Klassen, die leere Zeichenreihe und eine mit mehr als 100 Zeichen; insbesondere 101 Zeichen sollten getestet werden.

Wenn wir vermuten, dass die Reihenfolge der Zeichen eine Rolle spielt, dass also »aB« möglicherweise anders behandelt wird als »Ba«, dann ergeben sich auch dadurch weitere Klassen. Und schließlich müssen wir überlegen, ob die Kombination der genannten Klassen neue Klassen ergibt oder nicht, ob wir also beispielsweise zwischen maximal langen Zeichenreihen mit und ohne Kleinbuchstaben unterscheiden müssen.

Das Beispiel zeigt: Die Äquivalenzklassen können nicht einfach mechanisch festgelegt werden, Intuition und Erfahrung sind von großer Bedeutung. Und es zeigt auch, dass wir niemals sicher sein können, alle Klassen der starken Äquivalenz erkannt zu haben. Wird die Zeichenreihe beispielsweise vom Programm in Abschnitte zu je 32 Zeichen gegliedert, dann muss man mit Problemen rechnen, wenn gerade 32, 64 oder 96 Zeichen eingegeben werden, auch 33, 65 und 97 wären dann speziell zu betrachten.

Ein letztes (nicht erfundenes) Beispiel zeigt, warum wir nie sicher sein können, alle Klassen der starken Äquivalenz entdeckt zu haben: Ein kleines ADA-Demonstrationsprogramm zur rekursiven Berechnung der Fakultätfunktion wurde mit den Mitteln des Exception Handlings gegen alle Fehleingaben abgesichert, also gegen Eingabe negativer oder gebrochener Zahlen und gegen die Eingabe von Text. Zusätzlich wurde sichergestellt, dass die Berechnung der Fakultät bei der Eingabe einer großen Zahl nicht zu einem Zahlenüberlauf führt. Mit den Eingaben -5, 3.14, Blabla und 1000 funktionierte alles einwandfrei. Trotzdem brachte ein naiver Benutzer das Programm sofort zum Absturz: Er hatte als Argument 1000000 eingetippt. Aber war dieser Fall nicht abgefangen und getestet worden (durch die Äquivalenzklasse »Zahl zu groß« mit dem Repräsentanten 1000)? Nein. Denn die Eingabe einer sehr großen Zahl führt, bevor es zum Zahlenüberlauf kommt, zu einem Kellerüberlauf (stack overflow). Eine Million Rekursionsstufen waren zu viel. Von der Klasse »Zahl zu groß« musste also eine Klasse »Zahl viel zu groß« abgespalten werden.

19.5 Der Black-Box-Test

Bei einem Black-Box-Test betrachtet man das Programm als Monolithen, über dessen innere Beschaffenheit man nichts weiß und nichts wissen muss. Man prüft, ob das Programm tut, was die Spezifikation verlangt. Dies ist die wichtigste Form des Tests. Alle anderen Tests ergänzen den Black-Box-Test, sie können ihn nicht ersetzen.

Testfälle für den Black-Box-Test sollten vorbereitet werden, sobald die Spezifikation vorliegt (siehe Punkt 10 in Abschnitt 16.9). Dadurch ist nicht nur gewährleistet, dass die Testfälle rechtzeitig verfügbar sind, sondern die Spezifikation wird einer zusätzlichen Prüfung unterzogen, die oft Fehler und Lücken

aufdeckt. Denn wer einen Testfall, also Eingabe und Soll-Resultat entwirft, muss genau hinschauen und kann nicht übersehen, wenn die Spezifikation seine Fragen nicht oder nicht eindeutig beantwortet.

19.5.1 Die Ziele des Black-Box-Tests

Ein umfassender Black-Box-Test sollte

- alle Funktionen des Programms aktivieren (Funktionsüberdeckung),
- alle möglichen Eingaben bearbeiten (Eingabeüberdeckung),
- alle möglichen Ausgabeformen erzeugen (Ausgabeüberdeckung),
- die Leistungsgrenzen ausloten,
- die spezifizierten Mengengrenzen ausschöpfen,
- alle definierten Fehlersituationen herbeiführen.

Die Funktionsüberdeckung geht von der Menge der Funktionen aus, die ein Programm anbietet. Die Eingabeüberdeckung ist auf die Menge möglicher Eingaben bezogen, die Ausgabeüberdeckung auf die Menge möglicher Ausgaben. Achtung, es geht hier nicht um alle im Detail unterschiedlichen Ein- und Ausgaben, also um vollständigen Test, sondern um die Klassen der Ein- und Ausgaben. Zum Beispiel fallen zwei Namen in dieselbe Klasse, wenn es nicht aus speziellen Gründen geraten erscheint, sie zu unterscheiden. Oft sind Ein- und Ausgabeüberdeckung durch die Funktionsüberdeckung impliziert, aber nicht immer.

Beispiel: Ein Programm, das Auskunft über Studenten gibt, biete die folgenden Funktionen: (a) Anfangsmeldung, (b) Einlesen einer Matrikelnummer, (c) Ausgabe der über den Studenten gespeicherten Informationen oder (d) einer Fehlermeldung (»Student existiert nicht«), (e) Endemeldung. Statt der Matrikelnummer kann auch der Name eingegeben werden. Wir können also alle Funktionen in Anspruch nehmen und dabei immer nur die Matrikelnummer verwenden. Wir haben dann die Funktionsüberdeckung, aber nicht die Eingabeüberdeckung erreicht.

Alle Funktionen außer d können wir mit einem einzigen Testfall abdecken. Fehlerfälle benötigen typischerweise jeweils einen eigenen Testfall, wenn der Programmablauf im Fehlerfall abgebrochen wird. Wenn ein Programm mit k verschiedenen Fehlerfällen enden kann, brauchen wir mindestens k+1 Testfälle, um die vollständige Funktionsüberdeckung zu erreichen; wenn sich verschiedene Funktionen gegenseitig ausschließen, sind es entsprechend mehr.

Die Grenzfälle (z. B. Namen minimaler und maximaler Länge sowie die angrenzenden Fehlerfälle) werden wie oben beschrieben speziell getestet.

19.5.2 Die Testfallauswahl für die Funktionsüberdeckung

Um die Testfälle für eine vollständige Funktionsüberdeckung zu gewinnen, kann man nach folgendem Schema vorgehen:

1. Suche in der Spezifikation der Anforderungen alle Funktionen und schreibe sie in eine Liste.
2. Suche in den Anforderungen alle Eingabe- und Ausgabegrößen.
3. Suche zu einer noch nicht ausgeführten Funktion aus der Liste die benötigten Eingabegrößen und die zu erzeugenden Ausgabegrößen.
4. Definiere für diese Funktion einen Testfall, d. h. die Eingabe und die erwartete Ausgabe. Markiere die von diesem Testfall ausgeführten Funktionen in der Liste.
5. Wiederhole die Schritte 3 und 4, bis alle Funktionen der Liste markiert sind.

Die Liste der Funktionen und der Markierungen, welche Funktionen in welchem Testfall ausgeführt werden, wird üblicherweise in Form einer Tabelle dargestellt, der sogenannten Funktionstestmatrix.

In den Abschnitten 19.8.2 bis 19.8.6 zeigen wir an einem etwas größeren Beispiel, wie die vorgestellten Black-Box-Techniken – Eingabe- und Ausgabeüberdeckung mit Hilfe von Äquivalenzklassen und die Funktionsüberdeckung – angewendet werden können.

19.5.3 Der zustandsbasierte Test

Endliche Automaten werden in der Informatik eingesetzt, um das Verhalten von Systemen zu spezifizieren. Formal ist ein endlicher Automat ein gerichteter Graph, dessen Knoten und Kanten Zustände und Zustandsübergänge darstellen. Zu jedem Zeitpunkt ist der Automat in genau einem Zustand; Eingaben lösen Zustandsübergänge aus.

Der Vorteil eines Zustandsautomaten ist seine Anschaulichkeit. Sowohl die Entwickler als auch die Benutzer haben intuitiv eine Vorstellung davon, welche Zustände das System durchläuft.

Der Zustand ist das Gedächtnis des Systems. Je nach Zustand kann eine Eingabe (ein Ereignis) unterschiedliche Effekte haben; eine Aktion kann ausgelöst werden, der Zustand kann sich ändern (Zustandsübergang). Zu Beginn ist der Automat im Startzustand; meist ist mindestens ein Zustand als Endzustand ausgezeichnet.

Betrachten wir als Beispiel eine sehr einfache Bankkontenverwaltung. Ein Konto kann eröffnet und geschlossen werden, es kann ein Betrag (b) gutgeschrieben und abgebucht werden, und der Kontostand kann abgefragt werden. Für jedes Konto sind der Kontostand (kst) und ein Überziehungslimit (limit) gespeichert. Ein Konto kann gesperrt werden, dann sind Gutschriften und Kontostandabfragen, aber keine Belastungen möglich; es kann, wenn es gesperrt ist, wieder entsperrt werden. Abbildung 19–5 zeigt einen Zustandsautomaten, der ein solches Konto modelliert.

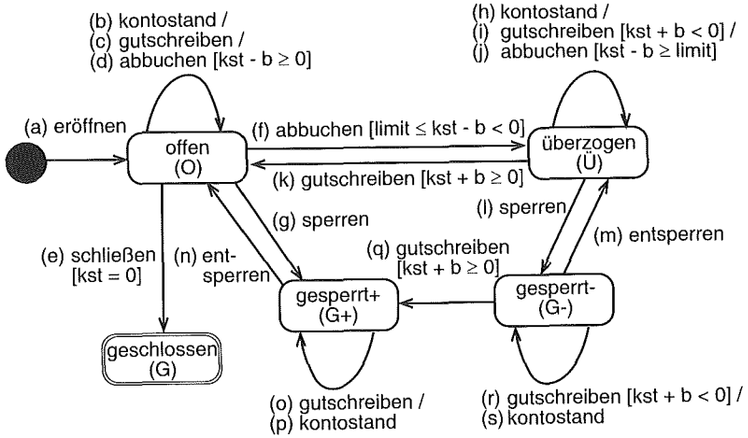

Abb. 19–5 *Beispiel für einen endlichen Automaten*

Zustandsautomaten werden im Software Engineering meist nur unvollständig beschrieben. Es gibt also eine Reihe von Ereignis-Zustand-Paaren, die nicht im Automaten aufgeführt sind. So wird im gezeigten Automat im Zustand »überzogen« das Ereignis »schließen« nicht betrachtet. Ein nicht spezifiziertes Ereignis-Zustand-Paar bedeutet, dass das Ereignis in diesem Zustand nicht erlaubt oder unmöglich ist.

Bei einem zustandsbasierten Test muss jeder Zustand mindestens einmal erreicht werden (*Zustandsüberdeckung*). Höhere Testgüte erzielt man, wenn von jedem Zustand aus in jeden möglichen Folgezustand gewechselt wird (im Beispiel in Abb. 19–5 etwa von Ü nach Ü, O und G-; *Zustandspaarüberdeckung*). Die Testgüte wird weiter gesteigert, wenn für jeden Zustandsübergang auch jedes Ereignis, das zum Zustandsübergang führt (im Beispiel die Ereignisse h, i und j für den Übergang von Ü nach Ü), im Test wirksam wird (*Transitionsüberdeckung*).

Um die Testfälle für den zustandsbasierten Test systematisch auszuwählen, werden in Binder (2000) sogenannte Roundtrip-Folgen durch den Zustandsautomaten bestimmt. Eine Roundtrip-Folge beginnt immer im Startzustand und endet in einem Endzustand oder in einem Zustand, der bereits in dieser oder einer anderen Roundtrip-Folge enthalten und dort nicht der letzte Zustand der Folge ist. Wenn der Test aller Roundtrip-Folgen keine Fehler zeigt, dann stimmt das Verhalten des Prüflings mit dem im Zustandsautomaten spezifizierten Verhalten überein. Der Test der Roundtrip-Folgen findet alle Zustandsübergangsfehler und deckt ggf. auf, dass Zustände fehlen.

Um die Roundtrip-Folgen zu ermitteln, wird aus dem Zustandsautomaten ein Zustandsübergangsbaum abgeleitet. Dies geschieht nach folgendem Algorithmus:

1. *Erzeuge den Wurzelknoten (Stufe 0 des Baums) und bezeichne ihn mit dem Startzustand. Setze die aktuelle Stufe k auf 0.*
2. *Bearbeite jeden Knoten der Stufe k:*

   Falls der Zustand, mit dem dieser Knoten bezeichnet ist, ein Endzustand ist oder bereits im Baum vorkommt, markiere den aktuellen Knoten als »terminal«.

   Andernfalls erzeuge für jeden ausgehenden Zustandsübergang und jede Bedingung eine neue Kante, die den aktuellen Knoten mit einem neuen Knoten auf Stufe k+1 des Baums verbindet. Bezeichne die neue Kante mit Ereignis/Bedingung/Aktion und den neuen Knoten mit dem Folgezustand.
3. *Falls es Knoten auf Stufe k+1 gibt, inkrementiere k und wiederhole Schritt 2.*

In unserem Beispiel wird aus dem Automaten in Abbildung 19–5 der Zustandsübergangsbaum in Abbildung 19–6 abgeleitet. (Abhängig von der Reihenfolge, in der die Knoten expandiert werden, entstehen unterschiedliche Übergangsbäume, die aber logisch äquivalent sind.)

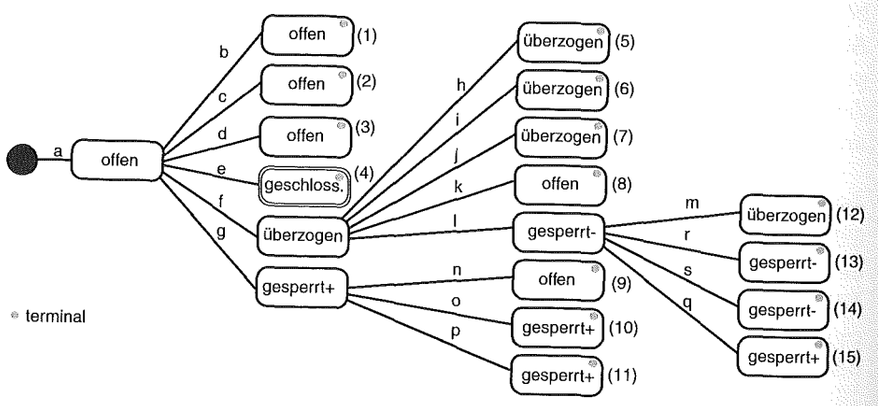

Abb. 19–6 *Zustandsübergangsbaum*

Um das durch den Zustandsautomaten spezifizierte Normalverhalten zu prüfen, müssen in unserem Beispiel Testfälle für 15 Roundtrip-Folgen entwickelt werden, je einer für jeden Weg von der Wurzel zu einem Blatt. Die Testfälle der folgenden Tabelle prüfen die drei Roundtrip-Folgen, die an den Blättern 7, 8 und 15 enden.

Folge	Startzustand, Ereignisse und Zwischenzustände	Endzustand, Soll-Wert kst
7	S, eröffnen → O, abbuchen (100) → Ü, abbuchen (200)	Ü, -300
8	S, eröffnen → O, abbuchen (100) → Ü, gutschreiben (500)	O, 400
15	S, eröffnen → O, abbuchen (100) → Ü, sperren → G-, gutschreiben (300)	G+, 200

Auch das Verhalten im Fehlerfall muss geprüft werden. Was soll beispielsweise passieren, wenn ein offenes Konto entsperrt wird oder der Kontostand durch eine Abbuchung unter das Überziehungslimit sinkt? Dazu benötigen wir für jede mögliche Kombination aus Zustand und Ereignis eine Angabe, wie der Automat reagieren soll. Dies kann recht übersichtlich in Tabellenform dargestellt werden. Die (evtl. durch Vorbedingungen erweiterten) Ereignisse bilden die Zeilen der Tabelle, die Zustände die Spalten. Die Tabelle wird gefüllt, indem für jedes Ereignis angegeben wird, ob es im betrachteten Zustand

- erlaubt (gültig) ist (√),
- ein Fehler ist, der behandelt werden muss (⊗),
- ausgeschlossen ist, da die Vorbedingung nicht erfüllt sein kann (–).

Für unser Beispiel entsteht damit die folgende Tabelle.

Ereignis / Vorbedingung	Reaktion im Zustand			
	O	Ü	G+	G-
kontostand	√	√	√	√
sperren	√	√	⊗	⊗
entsperren	⊗	⊗	√	√
schließen [kst = 0]	√	–	⊗	–
schließen [kst <> 0]	⊗	⊗	⊗	⊗
gutschreiben [kst + b ≥ 0]	√	√	√	√
gutschreiben [kst + b < 0]	–	√	–	√
abbuchen [kst - b ≥ 0]	√	–	⊗	–
abbuchen [limit ≤ kst - b < 0]	√	√	⊗	⊗
abbuchen [kst - b < limit]	⊗	⊗	⊗	⊗

Nun werden für alle ungültigen Ereignis-Zustand-Paare (markiert durch ⊗) Testfälle definiert. Beispielsweise prüft der folgende Test zur letzten Zeile der Tabelle die Reaktion auf einen Versuch, ein bereits überzogenes Konto stärker zu belasten, als das Überziehungslimit (hier vorgegeben mit -1000) erlaubt.

Startzustand, Ereignisse und Zwischenzustände	Erwartete Aktion	Endzustand, Soll-Wert kst
S, eröffnen → O, abbuchen(200) → Ü, abbuchen(900)	Fehlermeldung	Ü, -200

Jeder Prüfling, der sich als endlicher Automat beschreiben lässt, sei es eine Klasse, ein Modul oder das gesamte System, kann durch einen zustandsbasierten Test geprüft werden. Der Aufwand, um den Automaten, den Zustandsübergangs-

baum, die Ereignis-Zustand-Tabelle und die daraus abgeleiteten Testfälle zu erstellen, ist schon bei kleineren Automaten erheblich.

Nach jedem Schritt des zustandsbasierten Tests muss der erreichte Zustand kontrolliert werden. Dazu dienen spezielle Operationen, die man als *Zustandsreporter* bezeichnet. Falls es sie nicht bereits aus anderen Gründen gibt, müssen sie (als dauerhafte Erweiterung) realisiert werden; dazu muss der Quellcode des Prüflings verfügbar sein.

19.5.4 Der Test auf Basis der Anwendungsfälle

Testdaten werden aus der Spezifikation abgeleitet. Das gilt auch, wenn zur Spezifikation Anwendungsfälle verwendet werden. Dazu müssen neben den Anwendungsfalldiagrammen, die in UML notiert werden können, auch die natürlichsprachlichen Beschreibungen der Anwendungsfälle vorliegen.

Das Anwendungsfalldiagramm stellt alle Anwendungsfälle und ihre formalen Beziehungen dar. Andere wichtige Informationen – wie die Vor- und Nachbedingungen der Anwendungsfälle – werden im Diagramm nicht dargestellt. Diese Informationen sowie die einzelnen Interaktionsschritte sind in der detaillierten Beschreibung der Anwendungsfälle enthalten. Anwendungsfälle behandeln wir ausführlich in Abschnitt 16.7.3.

Das Überdeckungskriterium orientiert sich hier an den Anwendungsfällen; die Implementierung jedes Anwendungsfalls sollte in mindestens einem Test geprüft werden. Ebenso sollten alle Abhängigkeiten zwischen Anwendungsfällen (modelliert durch include- und extend-Beziehungen) im Test geprüft werden. Beim Test der einzelnen Anwendungsfälle stützen wir uns auf die detaillierten Beschreibungen.

In Abbildung 19–7 sind der Normalablauf und (verkürzt) die Sonderfälle aus dem in Abbildung 16–6 auf Seite 388 vorgestellten Anwendungsfall »Authentifizieren« wiedergegeben.

Normalablauf	1. Der Kunde führt eine Karte ein 2. Der BA42 liest d. Karte und sendet d. Daten z. Prüfung ans Banksystem 3. Das Banksystem prüft, ob die Karte gültig ist 4. Der BA42 zeigt die Aufforderung zur PIN-Eingabe 5. Der Kunde gibt die PIN ein 6. Der BA42 liest die PIN und sendet sie zur Prüfung an das Banksystem 7. Das Banksystem prüft die PIN 8. Der BA42 akzeptiert den Kunden und zeigt das Hauptmenü
Sonderfälle	2a Die Karte kann nicht gelesen werden 2b Die Karte ist lesbar, aber keine BA42-Karte 2c Das Banksystem ist nicht erreichbar 3a Die Karte ist nicht gültig oder gesperrt 5a Der Kunde bricht den Vorgang ab 5b Der Kunde reagiert nach 5 Sekunden nicht 6a Das Banksystem ist nicht erreichbar 7a Die erste oder zweite eingegebene PIN ist falsch 7b Die dritte eingegebene PIN ist falsch

Abb. 19–7 *Anwendungsfall »Authentifizieren« (Auszug)*

Auf Basis der Beschreibung des Anwendungsfalls können wir für jeden Ablauf einen Testfall entwickeln. Bei komplexen Interaktionen mit vielen Sonderfällen ist es jedoch angebracht, die Beschreibung vorher zu formalisieren. Wenn wir dazu eine grafische Notation verwenden, erhalten wir eine übersichtlichere Darstellung, und wir erkennen Fehler und Lücken leichter. Ryser (2003) schlägt für diesen Zweck Zustandsautomaten vor. Wir folgen Sneed und Winter (2001), die Aktivitätsdiagramme benutzen.

Ein Anwendungsfall wird komplett, mit Normalablauf und Sonderfällen, in ein Aktivitätsdiagramm übersetzt. Wir modellieren zuerst den Normalablauf und fügen dann Schritt für Schritt die Sonderfälle hinzu. Aus dem gezeigten Anwendungsfall können wir beispielsweise das in Abbildung 19–8 dargestellte Aktivitätsdiagramm entwickeln (der Normalablauf ist farblich hinterlegt, die verschiedenen Wege sind nummeriert).

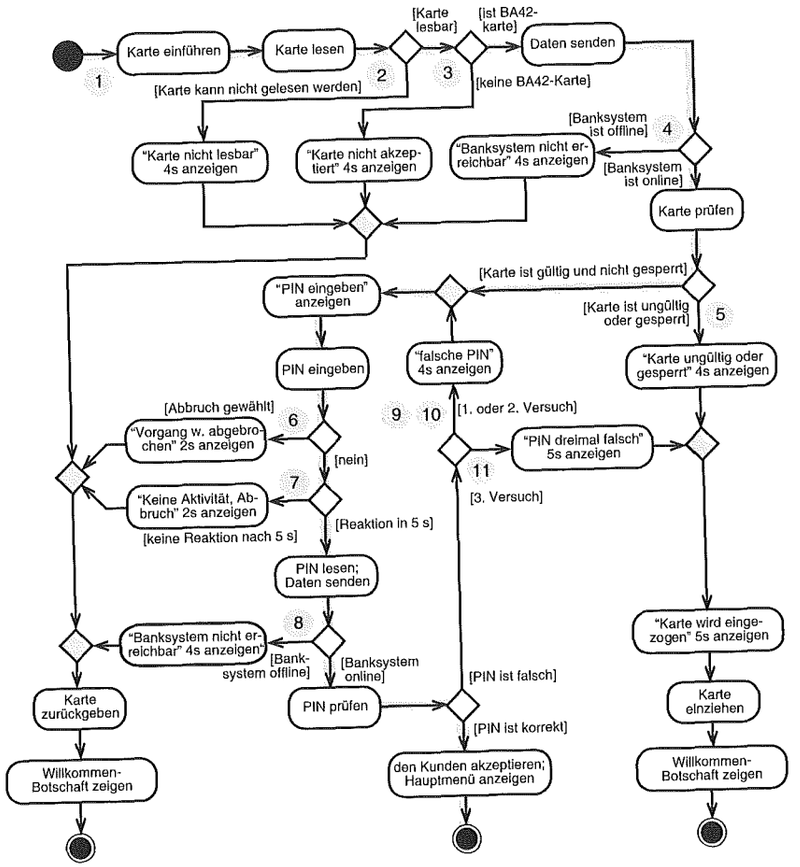

Abb. 19–8 *Aktivitätsdiagramm des Anwendungsfalls »Authentifizieren«*

Bei diesem Umsetzungsschritt stellt man möglicherweise fest, dass das im Anwendungsfall modellierte Verhalten unvollständig ist. Auch können einzelne Schritte zu grob formuliert sein, sie müssen dann detailliert werden. So muss der in unserer Use-Case-Beschreibung enthaltene Schritt 2 »Der BA42 liest die Karte und sendet die Daten zur Prüfung an das Banksystem« in die zwei Schritte »Karte lesen« und »Daten senden« aufgeteilt werden, damit die dazu angegebenen Sonderfälle präzise im Aktivitätsdiagramm modelliert werden können. Fehlende Abläufe und neue oder geänderte Schritte werden sowohl in das Aktivitätsdiagramm als auch in die Use-Case-Beschreibung integriert. Im Beispiel fehlt auch die Angabe, wie eine PIN genau aufgebaut ist. Diese Information wird im Test und natürlich auch für die Entwicklung benötigt, sie muss spezifiziert werden.

Beim Test auf der Basis von Anwendungsfällen wird jeder beschriebene Ablauf, also jeder Weg durch das Aktivitätsdiagramm, wenigstens einmal geprüft (Ablaufüberdeckung). Wir beginnen mit dem Normalablauf und prüfen dann die Sonderfälle. Auf diese Weise werden oft Fehler bei der Programmierung der Anwendungsfälle entdeckt, falsch oder unvollständig implementierte Anwendungsfälle und Abhängigkeiten zwischen den Anwendungsfällen, die nicht beachtet wurden.

Wie die Abbildung 19–8 zeigt, gibt es elf Wege durch das Aktivitätsdiagramm und damit elf Testfälle, um alle Wege zu durchlaufen. Diese Testfälle sind in der folgenden Tabelle 19–3 angegeben. Die Eingaben in eckigen Klammern erfolgen durch Mausklick, Knopfdruck, Menüauswahl o. Ä.; Reaktionen des BA42, die in einer Zelle stehen, geschehen gleichzeitig.

Weg	Anfangszustand	Eingabe/ Aktion	Soll-Reaktion Der BA42 ...
1	BA42-Karte mit PIN 1234	Karte einführen	meldet »PIN eingeben«
		1234 [OK]	zeigt das Hauptmenü an
2	Magnetstreifen der Karte ist beschädigt	Karte einführen	meldet »Karte nicht lesbar« (4 s)
			gibt die Karte zurück
			zeigt die Willkommen-Botschaft
3	Karte einer anderen Bank	Karte einführen	meldet »Karte nicht akzeptiert« (4 s)
			gibt die Karte zurück
			zeigt die Willkommen-Botschaft
4	BA42-Karte mit PIN 1234; Banksystem offline	Karte einführen	meldet »Banksystem nicht erreichbar« (4 s)
			gibt die Karte zurück
			zeigt die Willkommen-Botschaft
5	gesperrte Karte	Karte einführen	meldet »Karte ungültig oder gesperrt« (4 s); »Karte wird eingezogen« (5 s)
			zieht die Karte ein
			zeigt die Willkommen-Botschaft
6	BA42-Karte	Karte einführen	meldet »PIN eingeben«
		Abbruch [OK]	meldet »Vorgang wird abgebrochen« (2 s)
			gibt die Karte zurück
			zeigt die Willkommen-Botschaft

Weg	Anfangszustand	Eingabe/ Aktion	Soll-Reaktion Der BA42 ...
7	BA42-Karte	Karte einführen	meldet »PIN eingeben«
		> 5s warten	meldet »Keine Reaktion, Abbruch« (2 s)
			gibt die Karte zurück
			zeigt die Willkommen-Botschaft
8	BA42-Karte	Karte einführen	meldet »PIN eingeben«
		Banksystem offline schalten	
		3456 [OK]	meldet »Banksystem nicht erreichbar« (4 s)
			gibt die Karte zurück
			zeigt die Willkommen-Botschaft
9	BA42-Karte mit PIN 1234	Karte einführen	meldet »PIN eingeben«
		1212 [OK]	meldet »Falsche PIN« (4 s), »PIN eingeben«
		1234 [OK]	akzeptiert den Kunden; zeigt das Hauptmenu
10	BA42-Karte mit PIN 1234	Karte einführen	meldet »PIN eingeben«
		1212 [OK]	meldet »Falsche PIN«(4 s), »PIN eingeben«
		1111 [OK]	meldet »Falsche PIN«(4 s), »PIN eingeben«
		1234 [OK]	akzeptiert den Kunden; zeigt das Hauptmenü
11	BA42-Karte mit PIN 1234	Karte einführen	meldet »PIN eingeben«
		1212 [OK]	meldet »Falsche PIN« (4 s), »PIN eingeben«
		1111 [OK]	meldet »Falsche PIN« (4 s), »PIN eingeben«
		1235 [OK]	meldet »PIN 3x falsch« (5 s); »Karte wird eingezogen« (5 s)
			zieht die Karte ein
			zeigt die Willkommen-Botschaft

Tab. 19–3 *Testfälle für den Anwendungsfall »Authentifizieren«*

19.6 Der Glass-Box-Test

Das Wort »Glass Box« lässt erkennen, dass der Programmcode bei dieser Art des Tests sichtbar ist; das Programm ist wie eine Maschine im gläsernen Gehäuse, deren Arbeit man von außen nicht direkt beeinflussen, aber beobachten kann. In jedem Glass-Box-Test geht es darum, beim Testen bestimmte, auf den Code bezogene Überdeckungen zu erreichen. Beispielsweise kann das Ziel sein, eine Anweisungsüberdeckung von 80 % zu erzielen. Dann wird getestet, bis 80 % der ausführbaren Befehle im Programm mindestens einmal ausgeführt worden sind.

Um festzustellen, ob dieses Ziel erreicht ist, braucht man unbedingt ein Werkzeug, das den Prüfling instrumentiert, d. h. mit zusätzlichen Anweisungen ausrüstet, die die notwendigen Zählungen realisieren. Die Zählerstände werden über beliebig viele Programmläufe akkumuliert und auf Wunsch in einer anschaulichen Form angezeigt; beispielsweise können alle Teile des Programms, die noch nie ausgeführt wurden, mit einer speziellen Farbe markiert sein. Dann kann man untersuchen, wie Testfälle beschaffen sein müssten, um diese »weißen Flecken auf der Landkarte« zu beseitigen.

Zwei sehr populäre, aber falsche Vorurteile über den Glass-Box-Test wollen wir von Anfang an bekämpfen:

Für einen Glass-Box-Test muss man den Code lesen und verstehen.
Falsch!
Dann wäre der Glass-Box-Test selbst kleinerer Programme (einige tausend Zeilen) aus Aufwandsgründen nicht zu bewältigen. Man muss nur diejenigen Stellen analysieren, die bislang noch nicht zum gewählten Überdeckungsmaß beitragen. Meist reicht es aus, den *Zweck* des betreffenden Code-Abschnitts zu erkennen.

Beim Glass-Box-Test kommt es nur auf die Messung der Überdeckung an.
Falsch!
Bei jedem Test ist der Soll-Ist-Vergleich das Wichtigste, auch beim Glass-Box-Test. Denn es nützt gar nichts, wenn das Programm zwar mit allen erdenklichen Überdeckungsmaßen zu 100 % getestet wurde, aber falsche Resultate liefert.

Da wir im Glass-Box-Test mit instrumentierten Programmen arbeiten, ist die Regel verletzt, dass wir das Programm für den Test nicht verändern dürfen (Abschnitt 19.3.3). Darum müssen die Eingaben des Glass-Box-Tests auch mit dem nicht instrumentierten Programm verarbeitet werden. Weichen die Ergebnisse der beiden Tests (mit instrumentiertem und unverändertem Code) voneinander ab, dann hat das unterschiedliche Zeitverhalten oder die unterschiedliche Speicherbelegung der beiden Varianten Einfluss auf das Resultat (was meist einen Fehler anzeigt). Wir können den Glass-Box-Test natürlich auch komplett durchführen und am Ende ohne Instrumentierung des Programms wiederholen, um zu prüfen, ob die Instrumentierung die Resultate verändert hatte.

19.6.1 Überdeckungskriterien für den Glass-Box-Test

Bei einem Glass-Box-Test kann Anweisungs-, Zweig- oder Termüberdeckung angestrebt werden. Liggesmeyer (2002) behandelt in seinem Buch ausführlich auch Glass-Box-Testverfahren, die einigen analytischen Aufwand und spezielle Werkzeuge erfordern. Solche Verfahren kommen, wenn überhaupt, nur in sicherheitskritischen Anwendungen zum Einsatz.

- Die *Anweisungsüberdeckung* (Abschnitt 19.6.2) ist erreicht, wenn alle Anweisungen des Programms ausgeführt wurden.
- Die *Zweigüberdeckung* (Abschnitt 19.6.3) ist erreicht, wenn bei allen Verzweigungen des Programms alle möglichen Wege (Zweige) durchlaufen wurden.
- Die *Termüberdeckung* (Abschnitt 19.6.4) ist erreicht, wenn jeder logische Term, der den Ablauf in einer Verzweigung steuert, mit beiden möglichen Werten (TRUE und FALSE) wirksam geworden ist.

Diese Überdeckungen können mit Werkzeugen gemessen werden: Die Anweisungen, Zweige und Terme sind leicht zu erkennen und zu zählen, und sie können jeweils auch im Quellprogramm durch Zählanweisungen »instrumentiert« werden. Darum gibt es für alle gängigen Programmiersprachen Werkzeuge für die Anweisungs- und Zweigüberdeckung. Für die Termüberdeckung wären sie ebenso möglich.

Die Termüberdeckung impliziert die Zweigüberdeckung und diese – wenn das Programm keine pathologischen Eigenschaften (»toten Code«) aufweist – die Anweisungsüberdeckung.

In der Literatur wird meist auch die *Pfadüberdeckung* behandelt. Sie spielt aber aus Gründen, die in Abschnitt 19.6.6 erläutert werden, keine Rolle.

Die unterschiedlichen Überdeckungsarten können mit Hilfe eines Ablaufgraphen anschaulich dargestellt werden. Ein Ablaufgraph entspricht einem Flussdiagramm, bei dem alle unverzweigten Programmsequenzen durch Abstraktion zu einer einzigen Anweisung reduziert werden. Nur die Verzweigungen (IF, CASE, WHILE etc.) und Zusammenführungen des Ablaufs bleiben erhalten.

Als Beispiel für die nachfolgenden Erläuterungen der Auswahlkriterien dient die einfache Prozedur `BerechneBonus`, die auf Basis der Parameter `saldo`, `sparRate` und des ursprünglichen Bonuswerts (Parameter `bonus`) den neuen Bonuswert berechnet. Der Code der Prozedur und ihr Ablaufgraph sind in Abbildung 19–9 dargestellt. Eine kompliziertere Berechnung des Bonus in vielen Schritten hätte auf den Ablaufgraphen keine Auswirkung.

Da wir hier nur den Code betrachten, fehlt eine Spezifikation für das Beispiel; darum wirkt die Angabe der Soll-Resultate künstlich. In einem echten Test müssen die Soll-Resultate natürlich aus der Spezifikation abgeleitet werden.

```
procedure BerechneBonus (saldo, sparRate: Integer;
                         bonus: in out Integer) is
begin
  if (saldo > 1000) and (sparRate > 100) then --B1
    bonus := bonus * 5;                       --Sa
  end if;
  if (saldo > 5000) or (bonus > 50) then      --B2
    bonus := bonus * 2;                       --Sc
  end if;
end BerechneBonus;
```

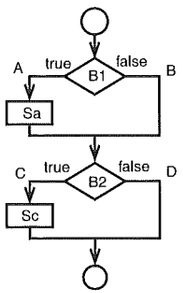

Abb. 19–9 *Prozedur BerechneBonus mit Ablaufgraph*

19.6.2 Die Anweisungsüberdeckung

Für die Anweisungsüberdeckung wird wie für alle Überdeckungsmaße nur der ausführbare Code betrachtet. Wenn der Prüfling keinen »toten« Code enthält, also Code, der nicht erreichbar ist, kann eine vollständige Anweisungsüberdeckung erzielt werden. In unserem Beispiel gelingt das durch einen einzigen Testfall.

Testfall	Eingaben			Anweisungen				Soll-Resultat bonus
	saldo	sparRate	bonus	B1	Sa	B2	Sc	
F1	4000	150	20	X	X	X	X	200

In der Praxis ist es schwierig, eine Anweisungsüberdeckung von mehr als 80 % bis 85 % zu erreichen. Einer höheren Überdeckung stehen zwei Schwierigkeiten entgegen:

- Programme enthalten defensiven Code, der nur ausgeführt wird, wenn in anderen Programmteilen Fehler auftreten. Formal gesehen handelt es sich dabei um toten Code; praktisch ist er sinnvoll, weil er dazu beiträgt, Wartungsfehler zu erkennen.
- In Programmen müssen auch sehr ungewöhnliche Fälle abgefangen und behandelt werden. Oft ist es sehr schwierig, solche Fälle im Test zu simulieren. Das gilt besonders für Abhängigkeiten von Datum und Uhrzeit; es ist außerordentlich riskant und darum nicht zu empfehlen, die Uhr des Systems für einen Test zu verstellen. Darum kann man Sonderbehandlungen für bestimmte Zeiten (z. B. für die Zeit vom 24. Dezember bis 1. Januar) kaum testen.

Entsprechende Einschränkungen gelten natürlich auch für alle anderen Überdeckungsmaße.

19.6.3 Die Zweig-, auch Entscheidungsüberdeckung

Bei der vollständigen Zweigüberdeckung wird jeder Zweig, d. h. jede Verbindung zwischen den Knoten des Ablaufgraphen, mindestens mit einem Testfall durchlaufen.

Die Zweigüberdeckung ist ein strengeres Kriterium als die Anweisungsüberdeckung, wenn der Prüfling leere Zweige enthält, also Zweige, die keine Anweisungen haben. Sie entstehen durch bedingte Anweisungen (IF-THEN ohne ELSE-Zweig) und durch Zweige einer CASE-Anweisung, die keine Aktionen enthalten.

Mit dem Testfall F1 erreichen wir in unserem Beispiel die leeren Zweige B und D nicht, erzielen also nur eine Zweigüberdeckung von 50 %. Durch einen zweiten Testfall können wir die volle Zweigüberdeckung erreichen.

Testfall	Eingaben			Zweige				Soll-Resultat bonus
	saldo	sparRate	bonus	A	B	C	D	
F1	4000	150	20	X		X		200
F2	100	20	2		X		X	2

19.6.4 Die Termüberdeckung

Durch eine vollständige Zweigüberdeckung stellen wir sicher, dass jeder Ausgang einer Verzweigung durchlaufen wurde. Wir wissen aber bei komplexen Bedingungen nicht, ob die einzelnen logischen Bedingungen wirksam geworden sind. Betrachten wir als Beispiel eine Verzweigung:

```
if (A AND (B OR (C AND D))) OR E then ...
```

Darin stehen A, B, C, D und E für einfache (atomare) logische Terme, also Variablen, Funktionen oder Ausdrücke des Typs Boolean. Die Zeile könnte also beispielsweise lauten:

```
if (Anfang AND (a=b OR (a>0 AND b>0))) OR Sonderfall then ...
```

Die vollständige Zweigüberdeckung ist bereits erreicht, wenn A (=Anfang) immer FALSE, E (=Sonderfall) einmal FALSE und einmal TRUE war. Wir haben also trotz Zweigüberdeckung keinen Hinweis darauf, ob alle *einzelnen* Terme sich so auswirken, wie es der Programmierer beabsichtigt hat. Um das zu prüfen, brauchen wir die Termüberdeckung.

Vollständige Termüberdeckung ist erreicht, wenn jeder Term mindestens einmal den gesamten Ausdruck FALSE und einmal TRUE gemacht hat.

Im Beispiel oben gilt das nur für den Term E. Ist A FALSE, so ist der Wert des gesamten Ausdrucks gleich dem Wert von E. E steuert in diesem Fall den Ablauf, E ist *wirksam*.

Wir definieren also: Ein Term ist dann wirksam, wenn die Änderung seines Wertes (von TRUE nach FALSE oder umgekehrt) dazu führt, dass sich auch der Wert des gesamten Ausdrucks ändert.

Das kann man für eine bestimmte Belegung der übrigen Terme einfach testen, indem man den Ausdruck zweimal auswertet (mit TRUE und FALSE für den betrachteten Term). Ändert sich dadurch der Wert des gesamten Ausdrucks, so ist dieser Term wirksam. Man kann die Wirksamkeit aber auch in einem Zuge für alle Terme des Ausdrucks feststellen, indem man den Baum, der den Ausdruck repräsentiert (Abb. 19–10), analysiert. Dabei gilt:

- Für eine bestimmte Belegung der atomaren Terme hat jeder Knoten den Wert TRUE oder FALSE und die Eigenschaft »wirksam« (w) oder »unwirksam« (u).
- Die Wurzel des Baumes (d. h. der logische Ausdruck insgesamt) ist wirksam.

Ein Unterknoten eines UND-Knotens ist genau dann unwirksam, wenn der UND-Knoten unwirksam oder dessen anderer Unterknoten FALSE ist.
Ein Unterknoten eines ODER-Knotens ist genau dann unwirksam, wenn der ODER-Knoten unwirksam oder dessen anderer Unterknoten TRUE ist.
Eine Negation (NOT) kehrt nur den Wert um, hat aber im Übrigen für die nachfolgende Betrachtung keine Bedeutung.

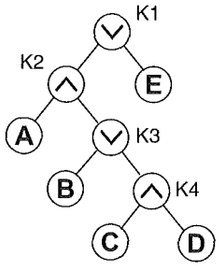

Abb. 19–10 *Baumdarstellung des logischen Ausdrucks (A AND (B OR (C AND D))) OR E*

Die Erklärung dazu ist einfach: Nur ein Wert, der sich »nach oben« auswirkt, ist im Sinne der Definition wirksam, nimmt also Einfluss auf den Ablauf des Programms.

In vielen Sprachen gibt es Operatoren mit Shortcut-Semantik (z. B. in ADA **and then / or else**, in JAVA && / ||). Dabei wird der rechte Knoten nicht ausgewertet, wenn der linke allein das Ergebnis bestimmt. In diesem Fall ist der linke unabhängig vom rechten so wirksam wie der übergeordnete Knoten. Nehmen wir an, dass im Beispiel K2 einen UND-Operator mit Shortcut-Semantik enthält und wirksam ist. Dann ist A mit dem Wert FALSE unabhängig von K3 wirksam.

Wir müssen den Baum für jeden Testfall also zweimal traversieren, einmal in Nachordnung, um die atomaren Terme mit Werten zu belegen und jedem UND- bzw. ODER-Knoten den Wert TRUE oder FALSE zuzuordnen, dann in Vorordnung, um jeden Knoten als wirksam oder unwirksam zu erkennen. So stellen wir für jeden atomaren Term fest, ob er wirksam ist und damit zur Termüberdeckung beiträgt.

Praktisch muss man also für jeden atomaren Term M zwei Wahrheitswerte M_F und M_T speichern; wird in einem Test festgestellt, dass M mit dem Wert FALSE (TRUE) wirksam ist, so wird M_F (M_T) auf TRUE gesetzt. Am Ende der Tests kann die Termüberdeckung als Anteil der auf TRUE gesetzten M_F und M_T abgelesen werden.

Wir können damit definieren: Ein Test erreicht die Termüberdeckung t, wenn bei den n atomaren Termen, die im Programm enthalten sind, $t \cdot 2n$ verschiedene wirksame Terme festgestellt wurden.

Beispiel zur Ermittlung der Termüberdeckung

Betrachten wir als Beispiel wieder den oben verwendeten Ausdruck:

```
if (A AND (B OR (C AND D))) OR E then ...
```

Vier Testfälle (beschrieben durch die jeweilige Belegung von A bis E) führen zu folgenden Belegungen und Überdeckungen (vgl. Abb. 19–10):

Testfall	Aspekt	A	B	C	D	E	K4	K3	K2	K1	wirksame Terme
1	Belegung	T	T	F	T	F	F	T	T	T	$\mathbf{A_T}$, $\mathbf{B_T}$
	Wirksamkeit	**T**	**T**	F	F	F	F	T	T	**T**	(alle neu)
2	Belegung	T	F	F	T	F	F	F	F	F	$\mathbf{B_F}$, $\mathbf{C_F}$, $\mathbf{E_F}$
	Wirksamkeit	F	**T**	**T**	F	**T**	T	T	T	T	(alle neu)
3	Belegung	T	F	T	T	F	T	T	T	T	A_T, $\mathbf{C_T}$, $\mathbf{D_T}$
	Wirksamkeit	**T**	F	**T**	**T**	F	T	T	T	T	(neu bis auf A_T)
4	Belegung	F	F	T	F	T	F	F	F	**T**	$\mathbf{E_T}$
	Wirksamkeit	F	F	F	F	**T**	F	F	F	T	(neu)

Jeder in einem Testfall wirksame atomare Term ergibt einen Beitrag 1 von $2 \cdot 5$, also 10 %. Testfall 1 erzielt 20 % Termüberdeckung, Testfall 2 30 %, Testfall 3 ebenfalls 30 %. A_T ist sowohl durch den Testfall 1 als auch durch den Testfall 3 abgedeckt. Wir erzielen also mit den Tests 1 bis 3 insgesamt 70 % Termüberdeckung.

Drei Fälle fehlen uns noch, nämlich A_F, D_F und E_T. Da es in den Tests 1 bis 3 gar keine entsprechende Belegung für A, D und E gab, setzen wir im Testfall 4 A und D auf FALSE, E auf TRUE. Das bringt uns aber nur einen Treffer, E_T, weil in dieser Belegung A und D unwirksam sind. Allgemein kann man leicht (durch Induktion) zeigen, dass für einen Ausdruck mit n atomaren Termen mindestens n+1 Fälle erforderlich sind, um vollständige Termüberdeckung zu erreichen.

Wir haben im Beispiel nur eine einzige Verzweigung betrachtet. In einem echten Programm mit mehreren Verzweigungen sinkt natürlich das Gewicht der einzelnen Verzweigungen. Will man mit der Termüberdeckung speziell die Verzweigungen mit komplexen Bedingungen, also mindestens zwei atomaren Termen, erfassen, zählt man nur diese (»Multitermüberdeckung«). Zählt man dagegen auch die einfachen Bedingungen mit (»if a < 0 then ...«), so bekommt man die Term-/Zweigüberdeckung; diese entspricht, wenn es keine Multiterme gibt, exakt der Zweigüberdeckung.

Diese Entsprechung zwischen Zweig- und Termüberdeckung gilt aber nicht, wenn das Programm Mehrfachverzweigungen (SWITCH- oder CASE-Anweisungen) enthält. Sie werden in der Zweigüberdeckung mitgezählt (bei n Ausgängen

mit dem Gewicht n), in der Termüberdeckung aber nicht, denn sie enthalten keine logischen Ausdrücke.

Enthält der Proband keine logischen Ausdrücke mit mehr als einem Term, dann wird die Termüberdeckung zur Zweigüberdeckung.

MC/DC Coverage

In der Literatur wird eine der Termüberdeckung sehr ähnliche Überdeckung als *modifizierte Bedingungs-/Entscheidungsüberdeckung* (modified condition/decision coverage) bezeichnet (Chilenski, Miller, 1994). MC/DC, ein wichtiges Kriterium bei der Prüfung von Software für die Luftfahrt, unterscheidet sich von der Termüberdeckung in folgenden Punkten:

- MC/DC ist nur für die vollständige Überdeckung definiert, nicht für teilweise Überdeckungen.
- Wie bei der Termüberdeckung muss jeder atomare Term sowohl mit dem Wert TRUE als auch mit dem Wert FALSE wirksam geworden sein. Bei MC/DC ist zudem gefordert, dass bei diesen beiden Tests alle übrigen atomaren Terme unverändert geblieben sind. Bei der Termüberdeckung genügt es, wenn der betrachtete Term in beiden Fällen wirksam war.
- Die Shortcut-Semantik bei der Auswertung logischer Ausdrücke (also der Abbruch einer Auswertung, wenn das Resultat feststeht) erfordert zusätzliche Festlegungen oder Änderungen in Richtung Termüberdeckung.

Unsere Termüberdeckung ist in diesen Punkten also etwas großzügiger (und damit einfacher) als die modifizierte Bedingungs-/Entscheidungsüberdeckung.

19.6.5 Schwierigkeiten bei der Zweig- und Termüberdeckung

Ein grundsätzliches Problem entsteht durch die Ausnahmebehandlung (Exception Handling). Die Anweisungüberdeckung erfordert nur, dass alle Exception Handlers ausgeführt werden. Weniger klar ist die Interpretation der Zweigüberdeckung: Wird die Ausnahme im Programm explizit ausgelöst, dann erfordert die Zweigüberdeckung, dass jede Auslösung auch getestet wird. Erfolgt die Auslösung aber an irgendeiner Stelle (etwa im Fall eines arithmetischen Überlaufs), so läuft diese Forderung ins Leere.

Wenn es zwischen verschiedenen Termen eine inhaltliche Kopplung gibt (weil auf dieselben Variablen Bezug genommen wird), kann es unmöglich werden, die Termüberdeckung zu erreichen. Wir haben eine ähnliche Situation vor uns wie mit der Anweisungsüberdeckung bei totem Code: Das Programm kann und sollte vereinfacht werden.

19.6.6 Die Pfadüberdeckung

Alle Überdeckungen, die im Glass-Box-Test gemessen werden können, betreffen elementare Bestandteile der Programme: Anweisungen, Verzweigungen und Bedingungen in den Verzweigungen; Pfade passen dazu nicht. Als Pfad bezeichnet man die vollständige Sequenz der Schritte von Anfang bis Ende einer Programmausführung; jede Ausführung eines Programms folgt genau einem Pfad. Die *Pfadüberdeckung* ist der Anteil der ausgeführten Pfade an der Gesamtzahl der Pfade.

Enthält ein Programm n Verzweigungen, aber keine Schleifen, dann gibt es darin höchstens 2^n Pfade. Durch Schleifen, die Verzweigungen enthalten, explodiert die Zahl der Pfade. Gibt es im Programm eine WHILE-Schleife, so ist die Zahl der Pfade im Allgemeinen nicht definiert, weil die Zahl der Iterationen nicht festgelegt ist; aber auch eine FOR-Schleife, die genau einhundertmal durchlaufen wird und eine Verzweigung enthält, kann bis zu 2^{100} ($> 10^{30}$) Pfade enthalten.

Allerdings ist in der Regel nur ein kleiner Teil dieser Pfade tatsächlich erreichbar, denn die Bedingungen in den Verzweigungen und Schleifen sind meist voneinander abhängig. Hängt in der oben beschriebenen FOR-Schleife die Verzweigung im Rumpf von der Laufvariablen i ab (z. B. davon, ob i eine gerade Zahl ist), dann gibt es nicht 2^{100} Pfade, sondern nur einen einzigen. Da die Frage, ob ein bestimmter Pfad erreichbar ist oder nicht, zu den nicht entscheidbaren Problemen gehört, haben wir keine Möglichkeit festzustellen, auf welche Gesamtheit die Pfadüberdeckung bezogen werden muss. Sie ist also nur in Spezialfällen messbar.

Wenn wir Äquivalenzklassen bilden, fragen wir uns, welche Eingabedaten das genau gleiche Verhalten des Programms auslösen, d. h. denselben Pfad verwenden. Darum ist die Ausführung jedes möglichen Pfades gleichbedeutend mit dem Test eines Repräsentanten aus jeder Äquivalenzklasse. Da wir die Pfadüberdeckung nicht messen können, bringt uns dieser Begriff keinen Vorteil.

19.6.7 Die Behandlung von Schleifen und die einfache Pfadüberdeckung

Nachdem sich gezeigt hat, dass die Pfadüberdeckung für die Praxis untauglich ist, stellt sich die Frage, ob das Kriterium so verändert werden kann, dass es anwendbar und nützlich wird. Man kann sich auf ein viel bescheideneres Überdeckungskriterium zurückziehen, das auf einer pragmatischen Behandlung der Schleifen beruht. Beizer (1983) empfiehlt, Schleifen wie folgt zu testen:

1. Umgehung der Schleife (Rumpf wird nicht ausgeführt)
2. eine Iteration (Rumpf wird genau einmal ausgeführt)
3. zwei Iterationen (Rumpf wird genau zweimal ausgeführt)
4. eine typische Anzahl von Iterationen
5. maximale Anzahl der Iterationen

Natürlich liegt hier eine Heuristik zu Grunde, die nicht zwingend sinnvoll ist. In vielen Fällen sind auch gar nicht alle fünf Möglichkeiten gegeben, beispielsweise dann nicht, wenn es sich um eine Laufschleife mit fester Wiederholungszahl handelt.

Auf dieser Grundlage wurde die sogenannte *einfache Pfadüberdeckung* definiert. Sie beruht auf dem Ansatz, bei Schleifen nicht zwischen Pfaden zu unterscheiden, die sich nur durch die Zahl der Iterationen unterscheiden. Wir müssen also nur die Pfade im Inneren des Rumpfes analysieren und im Übrigen nur die beiden Fälle 1 und 4 (oder statt 4 den Fall 2, 3 oder 5) aus der Liste oben behandeln.

Leider ist das fundamentale Problem damit nicht gelöst, wir kennen nach wie vor die Gesamtheit der möglichen Pfade nicht. Das wird verständlich, wenn man das Beispiel in Abbildung 19–9 so verändert, dass die beiden bedingten Anweisungen von wesentlich komplizierteren Ausdrücken abhängen, die sich auf die gleichen Variablen des Programms stützen. Nehmen wir an, dass in tausend Testfällen der Pfad A-D nie durchlaufen wurde. Dann liegt die Vermutung nahe, dass dieser Pfad wegen der Datenabhängigkeiten nicht erreichbar ist. Beweisen können wir das im Allgemeinen nicht. Wir wissen darum nicht, ob wir ohne Durchlaufen von A-D 75 % oder 100 % einfache Pfadüberdeckung erreicht haben.

19.6.8 Überdeckungskriterien für Programmkomponenten

Betrachtet man nicht einzelne Befehle, sondern Befehlsgruppen, z. B. Prozeduren, so kann man die Überdeckungskriterien des Glass-Box-Tests auf solche Gruppen übertragen. Damit entstehen die folgenden Makro-Überdeckungskritierien:

Programmeinheiten-Überdeckung
Jede Programmeinheit wird mindestens einmal aufgerufen.

Aufrufüberdeckung
Jeder aufrufbare Teil der Einheiten eines Programms wird wenigstens einmal aufgerufen (z. B. jede Methode einer Klasse).

Programmpfad-Überdeckung
Jede mögliche Ausführungssequenz der Programmeinheiten wird mindestens einmal durchlaufen.

Die Einwände gegen die Pfadüberdeckung gelten natürlich auch für die Programmpfad-Überdeckung.

19.7 Testen mit Zufallsdaten

Es ist verlockend, die mühsame Auswahl von Testfällen durch die Verwendung von Zufallswerten überflüssig zu machen, also die Eingaben nicht aufwändig auszuwählen, sondern von einem Programm erzeugen zu lassen. Selbst wenn Testfälle, die auf diese Weise generiert wurden, weniger erfolgreich sind als systematisch gewählte, kann man diesen Nachteil durch eine wesentlich höhere Zahl von Testfällen ausgleichen, denn die Kosten pro Testfall sind wesentlich geringer.

In der Praxis gibt es gegen diesen Ansatz, der sich von den oben vorgestellten radikal unterscheidet, drei Einwände:

- Es ist sinnlos, mit völlig beliebigen Mausklicks und Tastendrücken einen Benutzer simulieren zu wollen; die Testeingaben müssen eine gewisse Ähnlichkeit mit denen eines echten Benutzers haben. Entsprechendes gilt auch für Systeme, die in einen technischen Prozess eingebettet werden. Darum ist es notwendig, ein Benutzungsprofil zu entwickeln, also eine statistische Beschreibung der Einwirkungen von außen auf die Software. Die Zufallseingaben werden dann so generiert, dass ihre statistischen Eigenschaften der Vorgabe entsprechen. Die Entwicklung eines solchen Profils und eines Testdatengenerators ist aber in der Regel sehr aufwändig.
- Zu einem Testfall gehört ein Soll-Resultat. Bei einem Test, dessen Eingaben automatisch erzeugt werden, sollte auch die Auswertung des Tests automatisch erfolgen. Darum wird ein *Orakel* benötigt, also ein Programm, das entscheiden kann, ob das Resultat richtig ist. Natürlich ist es im Allgemeinen keineswegs klar, wie ein solches Orakel diese Entscheidung treffen kann.
- Punktfehler und Bereichsfehler mit kleinem Bereich werden sehr wahrscheinlich nicht gefunden. Sie können aber mit hohen Risiken verknüpft sein.

Vermutlich ist es vor allem der erste Punkt, der verhindert, dass der Test mit Zufallsdaten große Bedeutung erlangt. Die Literaturlage ist dünn; immer wieder wird eine Publikation von Duran und Ntafos (1984) zitiert, die allerdings Beispiele verwenden, in denen das Orakel trivial ist (Sortierung oder Suche). Einige Artikel zu diesem Thema findet man über die Webseite von Robinson (o.J.), darunter ist auch ein Artikel von Nyman (o.J.), der über eine praktische Anwendung bei Microsoft berichtet; er spricht beim Test mit Zufallswerten von »Monkey Testing«.

Im Cleanroom-Prozess (Abschnitt 10.5) ist diese Form des Tests als Standardverfahren vorgesehen. Allerdings steht dabei nicht die Fehlersuche im Vordergrund; der Test liefert primär eine Aussage zur Zuverlässigkeit des Programms (siehe Abschnitt 10.5.2, Teil *Der statistische Test*).