ein- und Ausgabeparameter der zu testenden Funktion in die Testprotokolle aufnimmt. In diesem Fall wurden jedoch ein großer Teil der CAN-Signale als Parameter übergeben - die Testprotokolle wurden dadurch sehr lang. Aus diesem Grund mussten die Testfälle auf kleine Services verteilt werden, um so nur wenige Tests ausführen zu lassen und die Testprotokolle halbwegs klein zu halten. Andernfalls stürzte das Tool ab, oder zumindest mussten sehr lange Wartezeiten (ca. 20Minuten für 200 Testfälle) auf die Darstellung der Protokolle eingerechnet werden. Da beim Modultest im schlimmsten Fall mit 0.5 Minuten zu rechnen sind, ist dies ein gravierender Unterschied.

Da die Protokolle in diesem Fall zu umfangreich werden, um noch dargestellt werden zu können, können die Tests auch nicht in der Nacht ausgeführt werden - die Ausführung selbst wäre zwar kein Problem, jedoch werden alle ausgeführten Tests von dem Tool in dasselbe Testprotokoll übernommen. Aus diesem Grund würde das entstehende Testprotokoll viel zu lang um von dem Tool dargestellt werden zu können. In diesem Fall blieben höchstens zwei Möglichkeiten: Entweder, das File würde von dem Tester in einem normalen Editor analysiert. Dies wäre sehr unkomfortabel, insbesondere da so mehrere Millionen Programmtextzeilen erzeugt werden. Die zweite Möglichkeit ist, dass ein Skript geschrieben würde, das nacheinander die Tests auswählt, ausführen lässt, und das Testprotokoll dann einzeln als html-Datei abspeichert. Mit Hilfe dieses Skripts könnten die Tests nachts ausgeführt werden, und der Tester könnte dann am folgenden Morgen die Protokolle analysieren. Ohne dieses Skript ist die Generierung und Analyse der Testprotokolle jedoch sehr zeitraubend und unkomfortabel.

Durch das Aufteilen der Testfälle auf verschiedene Services ergab sich ein weiteres Problem: RTRT wird instabil, falls ein Testknoten zu viele Kindknoten enthält. Um dieses Problem zu umgehen müssen weitere Testknoten eingefügt werden, so dass die Testfiles auf mehrere Knoten verteilt werden können. Die Testknoten können jedoch nicht einfach als Ganzes kopiert werden, sondern alle enthaltenen c-files und ptu-files müssen einzeln eingefügt werden. Da das Tool nach etwa acht neu eingefügten Testfiles instabil wird, ist auch hier der Zeitaufwand sehr hoch (unsere größte Komponente bestand

aus 23 c-Files und 32 ptu-Files).

Als Hauptvorteil hatten wir uns von dem Komponententest versprochen, dass die deutlich allgemeineren Requirements auf Komponentenebene für diese Tests ausreichen würden. Doch diese Idee erwies sich nicht als real: Um das Verhalten einer Funktion mit RTRT zu testen, müssen auch alle Variablen, die den Zustand des Systems darstellen, belegt werden. Soll eine Anzeige beispielsweise 10Sekunden blinken, so muss auch der Timer belegt werden, um testen zu können, dass das Signal tatsächlich blinkt. Aus diesem Grund werden ebenso viele Informationen benötigt, wie beim Modultest. Die Requirements müssen ebenso ausführlich sein, wie dort. Hier ergibt sich also kein echter Vorteil.

Ein Vorteil ist, dass die Signale sehr komfortabel verfolgt werden können. Das UML-Diagramm des Runtime-Tracing, das beim Modultest nicht sonderlich interessant ist, hilft im Komponententest sehr gut. Alle Funktionen, die als "instrumented" gekennzeichnet sind, werden in dem UML-Diagramm angezeigt, falls sie aufgerufen werden. So können gezielt die Funktionen ein- und ausgeblendet werden, die überprüft werden sollen. Weiterhin kann im Programmtext Coverage Report genau analysiert werden, welcher Programmtext überdeckt wurde, und es bestehen Links zu anderen Programmtextteilen (beispielsweise beim Aufruf einer anderen Funktion). Auf diese Weise können die Signale auch in der Komponente komfortabel verfolgt werden.

Zusammenfassend kommen wir zu dem Ergebnis, dass sich das Tool nicht besonders gut für unseren Komponententest eignet - zumindest nicht für große Komponenten. Nicht nur der Initialaufwand bei der Erstellung des Testbettes ist sehr hoch, auch die Erstellung der Testskripte ist verhältnismäßig mühsam im Vergleich zum Modultest, und das Tool wird instabil, wenn ein Testknoten zu viele Kinder besitzt. Auch die Analyse der Testprotokolle ist erschwert, da bereits ohne die zusätzlichen in den Tests spezifizierten Variablen sehr viele Eingangsgrößen getestet werden und die Protokolle daher sehr lang werden und vom Tool kaum noch dargestellt werden können. Durch all die Nachteile scheint der Aufwand nur in besonderen Fällen gerechtfertigt.

Die Probleme, die das Tool in unserem Fall beim Komponententest verursacht, ergeben sich aus der Struktur unserer Komponenten: die größeren Komponenten erhalten einen Großteil der CAN-Signale als Eingabeparameter. RTRT überprüft aber alle Signale aus den Eingabeparametern, so dass die Testprotokolle in dem Fall zu lang werden. Wollte man mit RTRT auch unsere Komponenten testen, so müßte deren Struktur geändert werden - ein sehr großer Aufwand.

Daher kommen wir zu dem Ergebnis, dass in unserem Fall das Tool vornehmlich für den Modultest oder für sehr kleine Komponenten verwendet werden sollte.

5.3 Bewertung des Tools

IBM wirbt damit, dass RTRT Testfälle automatisch generiert. Diese Werbung schafft allerdings sehr hohe und leider falsche Erwartungen. Das Tool generiert zwar tatsächlich Testfälle, allerdings bestehen diese nur aus einem Aufruf der Funktion, Variablen oder andere Signale werden hierbei nicht getestet. Diese Testfälle dienen also höchstens dazu, zu erkennen, ob das Projekt erfolgreich erstellt wurde - andernfalls würden beim Compilieren Fehler auftauchen. Als echte Testfälle kann man die automatisch generierten Testfälle aber nicht verwenden.

Die Testfälle müssen also doch per Hand von einem Menschen erzeugt werden. Dies ist eine zeitraubende Tätigkeit und außerdem können auf diese Weise auch die Testfälle selbst fehlerhaft werden. Eine Automatisierung wäre also wünschenswert. Es bestehen bereits verschiedenen Ansätze, die daran arbeiten, nach bestimmten Vorarbeiten aus formalen Anforderungen komplett automatisiert Testfälle erzeugen zu lassen. Allerdings sind die Verfahren noch nicht ausgereift.

Neben zufälliger Testdatenauswahl, die erfahrungsgemäß zu keiner guten Defekt-Überdeckung führt [Ham95], gibt es zwar auch bereits Whitebox- und Blackbox-Verfahren, die oft zu besseren Ergebnissen führen. Diese Verfahren sind aber noch nicht ganz ausgereift: Bei der Automatisierung der Testdatenauswahl mit Hilfe des Programmtexts treten noch einige Probleme auf, die deren praktische Anwendbarkeit wieder einschränken [AOS04]. Bei Blackbox-Testen ist zumindest eine Teilautomatisierung möglich [CCG+04]. Eine *vollautomatisierte* Testfallgenerierung ist jedoch in absehbarer Zeit für mittlere bis größere Systeme nicht zu erwarten [AOS04].

Doch selbst wenn die Verfahren ausgereift wären - eine der Voraussetzungen für die automatisierte Testdatengenerierung ist eine ordentliche, vollständige formale Spezifikation. Für die meisten Systeme ist es aber unmöglich sie formal zu spezifizieren, sei es aus Kostengründen, einem fehlenden Know-How der Entwickler bzw. des Kunden oder der schieren Komplexität des Systems. Daher steht eine vollautomatisierte Testfallgenerierung für die meisten Systeme nicht zur Verfügung. Das Problem liegt aber an der Komplexität des Problems und nicht an dem Werkzeug.

Ein weiterer Nachteil ist, dass das Tool nicht mit allen Makros umgehen kann. Werden statt normalen Variablen Makros verwendet, so kann RTRT diese zwar normalerweise problemlos auflösen, doch es treten Probleme auf, sobald eine Negation verwendet wird. In diesem Fall wird das Makro als “Expression” erkannt und somit kann nur noch der gewünschte Ausgabewert definiert werden, nicht jedoch der Eingabewert. Diesem Problem kann entgegengewirkt werden, indem in die Kodierregeln aufgenommen wird, dass solche Makros zu vermeiden sind, da sonst nicht jedes Verhalten einer Funktion mit dem Tool getestet werden kann.

Die Wiederverwendung von Testfällen spielt in der Industrie eine große

Vorteile	Nachteile
• Gute Eignung für den Modultest	• Kaum geeignet für Komponententest bei großen Komponenten mit CAN-Signalen als Eingabeparameter
• Automatische Isolierung des Testmoduls (vorausgesetzt die externen Schnittstellen sind in Headerfiles definiert)	• Kein Vergleich der Testauswertung zwischen Testläufen aus unterschiedlichen Projektständen möglich
• Automatischer Aufbau des Testrahmens	• Hoher Initialaufwand
• Beliebiger Compiler einsetzbar	• Keine echte automatische Testerstellung
• Automatisierte Testprotokollierung	
• Problemlose Wiederholbarkeit der Tests	
• Sehr gute Unterstützung des Whiteboxtests	

Tabelle 5.1: *Vor- und Nachteile des Tools Rational Test RealTime*

Rolle. Ein echter Vorteil des Tools ist die Plattformunabhängigkeit. Die Testfälle werden in einer plattformunabhängigen Sprache spezifiziert und nur bei Bedarf durch das Tool in tatsächlich ausführbaren Programmtext transformiert. Rational Test RealTime unterstützt hierbei fast jeden bekannteren Compiler für C/C++, Java oder Ada.

Ein Problem ist jedoch, dass ein Vergleich der Testergebnisse nur auf demselben Projektstand möglich ist. Soll also beispielsweise für jeden Checkpoint ein eigenes Projekt erstellt werden aber die unveränderten Tests wieder durchgeführt werden, so können die Tests zwar in das neue Projekt übernommen werden, aber der Vergleich, ob sich die Testfälle wie in dem vorigen Stand verhalten, muß von dem Tester durchgeführt werden. Zur Lösung dieses Problem kann jedoch ein Skript erstellt werden, dass den Vergleich vornimmt.

RTRT erstellt den Testrahmen innerhalb dessen das Testobjekt geprüft werden soll. Sind alle externen Funktionen zumindest in den Header-Files deklariert, so erstellt das Tool vollautomatisch die benötigten Testtreiber und Stubs. Dies bedeutet für den Tester einen großen Zeitgewinn - Zeit, die dann für das tatsächliche Testen verwendet werden kann.

Die Analysefunktionen sind komfortabel zu bedienen und leicht erlernbar. Zum Testen ist das Tool sehr gut geeignet, insbesondere der Programmtext Coverage Report ist hier sehr hilfreich. Durch die Markierung des ausgeführten Programmtexts wird schnell ersichtlich, welche Testfälle noch fehlen. Auch der gewünschte Überdeckungsgrad kann individuell eingestellt

werden und wird dann vom Tool protokolliert.

Wie im vorigen Abschnitt erläutert zeigt das Tool im Komponententest noch große Schwächen. Diese treten im Modultest jedoch nicht auf. Die beschriebenen Vor- und Nachteile des Tools finden sich in Tabelle 5.1 zusammengefasst. Die angesprochenen Nachteile des Tools im Modultest liegen teilweise in der Natur der Sache oder können durch eigene Skripts gelöst werden. Somit überwiegen die Vorteile des Programms. Aus diesem Grund kommen wir zu dem Ergebnis, dass sich das Tool für den Modultest sehr gut eignet.

Kapitel 6

Empirische Evaluierung

> Einem ist sie [die Wissenschaft]
> die hohe himmlische Göttin,
> dem Anderen eine tüchtige Kuh,
> die ihn mit Butter versorgt.
>
> Friedrich von Schiller

6.1 Relevante Aspekte der zugrunde liegenden Software

Dieses Kapitel präsentiert die Ergebnisse einer empirischen Studie, innerhalb deren wir verschiedene Testmethoden verglichen haben. Uns interessierte, welche Fehlerarten bei welchen Testmethoden entdeckt würden und wie sich verschiedene Überdeckungsgrade auf die Anzahl der entdeckten Fehler auswirken. Als Testobjekt verwendeten wir Software der Displayansteuerung für ein Fahrzeug, das mit Adaptive Cruise Control ausgestattet ist. In Kapitel 2.2.2 wurden die relevanten Aspekte der Software des ACC-Projekts bereits vorgestellt. In diesem Abschnitt sollen die wichtigsten Punkte noch einmal kurz zusammengefasst werden, bevor im nächsten Abschnitt die Methodik der Studie vorgestellt wird.

Die ACC Software ist in C geschrieben und wird mit dem Compiler Texas Instruments auf der Plattform ERCOSEK entwickelt. Alle "floating point" Operationen sind in Software realisiert, da diese nicht von der Plattform unterstützt werden.

Weiterhin enthält die getestete Software keine Schleifen und keine Rekursion. Die Module haben einen linearen Ablauf, auf Schleifen kann verzichtet werden, da ein Controler die Module in einem festen Taktzyklus aufruft. Goto- oder Continue-Konstrukte werden in den Kodierregeln der Software verboten. Aus diesem Grund haben wir uns bei der Studie auf die Analyse von Block- und Bedingungsüberdeckung beschränkt, Schleifenüberdeckung

oder die Überdeckung von Befehlen wie Goto, Continue oder Break wird hier nicht betrachtet.

Das System ist ein “embedded system”.

6.2 Präsentation der Studie

In dieser Arbeit wollten wir eine auf die ACC-Software abgestimmte Modultestmethode entwickeln. Hierfür haben wir zunächst verschiedene Testmethoden auf dem Programmtext des Moduls, das für die Displayansteuerung verantwortlich ist, angewendet und die Ergebnisse evaluiert. Untersucht haben wir die Unterschiede zwischen Blackbox- und Whiteboxtestmethoden, sowie beim Whiteboxtest zwischen verschiedenen Überdeckungsgraden. Gewertet haben wir dabei einerseits die Art und Anzahl der Fehler, die mit der jeweiligen Methode gefunden wurden, sowie andererseits die erreichten Überdeckungsgrade.

Für unsere Analyse haben wir drei Funktionen nach der Blackboxmethode getestet und drei Funktionen nach der Whiteboxmethode. Eine der betrachteten Funktionen wurde nach beiden Methoden getestet. Aus den gewonnen Ergebnissen entwickelten wir dann eine Test-Methodik. Diese überprüften wir danach an drei weiteren Funktionen des Moduls.

In diesem Kapitel sollen die Ergebnisse unserer Studie beschrieben werden.

6.2.1 Vorstellung der Methodik

Um Material für unsere Studie zu erhalten, haben wir drei Funktionen nach unserer Methode Blackbox getestet und drei Funktionen Whitebox getestet. Jeweils eine davon wurde sowohl nach Blackbox- wie auch nach Whiteboxmethoden getestet. Insgesamt wurden 1635 Zeilen Programmtext geprüft, ca 500 Zeilen davon wurden nach beiden Methoden überprüft, der Rest wurde zur Hälfte nach der Blackboxmethode und die andere Hälfte nach der Whiteboxmethode getestet. An weiteren 1174 Zeilen wurde das entwickelte Verfahren überprüft.

Als Blackboxmethode verwendeten wir das in Kapitel 4.4.1 vorgestellte Verfahren. Aus den Requirements wurden Äquivalenzklassen gebildet. Bei der anschließenden Testfallerstellung wurden insbesondere die Grenzfälle geprüft. Weiterhin wurden für die einzelnen Signale nur die in den Requirements spezifizierten Parameter kombiniert. Die nichtspezifizierten Eingabevariablen wurden implizit mit 0 belegt. Dies war notwendig, da das Modul mehrere hundert Eingangsvariablen hat, und somit die Anzahl an Testfällen sonst deutlich zu hoch geworden wäre. Allerdings wurden in unserer Studie durch dieses Vorgehen einige Fehler übersehen.

Das in Kapitel 4.4.2 und 5.1.2 vorgestellte Verfahren wurde für die Whiteboxmethode verwendet. Anhand des Kontrollflußgraphen erstellten

wir Tests für eine vollständige Anweisungsüberdeckung, Zweigüberdeckung, wie auch für eine vollständige einfache Bedingungsüberdeckung und eine minimale Mehrfachbedingungsüberdeckung. Da unser Programmtext keine Schleifen enthält verzichteten wir auch die Schleifenüberdeckung. Die Funktionsüberdeckung war für unsere Zwecke ein zu schwaches Kriterium, weshalb wir auch diese Überdeckung vernachlässigten.

Die Funktion, die nach beiden Methoden getestet wurde, wurde zuerst nach der Blackboxmethode und dann erneut nach der Whiteboxmethode getestet. Statt nur weitere Tests für die durch den Blackboxtest noch nicht überdeckten Programmtextfragmente zu erstellen, wurde bei dem Whiteboxtest vorgegangen als seien noch keine Blackbox-Tests erfolgt. Auf diese Weise wurde die Möglichkeit geschaffen, zu überprüfen, ob Fehler, die durch die Blackboxmethode gefunden würden, durch die Whiteboxmethode nicht aufgedeckt würden.

Anschließend prüften wir bei der Analyse der Ergebnisse, mit welcher Methode welche Fehlerarten gefunden wurden, sowie die Anzahl der gefundenen Fehler. Hier unterschieden wir nicht nur zwischen Whitebox- und Blackboxmethoden, sondern auch innerhalb des Whiteboxtests zwischen den unterschiedlichen gewählten Überdeckungsgraden. Tatsächlich kamen wir zu dem Ergebnis, dass sich die Art der gefundenen Fehler je nach Methode stark unterscheidet.

Durch die Blackboxmethode wurden vor allem Funktionsfehler aufgedeckt, wie auch fehlende Fehlerbehandlungen, die bereits in den Requirements nicht bedacht worden waren. Bei der Whiteboxmethode wurden neben normalen Funktionsfehlern auch Abhängigkeiten von weiteren Eingangsgrößen aufgedeckt, die laut Requirements nicht existieren sollten. Durch diese in den Requirements unbekannten Abhängigkeiten waren unerwünschte Zusatzfunktionen für die Blackboxtests versteckt geblieben, und erst beim Whiteboxtest entdeckt worden. Weiterhin wurde mit dieser Methode unerreichbarer und unnützer Programmtext entdeckt, sowie potentielle Fehlerquellen, die dadurch entstanden, dass das Verhalten mancher Signale in Abhängigkeit von falschen Eingangsgrößen bestimmt wurde - da diese Eingangsgrößen jedoch auf dieselben Speicherstellen verwiesen, wie die richtigen, trat keine Fehlerwirkung auf, und der Fehler war durch Blackboxmethoden nicht auffindbar. Eine genaue Erläuterung der Ergebnisse findet sich im folgenden Kapitel 6.2.2.

Aus den Ergebnissen der Analyse leiteten wir ein Testvorgehen ab, das an unsere Software angepasst ist. Da wir durch die Blackbox- und Whiteboxmethode sehr unterschiedliche Fehlerarten auffanden, und keine der Methoden alle Fehler der anderen Methode entdeckte, kamen wir zu dem Ergebnis, dass bei unserem Vorgehen eine Kombination beider Methoden notwendig ist.

Weiterhin wurden einige Fehler nur durch eine Bedingungsüberdeckung aufgedeckt. Eine Anweisungsüberdeckung ist daher zumindest bei sicher-

heitskritischen Modulen nicht ausreichend. Eine genaue Beschreibung der abgeleiteten Testmethodik findet sich in Kapitel 6.2.3.

Im Anschluß prüften wir unsere Testmethodik an drei weiteren Funktionen. Diese Funktionen entsprechen insgesamt 1174 Zeilen Programmtext, also etwa zwei Drittel des für die ersten Ergebnisse verwendeten Programmtexts. Da diese Funktionen weniger komplex waren, als die zuvor getesteten, lieferte der Blackboxtest in diesem Fall eine bessere Überdeckung und auch der Unterschied zwischen den Anzahlen der gefundenen Fehler variierte hier nicht so stark. Die Eigenschaften der Testmethoden, die bei den vorigen Tests zutage getreten waren, traten jedoch auch hier auf, und somit zeigte sich unsere abgeleitete Testmethodik als gut geeignet für Funktionen aus dem Modul.

Eine genaue Analyse dieser Ergebnisse folgt in Kapitel 6.2.4. Im Folgenden werden zunächst die Ergebnisse der Studie genauer beleuchtet.

6.2.2 Ergebnisse der Studie

In diesem Abschnitt sollen die Ergebnisse unserer Studie erläutert werden. Dabei beginnen wir mit einem Vergleich, wieviele Fehler je Methode und welche Fehlerarten durch die Blackboxtestmethode und welche durch die Whiteboxmethoden entdeckt wurden. Danach folgt ein Vergleich zwischen den verschiedenen Überdeckungsarten, um zu prüfen, ob sich der gewählte Überdeckungsgrad auf die Anzahl der gefundenen Fehler auswirkt. Zuletzt folgt eine kurze Zusammenfassung der Vor- und Nachteile der verschiedenen Methoden.

Vergleich der Fehlerarten und -zahlen

Bei unserer Analyse stellten wir fest, dass sich die Fehlerarten der durch die verschiedenen Methoden gefundenen Fehler stark unterscheiden. Durch die Blackboxmethode wurden Tests erstellt, die sich daran orientierten, was für Funktionalität implementiert sein *sollte*, bei der Whiteboxmethode hingegen wurde getestet welche Funktionalität tatsächlich vorlag. Somit unterschieden sich auch die Tests bei der Funktion, die sowohl Blackbox- wie auch Whitebox-getestet wurde, und die Tests entdeckten unterschiedliche Fehlerarten. Ein Überblick über die gefundenen Fehlerarten findet sich in Tabelle 6.1.

Durch die Blackboxmethode wurden elf Fehler entdeckt. Zusätzlich wurden durch diese Methode sieben fehlende Requirements entdeckt. Meist war nur das Verhalten des Moduls für den “Good case” definiert, was im Fehlerfall geschehen sollte, blieb undefiniert. Diese Fehler haben wir nicht direkt in unsere Fehlerstatistik aufgenommen, da aufgrund der fehlenden Requirements auch nicht geklärt werden konnte, ob sich der Programmtext an dieser Stelle korrekt verhielt, oder nicht. Wir haben die fehlenden Require-

Funktion	Blackboxtest	Whiteboxtest
Funktion 1	• 1 Designfehler • 2 Funktionsfehler	
Funktion 2	• 1 Designfehler	
Funktion 3	• 7 Funktionsfehler	• 4 Funktionsfehler, die auch durch BB-Methode entdeckt wurden • 2 weitere Funktionsfehler • 1 unerreichbarer Code • 1 unbenötigter Code • 3 Abfragebäume, die unerwünschte Zusatzfunktionen implementierten • 4 weitere unerwünschte Abhängigkeiten
Funktion 4		• 1 Vergleichsfehler • 2 Funktionsfehler
Funktion 5		• 1 Datenverarbeitungsfehler

Tabelle 6.1: *Vergleich der Fehlerarten nach angewendeter Testmethodik*

ments jedoch in unsere Fehlerdatenbank übernommen und als einen weiteren Fehler gezählt. Die anderen elf Fehler waren vor allem Funktionsfehler: Anzeigesignale wurden mit falschen Abhängigkeiten von den genannten Eingangsgrößen angesteuert oder das Prüfen von Freigabebits wurde vergessen.

Zwei der Fehler waren Designfehler: Eine Funktion sollte sich beim ersten Aufruf anders verhalten als danach - es gab aber keine Möglichkeit zu prüfen, ob es sich um den ersten Aufruf handelte. Der zweite Fehler war ein potentieller Fehler: Ist der Hauptschalter ausgeschaltet, so sollen keine Anzeigen erfolgen. Im Programmtext wird dies jedoch nicht überprüft, sondern es wird implizit angenommen, dass sich diese Information im Status des Systems wiederfindet. In dem momentanen Stand des Zustandübergangsdiagramms ist dies der Fall. Sollte sich hier jedoch etwas ändern, so könnte dies zu Fehlverhalten führen.

An diesem Beispiel wird auch eine Problematik deutlich: mit RTRT läßt sich jeder Zustand des Systems simulieren - auch solche, die von der Software niemals erreicht werden könnten. Ein Tester benötigt sehr viel Hintergrundwissen über das Projekt, um entscheiden zu können, welche entdeckten Fehler tatsächlich auftreten können, und welche Fehlerzustände nie erreicht werden können. Diese Problematik wird in Anhang 8 genauer erläutert. Im Rahmen dieser Diplomarbeit wurden für diese Fragen die Entwickler selbst zu Rate gezogen.

Während durch die Blackboxmethode vor allem fehlende Fehlerbehandlungen und Funktionsfehler aufgedeckt wurden, so wurden durch die Whiteboxmethoden neben Funktionsfehlern sieben Abhängigkeiten von weiteren Eingangsgrößen aufgedeckt, die laut Requirements nicht existieren sollten. Drei davon führten zu ungewünschte Zusatzfunktionen, die anderen zu Einschränkungen. Doch da diese Abhängigkeiten nicht in den Requirements spezifiziert worden waren, blieben sowohl die Zusatzfunktionen wie auch die Einschränkungen im Blackboxtest unentdeckt, und wurden erst beim Whiteboxtest aufdeckt.

Weiterhin wurden mit dieser Methode je einmal unerreichbarer und überflüssiger Programmtext sowie zwei potentielle Fehlerquellen entdeckt, die dadurch entstanden, dass das Verhalten mancher Signale in Abhängigkeit von falschen Makros bestimmt wurde - da diese Makros jedoch auf dieselben Speicherstellen verwiesen, trat keine Fehlerwirkung auf, und der Fehler war durch Blackboxmethoden nicht auffindbar. Tatsächlich wurde der Programmtext in einem späteren Stand jedoch so verändert, dass der Fehler auftrat - hier war der potentielle Fehler also wichtig. Insgesamt wurden mit dieser Methode 19 Fehler gefunden. Eine Funktion wurde zunächst nach der Blackboxmethode und danach nach der Whiteboxmethode getestet.

Durch die Blackboxmethode wurden hier sieben Fehler entdeckt, durch die Whiteboxmethode sogar fünfzehn Fehler. Interessanterweise bildeten die durch die Blackboxmethode gefundenen Fehler keine Teilmenge der durch die Whiteboxmethode gefundenen Fehler. Drei Fehler wurden nur durch die Blackboxmethode gefunden. Alle davon entstanden, da die Requirements nicht so genau analysiert worden waren und mehr auf den Programmtext geachtet worden war: Das Requirement, dass keine Anzeige erfolgen soll, falls der Hauptschalter ausgeschaltet ist, stand an einem übergeordneten Platz und wurde in den Tests gar nicht beachtet, bei den zwei anderen Fehlern verhielt sich der Fall ähnlich. Weiterhin wurden keine der fehlenden Fehlerbehandlungen entdeckt, die durch die Blackboxmethode aufgedeckt worden waren.

Dies zeigt, dass in beiden Verfahren unterschiedliche Fehlerarten gefunden werden. Beim Testen nach der Blackboxmethode wird Verhalten getestet, das den Requirements zufolge gezeigt werden sollte. Mit dieser Methode wird sehr genau nach den Requirements getestet, daher werden vermutlich weniger Requirements übersehen. Weiterhin eignet sich diese Methodik, fehlende Fehlerbehandlungen aufzuspüren. Dafür können Zusatzfunktionen höchstens zufällig entdeckt werden - sind im Programmtext noch zusätzliche Abhängigkeiten von weiteren Eingangssignalen vorhanden, so ist es sehr unwahrscheinlich, diese im Blackboxtest aufzuspüren. Da die Module mehrere hundert Eingangssignale haben, können in den einzelnen Tests nicht alle davon geprüft werden, sondern es muß sich bei der Testfallerstellung auf die für das Verhalten relevanten Signale beschränkt werden. Welche Signale davon relevant sind, muß in den Requirements spezifiziert sein. Somit wer-

den weitere, nicht in den Requirements erwähnten, Abhängigkeiten mit dem Blackboxtest kaum erkannt. Diese Fehler können dafür mit dem Whiteboxtest erkannt werden.

In unserem Programmtext wurden sieben weitere Abhängigkeiten gefunden, die laut Requirement nicht existieren sollten. Drei davon führten zu ganzen Anweisungsbäumen, die unerwünschte Zusatzfunktionen implementierten, und die beim Blackboxtest nicht entdeckt worden waren. Weiterhin eignet sich das Whiteboxtesten gut, um potentielle Fehler aufzudecken, wie im Fall der fälschlicherweise verwendeten Makros, die jedoch auf dieselbe Speicherstelle zeigten, wie die richtigen. Auch dieser Fehler konnte durch die Blackboxmethode nicht festgestellt werden. Zusätzlich kann der Programmtext mit dieser Methode von unnützen und unerreichbaren Programmtextfragmenten bereinigt werden. Da die Fehlerarten der durch die Methoden gefundenen Fehler also stark variieren, und keine der Methoden die andere Methode subsumieren kann, sollte eine Kombination beider Methoden gewählt werden.

Vergleich der Überdeckungsarten

Der zweite Punkt, den wir untersuchen wollten, ist ein Vergleich der verschiedenen Überdeckungsarten. Die Frage die sich stellt ist, ob ein höherer Überdeckungsgrad auch zu einer höheren Fehlerfindungsrate führt sowie, ob bestimmte Überdeckungsarten mehr Fehler zu Tage bringen. Während [HLL94] zu dem Ergebnis kam, dass vermutlich eine Korrelation zwischen der Anzahl der gefundenen Fehler und der erreichten Programmtextüberdeckung besteht, kam [BP99] zu dem Ergebnis, dass eine höhere Testüberdeckung nicht unbedingt zu einer höheren Fehlererkennung führen muß, sondern die erhöhte Fehlerfindungsrate auch an einer höheren Testintensität liegen kann.

Um diese offenen Fragen zu klären, untersuchten wir in diesem Rahmen vier Überdeckungsarten: die Anweisungs- und Zweig-Überdeckung der Blocküberdeckungsart sowie die einfache Bedingungsüberdeckung und die minimale Mehrfachbedingungsüberdeckung. Nicht bei jeder Funktion war eine vollständige Überdeckung möglich oder gewollt: In einer Funktion wurden als Eingangsgröße Makros verwendet, die eine Negation beinhalteten. RTRT kann Makros normalerweise problemlos auflösen, doch sobald eine Negation verwendet wird, wird das Makro als “Expression” erkannt und somit kann nur noch der gewünschte Ausgabewert definiert werden, nicht jedoch der Eingabewert. Aus diesem Grund konnte bei dieser Funktion nicht jedes Verhalten geprüft werden, sondern nur diejenigen, die unabhängig von diesen Makros waren. Bei dieser Funktion waren daher auch nur 75% des Programmtexts abdeckbar (bei Anweisungsüberdeckung).

Weiterhin entdeckten wir unerreichbaren Programmtext - dieser kann per Definition nicht überdeckt werden. Dieser Code war nur aufgrund der Spezi-

fikation von Eingangsgrößen nicht erreichbar, beispielsweise konnten manche Kombinationen von Eingangssignalen nicht vorkommen. Da der Code aber nur aufgrund der Semantik der Signale nicht erreichbar war, konnte er nur von einem Menschen als unerreichbar erkannt werden.

Für unnötige Abfragen - beispielsweise wenn bei einer komplexen Bedingung zunächst geprüft wurde, ob wir in einem von drei Zuständen waren und zusätzlich ob nicht in einem vierten - schrieben wir ebenfalls keine Testfälle für die unnötige zusätzliche Abfrage. Auch für offensichtlich falschen Programmtext, wie im Fall der unerwünschten Zusatzfunktionen, schrieben wir keine Testfälle. Aus diesem Grund haben wir nicht immer eine vollständige Überdeckung erreicht wie in Tabelle 6.2 ersichtlich. Da hier jedoch nur Programmtext nicht überdeckt wurde, der sowieso gelöscht werden sollte, könnte hier auch die Normierung geändert werden - denn der Programmtext, der weiter bestehen bleiben soll, wurde durchaus zu 100% überdeckt, bezogen auf jede der geprüften Überdeckungen.

Durch die Blackboxmethode erreichten wir in unserem Beispielprojekt im Schnitt eine zwischen 15 und 20% geringere Überdeckungsrate verglichen mit dem Whiteboxtest. Bei den Blocküberdeckungen bewegte sich der Unterschied im oberen Level, bei den Bedingungsüberdeckungen eher im unteren. Damit erreichten wir durch die Blackboxmethode nur eine mittlere Überdeckung[1]. Bei den kleineren und weniger komplexen Funktionen stieg der erreichte Überdeckungsgrad, eine Funktion wurde mit dieser Methode sogar vollständig überdeckt - bezogen auf jede der geprüften Überdeckungen. Doch je komplexer und länger der Programmtext, umso mehr sank der erreichte Überdeckungsgrad. Mit dieser Überdeckung wurden insgesamt elf Fehler gefunden, sieben davon in der Funktion, die sowohl nach der Blackboxmethode wie auch nach der Whiteboxmethode getestet wurden. Beim Whiteboxtest wurden insgesamt 19 Fehler entdeckt. Acht davon wurden bereits bei der normalen Anweisungsüberdeckung aufgedeckt. Hierzu gehörten beispielsweise die unerwünschten Zusatzfunktionen - die Existenz dieser Anweisungsbäume konnte mit den Requirements nicht erklärt werden, und somit konnten für diese Anweisungen keine Tests geschrieben werden. Auch der unerreichbare Programmtext wurde mit dieser Überdeckung bereits gefunden.

In zehn Fällen jedoch lag der Fehler in den Bedingungen versteckt, vier der Fehler wurden trotzdem bereits durch die Blocküberdeckungen entdeckt. Doch die anderen sechs konnten erst durch die Mehrfachbedingungsüberdeckung festgestellt werden. Einer dieser Fehler war eine unnötige Abfrage. Die anderen fünf entstanden durch Abhängigkeiten von weiteren Eingangsgrößen. Hier wurden also relevante Fehler durch die Mehrfachbe-

[1] Da der Aufwand für eine sehr hohe Überdeckung nicht linear steigt, ist auch der Überdeckungsgrad nach unserer Definition aus Abschnitt 7.4 nicht gleichmäßig verteilt. Wir setzen eine sehr niedrige Überdeckung auf 0-50%, niedrig: 50-65%, mittel: 65-85%,hoch:85-95%, sehr hoch: über 96%.

Funktion	Anweisung	Zweig	Einfache Bedingung	Min. Mehrfachbed.	Mehrfachbed.
Funktion 1 (BB)	74.6%	75.3%	76.9%	100%	85.7%
Funktion 2 (BB)	100 %	100%	100%	100%	100%
Funktion 3 (BB)	69%	69%	75.6%	75.8%	74.5%
Funktion 3 (WB)	96.6%	96.6%	96.2%	93.9%	90.9%
Funktion 4 (WB)	100%	100%	100%	100%	100%
Funktion 5 (WB)	100%	100%	92.9%	85.7%	88.2%
Funktion 6 (BB)	100%	100%	100%	100%	100%
Funktion 6 (WB)	100%	100%	100%	100%	100%
Funktion 7 (BB)	85.7%	90.9%	90%	90%	90%
Funktion 7 (WB)	100%	100%	100%	100%	100%
Funktion 8 (BB)	80%	87%	74.7%	61%	64%
Funktion 8 (WB)	94.5%	100%	84.4%	71%	74.8%

Tabelle 6.2: *Vergleich der erreichten Überdeckung*

Funktion	Fehlerart	Überdeckung
Funktion 3	• 4 Funktionsfehler, die auch durch BB-Methode entdeckt wurden	Anweisungsüberdeckung
	• 1 weiterer Funktionsfehler	(minimale) Mehrfachbedingungsüberdeckung
	• 1 weiterer Funktionsfehler	Anweisungsüberdeckung
	• 1 unerreichbarer Code	Anweisungsüberdeckung
	• 1 unbenötigter Code	(minimale) Mehrfachbedingungsüberdeckung
	• 3 Abfragebäume, die unerwünschte Zusatzfunktionen implementierten	Anweisungsüberdeckung
	• 4 weitere unerwünschte Abhängigkeiten	(minimale) Mehrfachbedingungsüberdeckung
Funktion 4	• 1 Vergleichsfehler	Anweisungsüberdeckung
	• 2 Funktionsfehler	Anweisungsüberdeckung
Funktion 5	• 1 Datenverarbeitungsfehler	Anweisungsüberdeckung

Tabelle 6.3: *Vergleich der entdeckten Fehlerarten nach angewendeter Überdeckungsmethodik*

dingungsüberdeckung hinzugewonnen. Ein Vergleich zwischen den durch die betrachteten Bedingungsüberdeckungen gefundenen Fehler führte zu dem Ergebnis, dass hier bei der Mehrfachbedingungsüberdeckung und der minimalen Mehrfachüberdeckung genau dieselben Fehler entdeckt wurden, die einfache Bedingungsüberdeckung fand etwas weniger Fehler, da hier nicht alle Anweisungen überdeckt wurden. In Tabelle 6.3 wird gezeigt, welche Fehler bei welcher Überdeckung gefunden wurden.

Weiterhin wurden die zusätzlichen durch den Whiteboxtest entdeckten Fehler fast gänzlich in dem durch die Blackboxmethode unabgedeckten Programmtext entdeckt - nur der potentielle Fehler befand sich in durch die Blackboxmethode abgedecktem Programmtext. Dieser hätte jedoch auch durch ein Programmtextreview entdeckt werden können. In diesem Fall wurden durch eine höhere Programmtextüberdeckung also auch mehr Fehler entdeckt. Wir kommen daher zu dem Ergebnis, dass eher der Theorie von [HLL94] zuzustimmen ist.

Zusammenfassung der Ergebnisse

Der Vergleich zwischen den Fehlerarten die durch die Blackboxmethode und denen die durch die Whiteboxverfahren entdeckt wurden, zeigte, dass beide Verfahren sehr unterschiedliche Fehlerarten entdecken. Durch die Blackboxmethode wird auf Funktionalität geprüft, die laut Spezifikation vorhanden sein sollte - aus diesem Grund eignet sich diese Methode insbesondere um fehlende Fehlerbehandlungen oder Spezialfälle aufzuspüren. Da der Tester mit dieser Methode auch nur die Requiremenents zur Verfügung hat, werden diese genauer analysiert und somit werden unerfüllte Requirements mit dieser Methode eher gefunden - beim Whiteboxtest wurden in unserem Fall drei solche Fehler übersehen, die durch die Blackboxmethode aufgedeckt worden waren. Auch unvollständige Requirements werden mit dieser Methode schnell entdeckt.

Die Whiteboxverfahren brachten andere Fehlerarten zu Tage: am schwersten fielen drei alte Anweisungsbäume ins Gewicht, die laut Requirements nicht mehr existieren sollten. Diese Zusatzfunktionen waren jedoch nie gelöscht worden. Durch die Blackboxmethode waren sie nicht auffindbar, erst durch die Whiteboxverfahren wurden sie entdeckt. Auch unerreichbarer und überflüssiger Programmtext konnte nur durch diese Methode entdeckt werden, sowie der potentielle Fehler, bei dem falsche Eingangsgrößen verwendet worden waren, die jedoch auf dieselbe Speicherstelle verwiesen wie die richtige. Da die Methoden also so unterschiedliche Fehlerarten entdecken, und beide Fehlerarten ähnlich wichtig sind, sollte bei dieser Software eine Kombination aus beiden Methoden für das Testverfahren angestrebt werden.

Beim Whiteboxverfahren bleibt weiterhin die Frage, welche Überdeckung gewählt werden sollte. In unserer Studie stellten wir fest, dass viele Fehler bereits durch die Anweisungsüberdeckung entdeckt werden konnten. Ein Viertel der Fehler wurde aber erst durch die Bedingungsüberdeckungen entdeckt. Der Zeitaufwand für eine vollständige minimale Mehrfachbedingungsüberdeckung ist jedoch deutlich höher als für die Anweisungsüberdeckung. Aus diesem Grund sollte hier das Risiko abgewogen werden: ist das Modul sicherheitskritisch, so sollte auf die minimale Mehrfachbedingungsüberdeckung nicht verzichtet werden, andernfalls könnte eine Anweisungsüberdeckung ebenfalls ausreichen.

Im Folgenden sollen die Folgerungen, die aus den Ergebnissen der Studie abgeleitet werden können, genauer erläutert werden.

6.2.3 Ableitung einer erfolgreichen Testmethodik

Da die Studie zeigte, dass durch die Blackboxmethode und die Whiteboxverfahren sehr unterschiedliche Fehlerarten aufgedeckt wurden, die jedoch in beiden Fällen hohe Relevanz aufweisen, sollte eine Kombination beider Verfahren verwendet werden. Durch die Blackboxmethode wird getestet, ob die

Funktionalität, die laut Spezifikation vorhanden sein soll, auch implementiert wurde - aus diesem Grund eignet sich diese Methode insbesondere um fehlende Fehlerbehandlungen oder Spezialfälle aufzuspüren. Da der Tester mit dieser Methode auch nur die Requiremenents zur Verfügung hat, werden diese genauer analysiert und somit werden unerfüllte Requirements mit dieser Methode eher gefunden - beim Whiteboxtest wurden in unserem Fall drei solche Fehler übersehen, die durch die Blackboxmethode aufgedeckt worden waren. Auch zum Aufspüren unvollständiger Requirements eignet sich die Blackboxmethode.

Die Whiteboxverfahren sollen insbesondere dazu verwendet werden, unerwünschte Zusatzfunktionen aufzuspüren, sowie unerreichbaren und unnötigen Programmtext zu entdecken. Auch potentielle Fehler, wie beispielsweise der bei dem falsche Eingangsgrößen verwendet worden waren, die jedoch auf dieselbe Speicherstelle verwiesen wie die richtige, können nur durch diese Methode entdeckt werden.

Dass bei der Whiteboxmethode drei Funktionsfehler übersehen wurden, die durch die Blackboxmethode entdeckt worden waren, zeigt ein inhärentes Problem der Whiteboxtests: Da dem Tester hier auch der Programmtext vorliegt, steigt das Risiko, dass die Requirements weniger genau analysiert werden und dem Programmtext "vertraut" wird. Diesem Problem kann begegnet werden, indem derselbe Tester eine Funktion zunächst nach der Blackboxmethode und danach nach der Whiteboxmethode testet. Die genaue Analyse der Requirements und das Melden unvollständiger Requirements im Blackboxtest bereitet so den Boden für einen besseren Whiteboxtest. Wurden die Blackboxtests zuvor von demselben Tester ausgeführt, so besteht eine gute Chance, dass ihm die notwendigen Anforderungen noch genauer im Hinterkopf bleiben, und solche Fehler daher seltener unterlaufen. Aus diesem Grund ist es sinnvoll, wenn derselbe Tester zunächst eine Funktion nach der Blackboxmethode und danach nach der Whiteboxmethode testet.

Weiterhin wurden die zusätzlichen durch den Whiteboxtest entdeckten Fehler fast gänzlich in dem durch die Blackboxmethode unabgedeckten Programmtext entdeckt - nur der potentielle Fehler befand sich in abgedecktem Programmtext. Somit scheint das Risiko, durch einen reduzierten Whiteboxtest, der nur die unabgedeckten Programmtextteile prüft, relevante Fehler zu übersehen, gering zu sein - da der Mehraufwand für einen vollständigen Whiteboxtest, der auch die Programmtextteile, die bereits durch den Blackboxtest geprüft wurden überdeckt, jedoch hoch ist, lohnt dieser Aufwand im Normalfall nicht. Daher kommen wir zu dem Schluß, dass nur bei den sicherheitskritischen Modulen ein vollständiger erneuter Whiteboxtest in Erwägung gezogen werden soll, andernfalls sollte die Whiteboxmethode nur gezielt für den noch unabgedeckten Programmtext eingesetzt werden.

Beim Whiteboxverfahren bleibt weiterhin die Frage, welche Überdeckung gewählt werden sollte. In unserer Studie stellten wir fest, dass viele Fehler

bereits durch die Anweisungsüberdeckung aufgedeckt wurden. Ein viertel der Fehler wurde jedoch erst durch die Bedingungsüberdeckungen gefunden. Dies zeigt, dass komplexe Bedingungen fehleranfällig sind. Sind daher komplexe Bedingungen im Programmtext enthalten, so müssen diese intensiv getestet werden. Der Zeitaufwand für eine vollständige minimale Mehrfachbedingungsüberdeckung ist jedoch deutlich höher als für die Anweisungsüberdeckung. Aus diesem Grund sollte hier das Risiko abgewogen werden: ist das Modul sicherheitskritisch, so sollte auf eine vollständige minimale Mehrfachbedingungsüberdeckung nicht verzichtet werden, andernfalls könnte eine niedrigere Überdeckung ebenfalls ausreichen. Enthält das Programm keine komplexen Bedingungen, so reicht eine Anweisungsüberdeckung aus. Bei weniger komplexen Modulen kann daher ein Blackboxtest bereits ausreichend sein - da dieser bei einfachen Modulen bereits eine vollständige Anweisungsüberdeckung erreichen kann. Eines der getesteten Module erreichte bereits durch die Blackboxtests eine vollständige Überdeckung. In diesem Fall müssten natürlich keine weiteren Whiteboxtests durchgeführt werden.

Eine vollständige Überdeckung kann nicht immer erreicht werden - bei unerreichbarem Programmtext kann dieser per Definition nicht abgedeckt werden, bei unnötigem oder unerwünschtem zusätzlichem Programmtext macht es wenig Sinn diesen mit abzudecken. Dieser Programmtext muß sowieso gelöscht werden. Jedoch sollte bei sicherheitskritischen Modulen eine vollständige Überdeckung des Programmtexts, der beibehalten werden soll, angestrebt werden - sowohl bezüglich der Anweisungs- wie auch bezüglich der minimalen Mehrfachbedingungsüberdeckung. Bei weniger sicherheitskritischen Modulen kann zunächst der geforderte Überdeckungsgrad bezüglich der Bedingungsüberdeckung gesenkt werden, bis hin zur reinen Anweisungsüberdeckung. Bei sehr stabilen und einfachen Modulen ist auch ein Blackboxtest ausreichend.

Da die Methoden unterschiedliche Fehlerarten entdecken, und beide Fehlerarten wichtig sind, sollte eine Kombination beider Methoden für das Testverfahren verwendet werden. Da der Whiteboxtest den Tester dazu verleitet, die Requirements weniger genau zu analysieren und stärker anhand des Programmtexts zu arbeiten, sollte diese Methode erst nach dem Blackboxtest angewendet werden. Auf diese Weise könnte diese Problematik eingegrenzt werden, da sich der Tester in diesem Fall bereits bei den Blackboxtests sehr genau mit den Requirements auseinander setzen mußte. Allerdings ist dieses Argument nur stichhaltig, falls derselbe Tester die Blackbox- und Whiteboxtests vornimmt. Aus diesem Grund wird vorgeschlagen, dass beide Verfahren von demselben Tester angewendet werden, und die Tester also nur die zu testenden Funktionen untereinander aufteilen, statt das methodische Vorgehen aufzuteilen.

Funktion	Blackboxtest	Whiteboxtest
Funktion 6	• 1 Datenverarbeitungsfehler	• 1 Datenverarbeitungsfehler
Funktion 7	• 1 Designfehler	• 1 Designfehler
Funktion 8	• 2 Funktionsfehler	• 4 Funktionsfehler • 1 unerwünschte zusätzliche Abhängigkeit

Tabelle 6.4: *Vergleich der Fehlerarten, die bei der Überprüfung der Methodik entdeckt wurden*

6.2.4 Überprüfung der Testmethodik

Um unsere Testmethodik zu überprüfen, testeten wir drei weitere Funktionen desselben Moduls zunächst nach unserer abgeleiteten Testmethodik. Der Code dieser Funktionen war bis dahin unbekannt. Zum Überprüfen der Methodik testeten wir die Funktionen zunächst nach der Blackboxmethode. Anschließend überdeckten wir den noch unabgedeckten Programmtext durch die Whiteboxmethode. Als Test für die Methodik schrieben wir danach weitere Tests nach der Whiteboxmethode, die nocheinmal den gesamten Programmtext überdeckten - auch den, der zuvor nur durch die Blackboxmethode überdeckt wurde. Auf diese Weise wollten wir überprüfen, ob wir durch die Eingrenzung der Whiteboxtests auf den nicht bereits abgedeckten Programmtext Fehler übersehen haben. In Tabelle 6.4 wird dargestellt, welche Fehlerarten auf diese Weise entdeckt wurden.

Die für den Verfahrenstest verwendeten Funktionen beliefen sich auf insgesamt 1174 Zeilen Programmtext, also etwa 500 Zeilen weniger als bei dem vorigen Test. Sie waren weiterhin weniger komplex als die zuvor getesteten Funktionen. Daher rechneten wir bereits mit weniger Fehlern als zuvor. Der Unterschied der Anzahl der gefundenen Fehler mittels der zwei Methoden verhielt sich ähnlich wie in dem vorigen Fall: dort wurden bei der Funktion, die nach beiden Methoden getestet wurde, 7 Fehler durch die Blackboxmethode und 15 Fehler durch die Whiteboxmethode entdeckt. Die Fehler standen also in einem Verhältnis von 7 : 15, also ca 1 : 2. In dem Überprüfungstestset wurden durch das Blackboxverfahren vier Fehler entdeckt, sieben durch das Whiteboxverfahren, auch hier wurde also etwa das Verhältnis 1 : 2 erhalten, wenn auch hier prozentual mehr Fehler bereits durch die Blackboxmethode entdeckt worden waren. Durch die Whiteboxtests wurden hier also etwas weniger weitere Fehler entdeckt.

In diesem Fall wurden alle Fehler, die durch die Blackboxmethode gefundenen wurden, auch durch die Whiteboxtests entdeckt. Allerdings wurden hier durch die Blackboxmethode auch nur Funktionsfehler sowie ein Datenverarbeitungsfehler entdeckt - es wurden keine Fehler der Require-

ments gefunden, die durch die Whiteboxtests schwierig zu entdecken wären. Die Whiteboxtests fanden drei zusätzliche Fehler: einen Kodierfehler, bei dem statt eines “if else” nur ein “if” verwendet wurde, sowie eine weitere Abhängigkeit und einen potentiellen Fehler, der aufgrund impliziter Annahmen im aktuellen Stand keine Fehlerwirkung verursachen kann. Auch hier befanden sich die Fehler nur in durch die Blackboxtests unabgedecktem Programmtext. Die niedrigere Komplexität der Funktionen äußerte sich auch in den Überdeckungsgraden, die bereits durch die Blackboxmethode erreicht wurde. Eine Funktion erreichte bereits durch diese Methode eine vollständige Überdeckung. Bei den anderen Funktionen belief sich der Unterschied auf etwa 8 bis 12% - verglichen mit den zuvor erhaltenen Differenzen von 15-20% also ein deutlicher Unterschied. Allerdings waren diese Funktionen wie erwähnt weniger komplex als die zuvor getesteten. Somit passen die Ergebnisse doch gut in das Bild, das wir durch unsere vorigen Analysen erhalten haben: Bei weniger komplexen Funktionen sind Blackboxtests effizienter als bei sehr komplexen.

Die Überprüfung mit den zusätzlichen, von den Blackboxtests unabhängigen, Whiteboxtests zeigte dass keine Fehler durch die Beschränkung der Whiteboxtests auf den unabgedeckten Programmtext übersehen worden waren - bei allen drei Funktionen wurden durch die zusätzlichen Whiteboxtests genau dieselben Fehler entdeckt wie bei unserem ersten Testvorgehen. Somit scheint das Risiko, durch den reduzierten Whiteboxtest relevante Fehler zu übersehen, gering zu sein - da der Mehraufwand für einen vollständigen Whiteboxtest, der auch die Programmtextteile, die bereits durch den Blackboxtest geprüft wurden überdeckt, jedoch sehr hoch ist, lohnt dieser Aufwand im Normalfall also nicht.

Aus diesem Grund kommen wir zu dem Ergebnis, dass unser Testvorgehen sinnvoll ist, und sich für die verwendete Software gut eignet.

Kapitel 7

Der Einsatz von Metriken im Testzyklus

> Es gibt nichts Praktischeres als eine gute Theorie
>
> Immanuel Kant

Nehmen Sie an, Sie seien Testmanager für den Modultest der ACC-Abteilung bei Bosch. Ihre Aufgabe ist es, zu entscheiden, wann eine Komponente sicher genug ist, um freigegeben zu werden. Die Sicherheit der Komponente kann beispielsweise begründet sein durch eine Schätzung der Anzahl von sicherheitskritischen Fehler, die die Komponente in den nächsten zehn Jahren verursachen könnte. Aufgrund der Sicherheitsrelevanz muss Ihre Entscheidung gut begründet sein.

Für solche Aufgaben benötigt man Metriken, die die Qualität der Software abschätzen. Wir haben im Umfang dieser Arbeit eine neue Methode entwickelt, die diese Aufgabe erleichtern soll.

Bevor wir unsere Methode vorstellen, soll zunächst etwas Hintergrundinformation zu Metriken im Allgemeinen sowie zu ihrem aktuellen Stand vermittelt werden. In dem ersten Unterkapitel definieren wir den Begriff "Metrik" und zeigen, für welche Aufgaben im Software-Entwicklungsprozess Metriken benötigt werden. Danach stellen wir drei Ansätze zur Berechnung von Metriken gegenüber: *Fehlerbasierte Metriken, Testfallbasierte Metriken* sowie *Testobjektbasierte Metriken.* Uns erscheint bei der Beurteilung deren Eignung für die Aufgabe, die Anzahl der in der Software verbleibenden Fehler abzuschätzen, der testobjektbasierte Ansatz am geeignetsten. Daher stellen wir im darauf folgenden Abschnitt drei gegenwärtig verwendete Metriken dar, die alle den testobjektbasierten Ansatz verfolgen, die jedoch jeweils von gänzlich verschiedenen Blickwinkeln auf das Problem eingehen.

Die von uns entwickelte Metrik, die wir in dem hierauf folgenden Abschnitt erläutern, verbindet die Methoden der drei gegenwärtigen Metriken.

Nach der Präsentation unserer Metrik folgt ihre Bewertung. Wir haben unser Modell auf zwei Aspekte geprüft: Zunächst haben wir innerhalb unseres Modells den Programmtext-Überdeckungsgrad verändert und die Auswirkungen auf die Schätzung der Fehler, die durch die Tests gefunden werden, beobachtet. Dieses Verhalten haben wir sodann mit unseren Ergebnissen aus den tatsächlichen Tests verglichen.

Weiterhin haben wir getestet, wie gut sich das Modell zur Testplanung eignet. Indem wir in dem Modell forderten, dass nach dem Testen höchstens zwei unbekannte Fehler in der Software verbleiben dürfen, berechneten wir die Auswirkungen auf die anderen Knoten und bekamen somit Abschätzungen über die hierfür benötigte Güte der Tester sowie den dafür benötigten Programmtext-Überdeckungsgrad.

Im letzten Abschnitt folgt ein Ausblick, in welcher Hinsicht das Modell noch verfeinert werden könnte.

7.1 Definition und Nutzen von Metriken

Der Testprozess oder Testzyklus gliedert sich in die Phasen Testplanung, Testspezifikation, Testdurchführung, Testprotokollierung und Testbewertung (vgl. Kapitel 4.3).

Die grobe Testplanung soll möglichst früh erstellt und im Testkonzept festgehalten werden. Sobald ein Testzyklus konkret ansteht, muss diese Grobplanung an die aktuelle Projektsituation angepasst werden. Zu berücksichtigen sind hierbei der tatsächliche Entwicklungsstand, die bereits erzielten Testergebnisse sowie die vorhandenen Ressourcen.

Um die Testplanung, also die Entscheidung, welches Modul/welche Funktion wie genau geprüft werden soll, nicht willkürlich treffen zu müssen, wurden verschiedene Metriken entwickelt, die einen Anhaltspunkt geben sollen, welche Module besonders fehleranfällig sind und daher besonderer Aufmerksamkeit bedürfen.

Definition 1. *Eine Software(qualitäts)metrik ist eine Funktion, die ein Programm auf einen Zahlenwert abbildet. Dieser berechnete Wert soll zeigen, wie gut eine Eigenschaft des Programmtexts erfüllt ist, oder helfen, die Softwarequalität des Programmtexts zu bestimmen (IEEE Standard 1061) [FN00].*

Eine Testmetrik misst eine Eigenschaft eines Testfalls, Testlaufs oder Testzyklus mit Angabe der zugehörigen Messvorschrift [SL04].

Damit ist eine Softwaremetrik also eine Maßzahl für eine Eigenschaft oder ein Qualitätsmerkmal von Software, während eine Testmetrik eine Eigenschaft des Tests abmisst. Verschiedene Metriken, die von der NASA angewendet werden, werden in [RHS98] vorgestellt, weitere Metriken finden sich in [FN04, NB05a, NB05b, MD98, MD97, HLL94, Gra92].

7.2 Verschiedene Klassen von Metriken

Welche Eigenschaft eine Metrik darstellen soll, hängt vom Nutzer ab. Für das Management sind Metriken interessant, die den Aufwand und die Kosten von Softwareentwicklungsprozessen veranschlagen. Entwickler und Tester hingegen sind an Metriken interessiert, die die Anzahl an Restfehlern bzw. die den erreichten Qualitätsgrad der Software abschätzen. Metriken, die die Güte der Testfälle abschätzen, sind nur für Tester interessant.

Für den Testmanager sind Metriken für drei Aufgaben relevant. Erstens können Metriken in der Planungsphase dazu verwendet werden, ein Bild zu erhalten, welche Komponente wie genau getestet werden muss. Zweitens muss die Überwachung und Erfolgskontrolle der laufenden Tests anhand objektiver, im Testkonzept spezifizierten, Testmetriken geschehen. Drittens kann das Testendekriterium anhand verschiedener Metriken bestimmt werden. Für die erste und dritte Aufgabe wird also eine Softwaremetrik benötigt, für die zweite eine Testmetrik.

Für ein erfolgreiches Qualitätsmanagement benötigen wir Softwaremetriken, die die erreichte Qualität der Software messen. Die ideale Software würde immer korrekt laufen. Um die Qualität der Software zu bestimmen, benötigen wir also eine Metrik, die die Anzahl an Restfehlern in der Software abschätzt. Im Folgenden werden daher Metriken besprochen, die dieses Ziel aufweisen.

Man kann zwischen drei Ansätzen unterscheiden [SL04]: *Fehlerbasierte Metriken, Testfallbasierte Metriken* sowie *Testobjektbasierte Metriken.*

Fehlerbasierte Metriken beruhen auf der Anzahl der im Test gefundenen Fehler. Diese Metriken sind problematisch, da die Aussage, wie viele Fehler während der Tests gefunden wurden, eigentlich keine Information darüber liefert, wie viele Fehler danach in der Software vorliegen [FN00]: Wurden wenig Fehler gefunden, so kann dies daran liegen, dass die Software tatsächlich einen hohen Qualitätsanspruch erfüllt - genausogut ist es aber auch möglich, dass nur nachlässig getestet wurde. Andererseits kann ein hoher Wert an gefundenen Fehler sowohl als Indiz für ein mangelhaftes Programm gewertet werden wie auch als Grund dafür verwendet werden, dass das Programm nun nahezu korrekt laufen müsste - die Fehler wurden ja bereits entdeckt und beseitigt. Da laut empirischen Studien die Anzahl der in der Testphase gefundenen Fehler keine Aussage über die Anzahl der im Testobjekt verbleibenden Fehler zulässt [FN00, FO00], sollte diese Art der Metrik vermieden werden.

Testfallbasierte Metriken beruhen auf der Anzahl spezifizierter oder geplanter Tests. Auch diese Metriken können nur begrenzt Schlüsse über die erreichte Qualität der Software zulassen. Mit zunehmender Testabdeckung werden aber vermutlich weniger Fehler übersehen. Allerdings müssen die

Testfälle auch unterschiedlich genug sein. Werden weitere hundert Testfälle spezifiziert, die alle einen bestimmten Programmpfad testen, während ein anderer Pfad ungetestet bleibt, so ist der Sicherheitszugewinn der durch die hundert Testfälle erreicht wird nur sehr gering [SL04, FN00]. Solche Überlegungen müssen also in die Metrik mit aufgenommen werden um einen Schluss auf die Softwarequalität des Testobjekts zuzulassen. Für eine Softwaremetrik kann dieser Ansatz also eventuell nicht ausreichend sein, für eine Testmetrik ist er aber gut geeignet.

Testobjektbasierte Metriken beruhen auf Informationen die mit Hilfe des Testobjekts selbst gewonnen werden können. Solche Informationen wäre beispielsweise die gewonnene Programmtext-Überdeckung, aber auch die zyklomatische Zahl als Maßzahl der Komplexität des Testobjekts, oder die Historie des Testobjekts (wie oft wurden Änderungen vorgenommen? Wie viele Programmierer haben an dem Programmstück gearbeitet? [NB05b]). Diese Metriken erscheinen uns am besten geeignet, Rückschlüsse auf die Qualität der Software zu ziehen, daher sollen diese Metriken im Folgenden genauer beleuchtet werden.

7.3 Testobjektbasierte Metriken

Eine sehr beliebte Softwaremetrik ist die zyklomatische Zahl. Sie misst die Komplexität von Funktionen, indem sie die Anzahl an Anweisungsequenzen mit der Verzweigungszahl in Beziehung setzt. In der Industrie wird diese Metrik gern verwendet, da sie leicht durch statische Verfahren zu berechnen ist. Beispielsweise verwendet Hewlett Packard diese Metrik mit der Maxime, dass jedes Modul dessen zyklomatische Zahl größer als 16 ist redesignt werden muss [Gra92]. Allerdings ist laut Fenton und Ohlsson der Wert dieser Metrik nicht bestätigt [FO00]. Sie haben in empirischen Studien keinen Hinweis erhalten, dass diese Metrik tatsächlich zu einer besseren Fehlerschätzung führt als andere Metriken.

Ein anderer sehr interessanter Ansatz wurde vom Microsoft Research Center entwickelt [NB05b]. Diese Metrik berücksichtigt besonders die Veränderungen, denen die Komponente innerhalb eines bestimmten Zeitrahmens ausgesetzt ist. Da sich Softwaresysteme weiterentwickeln und aufgrund nachträglicher Anforderungsänderung, Programmtextoptimierung oder Security fixes fortwährend verändern, erscheint es sinnvoll, diese Veränderungen in die Metrik mit einzubeziehen. Bei nachträglichen Umgestaltungen des Programmtexts kann schnell eine Randbedingung oder eine Abhängigkeit übersehen werden. Programmtext, der großen Veränderungen unterworfen ist, könnte daher fehleranfälliger sein, als Programmtextstücke, die sich nur wenig geändert haben. Es macht also Sinn, den eingeplanten Testaufwand an diese Veränderungen anzupassen. Nagappan und Ball erzielten mit dieser

Methode sehr gute Ergebnisse bei der Fehlerdichteschätzung für den Windows Server 2003 [NB05b].

Fenton und Neil schlagen einen dritten Ansatz vor, der die Möglichkeit bietet, verschiedene Teilaspekte gewichtet nach Wahrscheinlichkeit in die Metrik mit einzubeziehen. Mit Hilfe von "Bayesischen Netzwerken" können messbare Eigenschaften der Software mit empirischen Ergebnissen, Expertenwissen und kausalen Zusammenhängen zu einem Modell vereint werden, mit dessen Hilfe die Wahrscheinlichkeit des Auftretens eines sicherheitskritischen Fehlers berechnet werden kann [FN04]. Dieser Ansatz ist insbesondere daher äußerst interessant, weil das der Berechnung zugrunde liegende Modell sehr genau an die spezielle Software angepasst werden kann. Weiterhin stehen bereits zwei Tools zur Verfügung, mit deren Hilfe die Modelle komfortabel erstellt werden können: Hugin [A/S05] und AgenaRisk [Age05].

In Abbildung 7.1 wird gezeigt, wie ein solches Modell als Testendekriterium verwendet werden kann. Das Modell erhält Informationen aus dem Entwicklungsprozess als Eingangsgröße. Daraus berechnet es eine erwartete Gesamtfehlerzahl. Wurden nicht ausreichend Fehler gefunden, so müssen weitere Tests durchgeführt werden, bis das Restrisiko, das durch die in der Software verbleibenden Fehler bestimmt wird, akzeptabel erscheint.

Abbildung 7.2 ist ein Beispiel für eine solche Softwaremetrik, die mit dem Tool AgenaRisk erstellt wurde. Die Knoten des Netzwerks repräsentieren Teilaspekte des Modells, die Kanten stellen kausale Zusammenhänge zwischen den Teilaspekten dar. Weiterhin erhält jeder Knoten eine Wahrscheinlichkeitstabelle, in der für jeden Zustand, den der Knoten einnehmen kann, vermerkt wird, wie wahrscheinlich dieser Zustand ist. Hierbei ist zu beachten, dass die Wahrscheinlichkeitstabellen von Knoten mit Vorgängern auch die möglichen Zustände der Vorgängerknoten berücksichtigen.

7.4 Präsentation unserer Softwaremetrik

Da die Methodik von Fenton uns am mächtigsten erschien, wir den Ansatz der Microsoft-Researchgruppe aber nicht von der Hand weisen wollten, entschlossen wir uns, beide Ansätze für unsere Softwaremetrik zu verwenden. Wir modifizierten daher das von Fenton in [FN04] vorgestellte Modell zur Risikoanalyse der Sicherheit von Komponenten einer Software so, dass auch die Veränderlichkeit aus der von Microsoft vorgestellten Metrik berücksichtigt wurde. Zusätzlich erweiterten wir das Modell um weitere zwei Faktoren: neben der Veränderlichkeit nahmen wir auch den Test-Überdeckungsgrad als Indikator für die Güte der Tests sowie die zyklomatische Zahl als Indiz für die Komplexität des Programmtexts in das Modell auf. In Abbildung 7.2 ist das resultierende Modell abgebildet.

Der Übersichtlichkeit halber wurden die Knoten, die wir hinzugefügt haben, pink unterlegt. Die Knoten, die Zusammenhänge des Modells dar-

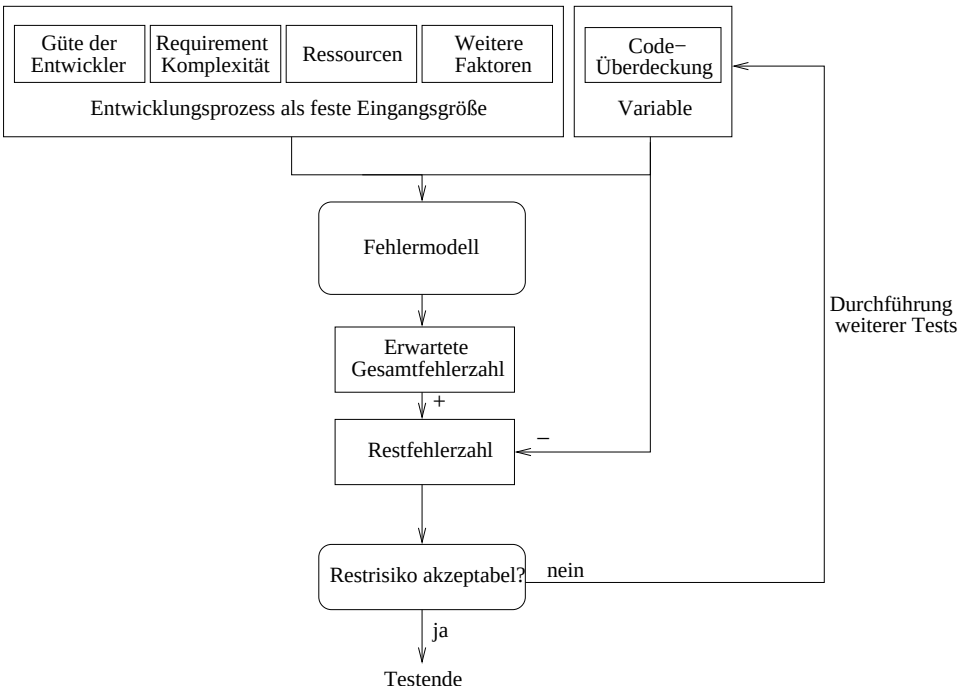

Abbildung 7.1: *In dieser Abbildung erhält das Fehlermodell als Eingangsgrößen Informationen aus dem Entwicklungsprozess. Daraus berechnet es die wahrscheinliche Gesamtfehleranzahl. Nach Durchführung der Tests wird die vom Modell geschätze Fehlerzahl mit der Anzahl der in den Tests gefundenen Fehler verglichen. Sinkt die Restfehlerzahl unter ein zuvor spezifiziertes Limit, so können die Tests beendet werden, andernfalls müssen weitere Tests durchgeführt werden.*

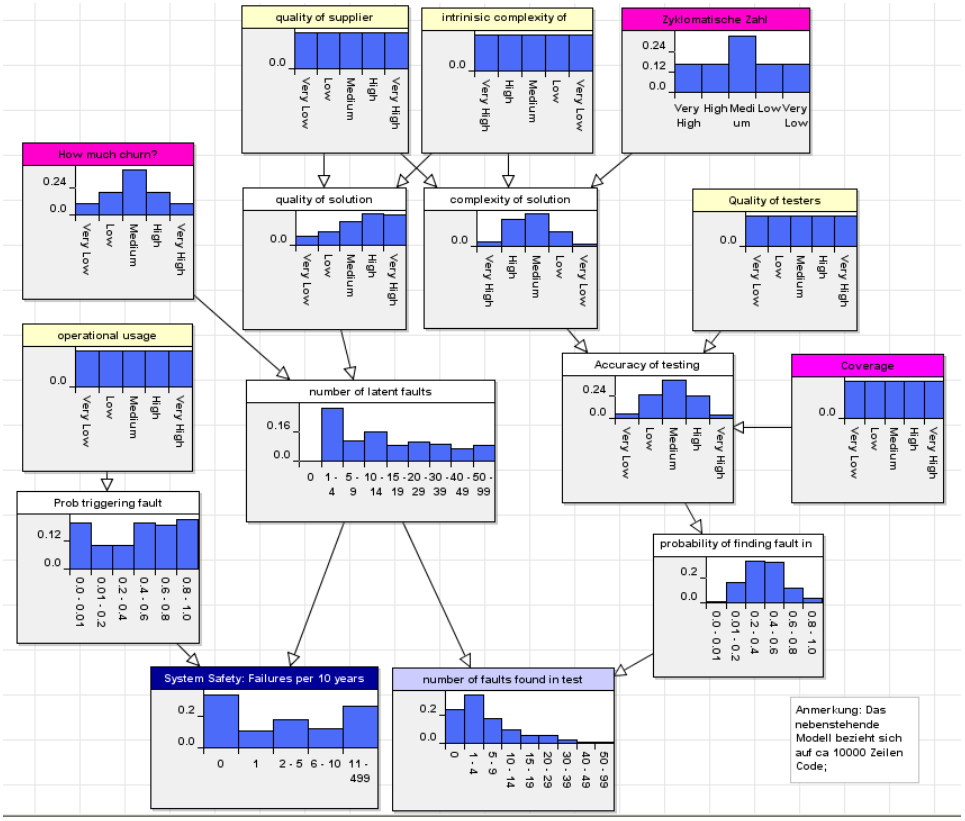

Abbildung 7.2: *Das von Fenton vorgestellte Modell wurde von uns um die drei rosa markierten Faktoren erweitert. Neben der zyklomatischen Zahl als berechenbares Indiz für die Komplexität des Programmtexts, nahmen wir noch den Aspekt der Veränderlichkeit des Programmtexts sowie den Testüberdeckungsgrad in unser Modell auf.*

stellen, aber für den Endnutzer nicht von Bedeutung sind, wurden weiß unterlegt. Die relativen Metriken von Nagappan und Ball wurden zunächst in einem einzelnen Knoten, der die Veränderung, der der Programmtext ausgesetzt wird, bemisst, zusammengefasst.

Aus dem Bild 7.2 ersichtlich, berücksichtigt das entstehende Modell folgende Faktoren:

- Eine Schätzung, wie oft die zu betrachtende Komponente im Einsatz ist
- Die Veränderung, der der Programmtext ausgesetzt ist
- Die zyklomatische Zahl als Indiz für die Komplexität des Programmtexts
- Die Komplexität der Requirements
- Die Qualität der Softwareentwickler
- Die Qualität der Tester
- Der durch die Tests erreichte Programmtext-Überdeckungsgrad
- Die Anzahl der Fehler, die durch die Tests gefunden wurde

Aus diesen Faktoren wird eine Schätzung der Sicherheit der Komponente sowie der Gesamtzahl der tatsächlich in der Komponente vorhandenen Fehler getroffen.

Der durch die Tests erreichte Programmtext-Überdeckungsgrad ist folgendermaßen zu verstehen: Eine sehr niedrige Überdeckung soll einer Programmtextüberdeckung von 0-50% entsprechen, eine niedrige 50-65%, eine mittlere 65-85%, eine hohe 85-95% und eine sehr hohe erreicht über 96% Überdeckung. Wir haben eine ungleichmäßige Verteilung gewählt, da der Aufwand für eine sehr hohe Überdeckung ebenfalls nicht linear steigt. Diese Festsetzung soll bei der Verwendung des Modells berücksichtigt werden.

7.5 Bewertung unserer Softwaremetrik

Die von uns entwickelte Metrik wollen wir für zwei Zwecke verwenden: Erstens wollen wir in der Testplanung mit ihrer Hilfe bestimmen, wie genau eine Komponente getestet werden soll. Zweitens soll sie am Ende des Testens als Testendekriterium fungieren: Der Testvorgang soll erst beendet werden dürfen, wenn laut Modell nur noch sehr wenig verbleibende Restfehler in der Komponente versteckt sind, und das Modul als sicher genug eingestuft werden kann. In diesem Abschnitt soll untersucht werden, wie gut sich die Metrik für diese Zwecke eignet.

7.5.1 Überprüfung des Modells

Um das unserer Metrik zugrunde liegende Modell zu überprüfen, wählten wir den Pfad, der dem Testendekriterium zugrunde liegt. Zur Überprüfung setzten wir ein Szenario, dass unsere Entwicklungs- und Testumgebung darstellt. Dieses Szenario berechneten wir durch das Modell, indem wir zunächst unsere Test-Ergebnisse sowie die festen Eingangsgrößen einsetzten - das Modell berechnete dann aus diesen Größen die resultierenden Größen für die restlichen Knoten. In dem resultierenden Szenario gingen wir dann rückwärts vor: Das zuvor berechnete Szenario wurde festgesetzt, die anderen Größen variabel. Danach berechneten wir in Abhängigkeit von der Qualität der Tests wie viele Fehler beim Testen mit welcher Wahrscheinlichkeit entdeckt werden. Gleichzeitig analysierten wir, wie viele Fehler wir tatsächlich bei den verschiedenen Programmtext-Überdeckungsgraden gefunden hatten, und verglichen diese Zahl sodann mit der vom Modell berechneten.

Zunächst wurden die Randbedingungen der Software-Entwicklung festgelegt. Um diese Randbedingungen zu ermitteln, setzten wir zunächst unsere Ergebnisse in das Modell ein, und prüften, was für Auswirkungen das auf die Randbedingungen der Softwareentwicklung hatte: Insgesamt haben wir 28 Fehler gefunden. Diese hohe Fehlerzahl erreichten wir nur bei sehr hoher Programmtext-Überdeckung, daher setzen wir die Programmtextcoverage auf “very high”. Weiterhin setzen wir Churn, also die Veränderung, der der Code ausgesetzt ist [NB05b], auf “high”, da der Programmtext oft verändert wird und viele verschiedene Mitarbeiter daran arbeiten. Auch Usage wurde auf “high” gesetzt, da wir davon ausgehen, dass ein Fahrer das ACC-System auch sehr viel nutzt.

Unter diesen Voraussetzungen berechnet das Modell, dass die Entwickler vermutlich schlecht (28%) bis mittelmäßig (30%) gearbeitet haben und dass die Komplexität der Requirements zwischen hoch (38%) und sehr hoch (36%) anzusiedeln ist. Weiterhin schätzt es, dass mit 10% Wahrscheinlichkeit alle Fehler gefunden wurden, mit 39% Wahrscheinlichkeit wurden bis zu zehn Fehler übersehen und mit 51%iger Wahrscheinlichkeit sind sogar noch mehr Fehler in der Software.

Setzen wir nun zusätzlich die Güte der Tester auf “high”, so sieht das Bild etwas optimistischer aus. Die Güte der Softwareentwickler verschiebt sich stärker in den “mittelmäßig”-Bereich (schlecht: 26%, mittelmäßig: 32%, hoch: 16%), die Komplexität der Requirements ist nun eher bei hoch anzusiedeln (sehr hoch: 31%, hoch: 39%, mittelmäßig: 29%). Besonders bemerkbar macht sich unsere Änderung aber in der Schätzung der Gesamtfehleranzahl: Die Wahrscheinlichkeit, alle Fehler entdeckt zu haben, bleibt bei 10%, doch die Wahrscheinlichkeit, dass wir nur bis zu 10 Fehler übersehen haben, steigt von 39% auf 47%. Die Wahrscheinlichkeit, dass noch mehr Fehler übersehen wurden, sinkt auf 42%.

Weiterhin schätzen wir die Komplexität der Requirements mit mittel ein.

Durch diese Änderung erhielten wir die Schätzung, dass wir mit 19% Wahrscheinlichkeit alle Fehler entdeckt haben und mit 78% Wahrscheinlichkeit nur bis zu 10 Fehler übersehen haben. Die Güte der Softwareentwickler verschiebt sich wieder in den schlechten Bereich (sehr schlecht: 32%, schlecht: 33%, mittelmäßig: 34%).

Somit legten wir die Randbedingungen für den Test des Modells wie folgt fest: Die Qualität der Entwickler wird auf "low" gesetzt, die Komplexität der Requirements auf "medium" und die Qualität der Tester auf "high". Churn bleibt "high", Art der Nutzung ebenso. im Folgenden betrachten wir nun, wie sich die verschiedenen Programmtext-Überdeckungsgrade laut Modell auf die Anzahl der gefundenen Fehler auswirkt, und vergleichen die Zahlen mit unseren tatsächlichen Zahlen.

Unter diesen Voraussetzungen berechnet das Modell, dass mit 98% Wahrscheinlichkeit 20-39 Fehler in der Komponente enthalten sind.

Setzen wir die Programmtext-Überdeckung auf "very low", so werden mit 72% Wahrscheinlichkeit 0-9 davon im Test gefunden (0: 11%, 1-4: 31%, 5-9: 30%). Lassen wir nun die Programmtextüberdeckung langsam steigen so steigt auch die Anzahl der gefundenen Fehler: bei einer noch immer niedrigen Programmtextüberdeckung werden laut Modell vermutlich 1-14 Fehler entdeckt (78%), bei einer mittleren voraussichtlich 5-19Fehler (80%); Wird die Programmtextüberdeckung auf "high" erhöht, so steigt die Zahl der entdeckten Fehler auf 10-29 mit Wahrscheinlichkeit 81%, bei einer weiteren Erhöhung der Überdeckung steigt die Wahrscheinlichkeit auf 86% (10-15:26%,15-19:30%,20-29:30%).

Vergleichen wir dies nun mit unseren realen Ergebnissen. Durch die Blackboxtests wurden 11 Fehler entdeckt.

Beim Whiteboxtest wurden 19 Fehler entdeckt, wobei hier auch Fehler mitzählen, die bereits durch Blackboxtests gefunden wurden, die aber auch durch die Whiteboxmethoden aufgedeckt wurden.

Durch die Blackboxmethode erreichten wir in unserem Beispielprojekt im Schnitt eine zwischen 15 und 20% geringere Überdeckungsrate verglichen mit dem Whiteboxtest. Bei den Blocküberdeckungen bewegte sich der Unterschied im oberen Level, bei den Bedingungsüberdeckungen eher im Unteren. Damit erreichten wir durch die Blackboxmethode nur eine mittlere Überdeckung[1]. Unter diesen Voraussetzungen bemißt das Modell die Wahrscheinlichkeit 5-9 Fehler zu finden auf 27%. Dies ist jedoch nur die zweithöchste Wahrscheinlichkeit - mit höchster Wahrscheinlichkeit (32%) sollten sogar 10-14 Fehler gefunden werden. Wir haben elf Fehler gefunden, befinden uns damit in dem Bereich, den das Modell als wahrscheinlichsten Wert berechnet hat. Somit passt unser Ergebnis gut zu der aus dem Modell

[1]Da der Aufwand für eine sehr hohe Überdeckung nicht linear steigt, ist auch der Überdeckungsgrad in unserem Modell nicht gleichmäßig verteilt. Wir setzen eine sehr niedrige Überdeckung auf 0-50%, niedrig: 50-65%, mittel: 65-85%,hoch:85-95%, sehr hoch: über 96%; vgl. Abschnitt 7.4

berechneten Schätzung.

Bei der Erstellung der Tests mit Hilfe der Whiteboxmethode haben wir bis zum Erreichen der 100% Programmtext-Überdeckung getestet - unerreichbaren Programmtext nicht mitgezählt. Somit erreichten wir eine sehr hohe Programmtext-Überdeckung. Für diesen Fall berechnet das Modell die gleiche Wahrscheinlichkeit von 30% sowohl für den Fall 15-19 sowie für den Fall 20-29 Fehler zu finden. Verglichen mit unserem Ergebnis von 19 Fehlern passt auch dieses Ergebnis gut.

Dies ist natürlich nur ein Fall, an dem das Modell überprüft werden konnte. Für eine genauere Überprüfung müsste eine breitere Datenbasis erstellt werden. Es müssten weitere Tests an anderen Modulen durchgeführt werden, um die Ergebnisse mit denen des Modells zu vergleichen. Dabei sollten sowohl Module, die unter anderen Entwicklungsbedingungen entstanden, wie auch solche mit ähnlichen Bedingungen geprüft werden. Im Rahmen dieser Diplomarbeit war dies nicht möglich, für unseren Fall zeigte das Modell jedoch sehr gute Prognosen.

7.5.2 Vergleich mit Fentons Metrik

Das von Fenton in [FN04] vorgestellte Modell bemißt die Güte der Tests nur durch die Qualität der Tester - der Programmtext-Überdeckungsgrad der Tests geht in dieses Modell nicht ein. Auch der Grad der Veränderung, der der Programmtext ausgesetzt ist, wird in diesem Modell nicht berücksichtigt.

Setzen wir nun in diesem Modell das Szenarium wie vorhin: Die Entwickler halten wir für schlecht ausgebildet, die Komponenten-Requirements für mittelmäßig komplex und die Tester für gut. Das Modell berechnet dann, dass sich mit 48% Wahrscheinlichkeit 10-14 Fehler in der Software finden und mit ebenfalls 48% Wahrscheinlichkeit 15-19 Fehler. Verglichen mit unserem Modell, das die Wahrscheinlichkeit für eine Gesamtfehlerzahl von 20 bis 39 Fehler mit 98% bemisst, fällt diese Vorhersage deutlich optimistischer aus. Der Faktor der Veränderung des Programmtexts, der unser Ergebnis pessimistischer ausfallen lässt, fehlt hier. Tatsächlich haben wir 28 Fehler gefunden, also zehn mehr als laut Fentons Modell im wahrscheinlichsten Fall enthalten sein dürften.

Vergleichen wir nun die Zahl der Fehler, die voraussichtlich in den Tests entdeckt werden sollen. In Fentons Modell beträgt die Wahrscheinlichkeit, dass ein bis vier Fehler gefunden werden 31%, für fünf bis neun Fehler sogar 50% und für 10-14 Fehler noch 15%. Diese niedrige Fehlerrate muss allerdings auch im Zusammenhang mit der ebenfalls deutlich niedrigeren Schätzung für die Gesamtfehlerzahl betrachtet werden. Da dieses Modell von 10 bis 19 Fehlern insgesamt ausgeht, würden ca. 50% der Fehler durch Testen entdeckt.

In unserem Modell gehen wir von insgesamt 20 bis 39 Fehlern in der Komponente aus (98%). Die Fehlerfindungsrate ändert sich in Abhängigkeit von

dem gewählten Überdeckungsgrad. Bei einer mittleren Überdeckung würden laut unserem Modell ebenfalls etwa 50% der Fehler gefunden, danach mit steigender Überdeckung immer mehr bis zu etwa 75% der Gesamtfehlerzahl.

In unseren Tests wurde offensichtlich, dass wir abhängig von der gewählten Überdeckungsart aber auch abhängig von dem Überdeckungsgrad unterschiedlich viele Fehler gefunden haben. So haben wir beispielsweise bei 80% Überdeckung durch Blackboxtests nur neun Fehler gefunden, steigerten wir die Überdeckung auf 100%, so fanden wir hingegen 23 Fehler. Der Grad der Überdeckung liefert bei der untersuchten Software also ein wichtiges Kriterium zur Einschätzung der Güte der Tests, und liefert somit eine gute Erweiterung gegenüber Fentons Metrik.

Weiterhin stellten wir bei den Tests fest, dass bei Funktionen, die sich oft änderten, deutlich mehr Fehler gefunden wurden, als bei solchen, die verhältnismäßig stabil waren. Allerdings waren die Funktionen, die sich oft änderten auch komplexer als die anderen. Ob die erhöhte Fehlerrate also an den vielen Veränderungen oder an der erhöhten Komplexität lag, ist aus unseren Daten nicht festzustellen. Trotzdem denken wir, dass die Erweiterung unseres Modells durch den Faktor der Veränderungen denen der Programmtext ausgesetzt wird sinnvoll ist. Dass unser Modell eine so hohe Gesamtfehlerzahl berechnet, liegt an diesem Faktor sowie dem der zyklomatischen Zahl. Da wir die zyklomatische Zahl jedoch nur mit “mittel” in das Modell eingehen lassen, ist der Unterschied zwischen den Berechnungen der Gesamtfehlerzahl in Fentons und unserem Modell vornehmlich auf den Veränderungsfaktor zurückzuführen. Der berechnete Unterschied beträgt bis zu 20 Fehler, wobei die Schätzung unseres Modells deutlich plausibler ist. Auch hier liefert unsere Erweiterung also eine Verbesserung der Vorhersage verglichen mit Fentons Metrik.

Unser Modell unterscheidet sich an drei Stellen von Fentons Modell: Unser Modell wurde um den Programmtext-Überdeckungsgrad, die zyklomatische Zahl wie auch um den Grad der Veränderung, der die Software ausgesetzt ist, erweitert. Durch die Vergleiche mit unseren Testergebnissen konnten wir zeigen, dass unser Modell sich besser an die Eigenschaften unserer Software anpasst und die Schätzungen unseres Modells für unsere Software gut geeignet waren. Aus diesem Grund denken wir, dass die Veränderung, die wir an Fentons Modell vorgenommen haben, für unsere Software sehr erfolgreich ist, und unser Modell daher in unserem Zusammenhang eine bessere Grundlage für die Testplanung sowie als Testendekriterium darstellt.

7.5.3 Verwendung der Metrik zur Testplanung

Unser Ziel war es, eine Metrik zu gewinnen, die wir sowohl für die Testplanung wie auch später als Testendekriterium verwenden können. Innerhalb der letzten zwei Abschnitte haben wir bereits gezeigt, dass unser Modell die Eigenschaften unserer Software gut in die Schätzungen miteinbezieht und

dass die Metrik sich gut als Testendekriterium eignet. Doch wie sieht es bei der Testplanung aus?

Für die Testplanung müssen zunächst die Randbedingungen festgesetzt werden, also die Güte der Softwareentwickler, die Komplexität der Requirements, die Schätzung, wie oft die Komponente im Einsatz sein wird, sowie das Maß der Veränderung, der der Programmtext unterlegen ist. Daraus ergibt sich eine geschätzte Zahl der Fehler, die in der Komponente verborgen sind. Wir haben nun geprüft, wie sich das Modell verhält, wenn wir die Zahl der im Test gefundenen Fehler variieren und welche Auswirkungen dies auf den Grad der Programmtextüberdeckung sowie auf die geforderte Qualität der Tester hat.

Wieder gehen wir davon aus, dass unsere Software-Entwickler gut ausgebildet sind und die Requirements mittelmäßig komplex sind. Die Komponente wird voraussichtlich viel genutzt. Dann berechnet unser Modell, dass wir mit fünf bis vierzehn Fehler in der Komponente rechnen müssen (mit Wahrscheinlichkeit ca 80%). Wir würden gerne alle Fehler finden - in der Realität ist diese Ziel kaum zu erreichen, schon da man nicht feststellen kann, wie viele Fehler tatsächlich noch vorhanden sind. Im Modell ist das aber kein Problem, hier können wir sowohl die Gesamtzahl der Fehler festsetzen wie auch die Zahl der Fehler, die im Test gefunden werden.

Setzen wir nun die Zahl der gefundenen Fehler in denselben Bereich wie die Gesamtzahl der Fehler. Wenig überraschend erhalten wir nun die Information, dass dann die Qualität der Tester sehr gut sein muss und auch ein sehr hoher Überdeckungsgrad notwendig ist. Allerdings unterscheidet sich die Notwendigkeit dieser Eigenschaft: Ist die zyklomatische Zahl sehr hoch, so ist die Komplexität des Programmtexts vermutlich hoch. Dies macht es schwieriger, gute Tests zu entwerfen. In diesem Fall steigt die Wahrscheinlichkeit, dass, um solche guten Ergebnisse zu erzielen, sehr gute Tester beschäftigt werden müssen und der Programmtext-Überdeckungsgrad sehr hoch sein muss, auf jeweils fast 60%. Ist die zyklomatische Zahl sehr niedrig so sinkt diese Zahl um ca 5%. Hier gibt das Modell also Anhaltspunkte über die Güte der Tester: sollen die Ziele erreicht werden, so müssen sehr gute Tester beschäftigt werden. Weiterhin ist ein sehr hoher Überdeckungsgrad notwendig. Daher kann der Testmanager darauf schließen, dass für dieses Modul sehr viel Test-Zeit eingeplant werden muß.

Weiterhin interessant ist folgende Beobachtung: falls wir die Gesamtzahl der Fehler als geringer einschätzen und somit auch weniger Fehler finden müssen, so sinkt ebenfalls die Notwendigkeit sehr gute Tester zu beschäftigen und einen sehr hohen Überdeckungsgrad zu erreichen (beträgt allerdings trotzdem noch ca 45%). Dies wirkt zunächst etwas eigenartig - wenn alle Fehler gefunden werden sollen, so benötigt man bestimmt sehr gute Tester und einen sehr hohen Überdeckungsgrad. Eine möglicher Grund für dieses Verhalten kann allerdings sein, dass eine niedrigere Gesamtfehlerzahl wiederrum auf eine niedrigere Komplexität des Programmtexts schließen lässt,

was dann auch in einfacheren Tests resultiert. Trotzdem sollte sich der Testplaner dessen bewusst sein - im Zweifelsfall sollte die Gesamtfehlerzahl daher eher pessimistisch eingeschätzt werden.

Zusammenfassend kommen wir zu dem Schluss, dass sich das Modell auch sehr gut für die Testplanung eignet. Es bietet dem Testplaner eine gute Hilfe, all die verschiedenen Zusammenhänge und Einflüsse im Auge zu behalten und somit eine gut begründete Entscheidung treffen zu können, welche Komponente wie genau getestet werden muss.

7.5.4 Weitere mögliche Verfeinerungen der Metrik

Wir haben in dieser Arbeit die Softwaremetrik von Fenton aus [FN04] um drei Aspekte erweitert: Veränderlichkeit des Programmtexts, die zyklomatische Zahl als Indiz für die Komplexität des Programmtexts sowie den Überdeckungsgrad als Indiz für die Güte der Tests. Wir haben gezeigt, dass unser Modell für unsere Software sehr gut geeignet ist - sowohl in der Testplanungsphase wie auch als Testendekriterium. Weiterhin haben wir gezeigt, dass die Schätzungen unserer Metrik für unsere Software auch deutlich akkurater ausfallen, als die des Modells von Fenton.

Interessant wäre die Frage, ob eine weitere Verfeinerung des Modells zu weiteren großen Verbesserungen führen würde. Denkbar wäre beispielsweise, die Veränderungen, denen der Programmtext ausgesetzt ist, genauer zu unterteilen - beispielsweise in der Genauigkeit, die in [NB05b] vorgeschlagen wird.

Eine andere interessante Verfeinerung wäre die Schätzung der Komplexität. Wir haben hierzu bereits die zyklomatische Zahl als Indiz für die Komplexität des Programmtexts eingeführt. Denkbar wäre es aber auch, das Modell noch um weitere Indikatoren zu erweitern. Hierfür würde sich beispielsweise die Halstead Metrik oder andere Lines-of-Programmtext-Metriken eignen.

Die dritte Erweiterung, die wir für sinnvoll erachten würden, wäre eine Einteilung der Komponente in verschiedene Kritikalitätsstufen. Beispielsweise wäre es für Unternehmen, die nach dem DO-178B Standard testen, praktisch, wenn das Modell auch die Kritikalitätsklasse der Komponente verwenden würde. Bei einem hohen Sicherheitsrisiko könnte das Modell dann eine höhere Testintensität fordern, als bei einem mittleren Sicherheitsrisiko.

7.6 Zusammenfassung der Ergebnisse

Eine Software(qualitäts)metrik ist eine Funktion, die ein Programm auf einen Zahlenwert abbildet. Dieser berechnete Wert soll zeigen, wie gut eine Eigenschaft des Programmtexts erfüllt ist, oder helfen, die Softwarequalität des Programmtexts zu bestimmen (IEEE Standard 1061) [FN00]. Eine Test-

metrik misst eine Eigenschaft eines Testfalls, Testlaufs oder Testzyklus mit Angabe der zugehörigen Messvorschrift [SL04].

Es gibt viele verschiedene Ansätze, um eine Metrik zu bestimmen. Eine der mächtigsten Varianten beruht auf Bayesischen Netzwerken [FN04], mit deren Hilfe verschiedene Teilaspekte wie Expertenwissen, empirische Ergebnisse, kausale Zusammenhänge und messbare Eigenschaften des Systems mit Wahrscheinlichkeit kombiniert werden können. Da die zugrunde liegenden Modelle aber sehr komplex sind, eignet sich diese Metrik vor allem für die Risikoanalyse in der Testplanungsphase sowie als Testendekriterium. Für die Testzyklus Überwachung sollten einfachere Metriken wie Programmtextüberdeckungskriterien verwendet werden.

Die allgemeinen Vor- und Nachteile der Methode der Bayesischen Netzwerke sind in Tabelle 7.1 zusammengefasst. In unserem Fall haben wir sehr positive Erfahrungen mit dieser Metrik gemacht.

Wir haben in dieser Arbeit Fentons Softwaremetrik aus [FN04] um drei Aspekte erweitert: Veränderlichkeit des Programmtexts, die zyklomatische Zahl als Indiz für die Komplexität des Programmtexts sowie den Überdeckungsgrad als Indiz für die Güte der Tests. Wir konnten zeigen, dass unser Modell für unsere Software sehr gut geeignet ist - sowohl in der Testplanungsphase wie auch als Testendekriterium. Weiterhin haben wir gezeigt, dass die Schätzungen unserer Metrik für unsere Software auch deutlich akkurater ausfallen, als die des Modells von Fenton.

Offen bleibt die Frage, ob eine weitere Verfeinerung des Modells zu ähnlich großen Verbesserungen führen würde. Denkbare Verfeinerungen wären, die Veränderungen, denen der Programmtext ausgesetzt ist, genauer zu unterteilen, die Komplexität des Programmtexts mit weiteren Indikatoren wie beispielsweise der Halstead Metrik einzuschätzen sowie eine Einteilung der Komponente in Kritikalitätsstufen zu ermöglichen. Weitere Untersuchungen in diesem Bereich könnten zu interessanten Ergebnissen führen.

Weiterhin stellt sich die Frage, ob das Modell auch auf höhere Ebenen übertragen werden kann. Bisher haben wir es nur am Beispiel des Modultests analysiert. Nun stellt sich die Frage, ob und wenn was für Änderungen vorgenommen werden müssten, um es an den gesamten Testprozess anzupassen.

Vorteile	Nachteile
• Viele verschiedene Informationen können kombiniert werden (Expertenwissen, empirische Daten, gemessene Daten, Wahrscheinlichkeit, ...) • Kausale Zusammenhänge können modelliert werden; • Unvollständige Informationen können durch das Modell aufgefangen werden; • Die Metrik kann leicht an die Software angepasst werden. • Das Modell kann auch "rückwärts" verwendet werden: Wie müssen die anderen Werte gesetzt werden, damit der Knoten "Component Safety" den Zustand "Very high" erreicht?	• Ist das Modell unzureichend, so können auch die Ergebnisse der Testmetrik zu falschen Schlüssen verleiten. • Bei der Erstellung des Modells wird viel subjektives Wissen verwendet.

Tabelle 7.1: *Bewertung einer Metrik beruhend auf Bayesischen Netzwerken*

Kapitel 8

Testmanagement Empfehlungen

Das Tool RTRT bietet sehr gute Möglichkeiten, Fehler schnell zu finden. Durch die Coveragereports kann genau analysiert werden, welcher Programmtext durch einen fehlerhaften Testfall durchlaufen wurde, und somit kann der Suchraum für den Fehler stark eingeschränkt werden. Auf diese Weise wird es dem Tester sehr erleichtert, den Fehler zu finden. Problematisch ist dann jedoch die Entscheidung, ob ein Fehler ein tatsächlich auftretender Fehler ist, oder ob er nur in einem Zustand auftreten kann, der sowieso unerreichbar ist.

Wie man in Abbildung 8.1 sehen kann, sind von einem definierten Startzustand nur endlich viele Zustände erreichbar. Die restlichen Zustände sind für das System eigentlich nicht relevant, da das System diese Zustände nie einnehmen wird. Diese Fehler könnten nur potentielle Probleme darstellen, da eine Änderung des Systems auch eine Änderung der Menge der erreichbaren Zustände nach sich ziehen könnte. Soll der Tester nur den Ist-Zustand überprüfen, so müßte er nur Testfälle für die erreichbaren Zustände schreiben - welche Zustände jedoch erreichbar sind und welche nicht, dies zu entscheiden benötigt ein großes Hintergrundwissen über die Projektsoftware.

Somit kann die Analyse, ob der gefundene Fehler auftreten kann, nur von jemandem ausgeführt werden, der ein großes Verständnis für die Abläufe der Software besitzt, denn er muß in der Lage sein, zutreffende Annahmen machen zu können, ob der Fehler in einem Zustand auftreten kann, der durch den Startzustand erreichbar ist.

Dies würde dafür sprechen, dass der Tester aus dem Entwicklungsteam kommt, da so sicher gestellt werden kann, dass er ausreichend Hintergrundwissen zu dem System hat. Am meisten Hintergrundwissen zu einem Modul hat der Entwickler des Moduls selbst. Jedoch sprechen viele Gründe dagegen, den Entwickler sein Modul testen zu lassen: wie in Abschnitt 3.5 erläutert gehen Entwickler meist mit der falschen Einstellung an den Test-

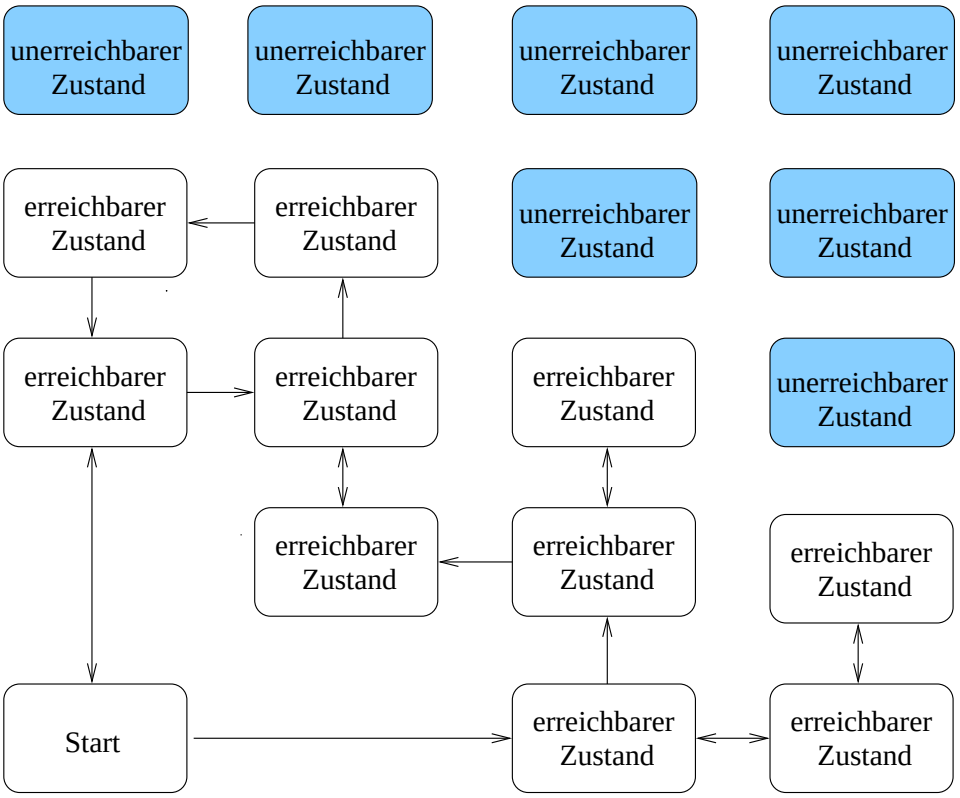

Abbildung 8.1: *In dieser Abbildung wird ein Beispiel für ein stark vereinfachtes Zustandübergangsdiagramm eines Systems dargestellt. Vom Startzustand sind einige Zustände erreichbar, die unerreichbaren Zustände sind schattiert dargestellt. Soll der Ist-Zustand eines Systems überprüft werden, so muß nur das Verhalten der erreichbaren Zustände überprüft werden. Die Frage, welche Zustände eines komplexen Systems jedoch unerreichbar sind, erfordert viel Hintergrundwissen über das System.*

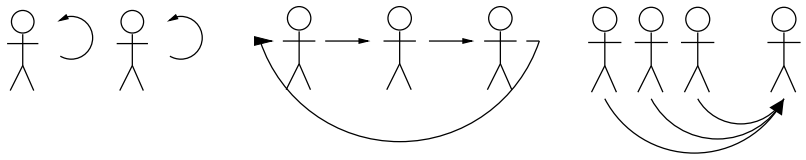

Abbildung 8.2: *Es sind verschiedene Modelle zur Aufgabenteilung zwischen Entwicklung und Test möglich. Im ersten Fall liegt das Testen ausschließlich in der Verantwortung des einzelnen Entwicklers. In der mittleren Graphik liegt das Testen zwar noch in der Verantwortung des Entwicklerteams, jedoch testen die Entwickler nicht mehr ihre eigenen Module, sondern die ihrer Kollegen. Die dritte Graphik zeigt eine Möglichkeit, in der die Entwickler weniger in den Testprozess integriert werden. Entweder wird hier ein Entwickler des Teams dazu abgestellt, alle Testarbeiten des Teams durchzuführen, oder es wird ein eigenes Testteam ins Leben gerufen, dass nur noch für das Testen zuständig ist und keine Entwicklung mehr vornimmt. Dieses Testteam kann zum Projekt gehören, oder es kann eine separate Organisation beauftragt werden.*

prozess [Mye82]. Sie wollen zeigen, dass ihr Modul läuft, nicht Fehler finden. Weiterhin ist es in dieser Konstellation höchst unwahrscheinlich, dass Fehler, die ein Mißverständnis der Anforderungen als Ursache haben, erkannt werden - der Entwickler ist ja überzeugt, die Anforderungen richtig verstanden zu haben.

Eine andere Möglichkeit ist, dass das Testen zwar in der Verantwortlichkeit des Entwicklerteams liegt, aber die Entwickler ihre Programmteile gegenseitig testen. Diese Möglichkeit bekämpft einen Teil des Problems der Blindheit gegenüber eigenen Fehlhandlungen. Liegt der Fehler in einem Mißverständnis des Entwicklers begründet, so kann ein Kollege diesen Fehler aufspüren. Liegt er jedoch bereits in einem höheren Design, das gemeinsam ausgearbeitet wurde, so kann die Blindheit wieder auftreten.

Sollen die Entwickler weniger in den Testprozess integriert werden, so gibt es drei Möglichkeiten [SL04]: Entweder einer der Entwickler übernimmt alle Testarbeiten des Teams. Dann muß er allerdings auch seine eigenen Programmteile testen. Oder es wird ein zusätzliches Testteam ins Leben gerufen, welches nicht selbst entwickelt sondern nur testet. Dieses Testteam kann entweder zum Projekt gehören, oder aber eine separate Organisation wird beauftragt. Wird eine externe Organisation beauftragt, so wäre es allerdings wünschenswert, wenn die Tester zumindest kurz an dem Projekt als Entwickler arbeiten, um sich so das notwendige Hintergrundwissen anzueignen.

Die verschiedenen Möglichkeiten sind in Abbildung 8.2 grafisch dargestellt.

Kapitel 9

Zusammenfassung und Ausblick

> Alles Wissen und alles
> Vermehren unseres Wissens
> endet nicht mit einem
> Schlußpunkt, sondern mit einem
> Fragezeichen.
>
> Hermann Hesse

Diese Arbeit hatte zum Ziel, eine Modultestumgebung für den Modultest der Softwarekomponenten eines ACC-Systems aufzubauen und eine sinnvolle Teststrategie zu entwickeln.

Um die Ergebnisse der Arbeit vorzustellen, mußten wir zunächst auf drei Ebenen Grundlagen definieren: Es mußten Informationen zu dem Projekt selbst gegeben werden, auf der diese Arbeit beruht. Weiterhin wurden Grundlagen des Softwaretests benötigt - von den Definitionen wichtiger Begriffe hin zu fundamentalen Testprinzipien und Testmethoden. Als dritter Aspekt mußte das verwendete Testprogramm RTRT vorgestellt werden, und hinsichtlich der Vor- und Nachteile bezüglich der gestellten Aufgabe bewertet werden.

Um diese Grundlagen zu schaffen, haben wir zunächst in Kapitel 2 das ACC-System vorgestellt. Zunächst allgemein, in welcher Ausbaustufe es sich befindet und welche zusätzliche Funktionalität für zukünftige Serienstände geplant sind, danach die Displayfunktionalität im Besonderen, da unserer Studie der Programmtext der Displayansteuerung zugrunde liegt und in den Beispielen diese Funktionalität verwendet wird.

Danach folgte in Kapitel 3 eine kurze Zusammenfassung der Grundlagen des Softwaretestens. Es wurden verschiedene Begriffe wie “Validierung”, “Verifikation”, “Fehler” und “Fehlerwirkung” definiert, und Myers grundlegende Testprinzipien wurden vorgestellt. Aufbauend darauf wurde in Kapitel 4 der Testprozess in den Softwarelebenszyklus eingegliedert. Hierfür

wurde zunächst das allgemeine V-Modell vorgestellt, gefolgt von einer Erläuterung der grundlegenden Teststufen sowie des fundamentalen Testprozesses. In diesem Kapitel wurden auch die wichtigsten Testmethoden vorgestellt, sowie das Vorgehen nach den Methoden an ausgewählten Beispielen verdeutlicht.

Der dritte Aspekt, die Vorstellung des Tools RTRT, erfolgte in Kapitel 5. Ein besonderes Augenmerk lag hierbei darauf, zu zeigen, welche Hilfen das Tool leistet, für welche Aufgaben es sich eignet und für welche Aufgaben es weniger geeignet ist. Im Anhang A wurde weiterhin exemplarisch gezeigt, wie das Testvorgehen nach den verschiedenen Methoden mit dem Tool umgesetzt werden kann. Wir kamen anhand unserer im Laufe der Studie gewonnenen Erfahrungen zu dem Schluß, dass sich das Tool bei unserer Software für den Modultest sehr gut eignet, für den Komponententest jedoch nur begrenzt einsetzbar ist.

Nachdem die Grundlagen bekannt waren, konnte die Studie selbst in Kapitel 6 vorgestellt werden. Zur Entwicklung einer sinnvollen Testmethodik haben wir Programmtext von der Displayansteuerung der ACC-Software von Bosch mit verschiedenen Testmethoden getestet, um zu sehen, welche Methode welche Fehlerarten findet und wieviele Fehler mit ihr entdeckt werden.

Die Studie zeigte, dass durch die Blackboxmethode und die Whiteboxverfahren sehr unterschiedliche Fehlerarten aufgedeckt wurden, die jedoch in beiden Fällen hohe Relevanz aufwiesen. Aus diesem Grund kamen wir zu dem Ergebnis dass hier eine Kombination beider Methoden verwendet werden muß. Die Blackboxmethode fand vornehmlich Fehlerarten wie fehlende Fehlerbehandlungen oder unbehandelte Spezialfälle, sowie unerfüllte und unvollständige Requirements. Da der Tester mit dieser Methode auch nur die Requiremenents zur Verfügung hat, werden diese genauer analysiert und somit werden unerfüllte Requirements mit dieser Methode eher gefunden - beim Whiteboxtest wurden in unserem Fall drei solche Fehler übersehen, die durch die Blackboxmethode aufgedeckt worden waren.

Die Whiteboxverfahren eigneten sich vornehmlich dafür, unerwünschte Zusatzfunktionen aufzuspüren, sowie unerreichbaren und unnötigen Programmtext zu entdecken. Auch potentielle Fehler, wie beispielsweise der, bei dem falsche Eingangsgrößen verwendet worden waren, die jedoch auf dieselbe Speicherstelle verwiesen wie die richtige, konnten nur durch diese Methode entdeckt werden. Die zusätzlichen durch den Whiteboxtest entdeckten Fehler befanden sich mit einer Ausnahme gänzlich in dem durch die Blackboxmethode unabgedeckten Programmtext - nur der potentielle Fehler befand sich in abgedecktem Programmtext. Somit scheint das Risiko, durch einen reduzierten Whiteboxtest, der nur die unabgedeckten Programmtextteile prüft, relevante Fehler zu übersehen, gering zu sein.

Beim Whiteboxverfahren blieb weiterhin die Frage, welche Überdeckung gewählt werden sollte. In unserer Studie stellten wir fest, dass viele Fehler

bereits durch die Anweisungsüberdeckung aufgedeckt wurden. Ein Viertel der Fehler wurde jedoch erst durch die Bedingungsüberdeckungen gefunden. Dies läßt darauf schließen, dass komplexe Bedingungen eher fehleranfällig sind.

Weiterhin zeigten unsere Ergebnisse, dass eine vollständige Überdeckung nicht immer erreicht werden kann - bei unerreichbarem Programmtext kann dieser per Definition nicht abgedeckt werden, bei unnötigem oder unerwünschtem zusätzlichem Programmtext macht es wenig Sinn diesen mit abzudecken, da dieser Programmtext sowieso entfernt werden muß. Jedoch sollte bei sicherheitskritischen Modulen eine vollständige Überdeckung des Programmtexts, der beibehalten werden soll, angestrebt werden - sowohl bezüglich der Anweisungs- wie auch bezüglich der minimalen Mehrfachbedingungsüberdeckung.

Aus den Ergebnissen wurde eine sinnvolle Teststrategie abgeleitet. Da die durch die Methoden entdeckten Fehlerarten sich stark unterschieden, kamen wir zu dem Schluß, dass eine Kombination beider Methoden verwendet werden muß. Da fast alle zusätzlichen durch den Whiteboxtest entdeckten Fehler in den vom Blackboxtest unabgedeckten Programmteilen lagen, kamen wir zu dem Schluß, dass der Whiteboxtest nach dem Blackboxtest durchgeführt werden sollte und nur gezielt die noch unabgedeckten Programmtextteile prüfen soll. Nur bei besonders kritischen Modulen kann ein vollständiger weiterer Whiteboxtest angedacht werden. Je nach Sicherheitskritikalität des Moduls soll eine entsprechende Überdeckung für den Whiteboxtest gewählt werden: Ist es sicherheitskritisch und enthält das Modul komplexe Bedingungen, so sollte eine vollständige minimale Mehrfachüberdeckung angestrebt werden. Je weniger kritisch das Modul umso niedriger kann die geforderte Überdeckung ausfallen, allerdings sollte zumindest ein Blackboxtest durchgeführt werden. Falls dieser eine sehr niedrige Anweisungsüberdeckung liefert, sollten die unüberdeckten Programmtextteile zumindest eines Programmtextreviews unterworfen werden.

Die Überprüfung unserer Teststrategie anhand dreier weiterer Funktionen unterstützte unsere These, dass diese Strategie für die Beispielsoftware gut geeignet ist. Durch zusätzliche Whiteboxtests wurden keine weiteren Fehler entdeckt, unser Verfahren scheint daher ausreichend genau zu sein. Bei der Entwicklung der Teststrategie unterschieden wir bereits zwischen sicherheitskritischen und weniger kritischen Modulen. Wie jedoch kann bestimmt werden, welches Modul sicherheitskritisch bzw. besonders fehleranfällig ist? Mit dieser Frage befassten wir uns im zweiten Teil der Studie in Kapitel 7.

Eine mögliche Lösung ist die Verwendung einer Metrik, die die Kritikalität eines Moduls abschätzt. Eine solche Metrik nimmt bestimmte Eingangsgrößen und berechnet daraus ein Ergebnis. Eine einfache Metrik ist beispielsweise die zyklomatische Zahl, die aus dem Verhältnis der Knoten und Kanten des Kontrollflußgraphen eines Programms dessen Komplexität

zu berechnen versucht. Diese Metrik berechnet also nur einen kleinen Aspekt der Software. Soll jedoch die Kritikalität eines Moduls abgeschätzt werden, so müssen viele verschiedene Aspekte berücksichtigt werden, einige davon sind vielleicht auch nur Erfahrungswerte. In diesem Fall eignen sich komplexere Modelle, die den Entwicklungsprozess nachbilden und auch Erfahrungswerte einfließen lassen können. Ein solches Modell wurde von Fenton in [FN04] vorgestellt. Wir haben dieses Modell weiterentwickelt und an unsere Software angepasst. Im Vordergrund stand hier, eine Metrik zu entwickeln, die, angepasst auf die Beispielsoftware, gute Vorhersagen liefern kann, wie viele Fehler sich in einem Modul befinden. Anhand dieser Vorhersagen kann dann geplant werden, welche Module besonders intensiv getestet werden müssen, und auch das Testende kann in Abstimmung mit diesen Ergebnissen erfolgen. Wir überprüften die Vorhersagen unserer Metrik mit den tatsächlichen Ergebnissen der Studie aus Kapitel 6. Die Vorhersagen passten sehr gut. Weiterhin trafen sie deutlich besser auf unsere Ergebnisse zu, als die Vergleichsvorhersagen von Fentons ursprünglicher Metrik. Wir kamen daher zu dem Schluß, dass unsere Metrik schon sehr gut für unsere Software geeignet ist, und dieser Weg weiter verfolgt werden sollte.

Somit haben wir unser Ziel erreicht: wir haben eine Modultestumgebung mit dem Tool RTRT aufgebaut und eine sinnvolle an die Beispielsoftware angepasste Testmethodik entwickelt. Diese Methodik bestimmt in Abhängigkeit von der Sicherheitskritikalität eines Moduls, wie genau es getestet werden soll und welche Methoden angewendet werden sollen. Die Einteilung der Module bezüglich ihrer Kritikalität erfolgt mit Hilfe der entwickelten Metrik.

Nach dem momentanen Stand ist dieses Vorgehen das sinnvollste. Wir denken, dass mit diesem Vorgehen viele Fehler entdeckt und somit schon früh behoben werden können. Auf diese Weise kann die Zuverlässigkeit der Produkte erhöht werden.

Anhang A

Exemplarisches Testvorgehen mit RTRT

In diesem Abschnitt soll exemplarisch gezeigt werden, wie das Testen mit Hilfe des Tools RTRT erfolgt. Hierzu wird zunächst behandelt, wie man ein Projekt aufsetzt; Danach wird die Skriptsprache für die Testfallgenerierung vorgestellt und deren Anwendung exemplarisch an einem Beispiel verdeutlicht. Anschließend wird das Vorgehen zur Analyse der Ergebnisse dargestellt, wieder anhand des Beispiels.

A.1 Erstellung eines neuen Projekts

Vor der Erstellung eines neuen Projekts mssen folgende Fragen geklärt werden:

- Welches Modul bzw. welche Komponente soll getestet werden?
- Welche Module sollen integriert werden?
- Welche Funktionen sollen mit Platzhaltern versehen und welche sollen in das Testobjekt integriert werden?

Was ist der Unterschied zwischen Modulen, die getestet werden sollen, und solchen, die nur integriert werden? Ein Unterschied ist, dass die Funktionen von testbaren Modulen über die Testumgebung direkt aufgerufen werden können, während integrierte Module lediglich über die Funktionen der testbaren Module aufgerufen werden können.

Weiterhin kann die Testprotokollierung(beispielsweise die Programmtextüberdeckungsrate der integrierten Funktionen)ausgeschaltet werden, und nur bei Bedarf abgerufen werden. Es ist so möglich, sich ausschließlich die Programmtextüberdeckung der zu testenden Funktionen ausgeben zu lassen, die der integrierten Funktionen wird dann gar nicht erst berechnet.

Der Vorteil von integrierten Funktionen ist, dass Funktionen, die man für fehlerfrei hält, in das Modul eingebettet werden können, und man sich so die Erstellung der Platzhalter sparen kann. Beispielsweise macht es wenig Sinn, Funktionen aus Bibliotheken zu stubben. Bei solchen Funktionen wird angenommen, dass sie funktionieren - sie selbst im Stub neu zu definieren ist fehleranfälliger, als sie einfach zu übernehmen. Bibliotheken sollten daher generell in das zu testende Modul integriert werden.

Bei der Erstellung eines neuen Projekts baut RTRT das Testbett auf. Hierfür muss das Tool die Testtreiber generieren, sowie die externen Funktionen auffinden und entsprechende Platzhalter erstellen. Damit das Tool herausfinden kann, was für Parameter die externen Funktionen erwarten und welche Rückgabewerte sie liefern, müssen die externen Funktionen in header-files deklariert werden. Daher muss in jedem File, das getestet werden oder das in das Testobjekt integriert werden soll, ein Header inkludiert werden, der diese Deklarationen liefert oder den Ort der Deklarationen definiert. Weiterhin muss ein Compiler installiert sein.

Danach kann das Programm gestartet werden. Auf der Startseite muss “New Project” gewählt werden. Nach der Wahl eines passenden Projektnamens und des Speicherorts muss noch der Compiler der Entwicklungsumgebung ausgewählt werden. Nach der Definition des Rahmens des Projekts, folgt nun die Generierung des Testbettes. Unter “Activities” auf der Startseite wird “Component Testing” selektiert. Danach werden die Testfiles, die getestet werden sollen, ausgewählt.

Anschließend müssen in den “Configuration Settings” unter “Compiler” die relevanten Bibliotheken eingebunden werden. Im nächsten Schritt können unter “Generation Settings” zusätzlich weitere Files definiert werden, deren Funktionen in das zu testende Modul integriert werden sollen. Danach kann das Testbett erzeugt werden.

War die Erzeugung erfolgreich, so wurde durch das Tool bereits das erste Testfile generiert. Dieses besteht zunächst nur aus der Deklaration der benötigten Platzhalter für die externen Funktionen und Datenstrukturen, sowie simpler Testfälle, die nur aus einem Aufruf der Funktionen des Testmoduls bestehen. Diese Testfälle sind allerdings noch zu simpel, um die Funktion des Moduls zu testen. Im folgenden Abschnitt soll daher gezeigt werden, wie diese Testfälle geschrieben werden können.

In Abbildung A.1 wird ein Ausschnitt aus einem Projektfenster dargestellt: In diesem Projekt werden die Module “dal” und “dag” getestet. Beide Knoten besitzen gelb markierten Testknoten als Kinder, sowie integrierte c-files. Die c-files werden blau markiert, falls sie instrumentiert werden, andernfalls sind sie weiß markiert.

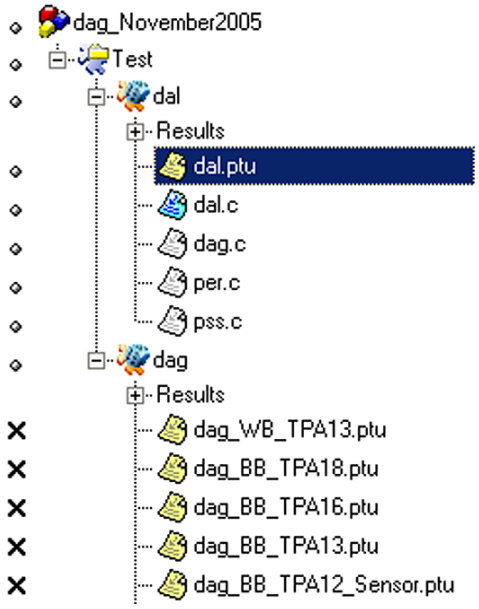

Abbildung A.1: *In diesem Ausschnitt aus dem Projektfenster wird ein Projekt mit dem Namen "dagNovember2005" gezeigt, in dem die Module "dal" und "dag" getestet werden.*

A.2 Generierung eines Testskripts

In diesem Abschnitt wird beschrieben, wie Testfälle für Tests in RTRT geschrieben werden können. Dazu wird zunächst die Skriptsprache beschrieben. Hierbei beschränken wir uns auf die Aspekte der Sprache, die zum Testen der Software des Beispielprojekts relevant sind. Weitere Möglichkeiten der Skript-Sprache werden in [Cen05] genauer erläutert.

Anschließend wird am Beispiel der Fahrerübernahmeaufforderung(vgl. Kapitel 2.2.1) gezeigt, auf welche Weise die erdachten Testfälle in RTRT umgesetzt werden können.

Kurzvorstellung der Skriptsprache Bei der ersten Erstellung eines Projekts generiert RTRT ein Testfile, in dem jede Funktion genau einmal aufgerufen wird. Im Folgenden wird gezeigt, wie dieses automatisch generierte Testfile so modifiziert werden kann, dass eigene Tests ausgeführt werden.

Die Instruktionen in einem Testfile haben folgende Eigenschaften:

- Alle Anweisungen beginnen mit einem Keyword
- Anweisungen beginnen am Anfang der Zeile und enden am Ende der Zeile. Soll eine Anweisung über mehrere Zeilen hinweg reichen, so müssen die Zeilen durch ein "&" verknüpft werden.

- Anweisungen dürfen höchstens 2000 Zeichen lang sein.

Testskripts werden in drei Levels strukturiert: Die erste Stufe ergibt sich durch den SERVICE-Namen, die zweite Stufe durch den TEST-Namen, die dritte durch die FAMILY-Klasse. Im Folgenden zunächst ein Beispiel für ein typisches Testfile:

```
--Beispiel fuer ein Testskriptfile
HEADER add,1,1
  BEGIN
    SERVICE add
         TEST 1
            FAMILY nominal
            ELEMENT
              VAR variable1, init =0, ev = init
              VAR variable2, init =1, ev = init
              VAR ergebnis,  init==,  ev = 1
              #add(variable1, variable2)
            END ELEMENT
         END TEST
    END SERVICE
```

Die in Großbuchstaben geschriebenen Keywords strukturieren den Test. `HEADER` spezifiziert den Namen und die Versionsnummer des Moduls, das getestet wird, sowie die Versionsnummer des Testobjekts. Im Testreport wird diese Information angezeigt. `BEGIN` zeigt an, wo die Generierung des tatsächlichen Testprogramms beginnt.

Unter `SERVICE` werden verschiedene Testfälle zusammengefasst. Beispielsweise bietet es sich an, einen Service pro Funktion anzubieten. Beim Komponententest werden allerdings kleinere Testgruppen benötigt, da die Testprotokolle sonst zu lang und unübersichtlich werden. Hier kann beispielsweise ein Service für jedes zu prüfende Signal eingeführt werden. Jeder `SERVICE` benötigt einen eigenen Namen. Ein `SERVICE` Block wird durch das Keyword `END SERVICE` beendet.

Jeder `TEST` hat innerhalb eines `SERVICE` Blocks einen eindeutigen Bezeichnernamen. Weiterhin gehört er einer `FAMILY`-Klasse an, und enthält einen `ELEMENT`-Block. Analog zu dem Serviceblock endet der Testblock durch das Keyword `END TEST`. In einem Test wird das Verhalten der Funktion unter verschiedenen Bedingungen getestet. Im Blackboxtest bietet es sich an, einen Test pro Äquivalenzklasse zu schreiben, im Whiteboxtest einen Test pro Zweig.

Das Keyword `FAMILY` kann dazu verwendet werden, verschiedene Tests zu einer Familie zusammenzufassen. Beispielsweise ist es möglich, alle Tests derselben `FAMILY`-Klasse auszuführen. Da allerdings im Komponententest schon einzelne Tests sehr große Testprotokolle generieren, ist es in diesem

Kontext nicht sinnvoll, mehrere Tests gleichzeitig auszuführen. Hier kann die Strukturierung in `FAMILY`-Klassen nur als zusätzliches Strukturmittel für den Tester hilfreich sein.

Der `ELEMENT`-Block beschreibt die Testphase und wird durch das Keyword `END ELEMENT` beendet. Im `ELEMENT`-Block werden die zu testenden Variablen deklariert, deren Initialzustand und erwarteter Endzustand festgelegt, und die zu testenden Funktionen aufgerufen. In diesem Beispiel wird getestet, ob die Variablen `variable1` und `variable2` unter der Belegung $0, 1$ richtig addiert und das Ergebnis in `ergebnis` geschrieben wird.

Die Element-Blöcke beinhalten damit den wichtigsten Teil der Tests: Hier werden die eigentlichen Tests spezifiziert. Dazu müssen zunächst die Variablen, die getestet werden sollen, spezifiziert werden. Hierfür wird die `VAR`-Anweisung benötigt. Diese erwartet drei Parameter: den Namen der Variable, die getestet werden soll, den Initialzustand der Variable(durch `init` gekennzeichnet), sowie ihren erwarteten Wert nachdem die Prozedur ausgeführt wurde(durch `ev` gekennzeichnet). Folgende Deklarationen sind zulässig:

```
VAR variable1, init = 1,         ev ==
VAR variable2, init IN {0,1},    ev IN {2,3}
VAR ergebnis,  init (variable2) WITH {0,1},
               & ev IN {1,1}, DELTA=5%
```

Bei der ersten Deklararation wird `variable1` der Initialwert “1” zugewiesen, das `ev==` bedeutet, dass der erwartete Wert nach der Prozedur nicht getestet werden soll. Auch `init==` wäre zulässig. In der zweiten Zeile erhält `variable2` zunächst den Initialwert “0” mit erwartetem Endwert “2”, und im nächsten Testdurchlauf den Initialwert “1” mit erwartetem Endwert “3”. Falls mehrere Variablen mit `init IN` initialisiert werden, so werden alle kombinatorischen Möglichkeiten getestet. Wird beispielsweise eine Variable mit drei verschiedenen Werten initialisiert, und eine weitere mit zwei Werten, so werden auf diese Weise sechs Testfälle unterschieden. So können sehr elegant sehr schnell viele Testfälle geschrieben werden. Manchmal sollen aber Variablen aneinander gebunden werden. Beispielsweise ist `ergebnis` in unserem Beispiel von `variable1` und `variable2` abhängig. `variable1` wurde festgesetzt, aber `variable2` erhält je nach Testlauf verschiedene Werte. Die `ergebnis`-Variable muss also an `variable2` gebunden werden, so dass der erwartete Endwert in Abhängigkeit von `variable2` bestimmt werden kann. In der dritten Zeile wird gezeigt, wie dies in dieser Skriptsprache erreicht wird. Nach dem `init` folgt in Klammern der Name der Variable, an die die entsprechende Variable gebunden werden soll, danach mit `WITH` $\{\ldots\}$” wie `ergebnis` initialisiert werden soll. Zu beachten bleibt, dass die Anzahl der Initialisierungswerte bei Variablen, die aneinander gebunden sind, übereinstimmen muss.

Weiterhin kann es sinnvoll sein, eine Toleranz für den erwarteten Endwert anzugeben. Im Falle der ACC-Software werden beispielswesie keine Floatingpoint-Operationen unterstützt. Aus diesem Grund werden viele Normierungen verwendet, um Rundungsfehler minimal zu halten. Trotzdem können diese auftreten. Für diesen Fall kann durch das `DELTA` bestimmt werden, wieviel Toleranz erlaubt ist, also wie hoch der Rundungsfehler werden darf. Für das `DELTA` dürfen sowohl feste Werte wie auch relative Größen(5%) verwendet werden.

Werden statt normalen Variablen Makros verwendet, so kann RTRT diese normalerweise problemlos ausreferenzieren, und die Testfallerstellung verläuft wie bei normalen Variablen. Probleme treten auf, sobald eine Negation verwendet wird. In diesem Fall wird das Makro als "Expression" erkannt und somit kann nur noch der gewünschte Ausgabewert definiert werden, nicht jedoch der Eingabewert. Aus diesem Grund sollte auf solche Makros verzichtet werden, da sonst nicht jedes Verhalten einer Funktion getestet werden kann.

Wenn das Testbett erstellt wird, muss spezifiziert werden, welche Funktionen integriert werden sollen. Alle anderen Funktionen werden als extern verstanden und somit müssen für sie Platzhalter definiert werden. Diese Platzhalter speichern die Eingabewerte mit denen die entsprechenden Funktion aufgerufen wird und liefern spezifizierte Ausgabewerte der Funktion zurück. RTRT erzeugt die Platzhalter automatisch. Wie oft die Funktion jedoch aufgerufen werden darf, und mit welchen Ein- und Ausgabewerten, muß durch den Tester spezifiziert werden.

Die durch RTRT erzeugte Definition der Platzhalter findet sich in dem `DEFINE STUB ... END DEFINE`- Block. Zunächst muss hier festgelegt werden, welche Ein- und Ausgabewerte getestet werden sollen. Hierbei wird zwischen vier Möglichkeiten unterschieden:

- "*_in*" vor dem Parameter bedeutet, dass er als Eingangswert getestet werden soll
- "*_out*" bedeutet, das dessen Ausgabewert bestimmt werden soll
- "*_inout*" Parameter werden zunächst getestet und dann mit Ausgabewert belegt
- "*_no*" Parameter werden weder getestet noch belegt

Eine mögliche Definition würde lauten:

```
DEFINE STUB beispiel
    #int externe_funktion(int _in var1, int _no var2)
END DEFINE
```

In dieser Definition ist der Platzhalter für `externe_funktion` so spezifiziert, dass der Eingangswert von `var1` getestet würde, jedoch nicht belegt, `var2` würde weder getestet noch belegt.

Wird ein Platzhalter für eine externe Funktion nur definiert, aber nirgends benutzt, so wird implizit angenommen, dass die Funktion in keinem Testfall aufgerufen wird. Sollte sie doch aufgerufen werden, so schlägt der entsprechende Test fehl. Aus diesem Grund müssen auch in Testfällen, in denen eine Funktion nicht getestet werden soll, diese aber aufgerufen wird, deren Platzhalter spezifiziert werden.

Platzhalter können in einer `ENVIRONMENT`-umgebung oder in einem Test deklariert werden. Die Deklaration des Platzhalters aus dem obigen Beispiel könnte wie folgt spezifiziert sein:

```
STUB externe_funktion 1=>(2,3)9, 2=>(3,4)4, others=>(0,0)1
```

Dies würde bedeuten, dass erwartet wird, dass `externe_funktion` mindestens zweimal aufgerufen wird. Beim ersten Aufruf(1 =>) soll getestet werden, ob `var1==2`, `var2` sollte den Wert drei besitzen, wird aber nicht geprüft(aufgrund der Platzhalterdefinition). Die Funktion soll neun zurückliefern. Beim zweiten Aufruf wird `var1` auf "3" getestet, die Funktion soll vier zurückliefern. Alle Aufrufe danach(`others =>`) sollen "1" zurückliefern.

Beim Testen müssen oft mehrere Testfälle geschrieben werden - beim Whiteboxtest mit Zweigüberdeckung beispielsweise für jeden Zweig ein Testfall, beim Blackboxtest ein Testfall für jede Äquivalenzklasse. Meist müssen für diese Tests viele Eingangsgrößen definiert werden, doch nur wenige davon ändern sich zwischen den Tests. In diesem Fall ist es sinnvoll, die Variablen, die sich nicht ändern, in einem Environment zu deklarieren. Dieses Environment kann dann bei allen Testfällen integriert werden.

In einem Serviceblock können mehrere Environments verwendet werden. Die Deklaration eines Environments sieht folgendermaßen aus:

```
ENVIRONMENT umgebung1
    STUB externe_funktion 1=>(2,2)1, others=>(0,0)1
    VAR var1, init=1, ev =3
    VAR var2, init=2, ev =4
END ENVIRONMENT
USE umgebung1
```

In diesem Beispiel würde `umgebung1` die Belegung der Variablen `var1`, `var2` sowie die der externen Funktion `externe_funktion` spezifizieren.

Im Folgenden wird gezeigt, wie die Sprache zur Testfallerstellung für Blackboxtests verwendet werden kann. Danach wird das Vorgehen beim Whiteboxtest veranschaulicht.

A.2.1 Testfallerstellung für Blackboxtests am Beispiel

Bei Blackboxtestmethoden werden die Tests ausschließlich anhand der Requirements erstellt. Um zu zeigen, wie Testfälle nach unserer Blackboxmethode erstellt werden können, verwenden wir die Requirements zu der Fahrerübernahmeaufforderung-Anzeige der Primäranzeigen(vgl. Kapitel 2.2.1).

Wie genau nach der Blackboxmethode Testfälle definiert werden können, haben wir bereits in Abschnitt 4.4.3 gezeigt. Im Folgenden wird die Umsetzung dieses Beispiels erläutert.

Wir schreiben nun ein Testskript, das alle Äquivalenzklassen, in denen das Signal leuchtet, zusammenfasst - nur die Möglichkeit, dass sich das System im `Suspend`-Modus befindet, soll gesondert betrachtet werden. Den `Suspend`-Zustand gesondert zu betrachten ist sinnvoll, da andernfalls die drei Variablen zur Bedingung der Fahrerübernahmeaufforderung mit der `StateInternal` Variable verknüpft werden müssten. Indem wir den `Suspend`fall gesondert betrachten, müssen nur die normalen Bedingungen verknüpft werden, so kann bei der Testfallspezifikation stark vereinfacht werden.

Es besteht also mindestens eine Fahrerübernahmeaufforderung, entweder leuchtet das Symbol bereits und der `ToggleTimer` ist noch nicht abgelaufen, oder das Signal wurde noch nicht beleuchtet und soll nun auf Beleuchtung springen. Aus Gründen der Übersichtlichkeit wurden hier auf einige Zustände für `GetInternalState` verzichtet. In einem echten Test sollten allerdings alle Möglichkeiten geprüft werden.

Zunächst müssen die Variablen, die den Zustand des Systems spezifizieren sowie die Eingangs- und Ausgangsvariablen bestimmt werden. In Abschnitt 4.4.3 haben wir diese Variablen bereits in Tabelle 4.2 zusammengefasst. Normale Variablen können einfach als `VAR` in das Testskript aufgenommen werden. In der Beispielsoftware werden allerdings oft Makros verwendet. Diese kann RTRT nicht immer selbst auflösen. Daher müssen diese Makros vom Tester aufgelöst werden. In der Fahrerübernahmeaufforderung findet sich ein solches Makro beispielsweise unter `SignalLeuchtet?`. Im dazugehörigen C-Programmtext wird das Signal wie folgt definiert:

```
#define SignalLeuchtet?  ((ACCByte6 &= (0x40)) >> 6)
```

Soll also geprüft werden, ob das Signal leuchtet, so muss das siebte Bit des `ACCByte6` geprüft werden. Statt `SignalLeuchtet?` wird im Test also `ACCByte6` geprüft, weiterhin bietet es sich an, nach Aufruf der Funktion eine Bitmaske über das Byte zu legen, um so andere Signale auszublenden. Somit ergibt sich folgender Zwischenstand:

```
TEST Fahreruebernahmeaufforderung_ON
    FAMILY nominal
        ELEMENT
```

```
            VAR GetInternalState, init IN {InitState,
                        & IrreversibleFailureState,
                        & ReversibleFailureState,
                        & RejectState, ActiveControlState},
                        & ev==
            //Fahrerübernahmeaufforderung
            VAR GetTORactive, init==, ev==
            VAR GetLatentWarningFilt(), init==, ev==
            VAR GetPreWarningFilt(), init==, ev==

            VAR ToggleTimer, init==, ev==

            //SymbolLeuchtet
            VAR ACCByte6, init==, ev==

            //nun der Aufruf der Funktion und Bitmaske
            #CalcMsgAcc();
            #ACCByte6 &=0x40;

        END ELEMENT
END TEST
```

Nun müssen wir den Test weiter spezifizieren. Es soll mindestens eine Fahrerübernahmeaufforderungsbedingung gelten. Da wir nur die Fälle prüfen wollen, in denen keine Fehlermaskierung auftreten kann, betrachten wir nur die Fälle, in denen *genau eine* der Bedingungen gilt. Die drei Bedingungsvariablen müssen also aneinander gebunden werden, und dann so belegt werden, dass immer nur eine der Variablen mit *true* belegt wird. Daraus ergibt sich für die Variablen folgende Belegung:

```
//Fahrerübernahmeaufforderung
    VAR GetTORactive, init IN {1,0,0}, ev=init
    VAR GetLatentWarningFilt(),
        & init (GetTORactive) WITH {0,1,0},  ev=init
    VAR GetPreWarningFilt(),
        & init (GetTORactive) WITH {0,0,1},  ev=init
```

Weiterhin muss der `ToggleTimer` mit `ACCByte6` verknüpft werden, da das Signal nur leuchten soll, falls die Anzeige bereits leuchtet und der Timer noch nicht abgelaufen ist, oder falls die Anzeige gerade nicht leuchtete, nun aber auf Leuchten springen soll(`ToggleTimer` ist abgelaufen). Der Timer wird mit einem Anzeigeparameter des Werts 538 verglichen. Ist der Timer größer als der Anzeigeparameter, so ist der Timer abgelaufen und wird auf 0 gesetzt. Andernfalls wird er um 5 erhöht. Leuchtet die Anzeige, so ist das siebte Bit gesetzt. Dies führt zu folgender Belegung:

```
//Fall 1-4: Anzeige leuchtet, ToggleTimer nicht abgelaufen
//Fall 5,6: Anzeige leuchtet nicht, ToggleTimer abgelaufen
// 7.Bit bei ACCByte6: 0x40 = 64(dez)

VAR ToggleTimer,  init IN {0,1,537,538,539,590},
                  & ev IN {5,6,542,543, 0,0}
VAR ACCByte6,     init (ToggleTimer) WITH {255,64,255,64,63,0},
                  & ev IN {64,64,64,64, 64,64}
```

Fügen wir nun unsere Zwischenergebnisse in das Testfile ein, so ergibt sich die folgende Belegung. Hierbei sind die Variablen für die Bedingung der Fahrerübernahmeaufforderung verknüpft(drei Möglichkeiten) sowie der `ToggleTimer` mit dem Signal(sechs Möglichkeiten). Im `InternalState` werden zwischen fünf Möglichkeiten unterschieden. Somit ergeben sich $3 \cdot 6 \cdot 5 = 90$ verschiedene Testdurchläufe für diesen Test.

```
SERVICE Fahreruebernahme
SERVICE_TYPE extern

 TEST Fahreruebernahmeaufforderung_ON
  FAMILY nominal
     ELEMENT
        VAR GetInternalState,
             & init IN {InitState, IrreversibleFailureState,
                        & ReversibleFailureState,
                        & RejectState, ActiveControlState},
                        & ev==

        //Fahrerübernahmeaufforderung
        VAR GetTORactive, init IN {1,0,0}, ev=init
        VAR GetLatentWarningFilt(),
             & init (GetTORactive) WITH {0,1,0},  ev=init
        VAR GetPreWarningFilt(),
             & init (GetTORactive) WITH {0,0,1},  ev=init

        VAR ToggleTimer,
             & init IN {0,1,537,538,  539,590},
             & ev   IN {5,6,542,543,  0,0}
        VAR ACCByte6,
             & init (ToggleTimer) WITH {255,64,255,64,  63,0},
             & ev                    IN {64, 64,64, 64,  64,64}
        //nun der Aufruf der Funktion
        #CalcMsgAcc();
        #ACCByte6 &=0x40;
```

```
    END ELEMENT
 END TEST
END SERVICE
```

Nun ist der erste Testfall umgesetzt worden. Als nächstes müssten die Testfälle zum `Suspend`Mode realisiert werden.

Im nächsten Abschnitt wird gezeigt, wie die Testfälle für Whiteboxtests umgesetzt werden.

A.2.2 Testfallerstellung für Whiteboxtests am Beispiel

Im vorigen Abschnitt wurde gezeigt, wie die erdachten Testfälle in unserer Blackboxmethode umgesetzt werden können. In diesem Abschnitt wird dargestellt, wie dies nach der Whiteboxmethode erfolgen kann.

Bei Whiteboxtestmethoden werden die Tests anhand der Requirements sowie des Programmtexts erstellt. Um zu zeigen, wie Testfälle nach unserer Whiteboxmethode erstellt werden können, verwenden wir wie zuvor die Requirements zu der Fahrerübernahmeaufforderungs-Anzeige der Primäranzeigen (vgl. Kapitel 2.2.1). Wie genau Testfälle nach einer Whiteboxmethode entwickelt werden können, haben wir bereits in Abschnitt 4.4.4 gezeigt. im Folgenden wird die Umsetzung dieses Beispiels erläutert.

In dem Beispiel wurden bereits verschiedene Testfälle, die sich aus unterschiedlichen Überdeckungsgraden ergeben, vorgestellt. Welche Überdeckung gewählt wird, muss bereits im Projektplan spezifiziert werden. Sollte damit das Testendekriterium(vergleiche Kapitel 7) nicht erreicht werden können, so muss die Überdeckung gegebenenfalls verfeinert werden. Wir verwenden in unserem Beispiel mindestens Zweigüberdeckung.

Die Idee ist dabei, einen Testfall für jeden Zweig des zugehörigen Flußdiagramms zu schreiben. Je nach gewählter Bedingungsüberdeckung benötigen wir unterschiedlich viele Belegungen, die zu diesem Zweig führen. Da die modifizierte Bedingungsüberdeckung am genauesten unterscheidet, wählen wir zu Anschauungszwecken diese Überdeckung.

Wie bereits in Abschnitt 4.4.4 angesprochen, existieren für den linken Zweig keine Requirements. Dieser kann demnach nicht überprüft werden. Daher wird das Fehlen der Requirements in die Fehlerdatenbank übernommen.

Wir zeigen nun exemplarisch wie ein Testfall für den sechsten Zweig umgesetzt werden kann. Zunächst müssen die Variablen so gesetzt werden, dass dieser Zweig erreicht wird. Dazu muss die Variable `TORTestOver` auf *true* gesetzt werden. In unseren Requirements steht dazu nichts. Aus diesem Grund muss hier eine Notiz in die Fehlerdatenbank geschrieben werden, und beim Requirementmanager nachgefragt werden, ob dies Verhalten so gewünscht ist. Um die Tests fortzusetzen belegen wir:

```
VAR TORTestOver, init =1, ev=init
```

Weiterhin darf keine der Fahrerübernahmebedingungen gelten. Die erste der Bedingungen ist eine zusammengesetzte Bedingung. Da wir die modifizierte Bedingungsüberdeckung gewählt haben, muss hier genau unterschieden werden. Die erste Bedingungsvariable muss somit mit dem internen Zustand des Systems verknüpft werden. Die anderen zwei Bedingungsvariablen müssen auf $false$ gesetzt werden. Es ergibt sich folgende Belegung:

```
//Fahrerübernahmeaufforderung
 VAR GetTORactive,
    & init IN {0,0,0,0,0, 1}, ev=init
 VAR GetLatentWarningFilt(), init =0, ev=init
 VAR GetPreWarningFilt(),    init =0, ev=init

 VAR GetInternalState,
    & init (GetTORactive) WITH {InitState,
    & IrreversibleFailureState,
    & ReversibleFailureState, ActiveControlState,
    & RejectState, SuspendState}, ev==
```

Jedesmal, wenn der interne Zustand nicht im `SuspendState` ist, muss `GetTORactive` $false$ sein. Ist der Zustand im `SuspendState` so wird `GetTORactive` mit $true$ belegt. Auf diese Weise sollen Fehlermaskierungen vermieden werden.

Weiterhin soll, da keine Fahrerübernahmebedingung gilt, das Symbol auch nicht leuchten. Der erwartete Endwert des `ACCByte6` muss daher auf 0 gesetzt werden. Um zu prüfen, dass das Bit aktiv gesetzt wird, bietet es sich an, mindestens zwei Fälle zu unterscheiden: den Fall, wo das Bit aktiv gesetzt wird, und den, in dem das Bit bereits gesetzt war.

Es ergibt sich folgender Testfall:

```
SERVICE FahreruebernahmeWB
SERVICE_TYPE extern

 TEST Fahreruebernahmeaufforderung_Zweig6
  FAMILY nominal
     ELEMENT

        VAR TORTestOver, init =1, ev=init

        //Fahrerübernahmeaufforderung
        VAR GetTORactive,
            & init IN {0,0,0,0,0, 1}, ev=init
        VAR GetLatentWarningFilt(), init =0, ev=init
        VAR GetPreWarningFilt(),    init =0, ev=init
```

```
        VAR GetInternalState,
           & init (GetTORactive) WITH {InitState,
           & IrreversibleFailureState,
           & ReversibleFailureState, ActiveControlState,
           & RejectState, SuspendState}, ev==

        VAR ACCByte6,
             & init IN {64,0},
             & ev   IN {0,0}
        //nun der Aufruf der Funktion
        #CalcMsgAcc();
        #ACCByte6 &=0x40;
    END ELEMENT
 END TEST
END SERVICE
```

Für `GetTORactive` und das verknüpfte `GetInternalState` wurden in diesem Beispiel sechs Zustände spezifiziert. `ACCByte6` unterscheidet nach zwei Zuständen, alle anderen Variablen werden fest gesetzt. Somit ergeben sich $2 \cdot 6 = 12$ Testdurchläufe für diesen Testfall.

Nach demselben Prinzip müssten nun für die anderen Zweige Testfälle definiert werden. Ist die Erstellung der Testfälle beendet, müssen die Ergebnisse analysiert werden. Dieses Vorgehen wird im nächsten Abschnitt erläutert.

A.3 Analyse der Ergebnisse

In den letzten zwei Abschnitten wurde gezeigt, wie Testfälle geschrieben werden können. In diesem Abschnitt soll nun gezeigt werden, wie danach die Ergebnisse ausgewertet werden können.

Sind die Testfälle fertig spezifiziert, so kann der Test ausgeführt werden. Hierzu muss zunächst der Test, der ausgeführt werden soll, ausgewählt werden. Dies geschieht, indem man mit der rechten Maustaste auf das Hauptptu-file klickt. Dann öffnet sich ein Untermenü. In diesem muss der Menüpunkt “Testselection” gewählt werden, anschließend können die Tests ausgewählt werden, die ausgeführt werden sollen. Es kann nach `SERVICE`, `FAMILY` und Testnamen ausgewählt werden (vgl. Kapitel A.2).

Weiterhin müssen die Informationen, die protokolliert werden sollen, ausgewählt werden. Dies kann in der Hauptmenüleiste unter “Projects” → “Settings” erfolgen. In dem sich öffnenden Fenster kann die Konfiguration des Systems verändert werden. Unter “Runtime Analysis” können bei “Programmtext Coverage” → “Instrumentation Control” die gewünschten Überdeckungsarten festgelegt werden. RTRT protokolliert dann bei der Test-

ausführung diese Überdeckungsarten, und liefert im Programmtext Coverage Report Informationen zu folgenden drei verschiedenen Aspekten:

- eine allgemeine Übersicht, wie viel Prozent der gewählten Überdeckungsart über den gesamten Programmtexts des Testmoduls erreicht wurde;
- eine genauere Übersicht, welche Funktion welchen Überdeckungsgrad erreicht hat;
- SourceProgrammtext, in dem markiert ist, welcher Programmtext des getesteten Moduls bereits überdeckt wurde und welcher Programmtext noch nicht besucht wurde.

. Während der Testausführung erstellt RTRT die Analyseprotokolle. Für uns sind die Testprotokolle sowie die Programmtext Coverage Protokolle interessant. Im Folgenden wird daher nur auf diese Reports eingegangen.

Oft schlagen viele Testfälle aufgrund derselben Fehlerursache fehl. Daher macht es Sinn, die fehlgeschlagenen Testfälle zusammen auszuwerten, statt jeden Testfall einzeln.

Es muss überprüft werden, ob der eigene Testfall richtig spezifiziert wurde. Ist dies der Fall, kann der Fehler gesucht werden. Für diesen Zweck eignet sich der Coverage Report. Da in dem Programmtext sehr genau markiert ist, welcher Code durchlaufen wurde, und welcher nicht, wird der Suchraum, in dem der Fehler liegen kann, stark eingeschränkt. Auf diese Weise kann die Fehlersuche deutlich effizienter ablaufen. Ist der Fehler gefunden, kann er in die Fehlerdatenbank aufgenommen werden.

Die Coverage Reports eignen sich jedoch nicht nur für die Fehlersuche, sie berechnen auch die Statistiken, wie viel Programmtext insgesamt bereits abgedeckt wurde (in Abhängigkeit von dem gewählten Überdeckungsgrad). Diese Statistiken werden für das Testendekriterium benötigt. Weiterhin geben sie Informationen darüber, was für Testfälle noch geschrieben werden müssen, um die Überdeckung zu erhöhen. Beispielsweise wird bei der “Modified Condition Coverage“ gezielt ausgegeben, welche Bedingungen noch nicht ausreichend geprüft wurden, und welche Belegungen hierfür noch zu testen sind. Auf diese Weise unterstützt RTRT die Spezifikation von Testfällen.

Literaturverzeichnis

[91201] ISO/IEC 9126. Software engineering - Product quality - Part1: Quality Model. Technical report, ISO/IEC 9126, 2001.

[Age05] Agena Limited, 32-33 Hatton Garden, London EC1N 8DL. *Agena Ltd*, 2005.

[AOS04] O. Armbrust, M. Ochs, and B. Snoek. Stand der Forschung von Software-Tests und deren Automatisierung. IESE-Report 068.04/D, Frauenhofer Institut, June 2004.

[A/S05] Hugin A/S. *www.hugin.com*. Hugin Expert A/S, P.O. Box 8201 DK-9220 Aalborg, Denmark, 2005.

[Boe73] B.W. Boehm. Software and its Impact: A quantitative Assessment. *Datmation*, 19:48–59, 1973.

[Boe79] B.W. Boehm. Guidelines for Verifying and Validation Software Requirements and Design Specification. In *Euro IFIP*, pages 711–719. Euro IFIP, 1979.

[BP99] L. Briand and D. Pfahl. Using Simulation for assessing the Real Impact of Test Coverage on Defect Coverage. IESE-Report 018.99/E, Frauenhofer Institut—Experimentelles Software Engineering, march 1999.

[BPS00] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Software Practice and Experience*, 30(7):775–802, 2000.

[CCG+04] A. Cain, T.Y. Chen, D. Grant, P. Poon, S. Tang, and T. Tse. *Software Engineering Research and Applications, Lecture Notes in Computer Science*, volume 3026, chapter An Automatic Test Data Generation System Based on the Integrated Classification-Tree Methodology, pages 225—238. Springer-Verlag, Heidelberg, Germany, 2004.

[Cen02] M. V. Cengarle. Inspection and Testing: Toward Combining both Approaches. IESE-Report 024.02/E, Frauenhofer Institut—Experimentelles Software Engineering, 2002.

[Cen05] IBM Publication Center. *Rational Test RealTime Reference Manual Version 2003.06.13.* IBM, www-1.ibm.com/support/docview.wss?rs, Oktober 2005.

[DDH72] O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. *Structured Programming.* Academic Press, London, 1972.

[FN00] N.E. Fenton and M. Neil. *22nd International Conference on Software Engineering—The Future of Software Engineering*, chapter Software Metrics: Roadmap, pages 357–370. ACM Press, 2000.

[FN04] N.E. Fenton and M. Neil. Combining evidence in risk analysis using Bayesian Networks. *Safety Critical Systems Club Newsletter*, 13(4):8–13, 2004.

[FO00] N.E. Fenton and N. Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering*, 26(8):797–814, 2000.

[Gib94] W. Gibbs. Software's Chronic Crisis. *Scientific American*, 271(3):86–95, 1994.

[Gle96] J. Gleick. A Bug and a Crash. *New York Times Magazine*, page 3p., Dezember 1996.

[Gor89] M.J.C Gordon. Lectures on the specification and verification of hardware. Course Notes, University of Cambridge, 1989.

[Gra92] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[Gre03] D. Grell. Rad am Draht—Innovationslawine in der Autotechnik. *c't*, 14(170), 2003.

[Ham95] Richard G. Hamlet. Software Quality, Software Process, and Software Testing. *Advances in Computers*, 41:191–229, 1995.

[Hau03] H. Haupt. Immer häufiger Pannen durch mangelhafte Elektronik. *SpiegelOnline*, 21.1.2003.

[HLL94] J.R. Horgan, M.R. Lyu, and S. London. Achieving Software Quality with Testing Coverage Measures. *IEEE Computer*, pages 60–69, 1994.

[HS01] Gerard J. Holzmann and Margaret H. Smith. Software model checking: Extracting Verification Models from Source Code. *Software Testing, Verification and Reliability*, 11(2):65–79, 2001.

[Kro99] Thomas Kropf. *Introduction to Formal Hardware Verification*. Springer Verlag, Heidelberg, 1999.

[Kur97] Robert P. Kurshan. Formal Verification in a Commercial Setting. In *Design Automation Conference*, pages 258–262, 1997.

[Lan96] G. Le Lann. The Ariane 5 Flight 501 Failure - A Case Study in System Engineering for Computing Systems. Technical Report RR-3079, Institut National de Recherche en Informatique et en Automatique, 1996.

[LT93] N. G. Leveson and C. S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, 1993.

[MD97] Y. K. Malaiya and J. A. Denton. What do software reliability parameters represent? In *Proc. International Symposium on Software Reliability Engineering*, pages 124–135, Albuquerque, NM, November 1997.

[MD98] Y.K. Malaiya and J. A. Denton. Estimating the number of defects: A simple and intuitive approach. In *International Symposium of Software Reliability Engineering*, pages 307–315, Paderborn, Germany, November 1998.

[MIS05] The Motor Industry Software Reliability Association MISRA. Misra-homepage. http://www.misra.org.uk, Dezember 2005.

[Mye78] G. J. Myers. A controlled Experiment in Program Testing and Code Walkthroughs/Inspections. *Communications of the ACM*, 21(9):760–768, 1978.

[Mye82] Glenford J. Myers. *Methodisches Testen von Programmen*. R. Oldenbourg Verlag, München, 1982.

[NB05a] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 580–586, New York, NY, USA, 2005. ACM Press.

[NB05b] Nachiappan Nagappan and Thomas Ball. Use of Relative Code Churn Measures to Predict System Defect Density. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 284–292, New York, NY, USA, 2005. ACM Press.

[Neu95] Peter G. Neumann. *Computer related Risks.* ACM Press—Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[Per03] William E. Perry. *Software testen.* mitp-Verlag, Bonn, 2003.

[PKS00] M. Pol, T. Koomen, and A. Spillner. *Management und Optimierung des Testprozesses.* dpunkt.verlag, Heidelberg, 2000.

[Pol05] Produktbeschreibung Polyspace. http://www.polyspace.com, Dezember 2005.

[QA 05] Produktbeschreibung QA-C. http://www.qa-systems.de, Dezember 2005.

[Rep96] Inquiry Board Report. Ariane 5 - Flight 501 Failure. page 18 p., 1996.

[RHS98] L. Rosenberg, T. Hammer, and J. Shaw. Software Metrics and Reliability. *9th International Symposium—Best Paper Award*, page 8p., November 1998.

[Rie97] E.H. Riedemann. *Testmethoden für sequentielle und nebenläufige Softwaresysteme.* Teubner-Verlag, Stuttgart, 1997.

[SL04] Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester.* dpunkt.verlag, Heidelberg, 2004.

[Spi04] Die Krux mit der Elektronik. *SpiegelOnline*, 30.4.2004.

[Vec05] Produktbeschreibung CANalyzer. http://www.canalyzer.de, Dezember 2005.