Teil III

Implementationsorientiertes Testen

In diesem Teil werden die in Abschnitt 3.4.3 nur kurz erwähnten Testansätze für die implementationsorientierte Teststrategie ausführlich behandelt.
Diese Testansätze orientieren sich am Kontrollfluß, am Datenfluß oder an den Ausdrücken bzw. Anweisungen im Programm.
Für jeden Testansatz werden folgende Aspekte behandelt, die mit Kosten und Nutzen zu tun haben:

- Vergleich bezüglich Zahl und Art der erforderlichen Testdaten
- Aufwand für die Erzeugung einzelner Testdaten
- Bewertung der Teststrategien (welche und wieviel Fehler werden aufgedeckt)
- Messung der Testwirksamkeit
- Erzeugung von Testdaten
- Erreichbarkeit einer hundertprozentigen Testwirksamkeit

Die Kenntnis und gezielte Anwendung der Methoden dieses Teils des Buches ist für einen gründlichen Test von Software unverzichtbar, denn ein spezifikationsorientierter Test kann nicht alle Fehler im Programm entdecken.
Das datenbereichsbezogene Testen würde beispielsweise ausreichen, wenn die Eingabewerte in den dort ermittelten Äquivalenzklassen tatsächlich bzgl. der Fehleraufdeckungsfähigkeit äquivalent wären. Das setzt voraus, daß Testdaten, die als äquivalent betrachtet werden, auch vom Programm äquivalent bearbeitet werden. Das heißt aber, daß die Programmierer die Anforderungsspezifikation bzw. die Entwurfsspezifikation richtig umgesetzt haben — was aber gerade zu prüfen ist.
Außerdem werden beim datenbereichsbezogenen Testen nur Stichproben aus den Äquivalenzklassen getestet; für gewisse Werte, die nur durch Zufall getestet werden, kann das Programm von der Spezifikation abweichen. Das kann ein unabsichtlicher Fehler sein oder sogar ein absichtlicher Fehler. Das Vergessen oder falsche Hinschreiben von Anweisungen sind unabsichtliche Fehler. Ein absichtlicher Fehler ist beispielsweise die folgende, nicht spezifizierte Abfrage:

if Personalnummer = 99007 **then** überweise-Geld-in-die-Schweiz

Dies ist ein Fall von Computerkriminalität. Solche Fehler können natürlich nur sicher entdeckt werden, wenn Testdaten erstellt werden, die aus der Programmstruktur abgeleitet werden.

7 Kontrollflußbezogenes Testen

Beim kontrollflußbezogenen Testen orientiert sich die Testdatenerzeugung an den Kontrollflußgraphen der Prozeduren und Funktionen, die im Programm bzw. Modul vorkommen. (Im folgenden wird in beiden Fällen von „Programm“ gesprochen.)
Das kontrollflußbezogene Testen setzt also voraus, daß die zu testenden Teile als Kontrollflußgraph vorliegen bzw. in diese Form transformiert werden.

7.1 Kontrollflußgraphen und ihre Eigenschaften

DEFINITION 7.1.1 (KONTROLLFLUSSGRAPH, KONTROLLFLUSSSCHEMA)

1. *Der* **Kontrollflußgraph** *eines Programms ist ein gerichteter Graph. Seinen Knoten sind (als Knotenmarkierung) Anweisungen oder Entscheidungsprädikate des Programms zugeordnet und seine Kanten stellen die Verbindungen zwischen Anweisungen und Anweisungsnachfolgern dar.*
   Dabei können ganze Anweisungsfolgen, die keine bedingten Anweisungen enthalten, einem Knoten zugeordnet sein[1]*. Das komplette Entscheidungsprädikat einer bedingten Anweisung ist dagegen getrennt von dem Rest der bedingten Anweisung einem sogenannten* **Entscheidungsknoten** *zugeordnet, wobei für jeden Ausgang der Entscheidung eine Kante zu einem Nachfolgerknoten existiert, die mit* ja (true) *bzw.* nein (false) *markiert ist. Diese Kanten heißen* **Entscheidungskanten**. *Wenn eine Kante von Knoten k zu Knoten l führt, heißt l* **Nachfolger(knoten)** *von k und k* **Vorgänger(knoten)** *von l. Um Initialisierungsvorgänge (z. B. Parameterübergabe bei Prozeduren) modellieren zu können, enthält ein Kontrollflußgraph nur einen Knoten (genannt:* **Anfangsknoten**)*, der keinen Vorgängerknoten hat und auch nur eine ausgehende Kante, genannt* **Anfangskante**, *besitzt. Außerdem soll ein Kontrollflußgraph nur einen Knoten, genannt* **Endknoten**, *enthalten, der keinen Nachfolger hat*[2].

2. *Das* **Kontrollflußschema** *zu einem Kontrollflußgraphen entsteht durch Ersetzung der Anweisungen und Entscheidungsprädikate durch formale Bezeichner.*

[1] Das ist für die Betrachtung des Kontrollflusses in diesem Kapitel ausreichend, nicht aber bei den hieraus abgeleiteten Datenflußgraphen (genaueres siehe Kapitel 8, Def. 8.1.1).

[2] Treten bei einer naheliegenden Modellierung mehrere Endknoten (Knoten ohne Nachfolger) auf, können diese immer durch eine weitere Kante mit *einem* (neuen) Endknoten verbunden werden bzw. selbst zu einem Knoten vereinigt werden, falls ihnen keine verschiedenen Programmanweisungen zugeordnet sind.

Falls es auf die Unterscheidung nicht ankommt, wird in beiden Fällen (1 und 2) von **Kontrollflußgraphen** *gesprochen.*

Die folgenden Eigenschaften von Kontrollflußgraphen sind für die Aufstellung von Testkriterien von Interesse, die sich an den Entscheidungen im Programm orientieren.

DEFINITION 7.1.2
Ein **Entscheidungs-Entscheidungs-Weg** *ist ein Wegstück, welches bei einem Entscheidungsknoten oder dem Anfangsknoten beginnt und alle folgenden Knoten und Kanten bis zum nächsten Entscheidungsknoten bzw. bis zum Endknoten des Kontrollflußgraphen (einschließlich) enthält.*

Dies ist eine Folge von Knoten und Kanten mit folgender Eigenschaft:

SATZ 7.1.1
Wird die erste Kante eines Entscheidungs-Entscheidungs-Weges w ausgeführt, so werden alle Knoten und Kanten dieses Wegs w ausgeführt.

Verschiedene Entscheidungs-Entscheidungs-Wege können sich überlappen, und zwar zwischen einem **Vereinigungsknoten**, in den mehr als eine Kante führt, und dem nächsten Entscheidungsknoten. Daher sind die überschneidungsfreien Teilwege von Entscheidungs-Entscheidungs-Wegen von Interesse, die also keine gemeinsamen Kanten enthalten.

DEFINITION 7.1.3
Ein **Segment** *ist ein Wegstück mit folgenden Eigenschaften:*

1. *Der erste Knoten des Wegstücks ist der Anfangsknoten des Kontrollflußgraphen oder ein Entscheidungsknoten oder ein Vereinigungsknoten.*
2. *Der letzte Knoten ist der Endknoten des Kontrollflußgraphen oder ein Entscheidungsknoten oder ein Vereinigungsknoten.*
3. *Alle anderen Knoten haben nur eine Eingangs- und eine Ausgangskante.*

Die Begriffe seien an folgendem Beispiel erläutert.

BEISPIEL 7.1.1
Das Kontrollflußschema aus Abbildung 7.1 hat die Knoten A, B, C, D, E, F, G, wobei A Anfangsknoten und G Endknoten ist. C und D sind Entscheidungsknoten. Bei C handelt es sich um einen Entscheidungsknoten für eine while-Schleife, bei D um einen Entscheidungsknoten für eine if-then-Fallunterscheidung mit leerem else-Zweig. Entscheidungsknoten für for-Schleifen, repeat-until-Schleifen oder case-Anweisungen kommen im Beispiel nicht vor, sind aber mögliche Entscheidungsknoten.

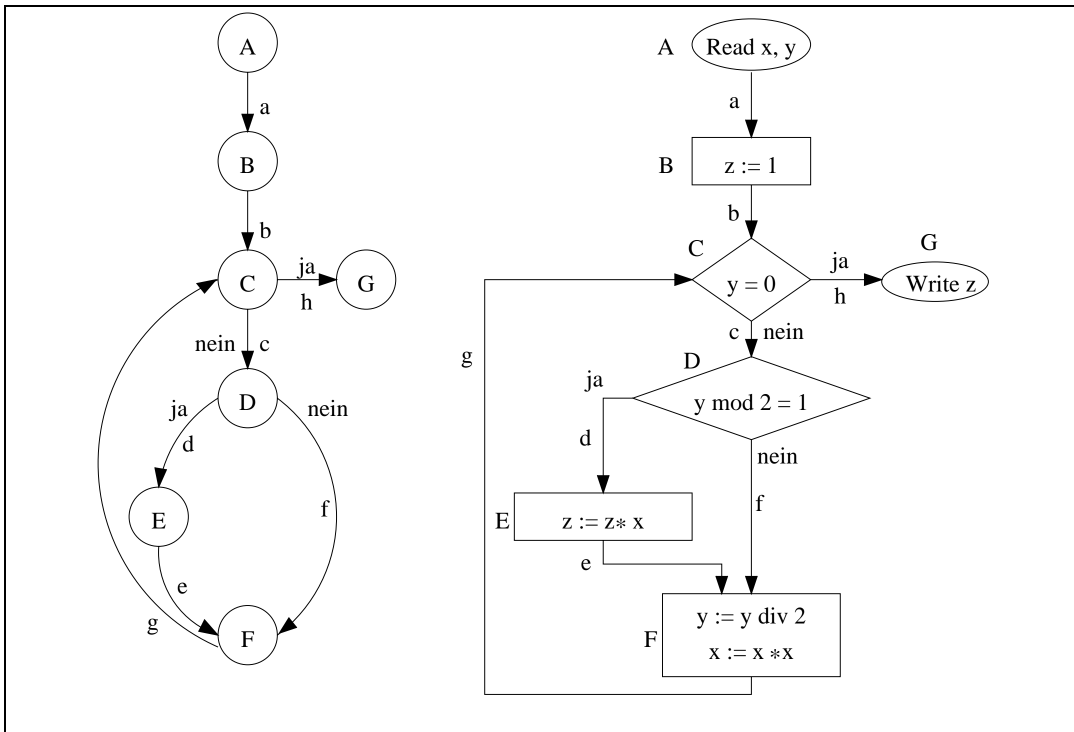


Abb. 7.1: Kontrollflußschema (links) und Kontrollflußgraph (rechts) zur Berechnung von $z = x^{|y|}$ für ganzzahliges y

Zur Verdeutlichung sind beim Kontrollflußgraphen die Entscheidungsknoten als Rauten, Anfangs- und Endknoten als Ellipsen und die anderen Knoten als Rechtecke dargestellt. Diese graphische Darstellung heißt auch **Flußdiagramm**. *Zwecks besserer Vergleichbarkeit mit dem Kontrollflußgraphen sind die Knoten hier zusätzlich mit den formalen Bezeichnern des Kontrollflußgraphen markiert.*

Kontrollflußgraph und Kontrollflußschema haben die Kanten a, b, c, d, e, f, g, h, wobei c, h, d, f Entscheidungskanten sind.

Es gibt folgende Entscheidungs-Entscheidungs-Wege:

E_1: A, B, C (bzw. als Kantenfolge: a, b)
E_2: C, G (bzw. h)
E_3: C, D (bzw. c)
E_4: D, E, F, C (bzw. d, e, g)
E_5: D, F, C (bzw. f, g)

Als Teilwege der Entscheidungs-Entscheidungs-Wege gibt es folgende Segmente:

S_1: A, B, C (bzw. als Kantenfolge: a, b)
S_2: C, G (bzw. h)

S_3: C, D *(bzw. c)*
S_4: D, E, F *(bzw. d, e)*
S_5: D, F *(bzw. f)*
S_6: F, C *(bzw. g)*

Aus E_4 und E_5 wird also der gemeinsame Teilweg S_6 bzw. die Kante g abgetrennt.

Die folgenden Eigenschaften und Komponenten von Kontrollflußgraphen sind für die Aufstellung von Testkriterien von Interesse, insbesondere wenn sie sich an den Schleifen im Programm orientieren.

Definition 7.1.4 ([einfacher] Zyklus, vollständiger Weg, Wege(T))

1. *Ein* **Zyklus** *ist ein Weg im Kontrollflußgraphen mit mindestens zwei Knoten, der an demselben Knoten beginnt und endet.*
2. *Ein* **einfacher Zyklus** *ist ein Zyklus, bei dem alle Knoten (außer dem Anfangs- und Endknoten) verschieden sind.*
3. *Ein* **vollständiger Weg** *(durch das Programm) ist ein Weg vom Anfangsknoten zum Endknoten des Kontrollflußgraphen des Programms oder ein unendlich langer Weg, der beim Anfangsknoten beginnt.*
4. *Für eine Menge T von Testdaten ist* **Wege(T)** *die Menge der vollständigen, endlichen Wege w des Kontrollflußgraphen, für die es ein Testdatum t aus T gibt, das den Weg w ausführt. Wenn zusätzlich die Abhängigkeit von Programm P ausgedrückt werden soll, wird Wege(T) mit* **Wege(T, P)** *bezeichnet.*
   $\mathbf{Wege}_\infty(\mathbf{T})$ *bzw.* $\mathbf{Wege}_\infty(\mathbf{T}, \mathbf{P})$ *bezeichnen die entsprechenden Wegemengen, bei denen auch* unendlich *lange Wege erlaubt sind.*

Beispiel 7.1.2
Abbildung 7.1 enthält die einfachen Zyklen c, d, e, g und c, f, g und z. B. den (nicht einfachen) Zyklus c, d, e, g, c, f, g. Die Kantenfolgen $f_1 = a, b, h$ und $f_2 = a, b, c, d, e, g, h$ sind vollständige Wege, die unendliche Folge $a, b, c, d, e, g, c, d, e, g, \ldots$ ist ebenfalls ein vollständiger Weg, bei dem sich c, d, e, g unendlich oft wiederholt. Für die Testdatenmenge $T = \{t_1, t_2\}$ mit $x = 0$, $y = 0$ bei t_1 und $x = 0$, $y = 1$ bei t_2 gilt Wege(T) $= \{f_1, f_2\}$, da bei $x = 0$, $y = 0$ die Kantenfolge $f_1 = a, b, h$ ausgeführt wird (und $f_2 = a, b, c, d, e, g, h$ bei $x = 0$, $y = 1$).

Im folgenden wird vorausgesetzt, daß die Kontrollflußgraphen strukturiert sind. Bei **strukturierten** Kontrollflußgraphen haben alle Schleifen genau einen Eingangsknoten (der zur Schleife gehört) und genau einen Ausgangsknoten (der nicht zur

Schleife gehört); die Schleifen überlappen sich nicht teilweise, d. h. bei sich überlappenden Schleifen ist eine Schleife vollständig in der anderen enthalten[3].

Bei strukturierten Kontrollflußgraphen bzw. Programmen werden folgende Arten unterschieden: **stark strukturierte** Programme, die nur die Sequenz, die Alternative und die kontrollierte Iteration (*while*- oder *repeat*-Schleife) als Kontrollstrukturen zulassen (vgl. Kapitel 4.3), und **(schwach) strukturierte** Programme, bei denen ein Sprung („*exit*") aus einer Schleife heraus (zum direkten Nachfolgerknoten[4]) erlaubt ist, wobei die Schleife auch ohne kontrollierendes Prädikat (*loop*-Schleife[5]) sein darf.

Beispiel 7.1.3 (strukturiertes Programm)
Der Kontrollflußgraph aus Abbildung 7.2 beschreibt ein (schwach) strukturiertes Programm (vgl. Abbildung 7.3, rechts). Es ist eine Sequenz aus den Anweisungen (Knoten) 0, 1, einer loop-Schleife und Anweisung 11. Die loop-Schleife ist eine Sequenz aus Knoten 2 und einer Alternative mit Entscheidungsknoten 3 bzw. P1, ja-Nachfolger 4 (mit exit nach 11) und einer weiteren Alternative als nein-Nachfolger: Entscheidungsknoten 5 bzw. P2 mit ja-Nachfolger 6 und exit von 7 nach 11 und nein-Nachfolger „Alternative von 9 und 10 mit Entscheidungsknoten 8 bzw. P3".

Strukturierte Kontrollflußgraphen sind **wohlgeformt**, d. h. daß es für jeden Knoten k einen Weg vom Anfangsknoten zum Endknoten gibt, auf dem Knoten k liegt. Es gibt also keine unerreichbaren Knoten im Kontrollflußgraphen und auch keine Schleifen, die keinen Ausgang besitzen[6].

Bei einer *while*-Schleife ist der Eingangsknoten durch eine Kante mit dem Ausgangsknoten verbunden (s. Abbildung 7.3, links). Daher gibt es für alle $i \geq 0$ Wege, welche die *while*-Schleife i-mal durchlaufen.

Beispiel 7.1.4 (while-Schleife)
In Abbildung 7.1 bilden die Knoten C, D, E, F und die Kanten c, d, e, f, g eine while-Schleife mit Eingangsknoten C und Ausgangsknoten G, der durch die Kante h verbunden ist. Der Weg abh durchläuft die Schleife genau 0-mal. Die Kantenfolge c, d, e, g durchläuft die Schleife 1-mal; die Kantenfolge c, d, e, g, c, f, g durchläuft die Schleife 2-mal.

[3] Durch statische Analyse (s. Abschnitt 12.2.1) läßt sich überprüfen, ob Programme und ihre Kontrollflußgraphen diese Eigenschaften haben; falls nein, läßt sich das Programm entsprechend modifizieren, ohne sein Verhalten in den Fällen, wo die Berechnung endet, zu verändern.

[4] Eine weitere Variante ist der erlaubte Sprung an das Schleifenende, der aber einem normalen *goto* entspricht.

[5] Das ist eine *repeat*-Schleife, bei der das Schleifenprädikat einen konstanten Wert hat, so daß stets der Rücksprung erfolgt. Eine solche Schleife kann also nur mit *exit* verlassen werden.

[6] Auch das Vorliegen dieser Eigenschaft kann überprüft und durch entsprechende Veränderung des Programms hergestellt werden.

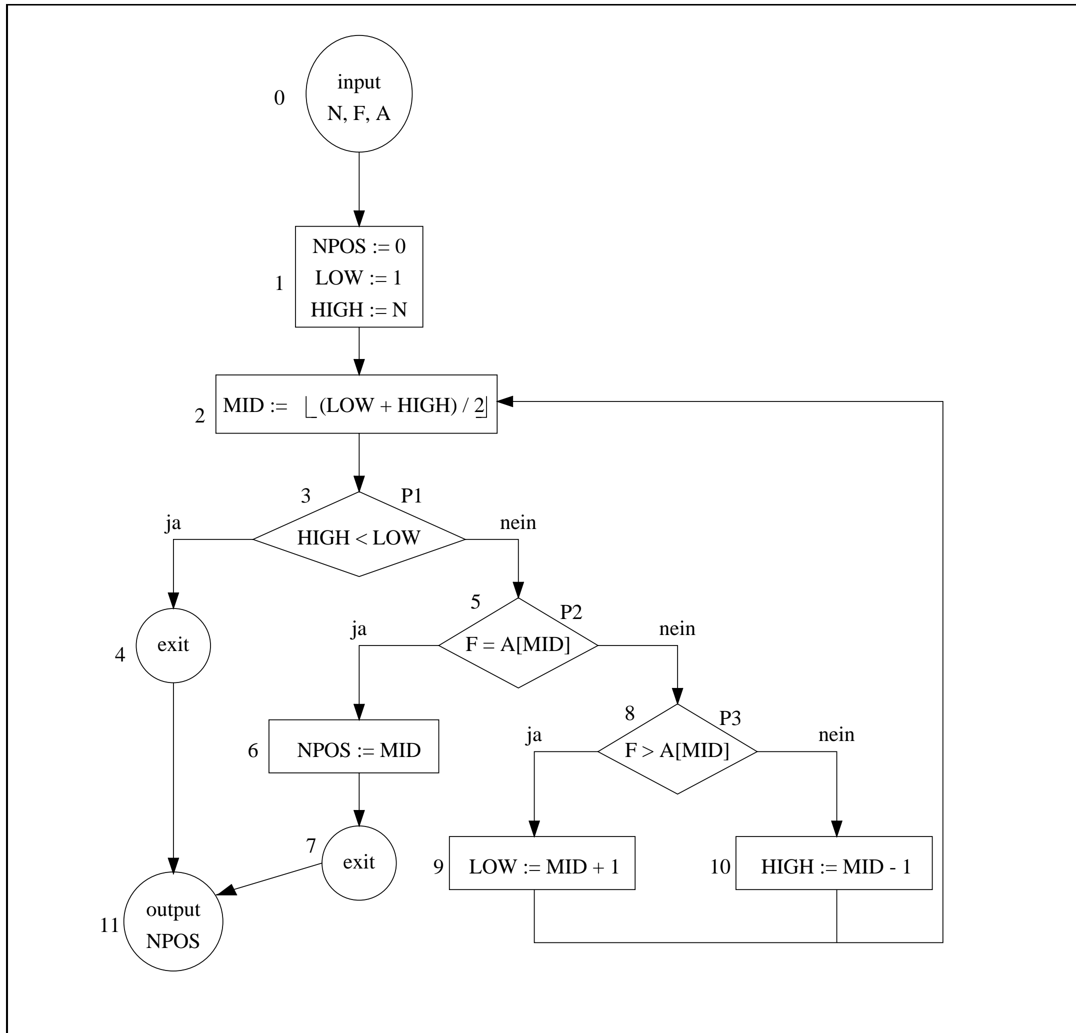


Abb. 7.2: Suchprogramm NPOS := SEARCH(N, A, F)

Bei einer *repeat*-Schleife gibt es einen „letzten“ Knoten L im Rumpf der Schleife, der sowohl mit dem Eingangsknoten als auch mit dem Ausgangsknoten durch je eine Kante verbunden ist (s. Abbildung 7.3, Mitte). Es gibt aber keine Kante vom Eingangsknoten zum Ausgangsknoten der *repeat*-Schleife. Daher gibt es keinen Weg, der die *repeat*-Schleife 0-mal durchläuft, aber für jedes $i \geq 1$ gibt es Wege, welche die Schleife i-mal durchlaufen.

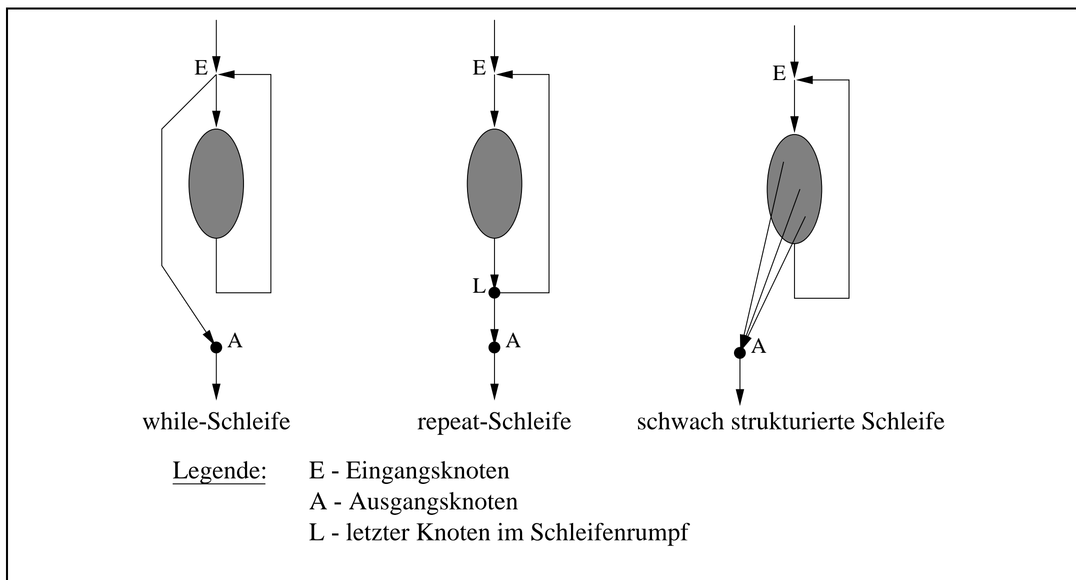


Abb. 7.3: Verschiedene Schleifenarten mit Eingangs- und Ausgangsknoten

Eine *for*-Schleife wird im Kontrollflußgraphen wie eine *while*-Schleife dargestellt, da der Unterschied nur in der Syntax besteht und bei entsprechender Umformung *for*-Schleifen Spezialfälle von *while*-Schleifen sind.

7.2 Methoden des kontrollflußbezogenen Testens

Die verschiedenen Methoden des kontrollflußbezogenen Testens haben folgende Gemeinsamkeiten:
Es wird festgelegt, welche Kontrollflußwege bzw. -wegstücke bzw. welche Konstrukte im Programm wie oft ausgeführt werden sollen. Ein Testwirksamkeitsmaß, abgekürzt „TWM“, gibt dann den Prozentsatz der Weg(stück)e bzw. Konstrukte an, bei denen diese Forderung beim Ausführen einer Testdatenmenge erfüllt ist.
Das Ziel des Testens ist natürlich weiterhin die Aufdeckung von Fehlern, wobei in Bezug auf die Programmstruktur folgende Fehlerarten unterschieden werden (vgl. Zuweisungs- und Abfragefehler auf S. 28):

DEFINITION 7.2.1 (BERECHNUNGS- UND [UNTER-]BEREICHSFEHLER)

Berechnungsfehler *(z. B. durch Zuweisungsfehler) sind Fehler, bei denen zwar der richtige Kontrollflußweg im Programm ausgeführt wird, aber mindestens ein berechneter Variablenwert falsch ist.*

Bereichsfehler *(z. B. durch Abfragefehler) sind Fehler, bei denen nicht der richtige Kontrollflußweg im Programm ausgeführt wird. Dies liegt daran, daß der* **Eingabebereich**[7] *des vorliegenden Kontrollflußweges nicht mit dem Eingabebereich des entsprechenden Kontrollflußweges im korrekten Programm übereinstimmt.*

Unterbereichsfehler *(als spezielle Bereichsfehler) sind Fehler, bei denen ein Kontrollflußweg zu wenig oder zu viel im Programm vorkommt, d. h. es fehlt eine Abfrage oder es gibt eine zusätzliche, falsche Abfrage (vgl. das Beispiel in der Einleitung von Teil III).*

Die verschiedenen Methoden des kontrollflußbezogenen Testens unterscheiden sich in der Festlegung der Art der Kontrollflußwege oder -wegstücke oder Konstrukte und in der geforderten Häufigkeit ihrer Ausführung.

Im folgenden werden die verschiedenen Methoden vorgestellt — zu Beginn die Methoden mit den einfachsten Kriterien, die sehr grob sind und daher nur wenige Fehler aufdecken können.

7.2.1 Anweisungsüberdeckung

Da jede Anweisung im Programm einen Fehler enthalten kann, der sich als Fehlverhalten bemerkbar machen kann, werden bei der ersten Strategie der kontrollflußorientierten Testdatenerzeugung die Anweisungen als auszuführende Programmkonstrukte gewählt.

DEFINITION 7.2.2
Eine Testdatenmenge T erfüllt die **C_0-Überdeckung** *(für Programm P) g. d. w. es für jede Anweisung A des Programms P ein Testdatum t aus T gibt, das die Anweisung A ausführt.*

Mit dem Begriff *Wege(T)* (s. Definition 7.1.4, 4) läßt sich Definition 7.2.2 auch folgendermaßen formulieren:

> Eine Testdatenmenge T erfüllt die C_0-Überdeckung (für Programm P) genau dann wenn es für jeden Knoten k des Kontrollflußgraphen von P einen Weg aus *Wege(T, P)* gibt, zu dem k gehört.

[7] die Menge aller Eingabewerte, die einen Weg ausführen

Die C_0-Überdeckung wird auch **Anweisungsüberdeckung** genannt.
Hier wird — wie im folgenden — von **Überdeckung** eines Konstrukts gesprochen, wenn jedes Vorkommen eines Konstrukts der betrachteten Art mindestens einmal ausgeführt wird.

Zur Anweisungsüberdeckung kann ein entsprechendes Testwirksamkeitsmaß[8] definiert werden. Dabei wird folgende Sprechweise benutzt:

> Eine Anweisung A (bzw. ein Entscheidungsausgang e) wird unter T **ausgeführt** bzw. **überdeckt** g. d. w. der zur Anweisung A gehörende Knoten k (bzw. die zum Entscheidungsausgang e gehörende Kante l) in einem Weg von *Wege(T)* vorkommt.

DEFINITION 7.2.3 (TESTWIRKSAMKEITSMASS 0)

$$\mathbf{TWM_0} := \frac{\textit{Zahl der unter } T \textit{ überdeckten Anweisungen}}{\textit{Zahl aller Anweisungen}}$$

wobei T eine Menge von Testdaten ist.
Falls die Abhängigkeit des Testwirksamkeitsmaßes von der Testdatenmenge T (und vom betrachteten Programm P) ausgedrückt werden soll, wird $\mathbf{TWM_0(T)}$ *bzw.* $\mathbf{TWM_0(T, P)}$ *statt TWM_0 geschrieben.*

7.2.2 Zweig-, Segment- und Entscheidungsüberdeckung

Die Anweisungsüberdeckung ist zwar ein notwendiges Kriterium, um potentielle Fehler aufzudecken, aber keineswegs hinreichend. Gewisse Kontrollflußfehler werden nicht entdeckt, wie folgendes Beispiel zeigt.

BEISPIEL 7.2.1
Für das Programm aus Abbildung 7.4 sei x_0 der Anfangswert von x, y_0 der Anfangswert von y, beide Werte seien definiert. Dann ergibt sich folgendes:

für $x_0 > 0$ gilt für beide Programme: $x = (x_0)^2$
$y = (x_0)^2 + 5$
für $x_0 \leq 0$ gilt: $x = x_0$ *für beide Programme*
$y = y_0$ *für das falsche Programm*
$y = x_0 + 5$ *für das richtige Programm*

[8] (engl.) test effectiveness. Der Begriff ist etwas irreführend (vgl. Fußnote 22 zu Beginn von Kapitel 6.4). Die Wirksamkeit in Bezug auf die Fehleraufdeckung (genaueres s. Kap. 10.3) wird damit nur indirekt erfaßt.

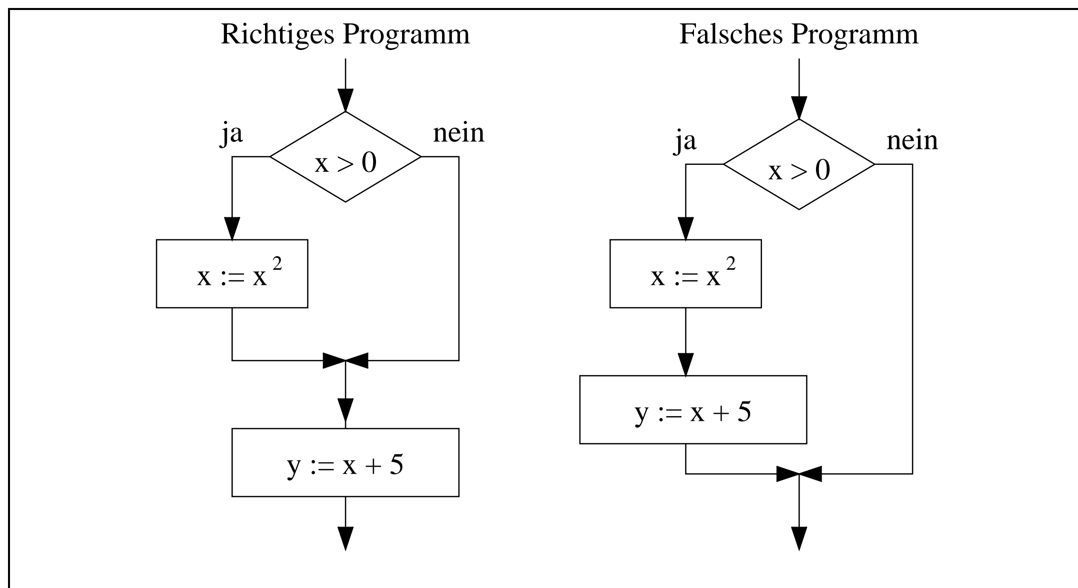


Abb. 7.4: Kontrollflußfehler

Die Testdatenmenge T enthalte nur folgendes Testdatum t_1:

(Eingabedatum: $x = 5$, $y = 0$; Solldatum: $x = 25, y = 30$).

T erfüllt die C_0-Überdeckung, aber deckt den Fehler nicht auf. Nur ein Testdatum t_2, das den negativen Ausgang der Entscheidung ausführt, deckt den Fehler auf:

t_2: Eingabedatum: $x = -1$, $y = 0$; Solldatum: $x = -1$, $y = 4$;
Istdatum: $x = -1$, $y = 0 \neq 4$.

Mit den Testdaten müssen also zumindest alle von bedingten Anweisungen ausgehenden Zweige bzw. die entsprechenden Kanten im Kontrollflußgraph überdeckt werden.

Definition 7.2.4

Eine Testdatenmenge T erfüllt die **C_1-Überdeckung** *(für Programm P) g. d. w. es für jede Kante k im Kontrollflußgraphen von P einen Weg in Wege(T, P) gibt, zu dem k gehört.*

Die C_1-Überdeckung wird auch **Zweigüberdeckung** *genannt, da das Wort „Zweig“ als Synonym für den Begriff „Kante“ gebraucht wird.*

Der folgende Satz 7.2.1 zeigt, daß man sich für die Feststellung der Zweigüberdeckung auf Entscheidungskanten, Segmente oder Entscheidungs-Entscheidungs-Wege beschränken kann.

SATZ 7.2.1

Für eine Testdatenmenge T gilt:
A): T erfüllt die C_1-Überdeckung
g. d. w.
B): jede Entscheidungskante und die Anfangskante wird unter T ausgeführt
g. d. w.
C): jedes Segment wird unter T ausgeführt
g. d. w.
D): jeder Entscheidungs-Entscheidungs-Weg wird unter T ausgeführt.

Beweis:
Der Beweis der Äquivalenz der vier Aussagen wird durch zyklische Implikationen $A) \Rightarrow D) \Rightarrow C) \Rightarrow B) \Rightarrow A)$ geführt.
Teilbeweis $A) \Rightarrow D) \Rightarrow C) \Rightarrow B)$:
Dies folgt direkt aus den Definitionen 7.2.4 (C_1-Überdeckung), 7.1.2 (Entscheidungs-Entscheidungs-Weg), 7.1.3 (Segment), 7.1.1 (Entscheidungs- und Anfangskante).
Teilbeweis $B) \Rightarrow A)$*:*
Sei k eine beliebige Kante des Kontrollflußgraphen. Da strukturierte und somit wohlgeformte Kontrollflußgraphen vorausgesetzt werden, existiert ein Weg w vom Anfangsknoten zur Kante k.
Fall 1): Der Weg enthält keine Entscheidungskanten.
Dann wird jede Kante des Weges — also auch k — ausgeführt, da die Anfangskante des Weges nach Voraussetzung B ausgeführt wird.
Fall 2): Der Weg w enthält Entscheidungskanten.
Sei k_e die letzte Entscheidungskante auf dem Weg. Da k_e nach Voraussetzung B ausgeführt wird, wird auch der Rest des Weges — inklusive k — nach Satz 7.1.1 ausgeführt.

q. e. d.

Wegen der Aussagen B und C von Satz 7.2.1 wird die C_1-Überdeckung auch **Entscheidungsüberdeckung** oder **Segmentüberdeckung** genannt und ihr kann (für Programme mit Entscheidungen) das folgende Testwirksamkeitsmaß zugeordnet werden.

DEFINITION 7.2.5 (TESTWIRKSAMKEITSMASS 1)

$$\mathbf{TWM_1} := \frac{\textit{Zahl der unter } T \textit{ überdeckten Entscheidungskanten}}{\textit{Zahl aller Entscheidungskanten}}$$

wobei T eine Menge von Testdaten ist.
Falls die Abhängigkeit des Testwirksamkeitsmaßes von der Testdatenmenge T (und vom betrachteten Programm P) ausgedrückt werden soll, wird $\mathbf{TWM_1(T)}$ *bzw.* $\mathbf{TWM_1(T, P)}$ *statt* TWM_1 *geschrieben.*

7.2.3 Kombination von Segmenten

Bisher wurde nur die Überdeckung von Konstrukten im Kontrollfluß der Programme gefordert, die sich jeweils an einer bestimmten Stelle im Kontrollflußgraphen befinden. Wenn aber jede Anweisung und jedes Segment ausgeführt wird und kein Fehler auftritt, kann dennoch bei einer bestimmten Kombination von Anweisungen oder Segmenten ein Fehler auftreten.

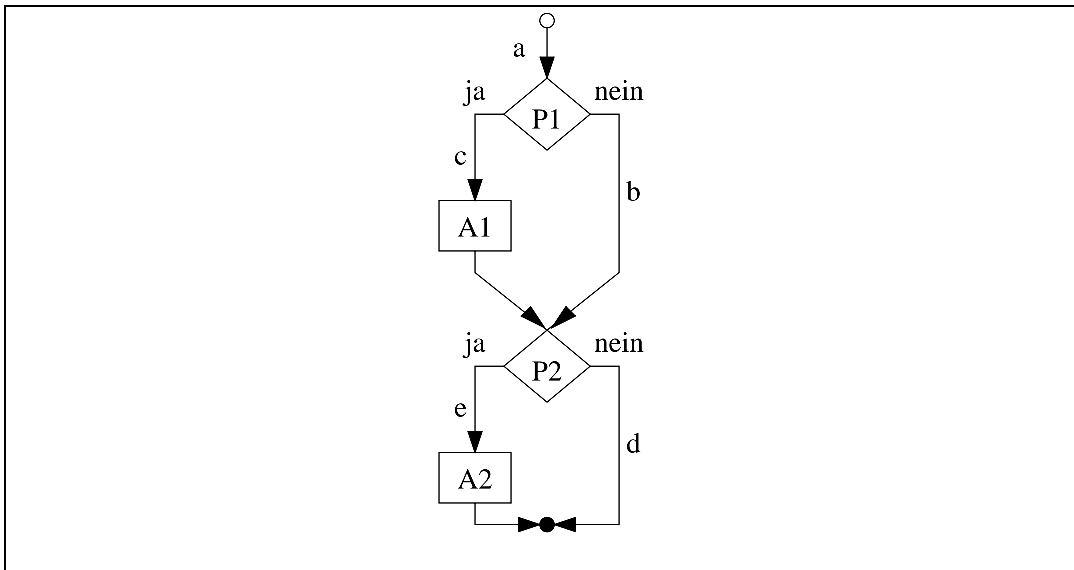


Abb. 7.5: Kontrollflußgraph mit leeren Segmenten b und d

Beispiel 7.2.2

Für eine C_1-Überdeckung des Programms aus Abbildung 7.5 reicht die Ausführung aller vier Entscheidungszweige bzw. aller fünf Segmente aus. Folgende zwei Testdaten t_1 und t_2 erfüllen dies:

$t_1 : P1 = false$ *und* $P2 = true$*: die Segmentfolge* abe *wird ausgeführt;*

$t_2 : P1 = true$ *und* $P2 = false$*: die Segmentfolge* acd *wird ausgeführt.*

Damit werden alle Segmente a, b, c, d, e und die Segmentkombinationen ab, ac, be und cd ausgeführt.

Die (möglichen) Segmentkombinationen bd und ce werden also nicht ausgeführt. Bei diesen Kombinationen kann aber gerade ein Fehlverhalten auftreten (vgl. Übung 7.5).

Beispiel 7.2.2 gibt Anlaß, die Ausführung von bestimmten Kontrollflußwegen zu fordern. Die weitestgehende Forderung wäre, alle Wege im Programm zu testen. Das entsprechende Überdeckungskriterium heißt **Pfadüberdeckung**, da Pfad ein Synonym für Weg ist. Für Programme mit Schleifen, die keine obere Grenze für die Iterationszahl haben (wie das Programm aus Abbildung 7.1), gibt es aber unendlich

viele Wege. Selbst für Programme ohne Schleifen, aber mit k aufeinanderfolgenden Verzweigungen, gibt es schon 2^k verschiedene Wege, also für großes k zu viele Wege (s. Abbildung 7.6). Es müssen also weniger starke Kriterien herangezogen werden, als Minimalforderung etwa folgende Kombination von Segmenten bzw. Entscheidungs-Entscheidungs-Wegen.

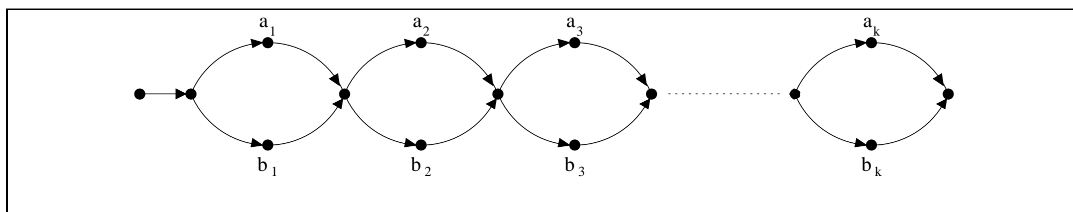


Abb. 7.6: k aufeinanderfolgende Verzweigungen

DEFINITION 7.2.6

Eine Testdatenmenge T erfüllt die **C_{SP}-Überdeckung** *g. d. w. es für jedes Paar von Entscheidungs-Entscheidungs-Wegen q und r, die im Kontrollflußgraph direkt aufeinanderfolgen, ein Testdatum t in T gibt, das die Folge qr ausführt. (qr ist Teil eines Weges aus Wege(T)).*
Die C_{SP}-Überdeckung heißt auch **Segmentpaareüberdeckung**[9].

BEISPIEL 7.2.3

In dem Programm von Abbildung 7.5 werden mit den Testdaten t_1 und t_2 der Testdatenmenge von Beispiel 7.2.2 die Segmentpaare ab, ac, be und cd ausgeführt. Für die Segmentpaare bd und ce braucht man noch zwei zusätzliche Testdaten t_3 und t_4:

t_3: *P1 = P2 = false,*
t_4: *P1 = P2 = true.*

Die C_{SP}-Überdeckung kann verallgemeinert werden, indem nicht Paare, sondern n-Tupel von n aufeinanderfolgenden Entscheidungs-Entscheidungs-Wegen (mit $n > 2$) betrachtet werden.

DEFINITION 7.2.7

Eine Testdatenmenge T erfüllt die **$C_S(n)$-Überdeckung** *g. d. w. es für jede Folge von n Entscheidungs-Entscheidungs-Wegen $S_1, \ldots, S_n$, die im Kontrollfluß direkt aufeinanderfolgen, ein Testdatum t in T gibt, das die Folge $S_1, \ldots, S_n$ ausführt. [$S_1, \ldots, S_n$ ist Teil eines Weges aus Wege(T)].*

[9] da Segmente oft mit Entscheidungs-Entscheidungs-Wegen übereinstimmen (vgl. Beispiel 7.1.1)

BEISPIEL 7.2.4 ($C_S(3)$-ÜBERDECKUNG)
Für den Kontrollflußgraphen aus Abbildung 7.1 lassen sich z. B. folgende Paare und Tripel von Entscheidungs-Entscheidungs-Wegen E_i bilden (vgl. Beispiel 7.1.1 auf Seite 189 f. für die Bezeichnungen E_i):

1. *alle Paare (bzw. Kantenfolgen)*

$E_1E_2 = ab\ h$	$E_1E_3 = ab\ c$
$E_3E_4 = c\ deg$	$E_3E_5 = c\ fg$
$E_4E_2 = deg\ h$	$E_4E_3 = deg\ c$
$E_5E_2 = fg\ h$	$E_5E_3 = fg\ c$

2. *einige Tripel (bzw. Kantenfolgen)*

$E_1E_3E_4 = ab\ c\ deg$	$E_1E_3E_5 = ab\ c\ fg$
$E_3E_4E_2 = c\ deg\ h$	$E_3E_4E_3 = c\ deg\ c$
$E_4E_3E_4 = deg\ c\ deg$	$E_4E_3E_5 = deg\ c\ fg$

Eine Variante der Segmentpaare- und $C_S(n)$-Überdeckung ist die LCMS[10]-Überdeckung, die sich allerdings an der Reihenfolge der Segmente im Programmtext orientiert. Deshalb werden Folgen von *then*-Zweigen anders behandelt als entsprechende *else*-Zweige. Das kann berechtigt sein, wenn die *then*-Zweige die Standardfälle und die *else*-Zweige die Sonderfälle enthalten. (Genaueres siehe [WHH 80].)
Eine andere Annäherung an die Forderung, alle Wege (ohne mehrfache Schleifendurchläufe) auszuführen, ist folgendes Kriterium: Es geht vom ungerichteten (!) Kontrollflußgraphen aus, bei dem der Endknoten durch eine zusätzliche Kante mit dem Anfangsknoten verbunden wird, und einem **Spannbaum**[11] für diesen (streng zusammenhängenden) Graphen. Jede Kante des (erweiterten) Kontrollflußgraphen, die nicht zum Spannbaum gehört (**charakteristische Kante** genannt), bildet dann mit (einigen) Kanten des Spannbaums genau einen Zyklus, **fundamentaler Zyklus** genannt. Daraus läßt sich wiederum eine gerichtete Teilfolge im ursprünglichen Kontrollflußgraphen bilden, die **fundamentaler Weg** genannt wird. Als Testkriterium wird gefordert, daß **alle fundamentalen Wege** ausgeführt werden. (Aus den fundamentalen Zyklen kann jeder vollständige Weg [ohne mehrfache Schleifendurchläufe] durch „exklusive" Vereinigung [entspricht dem „exklusiven oder"] gebildet werden.)

BEISPIEL 7.2.5 (FUNDAMENTALE WEGE)
Aus dem Kontrollflußgraphen aus Abbildung 7.1 ergibt sich ein Spannbaum, wenn man z. B. die Kanten e und g wegläßt (alternativ: f und g weglassen). Nicht zum Spannbaum gehören also die charakteristischen Kanten e, g und die zusätzliche Kante i vom Endknoten G zum Anfangsknoten A. Fundamentale Zyklen sind dann def

[10] lineare Codesequenz mit Sprung
[11] Ein Spannbaum eines Graphen G ist ein maximaler Teilgraph von G, der (noch) ein Baum ist.

(für Kante e), cfg (für Kante g), abhi (für Kante i); fundamentale Wege also de, cfg, abh. Die Ausführung der drei fundamentalen Wege erfordert also (vgl. Abbildung 7.1) einen Test mit $y = 0$ (für abh), einen Test mit $y \neq 0$ und $y \mod 2 = 1$ (für de), sowie einen Test mit $y \neq 0$ und $y \mod 2 = 0$ (für cfg). Der vollständige Weg $abcdegh(i)$ ergibt sich durch exklusive Vereinigung der Zyklen $abhi$, cfg und def. (Dabei fällt f weg, da f zweimal vorkommt und beim „exklusiven oder" $f \oplus f = 0$ gilt.)

7.2.4 Schleifenüberdeckung

Die folgenden Überdeckungskriterien bewerten Testdatenmengen unter dem Aspekt der Ausführung von Schleifen im Programm. Um den Kontrollfluß vollständig zu testen, müßten Schleifen u-mal, $(u+1)$-mal, $(u+2)$-mal bis o-mal durchlaufen werden, wenn u die minimale (untere) und o die maximale (obere) Anzahl der möglichen Schleifendurchläufe ist. Dies ist bei *for*-Schleifen mit konstanten Grenzen einfach, da dann $o = u$ ist. Bei allen anderen Schleifen kann die Bestimmung der minimalen und maximalen Anzahl schwierig sein.
Oft ist die Anzahl sogar unbegrenzt. Dann hilft nur ein vollständiger Beweis (vgl. „formale Verifikation" in Abschnitt 3.3.1 bzw. Kapitel 12.4) oder eine Beschränkung auf endlich viele Klassen von den unendlich vielen möglichen Wegen durch die Schleife. Aus jeder Klasse ist dann ein repräsentativer Weg zu wählen[12].

Die folgenden Schleifenüberdeckungskriterien unterscheiden sich durch die *Feinheit* der Bildung von *Wegeklassen.* Bei der Definition dieser disjunkten Klassen muß festgelegt werden, wann zwei Wege zur selben (Äquivalenz-) Klasse gehören sollen. Ein Ansatz besteht darin, die Wegstücke, die eine Schleife mehr als k-mal durchlaufen, einfach nicht zu beachten.

Falls Schleifen keine anderen Schleifen enthalten, können die folgenden Definitionen vereinfacht werden, z. B. die $C_i(k)$-Überdeckung (s. Definition 7.2.10) zu der Forderung „jede Schleife ist j-mal (auf alle möglichen Arten) zu durchlaufen, $j = 0, 1, \ldots, k$". Im Falle von geschachtelten Schleifen ist aber unklar, was das für äußere und innere Schleifen bedeuten soll. Daher werden die Begriffe auf die folgende Art präzisiert (Definition 7.2.8 bis 7.2.11) und erläutert (Beispiele 7.2.6 bis 7.2.9).

[12] Was im folgenden über Schleifen gesagt wird, gilt entsprechend für den Fall, daß Berechnungswiederholungen durch mehrfachen Aufruf derselben Funktion (**Rekursion**) anstatt durch Schleifen im Kontrollfluß (**Iteration**) implementiert werden.

DEFINITION 7.2.8
Zwei Wege im Kontrollflußgraphen heißen **k-äquivalent**, *wenn sie schleifenfrei und identisch sind oder wenn für jede Schleife gilt: sie durchlaufen die Schleife*

1. *weniger als k-mal und sind identisch (bzw. in inneren Schleifen k-äquivalent) oder*

2. *mindestens k-mal, wobei*

   (a) *die ersten $k-1$ Durchläufe identisch (bzw. in inneren Schleifen k-äquivalent) sind,*

   (b) *der k-te Durchlauf identisch (bzw. in inneren Schleifen k-äquivalent) ist, aber sich darin unterscheiden kann, daß die Kante zum Eingangsknoten der Schleife fehlt, wenn die Schleife genau k-mal durchlaufen wird*[13],

   (c) *weitere Durchläufe ($k+1, k+2, \ldots$) beliebig aussehen oder auch nicht vorhanden sein dürfen.*

Für $k = 2$ Durchläufe kann das Äquivalenzkriterium noch abgeschwächt werden, indem alle zweiten und weiteren Durchläufe — falls vorhanden — als äquivalent betrachtet werden.

DEFINITION 7.2.9
Zwei vollständige Wege im Kontrollflußgraphen heißen **schwach 2-äquivalent**, *wenn sie schleifenfrei und identisch sind oder wenn wenn für jede Schleife gilt: sie durchlaufen die Schleife*

1. *höchstens einmal und sind identisch (bzw. in inneren Schleifen schwach 2-äquivalent) oder*

2. *mindestens zweimal, wobei der erste Durchlauf identisch ist (bzw. in inneren Schleifen schwach 2-äquivalent).*

Mit Hilfe der obigen Äquivalenzbegriffe kann nun das folgende allgemeine Schleifen-Überdeckungskriterium definiert werden.

DEFINITION 7.2.10
Für $k > 0$ erfüllt eine Testdatenmenge T die $\mathbf{C_i(k)}$**-Überdeckung** *g. d. w. Wege(T) mindestens einen Weg aus jeder Klasse von k-äquivalenten Wegen enthält.*
(Beachte: Wege(T) ist die beim Ausführen der Tests aus T erzeugte Menge von ausgeführten vollständigen Wegen, s. Def. 7.1.4 (4) auf S. 191.)
Die $C_i(k)$-Überdeckung heißt auch **strukturierte Pfadüberdeckung**.

Da die strukturierte Pfadüberdeckung für große Werte von k die Überdeckung von sehr vielen Wegen fordert, sind nur kleine Werte von k, insbesondere $k = 1$ und $k = 2$, praktikabel. Für diese Werte erhält das Kriterium daher besondere Namen.

[13] Dies kommt nur bei *repeat*-Schleifen vor, die genau k-mal durchlaufen werden.

Definition 7.2.11 ([starke/schwache] C_{GI}-Überdeckung)

1. *Eine Testdatenmenge T erfüllt die* **starke $\mathbf{C_{GI}}$-Überdeckung** *g. d. w. die Testdatenmenge T die $C_i(2)$-Überdeckung erfüllt.*

2. *Eine Testdatenmenge T erfüllt die* **$\mathbf{C_{GI}}$-Überdeckung** *g. d. w. Wege(T) mindestens einen Weg aus jeder Klasse von schwach 2-äquivalenten Wegen enthält.*

3. *Eine Testdatenmenge T erfüllt die* **schwache $\mathbf{C_{GI}}$-Überdeckung** *g. d. w. die Testdatenmenge T die $C_i(1)$-Überdeckung erfüllt.*

Die C_{GI}-Überdeckung heißt auch **Grenze-Inneres-Überdeckung** *(boundary-interior coverage), da dabei für eine Schleife folgende zwei oder drei Fälle unterschieden werden:*

1. *0-maliger Durchlauf (nur bei while-Schleifen möglich),*

2. *1-maliger Durchlauf,*

3. *mehrmaliger Durchlauf der Schleife.*

Die Fälle 1 und 2 stellen dabei Grenzfälle dar, im Falle 3 wird die Iteration der Schleife getestet.

Die Aufwandsproblematik beim strukturierten Pfadtesten mit $k > 1$ und bei der starken, „normalen" und schwachen C_{GI}-Überdeckung sei an folgenden Beispielen erläutert, die sich auf den Kontrollflußgraphen aus Abbildung 7.7 beziehen.

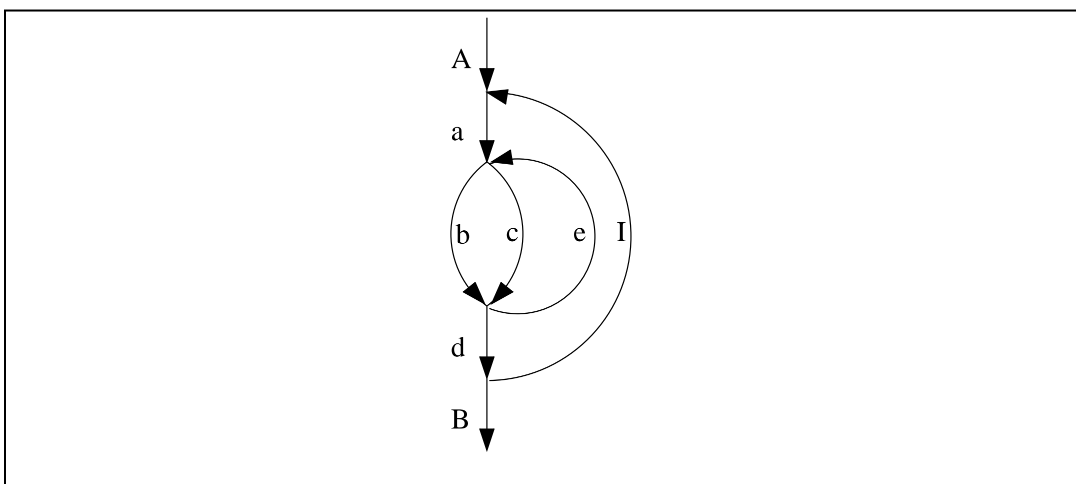


Abb. 7.7: Geschachtelte *repeat*-Schleifen

BEISPIEL 7.2.6 (SCHWACHE C_{GI}-ÜBERDECKUNG)
Für die innere repeat-Schleife des Kontrollflußgraphen aus Abbildung 7.7 erhält man folgende Klassen auszuführender Wegstücke, die 1-äquivalent sind:

- *0-mal durch den Schleifenkörper: das gibt es nicht,*
- *mindestens 1-mal durch den Schleifenkörper: bx, cx*
  (wobei x weitere Durchläufe der inneren Schleife beschreibt oder das leere Symbol ist, falls keine weiteren Durchläufe vorkommen).

Sei M die Menge dieser beiden Wegeklassen bx und cx. Dann erhält man für die äußere Schleife folgende Klassen auszuführender Wege bzgl. 1-Äquivalenz:

- *0-mal durch den Schleifenkörper: das gibt es nicht,*
- *mindestens 1-mal durch den Schleifenkörper:*
  $AaydzB$ mit $y \in M$ (wobei z keine oder weitere Durchläufe der äußeren Schleife beschreibt).

Also gibt es die beiden Klassen

- *AabxdzB (x, z wie oben)*
- *AacxdzB (x, z wie oben)*

Es sind also nur zwei Tests (je einer pro Klasse) durchzuführen, um das Kriterium schwache C_{GI}-Überdeckung *zu erfüllen.*

BEISPIEL 7.2.7 (GRENZE-INNERES-ÜBERDECKUNG)
Für die innere repeat-Schleife des Kontrollflußgraphen aus Abbildung 7.7 erhält man folgende Klassen auszuführender Wegstücke, die schwach 2-äquivalent sind:

- *1-mal durch den Schleifenkörper: b und c;*
- *mindestens 2-mal durch den Schleifenkörper: bex und cex*
  (wobei x mindestens einen weiteren Durchlauf der inneren Schleife beschreibt).

Sei K die Menge dieser vier Wegeklassen b, c, bex, cex.
Dann erhält man für die äußere Schleife folgende Klassen auszuführender Wegstücke, die schwach 2-äquivalent sind:

- *1-mal durch den Schleifenkörper: $AaxdB$ mit x Element von K,*
- *mindestens 2-mal durch den Schleifenkörper: $AaxdIay$*
  (wobei x Element von K ist und y mindestens einen weiteren Durchlauf der äußeren Schleife beschreibt).

Da K vier Wegeklassen enthält, sind für x vier verschiedene Wegeklassen einzusetzen. Also sind $2 * 4 = 8$ Wege auszuführen, um die Grenze-Inneres-Überdeckung zu erfüllen.

Beispiel 7.2.8 (starke C_{GI}-Überdeckung bzw. $C_i(2)$-Überdeckung)
Für die innere repeat-Schleife des Kontrollflußgraphen aus Abbildung 7.7 erhält man folgende Klassen auszuführender Wegstücke, die 2-äquivalent sind:

- *1-mal durch den Schleifenkörper: b und c;*
- *mindestens 2-mal durch den Schleifenköper: bebx, becx, cebx und cecx. (Dabei beschreibt x keine oder weitere Durchläufe der inneren Schleife.)*

Sei L die Menge dieser sechs Wegeklassen b, c, $bebx$, $becx$, $cebx$, $cecx$. Dann erhält man für die äußere Schleife folgende Klassen auszuführender Wegstücke, die stark 2-äquivalent sind:

- *1-mal durch den Schleifenkörper: $AaxdB$ mit x Element von L,*
- *mindestens 2-mal durch den Schleifenkörper: $AaxdIaydzB$ (mit x, y Element von L und einem Wegstück z, das keine oder weitere Durchläufe der äußeren Schleife beschreibt).*

Da L sechs Wegeklassen enthält, sind also $6 + 6 * 6 = 42$ Wege auszuführen, um die $C_i(2)$-Überdeckung *zu erfüllen.*

Beispiel 7.2.9 ($C_i(k)$-Überdeckung)
Bei der $C_i(3)$-Überdeckung müßten 2.954 Wege ausgeführt werden, um das Programm mit dem Kontrollflußgraphen aus Abbildung 7.7 hinreichend zu testen. Es gibt nämlich $2+2^2+2^3 = 14$ Möglichkeiten, die innere Schleife bis zu 3-mal (bei einem Durchlauf der äußeren Schleife) zu durchlaufen. Daher gibt es $14+14^2+14^3 = 2.954$ Möglichkeiten, die äußere Schleife bis zu 3-mal zu durchlaufen.
Bei der $C_i(4)$-Überdeckung müßten sogar 837.930 Wege ausgeführt werden, um das Programm mit dem Kontrollflußgraphen aus Abbildung 7.7 hinreichend zu testen. Es gibt nämlich $2 + 2^2 + 2^3 + 2^4 = 30$ Möglichkeiten, die innere Schleife bis zu 4-mal (bei einem Durchlauf der äußeren Schleife) zu durchlaufen. Daher gibt es $30 + 30^2 + 30^3 + 30^4 = 837.930$ Möglichkeiten, die äußere Schleife bis zu 4-mal zu durchlaufen.

Praktikabler und sinnvoller als eine $C_i(k)$-Überdeckung für großes k ist daher eine C_{GI}-Überdeckung oder $C_i(2)$-Überdeckung, die ergänzt wird um einige große Durchlaufzahlen und/oder Durchlaufzahlen, die sich an Grenzwerten orientieren.

Beispiel 7.2.10
Ein Kalendererzeugungsprogramm, welches für jeden Tag ein Blatt ausdruckt, sollte mit den Iterationszahlen 365 und 366 getestet werden.

Für Schleifen sollte noch ihre Lage im Gesamtprogramm berücksichtigt werden. Man kann dabei geschachtelte (nested), verkettete (concatenated) und schrecklich unstrukturierte (horrible) Schleifen unterscheiden (genaueres siehe [Bei 83]).
Außerdem sollte sich der Test einer Schleife an der minimalen und maximalen Anzahl der Iterationen und an eventuell ausgeschlossenen Iterationszahlen orientieren, falls diese Werte bekannt sind.

Statt sich direkt an den Schleifen im Programm zu orientieren, kann man auch die Häufigkeit betrachten, mit der Knoten, Kanten oder Wegstücke des Kontrollflußgraphen durchlaufen werden. Damit erhält man Kriterien, die stärkere Anforderungen als die Zweigüberdeckung und schwächere Anforderungen als die Pfadüberdeckung stellen, aber orthogonal zu den bisherigen Schleifenkriterien liegen.

Definition 7.2.12 (k-iterativ, einfach, 2-Knoten-iterativ)
Ein vollständiger Weg w in einem Kontrollflußgraphen heißt

1. **k-iterativ**, *wenn jedes Wegstück höchstens k-mal iteriert in w vorkommt, d. h. es gibt keine Wegstücke u, v, x mit $w = uv^{k+1}x$, wobei v nicht die Länge 0 hat;*
2. **einfach**, *wenn jede Kante höchstens einmal in w vorkommt;*
3. **2-Knoten-iterativ**, *wenn jeder Knoten höchstens zweimal im Weg w vorkommt.*

Wege, die k-iterativ sind, können eine Schleife mehr als k-mal und einfache Wege können eine Schleife mehr als einmal durchlaufen, wenn sie bei jedem Schleifendurchlauf andere Wege bzw. völlig andere Kanten enthalten. Die Anzahl der einfachen Wege ist zwar deutlich beschränkt, die Anzahl der k-iterativen Wege ist erstaunlicherweise bei geeigneten Kontrollflußgraphen schon für $k = 1$ und $k = 2$ unendlich groß. Daher scheidet das Kriterium *teste alle k-iterativen Wege* als praktikables Testkriterium aus, kann also auch nicht als Formalisierung des intuitiven Begriffs „eine Schleife bis zu k-mal durchlaufen“ bzw. der strukturierten Pfadüberdeckung verwendet werden.

Beispiel 7.2.11
Bei dem Kontrollflußgraphen aus Abbildung 7.1 ist der Weg „abcdegh“ 1-iterativ. Es gibt aber einen unendlichen 2-iterativen Weg im Kontrollflußgraphen und damit unendlich viele 2-iterative vollständige Wege, die aus einem Anfangsstück dieses Weges bestehen, an das die Kante h angehängt wird.
Der unendliche Weg läßt sich aus dem unendlichen String konstruieren, den Thue

schon 1906 bzw. 1912 angegeben hat (s. Literaturreferenzen in [Rot 92]). Er enthält das Teilstück „xxyyxyxxyxyyxyxxyyxx", welches 2-iterativ ist. Darin ist in diesem Falle x durch „cdeg" (den linken Weg durch die Schleife) und y durch „cfg" (den rechten Weg durch die Schleife) zu ersetzen. Das ergibt — mit dem zusätzlichen Präfix „ab" und dem Postfix „h" — den vollständigen Weg „abcdegcdegcfgcfgcdegcfgcdegcdegcfgcdegcfgcfgcdegcfgcdegcdegcfgcfgcdegcdegh", der ebenfalls 2-iterativ ist (Beweis: siehe Proposition 2.3 in [Rot 92]).

Unendlich viele *1-iterative Wege* erhält man mit einer entsprechenden Konstruktion für einen Kontrollflußgraphen, der genau eine while-Schleife enthält, bei der es *drei* verschiedene Wege für einen Durchlauf der Schleife gibt. Diese Wege lassen sich aus dem unendlichen 1-iterativen String über einem Alphabet mit drei Zeichen (z. B. x, y, z) konstruieren, der ebenfalls von Thue bzw. von Lothaire stammt (s. [Rot 92]). Jedes Zeichen ist dann durch einen der drei Wege in der Schleife zu ersetzen.

7.3 Übungen

Übung 7.1:

Geben Sie den Kontrollflußgraphen und das Kontrollflußschema für den Textformatierer aus Beispiel 7.3.1 an. Stellen Sie dabei Segmente stets nur durch zwei oder drei Knoten dar (vgl. Definition 7.1.3):

(a) den ersten Knoten (Anfangsknoten, Entscheidungsknoten oder Vereinigungsknoten),

(b) den letzten Knoten (Endknoten, Entscheidungsknoten oder Vereinigungsknoten),

(c) die evtl. dazwischen liegenden Knoten zu einem Knoten zusammengefaßt.

Die beiden Anweisungen von Zeile 24 aus Beispiel 7.3.1 sind also z. B. durch einen Knoten darzustellen.
Stellen Sie die *for*-Schleife (Zeilen 15 und 16) durch eine *repeat*-Schleife dar, bei der die Zählvariable k entsprechend zu initialisieren und zu inkrementieren ist. (Wegen der Bedingung „$bufpos \neq 0$" in Zeile 6 wird die *for*-Schleife stets mindestens einmal durchlaufen).

Beispiel 7.3.1 (Textformatierer)
Das folgende Programm stellt eine Implementation des Textformatierers aus Kapitel 1.2 dar, wobei die Datendeklarationen weggelassen wurden, da sie aus der Benutzung erschlossen werden können. Die Prozeduren „inchar" und „outchar" arbeiten zeichenweise (lesen aus der Eingabedatei bzw. schreiben in die Ausgabedatei).

Die boolesche Variable „alarm“ steuert den Abbruch wegen Überschreitung der Zeilenlänge bei der Ausgabe, die Variable „fill“ speichert den Füllungsgrad der aktuellen Ausgabezeile (vgl. f in Beispiel 4.2.9 auf S. 84).

```
1   alarm := false; bufpos := 0; fill := 0;
2   repeat
3       inchar(c);
4       if (c=BL) or (c=NL) or (c=EOF)
5           then begin
6               if bufpos ≠ 0
7               then begin
8                   if (fill + bufpos < MAXPOS) and (fill ≠ 0)
9                   then begin
10                      outchar(BL); fill:=fill + 1;
11                  end;
12                  else begin
13                      outchar (NL); fill:= 0;
14                  end;
15                  for k := 1 to bufpos do
16                      outchar (buffer[k]);
17                  fill := fill + bufpos; bufpos := 0;
18              end
19          end
20          else
21              if bufpos = MAXPOS
22              then alarm := true;
23              else begin
24                  bufpos := bufpos + 1; buffer[bufpos] := c;
25              end;
26  until alarm or (c = EOF);
27
```

Übung 7.2:

Ermitteln Sie alle Entscheidungszweige, Entscheidungs-Entscheidungs-Wege, Segmente, Zyklen sowie Schleifen (mit Eingangs- und Ausgangsknoten) für folgende Kontrollflußgraphen:

(a) Abbildung 7.2 (Suchprogramm SEARCH),

(b) Kontrollflußgraph des Textformatierers (siehe Beispiel 7.3.1 bzw. Übung 7.1).

Übung 7.3:

Berechnen Sie die Testwirksamkeitsmaße $TWM_0(T, P)$ und $TWM_1(T, P)$ bzgl. Anweisungs- und Zweigüberdeckung für folgende Programme bzw. Kontrollflußgraphen:

(a) Programm aus Abbildung 7.1 jeweils getrennt für die Testdatenmengen $T = \{t_1\}$ und $T = \{t_2\}$:
t_1 mit Eingaben $x = 3, y = 4$,
t_2 mit Eingaben $x = 5, y = 0$.

(b) Suchprogramm SEARCH aus Abbildung 7.2 mit einem Testdatum mit Eingabedatum $N = 4, F = 5, A = (3, 5, 9, 15)$, d. h. $A[1] = 3, A[2] = 5$, etc.

Übung 7.4:

Betrachten Sie den Kontrollflußgraphen aus Übung 7.1 (Textformatierer).

(a) Bestimmen Sie alle Segmentpaare.

(b) Welche Segmentpaare sind nicht ausführbar?

Übung 7.5:

Überlegen Sie sich ein Programm mit einem Fehler, bei dem eine bestimmte Zweigüberdeckung den Fehler nicht aufdeckt, wohl aber eine Überdeckung aller Segmentpaare.

Hinweis: Es reicht ein Programm vom Typ:

```
if P1 then A1;
if P2 then A2;
```

Übung 7.6:

Geben Sie — nach Wahl eines Spannbaumes — für den Kontrollflußgraphen von Abbildung 7.2 alle fundamentalen Zyklen und Wege an.

Übung 7.7:

(a) Geben Sie eine möglichst kleine Menge von vollständigen Wegen an, welche die schwache, normale und starke C_{GI}-Überdeckung für den Kontrollflußgraphen aus Abbildung 7.2 erfüllen.
Hinweis: Die Wege 0, 1, 2, 3, 4, 11 und 0, 1, 2, 3, 5, 6, 7, 11 durchlaufen die Schleife „einmal“, obwohl sie die Schleife frühzeitig mit *exit* verlassen.

(b) Geben Sie eine Menge vollständiger Wege an, welche die schwache C_{GI}-Überdeckung für den Textformatierer (aus Beispiel 7.3.1 bzw. Übung 7.1) erfüllt.

Hinweis: Ignorieren Sie bei den Teilaufgaben (a) und (b), ob Wege ausführbar sind oder nicht. Dieses Problem wird erst in Kapitel 11 behandelt.

Übung 7.8:

1. Zeigen Sie, daß einfache Wege stets 1-iterativ sind — aber nicht umgekehrt.
2. Ist der kürzeste Weg aus einer Klasse k-äquivalenter Wege stets k-iterativ? (Beachten Sie, daß bei beiden Begriffen die Ausführbarkeit der betrachteten Wege *nicht* gefordert wird.)

7.4 Verwendete Quellen und weiterführende Literatur

Das Kriterium **Zweigüberdeckung** wurde z. B. schon 1960 von Senko für Prozeduren (Unterprogramme) praktiziert (s. [Sen 60]). Das Überdeckungskriterium **Pfadüberdeckung** wurde von Howden formuliert und heißt bei Sneed **C_7-Überdeckung** (s. [How 87], [Sne 88]). Die C_{SP}-Überdeckung findet man schon in der Übersicht von Miller (s. [Mil 78]); sie wurde von mir zur $C_S(n)$-Überdeckung verallgemeinert und von Woodward et al. zur **LCMS-** (bzw. **LCSAJ-**)**Überdeckung** abgewandelt (s. [WHH 80]). Die Forderung, **alle fundamentalen Wege** bzw. eine entsprechende Anzahl von Tests auszuführen, stammt von McCabe (s. [McC 76], S. 318).
Auf Fehler beim Schleifendurchlauf und dafür geeignete Testverfahren haben McCracken und Baker schon 1957 hingewiesen (s. [Bak 57]). Der Begriff **Grenze-Inneres-Überdeckung** stammt von Howden, allerdings in unklarer Formulierung. Er wurde von mir für geschachtelte Schleifen definiert, um den Fall „0 Durchläufe" erweitert und in drei Varianten (s. Def. 7.2.11) angeboten (vgl. [How 75]). Das Kriterium schwache C_{GI}-Überdeckung entspricht dem Konzept der **Vorwärtspfade**, d. h. allen Pfaden ohne die Iteration von Schleifen (zitiert nach Sneed, s. [Sne 88]). Die **$C_i(k)$-Überdeckung** bzw. **strukturierte Pfadüberdeckung** wurde ebenfalls von Howden vorgeschlagen (s. [How 75]). Von Pimont und Rault wurde eine sogenannte H-Sprache zur Auswahl von Wegen in *while*- und *repeat*-Schleifen vorgeschlagen, welche dem Begriff **2-Äquivalenz** bei der **$C_i(2)$-Überdeckung** entspricht (s. [PiR 76]). Die Ideen zum Testen von geschachtelten, verketteten und schrecklichen Schleifen stammen von Beizer (s. [Bei 83], Kapitel 2.3). Die Begriffe **einfacher** und **2-Knoten-iterativer Weg** entsprechen den Begriffen *all-simple-paths* und *all-loop-iteration-free paths* von Linnenkugel/Müllerburg (s. [LiM 90]).
Mit dem Begriff **k-iterativ** hat Ntafos versucht, die strukturierte Pfadüberdeckung zu formalisieren, was aber — siehe Beispiel 7.2.11 und [Rie 92b] — mißlungen ist (vgl. [Nta 88], S. 869).

8 Datenflußbezogenes Testen

8.1 Problemstellung und Modellbildung

In Kapitel 7 wurde die Pfadüberdeckung als wünschenswertes Ziel formuliert, welches aber wegen des enormen Aufwands nicht realisierbar ist. Jeder überdeckte Pfad entspricht dabei einer Äquivalenzklasse von Eingabewerten, die diesen Pfad ausführen. Um festzustellen, ob das Programm für diese Klasse von Eingabewerten ein korrektes Verhalten aufweist, ist es also *notwendig*, wenigstens ein Testdatum pro Pfad auszuführen[1].
Die vorgeschlagenen Annäherungen an die Pfadüberdeckung (Zweigüberdeckung, Segmentpaareüberdeckung, LCMS-Überdeckung, $C_i(k)$-Überdeckung für einen kleinen Wert von k, Grenze-Inneres-Überdeckung) fordern nur die Ausführung von relativ *kurzen* Wegstücken. Daher ist eine Zuordnung zu der Äquivalenzklasse von Eingabewerten, die zu einem *vollständigen* Pfad gehört, nicht möglich. Der Beitrag der Berechnungen, die auf diesen Wegstücken stattfinden, zur gesamten Berechnung im Programm ist ebenfalls völlig unklar und hängt von der Kontrollstruktur auf dem Wegstück ab. Sinnvoller erscheint es daher, sich ausdrücklich um die Berechnung auf den Wegstücken zu kümmern und dabei folgendes Fehlermodell vor Augen zu haben: Wenn eine Anweisung ein falsches Ergebnis liefert, so kann dies folgende Ursachen haben:

1. Die Anweisung ist falsch.
2. Die Anweisung ist korrekt, aber die referenzierten Werte werden vorher falsch berechnet.

Beispiel 8.1.1
Die Anweisung sei $A := B + C * D$.

1. *Die Anweisung kann falsch sein, z. B. muß es vielleicht korrekt $A := B * C * D$ heißen.*
2. *Die referenzierten Werte B, C oder D können vorher falsch berechnet worden sein.*

[1] Dieses Verfahren ist natürlich *kein hinreichendes* Verfahren, um Fehler auszuschließen, da nur *stichprobenartig* getestet wird.

Der erste Fall wird in Kapitel 9 genauer behandelt. Im zweiten Fall muß man den Kontrollfluß rückwärts verfolgen und sich die Anweisungen ansehen, die B, C und D berechnen. Falls diese Anweisungen korrekt sind, muß man dieses Verfahren rekursiv fortsetzen[2].
Die Idee beim datenflußbezogenen Testen ist nun, die Interaktion zwischen Anweisungen, die den Wert einer Variablen berechnen (definieren), und Anweisungen, die diesen Variablenwert benutzen (referenzieren), entsprechend zu testen. Die einzelnen Methoden unterscheiden sich darin, ob alle diese Interaktionen oder nur ein bestimmter Teil davon getestet werden soll. Es lassen sich also verschiedene Überdeckungsmaße definieren, die alle mehr oder minder starke notwendige Bedingungen zum Aufdecken von Fehlern im oben beschriebenen „Datenfluß" darstellen.
Die Definition dieser Maße orientiert sich wieder am Modell des Kontrollflußgraphen eines Programms bzw. Moduls, der allerdings um gewisse Angaben erweitert wird und dann Datenflußgraph genannt wird.

Bei der folgenden Definition werden die Mengen *DEF(k)*, *UNDEF(k)* und *REF(k)* gerade so definiert, daß der Datenfluß *zwischen* verschiedenen Knoten verfolgt werden kann. Die Referenzen von vorher undefinierten Variablen werden bei $REF(k)$ nicht berücksichtigt, da sie Datenflußanomalien darstellen, die durch *statische* Analyse festgestellt werden können (genaueres siehe Kapitel 12.2).

Definition 8.1.1 (Datenflussgraph, Datenflussschema)

1. *Ein* **Datenflußgraph** *ist ein Kontrollflußgraph, bei dem zusätzlich zu jedem Knoten k die Mengen DEF(k), UNDEF(k) und REF(k) gehören.*

   DEF(k) *ist die Menge der Variablen x, für welche die Anweisungsfolge f, die zum Knoten k gehört*[3]*, der Variablen x einen Wert zuweist, der nicht anschließend in f undefiniert wird.*

   UNDEF(k) *ist die Menge der Variablen x, für welche die Anweisungsfolge f, die zum Knoten k gehört, die Variable x in einen undefinierten Zustand überführt, ohne x anschließend in f neu zu definieren.*

   REF(k) *ist die Menge der Variablen x, für welche die Anweisungsfolge f, die zum Knoten k gehört, die Variable x referenziert, ohne daß x vorher in f undefiniert wird. (Dabei wird generell vorausgesetzt, daß ein* lokaler *Datenfluß*[4] *innerhalb eines Knotens k nicht vorkommt. Andernfalls ist die Anweisungsfolge entsprechend auf zwei oder mehrere Knoten aufzuteilen.)*

---

[2] Ein entsprechendes Vorgehen beim Fehlerlokalisieren (debuggen) beschreibt Weiser in [Wei 82].

[3] Es kann sich — je nach Abstraktionsgrad des Kontrollflußgraphen – um eine oder mehrere sequentiell auszuführende Anweisungen handeln, die zum Knoten k gehören.

[4] Ein **lokaler** Datenfluß innerhalb eines Knotens k liegt vor, wenn in der zu k gehörenden Anweisungsfolge nach der *Definition* einer Variablen eine *Referenz* dieser Variablen vorkommt, ohne daß die Variable zwischenzeitlich undefiniert wird.

Wenn der Datenflußgraph eine Funktion repräsentiert, die von einem aufrufenden Modul Informationen erhält (über Parameter oder globale Variablen), so wird ein Knoten k_{ein} zum Kontrollflußgraphen hinzugefügt. DEF(k_{ein}) ist die Menge der Variablen, die Informationen importieren. Von k_{ein} führt eine Kante zu dem bisherigen Startknoten des Kontrollflußgraphen.
Entsprechendes gilt, wenn die beschriebene Funktion Informationen an das aufrufende Modul zurückgibt (über Parameter oder globale Variablen): Ein Knoten k_{aus} wird zum Kontrollflußgraphen hinzugefügt und von dem bisherigen Endknoten des Kontrollflußgraphen führt eine Kante zum Knoten k_{aus}. REF(k_{aus}) ist die Menge der Variablen, die Informationen exportieren.

2. *Das* **Datenflußschema** *zu einem Datenflußgraphen entsteht durch Ersetzen der Anweisungen und Entscheidungsprädikate durch formale Bezeichner.*

Falls es auf die Unterscheidung nicht ankommt, wird in beiden Fällen (1 und 2) von **Datenflußgraphen** *gesprochen.*

Beispiel 8.1.2
Buffer(bufpos) := c sei die Anweisung, die zum Knoten k gehört.
In diesem Fall sind die Variablen, die auf der linken und rechten Seite des Zuweisungssymbols stehen, verschieden. Obwohl bufpos links steht, wird es nur (als Index) referenziert. Daher gilt: DEF(k) = {Buffer}, REF(k) = {bufpos, c}, UNDEF(k) = leere Menge.

Beispiel 8.1.3
X := X + 1 sei die Anweisung, die zum Knoten k gehört.
In diesem Fall gilt: DEF(k) = REF(k) = {X}, UNDEF(k) = leere Menge, da X (auf der rechten Seite der Anweisung) referenziert wird, bevor X (auf der linken Seite) neu definiert wird.

Beispiel 8.1.4
$X := A + B;\ Y := C * D;\ B := Z;\ FREE(A);\ Y := A + Z;$
sei die Anweisungsfolge, die zum Knoten k gehört.
Dann ist DEF(k) = {B, X, Y}, UNDEF(k) = {A} und REF(k) = {A, B, C, D, Z}. A gehört zu REF(k), denn A wird in der ersten Anweisung (X:=A+B) benutzt und erst in der vierten Anweisung undefiniert. [Die Benutzung von A in Y:=A+Z nach der Anweisung FREE(A) ist ein „negativer Fall“ für die Definition von REF(k)].
Wird die dritte Anweisung (B:=Z) durch „B:=Y“ ersetzt, ändert sich (nur) die Menge REF(k) zu {A, B, C, D, $\underline{Y}$, Z}. In Knoten k wird aber kein Wert von Y referenziert, der außerhalb von k (vorher) definiert wird (nur das ist für die Bestimmung der auszuführenden Wege interessant), sondern nur ein Wert, der lokal in k in „$Y := C * D$“ definiert wird. Es gibt also einen lokalen Datenfluß zwischen $Y := C * D$ und $B := Y$.

Daher muß die Anweisungsfolge $X := A + B$; $Y := C * D$ dem bisherigen Knoten k und die Anweisungsfolge $B := Y$; FREE(A); $Y := A + Z$ einem neuen Knoten l zugeordnet werden, der Nachfolger von k ist. Für diese Knoten gilt: DEF(k) = $\{X, Y\}$, UNDEF(k) = leere Menge, REF(k) = $\{A, B, C, D\}$, DEF(l) = $\{B, Y\}$, UNDEF(l) = $\{A\}$, REF(l) = $\{Y, Z\}$, da Variable A bei der Referenz undefiniert ist. Variable Y wird also in beiden Knoten definiert.

Die Zuordnung von bedingten Anweisungen zu Knoten des Datenflußgraphen sei so gewählt, daß diese Knoten — genannt **Entscheidungsknoten** — nur Referenzen von Variablen enthalten, d. h. *DEF(k) = UNDEF(k)* = leere Menge. Falls dies nicht so ist, muß das Programm vorher — bei der Modellierung durch den Datenflußgraphen — verändert werden.

BEISPIEL 8.1.5
In der Programmiersprache C kann der Variablen, von der die Entscheidung abhängt, noch in der bedingten Anweisung ein Wert zugewiesen werden:

if *((B = C+D))* ...

Dies muß im Datenflußgraphen durch zwei Knoten modelliert werden, wobei der zweite Knoten einziger Nachfolger des ersten ist.

Knoten 1: B = C+D;
Knoten 2: **if** *B* ...

Für eine Variable x wird noch folgende Sprechweise eingeführt:
„**Eine Definition** von x im Knoten k **erreicht eine Referenz** von x im Knoten l über den Weg w“ g. d. w. der Weg w im Kontrollflußgraphen von k nach l führt und die Variable x auf dem Weg nicht neu definiert oder undefiniert wird[5].
Mit diesen vorbereitenden Begriffen können nun die Überdeckungkriterien definiert werden.

8.2 Einfache Datenflußkriterien

Das folgende Kriterium beschreibt die Notwendigkeit von Testdaten, bei denen das Resultat jeder Zuweisung (Definition) wenigstens einmal benutzt wird.

DEFINITION 8.2.1
Eine Testdatenmenge T erfüllt das Kriterium **alle Definitionen** *g. d. w. es für jede Variable x und jede Definition von x* mindestens einen *Weg in Wege(T) gibt, auf dem die Definition eine Referenz von x erreicht.*

[5] Eine formale Definition dieses Sachverhalts lautet:
Sei $x \in DEF(k)$, $x \in REF(l)$, $w = (k_1, \ldots, k_m)$ ein Weg im Datenflußgraphen mit $k_1 = k, k_m = l$. Dann gilt für jedes i mit $1 < i < m : x \notin DEF(k_i), x \notin UNDEF(k_i)$.

Mit dem Kriterium *alle Definitionen* wird zu einer Variablendefinition nur eine Interaktion mit *einer* Referenz über *einen* bestimmten Weg getestet. Es werden also nicht alle Paare von Definitionen und Referenzen einer Variablen getestet, obwohl jede Referenz der Variablen fehlerhaft sein kann.

Beispiel 8.2.1
Sei T die Testdatenmenge, die in dem Flußdiagramm aus Abbildung 8.1 den Weg (A1, A2, P1, a, *P2, A3) ausführt. T erfüllt das Kriterium „alle Definitionen" für die Variablen X und B. Die Referenz von X in A4 wird aber nicht getestet.*

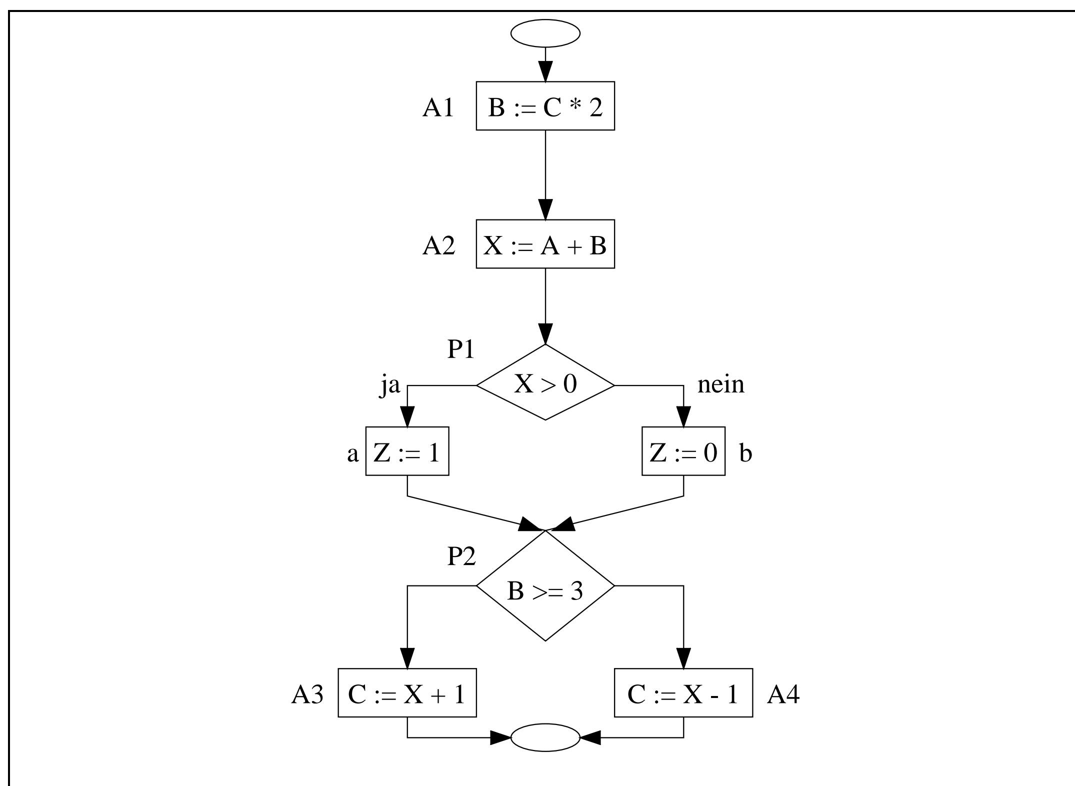


Abb. 8.1: Flußdiagramm mit Definitionen und Referenzen von B und X

Ein schärferes Kriterium, welches die genannte Anforderung erfüllt, ist das folgende.

Definition 8.2.2
Eine Testdatenmenge T erfüllt das Kriterium **alle DR-Interaktionen** *g. d. w. es für jede Variable x, jede Definition von x und jede Referenz von x, die davon erreicht wird,* mindestens einen *Weg in Wege(T) gibt, auf dem die Definition die Referenz von x erreicht.*

Mit dem Kriterium *alle DR-Interaktionen* werden also *alle* Paare von Definitionen und Referenzen einer Variablen getestet. Mit dieser Strategie werden allerdings nicht alle Entscheidungskanten ausgeführt.

BEISPIEL 8.2.2
Sei T die Testdatenmenge, die in dem Flußdiagramm von Abbildung 8.1 die Wege (A1, A2, P1, a, P2, A3) und (A1, A2, P1, a, P2, A4) ausführt. T erfüllt das Kriterium „alle DR-Interaktionen" für beide Variablen X und B. Entscheidungskante b wird aber nicht ausgeführt.

Da es für die Referenz einer Variablen in einem Entscheidungsknoten von Bedeutung ist, wie der *Ausgang* der Entscheidung ist, wird bei dem folgenden Kriterium die Referenz einer Variablen diesen ausgehenden Kanten zugeordnet.

DEFINITION 8.2.3
Eine Testdatenmenge T erfüllt das Kriterium **alle Referenzen** *g. d. w. für jede Variable x, jede Definition von x in einem Knoten k, jede Referenz von x in einem Knoten l, die von der Definition in k erreicht wird, und für jeden Nachfolgerknoten m von l die Wegemenge Wege(T)* mindestens ein *Wegstück $u * m$* [6] *enthält, wobei die Definition von x in k die Referenz von x in l über den Weg u erreicht.*

BEISPIEL 8.2.3
Sei T die Testdatenmenge, die in dem Flußdiagramm von Abbildung 8.1 die Wege (A1, A2, P1, a, P2, A3) und (A1, A2, P1, b, P2, A4) ausführt. T erfüllt das Kriterium „alle Referenzen" für beide Variablen X und B.
Die Wege (A1, A2, P1, a, P2, A4) und (A1, A2, P1, b, P2, A3) werden aber nicht ausgeführt.

Mit den Kriterien *alle DR-Interaktionen* und *alle Referenzen* werden alle Paare von Definitionen und Referenzen einer Variablen getestet, aber jeweils nur auf *einem* Weg von der Definition zur Referenz. Unter dem Gesichtspunkt des Datenflusses zwischen der Definition und der Referenz einer Variablen x ist dies natürlich ausreichend, da auf den Wegen zwischen Definition und Referenz keine Veränderung der Variablen x erfolgt, also auch kein Fehler bzgl. x auftreten kann.

Wie immer beim Testen muß nun eine Entscheidung zwischen zwei allgemeinen Anforderungen getroffen werden:

1. möglichst starke Forderungen an die Testdatenmenge, damit viele Fehler gefunden werden können;

[6] $u * m$ sei definiert als ein Wegstück, welches als Knotenfolge dargestellt ist. Es enthält die Knotenfolge $u = (k, \ldots, l)$ gefolgt von einem Knoten m, d. h. $u * m = (k, \ldots, l, m)$.
(Falls l keinen Nachfolger hat, muß auch Wege(T) nur das Wegstück u enthalten.)
Beispiel: Definition in $k = A$, Referenz in $l = C$, $u = (A, B, C)$, $m = D$.
Dies ergibt $u * m = (A, B, C, D)$.

2. möglichst geringe Forderungen an die Testdatenmenge, damit der Testaufwand nicht zu groß wird.

Als eine schwächere Überdeckung des Datenflusses, die dennoch die Zweigüberdeckung und das Testen aller Definitionen gewährleistet, bietet sich folgendes an:

Definition 8.2.4
Eine Testdatenmenge T erfüllt das Kriterium **alle Entscheidungsreferenzen/ einige Berechnungsreferenzen** *bzw. kurz:* **alle E-/einige B-Referenzen** *g. d. w. für jede Variable x und jede Definition von x in einem Knoten k gilt:*

1. *für jede Referenz von x in einem Entscheidungsknoten (im folgenden l genannt), die von der Definition in k erreicht wird, und jeden Nachfolger m von l enthält Wege(T)* mindestens *ein Wegstück $u * m$, wobei die Definition von x in k die Referenz von x in l über u erreicht;*

2. *falls es keine Referenzen von x in Entscheidungsknoten gibt, die von der Definition in k erreicht werden, enthält Wege(T)* mindestens *ein Wegstück u, auf dem die Definition von x in k irgendeine Referenz von x erreicht.*

Mit Teil 1 des obigen Kriteriums wird die Zweig- oder Entscheidungsüberdeckung erreicht. Mit Teil 2 wird das Kriterium *alle Definitionen* erfüllt.

Beispiel 8.2.4
T sei die Testdatenmenge, die in dem Flußdiagramm von Abbildung 8.1 die Wege (A1,A2,P1,a,P2,A3) und (A1,A2,P1,b,P2,A3) ausführt. T erfüllt das Kriterium „alle E-/einige B-Referenzen" für die Variable X. Die Berechnungsreferenz von X in A4 wird dabei nicht getestet.

Wenn bei der Überdeckung des Datenflusses die Zweigüberdeckung vernachlässigt wird, aber das Testen aller Definitionen und das Testen aller Referenzen in *Berechnungsknoten* (also nicht in Entscheidungsknoten) gewährleistet werden soll, bietet sich das folgende Kriterium an:

Definition 8.2.5
Eine Testdatenmenge T erfüllt das Kriterium **alle Berechnungsreferenzen/ einige Entscheidungsreferenzen** *bzw. kurz:* **alle B-/einige E-Referenzen** *g. d. w. für jede Variable x und jede Definition von x in einem Knoten k gilt:*

1. *für jede Referenz von x in einem Knoten (im folgenden l genannt), der nicht Entscheidungsknoten ist, aber von der Definition in k erreicht wird, enthält Wege(T)* mindestens *ein Wegstück u, wobei die Definition von x in k die Referenz von x in l über u erreicht;*

2. *falls die Definition von x in k nur in Entscheidungsknoten referenziert wird, enthält Wege(T)* mindestens *ein Wegstück u, auf dem die Definition von x in k irgendeine Referenz von x erreicht.*

Mit Teil 1 des obigen Kriteriums wird erreicht, daß das Kriterium *alle Referenzen* (jedenfalls für Berechnungsknoten) erfüllt ist. Mit Teil 2 wird das Kriterium *alle Definitionen* garantiert.

Beispiel 8.2.5
T sei die Testdatenmenge, die in dem Flußdiagramm von Abbildung 8.1 den Weg (A1, A2, P1, a, *P2, A3) ausführt. T erfüllt das Kriterium „alle B-/einige E-Referenzen" für die Variable B. Die Referenz von B auf der Entscheidungskante von P2 nach A4 wird dabei nicht getestet. Entsprechendes gilt, wenn man den obigen Weg durch den Weg (A1, A2, P1, b, P2, A3) ersetzt.*

Wenn eine stärkere Überdeckung unter Kontrollflußgesichtspunkten erreicht werden soll, bietet sich eine Annäherung an das Pfadtesten an, bei dem allerdings keine Iterationen von Schleifen vorgenommen werden.

Definition 8.2.6
Eine Testdatenmenge T erfüllt das Kriterium **alle DR-Wege** *g. d. w. für jede Variable x, jede Definition von x in einem Knoten k, jede Referenz von x in einem Knoten l, die von der Definition in k erreicht wird, und für jeden Nachfolgerknoten m von l die Wegemenge Wege(T)* jedes *Wegstück $u*m$*[7] *mit folgenden Eigenschaften enthält:*

1. *die Definition von x in k erreicht die Referenz von x in l über den Weg u,*

2. *u ist frei von Zyklen oder u ist ein einfacher Zyklus*[8]*.*

Mit dem Kriterium *alle DR-Wege* werden also für alle Paare von Definitionen und Referenzen (fast) alle Wege getestet, auf denen die Interaktion möglich ist. Außerdem werden wiederum alle Nachfolgerzweige mit einbezogen, falls eine Referenz in einem Entscheidungsknoten auftritt. Die Einschränkung „(fast) alle Wege" bezieht sich dabei auf die Ausnahme von Zyklen, die nicht einfach sind.

Beispiel 8.2.6 (alle DR-Wege für Abbildung 8.1)
Nur eine Testdatenmenge T, die alle vier Wege des Flußdiagramms von Abb. 8.1 ausführt, erfüllt das Kriterium „alle DR-Wege" für beide Variablen X und B.

Da das Programm aus Abbildung 8.1 keine Zyklen enthält, wird im folgenden Beispiel noch das Programm aus Abbildung 7.2 betrachtet.

[7] siehe Fußnote 6 zu Definition 8.2.3
[8] Zum Begriff (einfacher) Zyklus siehe Definition 7.1.4 auf Seite 191.

Beispiel 8.2.7 (alle DR-Wege für Abbildung 7.2)
Die folgenden Wegstücke sind für das Programm aus Abb. 7.2 als DR-Wege zu betrachten. Dabei sind die Knoten, in denen die entsprechenden Variablen definiert und referenziert werden, durch Fettdruck hervorgehoben.

für Variable N:

1.	(**0**,**1**,2)		

für Variable F und A:

2a.	(**0**,1,2,3,**5**,6)	3a.	(**0**,1,2,3,5,**8**,9)
2b.	(**0**,1,2,3,**5**,8)	3b.	(**0**,1,2,3,5,**8**,10)

für Variable NPOS:

4.	(**1**,2,3,4,**11**)		

für Variable LOW und HIGH:

5.	(**1**,**2**,3)	6a.	(**1**,2,**3**,4)
		6b.	(**1**,2,**3**,5)

für Variable MID:

7a.	(**2**,3,**5**,6)	9a.	(**2**,3,5,**8**,9)
7b.	(**2**,3,**5**,8)	9b.	(**2**,3,5,**8**,10)
8.	(**2**,3,5,**6**,7)	10.	(**2**,3,5,8,**9**,2)
		11.	(**2**,3,5,8,**10**,2)

für Variable NPOS:

12.	(**6**,7,**11**)		

für Variable LOW:

13.	(**9**,**2**,3)	14a.	(**9**,2,**3**,4)
		14b.	(**9**,2,**3**,5)

für Variable HIGH:

15.	(**10**,**2**,3)	16a.	(**10**,2,**3**,4)
		16b.	(**10**,2,**3**,5)

Diese Wegstücke sind in den folgenden vollständigen Wegen enthalten[9]*:*

w_1:	(0,1,2,3,5,6,7,11)	für 1,2a,5,6b,7a,8,12
w_2:	(0,1,2,3,4,11)	für 4,6a
w_3:	(0,1,2,3,5,8,9,2,3,5,8,10,2,3,5,8,9,2,3,4,11)	für 3a, 9a, 10, 13, 14a, 14b, 16b
w_4:	(0,1,2,3,5,8,10,2,3,4,11)	für 2b, 3b, 7b, 9b, 11, 15, 16a

Die Wege w_1 bis w_4 werden z. B. von den folgenden Testdaten t_1 bis t_4 ausgeführt:

t_1: *N = 1, F = A(1) = 2;*

t_2: *N = 0, restliche Werte beliebig;*[10]

[9] Falls ein Wegstück in mehreren Wegen enthalten ist, wird es hier nur bei einem der Wege angeführt, z. B. Wegstück 2b nur bei w_4, obwohl es auch in w_3 enthalten ist.

[10] Falls ein Array mit Länge N=0 nicht erlaubt ist (laut Eingabespezifikation), sind der Weg w_2 und die *DR*-Wege 4 und 6a nicht ausführbar.

t_3: N = 7, A(4) = 0, A(5) = 1, F = 2, A(6) = 3;
t_4: N = 1, F = 2, A(1) = 3.

Die Testdatenmenge T mit den Testdaten t_1, t_2, t_3, t_4 erfüllt also das Kriterium „alle DR-Wege“ für das Programm aus Abbildung 7.2.

8.3 Verkettung von Datenflüssen

Im folgenden wird eine Schar von immer stärkeren Kriterien vorgestellt, die unter Datenflußgesichtspunkten über die Anforderung *alle Referenzen* hinausgehen. Dabei wird die *Verkettung* von Wegen gefordert, auf denen jeweils die Definition einer Variablen eine Referenz erreicht. Dies entspricht gerade der (am Anfang dieses Kapitels vorgestellten) Rückwärtsverfolgung von Fehlern. Die Schar von immer stärkeren Kriterien ergibt sich durch Anforderungen an die Länge k dieser Ketten von Definitionen und Referenzen.

Definition 8.3.1
Für $k \geq 2$ heißt eine Folge $I = (l_1, x_1, l_2, x_2, \ldots, x_{k-1}, l_k)$ eine **k-DR-Interaktion** *genau dann wenn*

1. *$l_1, l_2, \ldots, l_k$ Knoten des Datenflußgraphen sind, wobei nur die Knoten l_1 und l_k identisch sein dürfen,*
2. *$x_1, \ldots, x_{k-1}$ Variable sind (die nicht notwendigerweise verschieden sind),*
3. *es für jedes i mit $1 \leq i < k$ einen Weg w_i im Datenflußgraphen gibt, so daß gilt: Variable x_i wird im Knoten l_i definiert und diese Definition erreicht eine Referenz von x_i in Knoten l_{i+1} über den Weg w_i.*

Definition 8.3.2
Sei $I = (l_1, x_1, \ldots, x_{k-1}, l_k)$ eine k-DR-Interaktion.
Ein Weg im Datenflußgraph der Form $w = w_1 \ldots w_{k-1}$ ist ein **Interaktionsweg für I** *g. d. w. für jedes i mit $1 \leq i < k$ w_i ein Weg ist, über den eine Definition von x_i im Knoten l_i eine Referenz von x_i im Knoten l_{i+1} erreicht.*

Beispiel 8.3.1
Für das Programm aus Abbildung 7.2 gilt folgendes:
$I_1 = (1, NPOS, 11)$ ist eine 2-DR-Interaktion mit der Knotenfolge (1,2,3,4,11) als Interaktionsweg.
$I_2 = (1, LOW, 2, MID, 6, NPOS, 11)$ ist eine 4-DR-Interaktion mit dem Interaktionsweg (1,2,3,5,6,7,11).
$I_3 = (2, MID, 10, HIGH, 2)$ ist eine 3-DR-Interaktion mit dem Interaktionsweg (2,3,5,8,10,2).
Damit beispielsweise der Interaktionsweg für I_2 ausgeführt wird, müssen als Testdaten etwa A = (1,5,7), N = 3, F = 5 gewählt werden.

Falls ein Knoten mehrere „unabhängige“ Anweisungen für verschiedene Variablen enthält, beschreiben die oben definierten Interaktionswege die Übergabe der definierten Variablenwerte an andere Variable *nicht* richtig. Daher sind in diesen Fällen die Anweisungen aufzuteilen auf verschiedene Knoten. Im folgenden soll daher jedem Knoten stets nur eine Anweisung zugeordnet sein.

Beispiel 8.3.2
Für das Programm aus Abbildung 7.1 wäre nach Definition 8.3.2 die Folge (A, x, F, y, C) *eine 3-DR-Interaktion mit der Knotenfolge* (A, F, C) *als Interaktionsweg:* x *wird in* A *definiert und erreicht eine Referenz in* F, y *wird in* F *definiert und erreicht eine Referenz in* C. *Tatsächlich erfolgt in* F *aber keine Verwendung des Wertes von* x *für die Berechnung von* y, *da die beiden Anweisungen* $y := y \;\; div \;\; 2$ *und* $x := x * x$ *unabhängig voneinander sind.* F *ist also aufzuteilen in die Knoten* $F1$ *mit Anweisung* $y := y \;\; div \;\; 2$ *und* $F2$ *mit Anweisung* $x := x * x$.

Nach diesen Vorbereitungen kann das erweiterte Datenflußkriterium *alle k-DR-Interaktionen* formal definiert werden.

Definition 8.3.3
Sei k *eine natürliche Zahl,* $k \geq 2$.
Eine Testdatenmenge T *erfüllt das Kriterium* **alle k-DR-Interaktionen** *g. d. w. für jedes* m *mit* $2 \leq m \leq k$, *für jede* m*-DR-Interaktion* $I = (l_1, x_1, \ldots, x_{m-1}, l_m)$ *und für jeden Nachfolgerknoten* n *von* l_m *die Menge Wege(T) mindestens ein Wegstück* $u * n$ [11] *enthält, wobei* u *ein Interaktionsweg für* I *ist.*

Für große Werte von k werden bei obigem Kriterium längere (aber nicht so viele) Wegstücke als (wie) beim Kriterium *alle DR-Wege* (s. Definition 8.2.6) gefordert; für $k = 2$ wird aber weniger gefordert, da für jede 2-DR-Interaktion *ein* Wegstück $u * n$ in Wege(T) ausreichend ist. Damit ist für $k = 2$ obiges Kriterium *alle k-DR-Interaktionen* mit dem Kriterium *alle Referenzen* (s. Definition 8.2.3) äquivalent.

Beispiel 8.3.3
Für das Programm aus Abb. 7.2 sind „alle DR-Wege“ aus Beispiel 8.2.7 auszuführen, um das Kriterium „alle k-DR-Interaktionen“ für $k = 2$ *zu erfüllen. Dies gilt, da es für alle 2-DR-Interaktionen im Programm aus Abb. 7.2 stets nur einen Interaktionsweg gibt, der zyklenfrei oder ein einfacher Zyklus ist. Daher ist mit dem Kriterium „alle Referenzen“ und „alle 2-DR-Interaktionen“ auch „alle DR-Wege“ erfüllt.*

Folgende Wegstücke sind zusätzlich auszuführen, um das Kriterium „alle 3-DR-Interaktionen“ zu erfüllen. Dabei sind die Knoten, in denen die entsprechenden Variablen definiert und referenziert werden, durch Fettdruck hervorgehoben.

[11] siehe Fußnote 6 zu Definition 8.2.3

für die Variablen N und HIGH[12]*:*

17. (**0**,**1**,**2**,*3*)

18a. (**0**,**1**,*2*,**3**,*4*)
18b. (**0**,**1**,*2*,**3**,*5*)

für die Variablen LOW bzw. HIGH und MID:

19a. (**1**,**2**,*3*,**5**,*6*)
19b. (**1**,**2**,*3*,**5**,*8*)
20. (**1**,**2**,*3*,*5*,**6**,*7*)

21a. (**1**,**2**,*3*,*5*,**8**,*9*)
21b. (**1**,**2**,*3*,*5*,**8**,*10*)
22. (**1**,**2**,*3*,*5*,*8*,**9**,*2*)
23. (**1**,**2**,*3*,*5*,*8*,**10**,*2*)

für die Variablen MID und NPOS:

24. (**2**,*3*,*5*,**6**,*7*,**11**)

für die Variablen MID und LOW:

25. (**2**,*3*,*5*,*8*,**9**,**2**,*3*)

26a. (**2**,*3*,*5*,*8*,**9**,*2*,**3**,*4*)
26b. (**2**,*3*,*5*,*8*,**9**,*2*,**3**,*5*)

für die Variablen MID und HIGH:

27. (**2**,*3*,*5*,*8*,**10**,**2**,*3*)

28a. (**2**,*3*,*5*,*8*,**10**,*2*,**3**,*4*)
28b. (**2**,*3*,*5*,*8*,**10**,*2*,**3**,*5*)

für die Variablen LOW und MID:

29a. (**9**,**2**,*3*,**5**,*6*)
29b. (**9**,**2**,*3*,**5**,*8*)
30. (**9**,**2**,*3*,*5*,**6**,*7*)

31a. (**9**,**2**,*3*,*5*,**8**,*9*)
31b. (**9**,**2**,*3*,*5*,**8**,*10*)
32. (**9**,**2**,*3*,*5*,*8*,**9**,*2*)
33. (**9**,**2**,*3*,*5*,*8*,**10**,*2*)

für die Variablen HIGH und MID:

34a. (**10**,**2**,*3*,**5**,*6*)
34b. (**10**,**2**,*3*,**5**,*8*)
35. (**10**,**2**,*3*,*5*,**6**,*7*)

36a. (**10**,**2**,*3*,*5*,**8**,*9*)
36b. (**10**,**2**,*3*,*5*,**8**,*10*)
37. (**10**,**2**,*3*,*5*,*8*,**9**,*2*)
38. (**10**,**2**,*3*,*5*,*8*,**10**,*2*)

Nur die Wegstücke 29a, 30, 31a, 32, 34a, 35, 36b und 38 sind nicht in den Wegen w_1 *bis* w_4 *von Beispiel 8.2.7 enthalten, werden also nicht durch die Testdaten* t_1 *bis* t_4 *von Beispiel 8.2.7 ausgeführt.*
Die nicht ausgeführten Wegstücke sind aber in den folgenden vollständigen Wegen w_5 *und* w_6 *enthalten und lassen sich z. B. mit den Testdaten* t_5 *und* t_6 *ausführen:*

w_5: *(0,1,2,3,5,8,9,2,3,5,8,9,2,3,5,6,7,11)*
w_6: *(0,1,2,3,5,8,10,2,3,5,8,10,2,3,5,6,7,11)*
t_5: $N = 5$, $A[3] = 6$, $A[4] = 7$, $A[5] = F = 8$,
t_6: $N = 7$, $F = A[1] = 1$, $A[2] = 2$, $A[4] = 3$.

Die Testdatenmenge T *mit den Testdaten* $t_1, t_2, t_3, t_4, t_5, t_6$ *erfüllt also das Kriterium „alle 3-DR-Interaktionen" für das Programm aus Abbildung 7.2.*

[12] Knoten 1 ist eigentlich aufzuteilen, damit die Referenz von *N* nur zur Definition von *HIGH* — und nicht von *NPOS* und *LOW* — verwendet wird. Dies wurde hier aber implizit berücksichtigt.

8.4 Parallele Betrachtung von Datenflüssen

Zu Beginn dieses Kapitels wurde die Rückwärtsverfolgung eines Fehlers in einer Anweisung, z. B. $A := B + C * D$, als Motiv für die Betrachtung des Datenflusses angeführt. Der Ansatz *alle k-DR-Interaktionen* nimmt einen Fehler in einer der Variablen B, C oder D an und verfolgt dies jeweils getrennt für B, C und D in langen Ketten von Definitionen und sie erreichenden Referenzen. Bei den im folgenden vorgestellten Kriterien werden dagegen Wege verfolgt, auf denen *gleichzeitig* die letzten Definitionen von B, C und D die Referenz in der betrachteten Anweisung $A := B + C * D$ erreichen.

Der Zusammenhang zwischen den Referenzen von Variablen in einer Anweisung und den zugehörigen Definitionen dieser Variablen wird folgendermaßen definiert:

DEFINITION 8.4.1 (DEFINITIONSKONTEXT, KONTEXTWEG)
Eine Menge $DK = \{(k_1, x_1), (k_2, x_2), \ldots, (k_n, x_n)\}$ *ist ein* **Definitionskontext** *für einen Knoten* k *des Datenflußgraphen mit* $REF(k)$[13] $= \{x_1, \ldots, x_n\} \neq \emptyset$, *wenn folgendes gilt:*

1. *für jedes* i *mit* $1 \leq i \leq n$ *ist* k_i *ein Knoten, in dem die Variable* x_i *definiert wird,*
2. *es existiert ein Weg* w, *genannt* **Kontextweg** *für* DK, *mit folgender Eigenschaft für jedes* i *mit* $1 \leq i \leq n$:

   es gibt eine Aufteilung von w *in zwei Wegstücke* w_i *und* v_i $(w = w_i v_i)$, *so daß die Definition von* x_i *im Knoten* k_i *die Referenz von* x_i *in* k *über den Weg* v_i *erreicht (vgl. Sprechweise nach Beispiel auf Seite 215).*

Der Kontextweg für einen Definitionskontext eines Knotens k enthält also alle Definitionen der Variablen, die in k referenziert werden. (Da die Definitionen von Variablen i. allg. an verschiedenen Stellen in w vorkommen, muß w in obiger Definition 8.4.1 in w_i und v_i aufgeteilt werden, um die jeweilige Definition von x_i zu markieren.)

BEISPIEL 8.4.1 (DEFINITIONSKONTEXT)
Für das Suchprogramm aus Abbildung 7.2 existieren für Knoten 2 mit der Anweisung $MID := \lfloor (LOW + HIGH)/2 \rfloor$ *folgende Definitionskontexte:*

$DK_1 = \{(1, LOW), (1, HIGH)\}$ *für den Kontextweg (1, 2);*
$DK_2 = \{(1, LOW), (10, HIGH)\}$ *für den Kontextweg (1, 2, 3, 5, 8, 10, 2);*
$DK_3 = \{(9, LOW), (1, HIGH)\}$ *für den Kontextweg (1, 2, 3, 5, 8, 9, 2);*
$DK_4 = \{(9, LOW), (10, HIGH)\}$ *für den Kontextweg (9, 2, 3, 5, 8, 10, 2) oder den Kontextweg (10, 2, 3, 5, 8, 9, 2).*

[13] vgl. Definition 8.1.1

Für Knoten 2 gibt es also vier verschiedene Definitionskontexte für die beiden referenzierten Variablen LOW und HIGH, da sie in Knoten 1, 9 oder 10 definiert werden können. Für den vierten Definitionskontext gibt es sogar verschiedene Kontextwege, da die Reihenfolge der Definitionen von LOW und HIGH nicht festgelegt ist.

Mit den Begriffen *Definitionskontext* und *Kontextweg* aus Definition 8.4.1 kann nun das Überdeckungskriterium formuliert werden, welches die Ausführung von mindestens einem Kontextweg pro Definitionskontext verlangt.

DEFINITION 8.4.2
Eine Testdatenmenge T erfüllt das Kriterium **Kontextüberdeckung** *g. d. w. für jeden Knoten k des Datenflußgraphen und jeden Definitionskontext DK von k die Wegemenge Wege(T)* mindestens *ein Wegstück enthält, welches Kontextweg für DK ist.*

Eine Verschärfung des obigen Kriteriums erhält man, wenn man den Definitionskontext als *geordnete Folge* der Tupel (k_1, x_1), $(k_2, x_2), \ldots, (k_n, x_n)$ betrachtet und für einen **geordneten Kontextweg** verlangt, daß die Knoten $k_1, k_2, \ldots, k_n$ in dieser Reihenfolge auf dem Kontextweg vorkommen. Zu einem (ungeordneten) Definitionskontext gemäß Definition 8.4.1 kann man also eventuell mehrere **geordnete Definitionskontexte** erhalten. Das Kriterium **geordnete Kontextüberdeckung** fordert dann gerade die Ausführung von mindestens einem geordneten Kontextweg für jeden geordneten Definitionskontext.

BEISPIEL 8.4.2 (GEORDNETER DEFINITIONSKONTEXT)
Für das Suchprogramm aus Abbildung 7.2 ergibt sich für Knoten 2 mit der Anweisung $MID := \lfloor (LOW + HIGH)/2 \rfloor$ *nur im Falle des Definitionskontextes* DK_4 *ein Unterschied zu Beispiel 8.4.1, da die Reihenfolge der Knoten auf den Kontextwegen in den anderen drei Fällen eindeutig festgelegt ist. Für* $DK_4 = \{(9, LOW), (10, HIGH)\}$ *gibt es die folgenden geordneten Definitionskontexte, die bei der geordneten Kontextüberdeckung beide auszuführen sind.*

$GDK_1 = ((9, LOW), (10, HIGH))$ *mit Kontextweg (9, 2, 3, 5, 8, 10, 2),*
$GDK_2 = ((10, HIGH), (9, LOW))$ *mit Kontextweg (10, 2, 3, 5, 8, 9, 2).*

Die folgende Strategie vereinigt die Strategie *alle k-DR-Interaktionen*, bei der lange Ketten von Definitionen und Referenzen untersucht werden, mit dem Ansatz der Kontextüberdeckung: Beim sogenannten **Definitionsbaumtesten** wird eine Teilmenge V der Ausgabevariablen des Programms ausgewählt. Der Datenfluß für die Variablen aus V wird dann von den Ausgabeanweisungen durch eine *Folge* von Definitionskontexten zurückverfolgt bis zum Beginn des Programms oder bis eine zyklische Benutzung der Variablen erreicht ist. Durch diese Strategie kann also — wie bei *alle k-DR-Interaktionen* — eine komplette Berechnungsfolge getestet werden und ihr Effekt mit der Spezifikation verglichen werden.

BEISPIEL 8.4.3 (DEFINITIONSBAUMTESTEN)
Für das Suchprogramm aus Abbildung 7.2 ist Anweisung 11 (output NPOS) die einzige Ausgabeanweisung. Definitionskontexte dafür sind $DK_{11,6} = \{(6, NPOS)\}$ *und* $DK_{11,1} = \{(1, NPOS)\}$. *Für* $DK_{11,1}$ *mit Kontextweg* $w = 1, 2, 3, 4, 11$ *ist nichts zurückzuverfolgen, für* $DK_{11,6}$ *sind dagegen die Definitionskontexte für die Referenz von MID im Knoten 6 zu ermitteln. Das ergibt nur* $DK_{6,2} = \{(2, MID)\}$ *und (vorläufig) den auszuführenden Weg* $w = 2, 3, 5, 6, 7, 11$. *Da in Knoten 2 die Variablen LOW und HIGH referenziert werden, ergibt das die vier Definitionskontexte aus Beispiel 8.4.1 und somit die folgenden auszuführenden Wege, wobei Anfang und Ende der neuen vier Kontextwege unterstrichen sind:*

$w_1 = (\underline{1}, \underline{2}, 3, 5, 6, 7, 11)$,
$w_2 = (\underline{1}, 2, 3, 5, 8, 10, \underline{2}, 3, 5, 6, 7, 11)$,
$w_3 = (\underline{1}, 2, 3, 5, 8, 9, \underline{2}, 3, 5, 6, 7, 11$,
$w_4 = (\underline{9}, 2, 3, 5, 8, 10, \underline{2}, 3, 5, 6, 7, 11)$ *oder* $w_4 = (\underline{10}, 2, 3, 5, 8, 9, \underline{2}, 3, 5, 6, 7, 11)$ *(je nach Wahl des Kontextweges für* DK_4 *in Beispiel 8.4.1).*

Die Wege w_1, w_2, w_3 *sind nicht mehr „nach vorn" verlängerbar, nur für* w_4 *ergibt sich für Knoten 9 bzw. 10 eine weitere Referenz von MID mit dem Definitionskontext* $\{(2, MID)\}$ *und somit für die beiden Alternativen (bei* w_4*) die zu testenden Wege*

$w = (\underline{2}, 3, 5, 8, \underline{9}, 2, 3, 5, 8, 10, 2, 3, 5, 6, 7, 11)$ *und*
$w' = (\underline{2}, 3, 5, 8, \underline{10}, 2, 3, 5, 8, 9, 2, 3, 5, 6, 7, 11)$.

Die Weiterverfolgung der Definitionskontexte DK_1 *bis* DK_4 *(aus Beispiel 8.4.1) führt dann zum „Abbruch" (Wegbeginn bei Knoten 1) bzw. zur Iteration (Wegbeginn bei Knoten 9 und 10).*

8.5 Übungen

Übung 8.1:

Geben Sie das Datenflußschema für den Textformatierer aus Beispiel 7.3.1 an.
Hinweis: Geben Sie als formale Bezeichner für die Knoten die Zeilennummern aus Beispiel 7.3.1 (auf S. 208) an, wobei aufzuspaltende Knoten (z. B. 15) als 15a, 15b etc. zu bezeichnen sind. Neben den Knoten sind die Mengen *DEF* und *REF* zu notieren. Gibt es nichtleere Mengen *UNDEF*?

Übung 8.2:

(a) Geben Sie für folgende Programme Wegstücke an, die das Kriterium *alle Definitionen* (s. Def. 8.2.1) erfüllen:

  i. das Programm aus Abbildung 7.1 (Berechnung von $z = x^{|y|}$);

  ii. den Textformatierer aus Beispiel 7.3.1 bzw. Übung 8.1.

(b) Geben Sie zusätzliche Wegstücke [zu den Wegstücken aus (a)] an, damit auch das Kriterium *alle DR-Interaktionen* (s. Def. 8.2.2) erfüllt ist (für beide Programme aus i und ii).

Hinweis: Lassen Sie in beiden Fällen [(a) und (b)] außer acht, ob die angegebenen Wegstücke ausführbar sind oder nicht. Fassen Sie Wegstücke zusammen, die den gleichen Anfang haben. Zur Ermittlung der zu betrachtenden Paare von Definitionen und Referenzen empfiehlt sich die Aufstellung einer „Kreuz-Referenz"-Tabelle, die für jede Variable die Zeilen angibt, wo sie definiert bzw. referenziert wird.

Übung 8.3:

(a) Geben Sie für folgende Programme Wegstücke an, die das Kriterium *alle Referenzen* (s. Definition 8.2.3) erfüllen:

  i. das Programm aus Abbildung 7.1;

  ii. den Textformatierer aus Beispiel 7.3.1 bzw. Übung 8.1.

(b) Welche Wegstücke der Lösung zu i können jeweils weggelassen werden, wenn als Kriterium nur *alle E-/einige B-Referenzen* oder nur *alle B-/einige E-Referenzen* verlangt wird?

Hinweis: Teilen Sie in der Kreuz-Referenz-Tabelle (aus Übung 8.2) die Referenzen in Entscheidungs- oder Berechnungs-Referenzen auf, damit Sie die zu erfüllenden Paare von Definitionen und Referenzen leichter ablesen können.

Übung 8.4:

Für welche Paare von Definitionen und Referenzen einer Variablen gibt es bei den Programmen aus Abbildung 7.1 ($z = x^{|y|}$) und Beispiel 7.3.1 (Textformatierer) einen Unterschied bei den Kriterien *alle Referenzen* und *alle DR-Wege* (vgl. Definition 8.2.3 und 8.2.6)?
Beachten Sie, daß die verschiedenen Wege beim Kriterium *alle DR-Wege* keine Zyklen enthalten dürfen bzw. nur genau aus einem einfachen Zyklus bestehen dürfen. Bei Abbildung 7.1 ist also ein Weg E, F, C, D, E erlaubt (einfacher Zyklus), nicht aber der Weg B, C, D, F, C, G.

Übung 8.5:
Ermitteln Sie die 3-DR-Interaktionen (s. Definition 8.3.1)

(a) für das Programm aus Abbildung 7.1 ($z = x^{|y|}$);

(b) für den Textformatierer aus Beispiel 7.3.1, allerdings nur für die 3-DR-Interaktionen, in denen zwei *verschiedene* Variablen vorkommen.

Geben Sie in beiden Fällen [(a) und (b)] jeweils einen Interaktionsweg (s. Definition 8.3.2) an, der zu der 3-DR-Interaktion paßt.

Übung 8.6:
Betrachten Sie das Programm aus Abbildung 7.1 und darin die Ausgabeanweisung „Write z" in Knoten G. Ermitteln Sie dazu die möglichen Definitionskontexte und verfolgen Sie diese weiter zurück, indem Sie für die definierenden Knoten wiederum die referenzierten Variablen und dazu die Definitionskontexte und Kontextwege ermitteln und verketten. Brechen Sie das Verfahren ab, wenn Zyklen in den Wegen entstehen. Orientieren Sie sich am Vorgehen bei Beispiel 8.4.3.

8.6 Verwendete Quellen und weiterführende Literatur

Das Überdeckungsmaß **alle Definitionen** wurde von Rapps und Weyuker definiert (s. [RaW 85]). Das Kriterium **alle DR-Interaktionen** wurde zuerst von Herman formuliert (s. [Her 76]). Rapps und Weyuker haben das Kriterium *alle DR-Interaktionen* abgewandelt, und zwar zu dem Kriterium **alle Referenzen** und zu den schwächeren Kriterien **alle E-/einige B-Referenzen** und **alle B-/einige E-Referenzen** (s. [RaW 85], S. 371). Das stärkere Kriterium **alle DR-Wege** wurde ebenfalls von Rapps und Weyuker vorgeschlagen (s. [RaW 85]).
Von Ntafos stammt das Kriterium **alle k-Tupel**, welches in abgewandelter Form von Clarke et al. formuliert wurde (s. [Cl& 89], S. 1321). In diesem Kapitel wurde die darin enthaltene Forderung nach bestimmten Schleifendurchläufen weggelassen und der reine Datenflußaspekt des Kriteriums als besonderes Kriterium **alle k-DR-Interaktionen** formuliert. Ursprünglich enthielt das Kriterium von Ntafos auch noch die Forderung nach Bedingungsüberdeckung (s. [Nta 84a], S. 251).
Die Begriffe **Definitionskontext** und **Kontextweg** wurden von Clarke et al. und Laski/Korel formuliert (vgl. [LaK 83], S. 349 ff., [Cl& 89], S. 1322). Im Unterschied zu Clarke et al. wurde hier im Buch — wie bei Laski und Korel — für einen Knoten k stets die ganze Menge *REF(k)* betrachtet. Das Kriterium **Kontextüberdeckung** stammt ebenfalls von Laski und Korel. Die Zusatzforderung **geordnete Kontextüberdeckung** haben Clarke et al. vorgeschlagen (s. [Cl& 89], S. 1322). Die mächtige, aber aufwendige Strategie **Definitionsbaumtesten** wurde von Laski vorgestellt (s. [Las 82]), s. auch [De& 87], S. 62 f.).

9 Ausdrucks-, anweisungs- und datenbezogenes Testen

9.1 Ausdrucks- und anweisungsbezogenes Testen

Das datenflußbezogene Testen in Kapitel 8 wurde damit motiviert, daß Fehler *in Anweisungen* aufgespürt werden sollen (s. Beispiel 8.1.1). Für diese Fehler gibt es zwei mögliche Ursachen:

1. Die Anweisung ist korrekt, aber die referenzierten Werte werden vorher falsch berechnet.

2. Die Anweisung ist falsch.

Die erste Ursache wurde in Kapitel 8 zum Ausgangspunkt der Anforderungen an die Testkriterien gemacht. In diesem Kapitel wird die zweite Ursache (falsche Anweisungen) betrachtet.

Um Berechnungsfehler wirklich aufzuspüren, reicht es nicht aus, einen Weg von einer Variablendefinition zu einer Variablenreferenz mit *irgendeinem* Wert für die Variable auszuführen. Vielmehr müssen die Anweisungen auf den Wegen mit solchen Werten getestet werden, bei denen mögliche Fehler *tatsächlich* bemerkt werden.

Wenn die Anweisungen grob modelliert werden, ist nur von Interesse, auf welche Variablen dabei *zugegriffen* wird und in welchen Variablen die Ergebnisse von Berechnungen *gespeichert* werden. Bei einer feineren Modellierung von Berechnungen und Entscheidungen werden die *Ausdrücke* und *Relationen* betrachtet, die in den Anweisungen vorkommen.
Es wird also folgendes unterschieden:

1. Datenzugriff

2. Datenspeicherung

3. arithmetischer Ausdruck

4. arithmetische Relation

5. Boolescher Ausdruck

Die möglichen Fehlerarten und die entsprechenden Anforderungen an die Testdaten, die diese Fehler aufdecken können, werden im folgenden behandelt. Als allgemeine Anforderung an die Testdaten gilt: *direkt* nach Ausführung der fehlerhaften Anweisung muß ein fehlerhafter Programmzustand vorliegen, und zwar für wenigstens eine Ausführung der fehlerhaften Anweisung. (Die stärkere Forderung, daß sich dieser Zustandsfehler auch bis zu einer Programmausgabe *fortpflanzt*, ist Grundlage der Mutationsanalyse [genaueres siehe Kap. 9.3]. Der hier gewählte Ansatz ist somit nur eine **schwache Mutationsanalyse**.)

1. **Datenzugriff**

Fehlerart: Zugriff auf eine falsche Variable

Testdaten: Alle Variablen im Programm müssen vor dem Zugriff verschiedene Werte haben. Dieses Kriterium heißt **Datenzugriffskriterium**.

Beispiel 9.1.1
Für das Programm aus Abbildung 7.2 soll $N \neq 0$ sein, damit in Anweisung 1 die Variablen LOW und HIGH verschiedene Werte erhalten, die bei fehlerhaftem Zugriff in Anweisung 2 oder 3 unterscheidbar sind.
Damit sich in Anweisung 2 auch MID davon unterscheidet, muß die Differenz von LOW und HIGH mindestens 2 sein, da z. B. LOW = 1, HIGH = 2 die Gleichheit $MID = \lfloor \frac{1+2}{2} \rfloor = 1 = LOW$ *ergibt. Dies ist für Zugriffe auf MID in Anweisung 5,6,8,9 oder 10 wichtig. Entsprechendes gilt für NPOS, A[MID] und Zugriffe in den anderen Anweisungen.*

2. **Datenspeicherung**

Fehlerart: Speicherung eines Wertes in einer falschen Variablen

Testdaten: Wird einer Variablen ein Wert zugewiesen, muß er anders als der bisherige Wert sein. Dieses Kriterium heißt **Datenspeicherungskriterium**.

Beispiel 9.1.2
Wählt man für das Programm aus Abbildung 7.2 die Eingabe N = 1 und Werte von F und A mit $F > A[1]$, so erhält MID in Anweisung 2 zweimal nacheinander den Wert 1 zugewiesen[1]. *Müßte der Wert 1 richtigerweise beim zweiten Mal einer anderen Variablen zugewiesen werden, so fällt dies bei obigen Testdaten (bei der Betrachtung von MID) nicht auf: MID hätte den richtigen Wert 1 trotz falscher Zuweisung behalten.*

[1]Beim ersten Mal $\lfloor \frac{1+1}{2} \rfloor = 1$, beim zweiten Mal (wg. $F > A[1]$ in Anweisung 8) den Wert $\lfloor \frac{2+1}{2} \rfloor = 1$.

3. **Arithmetischer Ausdruck**
   Die hier betrachteten arithmetischen Ausdrücke bestehen aus Variablen oder Konstanten als Operanden und den Operatoren $+, -, *, /$ und $**$ (Exponentiation). Enthält ein Ausdruck nicht die Division „/“, so ist er ein **Polynom**, falls nur eine Variable vorkommt, sonst ein **Multinom**.

   Fehlerarten:

   (a) einfache additive oder multiplikative Fehler,

   (b) ein Fehler in einem Polynom oder Multinom, der die Variablenmenge nicht ändert und den höchsten Exponenten im Ausdruck nicht erniedrigt.

   Testdaten:

   (a) Der betroffene (Teil-)Ausdruck muß einen Wert ungleich 0 erhalten, damit multiplikative Fehler gefunden werden. (Additive Fehler wirken sich immer aus). Dieses Kriterium heißt das **additive/multiplikative Fehler-Kriterium**.

   (b) Für ein Polynom vom Grad n sind $n+1$ unabhängige Testdaten ausreichend. Für ein Multinom mit höchstem Exponenten n ist eine Kaskadenmenge von k-Tupeln vom Grad $n+1$ ausreichend, wobei k die Anzahl der Variablen ist (genaueres s. [How 87] und [How 78d]). Diese Kriterien heißen **Polynom-** bzw. **Multinom-Kriterium**.

Beispiel 9.1.3 (zu einfachen Fehlern und Polynomen)
Bei dem Programm aus Abbildung 7.2 enthält die rechte Seite der Zuweisung in Zeile 9 den Ausdruck MID + 1.

(a) *Sei „$k*MID+1$“ der korrekte Ausdruck (für $k \neq 1$). Dann muß mit $MID \neq \underline{0}$ getestet werden.*
Sei „$k*(MID+1)$“ der korrekte Ausdruck (für $k \neq 1$). Dann muß mit $MID+1 \neq 0$, d. h. $MID \neq \underline{-1}$ getestet werden. (Falls „MID + c“ der korrekte Ausdruck ist (für $c \neq 1$), so ist jedes Testdatum für MID fehleraufdeckend.)

(b) *Falls der Ausdruck „$k*MID+c$“ korrekt ist (für $k \neq 1$), so genügen zwei Testdaten für dieses Polynom vom Grade 1, da $k*MID+c = MID+1$ nur für $MID = \frac{1-c}{k-1} =: s$ gilt. Ein Testdatum ungleich s deckt also die Abweichung auf, bei zwei verschiedenen Testdaten ist garantiert eines der Testdaten ungleich s.*

Falls beispielsweise der Ausdruck „MID^2+1“ korrekt ist, muß man (bei einem Polynom zweiten Grades) mit drei Testdaten testen, da die beiden Testdaten MID = 0 und MID = 1 den Unterschied zum Ausdruck „MID + 1“ nicht feststellen würden. Da der korrekte Exponent aber (beim implementationsorientierten Testen) nicht bekannt ist, läßt sich die hinreichende Anzahl der Testdaten nicht sicher ermitteln.

Die erforderlichen Testdaten bei dem jeweils angenommenen korrekten Ausdruck lassen sich tabellarisch zusammenstellen (s. Tabelle 9.1).

Nr.	korrekter Ausdruck	Bedingung für Testdaten
1	$k * MID + 1\ (k \neq 1)$	$MID \neq 0$
2	$k * (MID + 1)\ (k \neq 1)$	$MID \neq -1$
3	$MID + c\ (c \neq 1)$	beliebig
4	$k * MID + c\ (k \neq 1)$	$MID \neq \frac{1-c}{k-1}$
5	$MID^2 + 1$	$MID \neq 0$ und $MID \neq 1$

Tab. 9.1 Erforderliche Testdaten für angenommene korrekte Ausdrücke

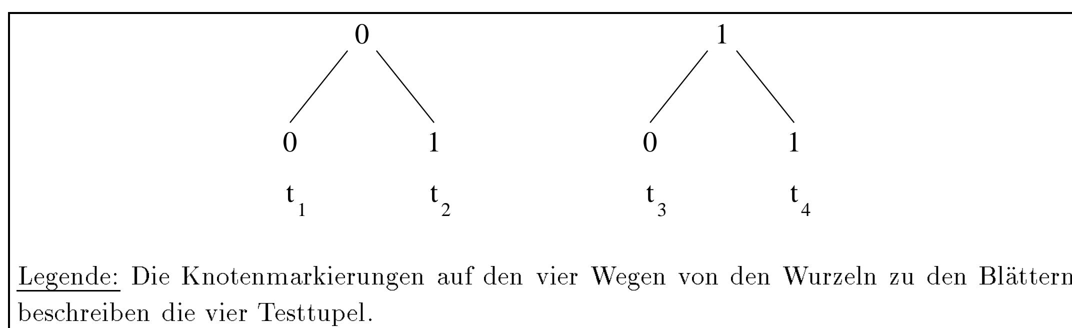
Im Fall 3 reicht also *ein* Testdatum, in den Fällen 1, 2 und 4 reichen *zwei* beliebige (verschiedene) Testdaten aus. Da dies alle Fehlerfälle für ein Polynom vom Grade 1 sind, reichen also dafür *zwei* beliebige (verschiedene) Testdaten aus. Im Falle 5 handelt es sich um ein Polynom vom Grade 2. *Drei* beliebige (verschiedene) Testdaten reichen dafür aus.

Hier zeigt sich das Problem bei dieser Methode: wenn man nicht weiß, ob der Grad des Polynoms durch den Fehler erniedrigt wurde, bzw. wenn man den Grad g des *korrekten* Polynoms nicht kennt, kann man auch nicht die Anzahl $g+1$ der ausreichenden Testdaten bestimmen.

Beispiel 9.1.4 (zu Multinomen)
Bei dem Programm aus Abbildung 7.2 enthält die rechte Seite der Zuweisung in Anweisung 2 den Ausdruck $\frac{LOW+HIGH}{2}$, wobei hier aus Vereinfachungsgründen die reellwertige Division angenommen wird[2].
Der Ausdruck hat also die Form $A = a * LOW + b * HIGH$ mit den Konstanten $a = b = 0,5 = \frac{1}{2}$, d. h. er ist ein (lineares) Multinom in den beiden Variablen LOW und HIGH, wobei der höchste Exponent den Wert $n = 1$ hat.

Um dieses Multinom von einem anderen Multinom B mit höchstem Exponenten 1 in LOW und HIGH zu unterscheiden, sind die folgenden vier Testdaten $t_1 = (0,0), t_2 = (0,1), t_3 = (1,0)$ und $t_4 = (1,1)$ der Art $t_i =$ (LOW, HIGH) ausreichend. Sie lassen sich nämlich durch die folgenden beiden Bäume beschreiben (siehe Abbildung 9.1) und sind damit (definitionsgemäß[3]*) eine Kaskadenmenge von k-Tupeln (mit $k = 2$) vom Grad $n + 1 = 2$ (vgl. obige Aussage 3(b) zu Testdaten bei arithmetischen Ausdrücken).*

[2] Der Ausdruck $\frac{LOW+HIGH}{2}$ mit ganzzahliger Division ist kein Multinom, da auf den Wert der reellwertigen Division noch die Abrundungsfunktion angewandt werden muß, die verschiedene Werte auf einen Wert abbildet, z. B. (ganzzahlig) $\frac{7}{3} = \frac{8}{3} = 2$. Daher reichen Testdaten, die einen Unterschied bei der reellwertigen Division nachweisen, nicht unbedingt aus.
[3] nach [How 78d] und [How 87]



Legende: Die Knotenmarkierungen auf den vier Wegen von den Wurzeln zu den Blättern beschreiben die vier Testtupel.

Abb. 9.1: Kaskadenmenge vom Grad 2

Diese Tests sind ausreichend zur Unterscheidung von Multinom A und Multinom B, wobei B die folgende allgemeine Form habe:
$B = c * LOW * HIGH + d * LOW + e * HIGH + f$ *mit Konstanten* c, d, e, f.
Angenommen, die Tests ergeben für A und B gleiche Ergebnisse; dann gilt:
$A(0,0) = 0 = B(0,0) = f$, *was* $f = 0$ *impliziert;*
$A(0,1) = b = B(0,1) = e + f$, *was (wegen* $f = 0$*)* $e = b$ *impliziert;*
$A(1,0) = a = B(1,0) = d + f$, *was (wegen* $f = 0$*)* $d = a$ *impliziert;*
$A(1,1) = a + b = B(1,1) = c + d + e + f$, *was (wegen* $f = 0, a = d, b = e$*)* $c = 0$ *impliziert.*

Also hat B auch die Form $B = a * LOW + b * HIGH$, *d. h. B ist identisch mit Multinom A.*
Durch logische Umkehrung gilt: ist B nicht identisch mit A, so erhält man in mindestens einem der Fälle $t_1 = (0,0), t_2 = (0,1), t_3 = (1,0)$ *oder* $t_4 = (1,1)$ *unterschiedliche Testergebnisse.* **q.e.d.**

4. **Arithmetische Relation**
   Seien A und B arithmetische Ausdrücke und r eines der Relationssymbole $<$, $=$, $>$, $\leq$, $\geq$ oder $\neq$.

   Fehlerarten:

   (a) Die Relation der Form „$A\ r\ B$“ enthält ein falsches Relationssymbol r.
   (b) „$A\ r\ B$“ ist falsch, die Relation „$(A + k)\ r\ B$“ müßte realisiert werden (mit einer Konstanten $k \neq 0$)[4].

   Testdaten:
   Es sind drei Testdaten nötig, bei denen die Differenz $A - B$ den größten negativen Wert (knapp unter 0), den Wert 0 und den kleinsten positiven Wert annimmt. Dieses Kriterium heißt das **arithmetische Relations-Kriterium**.

[4]Das ist äquivalent zum Fall „$A\ r\ (B - k)$“, d. h. einem Fehler auf der rechten Seite. (Man beachte, daß k auch negativ sein darf.)

BEISPIEL 9.1.5
Der Ausdruck in Anweisung 8 des Programms aus Abb. 7.2 ist „$F > A[MID]$“. Die erforderlichen Testdaten bei dem jeweils angenommenen korrekten Ausdruck sind in Tabelle 9.2 dargestellt.

Nr.	*korrekter Ausdruck*	*Bedingung für Testdaten*
1	„$F = A[MID]$“	$F = A[MID]$
2	„$F \geq A[MID]$“	$F = A[MID]$
3	„$F \leq A[MID]$“	$F = A[MID]$
4	„$F < A[MID]$“	$F = A[MID] + d, d > 0$
5	„$F \neq A[MID]$“	$F = A[MID] - d, d > 0$
6	„$F > A[MID] + k$“, $k \neq 0$	$F = A[MID] + d, 0 < \|d\| < \|k\|$ mit $d > 0$ *g. d. w.* $k > 0$

Legende: Die mittlere Spalte enthält die im Programm vorkommenden syntaktischen Ausdrücke. Die rechte Spalte beschreibt dagegen Relationen zwischen den Werten der angegebenen Variablen.

Tab. 9.2 *Erforderliche Testdaten für den Ausdruck „$F > A[MID]$“*

5. **Boolescher Ausdruck**
   Boolesche Ausdrücke können auf zwei Arten verwendet werden.

   a) Berechnung eines Entscheidungsausgangs im Kontrollfluß
   b) Berechnung des Wertes einer Booleschen Variablen

zu a: Wenn ein Boolescher Ausdruck direkt oder indirekt zur Berechnung des Ausgangs einer Entscheidung im Kontrollfluß dient, dann können die Testkriterien als Verfeinerung der Kriterien für die *Zweigüberdeckung* im Kontrollflußgraphen aufgefaßt werden (vgl. Def. 7.2.4 auf S. 197). Dies ist naheliegend, wenn die Übersetzung dieser Ausdrücke betrachtet wird.

BEISPIEL 9.1.6
*Die Verzweigung „**if** C **or** D **then** B_1 **else** B_2“ aus Abbildung 9.2 a) (mit Booleschen Ausdrücken C und D und Berechnungen B1 und B2) wird üblicherweise in eine Kontrollstruktur wie in Abbildung 9.2 b) übersetzt, die zwei Verzweigungen enthält. Man könnte also die Zweigüberdeckung für die Kontrollstruktur in Abb. 9.2 b) fordern, d. h. statt zwei sind vier Zweige zu überdecken. Von den beiden Fällen für „C **or** D“ wird dabei der Fall „C **or** $D = true$“ differenzierter betrachtet.*

Die obige Interpretation mit der Verfeinerung bzw. Übersetzung des Kontrollflusses kann im folgenden als Begründung für die Anforderung an Testdaten verwendet werden.

zu b: Boolesche Ausdrücke können zur Berechnung der Werte von Booleschen Variablen dienen, die ein eigenständiges Ergebnis eines Programms sind.

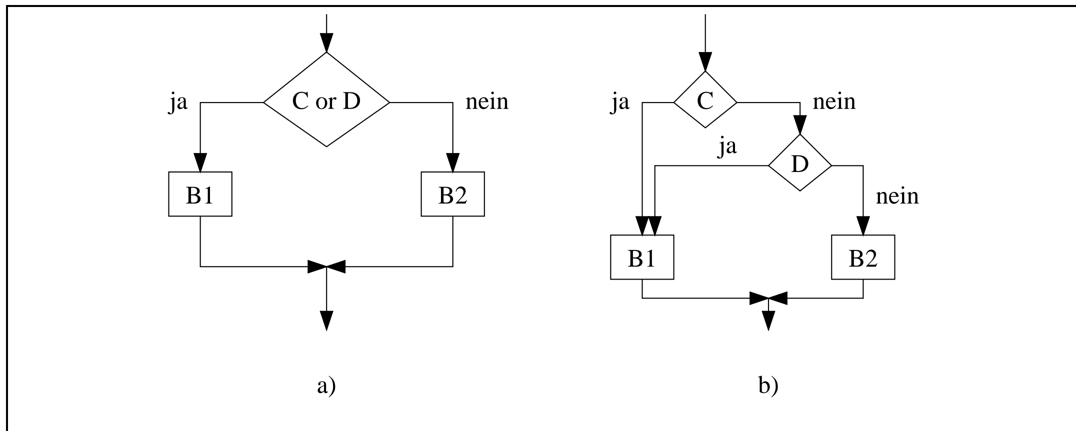


Abb. 9.2: Kontrollfluß bei dem Booleschen Ausdruck „C **or** D"

Im folgenden Kapitel 9.2 werden beide Arten von Booleschen Ausdrücken einheitlich behandelt. Allerdings werden die Erläuterungen und Beispiele immer nur auf den Fall „Boolescher Ausdruck in Entscheidungen im Kontrollfluß" bezogen. Damit werden die Verbindungen zu den Kontrollflußkriterien aus Kapitel 7 aufgezeigt.

6. **Ausdrücke (allgemein)**

Fehlerarten:

(a) überflüssige Teilausdrücke
(b) vertauschte (Teil-)Ausdrücke

Testdaten:

(a) Für jeden Ausdruck und jeden seiner Teilausdrücke muß es ein Testdatum geben, das diesen Ausdruck ausführt, so daß bei der Ausführung Ausdruck und Teilausdruck verschiedene Werte annehmen (damit der überflüssige Rest erkannt wird). Dieses Kriterium heißt **kürzere Ausdrücke**.
(b) Für jeden Ausdruck im Programm muß es ein Testdatum geben, das diesen Ausdruck ausführt und bei dessen Ausführung alle (Teil-)Ausdrücke im Programm verschiedene Werte haben (damit eine Vertauschung auffällt). Dieses Kriterium heißt **alle kürzeren Ausdrücke**.

BEISPIEL 9.1.7
Bei einem Test des Ausdrucks $x * (x - 2) + (x - 1)$ mit den Testdaten 0 und 2 ergibt sich für den Teilausdruck $x * (x - 2)$ stets der Wert 0. Daher wird der gesamte Ausdruck nicht von dem kürzeren Ausdruck $x - 1$ unterschieden. Entsprechendes gilt für den Teilausdruck $(x-1)$ und Testdatum $x = 1$. Also muß mit einem Testdatum ungleich 0, 1 und 2 getestet werden, um das Kriterium „kürzere Ausdrücke" zu erfüllen.

Das Kriterium **mehrfache Werte für Ausdrücke** fordert für komplette Ausdrücke, daß sie jeweils zwei verschiedene Werte annehmen. Damit verallgemeinert das Kriterium die Forderung für multiplikative Fehler bei Polynomen (vgl. die Diskussion zu Tabelle 9.1).

Alle aufgestellten Kriterien müssen sinngemäß angepaßt werden, wenn die Ausdrücke und Anweisungen nicht Zahlen, sondern komplexere Datenstrukturen enthalten.

9.2 Test Boolescher Ausdrücke

In einem ersten Ansatz (der später verfeinert wird) werden Boolesche Ausdrücke als Verknüpfung von atomaren Prädikattermen bzw. atomaren Bedingungen mit logischen Operatoren modelliert. Die Struktur der atomaren Bedingungen wird dabei nicht weiter analysiert. Es interessiert nur, daß sie die Werte *true* und *false* haben.

BEISPIEL 9.2.1
In einem Programm sei folgende Entscheidung enthalten:

$$\textbf{if } A > 1 \textbf{ and } (B = 2 \textbf{ or } C > 1) \textbf{ and } D \textbf{ then } \dots \textbf{ else } \dots$$

Dann ist

$$A > 1 \textbf{ and } (B = 2 \textbf{ or } C > 1) \textbf{ and } D$$

das entsprechende Entscheidungsprädikat mit den folgenden vier atomaren Prädikattermen:

$$A > 1, B = 2, C > 1, D \qquad .$$

Atomare Prädikatterme zeichnen sich dadurch aus, daß sie keine logischen Operatoren wie *and*, *or*, *not* enthalten, sondern höchstens Relationssymbole, wie z. B. „>" und „=". Man beachte, daß im obigen Beispiel „D" eine boolesche Variable sein muß, damit dieser Term die logischen Wahrheitswerte annehmen kann.

Denkt man an die Verwendung von Booleschen Ausdrücken in Entscheidungen im Kontrollfluß und an die Übersetzung der zusammengesetzten Entscheidungsprädikate wie in Abbildung 9.2b), so kann man die Forderung aufstellen, alle Zweige im übersetzten Kontrollflußgraphen auszuführen. Ein erster naheliegender Ansatz dazu ist das folgende Kriterium.

DEFINITION 9.2.1
Eine Testdatenmenge T erfüllt die **C_2-Überdeckung** *g. d. w. es für jede Verzweigung im Programm mit zwei Ausgängen*[5] *und für jeden atomaren Prädikatterm p des zur Verzweigung gehörenden Booleschen Ausdrucks zu jedem Wahrheitswert von p ein Testdatum t aus T gibt, bei dessen Ausführung p diesen Wahrheitswert annimmt.*

[5] Verzweigungen mit mehr als zwei Ausgängen (z. B. *case*-Anweisungen) können stets in eine äquivalente Kaskade von Verzweigungen mit zwei Ausgängen umgewandelt werden.

Die C_2-Überdeckung heißt auch **(atomare) Bedingungsüberdeckung**, da die atomaren Prädikatterme auch atomare Bedingungen genannt werden.

Für eine atomare Bedingungsüberdeckung ist leider *nicht* garantiert, daß der Test Stichproben für beide logischen Werte der kompletten Booleschen Ausdrücke enthält. Das bedeutet, daß eine atomare Bedingungsüberdeckung unter Kontrollflußkriterien nicht ausreichend ist, da sie keine Zweigüberdeckung garantiert.

Beispiel 9.2.2
Für das Programm aus Abbildung 9.3 erfüllen beispielsweise folgende Testdaten die atomare Bedingungsüberdeckung, aber nicht die Zweigüberdeckung:

t_1: *Testdatum:* $A = 2, B = 1, C = 4$; *Solldatum:* $C = 5$,
damit $A > 1, B \neq 0, A = 2$ *und* $C > 1$ *erfüllt ist.*

t_2: *Testdatum:* $A = C = 1, B = 0$; *Solldatum:* $C = 1$,
damit $A \leq 1, B = 0, A \neq 2$ *und* $C \leq 1$ *erfüllt ist.*

Im Kontrollflußgraphen von Abbildung 9.3 führt Testdatum t_1 *den Weg* abe *und Testdatum* t_2 *den Weg* abd *aus. Entscheidungskante* c *wird also nicht ausgeführt.*

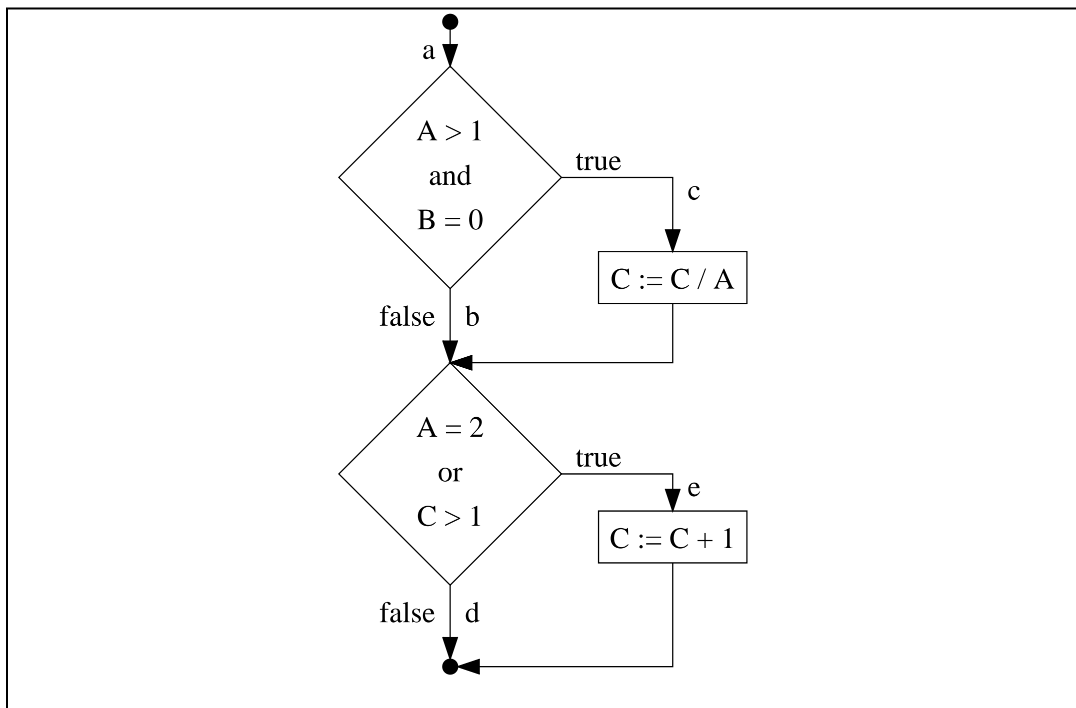


Abb. 9.3: Programm mit zusammengesetzten Entscheidungsprädikaten

Um die oben genannte Schwäche der atomaren Bedingungsüberdeckung auszumerzen, muß also wenigstens folgendes verlangt werden:

Definition 9.2.2
Eine Testdatenmenge erfüllt die **Zweig-/Bedingungsüberdeckung** *g. d. w. sie die C_2-Überdeckung (die atomare Bedingungsüberdeckung) und die C_1-Überdeckung (die Zweigüberdeckung) erfüllt.*

Für einen Booleschen Ausdruck B, der als Entscheidungsprädikat im Kontrollfluß fungiert, bedeutet die C_1-Überdeckung, daß der komplette Boolesche Ausdruck sowohl mit dem Wert $B = true$ als auch mit dem Wert $B = false$ auszuführen ist.

Beispiel 9.2.3
Für eine Zweig-/Bedingungsüberdeckung des Programms aus Abbildung 9.3 können folgende Testdaten t_1 und t_2 gewählt werden:

- t_1: *Testdatum: $A = 2, B = 0, C = 4$, Solldatum: $C = 3$, damit $A > 1$ und $B = 0$ und $A = 2$ und (nach $C := C/A$) $C > 1$ gilt und der Weg ace ausgeführt wird;*
- t_2: *Testdatum: $A = B = C = 1$, Solldatum: $C = 1$, damit der Weg abd wegen $A \leq 1$ und $B \neq 0$ und $A \neq 2$ und $C \leq 1$ ausgeführt wird.*

Es werden alle Wahrheitswerte der atomaren Bedingungen $A > 1, B = 0, A = 2, C > 1$ angenommen und alle Entscheidungskanten b, c, d, e ausgeführt. Es werden aber nicht alle Zweige des (wie bei Abbildung 9.2b) verfeinerten Kontrollflußgraphen ausgeführt: es fehlen die Fälle „$A > 1$ und $B \neq 0$“ bei der ersten Entscheidung und „$A \neq 2$ und $C > 1$“ bei der zweiten Entscheidung. Daher würde ein Fehler in der zweiten Entscheidung (z. B. „$A = 2$ or $C > 2$“ ist richtig und „$A = 2$ or $C > \underline{1}$“ ist falsch) nicht auffallen. Bei Testdatum t_1 ist nämlich die Abfrage „$C > 1$“ durch $A = 2$ maskiert und bei Testdatum t_2 ist wegen $C = 1$ sowohl „$C > 1$“ als auch „$C > 2$“ nicht erfüllt.

Im Beispiel 9.2.3 muß also der Fall $C > 1$ zusammen mit $A \neq 2$ ausgeführt werden, damit ein fehlerhafter logischer Wert von „$C > 1$“ auffällt. Um genügend Kombinationen von Wahrheitswerten von atomaren Bedingungen zu testen, ist demnach folgendes zu fordern:

Definition 9.2.3
Eine Testdatenmenge erfüllt die **minimale Mehrfachbedingungsüberdeckung** *g. d. w. für jede Verzweigung im Programm mit zwei Ausgängen und für den zur Verzweigung gehörenden Booleschen Ausdruck folgende Kombinationen der Wahrheitswerte der atomaren Prädikatterme, die mit ein- oder mehrstelligen logischen Operatoren verknüpft sind, ausgeführt werden: jede[6] mögliche Kombination von Wahrheitswerten, bei denen die Änderung des Wahrheitswerts eines Terms den Wahrheitswert der logischen Verknüpfung ändern kann.*
Diese Überdeckung wird auch **C_2(mM)-Überdeckung** *genannt.*

[6] Zur weiteren Einschränkung der Eingabekombinationen siehe Übung 9.4b).

Die $C_2(mM)$-Überdeckung entspricht der Idee der Testdatenauswahl beim Ursache-Wirkungs-Graphen (vgl. Abschnitt 4.2.3) und den entsprechenden Techniken beim Testen von Haftfehlern in der Hardware.

Beispiel 9.2.4 (and)

A	B	A and B
F	F	F
F	T	F
T	F	F
T	T	T

Wenn sich bei der Kombination $A = B = F$ ($false$) nur einer der beiden Werte fälschlicherweise zu T ($true$) ändert, ergibt das keine Änderung des Wertes des Ausdrucks „A and B“. Dies wird Maskierung des Fehlers genannt und man sagt, daß $A = B = false$ nicht sensitiv auf Wahrheitswertänderung reagiert. Daher sind nur die anderen drei Kombinationen zu testen (falls sie möglich sind[7]).

Beispiel 9.2.5 (equivalence)

A	B	A equivalence B
F	F	T
F	T	F
T	F	F
T	T	T

Jede Kombination ändert bei Änderung des Wahrheitswertes von A oder B den Wahrheitswert von „A equivalence B“. Daher sind alle vier Kombinationen zu testen (falls sie möglich sind).

[7] Dies ist dann der Fall, wenn die Wahrheitswerte der Terme unabhängig voneinander gebildet werden können. Bei „$A > 2$ *and* $A > 7$“ ist dies z. B. nicht der Fall, denn $(A > 7) = true$ impliziert $(A > 2) = true$. Das Prädikat läßt sich aber in diesem Fall zu „$A > 7$“ vereinfachen. Bei „$A < 2$ *or* $A > 7$“ ist die Kombination *true-true* ausgeschlossen, die restlichen drei Kombinationen (mit den Wertebereichen $A < 2, 2 \leq A \leq 7, 7 < A$) lassen sich aber nicht einfacher darstellen. Beim Ausdruck „$A > 2$ *or* $A < 7$“ ist die Kombination *false-false* nicht möglich, daher ist der Ausdruck stets (konstant) *true*.

Beispiel 9.2.6 (or)

1. *Teste eine Kombination, bei der alle Terme den Wert false haben.*
2. *Teste jede mögliche Kombination, bei der genau ein Term den Wert true hat (und alle anderen Terme den Wert false).*

Beispiel 9.2.7 (nand)

1. *Teste eine Kombination, bei der alle Terme den Wert true haben.*
2. *Teste jede mögliche Kombination, bei der genau ein Term den Wert false hat (und alle anderen Terme den Wert true).*

Beispiel 9.2.8 (Testdaten für das Programm von Abbildung 9.3)
Folgende drei Kombinationen sind für die erste Entscheidung zu testen:
$(A > 1, B = 0)$, $(A > 1, B \neq 0)$, $(A \leq 1, B = 0)$
[aber nicht $(A \leq 1, B \neq 0)$*].*
Für die zweite Entscheidung ist an dieser Stelle folgendes zu testen (wobei der Anfangswert für C evtl. anders sein muß, falls er in Zweig c verändert wird):
$(A \neq 2, C \leq 1)$, $(A \neq 2, C > 1)$, $(A = 2, C \leq 1)$
[aber nicht $(A = 2, C > 1)$*].*
Die Kombinationen werden durch folgende Testdaten ausgeführt:

t_1: *Testdatum:* $A = 3, B = 0, C = 3$; *Solldatum:* $C = 1$;
Weg acd und die Kombinationen $(A > 1, B = 0)$ *und* $(A \neq 2, C \leq 1)$ *werden ausgeführt. Der Anfangswert* $C = 3$ *wird dabei in Zweig c zu* $C = 1$ *verändert.*

t_2: *Testdatum:* $A = 2, B = C = 1$; *Solldatum:* $C = 2$;
Weg abe und die Kombinationen $(A > 1, B \neq 0)$ *und* $(A = 2, C \leq 1)$ *werden ausgeführt.*

t_3: *Testdatum:* $A = 1, B = 0, C = 2$; *Solldatum:* $C = 3$;
Weg abe und die Kombinationen $(A \leq 1, B = 0)$ *und* $(A \neq 2, C > 1)$ *werden ausgeführt.*

Die Kombination $(A = 2, C > 1)$ mußte in Beispiel 9.2.8 nicht getestet werden, da stillschweigend angenommen wurde, daß die Verknüpfung mit *or* korrekt ist und nicht etwa *exor* erforderlich ist. Wenn diese Annahme nicht gerechtfertigt ist, muß eine noch stärkere Anforderung an die Testdaten gestellt werden.

Definition 9.2.4
Eine Testdatenmenge erfüllt die **Mehrfachbedingungsüberdeckung** *g. d. w. für jede Verzweigung im Programm mit zwei Ausgängen und für den zur Verzweigung gehörenden Booleschen Ausdruck jede mögliche Kombination der Wahrheitswerte der atomaren Prädikatterme ausgeführt wird.*

Die Mehrfachbedingungsüberdeckung wird auch **C_2(M)-Überdeckung** *genannt.*

Obwohl die Kriterien $C_2(mM)$- und $C_2(M)$-Überdeckung stärkere Anforderungen an eine Testdatenmenge stellen als die Kriterien C_2- und C_1-Überdeckung, können immer noch Fehler in den Bedingungen auftreten.

Beispiel 9.2.9
Der Term „$A = 2$“ in der zweiten Entscheidung des Programms von Abbildung 9.3 sei ersetzt worden durch den Term „$A = 2 + (C - 2) * (C - 1)$“. Für die Testdaten t_1, t_2, t_3 zur $C_2(mM)$-Überdeckung aus Beispiel 9.2.8 ergibt sich stets $2 + (C - 2) * (C - 1) = 2$, da direkt vor der Auswertung des fraglichen Terms $C = 2$ oder $C = 1$ gilt.

Der Fehler wird in Beispiel 9.2.9 nicht bemerkt, da die Struktur der atomaren Prädikatterme (gemäß der anfangs gemachten Vereinfachung) nicht beachtet und daher nicht hinreichend getestet wird. Es handelt sich dabei um einen Bereichsfehler durch einen falschen Eingabebereich (s. Definition 7.2.1 auf Seite 195).

Man kann den Eingabebereich eines Kontrollflußweges, bei dem n Variablen eine Rolle spielen, geometrisch interpretieren, und zwar als Teilmenge im n-dimensionalen Raum. Bereichsfehler bewirken dabei z. B. eine Verschiebung der Oberfläche (der Grenzen) dieser Teilmenge.
Eine Methode zum Feststellen solcher Grenzverschiebungen heißt **Bereichsüberdeckung** bzw. **$\mathbf{C_2}$(B)-Überdeckung**:

Wird der Bereich durch *eine* logische *and*-Verknüpfung von dimensionserhaltenden[8] Relationen ($<, \leq, >, \geq$) beschrieben und haben die Eingabebereiche lineare Grenzen und eine Dimension von 2 (d. h. die Anzahl der vorkommenden Variablen ist 2), sind sie folgendermaßen zu testen: Für jede Grenze sind je zwei Testdaten auf der Grenze und je zwei knapp daneben anzugeben; dabei muß letzteres Testdatenpaar[9] außerhalb des Bereichs liegen, falls die Grenze zum Bereich gehört (bei „$\leq$“- und „$\geq$“-Relationen), sonst muß es innerhalb des Bereichs liegen. Bei aneinanderstoßenden Grenzen kann mindestens ein Testdatum für beide Grenzen genutzt werden. Die Testdaten sollen jeweils am Ende der Grenzen liegen.

Beispiel 9.2.10
Für die Bedingung „$A > 1$ and $C \leq 1$“ stellt der schraffierte Bereich in Abbildung 9.4 den Eingabebereich für Werte des Typs Real dar.
Für die waagrechte Grenze „$C \leq 1$“ (aber $A > 1$) sind die Testdaten t_1, t_2 auf der Grenze und t_3, t_4 außerhalb des Bereichs zu wählen. Anstelle von t_3 und t_4 kann auch t_{34} gewählt werden. Das Testdatum t_1 wird auch für die Grenze „$A > 1$“ (aber $C \leq 1$) genutzt. Bei einer Verschiebung (auch Drehung) der waagrechten Grenze „$C \leq 1$“ wechselt mindestens einer der Testpunkte t_1, t_2, t_{34} (bzw. t_3, t_4) in den

[8] Die Relationen „=“ und „$\neq$“ beschreiben z. B. bei zwei Variablen keine Fläche, sondern eine Gerade (Dimension 1 statt 2). Daher sind andere bzw. weitere Testdaten zu wählen (s. Übung 9.5).

[9] Statt dieses Paars reicht auch *ein* Testdatum aus, das in der Mitte von diesen beiden liegt, wenn beliebig kleine reelle Werte benutzt werden können.

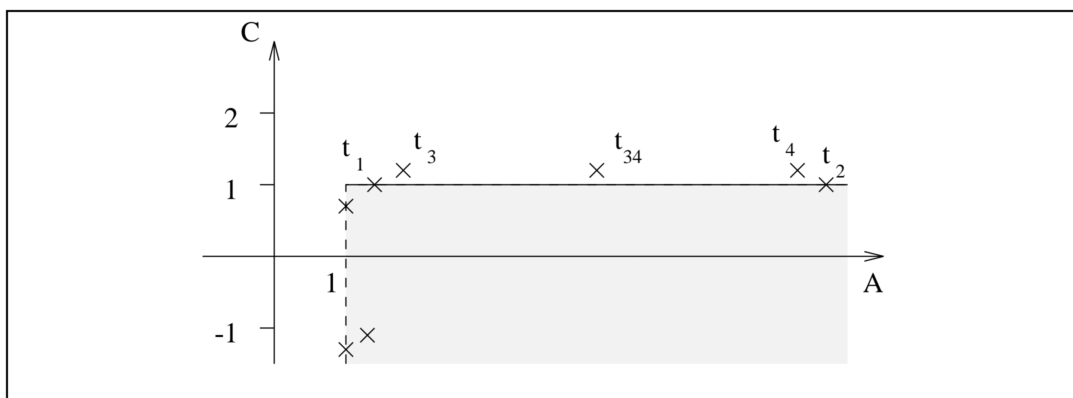


Abb. 9.4: Geometrisch dargestellter Eingabebereich „$A > 1$ *and* $C \leq 1$“

(bzw. aus dem) schraffierten Bereich (außer bei minimalen Fehlern, die geringer sind als der Abstand von t_{34} [bzw. t_3 oder t_4] von der Grenze).

Bei Verknüpfungen mit *or* können sich die Eingabebereiche für die einzelnen Terme überlappen. Damit entfallen gewisse Grenzen. Dies muß genau bestimmt werden, da Testdaten an den inneren Grenzen keine Fehler aufdecken können.

Beispiel 9.2.11

Für den Test von „$(A > 1$ and $C \leq 1)$ or $A > 2$“ müssen die beiden Testdaten t_2 und t_4 (bzw. t_{34}) aus Abbildung 9.4 (mit $C = 1$ bzw. $C = 1 + \delta$ und großem Wert von A) entfallen. Dafür braucht man vier (bzw. drei) neue Testdaten u_1 bis u_4 (bzw. u_{34} statt u_3 und u_4), welche die Fälle für A ($A = 2$ und $A = 2 + \delta$) mit den Fällen für C ($C = 1 + \delta$ bzw. großer Wert von C) kombinieren. Wegen der or-Verbindung des Terms aus Abb. 9.4 mit „$A > 2$“ dürfen keine Testdaten mit $C \leq 1$ und $A = 2 + \delta$ gewählt werden, da diese Fälle maskiert werden (s. Abbildung 9.5).

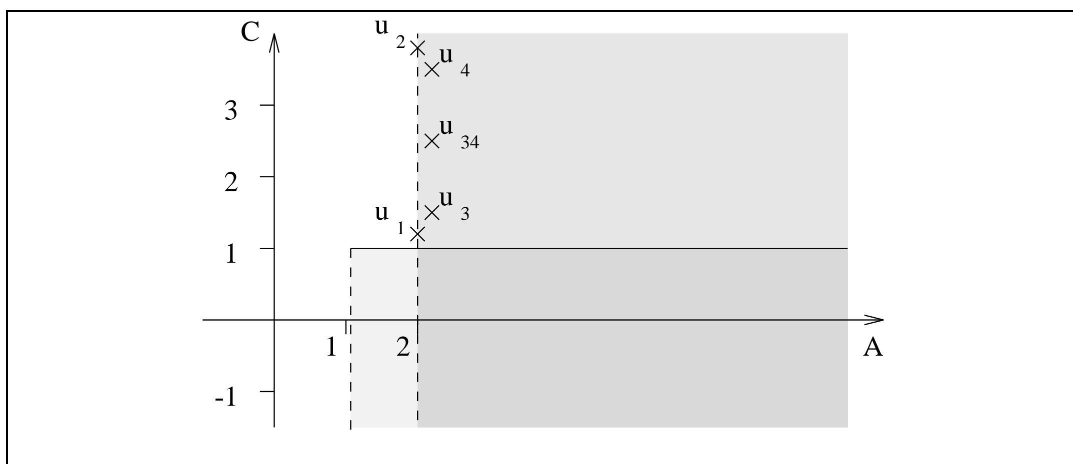


Abb. 9.5: Eingabebereich „$(A > 1$ *and* $C \leq 1)$ *or* $A > 2$“

Bei mehr als zwei (etwa v) Variablen spannen die linearen Grenzen eine Hyperebene im v-dimensionalen Raum auf. Pro Hyperebene sind dann v „linear unabhängige" Testpunkte auf der Ebene[10] und einer knapp außerhalb der Ebene erforderlich.

Bei nichtlinearen Grenzen muß je ein Testdatenpaar an jedes lokale Minimum und Maximum und an die Grenzen des Bereichs angelegt werden.

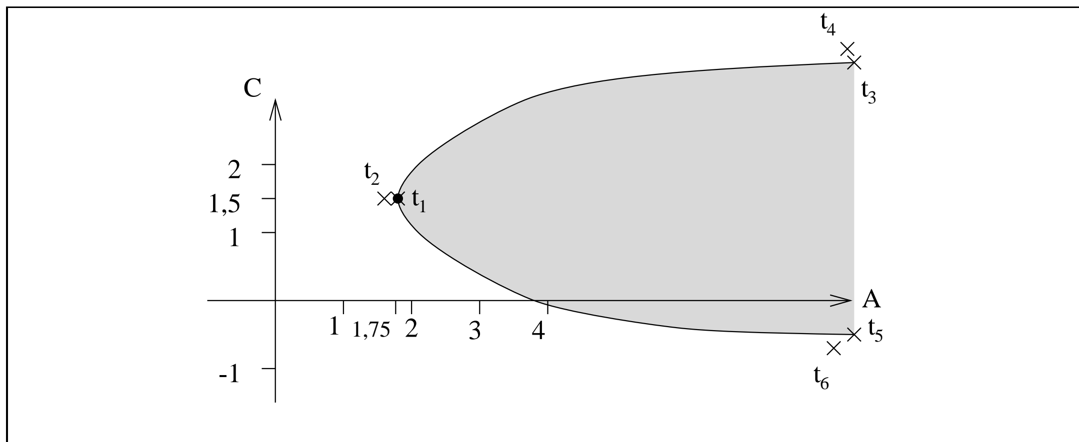


Abb. 9.6: Eingabebereich mit nichtlinearer Grenze „$A = (C - 1,5)^2 + 1,75$"

Beispiel 9.2.12

Für den Test von „$A \geq 2 + (C - 2) * (C - 1)$" bzw. das äquivalente Prädikat „$A \geq (C - 1,5)^2 + 1,75$" braucht man Testdaten mit folgenden Eingabedaten:
$t_1 : C = 1,5; A = 1,75$
$t_2 : C = 1,5; A = 1,74$
t_3, t_4 *: großer positiver Wert von* C, $A = (C - 1,5)^2 + 1,75$ *(bzw. 1,74)*
t_5, t_6 *: großer negativer Wert von* C, $A = (C - 1,5)^2 + 1,75$ *(bzw. 1,74)*
Damit wird z. B. die Abweichung von „$A \geq 2$" sicher gefunden (vgl. Beispiel 9.2.9: Da dort die Relation „$\geq$" durch „$=$" ersetzt ist, sind die Testdaten neben der Grenze, d. h. t_2, t_4, t_6, *nicht nötig).*

Beim Bereichstesten werden die Grenzen der atomaren Prädikatterme für bestimmte logische Verknüpfungen getestet. Ein ähnliches Vorgehen erhält man, wenn man Boolesche Ausdrücke durch eine Kombination der minimalen Mehrfachbedingungsüberdeckung (s. Seite 238) mit dem arithmetischen Relations-Kriterium testet (s. Seite 233). Die Bedingung, deren Wertänderung den Wert des booleschen Ausdrucks ändern kann, sollte dabei mit dem arithmetischen Relations-Kriterium getestet werden, die anderen Bedingungen sollten einen festen Wert haben, und zwar *true* bei *and*-Verbindungen und *false* bei *or*-Verbindungen (s. dazu Übung 9.3b). Beide

[10] Bei $v = 3$ legen drei linear unabhängige Punkte (die also nicht auf einer Geraden liegen) genau eine Ebene im dreidimensionalen Raum fest.

Verfahren (das Bereichstesten und die Kombination der minimalen Mehrfachbedingungsüberdeckung mit dem arithmetischen Relations-Kriterium) setzen voraus, daß die atomaren Bedingungen durch arithmetische Relationen beschrieben werden.

9.3 Mutationsanalyse

Die Testkriterien von Kapitel 9.1 und 9.2 fordern nur, daß *direkt* nach Ausführung der fehlerhaften Anweisung ein fehlerhafter Programmzustand vorliegt. Da sich ein solcher Fehler nicht unbedingt bis zu einer Programmausgabe fortpflanzt, ist eine Orientierung an diesen Testkriterien also keine hundertprozentige Garantie für die Aufdeckung der Fehler.

9.3.1 Ziel, Mittel und Vorgehen der Mutationsanalyse

Ziel der Mutationsanalyse ist die Feststellung, ob eine Testdatenmenge T bestimmte Fehler im Programm *garantiert* entdeckt, bzw. — falls nicht — soll die Fehlerfindungsqualität durch die Erzeugung entsprechender Testdaten verbessert werden.

Mittel der Mutationsanalyse ist die Erzeugung einer ganzen Reihe (Menge) von Programmen $P_1, \ldots, P_n$, die sich vom gegebenen Programm P nur „in einer Kleinigkeit" unterscheiden. Diese P_i heißen — in Analogie zur Biologie — **Mutanten**[11] von P.

Das Vorgehen der Mutationsanalyse besteht nun darin, auf jede Mutante P_i, $i = 1, \ldots, n$, solange Tests t der Testdatenmenge T anzuwenden und die Ergebnisse $P_i(t)$ mit dem Ergebnis $P(t)$ für das unveränderte Programm P zu vergleichen, bis die Mutante „tot" ist (oder klar wird, daß sie nicht „tot" zu kriegen ist).

Dabei sind die Begriffe „tot" und „lebendig" folgendermaßen definiert:

DEFINITION 9.3.1
Für ein Programm P und eine Testdatenmenge T gilt:

1. *Eine Mutante P_i ist* **lebendig** *(bezüglich T und P) g. d. w. $P_i(t) = P(t)$ für alle Tests t aus T gilt.*
2. *Eine Mutante P_i ist* **tot** *(bezüglich T und P) g. d. w. $P_i(t) \neq P(t)$ für mindestens einen Test t aus T gilt.*

[11] Laut Duden (20. Auflage, 1991) ist „der Mutant" sinnverwandt zu „die Mutante", bedeutet aber auch (besonders im Österreichischen) „Jugendlicher im Stimmwechsel".

9.3.2 Bewertung der Testergebnisse

Ein Mutationsanalyse-Ergebnis, bei dem alle Mutanten tot sind, erhöht die Zuversicht, daß die Testdatenmenge T „entsprechende“ Fehler im Programm P finden würde, d. h. entsprechende Fehler sind vermutlich nicht in P vorhanden.

Diese Einschätzung ist aus folgenden Gründen problematisch:

1. Eine lebendige Mutante P_i kann zum Programm P **äquivalent** sein, d. h. für alle Eingabedaten e gilt $P_i(e) = P(e)$. Daher muß *stets* $P_i(t) = P(t)$ für alle Testdaten t gelten.

   Zum Beispiel erzeugt die Veränderung der Abfrage
   „**if** X > 0 **then** ... **else** ... “
   zu
   „**if** $3 * X > 0$ **then** ... **else** ... “ ein äquivalentes Programm.

2. Es müssen genügend viele Mutanten erzeugt worden sein, die alle „typischen“ Fehler enthalten.

3. Der **Kopplungseffekt** müßte gelten: Testdaten, die einfache („kleine“) Fehler aufdecken, entdecken auch große Fehler (Folgen von mehreren Mutationen oder sogar Entwurfsfehler).

Da dies zweifelhaft ist, hat die Mutationsanalyse also auch ihre Grenzen.

Übrig bleiben noch folgende Probleme bzw. Fragestellungen:

1. Wie stark wächst die Zahl der notwendigen Mutanten und Testdaten mit der Programmgröße an? (Antwort: Zahl der Mutanten maximal quadratisch mit der Zahl der Variablenreferenzen, Zahl der Testdaten etwa linear.)

2. Wie erzeugt man genügend Testdaten, so daß alle zu P nicht äquivalenten Mutanten tot sind? (Wegen dieser Schwierigkeit hat man ca. 50% Aufwand für die letzten 10% der lebendigen Mutanten.)

3. Regressionstests sind sehr aufwendig. (Nach einer Programmänderung müssen alle Mutanten noch einmal getestet werden bzw. es muß durch Betrachtung des Kontroll- oder Datenflusses nachgewiesen werden, daß die Änderung sich nicht auf die Eigenschaft „tot (bzgl. T und P)“ auswirkt.)

9.4 Datenbezogenes Testen

Die Testmethoden der Kapitel 7 bis 9.3 haben sich an den Anweisungen eines Programms orientiert und sich auf Kontroll- und Datenfluß zwischen den Anweisungen bezogen. Jetzt orientieren sich die Methoden an den *Datenstrukturen* eines Programms. Die Anweisungsarten *Datenzugriff* und *Datenspeicherung* hatten damit schon am Rande zu tun. Es wurde aber nicht betrachtet, wie sich die Daten zusammensetzen und wie sie verwendet werden. Daten können als Eingabedaten oder Ausgabedaten verwendet werden. Die Zusammensetzung der Daten kann auf verschiedenen Abstraktionsebenen bzw. Verfeinerungsstufen betrachtet werden:

1. Die gröbste Einheit ist eine **Datenkapsel**. Eine Datenkapsel ist eine Menge von zusammenhängenden Daten, die in einer Zugriffseinheit abgespeichert sind.
   Ein Verbund (record), eine Tabelle oder eine Liste sind Beispiele für Datenkapseln.

2. Auf der nächsten Ebene werden die **Datenfelder** betrachtet, aus denen eine Datenkapsel besteht.

3. Für jedes Datenfeld kann eine Menge **repräsentativer Werte** betrachtet werden. Die Spezifikation dieser Werte ergibt sich nicht mehr allein aus der Programmstruktur. Vielmehr müssen Anleihen bei Äquivalenzklassen des spezifikationsorientierten Testens, ihren Grenzwerten und ihren Kombinationen gemäß der UWG-Methode gemacht werden.

4. Die Betrachtung von **Datenzuständen** ist theoretisch möglich. Damit sind alle möglichen Kombinationen von allen repräsentativen Werten für alle Felder einer Datenkapsel gemeint. Da die Anzahl der Datenzustände exponentiell mit der Anzahl der Felder und der Anzahl der repräsentativen Werte steigt, ist diese Betrachtungsebene für das Testen praktisch unmöglich.

Je nach Betrachtungsebene werden folgende **Datenüberdeckungen** gefordert:
Bei Ebene 1 reicht es, wenn irgendein Eingabefeld der Datenkapsel (durch die Testumgebung) oder irgendein Ausgabefeld der Datenkapsel (durch die Programmausführung) einen Wert erhält, um die Datenkapsel als Eingabe bzw. als Ausgabe getestet zu haben. Dieses Kriterium heißt **Datenkapselüberdeckung**.
Bei Ebene 2 muß jedes Eingabefeld mindestens einmal durch die Testumgebung einen Wert erhalten und jedes Ausgabefeld mindestens einen Wert durch die Programmausführung erhalten. Dieses Kriterium wird **Feldüberdeckung** genannt.
Bei Ebene 3 muß jeder spezifizierte repräsentative Eingabewert eines Feldes durch die Testumgebung definiert werden und jeder spezifizierte repräsentative Ausgabewert eines Feldes durch das Programm einen Wert erhalten. Dieses Kriterium heißt **Überdeckung repräsentativer Werte**.

Es ist empfehlenswert, die auf obige Art gewonnenen Ergebnisse der dynamischen Analyse zusammen mit den Ergebnissen der statischen Analyse und den Vorgaben aus der Spezifikation in einem Datenverzeichnis abzulegen.

9.5 Übungen

Übung 9.1:

Betrachten Sie die Anweisung *fill := fill + bufpos* aus dem Textformatierer-Programm (Beispiel 7.3.1 auf Seite 208). Erzeugen Sie Testdaten, die jeweils folgende Testkriterien erfüllen:

(a) Datenzugriffskriterium,

(b) Datenspeicherungskriterium,

(c) additives/multiplikatives Fehler-Kriterium,

(d) kürzere Ausdrücke,

(e) mehrfache Werte für Ausdrücke.

Geben Sie außerdem Testdaten an, die alle fünf Kriterien gleichzeitig erfüllen. Begründen Sie die Wahl der Testdaten, indem Sie zeigen, daß die angenommenen Fehler damit aufgedeckt werden können. (Beachten Sie die Wertebereiche für *fill* und *bufpos*.)

Übung 9.2:

Betrachten Sie folgenden Booleschen Ausdruck aus dem Textformatierer:

$(\mathit{fill} + \mathit{bufpos} < \mathit{MAXPOS})$ **and** $(\mathit{fill} \neq 0)$

Erzeugen Sie Testdaten, die jeweils die folgenden Testkriterien erfüllen und erläutern Sie die Unterschiede. Wählen Sie dabei — wenn möglich — Grenzwerte für die atomaren Prädikatterme $\mathit{fill} + \mathit{bufpos} < \mathit{MAXPOS}$ und $\mathit{fill} \neq 0$.

(a) atomare Bedingungsüberdeckung,

(b) Zweig-/Bedingungsüberdeckung,

(c) Mehrfachbedingungsüberdeckung,

(d) minimale Mehrfachbedingungsüberdeckung,

(e) Bereichsüberdeckung mit $v + 1$ Testdaten pro Grenze (bei v Variablen).

Beachten Sie, daß für die Konstante *MAXPOS* bei verschiedenen Testdaten zu einem Kriterium jeweils derselbe Werte gewählt werden sollte. Beachten Sie die Wertebereiche von *fill*, *bufpos* und *MAXPOS*.

Übung 9.3:

(a) Geben Sie Testdaten an, die das *arithmetische Relations-Kriterium* jeweils für folgende Relationen erfüllen (vgl. Übung 9.2):

   i. $fill + bufpos < MAXPOS$

   ii. $fill \neq 0$

(b) Variieren Sie die logischen Werte, die bei der minimalen Mehrfachbedingungsüberdeckung des gesamten Booleschen Ausdrucks aus Übung 9.2 gefordert werden, mit Testdaten, die das arithmetische Relations-Kriterium erfüllen.

Hinweis: Für die Wertebereiche der Variablen *fill* und *bufpos* und für die Wertewahl bei *MAXPOS* gilt dasselbe wie bei Übung 9.2.
Frage: Welche zusätzlichen Tests sind erforderlich, wenn *fill* negative Werte annehmen kann?

Übung 9.4:

Betrachten Sie den Booleschen Ausdruck aus Beispiel 9.2.1:

$A > 1$ **and** $(B = 2$ **or** $C > 1)$ **and** D

(a) Geben Sie Testfälle an, die jeweils die ersten vier Testkriterien aus Übung 9.2 (ohne Bereichsüberdeckung) erfüllen und erläutern Sie die Unterschiede.

(b) Welche Testfälle können bei der minimalen Mehrfachbedingungsüberdeckung weggelassen werden, wenn für jeden der vier atomaren Prädikatterme nur je ein Testfall gefordert wird, bei welchem ein Wertewechsel von *true* nach *false* bzw. von *false* nach *true* einen Wertewechsel des Gesamtausdrucks bewirkt.
Beispiel-Testfall 1:
$(A > 1) = true, (B = 2) = false, (C > 1) = true, D = true.$
Beispiel-Testfall 2:
$(A > 1) = false, (B = 2) = false, (C > 1) = true, D = true.$
Beide Testfälle reichen aus, um den Effekt des Wertewechsels des atomaren Prädikatterms „$A > 1$“ zu bemerken.

Übung 9.5:

In Kapitel 9.2 wurden beim Bereichstesten nur Bereiche getestet, die durch die Relationen $<, \leq, >, \geq$ abgegrenzt sind. Wieviele Testdaten sind notwendig, wenn der Bereich durch die Gleichheitsrelation „=“ abgegrenzt ist?

(a) Geben Sie die Anzahl der Testdaten an, wenn der Bereich durch eine Gleichheitsrelation mit nur zwei Variablen linear beschrieben ist. Wählen Sie als Beispiel $y = x + 3$.
Diskutieren Sie die Anzahl und Lage der Testdaten für folgende Fehlerfälle, bei denen der korrekte Bereich beschrieben ist durch:

i. Gleichheitsrelation $=$ mit anderer Grenze,

ii. Relation $\leq, <, >, \geq$ und gleiche oder andere Grenze,

iii. Ungleichheitsrelation $\neq$ und gleiche oder andere Grenze.

(b) Geben Sie die Anzahl der Testdaten an, wenn der Bereich durch eine Gleichheitsrelation mit n Variablen linear beschrieben ist, $n > 2$.

Übung 9.6:

(a) Testen Sie das Programm von Abbildung 7.2 auf S. 193 mit folgenden Testdaten:
$t_1 : N = 0$ (d. h. leeres Array), Rest beliebig
$t_2 : F = 2, N = 5, A = (2, 5, 7, 9, 12)$, d. h. $A[1] = 2$, etc.
$t_3 : F = 9$, N und A wie bei t_2
Gibt es Anweisungen oder Zweige, die mit t_1 bis t_3 nicht ausgeführt werden?

(b) Betrachten Sie folgende Mutanten des SEARCH-Programms:

- M1: Bei Anweisung 9 *(LOW := MID + 1)* wird „+1“ weggelassen.
- M2: Bei Anweisung 10 *(HIGH := MID − 1)* wird „−1“ weggelassen.
- M3: Mutante mit den Fehlern von M1 *und* M2.

Sind die Mutanten tot oder lebendig bezüglich $T = \{t_1, t_2, t_3\}$?

9.6 Verwendete Quellen und weiterführende Literatur

Der **fehlerorientierte Ansatz**, der Kapitel 8 und 9 zugrunde liegt (falsche Anweisungen oder Referenz von vorher falsch berechneten Werten) entspricht der Fehlerlokalisierungsmethode von Weiser und dem **RELAY-Modell** von Richardson/Thompson (s. [Wei 82], [RiT 88]). Theoretische Einsichten in dieses „fehlerbasierte“ Testen vermittelt Morell in [Mor 88]. Die Unterscheidung der fünf **Anweisungsarten** Datenzugriff, Datenspeicherung, arithmetischer Ausdruck, arithmetische Relation und Boolescher Ausdruck und entsprechende Ansätze zum Aufdecken von Fehlern in solchen Anweisungen stammen von Howden (s. [How 78d], [How 81], [How 87]). Das **kürzere Ausdrücke**-Kriterium wurde von Weiser übernommen (s. [WGM 85], shorter expressions). Es wurde ursprünglich von Hamlet vorgeschlagen (s. [Ham 77]). Das Kriterium **mehrfache Werte für Ausdrücke** stammt ebenfalls von Weiser (s. [WGM 85], multi-value expressions).

Die **C_2(mM)-Überdeckung** wurde von mir (s. [Rie 86]) unabhängig von dem sehr ähnlichen Begriff *sensitive test data* von Foster (s. [Fos 84]) definiert. Die weitere Reduzierung der Testfälle bei der $C_2(mM)$-Überdeckung (wie bei Übung 9.4b angeregt) entspricht dem Algorithmus BOR_GEN von Tai (s. [Tai 93]). Die stärkere Forderung **Mehrfachbedingungsüberdeckung** stammt von Myers (s. [Mye 79], Kap. 4). Die Idee, die $C_2(mM)$-Überdeckung mit dem arithmetischen Relations-Kriterium zu kombinieren, entspricht der **BRO**-(Boolesche und Relationale Operatoren)-**Überdeckung** von Tai (s. [Tai 93]). Die Empfehlung, bei der **Bereichsüberdeckung** die linearen Grenzen der Eingabebereiche bei v Variablen jeweils mit $v + 1$ bzw. $v + 3$ Testdaten zu testen, stammt von White/Cohen (s. [WhC 80], domain testing); die Empfehlung, besser mit $2 * v$ bzw. $3 * v$ Testdaten zu testen, dagegen von Clarke/Hassell/Richardson (s. [CHR 82]). Ein Ansatz, „lineare Fehler" in nichtlinearen Prädikaten oder Funktionen aufzudecken, wird von Afifi et al. in [AWZ 92] vorgestellt. Einen pragmatischen, fehlersensitiven Ansatz zur Testdatenermittlung präsentiert Foster in [Fos 85].
Das Konzept der **Mutationsanalyse** stammt von Budd et al. (s. [BLS 78], [Bud 81]). DeMillo gibt Abschätzungen zur Anzahl der notwendigen Mutanten und Testdaten an (s. [DeM 89]). Das Problem der Erzeugung von geeigneten Testdaten wird z. B. in [Rie 92a] behandelt, das Problem der Äquivalenz von Mutanten in [Zei 83], S. 339 f. Die Gültigkeit des **Kopplungseffekts** hat Offutt untersucht (s. [Off 92]). Abwandlungen der Mutationsanalyse wurden von Howden (**schwache Mutationsanalyse**) und Zeil (**Perturbationstest**) vorgeschlagen (s. [How 82a], [Zei 83]), wobei das **EQUATE-Testen** beide Ansätze vereint (s. [Zei 86]). Ansätze zur effizienten Speicherung der vielen Mutanten und schnellen Durchführung der Tests auf (Parallel-)Rechnern werden von Untch et al., Weiss/Fleyshgakker und Krauser et al. vorgestellt (s. [UOH 93], [WeF 93], [KMR 88]). Die Aufwandsreduktion durch selektive Mutationen bzw. schwache Mutationsanalyse untersuchen Offutt et al. empirisch (s. [ORZ 93], [OfL 94]). Das Konzept der **Mutationsanalyse** kann auch auf **Spezifikationen** angewandt werden; damit können z. B. fehlende Pfade im Programm erkannt werden (genaueres s. [BuG 85]).
Sneed hat vorgeschlagen, beim **datenbezogenen Testen** die Zusammensetzung der Daten auf verschiedenen Abstraktionsebenen bzw. Verfeinerungsstufen zu betrachten (s. [Sne 86]); von ihm stammt auch die Empfehlung, die Ergebnisse der dynamischen Analyse zusammen mit den Ergebnissen der statischen Analyse und den Vorgaben aus der Spezifikation in einem Datenverzeichnis abzulegen.
Wie ein Datenverzeichnis als Grundlage für die **Testdatenermittlung für Datenbanken** benutzt werden kann, wird z. B. von Deutsch und Beizer in ihren Büchern ausgeführt (s. [Deu 82], Kap. 5.2; [Bei 83], Kap. 8). Leser, die sich für spezielle Methoden zum **Testen von Compilern und Grammatiken** interessieren, seien auf die Bücher von Beizer, DeMillo und einen Artikel von Morganti verwiesen (s. [Bei 83], Kap. 7; [De& 87], Kap. 2.2.9 und Kap. 2.2.11; [Mor 84]).

10 Bewertung der implementationsorientierten Testkriterien

In den Kapiteln 7 bis 9 wurde eine Fülle von Testkriterien vorgestellt. Da beim Testen aus Aufwandsgründen nicht alle Methoden angewandt werden können, ist ein Vergleich und eine Bewertung der Kriterien dringend nötig, damit eine angemessene Menge von Kriterien ausgewählt werden kann.
Dabei können folgende Vergleichsmaßstäbe an die Testkriterien angelegt werden (vgl. Anfang von Kapitel 6).

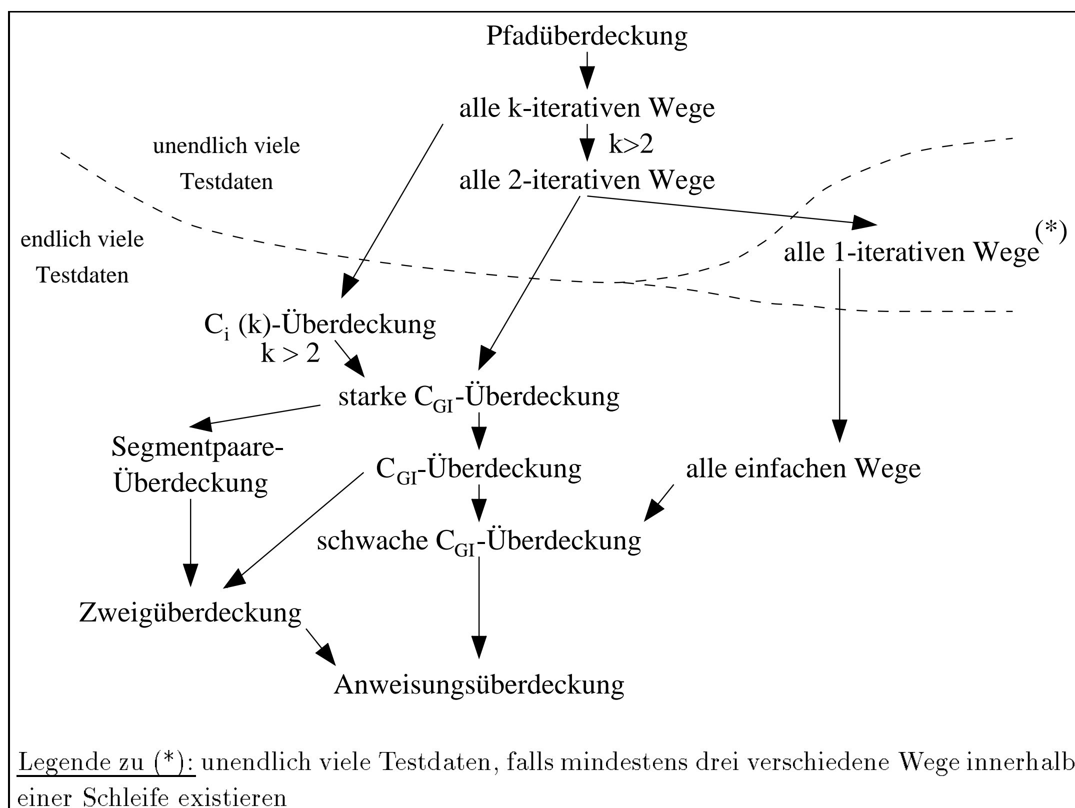
1. Ein Vergleich der Art „Testkriterium K_1 *enthält* Testkriterium K_2“ (siehe Kapitel 10.1).
2. Die *Anzahl* der Testdaten, die notwendig sind, um ein bestimmtes Kriterium zu erfüllen.
   Die Anzahl hängt natürlich von der Anzahl der Segmente oder Knoten des Kontrollflußgraphen oder sonstigen Größen des Programms ab. Daher können nur Abschätzungen für den schlimmsten Fall angegeben werden (siehe Kapitel 10.2).
3. Ein Testkriterium, welches ein anderes enthält, wird in vielen Fällen mehr *Fehler aufdecken* können[1]. Empirische Ergebnisse und formale Analysen für eine Menge von Programmen sollten aber eine solche Aussage mit konkreten Zahlen belegen oder mit Gegenbeispielen widerlegen (siehe Kapitel 10.3).

10.1 Enthaltensein und Unvergleichbarkeit von Testkriterien

Die kontrollflußbezogenen Testkriterien beziehen sich alle auf die Menge der Wege (in dem Kontrollflußgraphen des Programms), die von einer Testdatenmenge auszuführen sind. Sie sind daher gut zu vergleichen. Entsprechendes gilt für die datenflußbezogenen Testkriterien, da sie sich alle auf Wege im Datenflußgraphen des Programms beziehen. Da der Datenflußgraph eine Verfeinerung des Kontrollflußgraphen ist, lassen sich auch Beziehungen zwischen beiden Testkriterienarten angeben.

[1] Dies gilt für den Erwartungswert entsprechender Fehleraufdeckungsmaße, wenn ein Testkriterium für die Testabbruch-Entscheidung (aber nicht bei der Testdatenerzeugung) herangezogen wird (s. [Zhu 96], S. 253).

Die partielle Ordnung der Kriterien wird wegen der Übersichtlichkeit durch *zwei* gerichtete Graphen dargestellt. Eine Kante von Kriterium K_1 nach K_2 bedeutet, daß K_1 das Kriterium K_2 **strikt enthält**, d. h., daß K_1 Kriterium K_2 enthält, aber K_2 nicht K_1 enthält (K_1 enthält K_2 g. d. w. jede Testdatenmenge, die K_1 erfüllt, auch K_2 erfüllt[2]). Falls kein Weg von K_1 nach K_2 existiert, sind die Kriterien unvergleichbar.



Legende zu (*): unendlich viele Testdaten, falls mindestens drei verschiedene Wege innerhalb einer Schleife existieren

Abb. 10.1: Partielle Ordnung einiger kontrollflußbezogener Testkriterien

Die Gültigkeit der partiellen Ordnung der kontrollflußbezogenen Kriterien aus Abbildung 10.1 wird zum großen Teil von Ntafos bewiesen (s. [Nta 88]); nur die Kriterien *alle k-iterativen Wege*, *alle einfachen Wege*, *Segmentpaareüberdeckung* und die Differenzierung in die drei Arten der C_{GI}*-Überdeckung* sind dort nicht berücksichtigt. Die richtige Einordnung dieser Kriterien ergibt sich „direkt" aus den Definitionen, der Übung 10.1 sowie Satz 10.1.1. Dabei ist noch zu beachten, daß die Kriterien $C_i(k)$*-Überdeckung* und *alle k-iterativen Wege* einen Parameter k enthalten. Bei Variation dieses Parameters ergibt sich jeweils eine Schar von sich strikt enthaltenden Kriterien.

[2]Da die Enthaltensein-Relation die Fehleraufdeckungsrelation der Kriterien in vielen Fällen nicht hinreichend widerspiegelt, haben Frankl/Weyuker andere Relationen („K_1 [universally] [properly] covers K_2") vorgeschlagen (s. [FrW 93])

Satz 10.1.1

1. *Das Kriterium „starke C_{GI}-Überdeckung“ enthält das Kriterium „Segmentpaareüberdeckung“ strikt.*
2. *Das Kriterium „Segmentpaareüberdeckung“ enthält das Kriterium „Zweigüberdeckung“ strikt.*

Beweis:
Zunächst wird das Enthaltensein der Kriterien bewiesen.

1. Jeder Weg (der eine Schleife bis zu zweimal durchläuft) wird bei der starken C_{GI}-Überdeckung ausgeführt, also auch jedes Paar von Entscheidungs-Entscheidungs-Wegen. Damit ist die Segmentpaareüberdeckung erfüllt (s. Definition 7.2.6 auf S. 200).
2. Die Segmentüberdeckung impliziert schon die Zweigüberdeckung (s. Satz 7.2.1 auf S. 197). Also gilt dies erst recht für die Segmentpaareüberdeckung.

q. e. d.

Nun wird die Striktheit des Enthaltenseins der Kriterien bewiesen.

1. Beispiel 10.1.1
   Sei das folgende abstrakte Programm gegeben:
   if p **then** A **else** B;
   if q **then** C **else** D;
   if r **then** E **else** F;

   Eine Testdatenmenge T, welche die vier Wege ACE, ADE, BCF und BDF ausführt, erfüllt das Kriterium Segmentpaareüberdeckung. (Bei den Wegen wurde nur die Folge der Segmente pro Weg angegeben.)
   Der Weg BCE z. B. wird nicht ausgeführt. Die Testdatenmenge erfüllt also nicht die starke C_{GI}-Überdeckung, die jeden schleifenfreien Weg enthalten muß.
2. Beispiel 7.2.2 auf S. 199 zeigt, daß das Enthaltensein der Kriterien strikt ist.

q. e. d.

Der Beweis für die Gültigkeit der partiellen Ordnung der datenflußbezogenen (und kontrollflußbezogenen) Kriterien aus Abbildung 10.2 wird von Clarke et al. erbracht (s. [Cl& 89], Fig. 8) sowie Ntafos (s. [Nta 88], Fig. 1) und Übungen 10.1 und 10.2.

Die verschiedenen Bedingungsüberdeckungen lassen sich schlecht in Abbildung 10.1 und 10.2 einordnen, da sie (abgesehen von der Zweigüberdeckung) unabhängig sind von der Menge der Wege, die von einer Testdatenmenge auszuführen sind. Daher wird ein zusätzlicher Ordnungsgraph in Abbildung 10.3 angegeben.

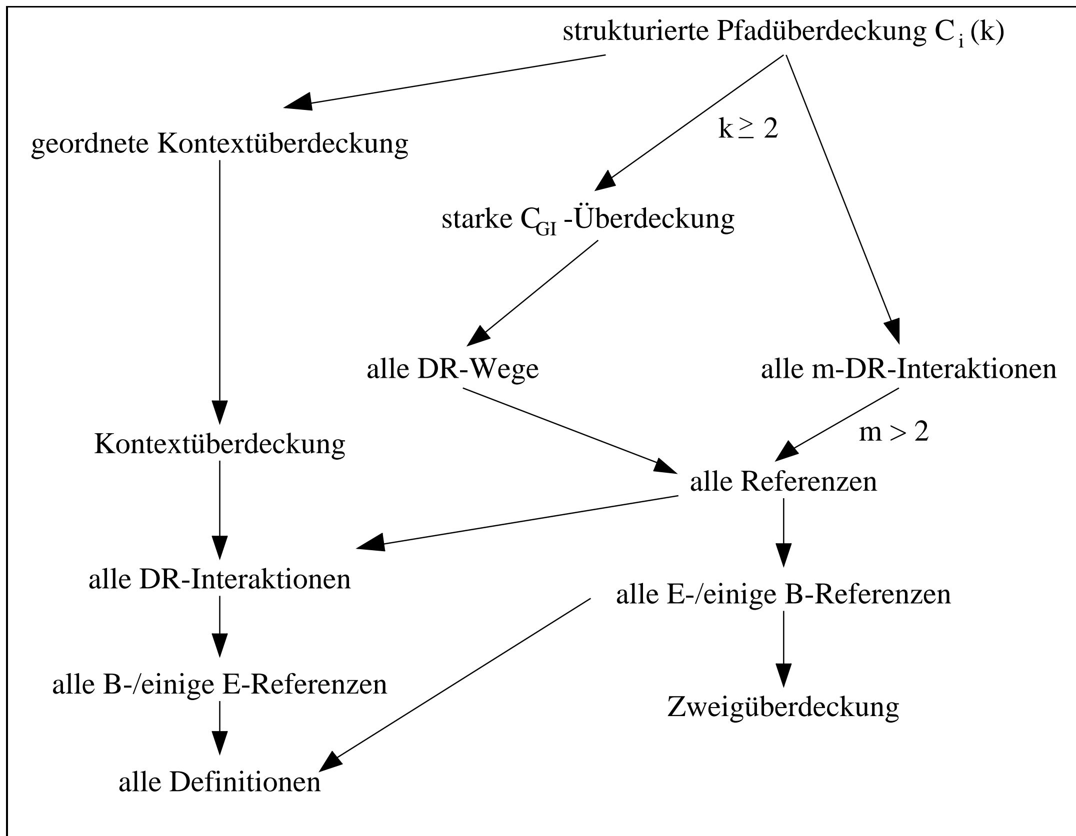


Abb. 10.2: Partielle Ordnung einiger kontrollflußbezogener und datenflußbezogener Testkriterien

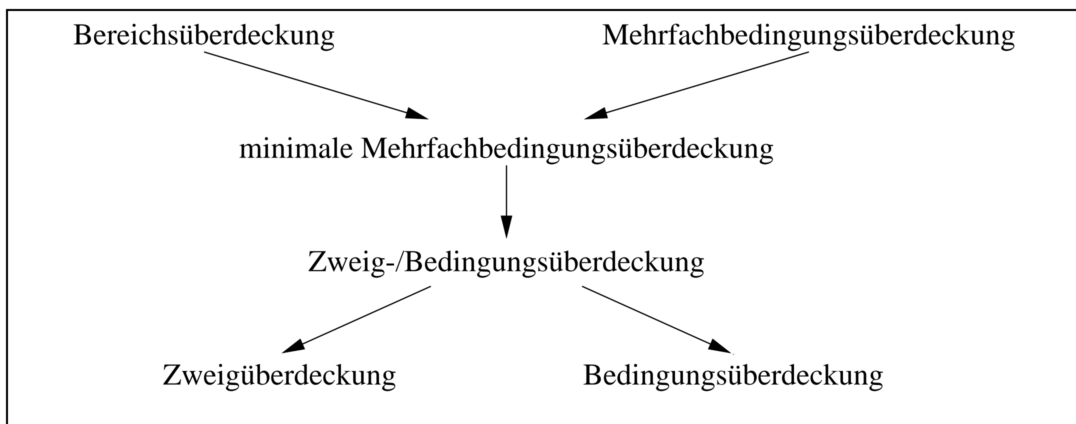


Abb. 10.3: Partielle Ordnung der Bedingungskriterien

Der Beweis des Enthaltenseins ergibt sich aus dem folgenden Satz, Übung 10.3 bzw. direkt aus den Definitionen. Das strikte Enthaltensein ergibt sich aus den Beispielen aus Kapitel 9.2.

Satz 10.1.2
Voraussetzung: In den Prädikaten der Entscheidungen kommen die konstanten logischen Operatoren mit Ergebnis true bzw. false nicht vor.
Dann gilt: Das Kriterium $C_2(mM)$-Überdeckung (minimale Mehrfachbedingungsüberdeckung) enthält

1. *das Kriterium C_1-Überdeckung (Zweigüberdeckung);*

2. *das Kriterium „(atomare) Bedingungsüberdeckung“, wenn nur die beliebigstelligen Operatoren „und“, „oder“, „exor“, die zweistellige „Implikation“ oder die einstelligen Operationen „Negation“ und „Identität“ vorkommen.*

Beweis:

1. Bei logischen Operatoren, die sowohl den Wert $true$ als auch den Wert $false$ annehmen können, müssen wegen der Anforderung in der Definition 9.2.3 von $C_2(mM)$ ebenfalls die Ausgänge $true$ und $false$ vorkommen.

2. Bei Prädikaten mit den logischen Operatoren „und“, „oder“, „exor“, „Implikation“, „Negation“ und „Identität“ werden gerade solche Kombinationen von Wahrheitswerten der einzelnen Terme ausgeführt, daß die einzelnen Terme und die gesamte Entscheidung sowohl die Werte *true* als auch *false* annehmen (vgl. Beispiele 9.2.4 bis 9.2.7).

q. e. d.

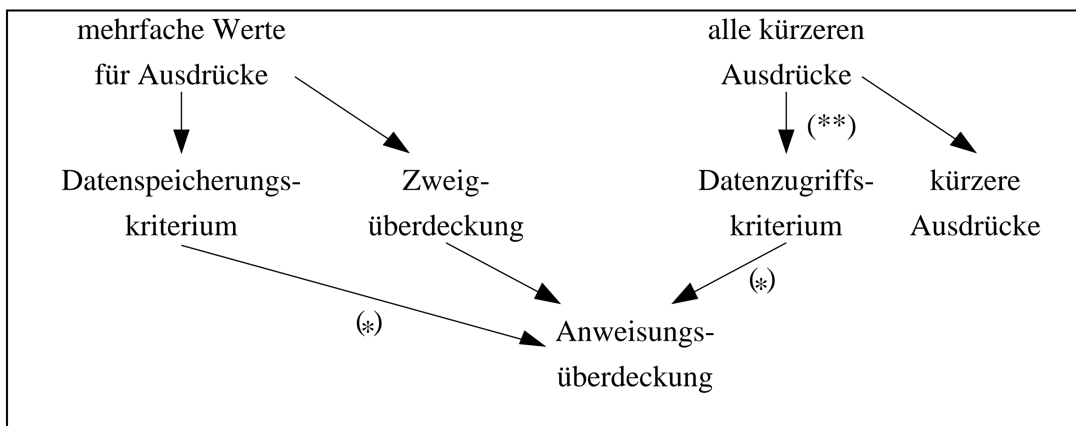


Abb. 10.4: Partielle Ordnung der ausdrucks- und anweisungsbezogenen Testkriterien

Die Kriterien, die sich nur auf einzelne Ausdrücke und Anweisungen im Programm beziehen, lassen sich kaum mit den wegebezogenen Kriterien vergleichen. Dies bedeutet gleichzeitig, daß die Kriterien Anlaß zu Testdaten geben, die andere Fehler aufdecken können als die wegebezogenen Tests. Wegen der Anforderungen an die Ausdrücke in Entscheidungsprädikaten gibt es nur eine Beziehung zum Kriterium *Zweigüberdeckung*. Wenn in wenigstens einer Anweisung eines Entscheidungs-Entscheidungs-Weges ein Zugriff (auf) bzw. eine Speicherung von Daten erfolgt [Bedingung (*)] , gibt es zusätzlich noch eine Beziehung zwischen dem *Datenzugriffskriterium* bzw. dem *Datenspeicherungskriterium* und der *Anweisungsüberdeckung*. Insgesamt gelten daher die in Abbildung 10.4 dargestellten Enthaltenseinbeziehungen und Unvergleichbarkeiten (s. auch Übung 10.4 und [WGM 85], S. 81). Dabei wird vorausgesetzt [Bedingung (**)], daß beim Kriterium *alle kürzeren Ausdrücke* auch die Variablen auf den linken Seiten von Zuweisungen, sowie globale Variablen und Parameter als „Ausdrücke“ interpretiert werden, oder daß alle diese Variablen und Parameter in mindestens einem Ausdruck im Programm vorkommen.

Aufgrund der Definition der Kriterien des datenbezogenen Testens (s. Kap. 9.4) sind die in Abbildung 10.5 dargestellten Beziehungen offensichtlich. Da die datenbezogene Sichtweise orthogonal zu den anderen Sichtweisen (Kontrollfluß, Datenfluß, Anweisungen und Ausdrücke) liegt, sind die Testkriterien unvergleichbar mit den anderen Testkriterien. Dies bedeutet wiederum, daß eine andere Klasse von Fehlern aufgedeckt werden kann, wenn datenbezogene Testkriterien herangezogen werden.

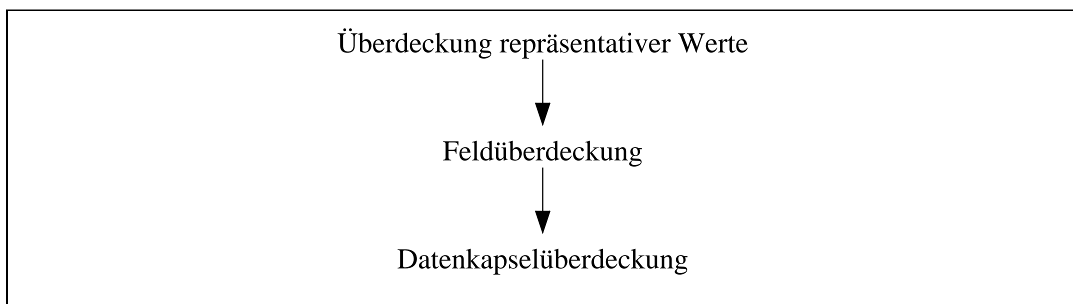


Abb. 10.5: Ordnung der datenbezogenen Testkriterien

Die Mutationsanalyse läßt sich mit den anderen Testkriterien nur vergleichen, wenn die Art der Mutationen genau spezifiziert wird — was bei Kapitel 9.3 offen gelassen wurde. Wird z. B. jeweils eine Anweisung durch den Aufruf einer Fehlerroutine (mit *exit* im Kontrollfluß) ersetzt, läßt sich damit die Anweisungsüberdeckung C_0 ermitteln.

Schränkt man die Forderungen der Testkriterien auf *ausführbare* Wegstücke ein, erhält man zwar sinnvolle — aber auch unentscheidbare — Kriterien (genaueres s. Abschnitt 11.2.2). Außerdem müssen manche Kriterien dann anders definiert werden, um das intuitive Konzept des Kriteriums richtig wiederzugeben. Beispielsweise müssen beim Kriterium *alle DR-Wege* mehrfache Durchläufe einer Schleife (z. B.

for $i := 1$ **to** 5) erlaubt werden, damit eine Definition vor der (*for*-)Schleife eine Referenz danach erreicht (genaueres s. [PaZ 95]; vgl. Übungen 7.4, 7.7, 8.2).

10.2 Anzahl der Testdaten pro Testkriterium

Für ein Programm mit n Segmenten kann die Anzahl der notwendigen Testdaten folgendermaßen abgeschätzt werden:

Satz 10.2.1
Falls die Anzahl der ausgehenden Kanten pro Entscheidungsknoten und die Anzahl der Variablen pro Segment durch eine Konstante begrenzt sind, dann gilt für ein Programm mit n Segmenten im schlechtesten Fall:

1. *Pfadüberdeckung erfordert eine unendliche Anzahl von Testdaten,*
2. *strukturierte Pfadüberdeckung $C_i(k)$, [starke/schwache] C_{GI}-Überdeckung sowie das Kriterium „alle DR-Wege“ erfordern $O(2^n)$ Testdaten,*
3. *die Datenflußkriterien „alle E-/einige B-Referenzen“, „alle B-/einige E-Referenzen“, „alle Referenzen“ und „Kontextüberdeckung“ erfordern $O(n^2)$ Testdaten,*
4. *die Kriterien „alle Definitionen“ sowie die Zweig- und die Anweisungsüberdeckung erfordern $O(n)$ Testdaten.*

Beweis:

1. Folgendes Programm erfordert unendlich viele Testdaten:

   ```
   while n > 0 do
         begin n := n − 1;
                 x := x + n;
         end
   ```

   Für jede natürliche Zahl n_0, die Anfangswert von n ist, durchläuft das Programm einen anderen vollständigen Weg. Also gibt es unendlich[3] viele ausführbare Wege, die zu testen sind.

2. Sei ein Programm gegeben, welches aus einer Sequenz von n *if-then-else*-Anweisungen besteht und daher die Maximalzahl von 2^n verschiedenen Wegen enthält. Sei x eine Variable, die nur beim Programmstart definiert wird und in der letzten *if-then-else*-Anweisung in beiden Zweigen referenziert wird. Dann erfordern die angegebenen Kriterien die Ausführung aller 2^n Wege.

[3] Bei der Darstellung von natürlichen Zahlen durch Integer-Variablen mit z. B. 2^{16} Werten gibt es tatsächlich „nur“ endlich viele — aber sehr viele Wege.

3. Der Beweis findet sich bei Ntafos (s. [Nta 88], S. 872). Die Grundidee ist dabei, daß bei allen Kriterien Paare gebildet werden müssen, wofür es $\binom{n}{2}$, d. h. $O(n^2)$, Möglichkeiten gibt. Die Variablenanzahl kann als zusätzlicher Faktor ausgeschlossen werden, da sie laut (berechtigter) Annahme nicht unabhängig von der Anzahl der Segmente variiert werden kann.

4. Für die genannten Kriterien reicht gerade die Ausführung aller Segmente aus. Der schlimmste Fall ist ein Programm P mit verschachtelten *if-then-else*-Anweisungen der Form „**if** B **then** $S1$ **else** $S2$“, wobei $S1$ und $S2$ entweder beide ebenfalls von dieser Form sind oder beide nicht weiter zerlegt sind. Die Schachtelungstiefe der *if-then-else*-Anweisungen sei k.
Dann enthält P die Anzahl $n = 2 + 4 + 8 + 16 + \cdots + 2^k = 2^{k+1} - 2$ von Segmenten, für deren Überdeckung 2^k Wege erforderlich sind. Da $n = 2^{k+1} - 2$ gilt, ist $2^k = \frac{n+2}{2}$, also von der Größenordnung $O(n)$, d. h. linear in n. (Für Entscheidungen mit $c > 2$ Ausgängen gilt entsprechendes.)

q. e. d.

Die Segmentpaareüberdeckung erfordert mehr Testdaten als die Anweisungs- oder Zweigüberdeckung. Bei einer Sequenz von *if-then-else*-Anweisungen reichen für die Anweisungs- oder Zweigüberdeckung z. B. zwei Testdaten aus, bei der Segmentpaareüberdeckung sind aber vier Testdaten notwendig[4] und bei passender Wahl der Testdaten auch hinreichend (s. Übung 10.7).
Im Normalfall liegt die Zahl der Testdaten oft unter den in Satz 10.2.1 angegebenen Werten[5]. Dennoch sind die Größenordnungen zu beachten. Unter praktischen Gesichtspunkten sind nur Testkriterien akzeptabel, die einen Aufwand von $O(n)$ oder höchstens $O(n^2)$ erfordern, wobei bei $O(n^2)$ nur kleine Module mit entsprechend kleinen Werten von n getestet werden können.

Beispiel 10.2.1
Für das Kriterium „alle DR-Wege“ ergab sich bei einer Untersuchung von Weiser et al. der in Tabelle 10.1 dargestellte Zusammenhang zwischen der Zahl der Anweisungen eines Moduls und einer minimalen[6] Zahl von Testdaten (s. [WGM 85], S. 84).

Beim Bedingungstesten kann bei den verschiedenen Kriterien nur der Aufwand für das Testen einer einzelnen Entscheidung miteinander verglichen werden. Wenn

[4] *then*- oder *else*-Zweig kombiniert mit dem nachfolgenden *then*- oder *else*-Zweig ergibt $2 * 2 = 4$ verschiedene Wegstücke

[5] Weyuker fand experimentell, daß die *alle DR-Wege*-Überdeckung nur etwa zweimal soviele Testdaten wie die *alle Definitionen*-Überdeckung benötigt (s. [Wey 88b], S. 192).

[6] „minimal“ in dem Sinne, daß die *alle DR-Wege*-Überdeckung verletzt wird, wenn irgendein Testdatum aus der Testdatenmenge entfernt wird. Bei den Modulen „Record“ und „File“ gibt es jedoch einen nicht ausführbaren bzw. extra nicht ausgeführten Weg. Beim Modul „Pattern“ ist die Erfüllung des Überdeckungskriteriums schwierig, da das Modul mehrere rekursive Prozeduren enthält. Daher werden 200 DR-Wege durch die 1607 Testdaten *nicht* überdeckt.

Modulname	Anweisungen	Testdaten
Queue	41	59
List	72	72
Record	78	221
File	105	233
Pattern	362	1607

Tab. 10.1 Testaufwand für das Kriterium *alle DR-Wege*

die Entscheidung n atomare Prädikatterme enthält, die unabhängig von einander Werte annehmen können, dann gilt der in Tabelle 10.2 dargestellte Zusammenhang (wobei die Fußnoten zu beachten sind).

Kriterium	Zahl der erforderlichen Tests pro Entscheidung mit n Termen
Bedingungsüberdeckung:	2
minimale Mehrfachbedingungsüberdeckung[7]:	$\leq n+1$
Bereichsüberdeckung mit linearen Grenzen in v Variablen:	$\leq n*(v+3)$
Mehrfachbedingungsüberdeckung[8]:	$\leq 2^n$

Tab. 10.2 Erforderliche Anzahl von Tests bei verschiedenen Kriterien

In der Praxis enthalten Entscheidungsprädikate oft nur ein bis zwei atomare Prädikatterme. Für $n = 1$ ist die Zahl der Tests also stets 2, außer bei der Bereichsüberdeckung, wo die Zahl der Tests gleich der Anzahl v der Variablen plus 3 ist. Wird im folgenden $v = n$ angenommen, so gilt: Für $n = 2$ sind die entsprechenden Werte 2, 3, $2*(v+3) = 10$ und 4 in obiger Reihenfolge. Erst ab $n = 6$ wird der große Aufwand für die Bereichsüberdeckung ($6*(v+3) = 6*9 = 54$ Testdaten) vom Aufwand für die Mehrfachbedingungsüberdeckung ($2^6 = 64$ Testfälle) überholt. Entsprechend komplexe Entscheidungen kommen z. B. in Computergraphikanwendungen vor:

Beispiel 10.2.2
Die Abfrage, ob ein Punkt (x, y, z) im dreidimensionalen Raum in einem achsenparallel liegenden Quader mit den Grenzen (x_1, x_2), (y_1, y_2), (z_1, z_2) liegt, erfordert folgende Abfrage mit sechs atomaren Prädikattermen:

$$x_1 \leq x \textbf{ and } x \leq x_2 \textbf{ and } y_1 \leq y \textbf{ and } y \leq y_2 \textbf{ and } z_1 \leq z \textbf{ and } z \leq z_2$$

Die Zahl der Testfälle für die vier Kriterien differiert hier deutlich:

[7] Bedingung $(*)$: Das Entscheidungsprädikat enthält nur den Operator *and* oder *or*. (Dann folgt die Abschätzung aus Ungleichung 6.9 in Abschnitt 6.2.1, da die UWG-Methode nach den gleichen Prinzipien vorgeht.) Die Bedingung $(*)$ kann entfallen, wenn die Testfälle wie in Übung 9.4b reduziert bzw. wie bei Tai's Algorithmus BOR_GEN gebildet werden (s. [Tai 93], Theorem 4).

[8] 2^n, falls alle 2^n Kombinationen erfüllbar sind, sonst weniger

Die Bedingungsüberdeckung erfordert nur zwei Testfälle, die allerdings keine Zweigüberdeckung implizieren, wie folgende Testfälle zeigen:

Testfall 1: $x < x_1 < x_2, y < y_1 < y_2, z < z_1 < z_2$

Testfall 2: $x_1 < x_2 < x, y_1 < y_2 < y, z_1 < z_2 < z$

Die minimale Mehrfachbedingungsüberdeckung erfordert $6 + 1 = 7$ *Testfälle, die Bereichsüberdeckung erfordert bei drei Variablen maximal* $(3 + 3) * 6 = 36$ *Testdaten und die Mehrfachbedingungsüberdeckung erfordert maximal* $2^6 = 64$ *Testfälle. Tatsächlich erfordert die Bereichsüberdeckung höchstens 18 Testdaten und die Mehrfachbedingungsüberdeckung nur 27 Testfälle (s. Übung 10.5). In diesem Fall ist also der lineare Aufwand bei der minimalen Mehrfachbedingungsüberdeckung (7) und der Aufwand bei der Bereichsüberdeckung (12 oder 18) spürbar geringer als der Aufwand bei der Mehrfachbedingungsüberdeckung (27).*

10.3 Aufgedeckte Fehler pro Testkriterium

Die vorhergehenden Abschnitte beschäftigten sich mit dem Aufwand, den die Erfüllung der Testkriterien bedingt. Der Aufwand wurde in Kapitel 10.1 durch einen Vergleich der Kriterien nur relativ angegeben und in Kapitel 10.2 als Anzahl der notwendigen Testdaten absolut angegeben. Der Aufwand für das Bestimmen eines Testdatums wurde hier außer acht gelassen, obwohl dies für die verschiedenen Kriterien differieren kann (genaueres siehe Kapitel 11.2).

Dem Aufwand für die Testdatenerzeugung — den Kosten — wird nun die Zahl und Art der aufgedeckten Fehler — der Nutzen — gegenübergestellt. Dabei wird der Nutzen aufgrund von empirischen Untersuchungen angegeben. Die diversen Untersuchungen unterscheiden sich allerdings vom Ansatz her. Es gibt folgende Untersuchungsarten: Fallstudien, formale Analysen, Mutationsanalyse und statistische Experimente[9]. Für einige wenige Methoden gibt es auch theoretische Untersuchungen, die allerdings von einschränkenden Annahmen über die Programme ausgehen.

Bei der Zuordnung der gefundenen Fehler zu den angewandten Methoden gibt es folgendes Problem:

Beispiel 10.3.1

Das Kriterium „Zweigüberdeckung“ sei zugrundegelegt. Wenn mindestens ein Segment noch nicht mit den bisherigen Testdaten überdeckt ist, muß man sich dieses Segment und einen Weg vom Programm- oder Modulanfang bis zu diesem Segment genau ansehen, um ein geeignetes Testdatum zu bestimmen. Falls dies überwiegend durch persönliches Hinsehen und nicht durch Werkzeugunterstützung geschieht, kann der Testperson ein Fehler auf dem betrachteten Weg im Programm auffallen. Dann stellt sich die Frage, welcher Methode der Fehler seine Entdeckung

[9] Zur Definition der Untersuchungsarten sei auf den Anfang von Kapitel 6.3 und auf Kapitel 9.3 verwiesen.

verdankt: der Zweigüberdeckung oder der Programminspektion (genaueres zu letzterem s. Kap. 12.1)?

Entsprechendes gilt, wenn mit Werkzeugunterstützung ein Bericht (Report) erstellt wird, der auf nicht überdeckte Konstrukte (Anweisungen, Zweige, Datenflußwege) und auf fehlende Testdaten für das Testen von Anweisungen und Ausdrücken hinweist: Durch sorgfältiges Lesen dieser Berichte konnten in einer Fallstudie von Girgis und Woodward in 25 von 80 Fällen Fehler „aufgedeckt“ werden, obwohl die Programme korrekte Ausgaben erzeugten, und in einem Fall konnte der Fehler nicht „aufgedeckt“ werden, obwohl das Programm inkorrekte Ausgaben lieferte.
Dies bedeutet, daß Girgis/Woodward unter *Aufdecken* eines Fehlers offenbar das Lokalisieren der Fehlerursache im Programmtext verstehen (s. [GiW 86], S. 71). Hier soll aber vom **Aufdecken eines Fehlers** gesprochen werden, wenn ein inkorrektes Verhalten des Programms beobachtet wird. Dies bedeutet in erster Linie die Abweichung der tatsächlichen Ausgaben von den erwarteten Ausgaben. Andere Abweichungen des Verhaltens des Programms vom erwarteten Verhalten sollten nur dann als „Aufdecken eines Fehlers“ interpretiert werden, wenn das korrekte Verhalten klar spezifiziert ist oder die Abweichung offensichtlich ist[10]. Umgekehrt soll beim Aufdecken eines Fehlers nicht verlangt werden, daß der Fehler schon lokalisiert wird. Diese Tätigkeit wird vielmehr gesondert betrachtet (genaueres s. Kapitel 15.2). Hier interessiert nur, daß das Programm bei bestimmten Eingaben ein falsches Verhalten zeigt.

10.3.1 Prozentzahlen der gefundenen Fehler

Mit dem *kontrollflußbezogenen* Testen kann jeweils die folgende Prozentzahl von Fehlern aufgedeckt werden:

Anweisungsüberdeckung: 18% bis 41% (s. [GiW 86], [BaS 87]),

Zweigüberdeckung: 16% bis 92% (s. [How 80], [How 78e], [GiW 86], [Hu& 94], [HHR 84], [Mil 77], [Nta 84a]),

LCMS-Überdeckung[11]: das 1,25fache (95% statt 75%) bis 2,5fache (85% statt 34%) der Zweigüberdeckung (s. [HHR 84], [GiW 86]),

strukturierte Pfadüberdeckung: das zweifache der Zweigüberdeckung (43% statt 21%, s. [How 78b]),

Pfadüberdeckung: das dreifache der Zweigüberdeckung (62% bis 64% statt 21%, s. [How 76], [How 78b]).

[10] Offenbar werden nicht alle Fehler entdeckt, die eigentlich gesehen werden können. Nach Basili und Selby werden nur 70 bis 80% der beobachtbaren Fehler gemeldet (s. [BaS 87], S. 1287, S. 1292).

[11] grob vergleichbar mit der Segmentpaare-Überdeckung

Die Ergebnisse hängen stark von der Art und Größe der untersuchten Programme, den vorhandenen bzw. extra eingestreuten Fehlern und der Untersuchungsmethode ab: Howden wendet die formale Analyse an, Ntafos läßt einfache Fehler durch ein System zur Mutationsanalyse erzeugen, Girgis/Woodward erzeugen ebenfalls Mutanten und nutzen Informationen aus dem umfangreichen Bericht und Informationen über nicht überdeckte Zweige zur Fehleraufdeckung, Hutchins et al. erzeugen (zufällig) 30 Testdatenmengen mit 100% Zweigüberdeckung und geben die Wahrscheinlichkeit an, daß damit ein Fehler gefunden wird.
Die Problematik der Bestimmung der aufgedeckten Fehler wird auch in einer formalen Analyse von Glass deutlich. Ursprünglich wird festgestellt, daß 81% der Fehler zu finden sind. Durch Vergleich mit einer früheren Studie korrigiert Glass den Wert auf nur 32% zu findende Fehler (s. [Gla 80]).

Über Fehler, die mit dem *datenflußbezogenen* Testen (s. Kapitel 8) aufgedeckt werden können, werden hier nur zwei Untersuchungen herangezogen: Hutchins et al. betrachten das Kriterium *alle DR-Interaktionen*, Girgis/Woodward untersuchen die Kriterien *alle Definitionen*, *alle E-/einige B-Referenzen* sowie *alle B-/einige E-Referenzen* (s. [Hu& 94], [GiW 86]).
Mit dem schwachen Kriterium *alle Definitionen* werden keine Bereichsfehler und nur 31% der Berechnungsfehler erkannt, insgesamt 24% aller Fehler. Das Kriterium *alle Entscheidungs-/einige Berechnungsreferenzen* ist fehleraufdeckender, insbesondere natürlich bei Bereichsfehlern: 68% davon werden erkannt, allerdings nur 23% Berechnungsfehler; in der Summe 34% aller Fehler. Das Kriterium *alle Berechnungs-/einige Entscheidungsreferenzen* deckt 57% aller Berechnungs- und 16% aller Bereichsfehler auf, zusammen 48% aller Fehler. Mit dem stärksten Kriterium *alle DR-Interaktionen* können Fehler mit 51% Wahrscheinlichkeit entdeckt werden (s. [Hu& 94], Table 4). Bei der kombinierten Anwendung von vier Datenflußkriterien können 70% aller Fehler — 68% der Bereichsfehler und 71% der Berechnungsfehler — aufgedeckt werden (s. [GiW 86], S. 69).

Das *ausdrucks- und anweisungsbezogene* Testen (s. Kapitel 9.1) wurde nur von Girgis und Woodward untersucht mit folgenden Ergebnissen (s. [GiW 86], S. 69): Mit dem *Datenzugriffskriterium* (falsche Variablenreferenz) werden 18% aller Fehler und sogar 42% der Bereichsfehler erkannt. Mit dem *Datenspeicherungskriterium* (falsche Variablendefinition) lassen sich dagegen 16% aller Berechnungsfehler bzw. 14% aller Fehler aufdecken. Mit dem Kriterium *arithmetische Relation* werden sogar 79% aller Bereichsfehler, aber nur 3% der Berechnungsfehler erkannt, in der Summe 21% aller Fehler.

Die Effektivität der vorgestellten Testmethoden läßt sich nach Girgis/Woodward folgendermaßen grob quantifizieren: kontrollflußbezogenes Testen erkennt 85% der Fehler, datenflußbezogenes Testen deckt 70% der Fehler auf, ausdrucks- und anweisungsbezogenes Testen erkennt 41% der Fehler (s. [GiW 86], S. 69). Da jeweils nur ein Teil der hier vorgestellten Testkriterien herangezogen wurde, ist dieser Vergleich mit Vorsicht zur Kenntnis zu nehmen.

10.3.2 Art der gefundenen Fehler

Als Fragestellung drängt sich auf, für welche Fehler die einzelnen Methoden besonders effektiv sind. Daraus kann abgeleitet werden, welche Methoden beim Testen gemeinsam angewendet werden sollten, um möglichst alle Fehler aufzudecken.

10.3.2.1 Kontrollflußbezogenes Testen

Berechnungsfehler werden schon bei der Anweisungsüberdeckung erkannt, wenn die Berechnungsanweisung produktiv ist, Fehler nicht maskiert werden und keine zufällige Korrektheit vorliegt.

> Eine Anweisung ist **produktiv**, wenn sich ihr Ergebnis auf die Endergebnisse auswirkt.
> Eine **Maskierung** eines Fehlers durch andere Werte liegt beispielsweise im folgenden Fall vor: Ein Fehler $A = 2$ statt $A = 1$ wirkt sich nicht aus, wenn dies später nur bei der Berechnung von $\lceil \frac{A}{2} \rceil$ verwendet wird.
> **Zufällige Korrektheit** liegt beispielsweise für zwei Ausdrücke $y = x$ und $y = x^2$ vor, wenn mit $x = 0$ oder $x = 1$ getestet wird. Die Wahrscheinlichkeit für zufällige Korrektheit ist bei einfachen Programmen, die nur Polynome oder Multinome berechnen[12], relativ klein. Hennell et al. haben dafür in vorliegenden Fallstudien kein einziges Testdatum gefunden (s. [HHR 84], S. 272). Bei Programmen, die Aufrufe von Prozeduren enthalten, kann dies aber sehr wohl auftreten. Die Ausdrücke x und $abs(x)$ — wobei $abs(x)$ den Absolutbetrag von x berechnet — haben z. B. für alle nichtnegativen Werte von x den gleichen Wert. Falls der Fehler in der Vertauschung von x und $abs(x)$ besteht, ergibt sich bei entsprechenden Testdaten eine zufällige Korrektheit.

Ausgabefehler können ebenfalls schon durch Anweisungsüberdeckung aufgedeckt werden, wenn sie nicht von der Größe der Ausgabewerte abhängen und unabhängig von Ausgabekombinationen sind.
Eingabefehler können nur mit entsprechender Sicherheit erkannt werden, wenn auf jede Eingabeanweisung eine Ausgabeanweisung als Kontrolle folgt (s. [HHR 84], S. 271).
Da bei der Zweigüberdeckung, der Segmentpaare-Überdeckung und der Pfadüberdeckung mehr Kombinationen von E/A-Anweisungen ausgeführt werden, steigt die Wahrscheinlichkeit zum Aufdecken solcher Fehler.

[12] vgl. Kapitel 9, Beispiele 9.1.3 und 9.1.4

Die Zweigüberdeckung ist für die Aufdeckung von Fehlern der folgenden Art effektiv (s. [How 80], [HHR 84], [GiW 86]):

1. Falsche Ausgabe in einem speziellen Zweig, z. B. eine falsche Fehlermeldung.

2. Durchlaufen eines unerwarteten Zweigs wegen eines Bereichsfehlers.

3. Nicht ausführbarer Code aufgrund eines Fehlers.

   Beispiel 10.3.2 (zu nicht ausführbarem Code)

   (a) *Eine Fehlerbehandlung ist nicht ausführbar, wenn der Fehler schon an anderer Stelle im Programm behandelt wird.*

   (b) *Bei einem Sortierprogramm soll bei $n \leq 11$ Zahlen durch Einfügen sortiert werden, sonst durch Quicksort. Durch einen Fehler wird auf $n \leq 1$ Zahlen abgefragt, so daß beim Programmteil für das Einfügen nicht erreichbarer Code entsteht.*

4. Abnorm häufiges Durchlaufen eines Zweiges aufgrund eines Fehlers.

5. Bereichsfehler durch falsche Variablenreferenz oder durch eine inkorrekte Konstante.
   Nach Girgis und Woodward werden solche Fehler zu 100% erkannt[13] (s. [GiW 86], S. 70).

6. Bereichsfehler durch einen falschen Relationsoperator.
   Solche Fehler werden zu 80% erkannt (s. [GiW 86], S. 70). Nicht oder nur mit geringer Wahrscheinlichkeit wird dabei die Vertauschung von ähnlichen Operatoren entdeckt, da dann nur bei Gleichheit der Werte ein Unterschied auftritt. Dabei gelten $<$ und $\leq$ sowie $>$ und $\geq$ als **ähnlich**.

Fehler der Art 1 werden durch Überprüfung der Programmausgabe erkannt, Fehler der Art 2, 3 und 4 durch die Betrachtung der Ausführungszahlen für die entsprechenden Zweige. Fehler der Art 5 und 6 können bei den vorliegenden Testdaten eine falsche Programmausgabe oder nur veränderte Ausführungszahlen für die entsprechenden Zweige bewirken.

Die LCMS-Überdeckung ist für die Aufdeckung der folgenden Fehler am effektivsten, was vermutlich ähnlich für die Segmentpaareüberdeckung gilt (s. [GiW 86]):

1. Alle Bereichsfehler werden zu 100% erkannt.
   Dies übertrifft die Ergebnisse der Zweigüberdeckung bei falschen Relationsoperatoren, bei falschen arithmetischen Operatoren und bei einer falschen Plazierung einer Anweisung.

[13] Es gab im Experiment nur fünf Fehler der Art „inkorrekte Konstante"; daß sie alle aufgedeckt wurden, muß an der speziellen Wahl der Testdaten liegen, die *nicht zufällig* erfolgte (s. [GiW 86], S. 65).

2. Berechnungsfehler durch falsche arithmetische Operatoren werden zu 90% aufgedeckt.

3. Berechnungsfehler durch falsche Variablendefinition bzw. Variablenreferenz werden zu 88% bzw. 78% erkannt.

Von 80 Fehlern wurden neun Fehler (acht Berechnungsfehler, ein Bereichsfehler) nur mit der LCMS-Überdeckung und nicht mit datenfluß- oder ausdrucks- und anweisungsbezogenem Testen aufgedeckt (s. [GiW 86], S. 71).

Mit der strukturierten Pfadüberdeckung kann ein Fehler gefunden werden, der bei der Zweigüberdeckung nicht entdeckt wird: In einem Programm zur Telegrammformatierung wird ein Fehler nur aufgedeckt, wenn innerhalb des ersten Schleifendurchlaufs der *then*-Zweig gewählt und anschließend die Schleife sofort verlassen wird (genaueres s. [How 78b], S. 304).

Mit der Pfadüberdeckung lassen sich theoretisch alle Berechnungs- und Bereichsfehler aufdecken, wenn gewisse Vorraussetzungen erfüllt sind. Dies drückt sich in den folgenden Sätzen aus. Dabei sei $\mathbf{P(x)}$ die Ausgabe eines Programms P mit dem Eingabewert x. $\mathbf{F(x)}$ sei die spezifizierte Ausgabe für den Eingabewert x. Für einen vollständigen Weg w von P sei $\mathbf{D_P(w)}$ der **Wegbereich** zu w, d. h. die Menge aller Eingabewerte, die den Weg w ausführen, und $\mathbf{f_w}$ die Funktion, die auf dem Weg w im Programm P berechnet wird.

SATZ 10.3.1 *(vgl. [How 76], Theorem 8)*
Mit der Pfadüberdeckung wird mindestens ein Berechnungsfehler bzgl. der Spezifikation F garantiert aufgedeckt g. d. w. das fehlerhafte Programm P einen ausführbaren Weg w hat mit: $P(x) \neq F(x)$ für jedes x aus $D_P(w)$.

Beweis: siehe Übung 10.7.

SATZ 10.3.2 *(vgl. [How 76], Theorem 9)*
Sei P' ein korrektes Programm bzgl. Spezifikation F, wobei je zwei Wege w und v in P' verschiedene Berechnungen enthalten, d. h. für die Berechnungsfunktionen f_w und f_v gilt: $f_w(x) \neq f_v(x)$ für jedes x aus dem ganzen Eingabebereich des Programms P'. Programm P enthalte keine Berechnungsfehler.
Dann gilt:
Mit der Pfadüberdeckung wird mindestens ein Bereichsfehler bzgl. der Spezifikation F garantiert aufgedeckt g. d. w. das fehlerhafte Programm P einen ausführbaren Weg w hat, so daß für den entsprechenden Weg w' im korrekten Programm P' gilt: $D_P(w)$ und $D_{P'}(w')$ sind disjunkt.

Beweis: siehe Übung 10.7.

Die praktische Anwendbarkeit der Sätze 10.3.1 und 10.3.2 ist aus folgenden Gründen eingeschränkt:

1. Pfadüberdeckung ist bei Programmen mit Schleifen nicht immer möglich, da dann beliebig viele vollständige Wege vorliegen können.

2. Es ist nicht generell entscheidbar, ob ein Weg ausführbar ist.

3. Die Definition der Berechnungs- und Bereichsfehler setzt einen Isomorphismus zwischen den Wegen des Programms P und einem korrekten Programm P' voraus. Dieser Isomorphismus liegt der Formulierung „entsprechender Weg“ in Satz 10.3.2 zugrunde.

4. Bei Satz 10.3.1 formuliert die Einschränkung „$P(x) \neq F(x)$ für jedes x aus dem Wegbereich $D_P(w)$“ das Verbot der zufälligen Korrektheit. Diese Forderung ist sehr stark, selten erfüllt und — falls zutreffend — schwer zu beweisen. Bei wenigen Ausnahmen von dieser Ungleichung spricht man von **meistens** (bzw. **fast immer**) **zuverlässigem** Testen (s. [How 76], S. 209).

5. Bei Satz 10.3.2 ist die Annahme „je zwei Wege in dem korrekten Programm enthalten verschiedene Berechnungen“ der Ausschluß der zufälligen Korrektheit, und die Bedingung „$D_P(w)$ und $D_{P'}(w')$ sind disjunkt“ schließt Unterbereichsfehler aus. Beides ist schwer zu beweisen und auch selten erfüllt.

Beim praktisch ausführbaren Pfadtesten nach den Kriterien der strukturierten Pfadüberdeckung $C_i(k)$ können daher folgende Fehler nicht oder nur unzureichend aufgedeckt werden (s. [Bei 83], S. 54 f.; [Gan 79], S. 31; [How 78b], S. 304 f.; [OmM 89], S. 64; [Sne 88], S. 307):

1. Fehlende Pfade bzw. Funktionen
   (Dies entspricht einem speziellen Fall, für den die entsprechenden Programmschritte vergessen wurden.)

2. Fehler bei der Kombination von Segmenten auf wichtigen Pfaden, die mehr als k Iterationen einer Schleife erfordern

3. Schnittstellenfehler

4. Fehlerhafte Benutzung anderer Funktionen
   (Dies wird beim *Modultest* schlecht erkannt.)

5. Initialisierungsfehler

6. Datenbankfehler

7. Unberücksichtigte Daten

8. Rundungsfehler beim Typ Real

9. Datensensitive Fehler (s. Beispiel 6.3.1)

Für das *kontrollflußbezogene* Testen läßt sich noch positiv vermerken, daß es Fehler gibt, die nur mit diesem Testansatz und nicht mit dem datenfluß- oder anweisungs- und ausdrucksbezogenen Testansatz aufgedeckt werden können. Nach Girgis/Woodward sind dies 15% aller Fehler (s. [GiW 86], S. 71).

10.3.2.2 Datenflußbezogenes Testen

Für die einzelnen Methoden des datenflußbezogenen Testens ergibt sich nach Girgis/Woodward folgendes (s. [GiW 86], S. 70 f.):
Für *Berechnungsfehler* aufgrund einer falschen Konstanten oder wegen einer falsch plazierten Anweisung ist das Kriterium alle Berechnungs-/einige Entscheidungsreferenzen am effektivsten: 88% bzw. 63% dieser Fehler werden gefunden.
Für *Bereichsfehler* aufgrund einer falschen Variablenreferenz oder einer falschen Konstanten ist das Kriterium alle Entscheidungs-/einige Berechnungsreferenzen am effektivsten: 100% dieser Fehler werden damit gefunden.
Folgende Fehler lassen sich dagegen schlecht aufdecken (s. [LaK 83], S. 353; [Nta 84a], S. 252; [GiW 86], Table 3):

- fehlende Pfade,
- Bereichsfehler durch falsch plazierte Anweisungen und falsche arithmetische Operatoren,
- Berechnungsfehler bei speziellen Werten.

Dies liegt daran, daß die Grenzwerte der Entscheidungsprädikate und spezielle Werte nicht getestet werden.
Dagegen werden Datentransformationsfehler — wie beispielsweise eine fehlende Zuweisung — gut erkannt.
Abschließend läßt sich positiv vermerken, daß es Fehler gibt, die nur mit dem datenflußbezogenen Testen gefunden werden. Nach Girgis/Woodward sind dies 9% aller Fehler (s. [GiW 86], S. 71).

10.3.2.3 Ausdrucks- und anweisungsbezogenes Testen

Für das Aufdecken von Fehlern in Ausdrücken gibt es eine Reihe von theoretischen Resultaten. Bei den Entscheidungsprädikaten, die in Entscheidungen vorkommen, wird dabei vorausgesetzt, daß das Prädikat (das implementierte und das korrekte) sich aus atomaren Bedingungen nur mit den Booleschen Operatoren *or*, *and* und *not* zusammensetzt. Eine **atomare Bedingung** ist dabei eine Boolesche Variable oder ein relationaler Ausdruck der Form „$A1\ r\ A2$", wobei $A1$ und $A2$ ein arithmetischer Ausdruck und r ein Relationsoperator ist, d. h. $<$, $\leq$, $=$, $\neq$, $>$ oder $\geq$. Ein Entscheidungsprädikat ohne relationale Ausdrücke heißt **Boolescher Ausdruck**.

Satz 10.3.3
Folgende Voraussetzungen seien gegeben:

1. *In den relationalen Ausdrücken der Entscheidungsprädikate kommt jede Variable insgesamt nur einmal vor.*
2. *Boolesche Ausdrücke enthalten jede Boolesche Variable nur einmal.*

Unter diesen Voraussetzungen gilt:

i. *Mit der minimalen Mehrfachbedingungsüberdeckung werden alle fehlerhaften Booleschen Operatoren von Booleschen Ausdrücken aufgedeckt.*

ii. *Mit der BRO-Überdeckung (Boolesche und Relationale Operatoren)*[14] *werden alle fehlerhaften Booleschen und relationalen Operatoren in Entscheidungsprädikaten aufgedeckt.*

„Aufdecken" der Fehler bedeutet hier, daß das Entscheidungsprädikat bzw. der Boolesche Ausdruck bei mindestens einem Testdatum einen falschen logischen Wert annimmt.

Beweis:
zu i.: Die minimale Mehrfachbedingungsüberdeckung fordert mindestens die Testfälle, die der Algorithmus BOR_GEN von Tai erzeugt und mit denen fehlerhafte Boolesche Operatoren aufgedeckt werden (s. Theorem 3 in [Tai 93]).
zu ii.: s. Theorem 5 in [Tai 93].

q. e. d.

Satz 10.3.3 hat leider Voraussetzungen, die praktisch verhindern, daß mit den angegebenen Kriterien 100% der genannten Fehler erkannt werden:

1. Die Entscheidung ist (meistens) nicht die letzte Anweisung im Programm. Daher kann der aufgetretene Entscheidungsfehler durch nachfolgende Berechnungen wieder ganz oder teilweise verdeckt werden.
2. In den Entscheidungsprädikaten kommt manchmal eine Variable an mehreren Stellen vor. Dann können die Werte der Teilausdrücke nicht mehr unabhängig voneinander bestimmt werden.

[14] s. Kapitel 9.6

BEISPIEL 10.3.3
Das Entscheidungsprädikat laute:

$$[(D \geq E) \textbf{ and } (X > Y)] \textbf{ or } [(D < E) \textbf{ and } (X + Z > Y)]$$

und es gelte stets $Z \geq 0$.
*Um Fehler im ersten Teilausdruck „$D \geq E$“ zu testen, muß der zweite Teilausdruck „$X > Y$“ den Wert true und „$(D < E)$ **and** $(X + Z > Y)$“ den Wert false haben. $(X > Y)$ = true impliziert aber wegen $Z \geq 0$ auch $(X + Z > Y)$ = true; daher muß $(D < E)$ = false sein. Somit kann ein Fehler in „$D \geq E$“ nicht für den Fall $(D \geq E)$ = false getestet werden.*

An diesem Beispiel wird auch deutlich, warum die Bedingungsüberdeckung zu wenig Fehler aufdeckt: Alle Tests mit dem Wert $(D \geq E) = false$ sind keine Tests des Teilausdrucks „$X > Y$“, da wegen $(D \geq E) = false$ dessen Auswertung i. allg. nicht erfolgt.

Fehlende Pfade im Programm und entsprechende fehlende Entscheidungen können mit der (minimalen) Mehrfachbedingungsüberdeckung nicht aufgedeckt werden. Das gleiche negative Ergebnis gilt für die Bereichsüberdeckung.
Die Bereichsüberdeckung kann dafür theoretisch alle Bereichsfehler aufdecken, wenn der Fehler nicht durch zufällige Korrektheit in den betroffenen Programmzweigen verdeckt wird. Das Fehlermodell bei der Bereichsüberdeckung, die **Grenzverschiebung**, ist allerdings unrealistisch:

> Aus dem Ausdruck $A > B$ kann beispielsweise durch eine kleine Grenzverschiebung der Ausdruck $A > 1,02 * B + 0,1$ bzw. $50 * A > 51 * B + 5$ werden. Solche Fehler sind aber extrem unwahrscheinlich.

Eine Orientierung an möglichen Fehlern im Ausdruck (wie in Kapitel 9.1) scheint daher sinnvoller und spart Testaufwand.

Für Ausdrücke in Berechnungen, die sich als *Polynome* in einer Variablen oder *Multinome* in mehreren Variablen schreiben lassen (vgl. Kapitel 9.1), gibt es befriedigende theoretische Resultate: Aufgrund der algebraischen Eigenschaften dieser Ausdrücke gibt es effektive Tests für die Korrektheit von Polynomen und Multinomen. Unter einem **effektiven** Test für ein Polynom wird dabei ein Test verstanden, der einen Fehler aufdeckt, der aus einem korrekten Polynom ein Polynom erzeugt, welches keinen kleineren Grad hat. Hat das zu testende, eventuell fehlerhafte Polynom einen Grad k, so gehört das korrekte Polynom also zur Klasse aller Polynome vom Grad $l \leq k$. Entsprechendes gilt für Multinome, bei denen keine Variablen entfernt worden sind. Unter diesen Voraussetzungen gilt:

SATZ 10.3.4

1. *Eine Testdatenmenge T ist ein effektiver Test für ein Polynom vom Grade k genau dann wenn T mindestens $k+1$ Testdaten enthält.*

2. *Für ein Multinom mit höchstem Exponenten k und mit l Variablen gibt es eine Testdatenmenge T mit $n \leq (k+1)^l$ Testdaten, die ein effektiver Test für das Multinom ist.*

3. *Für ein lineares Multinom mit l Variablen gibt es eine Testdatenmenge T mit $l+1$ Testdaten, die ein effektiver Test für das Multinom ist.*

Beweis: s. [How 87], Theoreme 4.5 bis 4.8.

BEISPIEL 10.3.4
$x^4 + 3x^2 + 7x + 5$ ist ein Polynom vom Grad vier. Ein effektiver Test benötigt mindestens fünf Testdaten, da $k+1$ unabhängige Werte ein Polynom vom Grad k eindeutig festlegen. $x^2y + 5xy^3 + 4xy + 7x + 6$ ist ein Multinom in den Variablen x und y mit höchstem Exponenten drei. Ein effektiver Test benötigt also höchstens $(3+1)^2 = 16$ Testdaten. $3x + y - 2z + 5$ ist ein lineares Multinom in den Variablen x, y, z, da es keine Produkte von Variablen enthält. Es gibt demnach einen effektiven Test mit vier Testdaten.

Die geforderte Zahl der Testdaten für Polynome und insbesondere Multinome ist (nach Satz 10.3.4) ziemlich groß. Die vielen Testdaten sind nur notwendig, um absolut sicher zu sein, daß keine Testdaten verwendet werden, bei denen das fehlerhafte Polynom bzw. Multinom zufällig korrekte Werte liefert. Tatsächlich haben z. B. zwei verschiedene Polynome vom Grade n nur an höchstens n Stellen gleiche Werte; zwei verschiedene Geraden — Polynome vom Grad 1 — haben nur einen Schnittpunkt. Bei einer zufälligen Auswahl eines Testdatums ist die Wahrscheinlichkeit gleich $\frac{n}{M}$, eines dieser zufällig korrekten Testdaten zu wählen, wobei M die Zahl aller möglichen Werte für x, also mindestens $2^{32}-1$, ist[15]. Ein „zufälliger" Test mit sehr kleinem Wert von $\frac{n}{M}$ heißt daher ein **statistisch effektiver Test**.
Entsprechendes gilt für Multinome:

SATZ 10.3.5
Eine Testdatenmenge T mit nur einem Testdatum ist ein statistisch effektiver Test für ein Multinom vom Grade k g. d. w. das Testdatum zufällig ausgewählt wird.

Beweis: s. [How 87], Theorem 4.8.

In der Praxis reicht also ein zusätzlicher, zufällig ausgewählter Test für alle Polynome oder Multinome, die auf einem bestimmten Weg im Programm berechnet werden.

[15] Bei Integer-Zahlen wird diese Anzahl erreicht, bei Real-Zahlen, die etwa mit 8 Byte dargestellt werden, gibt es sogar 2^{64} Werte.

Was die Kriterien der *Bedingungsüberdeckung* bzw. was das *ausdrucks- und anweisungsbezogene* Testen wirklich leistet, haben Girgis und Woodward untersucht (s. [GiW 86], S. 68 ff.). Das Datenzugriffskriterium deckt danach am effektivsten Bereichsfehler auf, die durch eine falsch plazierte Anweisung entstehen (100%), ansonsten 79% aller Bereichsfehler. Das arithmetische Relations-Kriterium deckt am effektivsten Bereichsfehler durch fehlerhafte arithmetische Operatoren, falsche konstante Werte und falsch plazierte Anweisungen auf (100%ige Aufdeckung der Fehler). Fehlerhafte relationale Operatoren werden dagegen nur zu 70% entdeckt.
Testdaten, die zum Aufdecken von Fehlern in arithmetischen Ausdrücken erzeugt werden (s. Kapitel 9.1, Kriterium *additive/multiplikative Fehler*) sind nach Girgis/Woodward relativ ineffektiv. Fehlende Berechnungen werden damit nur zu 56% aufgedeckt, falsche konstante Werte zu 13%, fehlerhafte Variablenreferenzen zu 11%, falsche relationale Operatoren zu 10%, alle anderen Fehlerarten überhaupt nicht (s. [GiW 86], S. 70).
In dem vorgestellten Experiment gibt es keine Fehler, die nur vom ausdrucks- und anweisungsbezogenen Testen aufgedeckt werden. Dieses Ergebnis hängt vermutlich damit zusammen, daß bei den *anderen* Methoden ein umfangreicher Bericht erstellt wird und das Programm jeweils gründlich inspiziert wird.

Für die Kriterien der Datenüberdeckung gibt es nur Aussagen von ihrem „Erfinder" Sneed. Danach ist Datenüberdeckung besonders effektiv für das Aufdecken von Auslassungsfehlern, Berechnungsfehlern und falschen Grenzwerten (s. [Sne 86]).

10.3.3 Zusammenfassung

Zum Abschluß dieses Teilkapitels wird noch einmal die Frage gestellt, welche Fehler sich mit dem struktur- bzw. implementationsorientierten Testen generell aufdecken lassen. Pessimistische Zahlen liegen von Howden vor: nur 19% der Fehler lassen sich finden (s. [How 80], S. 167). Mittlere Werte werden von Myers und Howden berichtet: 55% aller Fehler werden bei einer Kombination mit dem funktionsorientierten Testen gefunden (s. [Mye 78]), 64% durch Pfadtesten (s. [How 78b], S. 304). Die besten Werte ergeben sich aus der Untersuchung von Girgis/Woodward. Wird für jede Fehlerart die erfolgreichste Methode gewählt, so bleiben höchstens 9 von 80 Fehlern unentdeckt; 89% aller Fehler werden also aufgedeckt (s. [GiW 86], Table 1, Table 4).
Einigkeit herrscht allerdings darüber, welche Fehler nicht (oder jedenfalls schlecht) mit dem implementationsorientierten Testen aufgedeckt werden: Die Ursache für nicht erkannte Fehler liegt meist darin, daß beim reinen Pfadtesten *keine verschiedenen* Werte für denselben Pfad getestet werden. Durch Anforderungen des ausdrucks- und anweisungsbezogenen und datenbezogenen Testens werden dagegen mehrere Werte pro Pfad erforderlich.
Dennoch können folgende Fehler unerkannt bleiben: Abweichungen von der Spezifikation selbst, sowie Fehler bei extremen oder speziellen Werten, die für eine Funktion wichtig sind.

10.4 Übungen

Übung 10.1:

(a) Beweisen Sie das Enthaltensein der folgenden Kriterien aus Abbildung 10.1:
alle k-iterativen Wege $\rightarrow$ *$C_i(k)$-Überdeckung* $\rightarrow$ (für $k > 2$) $\rightarrow$ *starke C_{GI}-Überdeckung* $\rightarrow$ *C_{GI}-Überdeckung* $\rightarrow$ *schwache C_{GI}-Überdeckung;*
alle 1-iterativen Wege $\rightarrow$ *alle einfachen Wege* $\rightarrow$ *schwache C_{GI}-Überdeckung.*
Warum gilt dann auch: *alle 2-iterativen Wege* $\rightarrow$ *starke C_{GI}-Überdeckung*?

(b) Warum gilt *C_{GI}-Überdeckung* $\rightarrow$ *Zweigüberdeckung* und *schwache C_{GI}-Überdeckung* $\rightarrow$ *Anweisungsüberdeckung*, aber nicht *schwache C_{GI}-Überdeckung* $\rightarrow$ *Zweigüberdeckung*?

Übung 10.2:

Beweisen Sie das Enthaltensein der folgenden Kriterien aus Abbildung 10.2:
$C_i(k)$-Überdeckung $\rightarrow$ (für passendes k) $\rightarrow$ *alle m-DR-Interaktionen* $\rightarrow$ (für $m > 2$) $\rightarrow$ *alle Referenzen* $\rightarrow$ *alle E-/einige B-Referenzen* $\rightarrow$ *Zweigüberdeckung; $C_i(k)$* $\rightarrow$ (für passendes k) $\rightarrow$ *geordnete Kontextüberdeckung.*

Übung 10.3:

Beweisen Sie, daß die Bereichsüberdeckung die minimale Mehrfachbedingungsüberdeckung enthält. Dabei muß natürlich vorausgesetzt werden, daß die atomaren Prädikatterme *lineare* relationale arithmetische Ausdrücke sind, also z. B. $A+3*C \geq 4*B-D$. Nehmen Sie der Einfachheit halber an, daß die atomaren Prädikatterme nur mit dem logischen Operator *and* verknüpft sind.

Übung 10.4:

Führen Sie folgende Beweise durch:

(a) Beweisen Sie, daß das Kriterium *alle kürzeren Ausdrücke* unter der Bedingung (**) in Abbildung 10.4 das Datenzugriffskriterium enthält.

(b) Geben Sie Gegenbeispiele an, die zeigen, daß die in Abbildung 10.4 *nicht* durch Pfeile verbundenen Kriterien tatsächlich unvergleichbar sind. Überlegen Sie sich dazu vorab, welche Unvergleichbarkeit eine andere Unvergleichbarkeit impliziert (da umgekehrt aus $A \rightarrow B$ und $B \rightarrow C$ logisch $A \rightarrow C$ folgt).

Übung 10.5:

Zeigen Sie, daß bei Beispiel 10.2.2 (liegt ein Punkt in einem Quader?) für die Mehrfachbedingungsüberdeckung 27 Testfälle und für die Bereichsüberdeckung 18 oder 12 Testdaten ausreichend sind — je nachdem, ob klar ist, daß die Spezifikation des Quaders die Relationen „=“ und „$\neq$“ enthält oder nicht enthält. (Im ersten Fall ist der Quader als Hohlkörper spezifiziert.)

Übung 10.6:
Betrachten Sie anstelle des Quaders aus Beispiel 10.2.2 ein Rechteck mit den Grenzen (x_1, x_2), (y_1, y_2) und die entsprechende Abfrage, ob ein Punkt (x, y) in dem Rechteck liegt.
Überlegen Sie sich, welche Fehler (d. h. Abweichungen von dem gegebenen Rechteck) bei den folgenden Testverfahren nicht erkannt werden:

(a) Bedingungstest,

(b) minimale Mehrfachbedingungsüberdeckung (ohne bzw. mit Grenzwerten),

(c) Mehrfachbedingungsüberdeckung (ohne bzw. mit Grenzwerten),

(d) Bereichsüberdeckung.

Übung 10.7:
Bei einem Kontrollflußgraphen mit k aufeinanderfolgenden Verzweigungen gibt es 2^k verschiedene Wege, die bei der Pfadüberdeckung ausgeführt werden müssen (s. Abbildung 7.6 auf S. 200).

(a) Zeigen Sie, daß vier Wege ausreichen, um in diesem Falle alle Segmentpaare zu überdecken (s. Def. 7.2.6).

(b) Wie viele Wege werden benötigt, um die $C_S(3)$-Überdeckung zu erfüllen, d. h. alle Segmentripel (drei direkt aufeinanderfolgende Segmente) zu überdecken?

Übung 10.8:
Beweisen Sie die Sätze 10.3.1 und 10.3.2.

10.5 Verwendete Quellen und weiterführende Literatur

Da in diesem Kapitel viele „kleine“ Informationen aus diversen Quellen zusammengestellt wurden, sind die Quellenangaben an den entsprechenden Stellen gemacht worden.

11 Testdatenerzeugung und Testwirksamkeitsmessung

In den Kapiteln 7 bis 10 wurden eine Reihe von Kriterien vorgestellt, die Anforderungen an eine Testdatenmenge stellen. Beispielsweise stellt das Kriterium *Zweig-* bzw. *Entscheidungsüberdeckung* die Anforderung, daß die Testdatenmenge für jeden Entscheidungsausgang ein Testdatum enthält, das diesen Entscheidungsausgang ausführt. Für eine irgendwie erstellte Testdatenmenge muß dieses Kriterium nicht zu 100% erfüllt sein. Bei Definition eines geeigneten Testwirksamkeitsmaßes kann aber angegeben werden, zu wieviel Prozent das Kriterium erfüllt ist. Für die Anweisungsüberdeckung und Zweigüberdeckung wurden diese Maße entsprechend definiert (s. Definition von TWM_0 und TWM_1 in Kapitel 7 auf Seite 196 und 198). Für die anderen kontrollflußbezogenen, datenflußbezogenen, anweisungs-, ausdrucks- und datenbezogenen Testkriterien lassen sich in naheliegender Weise entsprechende Testwirksamkeitsmaße aufstellen. Dabei werden jeweils die überdeckten bzw. ausgeführten Konstrukte der Gesamtzahl aller Konstrukte der betrachteten Art gegenübergestellt. Bei der Entscheidungsüberdeckung werden beispielsweise als Konstrukte gerade alle Entscheidungsausgänge betrachtet.

In Kapitel 11.1 werden nun verschiedene Methoden vorgestellt, mit denen die Testwirksamkeit einer gegebenen Testdatenmenge bei der Ausführung des Programms mit den entsprechenden Testdaten gemessen werden kann. Falls die Testwirksamkeit nicht 100% beträgt, sind weitere Testdaten zu bestimmen, um das gewählte Testkriterium zu erfüllen. In Kapitel 11.2 werden daher Methoden für die Testdatenerzeugung vorgestellt, mit denen eine gegebene Testwirksamkeit erhöht werden kann.

11.1 Messung der Testwirksamkeit durch Instrumentierung

Die Testkriterien erfordern die Ausführung bestimmter Konstrukte oder die Ausführung von Kombinationen von Konstrukten. Daher kann die Testwirksamkeit einer Testdatenmenge dadurch gemessen werden, daß Meßpunkte in die entsprechenden Konstrukte eingebaut werden, welche die Ausführung dieser Konstrukte protokollieren.

11.1.1 Zweigüberdeckung

Um festzustellen, ob die Zweigüberdeckung erreicht ist bzw. welcher Prozentsatz der Entscheidungsausgänge überdeckt ist, genügt es festzustellen, welche Entscheidungs-Entscheidungs-Wege ausgeführt wurden (s. Satz 7.2.1, Teil D, Seite 198). Also muß nur in jeden Entscheidungs-Entscheidungs-Weg ein Meßpunkt eingebaut werden. Dies kann ein Zähler sein, der erhöht wird oder ein entsprechender Prozeduraufruf. Diese Veränderung des Programms wird **Instrumentierung** genannt.

Beispiel 11.1.1
Bei dem Programm aus Abbildung 7.1 auf S. 190 sind die Entscheidungs-Entscheidungs-Wege E1 bis E5 mit Meßpunkten zu versorgen (siehe Tabelle 11.1, vgl. Beispiel 7.1.1).

Meßpunkt in Kante	für Entscheidungs-Entscheidungs-Weg bzw.	Kantenfolge
a oder b	E1	(a,b)
h	E2	(h)
c	E3	(c)
d oder e	E4	(d,e,g)
f	E5	(f,g)

Tab. 11.1 Versorgung des Programms aus Abb. 7.1 mit Meßpunkten

Bei einer Schachtelung von Schleifen im Programm reicht es, die Überdeckung von Entscheidungs-Entscheidungs-Wegen in den innersten Schleifen zu überprüfen, um die hundertprozentige C_1-Überdeckung festzustellen.

Beispiel 11.1.2
Innerhalb der b-Schleife (b_1, b_2) existiert hier eine weitere Schleife (c_1, c_2). Es gilt: Wenn das Programm terminiert und (c_1, c_2) ausgeführt wird, dann werden alle Kan-

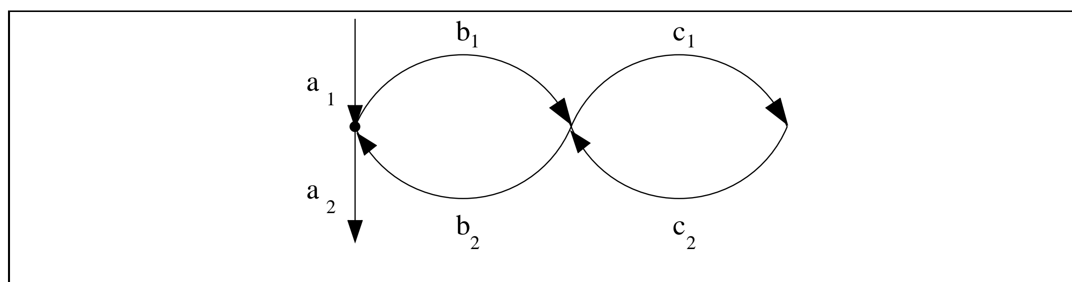


Abb. 11.1: Mehrfach geschachtelte Schleifen

ten durchlaufen. Also reicht für die Zweigüberdeckung die Ausführung von (c_1, c_2) aus, was mit einem Meßpunkt bei c_1 festgestellt werden kann.

Die Analyse der Schachtelung von Entscheidungen kann zu weiteren Einsparungen führen. Im Beispiel 11.1.1 macht die Überdeckung der „inneren“ Entscheidungs-Entscheidungs-Wege $E4$ (d, e, g) und $E5$ (f, g) die Prüfung der Überdeckung von $E1, E2$ und $E3$ überflüssig; also reichen Meßpunkte bei d und f.

Die mögliche Einsparung von Meßpunkten läßt sich genauer charakterisieren in Abhängigkeit von der Zahl n der Knoten und der Zahl e der Kanten im Kontrollflußgraphen.

Satz 11.1.1
Ein wohlgeformter[1] Kontrollflußgraph habe n Knoten und e Kanten.
Die Zahl t der Testdaten bzw. Testläufe sei bekannt und bei jedem Testlauf terminiere das Programm. Dann ist $e - n + 1$ die notwendige und hinreichende Zahl von Meßpunkten, um die Ausführungsanzahlen für alle Anweisungen und Zweige des Programms zu bestimmen.

Beweisidee:
Die Zahl der **charakteristischen** Kanten, die man zu einem Spannbaum des Graphen hinzufügen muß, um alle Kanten zu erhalten, ist $e-(n-1)$, da der Spannbaum alle n Knoten und somit $n - 1$ Kanten enthält. Auf jeder solchen charakteristischen Kante ist gerade ein Meßpunkt einzufügen, da die zugehörigen fundamentalen Wege unabhängig voneinander durchlaufen werden können (vgl. Abschnitt 7.2.3, Seite 201). Der Meßpunkt auf der charakteristischen Kante vom End- zum Anfangsknoten kann (und muß) entfallen, da die Anzahl t der Testläufe bekannt ist. Also ist mit e (= Zahl der Kanten im Kontrollflußgraphen ohne die Kante vom End- zum Anfangsknoten) und nicht mit $e + 1$ zu rechnen.

q. e. d.

Beispiel 11.1.3
Für den Kontrollflußgraphen von Abbildung 7.1 bilden die Kanten a, b, c, d, f, h einen Spannbaum, der jeden Knoten des Graphen enthält. Es fehlen die charakteristischen Kanten e und g, die je einen Meßpunkt erhalten. (Die Ausführung des fundamentalen Weges abh umgeht die Meßpunkte, zählt aber bei der Anzahl t mit.)

Beispiel 11.1.4
Für den Kontrollflußgraphen von Abbildung 11.1 bilden die Kanten a_1, b_1, c_1, a_2 einen Spannbaum. Es fehlen die charakteristischen Kanten c_2 und b_2, die je einen Meßpunkt erhalten. Im Unterschied zu den Forderungen für die Feststellung einer hundertprozentigen Entscheidungsüberdeckung ist der Meßpunkt bei b_2 ebenfalls notwendig. Falls die (c_1, c_2)-Schleife nicht ausgeführt wird, ist durch den Meßpunkt bei b_2 eine Aussage darüber möglich, ob wenigstens die (b_1, b_2)-Schleife ausgeführt wurde.

[1] s. S. 192

Die Minimierung der Zahl der Meßpunkte bewirkt, daß nur für einen Teil der Segmente direkt (durch Messung) bestimmt wird, wie oft sie bei einem Testlauf ausgeführt werden. Diese Ausführungszahl $z(S)$ muß dagegen für die nicht mit Meßpunkten versehenen Segmente S aus den Ausführungszahlen der anderen Segmente mit der **Kirchhoff'schen Regel**[2] berechnet werden. Dieser Berechnungsaufwand ist *linear* in der Zahl der Kanten im Kontrollflußgraphen: Wegen Satz 11.1.1 gibt es einen Knoten im Kontrollflußgraphen mit m ein- oder ausgehenden Kanten, von denen $m-1$ charakteristische Kanten sind, also mit einem Meßpunkt versehen sind. Die Ausführungszahl des nicht instrumentierten Segments läßt sich damit berechnen. Am anderen Ende dieses Segments liegt ein Knoten, auf den dieses Verfahren wiederum anwendbar ist, usw. Dabei wird jede Kante höchstens zweimal betrachtet.

Beispiel 11.1.5
Für den Kontrollflußgraphen aus Abbildung 7.1 mit den instrumentierten Kanten e und g ist F der Knoten mit drei Kanten e, f, g, von denen zwei instrumentiert sind. Also gilt nach der Kirchhoff'schen Regel:

$$z(f) = z(g) - z(e).$$

Für den Knoten D am anderen Ende von f erhält man daher:

$$z(c) = z(d) + z(f) = z(e) + z(f) = z(e) + z(g) - z(e) = z(g)$$

Für Knoten C und die Segmente ab, c, g, h gilt dann:

$$z(h) = z(ab) + z(g) - z(c) = t + z(g) - z(g) = t,$$

wobei t die Anzahl der Testläufe ist.

Beispiel 11.1.6
Für den Kontrollflußgraphen aus Abbildung 11.1 mit den instrumentierten Kanten b_2 und c_2 erhält man für die Segmente b_1, c_1c_2 und b_2:

$$z(b_1) + z(c_1c_2) = z(b_2) + z(c_1c_2), \text{ d. h. } z(b_1) = z(b_2).$$

Für die Segmente a_1, a_2, b_1, b_2 erhält man:

$$z(a_2) = z(a_1) + z(b_2) - z(b_1) = z(a_1) = t = \text{Zahl der Testläufe.}$$

Wie zu sehen ist, sind die nicht gemessenen Ausführungszahlen relativ einfach aus den gemessenen Zahlen zu berechnen. Das gleiche gilt natürlich, wenn nur von Interesse ist, ob die entsprechenden Zweige ausgeführt wurden oder nicht. Dann ist nur festzustellen, ob die Ausführungszahlen ungleich oder gleich Null sind.

[2] Die Summe der Ausführungszahlen auf den eingehenden Kanten ist gleich der Summe der Ausführungszahlen auf den ausgehenden Kanten eines Knotens.

Der Berechnungsmehraufwand *nach* Ausführung der Testdaten ist also gering. Dafür wird die Ausführung des Programms nicht so stark verlangsamt wie bei einer nicht minimalen Verwendung von Meßpunkten. Für Dialogprogramme oder gar Echtzeitprogramme ist dies ein wichtiger Gesichtspunkt.
Wie groß ist nun die Einsparung an Meßpunkten?
Wenn nur Entscheidungen mit zwei Ausgängen im Programm vorkommen, würde eine einfache Instrumentierung bei p Entscheidungen $2 * p$ Meßpunkte einbauen, nämlich in jeden Entscheidungsausgang einen. Bei der optimalen Strategie nach Satz 11.1.1 werden nur $e - n + 1$ Meßpunkte gebraucht. Bei einem Kontrollflußgraphen mit p Zweiwegeverzweigungen gilt aber $e - n + 1 = p$ (vgl. Kapitel 16.1, S. 440, zyklomatische Zahl $v(G) - 1$). Also werden statt $2 * p$ Meßpunkten nur p gebraucht.

11.1.2 Bedingungsüberdeckung

Für die Messung der atomaren Bedingungsüberdeckung reicht ein Instrumentieren von Zweigen nicht aus, vielmehr muß man protokollieren, ob die atomaren Bedingungen die Werte *true (T)* oder *false (F)* angenommen haben.

Beispiel 11.1.7
*Die Abfrage „***if** *A = 2* **or** *X > 1* **then** ... *" muß ersetzt werden durch:*

> **if** *A = 2* **then** *Merke(B1, T)* **else** *Merke(B1, F);*
> **if** *X > 1* **then** *Merke(B2, T)* **else** *Merke(B2, F);*
> **if** *A = 2* **or** *X > 1* **then** ...

Dabei protokolliert „Merke(Bi, w)", daß Bedingung i den Wert w angenommen hat.

Die in Beispiel 11.1.7 verwendete Technik der vorgezogenen Auswertung und Protokollierung der atomaren Bedingungen setzt voraus, daß die Auswertung der Bedingungen keine Seiteneffekte hat. Falls das dennoch der Fall ist, könnte man vermuten, daß eine andere Technik möglich ist, bei der jede atomare Bedingung B_i durch einen Aufruf *INS*(B_i, *i*) ersetzt wird. Im Beispiel 11.1.7 wäre die Abfrage zu ersetzen durch:

> **if** *INS(A = 2, 1)* **or** *INS(X > 1, 2)* **then** ...

Dabei ist *INS* eine boolesche Funktionsprozedur, die als Ergebnis den logischen Wert des ersten Parameters abliefert. Als Seiteneffekt protokolliert *INS* in einem Array an *der* Stelle, die durch den zweiten Parameter angegeben wird, ob dieser logische Wert wahr bzw. falsch ist.
Diese Technik versagt jedoch bei der üblichen optimierten Entscheidungsauswertung durch den Übersetzer. Bei *or*-Verknüpfungen bricht die Auswertung nach der ersten wahren Bedingung ab, bei *and*-Verknüpfungen nach der ersten falschen Bedingung.

Die weiteren Bedingungswerte werden also nicht protokolliert.
Die Bedingungsüberdeckung kann somit nur gemessen werden, wenn keine Seiteneffekte vorliegen oder (als Compileroption neuerdings oft möglich) eine vollständige Bedingungsauswertung erfolgt (was bei effizienter Programmierung aber oft nicht anwendbar sein kann).

Bei der normalen oder der minimalen Mehrfachbedingungsüberdeckung ist nach dem Festhalten der Werte der einzelnen Bedingungen zusätzlich bei jeder Entscheidung festzuhalten, welche Kombination der Bedingungswerte vorliegt. Im Falle von Seiteneffekten und einer optimierten Entscheidungsauswertung kann nur die folgende **Variante der (minimalen) Mehrfachbedingungsüberdeckung** gemessen werden. Bei einem n-stelligen $and(B_1, \ldots, B_n)$ sind dies folgende Fälle:

1. alle (atomaren) Bedingungen B_1 bis B_n gleich *true*,
2. für jedes i mit $1 \leq i \leq n$:
   (a) alle Bedingungen B_1 bis B_{i-1} gleich *true* (für $i = 1$ also keine Bedingung *true*),
   (b) Bedingung $B_i = false$,
   (c) Bedingungen B_{i+1} bis B_n werden nicht ausgewertet.

Bei einem n-stelligen *or* sind entsprechende Fälle (mit *true* und *false* vertauscht) zu messen.

Beispiel 11.1.8
Für die Abfrage aus Beispiel 11.1.7 ist für $B_1 = (A = 2)$, $B_2 = (X > 1)$ folgendes zu messen, da B_1 und B_2 mit or verknüpft sind:

1. $B_1 = false$, $B_2 = false$, *also* $A \neq 2$, $X \leq 1$,
2. $B_1 = false$, $B_2 = true$, *also* $A \neq 2$, $X > 1$,
   $B_1 = true$, B_2 *nicht ausgewertet, also* $A = 2$, X *beliebig.*

11.1.3 Ausdrucks- und datenbezogenes Testen

Beim ausdrucksbezogenen Testen muß jeder Ausdruck im Programm instrumentiert werden. Die Feststellung, ob ein bestimmtes Kriterium erfüllt ist, kann auf zwei Arten getroffen werden:

1. Wenn man pro Ausdruck die Folge aller Werte protokolliert, kann eine beliebige Auswertung nach Beendigung des Testlaufs angestoßen werden.
2. Die Kriterienüberprüfung erfolgt bei jeder Ausdrucksauswertung.

Beispiel 11.1.9
Beim Testen von arithmetischen Relationen der Form „A1 r A2“ (mit arithmetischen Ausdrücken A1 und A2 und Relationssymbol r) wird sofort protokolliert, ob die Differenz $A1 - A2$ den größten negativen Wert (knapp unter 0), den Wert 0 oder den kleinsten positiven Wert angenommen hat (vgl. S. 233).

Beim datenbezogenen Testen sind die E/A-Operationen zu instrumentieren. Bei jeder E/A-Operation ist festzustellen, welche Moduldaten ihre Werte geändert haben und ob die Änderung durch Eingabe oder Ausgabe verursacht wurde. Da die Zuordnung der Daten zu Feldern und Datenkapseln vorab bestimmbar ist, kann die Überdeckung von Datenkapseln, Feldern und repräsentativen Werten auf diese Art festgestellt werden (vgl. Kap. 9.4).

11.1.4 Segmentpaareüberdeckung, Schleifenüberdeckung und datenflußbezogene Kriterien

Mit den bisherigen Meßmethoden kann nur das Erfülltsein von Kriterien bestimmt werden, die sich auf einzelne punktuelle Konstrukte beziehen. Die Kriterien, die sich auf die Ausführung von Kontrollfluß- oder Datenfluß*wegen* beziehen, erfordern andere Meßmethoden. Dabei sind wieder zwei Fälle zu unterscheiden.

1. Es müssen bestimmte Wege im Kontrollflußgraphen ausgeführt werden: Dies ist nur festzustellen, wenn jeder Entscheidungs-Entscheidungs-Weg einen Meßpunkt enthält und die Folge der durchlaufenen Entscheidungs-Entscheidungs-Wege pro Testdatum (etwa in einer Datei) festgehalten wird. Allenfalls kann diese Folge begrenzt werden, indem bei Schleifen die Protokollierung nach k-maligem Durchlauf abgebrochen wird (für ein vorgegebenes k mit $k > 0$).

   Dieses Vorgehen wird z. B. bei folgenden Kriterien bzw. Konstrukten benötigt: Segmentpaareüberdeckung, $C_S(n)$-Überdeckung , Grenze-Inneres-Überdeckung, strukturierte Pfadüberdeckung $C_i(k)$, alle DR-Wege.
   Bei Segmentpaaren kann man allerdings eine Vereinfachung vornehmen. Jedes durchlaufene Segment[3] wird in einem Zwischenspeicher festgehalten. Beim Durchlaufen des nächsten Segments wird das Paar (gespeichertes Segment, aktuelles Segment) endgültig abgespeichert oder in einer vorbereiteten Tabelle aller Segmentpaare „angekreuzt“.

2. Es müssen nur Wege mit einer bestimmten Eigenschaft ausgeführt werden: Dies trifft bei allen anderen Datenflußkriterien zu.
   In diesem Fall sind vorab für jeden Knoten k des Kontrollflußgraphen die Mengen der Variablen zu bestimmen, die in k definiert, referenziert und undefiniert

[3] „Segment“ steht hier als Synonym für „Entscheidungs-Entscheidungs-Weg“ (vgl. Fußnote 9 auf Seite 200)

werden (*DEF(k), REF(k)* und *UNDEF(k)*). Je nach Kriterium ist außerdem eine Tabelle für *alle Definitionen*, *alle Referenzen*, *alle E-/einige B-Referenzen*, *alle B-/einige E-Referenzen* oder *alle k-DR-Interaktionen* aufzustellen. Sie enthält die jeweils geforderten Paare (oder Folgen von Paaren) von Definitionen und Referenzen von Variablen.
Bei der Ausführung der Tests muß eine Liste der aktuellen Definitionen geführt werden, wobei jede Definition durch das Paar (Variable, Knoten) beschrieben wird. Wenn ein neuer Knoten l ausgeführt wird, sind folgende Aktionen erforderlich (durch eine entsprechende Instrumentierung des Knotens):

(a) Eintrag in der Tabelle aller DR-Interaktionen für eine Definition und Referenz von x, wenn x in der Liste der aktuellen Definitionen und in *REF(l)* vorkommt.
Bei Referenzen in Entscheidungen ist dabei ggf. der folgende Entscheidungsausgang abzuwarten und mitzuprotokollieren.

(b) Falls x in *DEF(l)* oder *UNDEF(l)* vorkommt, ist die Liste der aktuellen Definitionen anschließend entsprechend zu ändern.

Bei dem Kriterium alle k-DR-Interaktionen für $k > 2$ sind die bisher ermittelten m-DR-Interaktionen mit $m < k$ jeweils in einer aktuellen Liste mitzuführen. Der Speicheraufwand ist also beträchtlich.

Bei der ungeordneten oder geordneten Kontextüberdeckung ist vorab die entsprechende Tabelle der Definitionskontexte zu erzeugen. Während der Ausführung eines Knotens ist bei der referenzierten Variablen festzustellen, welcher Definitionskontext diese Referenz erreicht hat. Je nach Anforderung sind die entsprechenden Definitionen als Menge zu beschreiben — falls es um ungeordnete Definitionskontexte geht — oder als Folge von Definitionen, falls es um geordnete Definitionskontexte geht.

11.1.5 Allgemeine Aspekte der Messung der Testwirksamkeit

Natürlich reicht es nicht aus, nur für *einen* Testlauf (d. h. für ein Testdatum) die Überdeckung der geforderten Konstrukte zu messen. Vielmehr müssen die Ergebnisse für eine Testdatenmenge oder auch für mehrere Testdatenmengen *akkumuliert* werden. Dies setzt eine entsprechende Speicherung und Fortschreibung der Ergebnisse in einer Datei oder Datenbank voraus.
Die Ausgabe der Testergebnisse in einem *Testprotokoll* ist ebenfalls von entscheidender Bedeutung. Für das Qualitätsmanagement ist dabei meistens nur von Bedeutung, zu wieviel Prozent ein Kriterium erfüllt wurde. Für die Planung eines Tests ist dagegen eine detaillierte, aber lesbare Information nötig, die angibt, welche Konstrukte durch die bisherigen Tests noch nicht überdeckt bzw. ausgeführt wurden. Die Menge und Art der anzugebenden Meßergebnisse sollte also per Parameter (in der Ausgabekomponente) einstellbar sein.

11.2 Testdatenerzeugung

Wenn gemessen wurde, daß die bisherigen Testdaten keine hundertprozentige Überdeckung der geforderten Konstrukte bewirken, muß versucht werden, die nicht überdeckten Konstrukte durch weitere Testdaten zu erreichen. Wie können entsprechende Testdaten erzeugt werden?

11.2.1 Lösungsansatz für die Testdatenerzeugung

Ein Lösungsansatz besteht darin, das Problem iterativ zu lösen. Die Problemstellung für einen Iterationsschritt ist die folgende:

1. Gegeben ist ein Programm und eine Testatenmenge T, die keine hundertprozentige Überdeckung erreicht. Damit ist implizit die Menge der durch T überdeckten Konstrukte (Anweisungen, Zweige, Wege, Prädikatterme usw.) und die Menge der nicht überdeckten Konstrukte gegeben.

2. Es ist ein nicht überdecktes Konstrukt auszuwählen und ein Testdatum zu bestimmen, das dieses Konstrukt ausführt.
   Die *Auswahl* eines solchen Konstrukts kann im Prinzip zufällig erfolgen. Bei einem deterministischen Vorgehen bietet sich allerdings bei der Zweigüberdeckung z. B. an, die Wege zu betrachten, die durch die bisherigen Testdaten ausgeführt werden. Von diesen Wegen ist ein kürzestes Anfangsstück bis zu einer Verzweigung zu bestimmen, für die ein ausgehender Entscheidungszweig noch nicht überdeckt wurde. Dieser Zweig ist als nächstes zu überdecken. Falls das geschieht, ist zu erwarten, daß auf dem neuen Weg noch viele andere Zweige liegen, die bisher nicht überdeckt wurden. Diese Strategie — genannt **Pfadpräfix-Strategie** — ist also i. allg. sinnvoll und muß für andere Konstrukte entsprechend abgewandelt werden.

Als Problem bleibt also „nur“, zu einem Konstrukt ein Testdatum zu finden, welches das Konstrukt ausführt.
Dafür kann folgender *Lösungsansatz* gewählt werden:

1. Es ist ein Weg im Kontrollflußgraphen vom Anfang bis zu dem betrachteten Konstrukt zu finden, der eine möglichst kurze Verlängerung bis zum Programmende hat.

2. Es ist zu bestimmen, welche Bedingungen die Eingabevariablen des Programms erfüllen müssen, damit der in Schritt 1 gewählte Weg auch ausgeführt wird.

Die Bestimmung eines Weges zu einem Konstrukt ist ein relativ einfaches graphentheoretisches Problem. Nur wenn ein Konstrukt z. B. ein Paar von Zweigen mit zusätzlichen Bedingungen ist, kann es einen hohen Berechnungsaufwand geben.
Oft gibt es mehrere Wege zu einem Konstrukt; insbesondere wenn auf einem passenden Weg eine Schleife liegt, können verschieden viele Schleifendurchläufe gewählt werden. Man sollte daher zuerst die kürzesten Wege bestimmen.

BEISPIEL 11.2.1 *(s. Abbildung 11.1)*
Segment c_1c_2 *wird durch den kürzesten Weg* $a_1b_1c_1$ *erreicht, aber auch durch* $a_1b_1b_2b_1c_1, a_1b_1b_2b_1b_2b_1c_1$, *allgemein durch einen Weg* $a_1((b_1b_2)^k)b_1c_1$ *mit* $k = 0, 1, 2, \ldots$

Die Bedingungen für die Eingabevariablen ergeben sich aus den zu erfüllenden Prädikaten in den Entscheidungsknoten auf dem Wege und wird daher **Wegebedingung** genannt. Die Prädikatbedingungen sind aber durch „symbolische Ausführung[4] rückwärts“ umzurechnen in Bedingungen für die Eingangsvariablen.

BEISPIEL 11.2.2
Bei dem Kontrollflußgraphen aus Abbildung 7.1 reicht bei geforderter C_1*-Überdeckung nach Satz 11.1.1 die Überdeckung der Entscheidungs-Entscheidungs-Wege* (d, e, g) *und* (f, g)*. Kürzeste Wege zum Erreichen dieser Entscheidungs-Entscheidungs-Wege sind:*

- *für (d,e,g) der Weg abcdeg,*
- *für (f,g) der Weg abcfg.*

1. *Für den Weg abcdeg ergibt sich als Bedingung:*

   (für d) y *mod* $2 = 1$ *und (für c)* $y \neq 0$.

   Jede ungerade Zahl erfüllt diese Bedingung[5].
   Eine möglichst kurze Verlängerung des Weges bis zum Programmende ist in diesem Fall der Weg abcdegh, mit der folgenden Bedingung:

   $y = 0$ *(bei h) und* y *mod* $2 = 1$ *(bei d) und* $y \neq 0$ *(bei c).*

   Die erste Teilbedingung muß noch rückwärts gerechnet werden. Da in Anweisung F der Wert von y *gemäß* $y := y$ *div* 2 *verändert wird, ist für den Anfangswert von* y *statt „*$y = 0$*“ zu fordern:*

   „y *div* $2 = 0$“.

[4] genaueres siehe Kapitel 12.3
[5] Wenn man y *mod* 2 auch für negative ganze Zahlen definiert, und zwar folgendermaßen: y *mod* $2 := (-y)$ *mod* 2 falls $y < 0$.

Also ergibt sich als Gesamtbedingung für den Weg abcdegh:

$(y\ div\ 2 = 0)$ *und* $(y\ mod\ 2 = 1)$ *und* $(y \neq 0)$.

Dazu gehört folgende Lösungsmenge L für die Eingabewerte von y, wobei Z die Menge der ganzen Zahlen ist[6]*:*

$$L = \{-1, 0, 1\} \cap \{y \in Z | y \text{ ungerade}\} \cap \{y \in Z | y \neq 0\} = \{-1, 1\}.$$

Ein Testdatum besteht also aus einem Tupel (x, y) *mit* $y = 1$ *oder* $y = -1$ *und beliebigem* x.

2. *Für den Weg abcfg ergibt sich als Bedingung:*

   (für f) $y\ mod\ 2 \neq 1$ *und (für c)* $y \neq 0$.

   Jede gerade Zahl (außer 0) erfüllt diese Bedingung.
   Für den kürzesten vollständigen Weg abcfgh ergibt sich:

   $y = 0$ *(bei h) und* $y\ mod\ 2 \neq 1$ *(bei f) und* $y \neq 0$ *(bei c).*

   Die erste Teilbedingung muß wieder rückwärts gerechnet werden. Damit erhält man als Gesamtbedingung für den Weg abcfgh:

   $(y\ div\ 2 = 0)$ *und* $(y\ mod\ 2 = 0)$ *und* $(y \neq 0)$.

   Dieses System hat keine Lösung, da die ersten beiden Bedingungen nur die Lösung $y = 0$ *haben, was der dritten Bedingung widerspricht. Der Weg abcfgh ist also nicht ausführbar. Wird dagegen ein Test mit einer positiven geraden Zahl (also etwa* $y = 2$*) gewählt, wird — als Verlängerung von abcfg — der Weg abcfgcdegh ausgeführt.*

11.2.2 Probleme bei der Testdatenerzeugung

Wie in Beispiel 11.2.2 (Teil 2) zu sehen ist, kann folgendes Problem bei der Ausrechnung der Eingabebedingungen auftreten:
Wenn ein Weg unausführbar ist, kann die ausgerechnete Wegebedingung nicht erfüllt werden. Dann ist ein längerer Weg auszuwählen. Es bleibt die Frage, ob man der Wegebedingung die Erfüllbarkeit ansehen kann. Bei Wegen mit einfachen Entscheidungen ist dies der Fall. Es gilt folgendes:

Satz 11.2.1
Das Ausführbarkeitsproblem für einen Kontrollflußweg ist entscheidbar, wenn die Wegebedingung linear in den Eingabevariablen bzw. -werten ist.

[6] Dabei wird angenommen, daß für negative Zahlen y gilt: $y\ div\ 2 = -(|y|\ div\ 2)$. Also z. B. $(-1)\ div\ 2 = -(1\ div\ 2) = -0 = 0$.

Beweisidee:
Mit den Techniken der linearen Programmierung kann eine Lösung gefunden werden bzw. gezeigt werden, daß keine Lösung existiert (s. [Zim 90]).

q. e. d.

Allgemein gilt jedoch:

SATZ 11.2.2
Das Ausführbarkeitsproblem für Kontrollflußwege ist nicht entscheidbar.

Beweis:
Es ist nicht entscheidbar, ob ein System von Ungleichungen eine Lösung besitzt (s. [Dav 73]). Die Wegebedingung für einen Kontrollflußweg stellt aber ein solches System von Ungleichungen dar.

q. e. d.

Folgende Probleme bei der Anweisungs- und Entscheidungs- bzw. Zweigüberdeckung sind leider ebenfalls nicht entscheidbar:

1. Wird eine gegebene Anweisung durch irgendeine Eingabe ausgeführt?

2. Wird ein gegebener Entscheidungsausgang durch irgendeine Eingabe ausgeführt?

Beim iterativen Erzeugen von Testdaten tritt noch das Problem der Minimierung der Testdatenanzahl auf: Ein später erzeugtes Testdatum t_j kann alle Konstrukte überdecken, die ein vorher erzeugtes Testdatum t_i überdeckt. Dann ist t_j unter Überdeckungsgesichtspunkten überflüssig und sollte weggelassen werden[7], um Testkosten (z. B. die Ermittlung der Solldaten und den Soll-/Ist-Vergleich) einzusparen. Das Weglassen solcher Testdaten hat praktisch fast keinen Einfluß auf die Fähigkeit der Testdatenmenge, Fehler aufzudecken, wie eine Untersuchung von 10 Programmen durch Wong et al. zeigt: bei 90 bis 95% Anweisungsüberdeckung bewirkt die Minimierung der Testdatenmengen z. B. eine Reduktion von durchschnittlich 51 auf 30 Testdaten, wobei damit durchschnittlich nur 2% (höchstens 7%) weniger Fehler aufgedeckt werden (s. [Wo& 95], Table 6).

[7] Ein alternativer Ansatz besteht darin, erst zum Schluß eine minimale (oder kostengünstigste) Teilmenge der Testdaten zu bestimmen, die alle vorgegebenen Konstrukte überdeckt (s. [Rie 92a], [Lan 94]).

11.2.3 Allgemeines Verfahren zur Testdatenerzeugung

Im folgenden wird ein allgemeines Verfahren vorgestellt, um die mit der „symbolischen Ausführung rückwärts“ bestimmten Wegebedingung zu vereinfachen und entsprechende Testdaten zu ermitteln. Dabei werden folgende Begriffe benutzt und folgende Annahmen gemacht:

1. In den Entscheidungsprädikaten kommen atomare logische Ausdrücke nur in der Form „A_1 r A_2“ vor, wobei A_1 und A_2 arithmetische Ausdrücke sind und r einer der sechs relationalen Operatoren $=$, $\neq$, $<$, $\leq$, $>$, $\geq$ ist.
2. Die Wegebedingungen sind logische Ausdrücke, die aus atomaren logischen Ausdrücken durch Anwendung der logischen Operatoren *not*, *and* und *or* gebildet werden.

Die Ermittlung der Testdaten geht dann folgendermaßen vonstatten:

1. Testdaten zur Erfüllung atomarer logischer Ausdrücke sind „leicht“ zu finden.
2. Testdaten zur Erfüllung einer Wegebedingung, welche nur aus der *and*-Verknüpfung atomarer logischer Ausdrücke besteht, sind leicht zu finden, wenn die benutzten Variablen in den Teilausdrücken alle verschieden sind. Schwierig ist es dagegen, falls eine Variable in mehreren Teilausdrücken vorkommt.

   Beispiel 11.2.3
   $L_1 : b - a > 0,1$; $L_2 : 2 * \frac{b-a}{3} \leq 0,1$
   L_1 wird erfüllt von Werten mit $b > a + 0,1$.
   L_2 wird erfüllt von Werten mit $b \leq a + 0,15$.
   Sollen beide Bedingungen erfüllt werden, muß der Durchschnitt der Lösungsmengen ermittelt werden. Dies ist hier einfach, da beide Bedingungen linear in a und b sind: $a + 0,15 \geq b > a + 0,1$.

3. Testdaten zur Erfüllung einer logischen Verbindung mit Operator *or* können ermittelt werden, indem für eine der Alternativen eine Lösungsmenge gesucht wird.
4. Operator *not* macht keine Probleme. Es ist das Komplement der Lösungsmenge zu bilden.

Als Hauptproblem bleibt also die Ermittlung von Testdaten, die einen logischen Ausdruck der Form „P_1 *and* P_2 *and* ... *and* P_n“ erfüllen, wobei die P_i atomare logische Ausdrücke sind, eventuell mit dem Präfix *not*.

Die folgende Lösungsmethode vereinfacht das Problem:

1. Beseitige alle *not*-Operatoren. Ersetze dabei einen Ausdruck „$not(A_1\ r\ A_2)$“ durch „$A_1\ r'\ A_2$“, wobei r' die zu r komplementäre Relation ist.

   BEISPIEL 11.2.4
   „$not(b - a \leq 0,1)$“ wird ersetzt durch „$b - a > 0,1$“.

2. Beseitige alle relationalen Operatoren außer „=“ gemäß folgender Äquivalenz ($\leftrightarrow$):

   $a \neq b \leftrightarrow \exists x \neq 0 : x = b - a$
   $a < b \leftrightarrow \exists x > 0 : x = b - a$
   $a \leq b \leftrightarrow \exists x \geq 0 : x = b - a$
   $a > b \leftrightarrow \exists x > 0 : x = a - b$
   $a \geq b \leftrightarrow \exists x \geq 0 : x = a - b$

   Diese Umformung ist damit motiviert, daß Gleichungen leichter zu lösen sind als Ungleichungen. Dafür wird die Einführung des Existenzquantors „$\exists x$“ („es existiert ein x mit folgender Eigenschaft“) in Kauf genommen.

   BEISPIEL 11.2.5 *(vgl. Beispiel 11.2.3)*

   (a) $b - a > 0,1$ *impliziert:* $\exists x > 0 : x = b - a - 0,1$
   (b) $\frac{2*(b-a)}{3} \leq 0,1$ *impliziert:* $\exists x \geq 0 : x = 0,1 - \frac{2*(b-a)}{3}$

3. Bilde die **Pränexe Normalform** der (quantifizierten) Ausdrücke. Dies geschieht dadurch, daß bei den einzelnen Ausdrücken für die Existenzquantoren verschiedene Variablen benutzt werden und alle Quantoren vor den Gesamtausdruck gezogen werden.
   Dies wird damit motiviert, daß einzelne Bedingungen möglichst zu einem Ausdruck verschmolzen werden sollen.

   BEISPIEL 11.2.6
   Die Ausdrücke a und b aus Beispiel 11.2.5 werden zu folgenden Ausdrücken umgeformt:

   $$\exists x > 0, \exists y \geq 0 : x = b - a - 0,1 \text{ und } y = 0,1 - \frac{2*(b-a)}{3}$$

4. Beseitige Variablen durch passende Kombination der Gleichungen.

   BEISPIEL 11.2.7 *(vgl. Beispiel 11.2.6)*
   Multipliziere „$x = b - a - 0,1$*“ mit 2 und „*$y = 0,1 - \frac{2*(b-a)}{3}$*“ mit 3, damit in beiden Fällen der gleiche Teilausdruck* $2*(b-a)$ *entsteht. Das ergibt folgende Bedingung:*

   $$\exists x > 0, \exists y \geq 0 : 2*x = 2*(b-a) - 0,2 \textit{ und } 3*y = 0,3 - 2*(b-a).$$

   Durch Kombination ergibt sich:

   $$\exists x > 0, \exists y \geq 0 : 2*x + 3*y = 0,3 - 0,2$$

5. Rechne konstante Terme aus und ermittle die Lösungen.

   BEISPIEL 11.2.8
   Für den Ausdruck aus Beispiel 11.2.7 ergibt sich:

   $$\exists x > 0, \exists y \geq 0 : 2*x + 3*y = 0,1$$

   Diese Bedingung ist leicht zu erfüllen. Mit $y = 0$ *erhält man z. B.* $x = 0,05$*. Einsetzung von* $x = 0,05$ *in die Gleichung von Beispiel 11.2.6 ergibt* $0,05 = b - a - 0,1$*; somit ist* $b - a = 0,15$*;* $a = 0$ *und* $b = 0,15$ *ist also z. B. eine Lösung.*

Obwohl die Lösungsmethode bei den gewählten Beispielen erfolgreich ist, können i. allg. folgende Komplexitätsprobleme auftreten:

1. Bei *großen* Programmen sind viele Wege mit einer Vielzahl von Entscheidungsbedingungen pro Weg zu behandeln, die beim „Rückwärtsrechnen“ oft transformiert werden müssen.

2. Es reicht oft nicht aus, einen *Schleifenkörper* nicht oder nur einmal zu durchlaufen.

   BEISPIEL 11.2.9

   (a) *Siehe Beispiel 11.2.2 auf Seite 284: Der Weg* $abcfgh$*, der die Schleife nur einmal durchläuft, ist nicht ausführbar.*

   (b) **for** *i := 1* **to** *15* **do** ...
   Bei dieser for-Schleife mit konstanter Iterationszahl sind alle Wege, welche die Schleife nicht genau 15mal durchlaufen, nicht ausführbar. Durch den 15maligen Durchlauf entstehen lange Wege mit komplizierten und langen Wegebedingungen.

3. Die Lösungsmenge wächst dramatisch durch Mehrdeutigkeiten bei indizierten Variablen.

   BEISPIEL 11.2.10
   x[i] := 2 + a; x[j] := 3; c := x[i].
   Bei dieser Anweisungsfolge gilt entweder „c = 2 + a“ falls $j \neq i$ ist oder „c = 3“ falls $i = j$ ist. Die Bedingung „c = 3“ kann also erfüllt werden für „i = j oder ($i \neq j$ und a = 1)“ .

4. Bei Programmen mit *Blockstruktur* oder *Prozeduraufrufen* muß der Gültigkeitsbereich der Variablen und die Art der Parameterübergabe (by value, by name, by reference) beachtet werden, damit die Wegebedingungen korrekt gebildet werden.
   Für jedes Programmkonstrukt muß also die Semantik der Werttransformation oder der Umgebungstransformation bei einem Blockanfang klar definiert sein und beachtet werden.
   Für eine Zuweisung „$x := A$“ mit Variable x und Ausdruck A (vom selben Typ) gilt folgendes: Für ein Prädikat R, das nach der Zuweisung gelten soll, muß vorher das Prädikat $R(x \rightarrow A)$ gelten. Dies ist das Prädikat, bei dem in R jedes Vorkommen der Variablen x durch den Ausdruck A ersetzt wird.

   BEISPIEL 11.2.11
   Sei R folgendes Prädikat: „$c > 1$“.
   Sei „$c := c/a$“ die betrachtete Zuweisung. Falls nach der Zuweisung R gilt, muß vor der Zuweisung $R(c \rightarrow c/a) = [c > 1](c \rightarrow c/a) = [c/a > 1]$ gelten.

5. Bei Zahlen vom Typ Real ist der Wahrheitswert von Prädikaten oft unvorhersehbar.

   BEISPIEL 11.2.12
   Durch Rundungsfehler kann das Prädikat „a = 0“ evtl. nie erfüllt werden, wenn die Variable a vom Typ Real ist.
   Dies ist natürlich ein Programmierfehler. Bei fallenden Werten für a ist $a \leq 0$ abzufragen, bei steigenden (negativen) Werten für a ist $a \geq 0$ abzufragen; sonst ist abzufragen, ob $|a| \leq \epsilon$ für einen geeigneten kleinen positiven Wert von ϵ gilt.

6. Hat man sich bei der Auswahl eines Weges für einen *nicht ausführbaren Weg* entschieden, wird dies evtl. erst nach der langen Berechnung der kompletten Wegebedingung aufgedeckt. Bei der Wahl eines anderen Weges kann sich dies wiederholen.

Anstatt viele Wege zu erzeugen, die sich später nach der Berechnung der Wegebedingung als nicht ausführbar herausstellen, sollte daher die Wegbestimmung und die Berechnung der Wegebedingung kombiniert werden. Folgendes Vorgehen erreicht dies:

1. Bestimme den Weg vom Programmanfang bis zur ersten Entscheidung. Dies ist der einzige „bisherige Weg".

2. Bestimme für alle bisherigen Wege alle Fortsetzungen bis zur jeweils nächsten Entscheidung und bestimme, ob diese Fortsetzungen ausführbar sind. Damit ergibt sich eine neue Menge „bisheriger Wege", welche die vorher existierenden bisherigen Wege als Anfangsstücke enthalten. Beende das Verfahren, falls diese Wege alle Konstrukte überdecken oder man beweisen kann, daß die nicht überdeckten Konstrukte niemals überdeckt werden können; andernfalls wiederhole Schritt 2.

Dieses Vorgehen entspricht der symbolischen Programmausführung (genaueres dazu s. Kapitel 12.3).

11.2.4 Absurdität der definierten Testkriterien

Die nicht ausführbaren Konstrukte führen die Testkriterien aus Kapitel 7 bis 10 ad absurdum. Es ist z. B. absurd, eine hundertprozentige Überdeckung aller Entscheidungen zu fordern, wenn nicht alle Entscheidungsausgänge ausführbar sind[8]. Daher sollten die Kriterien so formuliert werden, daß nur die hundertprozentige Überdeckung aller *ausführbaren* Konstrukte gefordert wird. Dann kann allerdings nicht entschieden werden, ob eine Testdatenmenge die hundertprozentige Überdeckung aller ausführbaren Konstrukte erfüllt, da die Ausführbarkeit diverser Konstrukte nicht entscheidbar ist. Es sollten aber alle Konstrukte, von denen bewiesen werden kann, daß sie unausführbar sind, aus dem Forderungskatalog herausgenommen werden; d. h. die Überdeckungsprozentsätze sind aufgrund der reduzierten Konstruktmenge zu berechnen.

[8]Ein solches Kriterium erfüllt nicht die **Anwendbarkeitseigenschaft** (applicability property, s. [Wey 88a], S. 669).

11.3 Übungen

Übung 11.1:

Betrachten Sie das Suchprogramm von Abb. 7.2 auf Seite 193.

(a) Bestimmen Sie, an welchen Stellen Meßanweisungen in das Suchprogramm einzubauen sind, damit das (der Zweigüberdeckung entsprechende) Testwirksamkeitsmaß TWM_1 (s. Def. 7.2.5 auf S. 198) direkt bestimmt werden kann.

(b) Geben Sie eine minimale Menge von Meßanweisungen an, welche zur Berechnung von TWM_1 ausreichen. Benutzen Sie dazu das Konzept der charakteristischen Kanten bei einem spannenden Baum (vgl. Satz 11.1.1 und Beispiele 11.1.3 und 11.1.4).

(c) Falls es bei Aufgabe (b) Meßanweisungen gibt, die bei Aufgabe (a) nicht vorkommen, versuchen Sie, ihre Lage so zu verschieben, daß sie eine Teilmenge der Meßanweisungen nach Aufgabe (a) sind und weiterhin TWM_1 berechnen können.

(d) Welche Meßanweisung von Aufgabe (a) ist zusätzlich zu wählen, damit mit den Meßanweisungen von Aufgabe (c) das Maß TWM_1 ohne Kenntnis der Anzahl der Testläufe berechnet werden kann?

Übung 11.2:

Bestimmen Sie (wie bei Übung 11.1) für den Textformatierer von Beispiel 7.3.1 auf S. 208:

(a) die Meßanweisungen für die direkte Zweigüberdeckung (TWM_1),

(b) eine minimale Menge von Meßanweisungen für TWM_1,

(c) ggfs. eine Verschiebung der Meßanweisungen von Aufgabe (b), so daß sie eine Teilmenge der Meßanweisungen von Aufgabe (a) sind,

(d) eine zusätzliche Meßanweisung, welche die Kenntnis der Anzahl der Testläufe überflüssig macht.

Hinweis: Fassen Sie die *for*-Schleife in Zeile 15 und 16 als *repeat*-Schleife auf (vgl. Übung 7.1) und formulieren Sie die Schleife so um, daß die entsprechenden Meßanweisungen eingebaut werden können. Überlegen Sie sich, wie die Lage und Form der Meßanweisungen für den Rücksprung bei dieser Schleife und bei der äußeren *repeat*-Schleife (Zeilen 2 bis 26) sein muß.

Übung 11.3:

(a) Instrumentieren Sie die Bedingungen in den Zeilen 4 und 8 des Textformatierers von Beispiel 7.3.1 auf S. 208 so, daß der Grad der Erfüllung der *minimalen* Mehrfachbedingungsüberdeckung festgestellt werden kann.
Überlegen Sie, welche Variante der (beiden) Techniken von Beispiel 11.1.7 dazu geeignet ist.

(b) Wie ist die Instrumentierung jeweils bei Aufgabe (a) zu verändern, wenn die Mehrfachbedingungsüberdeckung gemessen werden soll?

Übung 11.4:
Ermitteln Sie für den Textformatierer (von Beispiel 7.3.1) und den einzigen Testlauf mit dem Text *BEISPIELE␣SIND␣DIES␣POTZBLITZ*! und $MAXPOS = 9$ (vgl. Beispiel 4.2.14 auf S. 98) für die Variablen *alarm*, *buffer*, *fill* und *bufpos*, zu wieviel Prozent die Kriterien *alle DR-Interaktionen* (s. Def. 8.2.2 auf S. 216) und *alle Referenzen* (s. Def. 8.2.3 auf S. 217) jeweils erfüllt sind.
Versuchen Sie auch festzustellen, ob gewisse DR-Interaktionen überhaupt ausführbar sind, und geben Sie dann nur den Prozentsatz für die ausführbaren DR-Interaktionen an.

Übung 11.5:

(a) Ermitteln Sie für den Textformatierer (von Beispiel 7.3.1 auf S. 208) die Definitionskontexte der Anweisung *fill := fill + bufpos* in Zeile 17. Für welche dieser Definitionskontexte gibt es *ausführbare* Kontextwege?
Hinweis: Ermitteln Sie zuerst (getrennt) die Definitionen von *fill* und *bufpos*, welche die Referenz in Zeile 17 erreichen. Ermitteln Sie dann mögliche Kontextwege und stellen Sie fest, ob diese ausführbar sind.

(b) Werden die ausführbaren Kontextwege aus Aufgabe (a) bei einem Testlauf mit dem Text aus Übung 11.4 und $MAXPOS = 9$ ausgeführt?

Übung 11.6:
Stellen Sie für den Testlauf mit dem Text aus Übung 11.4 und $MAXPOS = 9$ fest, ob bei dem Textformatierer aus Beispiel 7.3.1

(a) das *arithmetische Relationskriterium* (s. Kap. 9.1) für die Ausdrücke $bufpos \neq 0$ (in Zeile 6), $fill + bufpos < MAXPOS$ (in Zeile 8) und $bufpos = MAXPOS$ (in Zeile 21) wenigstens in einem Fall erfüllt ist und (wenn nicht), ob es überhaupt erfüllbar ist;

(b) das *Datenzugriffskriterium* und das Kriterium *kürzere Ausdrücke* für den Ausdruck *fill + bufpos* (wie in Zeile 17) erfüllt ist;

(c) das *Datenspeicherungskriterium* für die Anweisung *fill := fill + bufpos* (in Zeile 17) erfüllt ist;

(d) die (minimale) Mehrfachbedingungsüberdeckung für das Entscheidungsprädikat „*fill + bufpos* < *MAXPOS* **and** (*fill*≠ 0)“ aus Zeile 8 erfüllt ist.

Übung 11.7:

Betrachten Sie den Kontrollflußgraphen aus Übung 7.1 (Textformatierer) bzw. das Programm aus Beispiel 7.3.1. Geben Sie einen Text (oder — falls nötig — mehrere Texte) als Programmeingabe an, so daß a) alle Segmente, b) alle ausführbaren Segmentpaare ausgeführt werden.
Hinweis: Bei der Auswahl der Wege durch den Kontrollflußgraphen können Sie sich an den (minimalen) Meßanweisungen aus Übung 11.2 orientieren.

Übung 11.8:

(a) Ermitteln Sie mit der Lösungsmethode von Abschnitt 11.2.3 (mit dem Beseitigen aller *not*-Operatoren bis zum Ausrechnen der konstanten Terme) eine Lösung für die Bedingung

$$\textbf{not}\ (3*A-C-B\leq 2)\ \textbf{and}\ (C+2*D=5)\ \textbf{and}\ (A\leq D)$$

(b) Ersetzen Sie in der ersten Teilbedingung den Term $3 * A$ durch $3 * A * B$ und ermitteln Sie dafür eine Lösung.

Hinweis: Bei *einer* Gleichung mit *mehreren* Variablen kann eine Lösung gefunden werden, indem passende Variablen mit dem Wert 0 belegt werden.

Übung 11.9:

Ermitteln Sie für den Weg *abcfgcdegh* des Kontrollflußgraphen aus Abbildung 7.1 die Eingabebedingungen mit dem Verfahren von Abschnitt 11.2.1 (vgl. Beispiel 11.2.2).

11.4 Verwendete Quellen und weiterführende Literatur

Die Idee zur **optimalen Instrumentierung** von Schleifen im Programm für die Messung der C_1-Überdeckung stammt von Miller (s. [Mi& 78]. Die Charakterisierung der möglichen Einsparung von Meßpunkten wurde von Probert übernommen (s. [Pro 82], Theorem 1). Die **fundamentalen Wege** im Kontrollflußgraphen entsprechen dabei den linear unabhängigen Wegen nach McCabe (s. [Chr 75], Kap. 9, S. 189–193). Genaueres zum minimalen Instrumentieren kann z. B. bei Ramamoorthy et al., Probert und Chusho nachgelesen werden (s. [RKC 75], [Pro 82], [Chu 87]). Die meßbare Variante der **(minimalen) Mehrfachbedingungsüberdeckung** hat Liggesmeyer vorgeschlagen (s. [Lig 90], Abschnitt 2.2.4). Von Weiser et al. stammen die Instrumentierungskonzepte für das Kriterium **mehrfache Werte für Ausdrücke** (s. [WGM 85], S. 83). Die Ideen zum Messen der Überdeckung von **Datenkapseln**, Feldern und repräsentativen Werten finden sich bei Sneed (s. [Sne 86]). Die Meßmethoden der Kriterien des ungeordneten oder geordneten **Datenkontextes** wurden von Howden übernommen (vgl. [How 87], Kap. 5.5).

Die Idee zur **iterativen** Lösung des Problems der **Testdatenerzeugung** findet sich z. B. bei Omar/Mohammed (s. [OmM 89]). Die **Pfadpräfix**-Strategie zur Auswahl von zu überdeckenden Zweigen haben Prather und Myers vorgeschlagen (s. [PrM 87]). Verfahren zur Bestimmung eines Weges zu den gewünschten Konstrukten finden sich bei Gabow (s. [GMO 76]). Das allgemeine Verfahren zur Ermittlung von **Wegebedingungen** und entsprechenden Testdaten stammt von Huang [s. [Hua 75]). Ramamoorthy et al. geben ein ähnliches Verfahren an (s. [RHC 76]). Heuristiken zur Reduzierung des Lösungsraums (z. B. Einschränkung der *Integer*-Werte auf −100 bis +100) schlägt Offutt vor (s. [Off 90]). Die Sätze 11.2.1 und 11.2.2 über die Entscheidbarkeit des **Ausführbarkeitsproblems** wurden von White übernommen (s. [Whi 81]). Dort sind auch weitere Resultate zur **Entscheidbarkeit** der Ausführung von Anweisungen und Entscheidungsausgängen zu finden, die 1979 von Elaine Weyuker bewiesen wurden. Die Auswirkung der nichtausführbaren Wege auf die Testdatenerzeugung und die Definition der Testkriterien wird in [HeH 85] und [Wey 88a] angesprochen. Die Neudefinition von **ausführbaren Datenflußkriterien** und **Kontrollflußkriterien** findet sich bei Frankl und Weyuker (s. [FrW 86]).

Zusammenfassung von Teil III

Die Testdatenerzeugung für das implementationsorientierte Testen orientiert sich ausschließlich an dem implementierten Programm. Nur die erwarteten Solldaten werden aus der Spezifikation abgeleitet.
In den 70er Jahren wurde vor allem die Kontrollstruktur des Programms betrachtet, und zwar in Form des Kontrollflußgraphen. Die Testkriterien formulieren dabei Anforderungen an die Wege im Kontrollflußgraphen, die von der Testdatenmenge auszuführen sind. Diese Wege sollen alle Knoten berühren (Anweisungsüberdeckung), alle Kanten enthalten (Zweigüberdeckung) oder alle Wege, die eine Schleife höchstens ein-, zwei- oder k-mal ausführen (Grenze-Inneres- Überdeckung, strukturierte Pfadüberdeckung). Das mächtigste Kriterium dieser Art ist die Pfadüberdeckung, welche die Ausführung aller Pfade verlangt. Da dies praktisch nicht möglich ist und trotz des großen Aufwands höchstens 64% der Fehler garantiert gefunden werden (s. [How 76], [How 78b]), wurden in den 80er Jahren Kriterien herangezogen, die sich am Datenfluß orientieren, der sich längs der Programmpfade abspielt. Dabei wird jeweils der Datenfluß zwischen der Definition und der Benutzung (Referenz) einer Variablen beobachtet, eine sogenannte DR-Interaktion.
Die Datenflußkriterien verlangen nun das Ausführen von gewissen Wegen, auf denen DR-Interaktionen stattfinden. Dabei werden für eine Definition nur gewisse Referenzen betrachtet (alle Definitionen, alle E-/einige B-Referenzen, alle B-/einige E-Referenzen) oder alle Referenzen, aber nicht alle Wege von einer Definition zur Benutzung (alle DR-Interaktionen). Nur bei dem Kriterium alle DR-Wege werden alle Wege für alle DR-Interaktionen betrachtet, die allerdings bei Schleifendurchläufen begrenzt sind. Mit der Kombination der drei einfachsten Teststrategien und der Analyse von Datenflußanomalien (genaueres s. Kap. 12.2) werden schon 70% aller Fehler gefunden. In Erweiterung dieser Kriterien können Ketten oder Bäume von DR-Interaktionen betrachtet werden. Bei der Betrachtung von Ketten wird für die Definition von A in A:=B+C die zur Referenz von B (oder C) gehörige Definition von B (oder C) in einer anderen Anweisung betrachtet, usw. Bei der Betrachtung von Bäumen werden die Variablen B und C im Beispiel parallel weiterbetrachtet. Damit kommt der sogenannte Definitionskontext für die Anweisung A:=B+C ins Blickfeld.

Mit den bisherigen Kriterien werden alle Wege im Programm nur mit *einem* Testdatum getestet. Daher kann zufällige Korrektheit (z. B. $2*2 = 2+2$ trotz Vertauschung von „+“ und „∗“) nicht ausgeschlossen werden. Um die Anweisungen, Ausdrücke und Datenkapseln des Programms besser zu testen, wurden Kriterien entwickelt, die sich an möglichen Fehlern in diesen Konstrukten orientieren.
Für die Entscheidungsprädikate werden dabei Vertauschungen der Booleschen Operatoren *and*, *or* und *not*, Vertauschungen der relationalen Operatoren der Bedingungen ($<, \leq, >, \geq, =, \neq$) und additive und multiplikative Fehler in den Konstanten der arithmetischen Ausdrücke betrachtet.

Bei den Berechnungsausdrücken werden ebenfalls Fehler in den additiven und multiplikativen Konstanten betrachtet. Bei Datenzugriff und Datenspeicherung wird die Vertauschung von Variablen beobachtet.
Bei Datenkapseln sind alle Felder der Datenkapseln zu testen, wobei möglichst alle spezifizierten repräsentativen Werte zu verwenden sind.

Die bisherigen Kriterien garantieren nicht, daß ein betrachteter Fehler sich bis zu einer Programmausgabe fortpflanzt. Mit dem Kriterium *additive/multiplikative Fehler* werden daher z. B. nur 13% der falschen konstanten Werte aufgedeckt. Für einen zuverlässigen Test ist daher eine stärkere Forderung — wie bei der Mutationsanalyse — notwendig.

Die vorgestellten Testkriterien lassen sich anhand der folgenden Fragestellungen vergleichen:

- Enthält ein Kriterium logisch ein anderes?
- Wie viele Testdaten sind für den Test eines Programms erforderlich?
- Welche Fehler und wieviel Prozent aller Fehler lassen sich mit einem Testkriterium aufdecken?

Für eine verläßliche Beurteilung, ob ein konkreter Test den vorab geforderten Qualitätsmaßstäben entspricht, ist die Messung der entsprechenden Testwirksamkeit dieses Tests erforderlich. Für die einzelnen Testkriterien wurde daher angegeben, wie der Grad der Erfüllung dieser Kriterien automatisch gemessen werden kann, wenn das Programm entsprechend mit Meßpunkten „instrumentiert“ wird. Dabei wurde darauf hingewiesen, daß eine hundertprozentige Erfüllung der Kriterien aus theoretischen und praktischen Gründen nicht immer möglich ist (Entscheidbarkeitsprobleme, Komplexitätsprobleme).

Es hängt von der gewünschten Softwarequalität und dem akzeptablen Testaufwand ab, welche Kriterien beim Testen herangezogen werden. Da das implementations- und spezifikationsorientierte Testen nicht alle Fehler aufdeckt und bisher nur relativ kleine, sequentielle Programme betrachtet wurden, müssen noch weitere Kriterien und entsprechende Methoden analysiert werden. Das geschieht in Teil IV.