UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Institut für Informatik
Arbeitsgruppe Softwaretechnik
Warburger Str. 100
33098 Paderborn

Validierung und Verifikation (inkl. Testen, Model-Checking und Theorem Proving)

Ausarbeitung
im Rahmen des Seminars

Analyse, Entwurf und Implementierung zuverlässiger Software

von

Torsten Bresser
Geibelstraße 6
33129 Delbrück

betreut durch

Dr. Holger Giese

Paderborn, Januar 2004

Inhaltsverzeichnis

1 Motivation **1**

2 Grundbegriffe und Definitionen **2**

2.1 Grundbegriffe . . . 3
2.2 Verifikation . . . 3
2.3 Validierung . . . 3
2.4 Fehler . . . 4
2.5 Testen . . . 4
2.6 Theorem Proving . . . 5
2.7 Model-Checking . . . 5

3 Verifikation **6**

3.1 Manuelle/nicht-formale Verifikation . . . 7
3.1.1 Testen . . . 7
3.1.2 Vorteile . . . 7
3.1.3 Nachteile . . . 8
3.2 Automatisierte/formale Verifikation . . . 8
3.2.1 Theorem Proving . . . 8
3.2.2 Model-Checking . . . 10

4 Validierung **13**

4.1 System Validation/Accepting Testing . . . 14
4.2 Environment Simulation . . . 14

5 Testen **15**

5.1 Formen . . . 15
5.1.1 Unterscheidungskriterium Ausführung . . . 15
5.1.2 Unterscheidungskriterium Verfügbare Informationen . . . 15
5.1.3 Unterscheidungskriterium Ebene der Testdurchführung . . . 16
5.1.4 Unterscheidungskriterium Test Coverage . . . 16

5.2 Dynamisches Testen 17
5.3 Analyse/Statisches Testen 18
5.4 Vorteile 18
5.5 Nachteile 19

6 Zusammenfassung **20**

Kapitel 1

Motivation

In immer mehr Bereichen des menschlichen Lebens werden Computer eingesetzt. Viele Rechner verrichten still und leise von den meisten Menschen unbemerkt ihren Dienst als so genannte eingebettete Systeme in nahezu jeder Art von technischen Geräten – von Waschmaschinen bis zum Pulsmesser. Eine große Zahl dieser Systeme dient der Erhöhung des Komforts bei der Bedienung der entsprechenden Systeme.

Immer mehr Computer werden aber auch in sicherheitskritischen Bereichen eingesetzt. Sie dienen in der Regel zur Erhöhung der Sicherheit. Beispiele für Systeme dieser Art finden sich u. a. in den folgenden Bereichen:

- Straßenverkehr (Bremsassistent und ESP im Auto, etc.)
- Luftfahrt (Autopiloten usw.)
- Gesundheitswesen
- Atomkraftwerke

Die in sicherheitskritischen Bereichen eingesetzte Hard- und Software besitzt die Eigenschaft, dass Fehler zu katastrophalen Folgen für Menschen und deren Umwelt führen können. Damit ist das vorrangige Ziel bei der Entwicklung von sicherheitskritischer Hard- und Software, Fehler soweit wie möglich zu verhindern bzw. schon frühzeitig im Entwicklungsprozess zu erkennen und zu beheben.

In den folgenden Kapiteln wird der Frage nachgegangen, wie genau dieses Ziel insbesondere bei der Entwicklung von Software erreicht werden kann. Es werden Techniken vorgestellt, die unter bestimmten Voraussetzungen dazu geeignet sind, die Abwesenheit bestimmter Fehler in einer Software nachzuweisen.

Kapitel 2

Grundbegriffe und Definitionen

Bevor in den folgenden Kapiteln genauer auf die Themen Verifikation, Validierung und Testen eingegangen wird, gibt dieses Kapitel einen groben Überblick über das gesamte Themengebiet. Dabei werden die wichtigsten - und wahrscheinlich nicht jedem geläufigen - Begriffe definiert und erläutert. Unter anderem wird der Frage nachgegangen, zu welchem Zweck und Zeitpunkt während der Software-Entwicklung verifiziert, validiert und getestet werden muss, um den Anforderungen an sicherheitskritische Anwendungen gerecht werden zu können.

Abbildung 2 zeigt ein vereinfachtes Modell einer Projekt-Entwicklung. Startend mit den Kundenanforderungen wird über verschiedene Stufen Schritt für Schritt das endgültige Produkt (Hard- oder Software) erstellt. Zu diesem Zweck wird der Output einer Phase als Input für die nachfolgende herangezogen, um durch Transformation verfeinert zu werden. Jeder nach unten zeigende Pfeil in Abbildung 2 steht dabei für eine Konvertierung der Beschreibung des Systems von einer Form in eine andere.

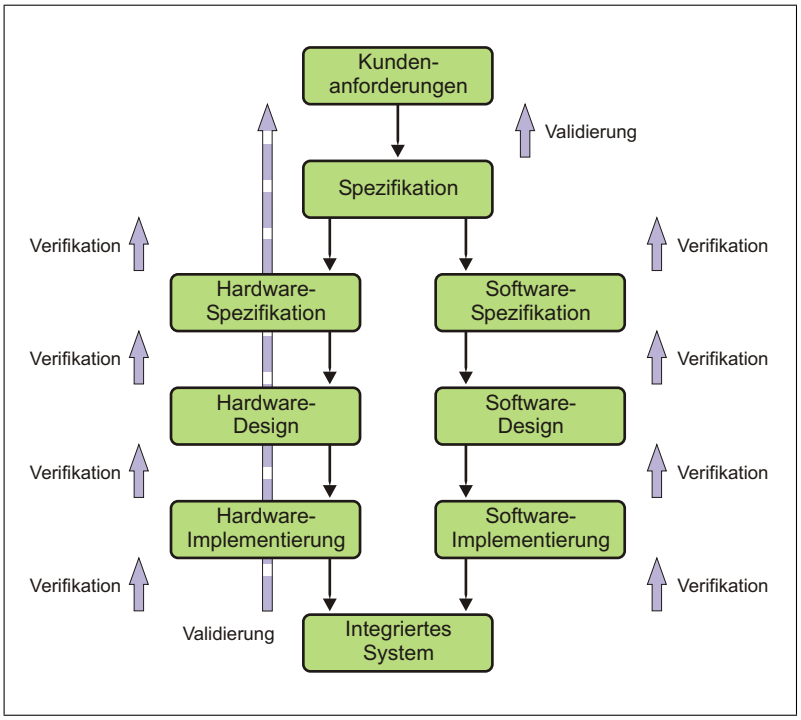

Abbildung 2.1: Einfaches Entwicklungsmodell mit formalem Verifikationsprozess

Dabei besteht grundsätzlich bei jeder Transformation die Gefahr, dass sich Fehler in den Entwicklungsprozess einschleichen, was insbesondere aus den folgenden Gründen problematisch ist:

- Fehler, die sich in frühen Entwicklungsphasen einschleichen, aber erst in späteren Phasen entdeckt werden, führen (meistens) zu sehr hohen Kosten.
- Fehler, die während des gesamten Entwicklungsprozesses unentdeckt bleiben, können vor allem bei sicherheitskritischen Anwendungen zu katastrophalen Folgen führen. Man denke in diesem Zusammenhang nur an das Versagen von Steuerungssoftware für Atomreaktoren. Fehler bei der Entwicklung müssen gerade bei sicherheitskritischer Hard- und Software (so gut wie möglich) verhindert werden. Wie die folgenden Kapitel zeigen, gibt es jedoch – außer bei trivialen Programmen – keine Möglichkeit, die Fehlerfreiheit von Software zu beweisen.

2.1 Grundbegriffe

Definition 2.1 (Spezifikation) *Eine Spezifikation ist ein Text, der die Syntax und Semantik eines bestimmten Bestandteiles beschreibt bzw. eine deklarative Beschreibung, was etwas ist oder tut.* [7]

Definition 2.2 (Modell) *Ein Modell ist ein bewusst konstruiertes Abbild der Wirklichkeit, das auf der Grundlage einer Struktur-, Funktions- oder Verhaltensanalogie zu einem entsprechenden Original von einem Subjekt eingesetzt bzw. genutzt wird, um eine bestimmte Aufgabe lösen zu können, deren Durchführung mittels direkter Operation am Original zunächst oder überhaupt nicht möglich bzw. unter gegebenen Bedingungen zu aufwendig oder nicht zweckmäßig ist.* [5]

Definition 2.3 (System) *Ein System ist eine Menge von Elementen, zwischen denen Beziehungen existieren und die dazu dient, eine bestimmte Aufgabe zu erfüllen und durch eine Menge von Modellen aus verschiedenen Blickwinkeln beschrieben wird. Ein System ist möglicherweise in weitere Teilsysteme zerlegt.* [7]

2.2 Verifikation

Um Fehler soweit wie möglich zu verhindern, bedient man sich während des Entwicklungsprozesses der so genannten Verifikation. Den Prozess der Verifikation veranschaulicht Abbildung 2. Sie zeigt, dass jeder Transformationsschritt einhergeht mit der Überprüfung auf Korrektheit der durchgeführten Aktivität. Dabei ist zu zeigen, dass die Beschreibung des Systems, die die Eingabe für die entsprechende Phase bildet, funktionell äquivalent zu ihrer Ausgabe ist.

Definition 2.4 (Verifikation) *Verifikation ist der Prozess des Überprüfens, ob die Ausgabe einer Lebenszyklusphase die durch die vorhergehende Phase spezifizierten Anforderungen erfüllt.* [3]

2.3 Validierung

Dem Diagramm ist weiterhin zu entnehmen, dass neben den Verifikationspfeilen noch aufwärts gerichtete Pfeile existieren, die mit Validierung beschriftet sind. Von Validierung spricht man immer dann, wenn eine Überprüfung auf Korrektheit mit Bezug auf die Kundenanforderungen durchgeführt wird. In diesem Fall lässt sich eine Verifikation nicht durchführen, da die Kundeninformationen nur eine nicht-formale Beschreibung des Systems beinhalten und somit keinen geeigneten Input für eine Verifikation bilden.

Definition 2.5 (Validierung) *Validierung ist der Prozess des Bestätigens, dass die Spezifikation einer Phase oder des Gesamtsystems passend zu und konsistent mit den Anforderungen des Kunden ist.* [3]

2.4 Fehler

Den vorherigen Absätzen war zu entnehmen, dass Verifikation und Validierung zur Reduzierung von Fehlern eingesetzt werden. Bei der Entwicklung von Hard- und Software gibt es jedoch verschiedene Arten von Fehlern und Fehlerzuständen, die einer genaueren Betrachtung bedürfen.

Definition 2.6 (Fehlerursache) *Eine Fehlerursache (Fault) ist ein Defekt in einem System.* [4]

Definition 2.7 (Fehlzustand) *Ein Fehlzustand (Error) ist eine Abweichung vom geforderten Verhalten des Systems oder Subsystems.* [4]

Definition 2.8 (Ausfall) *Ein Ausfall des Systems (System Failure) tritt auf, wenn das System bei der Durchführung einer Pflicht-Funktion versagt.* [4]

Fehlerursachen, Fehlzustände und Ausfälle bilden eine Art Hierarchie: aus Fehlerursachen werden Fehlzustände, die wiederum zu System-Ausfällen werden können. Handelt es sich bei der Komponente, in der ein Ausfall auftritt, um eine Subkomponente einer anderen, entsteht eine so genannte Fault/Failure-Kette. Dabei wird ein Ausfall der Unterkomponente zu einer Fehlerursache der übergeordneten Komponente.

Das Ziel von Validierung und Verifikation ist es, möglichst alle Fehlerursachen in einem System aufzuspüren um das Entstehen von Fehlzuständen und Ausfällen zu unterbinden. Die folgende Klassifizierung gibt einen (unvollständigen) Überblick über mögliche Faults.

Art	**Ausprägung**	**Erläuterung**
Natur		
	Zufällige Fehler	treten im Hardware-Bereich auf, z. B. durch Abnutzung oder äußere Einflüsse wie Strahlung
	logische, systematische (Design-)Fehler	können sowohl in Hard- als auch in Software auftreten; durch Validierung und Verifikation während der Entwicklungsphase wird versucht, diese Art von Fehlern zu vermeiden.
Dauer		
	permanent	
	transient	flüchtige, vergängliche Fehler
	intermittent	periodisch auftretende Fehler

2.5 Testen

Definition 2.9 (Testen) *Testen ist der Prozess des Verifizierens und Validierens eines Systems oder seiner Komponenten.* [3]

Wie später noch beschrieben wird, ist zum Testen eines Systems nicht unbedingt die Ausführung desselben notwendig. Normalerweise ist es nicht möglich, einen erschöpfenden Test durchzuführen, der Fehler vollständig ausschließen kann. Dazu müssten *alle* möglichen Eingaben eines Programms bzw. Moduls durchgetestet werden, was nur bei trivialen Programmen möglich ist.

2.6 Theorem Proving

Theorem Proving wird immer dann genutzt, wenn auf Basis einer gegebenen Logik, z. B. Aussagenlogik oder Prädikatenlogik erster Ordnung, Aussagen bewiesen werden sollen. Liegt die Spezifikation in einer Form vor, die von einem Theorem Prover verarbeitet werden kann, kann im Rahmen der Verifikation voll- oder teilautomatisch das Vorhandensein bestimmter Eigenschaften bzw. das Nichtvorhandensein bestimmter Fehler nachgewiesen werden. Im Detail wird das Theorem Proving in Kapitel 3.2.1 besprochen.

2.7 Model-Checking

Definition 2.10 (Model-Checking) *Model-Checking ist eine Technik zur vollautomatischen Verifikation reaktiver Systeme mit endlichem Zustandsbaum.*[6]

Basis des Model-Checkings, dass ebenso wie das Theorem Proving im Rahmen der Verifikation eingesetzt wird, bilden temporale Logik- bzw. endliche Automaten-Spezifikationen. Näheres zum Thema Model-Checking findet sich im Kapitel 3.2.2.

Kapitel 3

Verifikation

Wie bereits im vorhergehenden Kapitel beschrieben, ist Verifikation ein Prozess zum Aufspüren von Fehlern. Damit nimmt die Verifikation einen wichtigen Platz im Bereich der Entwicklung sicherheitskritischer Anwendungen ein.

Unabhängig von der eingesetzten Technik kann mittels Verifikation *nicht* nachgewiesen werden, dass das betrachtete Produkt fehlerfrei ist. Der Grund dafür liegt darin, dass die vom Kunden an das Produkt gestellten Anforderungen in nicht-formaler Form vorliegen und damit keinen geeigneten Input für den Verifikationsprozess bilden. Aufgabe der Verifikation ist damit, zu zeigen, dass *nach* dem Zeitpunkt der Spezifikations-Erstellung keine Fehler in den Entwicklungsprozess Einzug gehalten haben. Falls bereits die Spezifikation Fehler enthält, werden diese mittels Verifikation nicht (zwingend) nachgewiesen.

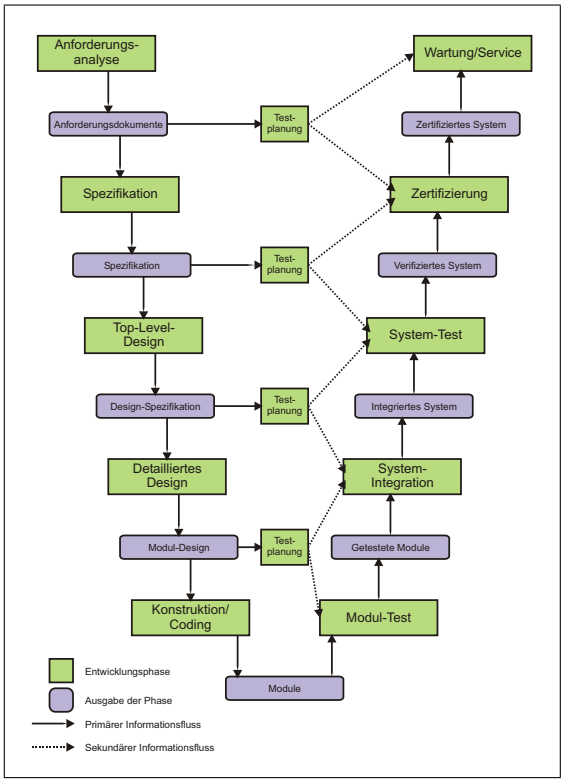

Abbildung 3.1: V-Modell inkl. Planung von Tests

Bei der Entwicklung sicherheitskritischer Anwendungen beansprucht die Verifikation – genau wie die Validierung – einen sehr großen Teil am Gesamtaufwand der Entwicklung für sich. Daraus ergibt sich die Notwendigkeit, alle mit der Verifikation in Zusammenhang stehenden Aktivitäten zu einem sehr frühen Zeitpunkt des Projektes zu planen, um die Kosten für diesen Teil des Projektes sowohl schätzen als auch minimieren zu können.

Wie eine solche Planung im Falle der Verifikationstechnik Testen aussehen kann, zeigt Abbildung 3.1. Das dort zu sehende V-Modell zeigt auf der linken Seite die Schritte des Projekts bis zur Implementierung. Die rechte Seite repräsentiert die Testphasen. Während das System schrittweise von einer Menge von Modulen zu einem integrierten System transformiert wird, muss es immer wieder verifiziert werden gegen frühere Transformationen der Spezifikation. So muss beispielsweise die Implementierung eines Moduls auf Übereinstimmung mit seinem Design und das Gesamtsystem auf Übereinstimmung mit seiner Spezifikation getestet werden. Jede neue Transformation der Spezifikation, die während einer auf der linken Seite gezeigten Projektphase durchgeführt wird, muss einhergehen mit der Erstellung eines Testplans. Dieser sollte beschreiben, wie die Implementierung der entsprechenden Transformation verifiziert werden kann.

Zum Verifizieren einer Software können unterschiedliche Techniken eingesetzt werden. Einige Autoren sind der Auffassung, dass unter den Begriff Verifikation nur formale Techniken wie Theorem Proving und Model-Checking fallen. Wie bereits der vorhergehende Abschnitt deutlich gemacht hat, werden an dieser Stelle auch nicht-formale Methoden wie das Testen behandelt.

3.1 Manuelle/nicht-formale Verifikation

3.1.1 Testen

Unter nicht-formaler Verifikation versteht man im Wesentlichen das dynamische und statische Testen, um Fehler, die sich während des Entwicklungsprozesses eingeschlichen haben, zu finden. Nicht-formal heißt diese Art der Verifikation, da sie nicht auf mathematischer x und Beweisen beruht. Was genau unter das Stichwort Testen fällt, wird später in einem eigenen Kapitel erklärt.

Nicht *zu* wörtlich sollte man das Wort *manuell* nehmen. Gerade bei großen Projekten, die über eine lange Zeit entwickelt werden, ist es üblich, die Tests mit Hilfe von Programmen (teil-)automatisch durchführen zu lassen. Oft wird jedoch die Auswertung der Ergebnisse von Testläufen manuell durchgeführt.

3.1.2 Vorteile

Wird die Verifikation mittels Tests durchgeführt, ergeben sich einige Vorteile im Vergleich zu den im Folgenden vorgestellten formalen Methoden. An dieser Stelle wird damit Kapitel 5 vorgegriffen, wo noch einmal speziell auf die Vor- und Nachteile von verschiedenen Testmethoden und -kriterien eingegangen wird.

- Das Testen erfordert eine weniger umfangreiche Ausbildung als formale Techniken. Testen gilt als eine der am leichtesten durchzuführenden Maßnahmen zur Erhöhung der Qualität von Software und Hardware. Es wird kein tiefgehendes mathematisches Wissen benötigt.
- Auch Systeme, die aufgrund ihrer Komplexität oder Struktur nicht mit formalen Techniken überprüft werden können, lassen sich mittels Tests verifizieren.

3.1.3 Nachteile

- Mit dem (dynamischen) Testen kann erst zu einem sehr späten Zeitpunkt begonnen werden, nämlich erst während bzw. nach der Implementierung. Wie bereits im einleitenden Kapitel erläutert, sind Fehler, die erst spät entdeckt werden, in der Regel wesentlich schwieriger und kostspieliger zu beheben als früh erkannte.
- Erschöpfendes Testen ist bei nicht-trivialen Programmen aufgrund von Zeit- und Speicherplatzrestriktionen nicht möglich. Ohne erschöpfendes Testen können jedoch Fehler übersehen werden. Mittels Testen lässt sich nicht die Abwesenheit, sondern nur das Vorhandensein von Fehlern nachweisen.

3.2 Automatisierte/formale Verifikation

Mathematische Logik bildet die Basis für die formale Verifikation. Wie eine Programmiersprache kombiniert Logik Syntax und Semantik. In den folgenden Unterkapiteln wird gezeigt, welche Möglichkeiten der Verifikation zur Verfügung stellen, wenn die Spezifikation in formaler Form, beispielsweise in Prädikatenlogik erster Ordnung, in Aussagenlogik oder in Form eines endlichen Automaten vorliegt.

3.2.1 Theorem Proving

Das Theorem Proving lässt sich auf Basis verschiedener Logiken durchführen. Peled [2] beschreibt das Theorem Proving anhand der Prädikatenlogik erster Ordnung und der eingeschränkteren Aussagenlogik. Um Aussagen auf Basis einer Spezifikation beweisen zu können, bedarf es eines Beweissystems, das passend zur verwendeten Logik ist. Das Beweissystem enthält Axiome und Regeln, mittels derer es z. B. möglich ist zu beweisen, dass eine Formel eine Tautologie ist oder dass eine Formel unter einer bestimmten, gegebenen Menge von Annahmen Gültigkeit besitzt. Ein automatischer Theorem Prover versucht durch geschickte Anwendung der Regeln die zu beweisende Aussage so umzuformen und zu vereinfachen, dass ihre Richtigkeit festgestellt werden kann.

Beispiel

Da normalerweise die von einem automatischen Theorem Prover erzeugte Ausgabe selbst bei einfachen Beweisen sehr lang ist, werden an dieser Stelle nur die ersten Schritte eines Beispielbeweises gezeigt. Der interessierte Leser sei auf weiterführende Literatur verwiesen. Zu beweisen ist die folgende propositionale Tautologie. Benutzt wird dabei das bei Peled [2] definierte Beweissystem.

$$((A \to C) \wedge (B \to C)) \to ((A \vee B) \to C) \tag{3.1}$$

Ein automatischer Theorem Prover würde daraus beispielsweise das folgende Unterziel erstellen (Prämissen erscheinen im Folgenden über, Konsequenzen unter dem Strich):

$$\frac{(A \to C) \wedge (B \to C)}{(A \vee B) \to C} \tag{3.2}$$

Die Prämisse lässt sich weiter vereinfachen, was zu dem folgenden Subziel führt:

$$\frac{\begin{array}{c} A \to C \\ B \to C \end{array}}{(A \vee B) \to C} \tag{3.3}$$

Die Hypothese kann aufgesplittet werden, so dass alles vor dem → Teil der Prämisse wird:

$$\frac{\begin{array}{c} A \to C \\ B \to C \\ A \vee B \end{array}}{C} \tag{3.4}$$

Der folgende Schritt zeigt erstmals, wie aus einem Subziel vom Theorem Prover zwei Unterziele generiert werden:

$$\frac{\begin{array}{c} A \to C \\ B \to C \\ A \end{array}}{C} \qquad \frac{\begin{array}{c} A \to C \\ B \to C \\ B \end{array}}{C} \tag{3.5}$$

Das übergeordnete Ziel (das *eigentliche* Beweisziel) und die vom Theorem Prover generierten Subziele bilden einen Beweisbaum, der bei jedem Beweis-Schritt wächst oder schrumpft (siehe Abbildung 3.2 für den Beweisbaum bis zum Schritt 3.5).

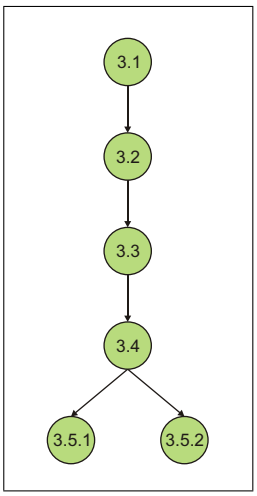

Abbildung 3.2: Beweisbaum, wie er von einem Theorem Prover erzeugt wird

Problematisch an automatischen Theorem Provern ist, dass Regeln in praktischen Beweissystemen nicht notwendigerweise nur kürzere Prämissen und Konsequenzen erzeugen, sondern unter Umständern auch längere. Daraus ergibt sich die Konsequenz, dass das Theorem Proving bereits für einfache Logiken (wie Aussagenlogik) zur Klasse der NP-vollständigen Probleme zählt. Für komplexere Logiken gehört das Theorem Proving oft zur Klasse der nicht entscheidbaren Probleme.

Deshalb können Theorem Prover trotz ausgeklügelter Heuristiken nicht garantieren, bestimmte Aussagen in polynomieller Zeit beweisen oder widerlegen zu können. In der Praxis benötigen automatische Theorem Prover die Mithilfe eines erfahrenen Menschens. Sie werden eher eingesetzt, um den Menschen beim Beweisen zu unterstützen und zur Disziplin beim Umgang mit Beweisen zu zwingen, als völlig selbstständig zu arbeiten.

Es gibt zwei verschiedene Arten von Theorem Provern, die dem Benutzer unterschiedliche Disziplin bei der Arbeit abverlangen:

- Der Puristen-Ansatz zielt einzig und allein darauf ab, die Qualität der verifizierten Software zu erhöhen. Der Theorem Prover benutzt dabei nur Regeln, deren Korrektheit bewiesen

wurde. Damit können sich in die Beweise keine Fehler einschleichen (unter der Voraussetzung, dass der Theorem Prover keine Fehler enthält und das zugrundeliegende Beweissystem ebenfalls fehlerfrei ist). Offensichtlicher Nachteil dieses Ansatzes ist, dass er sich sehr zeitaufwendig gestalten kann.

Ein bekannter Theorem Prover, der zu dieser Klasse gehört, ist HOL (Higher Order Logic, http://www.cl.cam.ac.uk/Research/HVG/HOL).

- Der praktische Ansatz verfolgt das Ziel, dem Benutzer den Beweis schnell zu liefern. Dazu kann der Benutzer dem System neue Axiome beibringen. Der Theorem Prover übernimmt diese ungeprüft und verwendet sie zur Konstruktion von Beweisen. Dieser Ansatz ist für sicherheitskritische Bereiche als wenig geeignet einzuschätzen. Die Praxis zeigt, dass Anwender von Theorem Provern oft zu optimistisch die erweiterten Möglichkeiten nutzen. Die Folge sind 'schwächere' Beweise bzw. (im schlimmsten Fall) völlig unsinnige Aussagen.

  Der Theorem Prover PVS (http://pvs.csl.sri.com) gehört zu dieser Klasse von Theorem Provern.

Vorteile

Theorem Prover sind eine große Hilfe bei der Durchführung von Beweisen. Falls die Komplexität nicht zu groß ist, besteht die Möglichkeit, dass ein Theorem Prover selbstständig eine Aussage beweisen kann. Selbst wenn das nicht der Fall ist und der Benutzer eingreifen muss, ergeben sich im Vergleich zum rechnerlosen Beweisen einige Vorteile:

- Theorem Prover können die Verlässlichkeit sicherheitskritischer Software steigern.
- Der Theorem Prover kann wichtige Hinweise geben, wie ein Beweis weitergeführt werden sollte (selbst dann, wenn er nicht in der Lage ist, den Beweis selbstständig zu beenden).
- Der Theorem Prover hilft, Fehler zu vermeiden, indem er jeden Beweisschritt auf Richtigkeit überprüft.
- Die Aufbereitung von Beweisen für die Dokumentation erfordert normalerweise großen Aufwand. Theorem Prover besitzen meist eine *Pretty-Printing*-Funktion für Beweise, die dann leicht in die Projekt-Dokumentation eingefügt werden können.

Nachteile

Das Kosten-/Nutzen-Verhältnis ist bei Theorem Provern genau abzuwägen. Der Einsatz eines Theorem Provers lohnt meist nur bei der Entwicklung sicherheitskritischer Anwendungen.

Theorem Prover benötigen aufgrund der Unentscheidbarkeit bei komplexeren Logiken die Unterstützung eines erfahrenen Menschen, der Erfahrung im Beweisen von logischen Aussagen besitzt. Die dafür notwendigen Kenntnisse besitzen normalerweise nur Mathematiker, Logiker oder ausgebildete Ingenieure.

Beim Theorem Prover, die den praktischen Ansatz verfolgen, besteht durch vom Menschen eingeschleuste Fehler die Gefahr, dass die Beweise nur eine eingeschränkte oder überhaupt keine Aussagekraft besitzen.

3.2.2 Model-Checking

Das Model-Checking ist eine weitere formale Technik zur Verifikation von Software. Wie die theoretische Informatik lehrt, ist die vollautomatische Verifizierung für eine breite Klasse von Programmen ein unlösbares Problem. Es kann nicht erwartet werden, dass ein Verifizierer entwickelt

wird, der als Eingabe ein Programm und eine Spezifikation erhält und anhand dieser Daten vollautomatisch entscheidet, ob das Programm die Spezifikation erfüllt.

Um in der Praxis nicht völlig auf die Unterstützung eines Computers beim Verifikationsprozess verzichten zu müssen, können die folgenden Maßnahmen getroffen werden:

- Beschränkung auf eine kleinere Klasse von Programmen, für die automatische Verifikationsalgorithmen existieren. Model-Checking verfolgt durch eine Beschränkung auf Programme mit endlich vielen Zuständen dieses Ziel.
- Statt das Gesamtsystem zu verifizieren, beschränkt man sich auf eine Verifizierung der sicherheitskritischen Teile, wie die dem Programm zugrundeliegenden Algorithmen oder Kommunikationsprotokolle.

Grundlagen des Model-Checking

Model-Checker erwarten als Eingabe ein Programm und eine Spezifikation in Form endlicher Automaten (bzw. in einer Form, die sich in einen endlichen Automaten überführen lässt, z. B. LTL-Formeln). Sowohl das Modell des Systems als auch die Spezifikation müssen dabei über demselben Alphabet definiert sein. Das Systemmodell $\mathcal{A}$ erfüllt die Spezifikation $\mathcal{B}$, falls die Sprache des Systems $\mathcal{A}$ eine Untermenge der Sprache der Spezifikation $\mathcal{B}$ ist, wenn also gilt:

$$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B}) \tag{3.6}$$

Ein Model-Checker hat damit festzustellen, ob die obige Bedingung richtig oder falsch ist. In der Praxis ist es jedoch schwierig, auf die Inklusion einer Sprache zu testen. Deshalb wird die Bedingung 3.6 für automatische Model-Checker abgewandelt. Sei $\overline{\mathcal{L}(\mathcal{B})}$ die Sprache $\sum^{\omega} \setminus \mathcal{L}(\mathcal{B})$ von Worten, die *nicht* von $\mathcal{B}$ akzeptiert werden, also das Komplement der Sprache $\mathcal{L}(\mathcal{B})$. Dann kann 3.6 umgeschrieben werden zu

$$\mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(\mathcal{B})} = \emptyset \tag{3.7}$$

Diese Bedingung lässt sich in der Praxis einfacher überprüfen. Ein Model-Checker verfolgt damit die folgende Strategie:

1. Bilde das Komplement des Automaten $\mathcal{B}$, also einen Automaten $\overline{\mathcal{B}}$, der die Sprache $\overline{\mathcal{L}(\mathcal{B})}$ erkennt.
2. Bilde den Schnitt der Automaten $\mathcal{A}$ und $\overline{\mathcal{B}}$.
3. Überprüfung, ob der Schnitt leer ist. Falls das der Fall ist, folgt $\mathcal{A}$ der Spezifikation $\mathcal{B}$. Anderenfalls bildet ein akzeptiertes Wort der nicht-leeren Schnittmenge ein Gegenbeispiel.

Für die Details, wie die Schnittmenge zweier Automaten konstruiert werden kann und wie auf das Nichtvorhandensein von Elementen in der Schnittmenge geprüft werden kann, sei auf die weiterführende Literatur verwiesen.

State Space Explosion

Die größte Herausforderung beim Model-Checking ist, das Problem der so genannten State Space Explosion in den Griff zu bekommen. Darunter versteht man das Problem, dass durch nebenläufige Programmteile die Zahl der Zustände des Gesamtsystems stark steigen kann. Im Worst-Case

besteht ein System aus n Komponenten mit jeweils m Zuständen, die nicht untereinander kommunizieren oder interagieren. Daraus ergibt sich eine Zahl von m^n Zuständen insgesamt, die ein Model-Checker bei seiner Arbeit zu berücksichtigen hat.

Um dem Problem der Zustandsraum-Explosion zu entgehen, wurden eine Reihe von Strategien entwickelt. Keine kann eine effiziente Lösung des Problems garantieren, aber sie haben sich in der Praxis als hilfreich herausgestellt. Die meisten Model-Checker, die auf dem Markt angeboten werden, nutzen eine oder eine Kombination aus mehreren der folgenden Strategien, die allesamt das Ziel verfolgen, den Zustandsraum durch geschickte Zusammenfassung von Zuständen zu verkleinern.

- Binary Decision Diagrams: Anstatt jeden Zustand im Zustandsraum als eigene Entität zu betrachten, wird eine Menge von Zuständen in einem gerichteten azyklischen Graphen (DAG) gespeichert. Durch geschickte Zusammenfassung isomorpher Unterbäume kann der Zustandsraum verkleinert werden.
- Partial Order Reduction: Erlaubt es durch Ausnutzung der Kommutativität nebenläufiger Transitionen, nur eine Teilmenge aller Ausführungen zu untersuchen (mit einem entsprechend kleineren Zustandsraum). Nutzt die Tatsache aus, dass in den meisten Spezifikationen nicht unterschieden wird zwischen Ausführungen, die sich nur in der Reihenfolge gleichzeitig ausgeführter Ereignisse unterscheiden.
- Symmetrie: Diese Methode basiert auf der Ausnutzung von Permutationen zwischen Komponenten eines Zustands. Genutzt wird diese Methode vorwiegend beim Model-Checking auf Basis von Hardware-Spezifikationen, so dass an dieser Stelle auf Einzelheiten verzichtet wird.

Vorteile

Model-Checking läuft zu einem großen Teil automatisch ab. Nur das Erstellen der Modelle und eventuell notwendige Abstraktionen sind durch den Benutzer vorzunehmen. Damit ist der manuelle Teil beim Model-Checking deutlich kleiner als bei anderen Maßnahmen zur Erhöhung der Software-Zuverlässigkeit.

Die dem Model-Checking zugrundeliegenden Ideen sind relativ einfach zu verstehen und zu implementieren, was zu einer großen Anzahl verfügbarer Tools führte. Vorteilhaft an Model-Checkern ist die automatische Generierung von Gegenbeispielen im Falle einer fehlgeschlagenen Verifizierung.

Nachteile

Größter Nachteil des Model-Checkings ist die Komplexität, die häufig den Eingriff eines Experten erfordert, der Parameter wie Größe des Stacks, des Speicherverbrauchs etc. einstellt. Erfahrung und Experimentierfreude sind also Voraussetzungen für den Einsatz von Model-Checkern. Falls Programme mit einem unendlichen oder sehr großen Zustandsraum oder mit komplexen Datenstrukturen verifiziert werden müssen, lässt sich Model-Checking u. U. nicht einsetzen.

In der Regel haben Model-Checker eigene Modellierungs-Sprachen. Es ist oft nicht möglich, Modelle in diesen Sprachen automatisch generieren zu lassen bzw. nur unter Nicht-Ausnutzung wünschenswerter Optimierungen und Heuristiken, die die meisten Model-Checker mitbringen. Das manuelle Erstellen von Modellen birgt die Gefahr der Einschleusung von Fehlern, so dass die Ausgabe eines Model-Checkers mit Vorsicht zu genießen ist.

Die Hardware-Anforderungen der Model-Checker sind trotz immer besserer und leistungsfähiger Rechner nicht zu unterschätzen. In der Praxis muss mit Programmabbrüchen aufgrund von Speichermangel gerechnet werden oder mit nicht vertretbaren Laufzeiten.

Kapitel 4

Validierung

Während die Verifikation jeweils den Output einer Entwicklungsphase auf Konsistenz mit der vorherigen Phase überprüft, wird die Validierung eingesetzt, um den Output einer Entwicklungsphase mit den Kundenanforderungen zu vergleichen. Dabei ist die Validierung nicht auf die Überprüfung bestimmter Phasen beschränkt, im Normalfall wird sie jedoch gegen Ende des Entwicklungsprozesses durchgeführt, um zu überprüfen bzw. zu zeigen, dass das integrierte System den Kundenanforderungen entspricht. Da Fehler, die erst zu diesem späten Zeitpunkt entdeckt werden, sehr teuer zu beheben sind, sollte jedoch auch schon zu Beginn der Entwicklung untersucht werden, ob die Spezifikation des Produktes die Kundenanforderungen richtig formalisiert. Bei sorgfältiger Verifikation der nachfolgenden Phasen kann die Wahrscheinlichkeit für das Auftreten von Fehlern bei der Validierung des Gesamtsystems stark gesenkt werden.

Im Gegensatz zur Verifikation, die formal oder nicht-formal durchgeführt werden kann, ist die Validierung ein nicht-formaler Prozess. Der Grund dafür ist, dass die Kundenanforderungen in informeller Form vorliegen. Die Benutzung menschlicher Sprache in den Anforderungsdokumenten kann zu unpräzisen, unvollständigen, möglicherweise sogar inkonsistenten und sich widersprechenden Aussagen führen. Die Möglichkeiten, Computer bei der Analyse solcher Dokumente einzusetzen, sind damit beschränkt.

Die zentrale Tätigkeit bei der Validierung ist – ebenso wie bei der nicht-formalen Verifikation – das Testen. Da dem Testen ein eigenes Kapitel gewidmet ist, soll an dieser Stelle nur kurz auf einige beim Validieren eingesetzte Testmethoden eingegangen werden. Alle im folgenden Kapitel beschriebenen Testformen lassen sich ebenso für die Validierung einsetzen.

Obwohl die Validierung unter Umständen eine der letzten Tätigkeiten während der Entwicklung einer Software ist, sollte die Planung schon zu einem sehr frühen Zeitpunkt stattfinden. Dazu wird ein so genannter Overall Safety Validierungsplan erstellt, in dem u. a. die folgenden Fragen beantwortet werden:

- Wann wird validiert?
- Welche Modi besitzt die zu validierende Software?
- Welche Arten von Tests werden durchgeführt (technische Strategie)?
- Welche Maßzahlen, Techniken und Prozeduren kommen zum Einsatz?
- Welche Pass-/Fail-Kriterien werden beim Testen benutzt?

Gerade im Bereich sicherheitskritischer Anwendungen reicht es nicht, den Overall Safety Validierungsplan zu erstellen und zu den geeigneten Zeitpunkten die Tests durchzuführen. Standards

wie der IEC 1508 fordern zusätzlich, dass verschiedene Personen, Abteilungen bzw. unabhängige Organisationen mit an der Validierung beteiligt werden, um ein Höchstmaß an Sicherheit zu erreichen und so viele Fehler wie möglich zu entdecken. Die folgende Tabelle gibt wieder, für welches Sicherheitsniveau welche Empfehlungen gegeben werden:

Grad der Unabhängigkeit	SIL 1	SIL 2	SIL 3	SIL 4
1. Unabhängige Person	+	+	–	–
2. Unabhängige Abteilung		+	+	–
3. Unabhängige Organisation			+	+

Das nächste Unterkapitel greift dem folgenden Kapitel vor und beschreibt kurz zwei mögliche Arten von Tests, die im Rahmen der Validierung häufig eingesetzt werden.

4.1 System Validation/Accepting Testing

Das System Validation Testing wird vom Auftragnehmer durchgeführt. Ziel dieser Art von Test ist es, dem Kunden die Abwesenheit von Fehlern zu demonstrieren. Für das System Validation Testing bieten sich die im nächsten Kapitel beschriebenen Testverfahren an. Insbesondere kann das System Validation Testing dazu genutzt werden, dem Auftraggeber z. B. durch das Anwenden von Stresstests positive Features oder besonders gelungene Umsetzungen der Anforderungen zu demonstrieren.

Im Gegensatz zum System Validation Testing wird das Accepting Testing vom Auftraggeber selbst durchgeführt. Ziel ist es, festzustellen, ob die abgelieferte Software den Anforderungen des Kunden voll gerecht wird.

4.2 Environment Simulation

Manchmal kann im Rahmen der Validierung ein sicherheitskritisches System nicht in seiner natürlichen Umgebung getestet werden. Als Grund dafür kommen zu hohe Kosten oder zu große damit verbundene Gefahren (z. B. beim Test einer Notabschaltung eines Atomreaktors) in Frage. In solchen Fällen greift man in der Regel auf die Environment Simulation zurück. Environment Simulation kann in zwei Formen vorkommen:

- Simulation natürlicher Umgebungen: Die simulierte Umgebung ist nicht konstruiert, sondern natürlich. Diese Form der Simulation zeichnet sich oft dadurch aus, dass das zur Verfügung stehende Wissen begrenzt ist. Dadurch gestaltet sich die Simulation häufig als schwierig.
- Simulation konstruierter Umgebungen: Bei dieser Form besteht die simulierte Umgebung aus einem oder mehreren anderen konstruierten Systeme. Damit stehen mehr Informationen als bei der vorherigen Form zur Verfügung. Aus Gründen der Einfachheit wird häufig nur eine eingeschränkte Simulation der an das zu testende System angrenzenden Komponenten durchgeführt.

Die Komplexität von Environment Simulation hängt von der Ebene ab, auf der die Tests durchgeführt werden. Meist sehr einfach sind Simulatoren, die die Umgebung eines einzelnen Moduls nachbilden. Die Simulation der Umgebung eines integrierten Systems hingegen kann eine sehr komplexe Aufgabe sein. Problematisch ist die Environment Simulation bei komplexem Datenaustausch oder bei kritischem Timing. Oft sind Simulatoren komplexer als die zu testende Einheit, für die sie entwickelt wurden. Hinzu kommt, dass bei der Entwicklung sicherheitskritischer Systeme die verwendeten Simulatoren (ebenso wie alle anderen Werkzeuge, die zur Validierung und Verifikation eingesetzt werden) maximal ein Level niedriger in der SIL-Einteilung eingestuft sein sollten als das zu testende System.

Kapitel 5

Testen

Wie bereits aus Kapitel 2 bekannt, werden Tests ebenso wie formale Verifikationstechniken (Model-Checking und Theorem Proving) eingesetzt, um die Qualität von Software zu erhöhen. Die Ergebnisse eines Tests können genutzt werden, um bestimmte Merkmale wie Sicherheit zu untersuchen. Da das (dynamische) Testen keine systematische Untersuchung *aller* möglichen Systemausführungen ist, können durch Tests nicht alle Fehler in einer Software (oder Hardware) gefunden werden. Damit ist die durch Testen erzielbare Software-Qualität weniger hoch als bei den umfassenderen formalen Verifikationstechniken. Nichtsdestotrotz ist das Testen eine wichtige (und die am meisten eingesetzte) Technik, um Fehler in Programmen zu entdecken. Vor allem in den Fällen, in denen ein gegebenes System zu groß oder komplex für die automatisierte Verifikation ist, gibt es kaum Alternativen zum Testen.

5.1 Formen

Tests können in verschiedenen Formen durchgeführt werden. Die folgenden Absätze gehen kurz auf die Unterscheidungskriterien ein und listen die verschiedenen Formen auf.

5.1.1 Unterscheidungskriterium Ausführung

Nach dem Kriterium der Ausführung werden in der Literatur die folgenden Testarten unterschieden:

- **Dynamisches Testen**: Diese Art des Testens beinhaltet die Ausführung des zu testenden Systems. In der Umgangssprache wird unter Testen immer diese Art von Test verstanden. Mehr zu diesem Thema findet sich in Kapitel 5.2.
- **Statisches Testen**: Statisches Testen ist die Untersuchung eines Systems oder einer Komponente ohne dessen/deren Ausführung. Diese Art der Programm-Analyse wird in Kapitel 5.3 näher beschrieben.

5.1.2 Unterscheidungskriterium Verfügbare Informationen

Nach den während des Testens zur Verfügung stehenden Informationen kann man die beiden folgenden Formen unterscheiden:

- **Black-Box-/Anforderungsbasiertes Testen**: Beim Black-Box-Testen wird ein System überprüft, ohne sich an der internen Struktur des Programms zu orientieren. Damit ist Black-Box-Testing immer eine Form von dynamischem Testen. Als Ausgangspunkt für die Tests dienen die Kundenanforderungen, auf deren Basis geeignete Testfälle erstellt werden. Black-Box-Testing kann auf Modul-Ebene durchgeführt werden, meist wird es jedoch für den Test des Gesamtsystems oder abgeschlossener Teilsysteme verwendet. Häufig eingesetzt wird Black-Box-Testing z. B. bei der Überprüfung von Compilern.
- **White-Box-/Transparent-Box-Testen**: White-Box-Testing ist das Testen unter Ausnutzung des Wissens über die interne Struktur des Systems. Diese Form des Testens wird in der Praxis häufiger angewandt als das Black-Box-Testing. In der Regel ist es einfacher, Testfälle zu generieren, wenn man Informationen über den Aufbau eines Systems besitzt. Statische Tests gehören immer auch zur Gruppe der White-Box-Tests, da eine Analyse ohne Kenntnis der Struktur eines Systems nicht durchführbar ist.

5.1.3 Unterscheidungskriterium Ebene der Testdurchführung

- **Unit (Module) Testing**: Die niedrigste Stufe, auf der Tests durchgeführt werden können. Das Testen beschränkt sich auf kleine Codestücke, die getrennt voneinander begutachtet werden.
- **Integration Testing**: Beim Integrationstest wird überprüft, ob unterschiedliche Codeteile – die möglicherweise von unterschiedlichen Teams entwickelt wurden – gut zusammenarbeiten.
- **System Testing**: Testen des Systems als Ganzes. In der Regel wird beim Systemtest die *Funktionalität* überprüft.
- **Accepting Testing**: Wird vom Kunden durchgeführt, um zu überprüfen, ob das System seinen Ansprüchen genügt.
- **Regression Testing**: Regressionstests werden im Rahmen der Wartung eines Systems durchgeführt. Sie sollen sicherstellen, dass nach dem Hinzufügen von Features oder dem Ändern von Funktionen das zuvor bereits getestete Gesamtsystem seine Aufgaben nach wie vor korrekt durchführt. Normalerweise werden (wenn möglich) dazu bereits vorher durchgeführte Tests wiederholt.
- **Stress Testing**: Darunter versteht man das Testen eines Systems unter extremen Bedingungen, z. B. mit sehr vielen Daten oder einer hohen Zahl von Benutzern.

5.1.4 Unterscheidungskriterium Test Coverage

Da Testen ein zeitaufwendiger Vorgang sein kann, möchte man mit einer möglichst geringen Anzahl von Testfällen möglichst viele Fehler finden. Dazu gruppiert man Ausführungen in Klassen, die mit hoher Wahrscheinlichkeit gleiche Fehler produzieren. Aus jeder dieser Klassen testet man eine Ausführung. Je mehr Ausführungen eine Klasse enthält, um so weniger muss getestet werden. Normalerweise ist es jedoch so, dass eine höhere Anzahl kleinerer Klassen die Wahrscheinlichkeit, Fehler zu finden, erhöht, was mit höherem Testaufwand bezahlt werden muss.

Die folgende Auflistung beschreibt kurz die verschiedenen Punkte, die unter das Unterscheidungskriterium Test Coverage fallen.

- **Statement Coverage**: Jede ausführbare Anweisung des Programms (wie Zuweisung, Eingabe, Test, Ausgabe) erscheint in wenigstens einem Testfall. Im Falle von bedingten Verzweigungen (`If`-Statements) wird beim Statement Coverage möglicherweise nur der `true`- oder nur der `false`-Zweig ausgeführt.

- **Edge Coverage**: Jede ausführbare Kante eines Flowcharts erscheint in einem Testfall. Das bedeutet, dass im Falle von bedingten Verzweigungen (`If`- oder `While`-Anweisungen) sowohl die `true`- als auch die `false`-Zweige ausgeführt werden müssen.
- **Condition Coverage**: Bedingungen in einem Programm sind boolsche Kombinationen von einfachen Formeln erster Ordnung. Jede ausführbare Bedingung muss in mindestens einem Testfall erscheinen, wo sie den Wert `true` hat, und in mindestens einem Testfall mit Wert `false`. Beispiel: Befindet sich in einem Programm eine zusammengesetzte Bedingung $x = y \wedge z > w$, so könnte ein Testfall $x = 3$, $y = 3$, $z = 5$ und $w = 7$ setzen (also $x = y$: `true`, $z > w$: `false`). Ein anderer Testfall muss dann Sorge dafür tragen, dass $x = y$ `false` und $z > w$ `true` wird (also beispielsweise durch $x = 3$, $y = 4$, $z = 7$, $w = 5$).
- **Edge/Condition Coverage**: Kombination der beiden letzten Punkte.
- **Multiple Condition Coverage**: Ähnlich dem Condition Coverage. Allerdings muss jede mögliche Kombination in jeder zusammengesetzten Bedingung überprüft werden. Im obigen Beispiel müssen also vier Testfälle kreiert werden. Diese Art des Testens ist in der Praxis selten durchführbar, da die Anzahl der Testfälle schnell zu groß wird.
- **Path Coverage**: Jeder Ausführungspfad in einem Programm muss von einem Testfall abgedeckt sein. Ein Test nach diesem Kriterium ist nicht immer möglich, da die Zahl der Ausführungspfade riesig sein kann. So führen Schleifen in der Regel zu einer enormen Anzahl von möglichen Ausführungspfaden.
- **Loop Coverage**: Loop Coverage widmet sich insbesondere, wie der Name bereits vermuten lässt, den Schleifen. Verschiedene Teststrategien existieren, z. B.
  - Überprüfung des Falls, in dem eine Schleife nicht ausgeführt wird.
  - Überprüfung des Falls, in dem eine Schleife einmal ausgeführt wird.
  - Überprüfung des Falls, in dem eine Schleife eine typische Anzahl Wiederholungen durchläuft.

5.2 Dynamisches Testen

Beim dynamischen Testen werden Testfälle ausgeführt, die bestimmte Aspekte des Systems untersuchen. Dynamische Tests können im natürlichen Umfeld des Systems durchgeführt werden oder in einer Simulation desselben. Im Falle einer Simulation muss berücksichtigt werden, dass oft nicht alle Aspekte der untersuchten Software mit einbezogen werden können. Beispielsweise ist die Begutachtung des Zeitverhaltens von Komponenten in einer simulierten Umgebung nur schwer zu testen. Mit diesem Problem müssen sich insbesondere Hardware-Hersteller auseinandersetzen, die vor dem teuren Gießen der Baupläne in Silizium oft eine Simulation der entwickelten Hardware durchführen, um frühzeitig Fehler zu entdecken.

- **Funktionales Testen**: Wie der Name bereits sagt, werden beim funktionalen Testen alle Funktionen eines Systems identifiziert und getestet, die in den Anforderungen definiert sind. Funktionales Testen gehört zum Black-Box-Testing, da keinerlei Wissen über die Implementierung des Systems benötigt wird.
- **Strukturelles Testen**: Voraussetzung für das strukturelle Testen ist die detaillierte Kenntnis der internen Struktur des zu testenden Systems. So können als besonders kritisch angesehene Pfade im Programm getestet werden.
- **Statistisches, zufallsbasiertes Testen**: Bei dieser Art des Testens werden zufällig (gültige) Eingaben für das zu testende System generiert. Das erhöht die Chance, Fehler, die durch systematische Tests nicht gefunden wurden, dennoch zu Tage treten zu lassen.

- **Generierung von Testfällen**: Bevor mit dynamischem Testen begonnen werden kann, müssen Testfälle generiert werden. Testfälle bestehen aus
  - einem Eingabe-Vektor
  - einem Ausgabe-Vektor, der die bei einem erfolgreichen Test erwartete Ausgabe enthält
  - Vorbegingungen, die erfüllt sein müssen, damit ein Test erfolgreich durchgeführt werden kann
  - Nachbedingungen, die den Zustand beschreiben, in dem sich das System nach einem erfolgreichen Test befindet

  Um die Robustheit einer Software zu erhöhen, werden beim dynamischen Testen manchmal absichtlich die Vorbedingungen verletzt. Damit kann überprüft werden, wie sich ein System unter anormalen Bedingungen verhält.

  Programmierer sollten nicht ihren eigenen Code testen, weil sie (unbewusst) dazu tendieren, ihre Testfälle so zu konstruieren, dass wenig Fehler zu Tage treten. In der Praxis wird dieser Ratschlag jedoch häufig nicht befolgt.

5.3 Analyse/Statisches Testen

Beim statischen Testen, oft auch als Analyse bezeichnet, werden die Merkmale eines Systems bzw. seiner Komponenten ohne seine/ihre Ausführung begutachtet. Beispiele für das statische Testen gibt die folgende Auflistung.

- **Design Reviews**: Systematische Begutachtung der bei der Erstellung des Systems entstandenen Dokumente durch Ingenieure bzw. Software-Entwickler.
- **Code Inspection/Walkthrough**: Unter Code Inspection bzw. Walkthroughs versteht man eine geführte Tour durch den Quelltext. Dabei versuchen die Entwickler des inspizierten Codes, ihre Kollegen von der Richtigkeit des Programms zu überzeugen.
- **Checklisten**: Ein allgemeiner Fragenkatalog, der für die kritische Abschätzung und Begutachtung aller Aspekte eines Systems genutzt werden kann.
- **Formale Beweise**: Siehe Kapitel 3.2.1 und 3.2.2.
- **Kontrollflussanalyse**: Eine Analyseform zum Aufspüren von schlechten und potenziell inkorrekten Programmstrukturen. Mittels der Kontrollflussanalyse lassen sich beispielsweise unerreichbarer Code oder Endlosschleifen aufdecken.
- **Datenflussanalyse**: Analyse der Daten und ihrer Transformationen in einem System. Genutzt wird die Datenflussanalyse z. B. zur Aufspüren von Variablen, die beschrieben, aber nie gelesen werden.

5.4 Vorteile

Wie in diesem Kapitel gezeigt wurde, gibt es viele verschieden Formen und Arten des Testens. Alle haben ihre Daseinsberechtigung und ihre ganz spezifischen Vor- und Nachteile. An dieser Stelle sollen die wichtigsten noch einmal kurz zusammengefasst werden.

Die Technik des Testens ist wohl allen Soft- und Hardware-Entwicklern bekannt. Testen kann relativ einfach erlernt werden, auch wenn für viele Arten des Testens Erfahrung auf diesem Gebiet hilfreich ist. In der Regel besitzt das Testen ein gutes Kosten-/Nutzen-Verhältnis. Voraussetzung

dafür ist allerdings, dass bereits beim Entwurf des Systems auf die spätere Testbarkeit Rücksicht genommen wurde.

Ein wichtiger Vorteil des Testens ist die Anwendbarkeit auch bei komplexen Systemen mit komplexen Datenstrukturen. Da, wo andere Techniken zur Verbesserung der Qualität schon versagen, ist das Testen meist (wenn auch mit evtl. hohem Aufwand, z. B. wenn der Einsatz von Simulatoren notwendig ist) noch möglich.

5.5 Nachteile

Neben den wünschenswerten Eigenschaften hat das Testen eine Reihe von Nachteilen. Dynamische Tests (abgesehen vom nicht praktikablen Exhaustive Testing) können nicht eingesetzt werden, um die Fehlerfreiheit von Systemen nachweisen zu können. Abhängig von der Art und Komplexität des zu testenden Systems können Tests sehr aufwendig sein, insbesondere beim Einsatz von Environment Simulatoren.

Kapitel 6

Zusammenfassung

Die in den vorangegangenen Kapiteln beschriebenen Verfahren erhöhen allesamt die Qualität der entwickelten Software. In nicht sicherkeitskritischen Bereichen wird man meist mit verschiedenen Formen des Testens auskommen. Werden an die Qualität des entwickelten Systems jedoch höhere Anforderungen gestellt, muss geprüft werden, welche der beschriebenen Verfahren zur Fehlervermeidung und -suche eingesetzt werden können. Dabei ist eine Kombination verschiedener Techniken nicht nur möglich, sondern wünschenswert. Beispielsweise wird man trotz der Nutzung eines Model-Checkers bei der Verifikation einer Software nicht auf das Testen verzichten können.

Die formalen Techniken Model-Checking und Theorem Proving sind für nicht sicherheitskritische Bereiche oft zu aufwendig. Eine Kosten-/Nutzen-Analyse kann bei der Entscheidung über den Einsatz solcher Techniken helfen.

Alles in allem kann durch die beschriebenen Verfahren eine hohe Sicherheit der entwickelten Soft- und Hardware erreicht werden. Absolute Sicherheit kann jedoch keine der Techniken garantieren.

Literaturverzeichnis

[1] Stewart Gardiner. *Testing Safety-Related Software*. Springer-Verlag London Limited, 1999.

[2] Doron A. Peled. *Software Reliability Methods*. Springer-Verlag New York, 2001.

[3] Neil Storey. *Safety-Critical Computer Systems*. Prentice Hall, 1996.

[4] Dr. Holger Giese. *Software Engineering for Safety-Critical Computer Systems*. http://www.uni-paderborn.de/cs/ag-schaefer/Lehre/Lehrveranstaltungen/Vorlesungen/SafetyCriticalComputerSystems/WS0304/. Universität Paderborn, 2003.

[5] Prof. Dr.-Ing. habil. Wilhelm Dangelmaier. *Grundlagen der Informationstechnik von Produktions- und Logistiksystemen (GIP)*. http://wwwhni.uni-paderborn.de/cim/. Universität Paderborn, 2002.

[6] Ekkart Kindler. *Modelchecking*. http://www.uni-paderborn.de/cs/kindler/Lehre/WS03/MC/. Universität Paderborn, 2003.

[7] Patrick Grässle, Henriette Baumann, Philippe Baumann. *UML projektorientiert – Geschäftsprozessmodellierung, IT-System-Spezifikation und Systemintegration mit der UML*. Galileo Computing, 2000.