6 Softwarequalität eingebetteter Systeme

Der Qualität von eingebetteter Software eine besondere Bedeutung zuzumessen, ist eine lohnenswerte Investition. Dies belegen zahlreiche Beispiele, bei denen fehlerhafte Software zu Schädigungen von Maschinen oder gar Menschen geführt hat. Einige prominente Fälle, wo Softwarefehler zu massiven Konsequenzen geführt haben, schildern wir daher in diesem Kapitel, bevor wir dann zentrale Begriffe der Softwarequalität diskutieren. Um die Softwarequalität überprüfen zu können, sind spezielle Prüftechniken erforderlich. Wir geben zunächst eine Übersicht über derzeit mögliche Prüftechniken und schildern dann ausgewählte Vertreter etwas näher.

Kapitelübersicht

Die Qualitätssicherung eingebetteter Systeme, gerade auch in der Automobiltechnik oder Avionik ist eine schwierige aber gleichwohl dringend erforderliche Entwicklungsaufgabe. Nach diesem Kapitel sollte der Leser die beiden Begriffe Zuverlässigkeit und Sicherheit sowie die damit in Bezug stehenden Begriffe verstanden haben und Verfahren zu ihrer Herstellung kennen.

6.1 Motivation

Beispiel Softwarefehler Airbus A-320

Ein „eindrucksvolles“ Beispiel für einen Softwarefehler ist die Bruchlandung eines *Airbus A-320* auf dem Warschauer Flughafen am 14. September 1993: Ein Lufthansa-Airbus fängt bei der Landung in Warschau Feuer. Bei dem Unfall sterben zwei Menschen, 54 werden verletzt. Ursache war eine Fehlkonstruktion des Sensors zur Erkennung der Bodenberührung: Im „Flight Mode“ ließ sich die zum Bremsen notwendige Schubumkehr nicht einschalten. Hier handelte es sich um keinen Pilotenfehler, sondern

um falsche Entwurfsentscheidungen der Konstrukteure und Software-Ingenieure.

Beispiel SW-Fehler Sojus-Kapsel

Ein anderes Beispiel, wo ein Softwarefehler beinahe zu einer Katastrophe geführt hatte, war die Landung einer in einer *„Sojus"-Kapsel* zurückgekehrten ISS-Mannschaft im Jahre 2003. Nach russischen Angaben führte ein Softwarefehler zu einem absturzartigen Wiedereintritt, bei dem ein vom Computer falsch berechneter Eintrittswinkel zu einer übermäßigen Hitzeentwicklung geführt hat.

Beispiel SW-Fehler Ariane 5

Einer der wohl bekanntesten Repräsentanten für mangelnde Softwarequalität ist der Absturz bzw. die ferngelenkte Zerstörung der europäischen Trägerrakete *Ariane 5* auf ihrem Jungfernflug am 4. Juni 1996 in Kourou. Die Rakete war mit vier Satelliten bestückt. Etwa 37 Sekunden nach dem Start erreichte die Ariane 5 eine Horizontalgeschwindigkeit von mehr als 32.768 internen Einheiten. Die in der Programmiersprache Ada realisierte Konvertierung dieses Wertes in eine vorzeichenbehaftete Integer-Variable führte daher zu einem Überlauf, der nicht abgefangen wurde, obwohl die Hardware redundant ausgelegt war. Da es sich hier jedoch um einen reinen Softwarefehler handelte, blieb diese Redundanz wirkungslos. Folglich wurden Diagnosedaten zum Hauptrechner geschickt, die dieser als Flugbahndaten interpretierte, so dass dieser unsinnige Steuerbefehle generierte. Die Rakete drohte daraufhin zu bersten und musste ferngelenkt gesprengt werden, um größeren Schaden aufgrund der noch niedrigen Höhe zu vermeiden. Interessanterweise war die Software von der Ariane 4 wiederverwendet worden und hatte dort auch problemlos funktioniert. Der Gesamtschaden belief sich auf 500 Millionen US-Dollar.

6.2 Begriffe

SW-Qualität

Der Begriff der Softwarequalität ist nach der Norm ISO/IEC 9126 wie folgt definiert:

Definition (Softwarequalität) nach ISO/IEC 9126:
Softwarequalität ist die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.

Der gesamte Standard ISO/IEC 9126 umfasst vier Teile, nämlich 9126-1 bis 9126-4.

Die eigentliche, konkrete Beurteilung der Qualität geschieht mit Hilfe von *Qualitätsmerkmalen.* Sie stellen Eigenschaften einer Funktionseinheit dar, anhand derer ihre Qualität beschrieben und beurteilt werden, die jedoch keine Aussage über den Grad der Ausprägung enthalten. Ein Qualitätsmerkmal kann ggf. hierarchisch in mehrere Teilmerkmale verfeinert werden (Liggesmeyer, 2002).

SW-Qualitätsmerkmale

Die ISO/IEC 9126 unterscheidet beispielsweise zwischen den folgenden *Softwarequalitätsmerkmalen:*

- Funktionalität,
- Zuverlässigkeit,
- Benutzbarkeit,
- Effizienz,
- Änderbarkeit und
- Übertragbarkeit.

Zwischen Qualitätsmerkmalen können Wechselwirkungen und Abhängigkeiten bestehen. Die Forderung nach einer insgesamt in jeder Hinsicht bestmöglichen Qualität ist daher unsinnig (Liggesmeyer, 2002).

Failure, Fault, Error

In der deutschen Sprache spricht man gerne etwas ungenau von einem „Softwarefehler". Richtig ist es aber vielmehr, zwischen den Begriffen „Fehlverhalten", „Fehler" und „Irrtum" zu unterscheiden (siehe DIN 66271). Auch im Englischen existieren hier drei unterschiedliche Ausdrücke:

- **Failure:** Es handelt sich um ein Fehlverhalten eines Programms, das während seiner Ausführung tatsächlich auftritt *(Fehlverhalten, Fehlerwirkung, äußerer Fehler).*
- **Fault:** Es handelt sich um eine fehlerhafte Stelle (Zeile) eines Programms, die ein Fehlverhalten auslösen kann *(Fehler, Fehlerzustand, innerer Fehler).*
- **Error:** Es handelt sich um eine fehlerhafte Aktion, die zu einer fehlerhaften Programmstelle führt *(Irrtum, Fehlhandlung).*

Qualitätssicherung

Auf Basis dieser Unterscheidung können wir folgende Feststellungen ableiten: Fehler (errors) bei der Programmentwicklung können zu Fehlern (fault)s in einem Programm führen, die ihrerseits Fehler (failure) bei der Programmausführung bewirken können. Die *konstruktive Qualitätssicherung* reduziert menschliche Fehler (errors). Die *analytische Qualitätssicherung* entdeckt Programm-

fehler (faults). *Testen* löst Laufzeitfehler aus (failures) und führt zur Entdeckung von Programmfehlern (faults).

Korrektheit

Neben dem Fehlerbegriff und den in der ISO/IEC 9126 aufgelisteten Softwarequalitätsmerkmalen steht die Definition des Qualitätsmerkmals der *Korrektheit* besonders im Vordergrund.

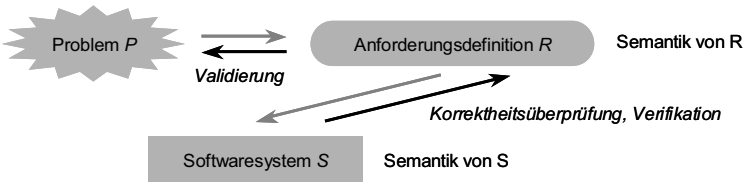
Bei (Endres, 1977) wird die Korrektheit erstmals als Synonym für Fehlerfreiheit als die Übereinstimmung zwischen dem beobachteten und dem gewünschten Verhalten definiert: „... ein Programm ist korrekt, wenn es frei ist von logischen Fehlern, d. h. wenn seine Implementierung übereinstimmt mit einer mehr oder weniger expliziten Spezifikation dessen, was das Programm tun soll."

Die IEEE definiert (vgl. ANSI 72983) Korrektheit als den Grad der Konsistenz zwischen Spezifikation und Programm bzw. als Grad der Erfüllung der Benutzererwartung durch ein Programm (Liggesmeyer, 2002).

Verifikation, Validierung

Die eben genannten Definitionen sind eher praxisorientiert als für eine optimale formale und damit „scharfe" Definition des Begriffs geeignet. Beispielsweise wird der Grad der Erfüllung der Erwartungen eines möglichen Benutzers in erheblicher Weise von der Auswahl der Befragten beeinflusst. Darüber hinaus können bereits die Programmspezifikation und die tatsächliche Benutzererwartung differieren. Aus diesem Grund unterscheidet man beim Nachweis, ob ein Programm mit seiner Spezifikation bzw. den Benutzeranforderungen übereinstimmt zwischen den beiden Begriffe der Verifikation bzw. der Validierung. Die *Verifikation* überprüft, ob das Programm mit seiner (formalen) Spezifikation übereinstimmt, die *Validierung* (oder Validation) stellt dagegen fest, ob die Spezifikation und die tatsächlichen Benutzeranforderungen identisch sind. Abbildung 6.1 gibt einen Überblick.

Abb. 6.1: Einordnung der Begriffe Korrektheit, Verifikation, Valididerung



In der Informatik hat sich darum der folgende Korrektheitsbegriff etabliert (Liggesmeyer, 2002):

- Korrektheit besitzt keinen graduellen Charakter, d. h. eine Software ist entweder korrekt oder nicht korrekt.

- Eine fehlerfreie Software ist korrekt.
- Eine Software ist korrekt, wenn sie konsistent zu ihrer Spezifikation ist.
- Existiert zu einer Software keine Spezifikation, so ist keine Überprüfung der Korrektheit möglich.

Fehlerfreie Softwaresysteme gibt es derzeit nicht und wird es mit hoher Wahrscheinlichkeit auch in mittelfristiger Zukunft nicht geben, sobald die Software einen gewissen Komplexitätsgrad überschreitet. Ein Fehlerzustand liegt oftmals darin begründet, dass sowohl während der Softwareentwicklung als auch bei der Überprüfung auf Fehler an bestimmte Ausnahmesituationen nicht gedacht wurde und dann auch nicht überprüft wurden (Spillner und Linz, 2003).

Prüftechniken

Im Folgenden wollen wir nun einige Techniken kennen lernen, welche in der Lage sind, größtenteils durch maschinelle Unterstützung, eine Überprüfung eines Programms hinsichtlich seiner Korrektheit durchzuführen. Diese Techniken werden im Allgemeinen als *Prüftechniken* bezeichnet. Bevor wir einige wenige, aber besonders gebräuchlich herausgreifen werden, wollen wir zunächst ein (mögliches und anerkanntes) Klassifikationsschema für Software-Prüftechniken angeben, das von (Liggesmeyer, 2002) vorgeschlagen wurde (vgl. Abbildung 6.2).

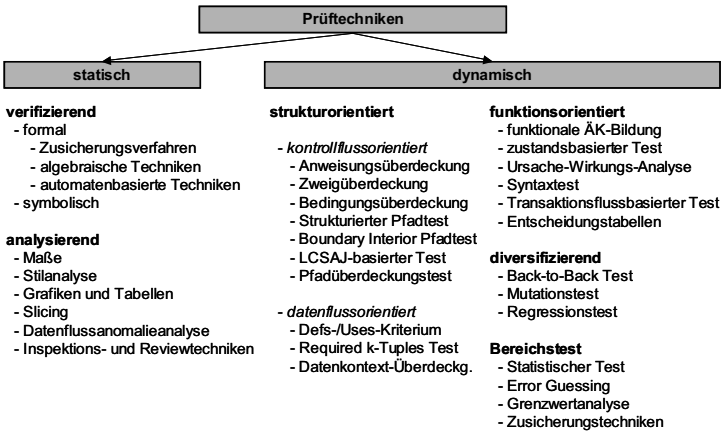


Abb. 6.2: Klassifikation von Software-Prüftechniken nach (Liggesmeyer, 2002)

Dieses Klassifikationsschema unterteilt in zwei Klassen, die statischen und die dynamischen Techniken. Sie unterscheiden sich dadurch, ob das zu überprüfende Programm zur Ausführung

kommen muss (dynamisch) oder nicht (statisch). Die meisten der dynamischen Prüftechniken besitzen eine hohe Praxisrelevanz. Besonders wichtig sind hierbei die funktionsorientierten sowie einige der kontrollflussorientierten Techniken. (Liggesmeyer, 2002).

6.3 Zuverlässigkeit eingebetteter Systeme

Neben der Echtzeitfähigkeit gilt die *Verlässlichkeit (bzw. Zuverlässigkeit oder Fehlertoleranz)* als die wichtigste nicht-funktionale Eigenschaft eingebetteter Systeme. Der Mensch versucht tendenziell, Sicherheit und Zuverlässigkeit durch Automatisierung zu erhöhen, obgleich die Statistik hier ein anderes Bild zeichnet (Schürmann, 2001): Etwa viermal so viele Unfälle passieren, wenn der Mensch *nicht* mehr in die Steuerung eingreifen kann.

Beispiel Flugzeuge

Unternehmen der Flugzeugindustrie (EADS, Honeywell, Boing) und US-amerikanische Behörden (NASA, DoD) planen derzeit die Entwicklung eines Notsystems, das verhindern soll, dass Verkehrsflugzeuge absichtlich oder unabsichtlich an Hindernissen zerschellen: Der Autopilot würde anstelle des Piloten in Gefahrensituation die Steuerung vollständig übernehmen, ohne dass dieser eingreifen könnte (Spiegel, 2003). Der Einsatz ist beispielsweise im Airbus A380 geplant, welcher das bis dato größte Verkehrsflugzeug werden soll. Ein derart autarkes System, das in Notsituationen zuverlässig das Steuer von Passagierjets übernimmt und sie aus dem Gefahrenbereich steuert, gilt als sehr erfolgsversprechend, denn auch ohne Bedrohung durch Terrorismus liegen nach wie vor den meisten Abstürzen Navigationsfehlern zugrunde.

Obwohl sich die Testergebnisse erster Entwicklungen als positiv erwiesen haben, gibt es sowohl unter Fluggästen, Flugsicherheitsbehörden als auch Piloten eine weit verbreitete Skepsis, die Kontrolle über das Flugzeug in Krisensituationen vollständig einem Computersystem (einem eingebetteten System) zu überlassen. Sie fordern die Möglichkeit, dass Crewmitglieder das System ausschalten können, womit es freilich zur Vermeidung von Terrorakten unverwertbar wäre.

Auch wenn ca. 80% aller Flugzeugkatastrophen durch menschliches Versagen verursacht werden (Schürmann, 2001), gibt es keine Statistik, die besagt, wie oft der Pilot ein Versagen der Technik korrigiert bzw. beherrscht.

Technisch gesehen wäre die Einführung dieses Sicherheitssystems ein eher marginaler technologischer Entwicklungsschritt. Bereits heute ist der Einsatz von Autopiloten bei Start, Navigation und Landung unter minimaler menschlicher Beteiligung Routine. Die neue Technologie würde dies weiterführen.

Mensch-Maschine-Kooperation

Besser wäre möglicherweise der Einsatz eines *Mensch-Maschine-Teams,* bei dem jeder das macht, was er besser kann: Eingebettete Computersysteme regeln die technische Umgebung in Standardsituationen und der menschliche Benutzer ist für die Behandlung von Ausnahmen und unvorhersehbaren Situationen zuständig. Natürlich begehen Menschen ständig Fehler, machen aber keine Aufgabe zweimal identisch. Eine Maschine dagegen begeht zwar seltener Fehler, ist aber im Fehlerfall nicht in der Lage, zu korrigieren. Menschliche Fehler können durch Plausibilitätsprüfungen erkannt werden, Maschinenfehler dagegen durch den sprichwörtlichen "gesunden Menschenverstand". Durch den Einsatz jeder Sicherheitsmaßnahme kann sich neues Gefahrenpotential ergeben, so beispielsweise bei einem Bahnübergang: Zwar erhöht der Einsatz von Schranken am Bahnübergang die Sicherheit, aber ohne Schranken besteht nicht die Gefahr, dass ein Fahrzeug zwischen den Schranken liegen bleibt. Darüber hinaus mag man bei beschrankten Bahnübergängen eher als bei unbeschrankten dazu neigen, diesen unachtsam zu überqueren, weil man sich auf das System verlässt. Wir wollen im Folgenden den Terminus der Zuverlässigkeit und verwandte Begriffe analysieren.

Definition (Zuverlässigkeit)

Definition (Zuverlässigkeit):
Zuverlässigkeit (engl. reliability) ist die Wahrscheinlichkeit, dass ein System seine definierte Funktion innerhalb eines vorgegebenen Zeitraums und unter den erwarteten Arbeitsbedingungen voll erfüllt, das heißt intakt ist und es zu keinem Systemausfall kommt.

Schließlich ist neben der Zuverlässigkeit auch die *Verfügbarkeit* ein wichtiges Gütemaß. Vom Begriff der Zuverlässigkeit lässt sich die Verfügbarkeit wie folgt abgrenzen:

Definition (Verfügbarkeit)

Definition (Verfügbarkeit):
Die Verfügbarkeit (engl. availability) eines Systems ist der Zeitraum gemessen am Anteil der Gesamtbetriebszeit des Systems, in dem es für den beabsichtigten Zweck eingesetzt werden kann.

Definition (Systemausfall)

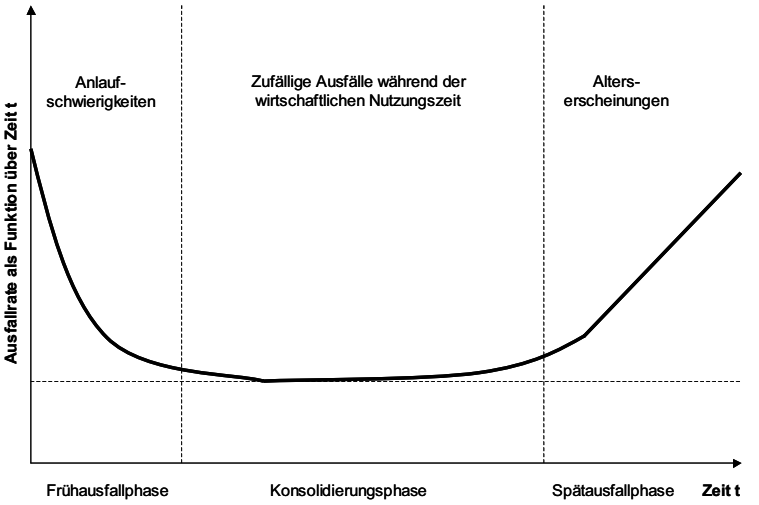
Definition (Systemausfall):
Ein Systemausfall (kurz: Ausfall, engl. failure) liegt vor, wenn ein System seine geforderte Funktion nicht mehr erfüllt.

Mit Ausfall ist hier zur Vereinfachung im Folgenden stets der Totalausfall des Systems gemeint. Wir betrachten keine Zwischenstufen der Funktionsfähigkeit (sogenannte *Änderungsausfälle*). Ein Beispiel hierfür wäre bei einem Fernseher der Tonausfall obwohl die Bildübertragung weiterhin einwandfrei funktioniert (Fränzle, 2002).

Ausfallrate, Badewannenkurve

Betrachtet man die *Ausfallsrate* (= Maß für die Anzahl von Ausfällen pro Zeiteinheit) einer elektronischen Komponente entlang ihrer zeitlichen Nutzung, so ergibt sich die sogenannte *„Badewannenkurve“,* deren Verlauf an den Querschnitt einer Badewanne erinnert; vgl. Abbildung 6.3. Ihr Name rührt daher, dass die Ausfallwahrscheinlichkeit einer Komponente am Anfang ihrer Nutzung sehr hoch ist, dann stark abfällt, um schließlich gegen Ende der Nutzungsdauer wieder stark anzusteigen.

Abb. 6.3: Badewannenkurve



Definition (Risiko)

Definition (Risiko):
Ein Risiko ist das Produkt der zu erwartenden Eintrittshäufigkeit (Wahrscheinlichkeit) eines zum Schaden führenden Ereignisses und des bei Eintritt des Ereignisses zu erwartenden Schadensausmaßes.

Definition (Grenzrisiko):
Mit Grenzrisiko bezeichnen wir das größte noch vertretbare Risiko.

Definition (Grenzrisiko)

Das Grenzrisiko ist auf rein quantitativer Basis ein schwer erfassbarer Wert. Die Anforderungsdefinition eines sicherheitskritischen eingebetteten Systems enthält im Regelfall neben Sicherheitsanforderungen sowie funktionalen und nicht-funktionalen Anforderungen auch Anforderungen bezüglich der Systemzuverlässigkeit.

So schreibt beispielsweise die oberste Luftfahrtbehörde der Vereinigten Staaten, die FAA (Federal Aviation Administration) für die Zertifizierung eines neuen Flugzeugtyps vor, dass die Wahrscheinlichkeit für eine sogenannte „catastrophic failure condition" kleiner als 10^{-9} pro Flugstunde ist, d. h. in einer Milliarde Stunden darf höchstens ein solcher Ausfall bezogen auf sämtliche Flugzeuge dieses Typs auftreten (Fränzle, 2002).

Analyse, Synthese

Derartige Zuverlässigkeitsanforderungen bzw. Ausfallraten können nicht empirisch durch Testen nachgewiesen werden. Folglich benötigt man Entwurfs- und Analysemethoden, die sicherstellen, dass ein eingebettetes System die Anforderungen an seine Zuverlässigkeit erfüllt. Genauer gesagt braucht es zum einen Methoden, um die Zuverlässigkeit konstruktiv in das System mit einzubauen *(Synthese),* sowie zum anderen Analyse-Methoden, welche die Zuverlässigkeit eines Systems aus der Zuverlässigkeit seiner einzelnen Komponenten schließen können *(Analyse).* Die Anforderungen an die Zuverlässigkeit, d. h. die zu erzielende Ausfallrate, veranlasst den Designer, bestimmte konstruktive Maßnahmen zu ergreifen und eine Systemarchitektur zu wählen, die diese Anforderungen gewährleistet. Während des weiteren Entwurfs und der Verfeinerung des Systems werden Analyseverfahren eingesetzt, die dann bestätigen, dass der vorgeschlagene Systementwurf den Zuverlässigkeitsanforderungen tatsächlich genügt.

Reliability Engineering

Beide Techniken sind Teil des sogenannten *Reliability Engineering,* das darauf abzielt, die Zuverlässigkeit eines Systems zu erhöhen, die Wahrscheinlichkeit eines Ausfalls zu minimieren und damit insgesamt die Sicherheit des Systems zu verbessern.

6.3.1 Konstruktive Maßnahmen

Fehlertoleranz, Redundanz

Die Zuverlässigkeit eines Systems wird durch Fehler (engl. faults) eingeschränkt. Es stellt sich nun die Frage, welche Maßnahmen man konstruktiv ergreifen kann, um mit diesen tatsächlich auftretenden Fehlern fertig zu werden. Dies geschieht durch die fehlertolerante Auslegung des Systems. *Fehlertoleranz* basiert immer auf einer Form von *Redundanz,* d. h. man macht das System komplexer als es bei Abwesenheit von Fehlern erforderlich wäre.

Unterschiedliche Formen von Redundanz sind die Verwendung zusätzlicher Hardware oder Software mit dem Ziel, Fehler zu erkennen oder zu tolerieren oder der Gebrauch zusätzlicher Information (Beispiele: Paritätsbits, Prüfsummen, fehlererkennende und/oder -korrigierende Codes).

Früher basierten die meisten Ansätze zur Fehlertoleranz auf dem Einsatz redundanter Hardware. Heutzutage werden dagegen oftmals vielfältige Mischformen eingesetzt, um optimale Fehlertoleranz zu erreichen (Fränzle, 2002).

6.3.1.1 Einsatz redundanter Hardware

Ansätze zur Fehlertoleranz

Fehlertoleranz durch redundante Hardware kann durch folgende unterschiedliche Ansätze erreicht werden:

- **Statische Redundanz**, die auf „voting“ (abstimmen) basiert: Hier werden identische Hardwarekomponenten parallel geschaltet und deren Berechnungsergebnisse von einer Voting-Komponente verglichen und anschließend ein Mehrheitsentscheid durchgeführt. Das Voting kann in einer oder mehreren Stufen hintereinander durchgeführt worden.
- **Dynamische Redundanz**, die auf Fehlererkennung und anschließender Rekonfiguration des Systems basiert. Wird ein Fehler erkannt, so schaltet das System auf eine Reservekomponente, beispielsweise den Standby Datenbankserver.
- Schließlich gibt es noch eine **hybride Methode**, welche die beiden vorgenannten Verfahren kombiniert. Dies funktioniert im Wesentlichen wie folgt: Mehrere identische und aktive Systemkomponenten sind über einen Umschalter mit einem Voter verbunden. Nur wenn eine Komponente ausfällt, maskiert der Voter diesen Fehler mithilfe eines Mehrheitsentscheids. Die feh-

lerhafte Komponente wird erkannt und durch Umschalten auf eine der Reservekomponenten ersetzt.

6.3.1.2 Einsatz redundanter Software

Ariane 5

Anders als bei der Hardware-Redundanz ist der mehrmalige Einsatz identischer (baugleicher) Software nicht zielführend. Ein Negativ-Beispiel hierfür hat der Jungfernflug der europäischen Trägerrakete Ariane 5 am 4. Juni 1996 geliefert, dessen Ursachen an dieser Stelle weiter analysiert werden sollen. Wie bereits zu Beginn dieses Kapitels erläutert, musste 37 Sekunden nach ihrem Start die Rakete ferngelenkt zerstört werden, weil sie die vorgeschriebene Flugbahn verlassen hatte und zudem damit zu rechnen war, dass sie in bewohntes Gebiet stürzen könnte. Die Ursache für die Fehlberechnung des Kurses lag daran, dass Software, die vom Vorgängermodell Ariane 4 ohne weitere Überprüfung übernommen wurde, einen Überlauf produzierte – kurioserweise ausgerechnet in einem Softwareteil, der für die Steuerung der Ariane 5 gar nicht nötig gewesen wäre. Obwohl diese Software doppelt vorhanden war, wurde natürlich zweimal der selbe Überlauf produziert.

Das einfache Replizieren von Software (wie bei Hardware) bringt also keinerlei Vorteile. Redundante Software muss demnach mehrfach parallel entwickelt werden. Da dies zu sehr hohen Kosten führt, ist ihr Einsatz hoch sicherheitskritischen Systemen vorbehalten. Folgende Ansätze können unterschieden werden:

Ansätze zur Redundanz

- **Statische Redundanz** (N-Versions Programming): Mehrere Entwicklerteams erstellen verschiedene Implementierungen eines Programms, die auf einem bzw. mehreren Mikroprozessoren nebenläufig (Zeitscheibenverfahren bzw. echte Parallelität) ablaufen. Es folgt ein Vergleich der Ergebnisse sowie ein Mehrheitsentscheid durch einen Voter. Aufgrund der hohen Implementierungskosten sowie ggf. der Verlangsamung der Ausführungszeit ist dieser Ansatz nur bei hochkritischen Systemen, wie z. B. dem Primary Flight Control System des Airbus A330/340 geeignet (Fränzle, 2002).
- **Dynamische Redundanz** (Recovery Blocks): Hier wird eine permanente Fehlererkennung verwendet, um die Funktionsfähigkeit einer Softwarekomponente während des Betriebs zu überprüfen. Wird ein Fehler erkannt, wird auf die Reservekomponente (alternative zweite Implementierung) umgeschaltet.

6.3.2 Analytische Verfahren

Bisher haben wir diskutiert, wie man durch Ausnutzung von Redundanz zuverlässige Gesamtsysteme aus möglicherweise weniger zuverlässigen Systemkomponenten *konstruieren* kann. Im Folgenden wollen wir kurz darauf eingehen, wie man die Zuverlässigkeit von Systemen *analysieren* kann. Unser Ziel wird dabei sein, die Zuverlässigkeit des Systems aus der Zuverlässigkeit seiner Komponenten zu folgern. Dies setzt natürlich voraus, dass wir die Zuverlässigkeit dieser Komponenten kennen.

Zuverlässigkeit

Wie wir bereits gesehen haben, kann die Zuverlässigkeit einer Komponente als Wahrscheinlichkeitswert zwischen 0 und 1 quantifiziert werden. Diese variiert aber in der Regel über den Verlauf der Zeit, wie wir bereits bei der „Badewannenkurve“ im Zusammenhang mit der Ausfallrate gesehen haben.

Die Badewannenkurve wird zur Beschreibung von Früh-, Zufalls- und Verschleißausfällen verwendet. Um den Funktionsverlauf möglichst ideal an die praktischen Ausfälle anpassen zu können, wird eine 3-parametrige Weibull-Verteilung verwendet. Der dritte Parameter, der zur Weibull-Verteilung eingeführt wird, ist ein Lageparameter, mit dem es möglich wird, die Gesamtverteilung aus einzelnen verschobenen Verteilungen zusammenzusetzen.

Kompositionale Zuverlässigkeit

Um unser Ziel verfolgen zu können, also die Zuverlässigkeit eines Gesamtsystems aus der Zuverlässigkeit seiner Komponenten zu berechnen, nehmen wir im Folgenden an, die Zuverlässigkeitswerte für alle Komponenten seien uns bekannt. Wir gehen bei der Berechnung der Gesamtzuverlässigkeit schrittweise vor, indem wir *induktiv* aus einfachen Systemen, angefangen bei Einzelkomponenten, komplexere bauen.

Dieses Zusammenbauen von Komponenten ist vergleichsweise einfach, da es nur die parallele sowie die serielle Kopplung von Komponenten gibt (hierbei bezeichnet $R_i(t)$ mit $0<R_i(t)<1$ die Zuverlässigkeit (engl. reliability) der i-ten Komponente zum Zeitpunkt t und R(t) mit $0<R(t)<1$ die Zuverlässigkeit des Gesamtsystems zum Zeitpunkt t):

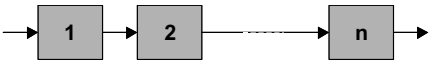
- **Serielle Kopplung** (vgl. Abbildung 6.4(a)): Das Gesamtsystem ist nur dann intakt, wenn *alle* in Serie geschalteten Komponenten intakt sind. Der Ausfall bereits einer beliebigen einzelnen Komponente bewirkt den Ausfall des Gesamtsystems. Die Zuverlässigkeit des Gesamtssystems errechnet sich als Produkt der Zuverlässigkeiten aller Systemkomponenten:

$$R(t) = R_1(t) * ... * R_n(t)$$

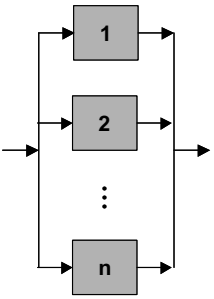
- **Parallele Kopplung** (vgl. Abbildung 6.4(b)): Das Gesamtsystem ist intakt, wenn mindestens eine ihrer parallel gekoppelten Komponenten intakt ist. Die Zuverlässigkeit des Gesamtsystems ist hier etwas komplizierter zu bestimmen. Sie ergibt sich als Produkt der Zuverlässigkeiten aller Systemkomponenten:

$$R(t) = 1-[(1-R_1(t)) * ...* (1-R_n(t))]$$

Das Gesamtsystem ist also nur dann defekt, wenn alle Komponenten defekt sind; die Wahrscheinlichkeit für einen Defekt ist eins minus den Wert für die Zuverlässigkeit.



(a) Serielle Kopplung



(b) Parallele Kopplung

Abb. 6.4: (a) serielle und (b) parallele Kopplung von Komponenten

Bei den Berechnungen haben wir dabei die *Annahme* getroffen, alle in Serie bzw. parallel geschalteten Komponenten seinen unabhängig. Dies erlaubt uns die Betrachtung ihrer Zuverlässigkeiten als *stochastisch unabhängige* Wahrscheinlichkeiten als Grundlage obiger Multiplikationen.

Die Berechnung der *Zuverlässigkeit eines komplexeren Systems* kann immer auf eine Kombination von serieller und paralleler Kopplung zurückgeführt werden. Dazu werden induktiv parallele und serielle Teilsysteme zu jeweils einem „neuen“ Block zusammengefasst.

6.3.3 Stochastische Abhängigkeit

Die vorgenannten Zuverlässigkeitsmodelle gehen davon aus, dass die Ausfallereignisse der einzelnen Systemkomponenten sowohl stochastisch unabhängig als auch unabhängig von der Situation sind bzw. sich zumindest hinreichend genau durch ein solches Modell abstrahieren lassen. Dies ist in der Praxis allerdings häufig nicht der Fall. Zum Beispiel ist die Ausfallwahrscheinlichkeit eines Cold Standby Datenbankservers (ein Ersatzsystem für den eigentlichen Datenbankserver) in der Standby-Phase extrem gering, im Aktivierungsfall, welcher ja gerade bei Ausfall des Primärsystems eintritt, nimmt sie aber signifikant zu.

Markov-Ketten

Um derartige Systeme adäquat modellieren zu können, benötigt man zustandsbasierte Modelle, die eine situationsabhängige Variation der Ereigniswahrscheinlichkeiten darstellen können. Eines der wohl bekanntesten derartigen stochastischen Modelle ist das der *Markov-Kette* bzw. des diskreten *Markov-Prozesses* (Krengel, 1991).

6.3.4 Gefahrenanalyse

Die Zuverlässigkeit eines Gesamtsystems ergibt sich nicht immer unmittelbar aus der seiner Komponentenstruktur. Für die Analyse der Zuverlässigkeit werden deshalb in der Regel auch systematische Suchverfahren angewandt, die den Zusammenhang zwischen Komponentenfehlern und Fehlfunktionen des Gesamtsystems aufzudecken vermögen. Diese *analytischen Suchverfahren* werden im Allgemeinen unter dem Terminus *Gefahrenanalyse* zusammengefasst.

Definition (Gefahr)

Definition (Gefahr):
Als Gefahr (engl. hazard) bezeichnet man dabei eine Sachlage oder Situation bzw. einen Systemzustand, in der bzw. in dem eine Schädigung der Umgebung (Umwelt, Mensch, Maschine) möglich ist.

Eine Gefahrensituation ist also eine Situation, in der das Risiko größer als das Grenzrisiko ist. Gefahren lassen sich im Prinzip auf die ihnen ursächlich zu Grunde liegenden Fehler (engl. faults), die ihrer Art nach zufällig, beispielsweise die Alterung einer

Komponente, oder systematisch, also in das System von vorne herein hineinkonstruiert, sein können, zurückverfolgen.

Definition (Unfall)

Definition (Unfall):
Tritt eine Schädigung dann tatsächlich ein, so bezeichnet man dieses Ereignis als Unfall (engl. accident).

Rückwärts-, Vorwärtsanalyse

Systematisierte Suchverfahren zur Gefahrenanalyse könnten prinzipiell an einer beliebigen Stelle im Ursache-Wirkungsgeflecht, das Fehler mit Gefahren verbindet, ansetzen. In der Praxis haben sich allerdings solche Verfahren durchgesetzt (Fränzle, 2002), die an einem der beiden Endpunkte ansetzen, die also entweder von den möglichen Gefahren ausgehend rückwärts *(Rückwärtsanalyse)* nach der zur jeweiligen Gefahr führenden Fehlerkombinationen suchen oder von möglichen Fehlern oder Fehlerkombinationen ausgehend vorwärts *(Vorwärtsanalyse)* die möglichen Gefahren bestimmen.

Ereignisbaum-analyse

Ein Beispiel der Vorwärtsanalyse ist die *Ereignisbaumanalyse,* ein Beispiel für die Rückwärtsanalyse (deduktiver Ansatz) die Fehlerbaumanalyse (engl. Fault Tree Analysis, kurz: FTA). Sie liefert eine graphische Übersicht möglicher Ursachen und auslösender Ereignisse für einen bestimmten Fehler im System. Die Systemmodellierung geschieht durch den Fehlerbaum, dessen Knoten Symbole für Ereignisse und deren logische Verknüpfungen sind.

FMEA

Ein weiterer, induktiver und systematischer Ansatz ist die *Failure Mode and Effect Analysis (FMEA)* zur Aufdeckung einzelner Fehler auf Komponentenebene und zur Ausarbeitung von Gegenmaßnahmen. Für jede Systemkomponente werden hierbei folgende Fragestellungen untersucht:

- Welche Fehler(-ursachen) können auftreten?
- Welche Folgen haben diese Fehler?
- Wie können diese Fehler vermieden bzw. das Risiko minimiert werden?

Ziele der FMEA

Die Fehlerliste führt dann zu einer Systemüberarbeitung, die wiederum eine neue Analyse notwendig macht; insgesamt ergibt sich ein iterativer Analyseprozess. Die FMEA hat folgende Ziele:

- Kein Fehler darf einen negativen Einfluss (auf redundante Systemteile) haben.
- Kein Fehler darf die Abschaltung der Stromversorgung eines defekten Systemteils verhindern.

- Kein Fehler darf in kritischen Echtzeitfunktionen auftreten.

Ausblick (Biologie):
Beim Bau fehlertoleranter eingebetteter Systeme kann man sicherlich von der Natur, beispielsweise dem menschlichen Körper einiges lernen. Man denke beispielsweise an das Herz, bei dem bei Ausfall des Sinusknotens (70 bis 80 Hz) der Atrioventrikularknoten (40 bis 60 Hz) dessen Funktion mit verminderter Leistungsfähigkeit übernehmen kann. Die Komponente Herz ist weiterhin verfügbar.

Ein anderes Beispiel aus der Biologie ist die DNA-Reparatur mit Fehlertoleranz: Marburger Max-Planck-Forscher haben entdeckt, wie Zellen nicht nur die Effizienz, sondern auch die Genauigkeit von DNA-Reparaturen steuern können (Stelter und Ulrich, 2003). Für die Verdopplung des menschlichen Erbgutes sind spezielle Enzyme verantwortlich, die sogenannten DNA-Polymerasen. Ihre Kopiergenauigkeit trägt entscheidend zur exakten Weitergabe der genetischen Information einer Zelle bei. Beschädigungen der DNA blockieren allerdings diese Enzyme und würden die Zellteilung verhindern, wenn die Zelle nicht über eine Reihe anderer DNA-Polymerasen verfügen würde, die auf die Überwindung derartiger Blockaden spezialisiert sind. Diese gewissermaßen „Notfall-Spezialisten“ sind jedoch wegen ihrer Toleranz gegenüber beschädigter DNA weniger genau und können dadurch unerwünschte, im schlimmsten Falle krebsauslösende Mutationen verursachen. Jüngst wurde ein Signalweg entdeckt, der die Aktivität der Notfall-Polymerasen reguliert und so die Genauigkeit der Erbgutverdopplung mitbestimmt. Die Marburger Max-Planck-Wissenschaftler hoffen so, mit ihren Forschungsergebnissen einen Weg gefunden zu haben, der in Zellen die Entstehung unerwünschter Mutationen verhindert, ohne dass dadurch wichtige fehlerfreie Reparaturvorgänge beeinträchtigt werden. Das könnte ein wichtiger neuer Schritt zur Bekämpfung der Krebsentstehung sein.

6.4 Sicherheit eingebetteter Systeme

Wie ist nun der Zusammenhang zwischen *Zuverlässigkeit* (engl. reliability) und *Sicherheit* (engl. safety)? Ein Systemausfall ist oftmals die Ursache eines Unfalls und damit einer Unzuverlässigkeit. Der Begriff der Sicherheit ist jedoch kein Synonym für Zuverlässigkeit, sondern weiter gefasst als die Zuverlässigkeit. Die internationale Norm *ISO 8402* (ISO = International Standardization

Organization) definiert „safety“ wie folgt: „State in which the risk of harm (to persons) or damage is limited to an acceptable level“.

Verifikation, Korrektheit

Die Sicherheit eines Systems ist eine Eigenschaft, die eng daran gekoppelt ist, ob die Funktionsweise des implementierten Systems mit dessen *Spezifikation* tatsächlich übereinstimmt. Man spricht dann auch von der *Korrektheit* des Systems bezüglich einer Eigenschaft. Mathematische Verfahren und Techniken, die dies sicherstellen, fasst man unter dem Begriff der *Verifikation* (semiautomatisches Theorembeweisen, vollautomatisches Model Checking) zusammen.

Ein sicheres System ist nicht zwangsläufig auch ein zuverlässiges. Viele Unfälle treten auf, obwohl alle Systemkomponenten gemäß ihrer Spezifikation arbeiten. Umgekehrt muss der Ausfall einer Komponente nicht unbedingt die Sicherheit beeinträchtigen oder gar zu einem Unfall führen. Fällt beispielsweise die Zentralverriegelung eines Fahrzeugs aus, so ist dies im Normalbetrieb völlig unbedenklich. Anders ist dies bei einem Unfall, bei dem die Zentralverriegelung automatisch alle Türen entriegeln sollte, damit eine reibungslose Versorgung der Insassen möglich ist.

Als Folgerung kann man schließen, dass die Zuverlässigkeit in der Regel eine notwendige, jedoch keine hinreichende Bedingung für die Sicherheit ist. Eine mangelhafte Zuverlässigkeit ist nur für jene Gefahren und Unfälle verantwortlich, die durch Systemversagen verursacht werden, nicht aber für solche Gefahren, die von dem System bei korrekter Funktionsweise ausgehen.

Validierung

Von der Verifikation grenzt sich der Begriff der *Validierung* ab (siehe auch Abbildung 6.1). Letztere führt den Nachweis, ob das (spezifizierte) System mit den Anforderungen des Auftraggebers bzw. Benutzers einhergeht, wohingegen die Verifikation im Entwicklungsprozess später ansetzt und überprüft, ob das implementierte System mit der Systemspezifikation übereinstimmt und ob die Systemspezifikation bestimmte Eigenschaften erfüllt. Validierung ist also die Antwort auf die Frage, ob das *richtige System* (also das von Auftraggeber gewollte) gebaut wird. Dagegen gibt die Verifikation eine Antwort auf die Frage, ob das *System richtig* (also korrekt) konstruiert wird.

Vorsicht ist darüber hinaus insofern geboten, da der Begriff der Sicherheit, wie wir ihn hier verwenden mit Datenschutz, Kryptographie usw. nichts zu tun hat. Im Englischen spräche man in diesem Fall von „*security*“.

6.4.1 Testen

Für die Entwicklung von (eingebetteter) Software ist, wie wir noch in Kapitel 7 genauer sehen werden, eine systematische Vorgehensweise erforderlich. Dabei folgt man Methoden, die grundlegende Arbeitsschritte wie z. B. Analyse, Entwurf, Konstruktion, Testen und Abnahme beinhalten. Da Fehler in eingebetteten Systemen generell sehr kostspielig sind, wachsen auch die Anforderungen an die Softwarequalität. Um nun die Qualität der Software zu steigern und etwaige Folgekosten zu vermeiden, muss sie ausgiebig getestet werden. Da der Testaufwand im Gegensatz zur Codierung meistens höher liegt und somit die Entwicklungskosten nach oben treibt, sind Testverfahren, -sprachen, -methoden und -werkzeuge zu verwenden, die dem entgegenwirken können.

6.4.1.1 Überblick

Wie wir bereits festgestellt haben, löst das Testen von Software Laufzeitfehler (failures) aus und führt zur Entdeckung von Programmfehlern (faults). Dieser Prozess der Fehlerlokalisierung muss der Fehlerkorrektur stets vorausgehen. Bekannt ist zunächst lediglich die Fehlerwirkung (failure), nicht aber die genaue Stelle (fault) in der Software, die zu dem Fehler führt.

Debugging

Das Lokalisieren und Beheben von Fehlern wird oft auch als *Debugging* bezeichnet. Debugging und Testen werden – fälschlicherweise – oft gleichgesetzt; es handelt sich hier allerdings um zwei völlig unterschiedliche und getrennt zu betrachtende Aufgaben (Spillner und Linz, 2003). Während das Debugging das Ziel hat, Defekte bzw. Fehlerzustände zu beheben, ist es die Aufgabe des Testens, Fehlerwirkungen *gezielt und systematisch* aufzudecken.

Testen, Test, Testobjekt

Unter dem *Testen* von Software wird jede (im Allgemeinen stichprobenartige) Ausführung eines *Testobjekts*, also beispielsweise eines Programms oder dessen Komponenten verstanden, die seiner Überprüfung dient. Die Randbedingungen für die Ausführung des Tests müssen ebenfalls festgelegt sein. Ein Vergleich zwischen dem Sollverhalten (definiert durch die Spezifikation des Testobjekts) und dem Ist-Verhalten dient zur Bestimmung, ob das Testobjekt die (in seiner Spezifikation) geforderten Eigenschaften erfüllt. Oft wird der gesamte Prozess, ein Testobjekt auf systematische Weise auszuführen, um die korrekte Umsetzung der Anforderungen nachzuweisen, als *Test* bezeichnet.

Zum Testprozess gehören neben der Aktivität des Ausführens des Testobjekts mit *Testdaten* auch die Planung, Durchführung und das Auswerten der Tests. Man spricht hier auch vom *Testmanagement.* Ein *Testlauf* umfasst die Ausführung eines oder mehrerer Testfälle. Zu einem *Testfall* gehören die festgelegten Randbedingungen; meist handelt es sich hier um die Voraussetzungen zur Ausführung, die Eingabewerte und die erwarteten Ausgaben bzw. das erwartete Verhalten des Testobjekts. Ein Testfall sollte so gewählt werden, dass er in der Lage ist, eine bisher nicht bekannte Fehlerwirkung mit relativ hoher Wahrscheinlichkeit aufzudecken. Meistens werden mehrere Testfälle zu sogenannten *Testszenarien* zusammengefasst. Hierbei kann das Ergebnis eines Testfalls als Ausgangssituation für den darauf folgenden Testfall verwendet werden. Alle Testfälle werden dann im Zusammenspiel sukzessive in einem Testlauf zur Ausführung gebracht.

Testdaten, Testlauf, Testfall, Testmanagement, Testszenarien

Der Prüfer hat alle Testfälle zu ermitteln und zu organisieren. Danach kann er daraus seine Daten generieren und die Sollergebnisse bestimmen. Ein extra für den Test erstellter Ablaufplan hilft bei der Durchführung, um evtl. ausgelassene oder mehrmals wiederholte Testfälle zu verhindern. Alle zum Programm gehörenden Komponenten wie z. B. die Dokumentationen, verschiedenste Konfigurationsversionen, Installationsroutinen usw. sollten mitgetestet werden. Die Ergebnisse sind begleitend in einer Testdokumentation festzuhalten, um sie am Ende des Tests auswerten zu können.

TTCN-3 (Testing and Test Control Notation 3) ist die einzige standardisierte Spezifikationssprache für Tests (Quelle: Presseinformation TestingTech, August 2003, vgl. www.testingtech.de). Sie ermöglicht die Spezifikation und Implementierung von Testfällen sowie deren Ausführungsreihenfolge. Ferner unterstützt es verteiltes und funktionales Testen. Typische Anwendungsgebiete von TTCN-3 sind z. B. der Modul Test, der Internet Protokoll Test, der API Test u.v.m. Mit der Variante *TimedTTCN-3* wurde eine flexible Erweiterung von TTCN-3 zum Testen von Realtime Anforderungen realisiert.

TTCN-3

6.4.1.2 Ausgewählte Testverfahren

Im Folgenden wollen wir exemplarisch einige bekannte Testverfahren erläutern:

Black-Box Test, Funktionstest

Funktionstest (Black-Box Test):
Mit dem Funktionstest sollen Umstände entdeckt werden, bei denen sich der Prüfgegenstand nicht gemäß den Anforderungen bzw. Spezifikationen verhält. Die Testfälle werden hierbei aus den Anforderungen bzw. Spezifikationen abgeleitet. Der Prüfgegenstand wird als schwarzer Kasten (engl. black box) angesehen, d. h. der Prüfer ist nicht an der internen Struktur und dem Verhalten des Prüfgegenstandes interessiert. Die folgenden Black-Box Testfallentwurfsmethoden lassen sich unterscheiden:

- **Äquivalenzklassenbildung:** Ziel ist es, durch die Bildung von Äquivalenzklassen eine hohe Fehlerentdeckungswahrscheinlichkeit mit einer minimalen Anzahl von Testfällen zu erreichen. Dabei werden die gesamten Eingabedaten eines Programms in eine endliche Anzahl von Äquivalenzklassen unterteilt, so dass man annehmen kann, dass mit jedem beliebigen Repräsentanten einer Klasse die gleichen Fehler wie mit jedem anderen Repräsentanten dieser Klasse gefunden werden.
- **Grenzwertanalyse:** Hierbei wird versucht, Testfälle zu definieren, mit denen Fehler im Zusammenhang mit der Behandlung der Grenzen von Wertebereichen aufgedeckt werden können. Die Aufgabe der Grenzwertanalyse besteht nun darin, die Grenzen von Wertebereichen bei der Definition von Testfällen zu berücksichtigen. Ausgangspunkt sind die mittels Äquivalenzklassenbildung ermittelten Äquivalenzklassen. Im Unterschied zur Äquivalenzklassenbildung wird kein beliebiger Repräsentant der Klasse als Testfall ausgewählt, sondern Repräsentanten an den Grenzen der Klassen. Die Grenzwertanalyse stellt somit eine Ergänzung des Testfallentwurfs gemäß Äquivalenzklassenbildung dar.
- **Intuitive Testfallermittlung:** Bei der intuitiven Testfallermittlung wird versucht, die systematisch ermittelten Testfälle qualitativ zu verbessern indem ergänzende Testfälle mit eingebracht werden. Diese zusätzlichen Testfälle werden vom Prüfer aufgrund seiner Erfahrung mit sogenannten Standardfehlern erstellt.
- **Funktionsüberdeckung:** Hierbei gilt es Testfälle zu erstellen, mit denen nachgewiesen werden kann, dass die jeweilige Funktion vorhanden ist und auch ausgeführt werden kann. Der Prüfgegenstand soll somit sein Normalverhalten und das Ausnahmeverhalten zeigen.

Vorsicht: Alle funktionsorientierten Tests sind Black-Box Tests, jedoch nicht alle Black-Box Tests sind funktionsorientiert (z. B. der Back-to-Back Test) (Liggesmeyer, 2002). In früherer Literatur werden beide Begriffe fälschlicherweise als synonym betrachtet.

White-Box Test, Strukturtest

Strukturtest (White-Box Test):
Bei diesem Testverfahren wird die interne Struktur des Prüfgegenstandes untersucht, um aufgrund der Programmlogik und unter Berücksichtigung der Spezifikationen ablauforientierte Testfälle zu bestimmen. Beim Erstellen der Testfälle wird der angesprochene Bereich des Prüfgegenstandes betrachtet. Betrachtungsgegenstand können beispielsweise Pfade, Anweisungen, Zweige und Bedingungen sein. Die folgenden White-Box Testfallentwurfsmethoden lassen sich unterscheiden:

- **Pfadüberdeckung:** Bei der Pfadüberdeckung gilt es Testfälle zu erstellen, die eine geforderte Mindestanzahl von Pfaden (Programmabläufen) im Prüfgegenstand zur Ausführung bringen. Die Ausführung aller Pfade ist meistens aus Komplexitätsgründen nicht möglich. So würde der vollständige Pfadüberdekkungstest einer Fallunterscheidung mit n Fällen sowie umgebender Schleife mit k Iterationen $(n+1)^k + ... + (n+1)^2 + (n+1)$ unterschiedliche Pfade untersuchen müssen. Dies würde für n=4 und k=20 sowie einem Zeitaufwand von 5 Minuten pro Testfall (Spezifikation, Ausführung, Auswertung) in etwa zu einem theoretischen Zeitaufwand von 1 Milliarde Jahren führen. Es handelt sich also hier um ein rein theoretisches Kriterium, das aber als Vergleichsmaßstab für andere Testverfahren dient. Allerdings kann selbst dieses komplexe Verfahren noch nicht alle Fehler (z. B. Berechnungsfehler) entdecken, da es sich immer noch um keinen erschöpfenden Test aller möglichen Eingabewerte handelt.
- **Anweisungsüberdeckung (C0-Test):** Es werden Testfälle erstellt, die eine geforderte Mindestanzahl von Anweisungen im Prüfgegenstand zur Ausführung bringen. Hier handelt es sich um ein Minimalkriterium, da nicht mal alle Kanten des Kontrollflussgraphen betrachtet werden, und so bleiben viele Fehler unentdeckt.
- **Zweigüberdeckung (C1-Test):** Erstellt werden Testfälle, die bei jeder Entscheidung bzw. Fallunterscheidung (if-then-else) mindestens einmal den Then-Zweig sowie den Else Zweig durchlaufen. Hier handelt es sich um ein *realistisches* Minimalkriterium. Der Zweigüberdeckungstest umfasst auch den An-

weisungsüberdeckungstest. Fehler bei Schleifen oder anderer Kombination von Zweigen bleiben allerdings unentdeckt.

- **Bedingungsüberdeckung:** Unter Berücksichtigung der Spezifikation werden Bedingungen identifiziert und entsprechende Testfälle definiert. Die Testfälle werden anhand von Pfadablaufanalysen ermittelt. Jede Teilbedingung einer Kontrollflussbedingung (z. B. von If- oder While-Anweisung) muss einmal den Wert „true“ und einmal den Wert „false“ annehmen. Es werden folgende Sonderfälle unterschieden: Die atomare Bedingungsüberdeckung, die Mehrfachbedingungsüberdeckung sowie die minimale Mehrfachbedingungsüberdeckung. Auch die Bedingungsüberdeckung ist eine Obermenge der Anweisungsüberdeckung.

Als besonders effektiv hat sich die Kombination aus Funktionstest und Strukturtest erwiesen, da sich die Stärken und Schwächen beider Verfahren ausgleichen. Da die Ermittlung von Strukturtestfällen in der Regel aufwändiger ist, wird angeraten, den Funktionstest als erstes durchzuführen.

Klassifikationsbaum

Klassifikationsbaum Methode:
Mit der Klassifikationsbaum (engl. classification tree) Methode steht eine systematische und leicht erlernbare graphische Testmethode zur Verfügung, die zu redundanzarmen und fehlersensitiven Testfällen führt. Die grundsätzliche Idee der Klassifikationsbaum Methode ist es, zuerst die Menge der möglichen Eingaben für das Testobjekt getrennt auf verschiedene Weisen, unter jeweils einem geeigneten Gesichtspunkt zu zerlegen, um dann durch Kombination dieser Zerlegungen zu Testfällen zu gelangen.

Target Test

Test von Echtzeitanforderungen (Target Test):
Für diesen Test sind eine Reihe von Anforderungen durch ein Testsystem zu erfüllen. Hierzu zählt die Berücksichtigung von Echtzeitbedingungen. Um diese zu überprüfen, leitet der Prüfer Testfälle zur Feststellung der kürzesten und längsten Ausführungszeit aus dem Programmcode ab.

6.4.2 Manuelle Prüftechniken

Inspektion, Review, Walkthrough

Das manuelle Prüfen von Programmcode (also des Quelltexts eines Programms) und dessen zugehöriger Dokumente wie etwa dem Pflichtenheft sind eine in der Praxis häufig verwendete Prüftechnik. Das manuelle Prüfen existiert in zahlreichen Ausprägungen (Liggesmeyer, 2002). Man unterscheidet in der Regel zwischen

- der (formalen) Inspektion,
- dem (konventionellen) Review und
- dem (strukturierten) Walkthrough.

Die hier genannten manuellen Prüftechniken erfordern eine Prüfung des Prüfobjekts im Rahmen einer Gruppensitzung. Sie unterscheiden sich durch die Formalität des Vorgehens im Rahmen dieser Sitzung(en) und deren Aufwand. Formale Inspektionen sind ein besonders effektives, aber im Allgemeinen auch sehr zeitaufwändiges Mittel zur Lokalisation von Fehlern. Ein Walkthrough wird oft als weniger sorgfältiges Review betrachtet. Reviews in Sitzungstechnik nehmen eine mittlere Stellung ein. Sie verlangen einen geringeren Ressourceneinsatz (vor allem Arbeitszeit) als formale Inspektionstechniken.

Ein großer Vorteil manueller Prüftechniken ist, dass sie in der Lage sind, semantische Aspekte des Prüfobjekts zu beachten. Durch den Einsatz von Experten in den Sitzungen können inhaltliche Aspekte des Prüfobjekts bewertet werden und die Qualität zahlreicher unterschiedlicher Softwarequalitätsmerkmale beurteilt werden, wie etwa die Verständlichkeit, Änderbarkeit, Aussagekraft von Variablen und Kommentaren usw. Der Nachteil der manuellen Prüftechniken ist bereits durch ihre Namensgebung ausgedrückt: Derartige Tätigkeiten können nicht durch CASE-Tools (CASE = Computer Aided Software Engineering), also Softwarewerkzeuge, automatisiert werden.

Empirische Studien haben gezeigt (Möller, 1996), dass Softwarefehler, die in den frühen Phasen der Softwareentwicklung, also z. B. in der Analyse- oder Entwurfsphase, entstehen, vergleichsweise hohe Korrekturkosten verursachen. Der Aufwand für die Fehlerkorrektur wird umso höher, je größer die Distanz zwischen der Fehlerentstehung und der Fehlerentdeckung ist (Boehm 1982). Kostet das Beheben eines Fehlers während der Konzeptentwicklung der Software noch 1 Euro, so sind es in der Entwurfsphase schon 3 Euro, in der Phase der Implementierung

10 Euro und nach Abschluss aller Tests 50 Euro. Wird der Fehler schließlich erst während des Betriebs der Software gefunden, so liegt der Kostenfaktor im Vergleich zur Konzeptphase bei 150:1.

Somit erscheint es sinnvoll, gerade die Dokumente der frühen Entwicklungsphasen einer Prüfung zu unterziehen, wobei syntaktische Prüfungen durch CASE-Tools durchgeführt werden können, semantische Prüfungen jedoch eine manuelle Bewertung durch Menschen erfordern (Liggesmeyer, 2002). Hier nehmen die genannten manuellen Prüftechniken eine entscheidende Rolle ein.

6.4.3 Formale Verifikation

Es gibt mehrere Möglichkeiten wie etwa Simulation, Testen oder formale Verifikation, um die Korrektheit eines eingebetteten, reaktiven Softwareentwurfs zu überprüfen. Simulation und Testen sind die derzeit in der Praxis am weitesten verbreitete Ansätze. Die Simulation kann verwendet werden, um Spezifikationen auszuführen, das System zu analysieren und ggf. sogar dessen Verhalten zu visualisieren. Um jedoch einen möglichst hohen Anteil der Softwarefehler, beispielsweise 95 Prozent, mit Hilfe von Simulation oder Testen zu entdecken, sind sehr umfangreiche Simulations- bzw. Testverfahren erforderlich. Eine hundertprozentige Korrektheit der Spezifikation mit ihrer Hilfe nachzuweisen, ist allerdings nicht möglich. Die vollumfängliche Übereinstimmung des Softwareverhaltens mit der Anforderungsspezifikation kann nur durch formale Verifikation sichergestellt werden.

Verifikation

Verifikation im engeren Sinne bedeutet den formalen Nachweis, dass ein gegebenes (Software- oder Hardware-) System die ihm zugedachten Eigenschaften erfüllt, dass es also seine Spezifikation erfüllt. Neben eher traditionellen, auf der Zusicherung von Vor- und Nachbedingungen für Programmabschnitte basierten Methoden wie beispielsweise dem Hoare-Kalkül von C.A.R. „Tony“ Hoare wird heute vor allem das sogenannte Model Checking verwendet.

Model Checking

Model Checking wird sowohl im Bereich der Hardwareverifikation wie auch zunehmend im Bereich der Softwareverifikation verwendet. Es ist ebenfalls Logik-basiert und eignet sich besonders zur vollautomatischen Verifikation von Eigenschaften (z. B. Sicherheitseigenschaften) nebenläufiger, reaktiver Systeme. Bei der im Bereich des Model Checkings verwendeten Logik handelt es sich um Varianten der normalerweise im Rahmen eines selbst

naturwissenschaftlichen Studiums in der Regel eher selten gelehrten temporalen bzw. modalen Logik. Um Logik-basierte Verifikationssysteme verstehen und benutzen zu können, ist eine gewisse Vertrautheit mit den zu Grunde liegenden formalen Logiken und Kalkülen erforderlich, die wir im Rahmen dieses Buches jedoch nicht schaffen können. Dennoch soll im Folgenden eine kurze, pragmatische und dennoch präzise Einführung gegeben werden.

Eine sehr gute Definition von Model Checking findet sich in (Clarke und Schlinglo, 2001): „Model checking is an automatic technique for verifying correctness properties of safety-critical reactive systems". Es handelt sich um ein Entscheidungsverfahren, um zu prüfen, ob eine gegebene Kripke-Struktur ein Modell für eine angegebene temporallogische Formel ist. Für diese Prüfung ist in der Regel eine erschöpfende Durchsuchung *aller* Systemzustände erforderlich. Damit liegt die größte Herausforderung des Model Checkings auf der Hand, nämlich die automatische Verifikation einer in vielen Fällen „explodierenden" Anzahl von Zuständen. Symbolisches Model Checking sowie kompositionale Ansätze schaffen hier bis zu einem gewissen Grad Abhilfe.

Temporale Logik

Haben wir mit Hilfe einer bestimmten formalen Spezifikationstechnik, wie etwa Statecharts oder Esterel, das Verhalten eines reaktiven Systems definiert, möchten wir in der Regel einen Nachweis vitaler, also sicherheitskritischer Systemeigenschaften erbringen. Um solche Eigenschaften formal mittels Model Checking nachweisen zu können, müssen sie zunächst unter Zuhilfenahme einer geeigneten formalen Sprache spezifiziert werden. Klassische Logik ist ungeeignet, um die Dynamik veränderlicher (ggf. nicht-deterministischer) Systeme zu beschreiben.

Safety, Liveness

Hier müssen Aussagen wie etwa (A1) „irgendwann (in der Zukunft) wird an der Kreuzung die Ampel 1 grün und die Ampel 2 rot" oder (A2) „Ampel 1 und Ampel 2 sind zu keinem Zeitpunkt beide grün" getroffen werden können. Grundsätzlich wird bei temporallogischen Aussagen zwischen sogenannten *Safety-* sowie *Liveness-Eigenschaften* unterschieden. Eine Safety-Eigenschaft (engl. safety property) dient dazu, zu jedem Zeitpunkt und für jede (ggf. nicht-deterministische) Verhaltensalternative die Absenz von *unerwünschten* Zuständen zu vermeiden, vgl. Aussage (A2). Im Englischen wird dies oft prägnant durch „nothing bad will happen" zusammengefasst.

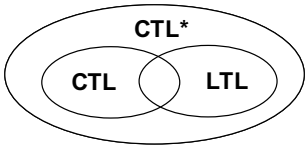
Eine Liveness-Eigenschaft dagegen stellt sicher, dass irgendwann in einem (ggf. nicht-deterministischen) Ausführungspfad des Systems ein *gewünschter* Zustand eintritt, siehe (A1). Dies fasst man im Englischen oft kurz mit „eventually, something good will happen" zusammen.

Temporale Logiken stellen hierfür adäquate Formalismen zu solchen Spezifikation dar. Eine temporale Logik stellt unterschiedliche Operatoren zur Verfügung, um Zeit auszudrücken. Es gibt unterschiedliche Arten der temporalen Logik, die sich sowohl in Semantik als auch in Ausdrucksmächtigkeit unterschieden; (Emerson, 1990) bietet diesbezüglich einen guten Überblick.

*CTL, LTL, CTL**

Eine weiteres Unterscheidungsmerkmal der temporalen Logiken ist die Modellierung der Zeit: Ein diskretes Zeitmodell ist ebenso möglich wie notwendig (für diskrete Systeme) wie ein kontinuierliches (für hybride Systeme). Die am weitesten verbreiteten diskreten temporalen Logiken sind die sogenannte verzweigende temporale Logik (engl. branching time logic) CTL, siehe (Clarke u. Emerson, 1981), die lineare temporale Logik (engl. linear time logic) LTL (Pnueli, 1977) sowie ihre gemeinsame Obermenge CTL* (Emerson u. Halpen, 1986); vgl. Abbildung 6.5. Diese Logiken besitzen eine unterschiedliche Ausdrucksmächtigkeit.

*Abb. 6.5: Klassifikation CTL, LTL, CTL**



Modell

Um mit Hilfe von Model Checking nachweisen zu können, dass eine Systemeigenschaft E von einer Spezifikation S erfüllt wird, benötigen wir ein *Modell* M(S) von S. Das Modell M(S) wird in der Regel durch eine bestimmte Art von endlichen Zustandsübergangssystemen definiert, nämlich durch sogenannte *Kripke-Strukturen* (Emerson, 1990). Sie sind wie folgt definiert:

Definition (Kripke-Struktur)

Definition (Kripke-Struktur):
Eine Kripke-Struktur ist ein Viertupel (S_0, S, R, L), wobei

- S eine endliche Menge von Zuständen (engl. states) darstellt, in der jeder Zustand eine Belegung innerhalb des zu modellierenden Systems repräsentiert;
- S_0 die Menge der Anfangszustände repräsentiert;
- $R \subseteq S \times S$ die Zustandsübergangsrelation darstellt und
- $L: S \rightarrow \wp(A)$ die Markierungsfunktion (von engl. labeling) repräsentiert, wobei A die Menge der Atome des Modells ist.

Model Checking bedeutet dann das Erbringen eines formalen Beweises, dass E in M(S) gilt, oder kurz $M(S) \models E$. Ein besonderer Vorteil des Model Checkings liegt darin, diesen Beweis vollautomatisch unter Zuhilfenahme sogenannter BDDs (Binary Decision Diagrams), also binärer Entscheidungsdiagramme (Bryant, 1986), durchführen zu können. In diesem Falle spricht man vom *symbolischen Model Checking* (McMillan, 1993).

Ein Model Checking-Werkzeug kennt somit zwei mögliche Antworten auf die Frage „$M(S) \models E$“: Ja, die Eigenschaft E gilt in M(S) oder nein, sie gilt nicht. Im letzteren Fall wird zusätzlich ein passendes Gegenbeispiel angegeben.

Theorem-beweisen

Da Model Checking nur für die Prüfung von Prüfobjekten mit einem endlichen Zustandsraum verwendet werden kann, hat sich in letzter Zeit im Bereich der formalen Verifikation sicherheitskritischer Systeme darüber hinaus auch das *interaktive Theorembeweisen* durchgesetzt. Hierbei werden die Verifikations-Aufgaben in einer möglichst ausdrucksstarken Logik formuliert (z. B. in Higher Order Logic, kurz: HOL). Ein Experte führt dann einen mathematischen Beweis der Korrektheit unter Zuhilfenahme eines interaktiven Beweissystems (z. B. „Isabelle“, siehe im Internet isabelle.in.tum.de) durch, das die vorgegebenen Schritte des Experten auf Korrektheit überprüft. „Leichte“ Teilbeweise werden dabei mit Hilfe vorgegebener Heuristiken (sogenannten Taktiken) zu lösen versucht.

6.5 Zusammenfassung

Eingebettete Systeme dringen in alle Lebensbereiche ein und übernehmen dort für den Menschen teilweise oder vollständig auch sicherheitskritische Aufgaben. Da eingebettete Systeme meist komplexe Steuerungsaufgaben in einer technischen Umgebung übernehmen, deren Defekt oder inkorrekte Funktionsweise zu ernsthaften Auswirkungen auf menschliches Leben haben kann, ist die Sicherung einer hohen Qualität dieser Systeme Pflicht.

Eine mangelhafte Softwarequalität solcher Systeme kann in einigen Anwendungsbereichen signifikante Gefährdungen hervorrufen. Beispiele hierfür wurden diskutiert. Die wesentlichen Begriffe der Softwarequalität haben wir definiert. Hierbei haben wir vor allem die beiden Begriffe der Zuverlässigkeit sowie der Sicherheit diskutiert und festgestellt, dass sich beide nicht ausschließen, aber auch nicht synonym betrachtet werden können. Obwohl die

Sicherheit hier der umfassendere Begriff ist, gibt es zuverlässige Systeme, die nicht sicher sind und umgekehrt.

Um eine möglichst hohe Softwarequalität garantieren zu können sind verschiedenste sowohl konstruktive als auch analytische Verfahren bekannt, wobei wir hier im Wesentlichen drei unterschiedliche Ansätze aus dem Bereich der letztgenannten skizziert haben.

ISEB, ASQF

Das lebenslange Lernen ist aber insbesondere im Bereich der Informationstechnologie unverzichtbar. Um im Weiterbildungsbereich für dieses Thema für bessere Vergleichbarkeit der Kursanbieter untereinander und für einen anerkannten Nachweis zu sorgen, ist in England das Information Systems Examinations Board (ISEB, vgl. www.iseb.org.uk) ins Leben gerufen worden, das unter anderem auch im Bereich des Softwaretestens tätig ist. Die Aktivitäten der ISEB wurden auch von anderen Ländern aufgegriffen und vergleichbare Initiationen gestartet. In Deutschland zeichnet vor allem die ASQF (vgl. www.asqf.de) und die Fachgruppe TAV (Test, Analyse und Verifikation von Software) der Gesellschaft für Informatik e. V. (kurz: GI) für dieses Thema verantwortlich (Spillner und Linz, 2003).