Teil II

Spezifikationsorientiertes Testen
Beim spezifikationsorientierten Test erfolgt die Testfallenstellung für ein Programm auf der Basis der Spezifikation des Programms. Mit den erstellten Testfällen werden Abweichungen zwischen dem spezifizierten (gewünschten) Verhalten eines Programms und dem implementierten Verhalten eines Programms getestet.

Voraussetzung für den spezifikationsorientierten Test ist das Vorhandensein einer präzisen und — wenn möglich — formalen Spezifikation. Unter einer formalen Spezifikation eines Programms wird eine Spezifikation verstanden, bei der die gewünschte Semantik des Programms mathematisch präzise beschrieben wird. Im Gegensatz dazu wird bei einer informellen Spezifikation die gewünschte Semantik des Programms in natürlicher Sprache oder in graphischer Notation beschrieben.

Beim spezifikationsorientierten Test werden die Methoden danach unterschieden, ob sie unsystematisch oder systematisch vorgehen. Bei den unsystematischen Methoden werden die Eingabedaten zufällig oder aufgrund von Erfahrung mit bekannten Fehlern ausgewählt (Kapitel 4.1). Die systematischen Methoden können eine informelle oder eine formale Spezifikationsbasis haben. Dabei wird die Menge der möglichen Eingabedaten eines Programms in geeignete Äquivalenzklassen aufgeteilt. Aus jeder Äquivalenzklasse werden ein oder mehrere Eingabedaten ausgewählt. Drei Gruppen von systematischen Methoden zur spezifikationsorientierten Testdatenerstellung werden hier vorgestellt: die datenbereichsbezogenen Testmethoden (Kapitel 4.2), die funktionsbezogenen Testmethoden (Kapitel 4.3) und Testmethoden auf der Basis formaler Spezifikationen, d. h. Pfadausdrücken (Kapitel 5.1) und algebraischen Spezifikationen (Kapitel 5.2).

Die Testansätze unterscheiden sich relativ stark, da die Semantik des Programms bei den verschiedenen Ansätzen sehr unterschiedlich modelliert wird. Beim datenbereichsbezogenen Testen wird beispielsweise nur der Wertebereich der Parameter formal beschrieben und die Berechnung der Funktionswerte nur informell. Beim funktionsbezogenen Testen wird die Komposition der Gesamtfunktion aus Einzelfunktionen beschrieben; beim Testen auf Basis von Pfadausdrücken werden die möglichen Reihenfolgen beschrieben, in denen die Einzelfunktionen aufgerufen werden können.

In Kapitel 4 und 5 werden die Testmethoden vorgestellt, in Kapitel 6 werden sie verglichen und bewertet. Dabei werden der Aufwand, die Voraussetzungen und die Art und Anzahl der Fehler, die damit aufgedeckt werden können, beschrieben.
4 Datenbereichsbezogenes und funktionsbezogenes Testen

4.1 Unsystematisches Datenbereichs- oder funktionsbezogenes Testen

Zu den unsystematischen Methoden gehören das zufällige Testen (random test) und die Fehlererwartungsmethode (error guessing).

Zufälliges Testen

Fehlererwartungsmethode

Die Fehlererwartungsmethode beinhaltet folgendes Vorgehen:

1. Die Testperson legt eine Liste aller möglichen Fehler oder fehlerträchtigen Situationen an, die bei der Erstellung eines Programms auftreten können. Die Liste wird aufgrund der eigenen Erfahrungen, beim Programmieren oder aufgrund von Berichten anderer Softwareentwickler erstellt (siehe Kapitel 2.2 über Fehlerhäufigkeiten).

2. Die Testperson versucht nachzu vollziehen, welche Überlegungen der Programmierer beim Lesen der Anforderungs- beziehungsweise Entwurfsspezifikation gehabt haben mag. Die Testperson konzentriert sich hierbei besonders auf Punkte, die der Programmierer nicht berücksichtigt oder falsch interpretiert haben könnte.

Beispiel 4.1.1
Ein Programm hat die Aufgabe, eine Liste von ganzzahligen Werten zu sortieren. Mögliche Eingabedaten nach der Fehlererwartungsmethode sind:
4.2 Systematisches datenbereichsbezogenes Testen

- die leere Eingabeliste,
- eine Eingabeliste, die schon sortiert ist,
- eine Eingabeliste, deren Werte alle gleich sind,
- eine Eingabeliste, in der die Werte falsch herum sortiert sind.

Da die Fehlererwartungsmethode sehr von der persönlichen Erfahrung der Testperson abhängig ist, wünscht man sich systematische Methoden. Ziel der systematischen Methoden ist es, eine endliche Anzahl repräsentativer Testfälle zu finden, die eine gute Approximation an den vollständigen oder idealen Test darstellen.

4.2 Systematisches datenbereichsbezogenes Testen


4.2.1 Äquivalenzklassen

Die Methode der Äquivalenzklassenbildung legt fest, wie auf der Basis einer informellen Spezifikation systematisch Äquivalenzklassen gebildet werden und zu Testzwecken Eingabedaten mit Hilfe der Äquivalenzklassen bestimmt werden. Für die gebildeten Äquivalenzklassen sollte idealerweise gelten, daß der Test mit einem beliebigen Wert \( x \) aus einer Klasse \( K \) in Bezug auf die Fehleraufdeckung zu einem Test mit einem anderen Wert \( y \) aus der gleichen Klasse \( K \) äquivalent ist. Allerdings kann die Testperson nie sicher sein, daß alle Werte einer Klasse zueinander äquivalent sind.

Die Methode der Äquivalenzklassenbildung besteht aus drei Schritten:

1. Aufstellung von Eingabebedingungen.
2. Bildung von Äquivalenzklassen.
3. Definition von Eingabedaten mit Hilfe der Äquivalenzklassen.

Schritt 1: Eingabebedingungen aufstellen
Zuerst werden die Eingabebedingungen des Programms anhand einer informellen Spezifikation des Programms gebildet.
Eine Eingabebedingung ist hierbei gewöhnlich ein Satz oder ein Abschnitt der Spezifikation.
Schrift 2: Äquivalenzklassen bilden
Zu jeder Eingabebedingung werden die gültigen und ungültigen Äquivalenzklassen bestimmt. Die gültigen Äquivalenzklassen enthalten gültige Eingabewerte, die als Normalfall behandelt werden. Die ungültigen Äquivalenzklassen enthalten ungültige Eingabewerte, die eine Fehlerbehandlung erfordern.

Richtlinien zur Bildung von Äquivalenzklassen:

1. Ist ein geordneter Wertebereich für eine EingabevARIABLE vorgegeben, so bilden die Werte des Wertebereichs eine gültige Klasse. Die Werte unterhalb der unteren Bereichsgrenze sowie die Werte oberhalb der oberen Bereichsgrenze bilden jeweils eine ungültige Klasse.

Beispiel 4.2.1
Der Wertebereich enthalte Werte zwischen 10 und 20 für eine EingabevARIABLE x.

Die gültige Äquivalenzklasse ist dann: \[10 \leq x \leq 20\],
die beiden ungültigen Äquivalenzklassen sind: \[x < 10\] und \[x > 20\].

2. Bei Datenstrukturen kann die Anzahl der Elemente der Datenstruktur vorgegeben sein.

Beispiel 4.2.2
Eine Liste enthalte 1 bis 255 Elemente, dann gilt:

die gültige Äquivalenzklasse umfaßt 1 bis 255 Elemente,
die beiden ungültigen Äquivalenzklassen enthalten 0 Elemente bzw. mehr als 255 Elemente.


Beispiel 4.2.3
Erlaubte Werte des Typs Farbe seien rot, gelb, und grün. Dann gilt:

die drei gültigen Äquivalenzklassen sind Klassen mit je einem Element: rot, gelb, grün;
die ungültige Äquivalenzklasse enthält alle anderen Farben: blau, violett, braun, ... 

Beispiel 4.2.4
Eine Zeichenfolge besteht aus Ziffern und/oder Buchstaben. Wenn zu vermuten ist, daß Ziffern und Buchstaben in diesem Zusammenhang eine unterschiedliche Bedeutung haben, dann gibt es zu dieser Bedingung die beiden folgenden gültigen (Äquivalenz-)Klassen, die sich überschneiden¹:

- Zeichenfolgen, die Ziffern enthalten,
- Zeichenfolgen, die Buchstaben enthalten.

Wenn Buchstaben und Ziffern gleich behandelt werden, dann reicht die Bildung einer gültigen Äquivalenzklasse „alphanumerische Zeichen“, die aus allen Ziffern und Buchstaben besteht. Die ungültige Äquivalenzklasse besteht aus Zeichenfolgen, die Sonderzeichen enthalten.

Eine weitere Eingabebedingung sei, daß das erste Zeichen der Zeichenfolge ein Buchstabe sein muß. Dann gibt es zusätzlich die gültige Äquivalenzklasse „erstes Zeichen ist ein Buchstabe“ und die ungültige Äquivalenzklasse „erstes Zeichen ist kein Buchstabe“.

Schritt 3: Eingabedaten definieren
Zur Definition der Eingabedaten mit Hilfe der gebildeten Äquivalenzklassen ist folgendes Vorgehen sinnvoll:

1. Jede gültige Äquivalenzklasse wird zwecks Identifizierung mit einer eindeutigen Zahl gekennzeichnet. Die zugehörigen ungültigen Äquivalenzklassen werden mit dieser Zahl und einem unterschiedlichen Buchstaben (a, b, c, etc.) gekennzeichnet.

2. Es wird ein Eingabedatum ausgewählt, so daß die einzelnen Komponenten (Testwerte²) des Eingabedatums möglichst viele der bisher nicht abgedeckten gültigen Äquivalenzklassen abdecken. Dieses Vorgehen wird solange wiederholt, bis alle gültigen Äquivalenzklassen abgedeckt sind.

(Ein Eingabedatum deckt eine Äquivalenzklasse ab, wenn es als Komponente einen Testwert enthält, der ein Element der Äquivalenzklasse ist. Ein Eingabedatum kann also mehrere Äquivalenzklassen — für verschiedene Eingabebedingungen — gleichzeitig abdecken.)


¹Daher sind diese Klassen keine Äquivalenzklassen im streng mathematischen Sinne, die ja überschneidungsfrei (disjunkt) sein müssen.
²Siehe Definition 3.5.2.3 auf Seite 61
Dieses Vorgehen ist zu wiederholen, bis alle ungültigen Äquivalenzklassen abgedeckt sind.
Mit der Zuordnung von je einem individuellen Eingabedatum zu jeder ungültigen Äquivalenzklasse wird erreicht, daß die Reaktion eines Programms bei der Entdeckung einer fehlerhaften Eingabe unbeeinflußt von weiteren fehlerhaften Eingaben beobachtet werden kann.

Beispiel 4.2.5

PRINT hat also folgende Syntax: PRINT <Dateiname> <Zeilenanzahl>
Der Dateiname besteht aus mindestens einem und bis zu sechs Zeichen, die Buchstaben oder Ziffern sein können. Das erste Zeichen des Dateinamens muß ein Buchstabe sein. Die Zeilenanzahl besteht aus mindestens einer und bis zu drei Ziffern. Die Zeilenanzahl ist größer als 0 und kleiner als 1000.

Tabelle 4.1 listet die Ergebnisse von Schritt 1 und 2 auf: die Eingabebedingungen, die gültigen (Äquivalenz-)Klassen (mit einer fortlaufenden Nummer in Klammern) und die ungültigen (Äquivalenz-)Klassen (mit der entsprechenden Nummer-Buchstaben-Kombination in Klammern).

<table>
<thead>
<tr>
<th>Eingabebedingung</th>
<th>gültige Klasse</th>
<th>ungültige Klasse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anzahl Parameter</td>
<td>zwei (1)</td>
<td>keine (1a), einer (1b), mehr als zwei (1c)</td>
</tr>
<tr>
<td>Dateiname (Länge)</td>
<td>1 bis 6 Zeichen (2)</td>
<td>0 Zeichen (2a), mehr als 6 Zeichen (2b)</td>
</tr>
<tr>
<td>Dateiname (Zeichen)</td>
<td>hat Buchstaben oder Ziffern (3)</td>
<td>hat sonstige Zeichen (3a)</td>
</tr>
<tr>
<td>Dateinamen (erstes Zeichen)</td>
<td>ist ein Buchstabe (4)</td>
<td>ist kein Buchstabe (4a)</td>
</tr>
<tr>
<td>Zeilenanzahl (Zeichen)</td>
<td>enthält nur Ziffern (5)</td>
<td>enthält Zeichen, das keine Ziffer ist (5a)</td>
</tr>
<tr>
<td>Zeilenanzahl (Ziffern)</td>
<td>1 bis 3 Ziffern (6)</td>
<td>mehr als 3 Ziffern (6a)&lt;sup&gt;3&lt;/sup&gt;</td>
</tr>
<tr>
<td>Zeilenanzahl (Größe)</td>
<td>größer als 0 und kleiner als 1000 (7)</td>
<td>kleiner oder gleich 0 (7a), größer oder gleich 1000 (7b)</td>
</tr>
</tbody>
</table>

Tab. 4.1 Eingabebedingungen und (Äquivalenz-)Klassen

Schritt 2: Eingabedaten definieren
Im folgenden werden Eingabedaten und — in Klammern — die mit dem jeweiligen Eingabedatum abgedeckten Äquivalenzklassen angegeben.

<sup>3</sup>Bemerkung: Die Äquivalenzklasse „Anzahl der Ziffern ist 0“ wird weggelassen, da sie Teil der Äquivalenzklasse 1b „Anzahl der Parameter ist 1“ ist.
4.2 Systematisches Datenbereichsbezogenes Testen

Eingabedatum zur Abdeckung aller gültigen Äquivalenzklassen:

```
PRINT abc1 22 (1, 2, 3, 4, 5, 6, 7)
```

Eingabedaten zur Abdeckung der ungültigen Äquivalenzklassen (die fehlerhaften Werte bzw. Zeichen sind unterstrichen):

1. PRINT __  
2. PRINT abc1 __  
3. PRINT abc1 22 13  
4. PRINT ___ 22  
5. PRINT abcdefh 22  
6. PRINT a-t 22  
7. PRINT a4a 22  
8. PRINT abc1 abc  
9. PRINT abc1 0456  
10. PRINT abc1 0  
11. PRINT abc1 1000

Beispiel 4.2.6

Für den Textformatierer aus Kapitel 1.2 können Bedingungen B1 bis B4 angegeben werden, die sich auf das letzte eingelesene Zeichen zurück und das damit aufgebaute (Teil-)Wort w beziehen.

B1: $z$ ist BL, NL oder EOF (end of file), d. h. formal: $z \in \{BL, NL, EOF\}$;

B2: $w$ hat höchstens MAXPOS Zeichen, d. h. formal: $\text{Länge}(w) \leq \text{MAXPOS}$;

B3: $w$ paßt noch in die aktuelle, teilweise gefüllte Zeile, d. h. formal: $(f + 1 + \text{Länge}(w)) \leq \text{MAXPOS}$.

Dabei wird angenommen, daß die Zeile schon $f$ Zeichen enthält und mit einem Wort ohne Blank aufhört.

B4: In der aktuellen Zeile sind schon Zeichen enthalten, d. h. $(f > 0)$.
### Tab. 4.2 Äquivalenzklassen für den Textformatierer

Die in Tabelle 4.2 dargestellten Äquivalenzklassen können zu den vier Bedingungen gebildet werden (die Nummer der Klasse steht in Klammern vor der Bedingung).

Folgender Testfall deckt die gültigen Äquivalenzklassen 1 bis 4 ab:

B1 und B2 und B3 und B4

Ein konkretes Testdatum für diesen Testfall wäre (bei \( \text{MAXPOS} = 80 \) und \( f = 30 \) Zeichen in der aktuellen Zeile) ein einzufügender Wort mit der Länge 45 und dem aktuellen (Trenn-)Zeichen \( z = \text{BL} \).

Folgende Testfälle decken je eine ungültige Äquivalenzklasse ab:

1. B1 und B2 und B3 und (nicht B4) \[ \text{deckt (4a) ab} \]
2. B1 und B2 und (nicht B3) und B4 \[ \text{deckt (3a) ab} \]
3. B1 und (nicht B2) und B4 \[ \text{deckt (2a) ab} \]
4. (nicht B1) und B2 und B3 und B4 \[ \text{deckt (1a) ab} \]

Bei Testfall 3 ist zu beachten, daß „nicht B2“ die Ungültigkeit von B3 (also „nicht B3“) impliziert. Die dritte Forderung bei Schritt 3 der Methode der Äquivalenzklassenbildung (eine und nur eine ungültige Klasse abzudecken) ist also in diesem Fall zu modifizieren.

Konkrete Testdaten sind für obige Fälle 1 bis 4 z. B.:

1. \( z = \text{BL}, \text{Länge}(w) = 55, \text{MAXPOS} = 80, f = 0, \) also
   \( \text{Länge}(w) \leq \text{MAXPOS}, \text{Länge}(w) \leq \text{MAXPOS} - f - 1 = \text{MAXPOS} - 1 = 79. \)

2. \( z = \text{NL}, \text{Länge}(w) = 55 \leq 80 = \text{MAXPOS}, \) also bei \( f = 30 > 0: \)
   \( 55 = \text{Länge}(w) > \text{MAXPOS} - f - 1 = 49 \)

3. \( z = \text{EOF}, \text{Länge}(w) = 82, \text{MAXPOS} = 80, f = 30 > 0, \) also
   \( \text{Länge}(w) > \text{MAXPOS}. \)

\(^4\text{zw. Äquivalenzklassen 1b und 4a}\)

19 Aug 2002 22:14
4. $z = D, \text{MAX POS} = 80, f = 30 > 0, \text{Länge}(w) = 45$, also

\begin{align*}
\text{Länge}(w) &= 45 \leq 80 = \text{MAX POS} \quad \text{und} \\
\text{Länge}(w) &= 45 \leq 49 = \text{MAX POS} - f - 1
\end{align*}

Bei der Äquivalenzklassenmethode ist die Güte der Testdaten abhängig von der Aussagekraft der Spezifikation. Im Beispiel 4.2.5 wird bei der Äquivalenzklassenbildung beispielsweise die Eingabe führender Nullen für den zweiten Parameter zugelassen, da darüber in der informellen Spezifikation keine Aussage gemacht wird. Nur deshalb sind die ungültigen Äquivalenzklassen (6a) und (7b) verschieden. Verhoben man dagegen führende Nullen, beschreiben (6a) und (7b) dieselbe Klasse von Werten.

Die Ermittlung der Eingabedaten hängt nicht nur von der Anzahl der Eingabebedingungen ab, sondern auch von den Antworten auf folgende Fragen:

- Welche Werte aus einer Äquivalenzklasse sollen gewählt werden?
- Welche Kombinationen von Eingabebedingungen sollen getestet werden, d. h., welche Kombinationen von Testwerten bilden ein Eingabedatum?

Diese Fragen werden in den folgenden Abschnitten 4.2.2 und 4.2.3 beantwortet.

### 4.2.2 Grenzwerte und spezielle Werte

Für die Auswahl von Werten aus einer Äquivalenzklasse, deren Werte aus einer geordneten Menge stammen, empfiehlt sich die Methode der Grenzwertanalyse, die in zwei Schritten durchgeführt wird:

**Schritt 1:** Es werden Testwerte ausgewählt, die sich direkt auf oder neben den beiden Grenzen einer Eingabeäquivalenzklasse befinden.

**Richtlinien für die Bestimmung der Testwerte für eine Eingabeäquivalenzklasse:**

1. Repräsentiert die Äquivalenzklasse einen Wertebereich, so sind die gültigen Testwerte der größte und kleinste Wert des Wertebereichs. Die ungültigen Testwerte sind diejenigen Werte, die außerhalb und in nächster Nähe zum Wertebereich liegen. Für einen Wertebereich, der beispielsweise mit „$1 \leq x \leq 10$, x ganzzahlig“ definiert ist, werden als gültige Testwerte 1 und 10 und als ungültige Testwerte 0 und 11 ausgewählt. Falls $x$ reellwertig ist, werden als ungültige Testwerte dagegen $1 - \epsilon$ und $10 + \epsilon$ (mit kleinstmöglichem $\epsilon > 0$) gewählt.

2. Wenn eine Äquivalenzklasse eine gültige Anzahl von Werten repräsentiert, dann wird die kleinste und die größte gültige Anzahl von Werten ausgewählt. Die benachbarten Anzahlen (unterhalb der kleinsten gültigen und oberhalb der größten gültigen) sind die ungültigen Testwerte.
Beispiel 4.2.7

Besteht eine Liste minimal aus einem Element und maximal aus 255 Elementen, dann sind die gültigen Testwerte 1 und 255; die ungültigen Testwerte sind 0 und 256, falls die Implementation diese Werte zuläßt. (Andernfalls kann der Fehler der Grenzwertüberschreitung nicht vorkommen.)


Beispiel 4.2.8

Im folgenden sollen mit der Methode der Grenzwertanalyse Eingabedaten für den im Beispiel 4.2.5 beschriebenen PRINT-Befehl bestimmt werden.

Abwendung von Schritt 1:
Zuerst wird eine Tabelle aufgestellt, die für jede Äquivalenzklasse die mit der Grenzwertanalyse bestimmten gültigen und ungültigen Testfälle angibt (s. Tabelle 4.3)\(^5\).

<table>
<thead>
<tr>
<th>Äquivalenzklasse</th>
<th>gültige Testfälle</th>
<th>ungültige Testfälle</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Anzahl der Parameter</td>
<td>2</td>
<td>1, 3</td>
</tr>
<tr>
<td>(2) Länge des Dateinamens</td>
<td>1, 6</td>
<td>0, 7</td>
</tr>
<tr>
<td>(6) Zeilenanzahl: 1 bis 3 Ziffern</td>
<td>1, 3</td>
<td>0, 4</td>
</tr>
<tr>
<td>(7) 0 &lt; Zeilenanzahl &lt; 1000</td>
<td>1, 999</td>
<td>0, 1000</td>
</tr>
</tbody>
</table>

Tab. 4.3 Gültige und ungültige Grenzwerte für den PRINT-Befehl

Eingabedaten, die zu den gültigen und ungültigen Testfällen passen, sind im folgenden aufgeführt. Für jedes Eingabedatum ist angegeben, ob die entsprechende Bedingung auch mit der Methode der Äquivalenzklassenbildung (siehe Beispiel 4.2.5 in Abschnitt 4.2.1) erfüllt würde.

\(^5\)Die gültigen Äquivalenzklassen 3, 4, 5 aus Beispiel 4.2.5 und Tabelle 4.1 beschreiben keine geordneten Wertebereiche; daher gibt es dafür keine Grenzwerte.
Zuerst werden in Tabelle 4.4 gültige Eingabedaten und in Klammern die Nummern der entsprechenden Äquivalenzklassen angegeben.

<table>
<thead>
<tr>
<th>Testdaten zur Grenzwertanalyse</th>
<th>erfaßt bei Äquivalenzklassen?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) PRINT abc1 22</td>
<td>ja</td>
</tr>
<tr>
<td>(2) PRINT a 100</td>
<td>eventuell</td>
</tr>
<tr>
<td>(2) PRINT abedef 200</td>
<td>eventuell</td>
</tr>
<tr>
<td>(6) PRINT abc 8</td>
<td>eventuell</td>
</tr>
<tr>
<td>(6) PRINT abc 345</td>
<td>eventuell</td>
</tr>
<tr>
<td>(7) PRINT abc 1</td>
<td>eventuell</td>
</tr>
<tr>
<td>(7) PRINT abc 999</td>
<td>eventuell</td>
</tr>
</tbody>
</table>

**Tab. 4.4** Testdaten für gültige Äquivalenzklassen des PRINT-Befehls

Nur beim ersten Eingabedatum stimmt die Bedingung „Anzahl der Parameter = 2“ mit der gültigen Äquivalenzklasse (1) aus Beispiel 4.2.5 überein. Bei den anderen sechs Eingabedaten liegen die Testwerte in der gültigen Äquivalenzklasse, aber die Äquivalenzklasse enthält noch andere Werte. Die Grenzwerte werden also bei der Methode der Äquivalenzklassenbildung nur eventuell ausgewählt.

<table>
<thead>
<tr>
<th>Testdaten zur Grenzwertanalyse</th>
<th>erfaßt bei Äquivalenzklassen?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1b) PRINT abc</td>
<td>ja</td>
</tr>
<tr>
<td>(1c) PRINT abc 20 300</td>
<td>eventuell</td>
</tr>
<tr>
<td>(2a) PRINT abc 20</td>
<td>ja</td>
</tr>
<tr>
<td>(2b) PRINT abedefg 20</td>
<td>eventuell</td>
</tr>
<tr>
<td>(6a) PRINT abc 4568</td>
<td>eventuell</td>
</tr>
<tr>
<td>(7a) PRINT abc 0</td>
<td>eventuell</td>
</tr>
<tr>
<td>(7b) PRINT abc 1000</td>
<td>eventuell</td>
</tr>
</tbody>
</table>

**Tab. 4.5** Testdaten für ungültige Äquivalenzklassen des PRINT-Befehls

In Tabelle 4.5 werden ungültige Eingabedaten angegeben. Nur beim ersten und dritten Eingabedatum stimmen die Grenzwertbedingungen mit den ungültigen Äquivalenzklassen von Beispiel 4.2.5 überein: „ein Parameter“ (Klasse 1b); „Länge des Dateinamens: 0“ (Klasse 2a). Bei den anderen Eingabedaten liegen die Grenzwerte in den ungültigen Äquivalenzklassen, aber diese enthalten noch andere Werte. Daher können die Grenzwerte bei der Methode der Äquivalenzklassenbildung eventuell gewählt werden.

Anwendung von Schritt 2:
Die Spezifikation des PRINT-Befehls sagt nichts über das Format der zu erwartenden Druckausgabe aus. Die Spezifikation wird deshalb um folgende Angaben erweitert:
1. Es werden maximal 20 Seiten gedruckt.
2. Eine Seite enthält bis zu 45 Zeilen.
3. Mit Ausnahme der letzten Seite müssen immer volle Seiten gedruckt werden.

Die folgenden Ausgabebedingungen können daraus abgeleitet werden:

1. Es können $X$ Seiten gedruckt werden, wobei $1 \leq X \leq 20$ gilt.
2. Die letzte Seite enthält $Y$ Zeilen, wobei $1 \leq Y \leq 45$ gilt.
3. Die ersten $X - 1$ Seiten enthalten jeweils 45 Zeilen.

Mit der Methode der Grenzwertanalyse ergeben sich folgende zu testende Ausgabewerte für $X$: 0, 1, 20, 21; für $Y$ ergeben sich folgende Werte: 0, 1, 45, 46. Dazu sind Eingabedaten zu ermitteln, welche diese Ausgabewerte erzeugen. Dabei wird angenommen, daß die Datei mit Namen „abc“ genügend Zeichen enthält, um mindestens 901 Zeilen zu füllen.

1. **PRINT** abc 0  \hspace{1cm} (X = 0, Y = 0)
2. **PRINT** abc 45  \hspace{1cm} (X = 1, Y = 45)
3. **PRINT** abc 900 \hspace{1cm} (X = 20, Y = 45)
4. **PRINT** abc 901 \hspace{1cm} (X = 21, Y = 1)

Der Ausgabewert $Y = 1$ wird auch mit folgendem Eingabedatum erzeugt:

5. **PRINT** abc 46

Dies bedeutet, daß die letzte zu druckende Seite eine Zeile enthalten sollte. Mit dem fünften Eingabedatum wird aber auch getestet, ob 46 Zeilen auf eine Seite gedruckt werden ($Y = 46$), welches einen ungültigen Ausgabewert darstellt.

**Beispiel 4.2.9**

Für den Textformatierer aus Kapitel 1.2 werden in Beispiel 4.2.6 (ab Seite 79) vier Bedingungen $B_1$, $B_2$, $B_3$, $B_4$ und jeweils vier gültige und ungültige Äquivalenzklassen angegeben (für das aktuell gelesene Zeichen $z$, für das aktuelle [Teil-]Wort $w$ und die aktuelle Anzahl $f$ der Zeichen in der Ausgabezeile):

\begin{align*}
(B_1): \ &z \in \{BL, NL, EOF\} \\
(B_2): \ &\text{Länge}(w) \leq \text{MAX POS} \\
(B_3): \ &\text{Länge}(w) \leq \text{MAX POS} - f - 1 \\
(B_4): \ &f > 0.
\end{align*}
4.2 Systematisches datenbereichsbezogenes Testen

Bedingung B1 bezieht sich auf eine Menge ohne Ordnung, also gibt es keine Grenzwerte; B2, B3 und B4 betreffen geordnete Werte, also gibt es Grenzwerte.


<table>
<thead>
<tr>
<th>Bedingung</th>
<th>Gültige Grenzwerte</th>
<th>Ungültige Grenzwerte</th>
</tr>
</thead>
<tbody>
<tr>
<td>B2</td>
<td>$L_\text{änge}(w) = \text{MAX} \text{POS}$</td>
<td>$L_\text{änge}(w) = \text{MAX} \text{POS} + 1$</td>
</tr>
<tr>
<td>B3</td>
<td>$L_\text{änge}(w) = \text{MAX} \text{POS} - f - 1$</td>
<td>$L_\text{änge}(w) = \text{MAX} \text{POS} - f$</td>
</tr>
<tr>
<td>B4</td>
<td>$f = 1$</td>
<td>$f = 0$</td>
</tr>
</tbody>
</table>

Tab. 4.6 Gültige und ungültige Grenzwerte für den Textformatierer

Ein Test mit Grenzwerten ist ein Test mit Werten, die eine spezielle Bedeutung haben. Im folgenden soll der Begriff „spezielle Werte“ aber in einem engeren Sinne benutzt werden. Dann sind spezielle Werte nur solche Werte, die unabhängig von der Äquivalenzklasseneinteilung eine Bedeutung haben. Die Bedeutung kann also nur vom allgemeinen Typ der Werte abhängen. Bei Zahlen haben z. B. die Werte 0 und 1 eine spezielle Bedeutung (neutrale Elemente der Addition und Multiplikation), bei Arrays sind dies entsprechende Arrays mit lauter Nullen oder lauter Einsen als Einträge, beim Typ „Character“ ist es das Blank.

Unter dem Kombinationsgesichtspunkt sind folgende Tests zu betrachten:

1. Die Werte von Eingabevariablen mit ähnlicher Bedeutung sollten verschieden sein.
2. Die Eingabewerte für die Elemente eines Arrays sollten verschieden sein.
3. Wenn zwei Arrays $A$ und $B$ Elemente aus derselben Menge speichern, sollten Elemente in korrespondierenden Positionen verschieden sein; d. h. für alle möglichen Werte von $i$ sollte gelten: $A(i) \neq B(i)$.

Alle drei Testarten 1 bis 3 erfüllen also die Regel „testen verschiedene Werte“.

4.2.3 Ursache/Wirkungsgraphen


Das PRINT-Programm könnte z. B. derart implementiert sein, daß es einen Ausgabepuffer in der Größe von 900 Zeilen verwaltet, also die maximale Ausgabegröße

19 Aug 2002 22:14


Die Ursache/Wirkungsgraph-Methode enthält folgende Schritte:

**Schritt 1:** Zuerst wird die Spezifikation in „bearbeitbare Stücke“ zerlegt, damit der aus der Spezifikation abgeleitete Ursache/Wirkungsgraph nicht zu komplex wird.

Beim Testen eines Mehrbenutzersystems sind z. B. die einzelnen Kommandos des Mehrbenutzersystems die „bearbeitbaren Stücke“ der Spezifikation. Beim Testen eines Übersetzers ist jede einzelne Anweisung der Sprache ein „bearbeitbares Stück“.

**Schritt 2:** Die Ursachen und Wirkungen des Programms werden anhand der (formalen) Spezifikation des Programms identifiziert.

Ursachen sind Eingabebedingungen, denen Boolesche Wahrheitswerte zugeordnet werden können, z. B. Fahrzeugtyp = PKW, Gehalt > 0, Gehalt ≤ 5200.

Eine Wirkung ist eine Ausgabebedingung oder eine Systemtransformation. Eine Systemtransformation ist eine Veränderung des Programm- bzw. Systemzustands, die für den Benutzer nicht direkt erkennbar ist.

Alle Ursachen und Wirkungen werden durch eine eindeutige Zahl oder durch ein Kürzel gekennzeichnet, damit sie einfach referenziert werden können.

**Schritt 3:** Die logischen Beziehungen zwischen den Ursachen und Wirkungen (d. h. der semantische Inhalt der Spezifikation) werden als gerichteter Boolescher Graph, genannt Ursache/Wirkungsgraph (kurz: UWG), dargestellt.

Es gibt folgende Verknüpfungen im Ursache/Wirkungsgraphen:

1. Jede Ursache oder Wirkung mit Zahl (oder Kürzel) $i$ wird durch einen Knoten $i$ dargestellt.

2. **Identische** Verknüpfung: Wirkung $j$ ist gleich Ursache $i$ (siehe Abb. 4.1 links).

---

3. **Negations-Verknüpfung**: Wirkung $j$ ist vorhanden, wenn Ursache $i$ nicht vorhanden ist (siehe Abbildung 4.1 rechts).

4. **Oder-Verknüpfung**: Wirkung $j$ ist vorhanden, wenn Ursache $i_1$ oder Ursache $i_2$ oder ... oder Ursache $i_n$ vorhanden ist (siehe Abbildung 4.2 links).

5. **Und-Verknüpfung**: Wirkung $j$ ist vorhanden, wenn Ursache $i_1$ und Ursache $i_2$ und ... und Ursache $i_n$ vorhanden ist (siehe Abbildung 4.2 rechts).

Für den Aufbau, die Darstellung und die Bedeutung von Ursache/Wirkungsgraphen gelten noch folgende Hinweise und Regeln:

1. Mit **und, oder, nicht** lassen sich bekanntlich alle logischen Verknüpfungen beschreiben. Allerdings müssen zwischen Ursachen und Wirkungen evtl. mehrere Kanten eingefügt werden, um Zwischenresultate zu berechnen.

2. Die Verknüpfungslogik entspricht den Regeln der Booleschen Algebra (mit $0 = False$, $1 = True$).

3. Der Graph darf keine Rückkopplungen bzw. Schleifen enthalten$^7$.

---

$^7$Die erlaubten Verbindungen entsprechen also den Verbindungen bei kombinatorischen Schaltungen.
4. Falls durch die UND-Knoten und ODER-Knoten die Richtung der Pfeile klar erkennbar ist, kann in den Darstellungen die Pfeilspitze weggelassen werden.

**Schritt 4:** Der UWG wird mit Einschränkungen, welche unmögliche Kombinationen von Eingangsbedingungen ausschließen, versehen. Die ausgeschlossenen Kombinationen werden bei der späteren Generierung der Testfälle (Belegungen der Ursachen mit Wahrheitswerten) in Schritt 5 und Schritt 6 nicht betrachtet.

Es gibt folgende Standardeinschränkungen:

**Einschränkung E (exklusiv)** Höchstens eine der Bedingungen $b_1, \ldots, b_n$ ist erfüllt (siehe Abb. 4.3 links).

**Einschränkung I (inklusiv)**
Mindestens eine der Bedingungen $b_1, \ldots, b_n$ ist erfüllt (siehe Abb. 4.3 Mitte).

**Einschränkung O (oder)**
Eine und nur eine der Bedingungen $b_1, \ldots, b_n$ ist erfüllt (siehe Abb. 4.3 rechts).
($O$ ist erfüllt, wenn E und I gleichzeitig erfüllt sind.)

**Einschränkung R (requires/erfordert)**
Die Erfüllung von Bedingung $a$ erfordert die Erfüllung von Bedingung $b$ (siehe Abbildung 4.4 links).

**Einschränkung M (maskiert)**
Die Erfüllung von Bedingung $a$ impliziert die Nichterfüllung von Bedingung $b$
(siehe Abbildung 4.4 Mitte).
Solche Maskierungen werden meist nur bei den Wirkungen angewandt.

Zu beachten ist, daß beim Vorliegen bestimmter Bedingungen andere Bedingungen irrelevant bzw. nicht anwendbar (weder wahr noch falsch) werden können. Daher wird noch folgende Einschränkung formuliert, die allerdings erst für die Erzeugung konkreter eingabedaten (in Schritt 7) eine Rolle spielt.

**Abb. 4.3: UWG: Exklusive, inklusive und Oder-Einschränkung**
Einschränkung IR (irrelevant)
Die Erfüllung von Bedingung \( a \) impliziert, daß Bedingung \( b \) irrelevant (und nicht anwendbar) ist (siehe Abbildung 4.4 rechts).

Beispiel 4.2.10
Ein Kommando zum Ausdrucken von Dateien habe einen ersten Operanden, der angibt, ob die Datei vom Beginn an auszudrucken ist („START“) oder von einer bestimmten Zeile an („Startzeile“). Dann gibt es z. B. die folgenden Eingabebedingungen bzw. Ursachen:

1. Der erste Operand ist „Startzeile“.
2. Der erste Operand ist „START“.
3. Der Operand „Startzeile“ enthält 1 bis 6 Zeichen.

Da sich die Bedingungen 1 und 2 gegenseitig ausschließen, gilt noch folgende Einschränkung: Wenn Bedingung 2 wahr ist, dann ist die Bedingung 3 nicht anwendbar.
d. h. sie wird in diesem Fall für die spätere Generierung der Eingabedaten nicht mehr betrachtet (siehe Abbildung 4.5).

Beispiel 4.2.11
Das Beispiel des Textformatierers aus Kapitel 1.2 ist schon ein „bearbeitbares Stück“, daher entfällt Schritt 1. Die Eingabebedingungen (Ursachen) für Schritt 2 sind schon in Beispiel 4.2.9 angegeben worden:

\[(B1): z \in \{BL, NL, EOF\}\]
\[(B2): \text{Länge}(w) \leq \text{MAX POS}\]
\[(B3): \text{Länge}(w) \leq \text{MAX POS} - f - 1\]
\[(B4): f > 0.\]

Dabei ist \(z\) das aktuell eingelesene Zeichen, \(w\) das aktuell aufgebaute Wort, \(f\) die Anzahl der Zeichen in der aktuellen Ausgabezeile und MAXPOS die maximale Länge der Ausgabezeile.

Folgende Wirkungen können aus der Beschreibung aus Kapitel 1.2 und der Präzisierung in Anhang A.3 identifiziert werden (wenn angenommen wird, daß die eingelesenen Zeichen in einem Puffer zu einem Wort zusammengesetzt werden)\(^8\):

- \(W_1 =\) Abbruch, da Wort zu lang
- \(W_2 =\) Wort im Puffer aufbauen
- \(W_3 =\) Wort in aktuelle Ausgabezeile einfügen
- \(W_4 =\) Wort in neue Ausgabezeile einfügen

Schritt 3 ergibt einen Ursache/Wirkungsgraphen, der sich mit folgenden Formeln beschreiben läßt; wobei „nicht“ durch „¬“ und „oder“ durch „\(\lor\)“ und „und“ durch „\(\land\)“ dargestellt werden:

\[
\begin{align*}
W_1 &= \neg B_2 \\
W_2 &= \neg B_1 \land B_2 \\
W_3 &= B_1 \land [(B_3 \land B_4) \lor (B_2 \land \neg B_4)] \\
W_4 &= B_1 \land B_2 \land \neg B_3 \land B_4
\end{align*}
\]

Bei Schritt 4 wird festgestellt, daß zwischen den Ursachen \(B_3\) und \(B_2\) eine Einschränkung \(R\) (requires/erfordert) besteht. Damit ergibt sich der in Abbildung 4.6 dargestellte UWG, wenn besondere Zwischenknoten für die Negation weggelassen werden. (Ende Beispiel 4.2.11)

\(^8\)Dies ist eine vereinfachte Beschreibung. Es fehlen die Wirkungen zu den Punkten 2, 3 und 6 bei der Lösung zu Testaufgabe 3 (siehe Anhang A.3 und Übungsaufgabe 4.5).
4.2 Systematisches datenbereichsbezogenes Testen


Dazu werden im folgenden (nur) die fehlerhaft implementierten Eingabebedingungen (Ursachen des UWG) \(^9\) ermittelt. Die Bedingung B2 \((\text{Länge}(w) \leq \text{MAX POS})\) könnte z. B. fälschlicherweise als \(\text{Länge}(w) < \text{MAX POS}\) implementiert worden sein, d. h. für \(\text{Länge}(w) = \text{MAX POS}\) liefert die korrekte Bedingung B2 den Wert „wahr“ (im folgenden als „1“ bezeichnet), die fehlerhaft implementierte Bedingung „\(\text{Länge}(w) < \text{MAX POS}\)“ aber den Wert „falsch“ bzw. „0“. Im folgenden wird der Fall „Bedingung im fehlerfreien Fall 1 und im fehlerhaften Fall 0 mit \(D\) sowie der Fall „Bedingung im fehlerfreien Fall 1 und im fehlerhaften Fall 1“ mit \(\overline{D}\) bezeichnet.

In Schritt 5 müssen daher die (fehler-)sensitiven Testfälle (Belegungen der Ursachen mit Wahrheitswerten) ermittelt werden. Das sind Testfälle, die Fehler der Art \(D\) und \(\overline{D}\) an den Ursachen einstellen und bei der betrachteten Wirkung beobachtbar machen. Ein Fehler \(D\) wird dabei durch Anlegen des logischen Wertes 1, ein Fehler \(\overline{D}\) durch Anlegen einer 0 eingestellt.

Wenn die betrachtete Ursache ein Eingang eines Knotens \(k\) ist, müssen die anderen Eingänge von \(k\) evtl. bestimmte logische Werte annehmen (sensitivierend genannt), damit ein Fehler \(D\) oder \(\overline{D}\) am Ausgang von \(k\) beobachtbar ist. Bei einem

\(^9\)Mit der vorgestellten Methode können auch falsch implementierte logische Verknüpfungen, z. B. UND statt ODER, zum Teil (aber nicht hundertprozentig) erkannt werden.
UND-Knoten mit einem Fehler $D$ oder $\bar{D}$ an einem Eingang müssen z. B. alle anderen Eingänge den Wert 1 annehmen, damit am Ausgang wieder ein Fehler ($D$ bzw. $\bar{D}$) beobachtbar wird. Falls der Ausgang des Knotens $k$ keine Wirkung (sondern nur eine Zwischenbedingung) ist, müssen evtl. weitere Belegungen von Ursachen vorgenommen werden, damit der Fehler schließlich bei einer Wirkung beobachtbar ist. Dabei sind für eine Ursache alle davon betroffenen Wirkungen zu betrachten, da eine Ursache für verschiedene Wirkungen auf unterschiedlichste Art fehlerhaft (oder korrekt) implementiert sein kann.

Wenn der Ursache/Wirkungsgraph für eine Wirkung ein Baum ist\(^{11}\), können die erforderlichen Testfälle (Belegungen der Ursachen mit den logischen Werten 0 oder 1) ohne Probleme gefunden werden. Im anderen Fall kann es Widersprüche bei den erforderlichen sensitivierenden Bedingungen oder Widersprüche zu den Restriktionen geben, die aufgelöst werden müssen.

Zusammenfassung der Regeln von Schritt 5:
Für jede Ursache $a$ und jede Wirkung $w$ des UWG ist folgendes zu erledigen:

1. Einstellen der Fehler $D$ und $\bar{D}$ an der Ursache.

2. Ermittlung eines Pfades $p$ von der Ursache $a$ zu der Wirkung $w$ im UWG.

3. Für alle Knoten auf dem Pfad $p$ mindestens eine sensitivierende Belegung der Ursachen finden, so daß die Fehler $D$ und $\bar{D}$ aus Schritt 1 an der Wirkung $w$ beobachtbar sind. (Falls es mehrere sensitivierende Belegungen für $a$ und $w$ gibt, sollte die Auswahl so erfolgen, daß in Schritt 6 eine minimale Anzahl von Testfällen für alle Ursachen und Wirkungen ermittelt werden kann).

Regeln für die Ermittlung der fehlererstellenden und sensitivierenden Belegungen für spezielle Knoten

1. UND-Knoten mit $n$ Eingängen $x_1, x_2, \ldots, x_n$ und Ausgang $y$:

   Vorüberlegung:

   Sei $i$ ein beliebiger Wert zwischen 1 und $n$.
   Wenn $x_i = D$ und $x_j = 1$ für alle $j \neq i$, dann gilt $y = D$.
   Wenn $x_i = \bar{D}$ und $x_j = 1$ für alle $j \neq i$, dann gilt $y = \bar{D}$.
   Wenn $x_i = D$ oder $x_i = \bar{D}$ und $x_j = 0$ für mindestens ein $j \neq i$, dann gilt $y = 0$.

\(^{10}\) Da diese Betrachtung längs eines Pfades im Ursache/Wirkungsgraphen von der Ursache bis zu einer Wirkung erfolgt, heißt die Methode auch Pfad-Sensitivierung.

\(^{11}\) Der keine Restriktionen zwischen Ursachen oder Wirkungen aufweist (vgl. Schritt 4)
4.2 Systematisches datenbereichsbezogenes Testen

Also sind die Belegungen in Tabelle 4.7 zu wählen.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>...</th>
<th>$x_n$</th>
<th>Eigenschaft</th>
<th>$y$ (fehlerfrei)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>1</td>
<td>alle Eingänge 1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>1</td>
<td>genau ein</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>...</td>
<td>1</td>
<td>Eingang 0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>...</td>
<td>1</td>
<td>alle</td>
<td>0</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>...</td>
<td>:</td>
<td>anderen</td>
<td>:</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>0</td>
<td>Eingänge 1</td>
<td>0</td>
</tr>
</tbody>
</table>

Tab. 4.7 Sensitive Belegungen für den UND-Knoten

Für jeden Eingang wird bei der ersten Belegung ein Fehler $D$ am Ausgang beobachtbar (ehlerfrei = 1, fehlerhaft = 0). Für die anderen Belegungen wird ein Fehler $D$ genau für den Eingang, der mit 0 belegt wird, am Ausgang beobachtbar.

Anstelle von $2^n$ Eingangskombinationen sind also nur $n + 1$ Testfälle erforderlich.

2. ODER-Knoten mit $n$ Eingängen $x_1, x_2, \ldots, x_n$ und Ausgang $y$:

Vorüberlegung:

Es gilt $x_1 \lor x_2 \lor \cdots \lor x_n = \neg (\neg x_1 \land \neg x_2 \land \cdots \land \neg x_n)$

Also gilt beispielsweise für $n = 2$:

$x_1 \lor x_2 = \neg (\neg x_1 \land \neg x_2) = 1 \iff \neg x_1 \lor \neg x_2 = 1$

$x_1 \lor x_2 = 0 \iff \neg x_1 \land \neg x_2 = 1$

Aussagen über die UND-Verknüpfung werden also zu Aussagen über die ODER-Verknüpfung, wenn man UND durch ODER, 0 durch 1 und 1 durch 0 ersetzt (Dualitätsprinzip).

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>...</th>
<th>$x_n$</th>
<th>Eigenschaft</th>
<th>$y$ (fehlerfrei)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>alle Eingänge 1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>genau ein</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>Eingang 1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>...</td>
<td>0</td>
<td>alle</td>
<td>1</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>...</td>
<td>:</td>
<td>anderen</td>
<td>:</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>1</td>
<td>Eingänge 0</td>
<td>1</td>
</tr>
</tbody>
</table>

Tab. 4.8 Sensitive Belegungen für den ODER-Knoten

Also sind bei den ODER-Knoten die Belegungen in Tabelle 4.8 zu wählen. Für jeden Eingang wird bei der ersten Belegung ein Fehler $D$ am Ausgang beobachtbar (ehlerfrei = 0, fehlerhaft = 1). Für die anderen Belegungen wird ein Fehler $D$ genau für den Eingang, der mit 1 belegt wird, am Ausgang beobachtbar.
Anstelle von $2^n$ Eingangskombinationen sind also — wie beim UND-Knoten — nur $n + 1$ Testfälle erforderlich.

3. **NEGATION**  
Es gilt $\neg D = \overline{D}$, $\neg\overline{D} = D$.  
Für den $D$-Fehler muß der Eingangswert 1, für den $\overline{D}$-Fehler der Eingangswert 0 angelegt werden. Es kann also keiner der beiden Testfälle 0 und 1 eingespart werden.

4. **IDENTITÄT**  
Es gilt $id(D) = D$, $id(\overline{D}) = \overline{D}$.  
Wie bei der Negation muß also für den $D$- und $\overline{D}$-Fehler am Eingang der Wert 1 bzw. 0 angelegt werden; am Ausgang erscheint allerdings derselbe Fehler (und nicht z. B. $\overline{D}$ statt $D$ wie bei der NEGATION). Es kann also ebenfalls keiner der beiden Testfälle 0 und 1 eingespart werden.

Zusammenfassung der Regeln für spezielle Knoten:
Bei einem UND-Knoten (siehe Abbildung 4.7 links) gilt:

- wenn $b = 1$ sein soll:  
  der Testfall „$a_1 \cdots a_n = 1$“ ist anzulegen,

- wenn $b = 0$ sein soll:  
  für jedes $i$ ($i = 1, \ldots, n$) ist der Testfall „$a_i = 0, a_j = 1$ für alle $j \neq i$“ anzulegen.

Bei einem ODER-Knoten (siehe Abbildung 4.7 rechts) gilt:

- wenn $b = 1$ sein soll:  
  für jedes $i$ ($i = 1, \ldots, n$) ist der Testfall „$a_i = 1, a_j = 0$ für alle $j \neq i$“ anzulegen.

- wenn $b = 0$ sein soll: der Testfall „$a_1 \cdots a_n = 0$“ ist anzulegen.
Für jeden Knoten müssen die Ausgänge 1 und 0 betrachtet werden. Bei der IDENTITÄT und NEGATION sind jeweils die Testfälle 0 und 1 am Eingang anzulegen.

Beispiel 4.2.12
Für den UWG des Textformatiers aus Beispiel 4.2.11 und Abbildung 4.6 sind folgende Testfälle erforderlich\textsuperscript{12}.

$W_1 = \neg B_2$:
Damit mögliche Fehler bei $B_2$ auffallen, muß $B_2$ beide Wahrheitswerte annnehmen (s. Testfälle $t_1, t_2$ in Tabelle 4.14 auf S. 97).

$W_2 = \neg B_1 \land B_2$:
Damit Fehler bei $B_1$ beobachtbar sind (bei $W_2$), muß $B_2 = 1$ sein und $B_1$ beide Werte annehmen. Umgekehrt sind Fehler bei $B_2$ nur beobachtbar, wenn $B_1 = 0$ und somit $\neg B_1 = 1$ ist und $B_2$ beide Werte annimmt. Das ergibt insgesamt die drei Testfälle $t_3, t_4, t_5$ aus Tabelle 4.14.

$W_3 = B_1 \land Z_3$, $Z_3 = Z_1 \lor Z_2$, $Z_1 = B_3 \land B_4$, $Z_2 = B_2 \land \neg B_4$:
Fehler bei $B_1$ sind nur zu erkennen, wenn $Z_3 = 1$ ist. Dazu kann eine beliebige Kombination $Z_1, Z_2$ gewählt werden (außer $Z_1 = Z_2 = 0$, was $Z_3 = 0$ bewirkt), die wiederum durch beliebige (aber passende) Werte von $B_2, B_3$ und $B_4$ einzuhalten sind.

Das ergibt die beiden Testfälle $T_{1,0}$ und $T_{1,1}$ von Tabelle 4.9 mit den angegebenen Alternativen $t_{6a}, t_{6b}$ bzw. $t_{7a}, t_{7b}$, wobei im folgenden die Kombinationen weggelassen werden, die zu Widersprüchen bei Ursachen oder zur Verletzung der Restriktion $(R)$ für $B_2$ und $B_3$ ($B_3 = 1$, $B_2 = 0$ verboten) führen.

<table>
<thead>
<tr>
<th>Testfall</th>
<th>$W_3 = B_1 \land Z_3$</th>
<th>$Z_3 = Z_1 \lor Z_2$</th>
<th>$Z_1 = B_3 \land B_4$</th>
<th>$Z_2 = B_2 \land \neg B_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{6a}$</td>
<td>$T_{1,0}$</td>
<td>0</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>$t_{6b}$</td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$t_{7a}$</td>
<td>$T_{1,1}$</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>$t_{7b}$</td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Tab. 4.9 Testfälle für $W_3$ bezgl. $B_1$

Fehler bei $B_2$ werden nur von $Z_2$ propagiert, wenn $\neg B_4 = 1$, also $B_4 = 0$ ist. Damit $Z_3$ den Fehler weiterleitet, muß $Z_1 = 0$ sein, damit $W_3$ den Fehler erkennt, muß $B_1 = 1$ sein. Wegen der Wahlmöglichkeiten für die Einstellung von $Z_1$

\textsuperscript{12}Die folgenden Abschnitte sind nur mit großer Konzentration nachvollziehbar. Dabei ist das in den Tabellen 4.7 und 4.8 und den Punkten 3 und 4 auf Seite 94 angegebene Sensitivierungsprinzip schrittweise auf die entsprechenden (Toler-)Formeln anzuwenden, die jeweils in den Kopfzeilen der Tabellen 4.9 bis 4.13 angegeben sind. Die Kompliziertheit (und damit Fehleranfälligkeit) der manuellen Berechnung unterstreicht die Wichtigkeit des Einsatzes eines Werkzeugs für die entsprechenden Berechnungen (vgl. [Bis 97]).
= 0 und der Restriktion R für B2 und B3 ergibt das die beiden Testfälle in Tabelle 4.10 mit den Alternativen \( t_{0a}, t_{0b} \) beim zweiten Testfall \( T_{2,1} \).

\[
\begin{array}{c|cc|ccc}
\text{Testfall} & Z2 = B2 \land \lnot B4 & \text{obligatorisch} & Z1 & B1 & B3 & B4 \\
\hline
\hline
\text{t}_{0a} & T_{0a} & 0 & 1 & 0 & 1 & 0 & 0 \\
\text{t}_{0b} & T_{0b} & 1 & 0 & 1 & 0 & 0 & 0 \\
\end{array}
\]

**Tab. 4.10** Testfälle für W3 bezüglich B2

Fehler bei B3 werden von Z1 propagiert, wenn B4 = 1 ist. Damit Z3 den Fehler weiterleitet, muß Z2 = 0 sein. W3 erkennt schließlich den Fehler, wenn zusätzlich B1 = 1 gilt.

Also gibt es die beiden Testfälle \( T_{3,0} \) und \( T_{3,1} \) in Tabelle 4.11 (mit den Alternativen \( t_{10a}, t_{10b} \) bei Testfall \( T_{3,0} \)), da Z2 = 0 beliebig eingestellt werden kann (unter Beachtung der Restriktion R für B2 und B3).

\[
\begin{array}{c|cc|cc}
\text{Testfall} & Z1 = B3 \land B4 & \text{obligatorisch} & Z2 = B2 \land \lnot B4 & \\
\hline
\hline
\text{t}_{10a} & T_{10a} & 0 & 1 & 0 & 1 & 0 & 0 \\
\text{t}_{10b} & T_{10b} & 1 & 0 & 1 & 0 & 0 & 0 \\
\end{array}
\]

**Tab. 4.11** Testfälle für W3 bezüglich B3


\[
\begin{array}{c|cc|cc|cc}
\text{Testfall} & Z2 = B2 \land \lnot B4 & \text{obligatorisch} & Z1 = B3 \land B4 & \\
\hline
\hline
\text{t}_{10} & T_{4,0} & 0 & 1 & 0 & 1 & 0 & 0 \\
\text{t}_{0a} & T_{4,1} & 1 & 0 & 1 & 0 & 0 & 0 \\
\end{array}
\]

**Tab. 4.12** Testfälle für W3 bezüglich B4 und Z2

Das ergibt für die Propagierung über Z2 die Testfälle aus Tabelle 4.12. Der Testfall \( B1 = B2 = B3 = 1, B4 = 0 \) würde zwar den Fehler \( \overline{D} \) bei B4 einstellen, aber auch den Fehler \( D \) bei \( \lnot B4 \). Daher ergibt sich \( Z1 = \overline{D}, Z2 = D \), also \( Z3 = \overline{D} \lor D = 1 \), und damit \( W3 = 1 \). Das ist der korrekte Wert, d. h. der Fehler wurde nicht propagiert. Daher ist nur der angegebene Testfall \( t_{0a} \) (\( B1 = B2 = 1, B3 = B4 = 0 \)) in der Lage, den Fehler \( \overline{D} \) bei B4 über Z2 nach W3 zu propagieren.

Es gibt keine zulässigen Testfälle, die den Fehler bei B4 über Z1 nach W3 propagieren können. Für den möglichen Testfall \( B1 = B2 = B3 = B4 = 1 \) gilt nämlich:
die Fehler \( D \) bei \( B_4 \) und \( \overline{D} \) bei \( \neg B_4 \) werden eingestellt. Es ergibt sich: \( Z_1 = D, Z_2 = \overline{D}, Z_3 = D \lor \overline{D} = 1 \), also \( W_3 = 1 \), der korrekte Wert.

\( W_4 = B_1 \land B_2 \land \neg B_3 \land B_4: \)

Da es sich nur um eine UND-Verknüpfung ohne Zwischenknoten handelt, reichen folgende Testfälle aus: \( B_1 = B_2 = \neg B_3 = B_4 = 1 \) entdeckt alle Fehler, bei denen einer der Eingänge von 1 auf 0 wechselt. Die vier Kombinationen, bei denen ein Eingang auf 0 gesetzt wird und alle anderen auf 1, entdecken jeweils den Wechsel des speziellen Eingangs von 0 auf 1. Also ergibt sich Tabelle 4.13. Dabei ist nur \( t_{11} \) ein neuer Testfall, alle anderen wurden schon für \( W_3 \) verwendet.

<table>
<thead>
<tr>
<th>Testfall</th>
<th>( B_1 )</th>
<th>( B_2 )</th>
<th>( \neg B_3 )</th>
<th>( B_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_{11b} )</td>
<td>( T_{W4,1} )</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( t_{11} )</td>
<td>( T_{W4,0,1} )</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( t_{10a} )</td>
<td>( T_{W4,0,2} )</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( t_{20} )</td>
<td>( T_{W4,0,3} )</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( t_{30} )</td>
<td>( T_{W4,0,4} )</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Tab. 4.13 Testfälle für \( W_4 \)

Tabelle 4.14 enthält alle ermittelten Testfälle, wobei bei den Testfällen \( t_1 \) bis \( t_{7a} \) nur ein Teil der Eingabebedingungen \( B_1 \) bis \( B_4 \) spezifizierte Werte annehmen muß.

<table>
<thead>
<tr>
<th>Testfälle pro betrachteter Wirkung ( W_1 ) bis ( W_4 )</th>
<th>( W_1 )</th>
<th>( W_2 )</th>
<th>( W_3 ) und ( W_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>( t_2 )</td>
<td>( t_3 )</td>
<td>( t_4 )</td>
</tr>
<tr>
<td>( B_1 )</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( B_2 )</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( B_3 )</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( B_4 )</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

| \( W_1 \) | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| \( W_2 \) | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| \( W_3 \) | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| \( W_4 \) | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Legende: Nur einer der Testfälle \( t_{5a}, t_{5b} \) ist erforderlich. 
Von den für \( W_3 \) alternativen Testfällen \( t_{7a} \) und \( t_{7b} \), \( t_{9a} \) und \( t_{9b} \), sowie \( t_{10a} \) und \( t_{10b} \) (s. Tabellen 4.9, 4.10, 4.11) sind \( t_{7b}, t_{9a}, t_{10a} \) und \( t_{10b} \) für \( W_4 \) erforderlich (s. Tabelle 4.13).

Tab. 4.14 Nicht optimierte Testfälle für den Textformatierer

Schritt 6: Die in Schritt 5 ermittelten Kombinationen von Ursachewerten und die im fehlerfreien Fall zu erwartenden Wirkungen sind nun in eine Wertetabelle einzutragen, wobei Kombinationen für verschiedene Ursachen, die sich nicht widersprechen, zusammengefaßt werden sollen.
Beispiel 4.2.13
Da die Testfälle $t_{76}$ und $t_{95}$ für W4 erforderlich sind (s. Tabelle 4.13), können die Alternativen $t_{76}$ und $t_{95}$ bei der Testfallermittlung für W3 (s. Tab. 4.9 und 4.10) entfallen. Damit ergeben sich folgende Testfälle, die notwendig sind und bei denen alle vier Bedingungen B1 bis B4 spezifizierte Werte haben: $t_{76}$, $t_{8}$, $t_{95}$, $t_{10a}$, $t_{10b}$, $t_{11}$. Die Testfälle, bei denen einige Werte unspezifiziert sind, werden davon z. T. abgedeckt: $t_{1}$ wird von $t_{8}$ abgedeckt (da bei $t_{8}$ ebenfalls $B2 = 0$ gilt), $t_{2}$ wird von $t_{76}$ abgedeckt, $t_{3}$ von $t_{11}$, $t_{5}$ von $t_{10b}$. Damit bleiben nur noch alternative Testfälle $t_{95}$ oder $t_{96}$ und der Testfall $t_{4}$ übrig.

Somit ergibt sich Tabelle 4.15 mit acht notwendigen Testfällen. Dabei sind bei einem Testfall nicht nur die Ergebnisse für die betrachtete Wirkung, sondern für alle Wirkungen eingetragen. Beim $F1$ wurde $B3 = 0$ ergänzt, da $B3 = 1$ im Widerspruch zu $B2 = 0$ und der Restriktion zwischen $B3$ und $B2$ stehen würden.

<table>
<thead>
<tr>
<th>Testfall</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
<th>F8</th>
</tr>
</thead>
<tbody>
<tr>
<td>deckt ab</td>
<td>$t_{4}$</td>
<td>$t_{5}, t_{11}$</td>
<td>$t_{6a}/t_{6b}$</td>
<td>$t_{1}, t_{8}$</td>
<td>$t_{10a}$</td>
<td>$t_{95}$</td>
<td>$t_{5}, t_{10b}$</td>
<td>$t_{2}, t_{76}$</td>
</tr>
<tr>
<td>B1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B3</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B4</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>W1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>W2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>W3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>W4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Tab. 4.15 Optimierte Testfälle für den Textformatierer

Tabelle 4.15 enthält nur acht Testfälle. Wegen Restriktion $R$ zwischen $B3$ und $B2$ (d. h. $B3 = 1$, $B2 = 0$ ist nicht möglich) gibt es statt $2^4 = 16$ nur $16 - 4 = 12$ mögliche Eingabefälle. Es wurden also vier Testfälle eingespart.

Schritt 7: Für jede Spalte der Wertetabelle wird ein Eingabedatum ermittelt. Die Eingabedaten sollen möglichst auch Grenzwerte berücksichtigen.

Beispiel 4.2.14
Zu den acht Testfällen aus Tabelle 4.15 können folgende Testdaten gebildet werden, wobei mehrere Fälle durch einen kompletten Eingabetext erfüllt werden können. Die Fälle $F2$ und $F7$ sowie $F3$ und $F8$ bzw. $F3$ und $F6$ treten paarweise nacheinander auf, d. h. bei $F2$ oder $F3$ wird ein Wort $w$ aufgebaut ($W2 = 1$) und bei anschließender Eingabe eines Trennzeichens $\varepsilon \in \{BL, NL, EOF\}$ wird $w$ in die aktuelle Ausgabezeile ($W3 = 1$) oder eine neue Ausgabezeile ($W4 = 1$) eingefügt. $F6$ behandelt $\varepsilon_{10a}$ und $t_{10b}$ sind für Wirkung $W4$ beide notwendig (s. Tabelle 4.13).
4.3 Systematisches funktionsbezogenes Testen

Die Methoden des funktionsbezogenen Testens gehen davon aus, daß die implemenierte Funktion \( f \) nur unwesentlich von der korrekten Funktion \( f^* \) abweicht. Diese Annahme wird auch **kompetente Programmierer-Hypothese** genannt. \( f^* \) gehört also zu der Menge \( F_j \) von Funktionen, die zu \( f \) ähnlich sind. Wenn eine Testdatenmenge alle Funktionen aus \( F_j \) von \( f \) unterscheiden kann, dann kann sie auch \( f \) von \( f^* \) unterscheiden, d. h. feststellen, daß \( f \) nicht korrekt ist. Formal läßt sich das folgendermaßen beschreiben: zu \( f \) existiert eine (relativ kleine) Menge \( F_j \) von Funktionen mit folgender Eigenschaft:

**q.e.d.**
1. $F_j$ enthält die korrekte Funktion $f^*$.

2. Für $f$ gibt es eine Teststrategie, d. h. für jedes $f' \in F_j$ (mit $f' \neq f$) kann die gewählte Testdatenmenge $T$ $f'$ von $f$ unterscheiden, d. h. es gibt einen Testfall $t$ in $T$, für den gilt: $f(t) \neq f'(t)$.

### 4.3.1 Funktionale Formen


Folgende Kompositionsarten treten auf:

1. alternative Auswahl,
2. kontrollierte Iteration,

**Kompositionsart 1:**

Eine Funktion $f$ der höheren Entwurfs Ebene besteht aus der Zusammenfassung von ähnlichen, aber unabhängigen Funktionen $f_1, \ldots, f_n$ der tieferen Ebene (s. Abb. 4.8).

![Diagramm](image)

**Legende:**

- Kontrollflußübergabe an tiefere Ebene
- Kontrollflußübergabe an höhere Ebene

**Abb. 4.8:** Kompositionsart 1: Alternative Auswahl

**Beispiel 4.3.1**

Auswahl zwischen drei verschiedenen statistischen Verteilungen ($f_1 = \text{Gleichverteilung}$, $f_2 = \text{Exponentialverteilung}$, $f_3 = \text{Poisson-Verteilung}$) durch einen Parameter der Auswahlfunktion $f$ der höheren Ebene.
Im einfachsten Fall \( n = 2 \) gibt es in \( f \) eine Entscheidung zwischen den Teilfunktionen \( f_1 \) und \( f_2 \). Dies läßt sich also darstellen als:

\[
f = \text{if } b \text{ then } f_1 \text{ else } f_2,\]

wobei \( b \) eine Boole'sche Funktion mit den Ergebniswerten \textit{true} und \textit{false} (bzw. \textit{wahr} und \textit{falsch}) ist.

Der Fall \( n > 2 \) läßt sich folgendermaßen auf den Fall \( n = 2 \) zurückführen:

\[
f = \text{if } b \text{ then } f_1 \text{ else } f'
\]

und \( f' \) wählt zwischen \( f_2, \ldots, f_n \) aus, also nur noch zwischen \( n - 1 \) Funktionen.

Durch iterative Anwendung dieser Darstellung kommt man also schließlich zum Fall \( n = 2 \).

Die Funktion \( f \) kann aus folgenden Gründen fehlerhaft sein:

1. \( f_1 \) ist fehlerhaft,
2. \( f_2 \) ist fehlerhaft,
3. \( b \) ist fehlerhaft.

In den Fällen 1 und 2 gilt folgendes: Falls \( f_1 \) oder \( f_2 \) sich wieder als funktionale Form beschreiben lassen, sind auf die entsprechenden Teilfunktionen die gleichen Testverfahren anzuwenden, die in diesem Kapitel dafür vorgeschlagen werden. Im anderen Fall können die Teilfunktionen datenbereichsbezogen, unsystematisch oder — falls schon durch Kontrollfluß, Datenfluß, Ausdrücke oder Anweisungen beschrieben — mit den Methoden von Teil III dieses Buches getestet werden.

Im Falle 3 gilt entsprechendes wie bei den Fällen 1 und 2: Falls die Boole'sche Funktion \( b \) sich als funktionale Form beschreiben läßt, sind die Teilfunktionen nach den Vorschlägen dieses Kapitels zu testen. Andernfalls liegt ein einfacher Boole'scher Ausdruck vor, der nach den Regeln von Teil III, Kapitel 9.1, zu testen ist.

Falls durch hinreichendes Testen ausgeschlossen werden kann, daß \( f_1 \) oder \( f_2 \) fehlerhaft sind, beschreibt der folgende Satz 4.3.1 die Verläßlichkeit des Tests von \( f \) in Abhängigkeit von der Verläßlichkeit des Tests für die Boole'sche Funktion \( b \).

---

14 Programmiertechnisch läßt sich die \( n \)-stellige Fallunterscheidung natürlich auch durch eine \textit{case}-Anweisung beschreiben (falls in der Programmiersprache vorhanden). Das ist aber semantisch äquivalent zur hier verwendeten verschachtelten \textit{if-then-else}-Konstruktion.
Satz 4.3.1
Voraussetzungen:

1. \( f = \text{if } b \text{ then } f_1 \text{ else } f_2. \)
2. \( B_b \) ist eine Menge von Booleschen Funktionen, die denselben Eingabebereich wie \( b \) haben.
3. \( T \) ist eine Testdatenmenge, für die gilt:

   (a) \( T \) unterscheidet jede Boolesche Funktion \( b' \) aus \( B_b \) von \( b \) (falls \( b' \neq b \)),
   (b) \( f_1(t) \neq f_2(t) \) für alle \( t \) aus \( T \).

4. \( F_f \) ist die Menge der Funktionen, die sich aus \( f \) ergibt, indem \( b \) nacheinander durch jedes \( b' \) aus \( B_b \) ersetzt wird.

Unter den Voraussetzungen 1 bis 4 gilt:
\( T \) kann jedes \( f' \) aus \( F_f \) von \( f \) unterscheiden (falls \( f' \neq f \)).

Beweis (durch Widerspruch):
Sei \( f' \neq f \) und \( f'(t) = f(t) \) für alle \( t \in T \). Da \( f' \) aus \( F_f \) ist, gilt wegen 1 und 4:
\( f' = \text{if } b' \text{ then } f_1 \text{ else } f_2 \) für ein \( b' \) aus \( B_b \) mit \( b' \neq b \) (aus \( b' = b \) würde \( f' = f \) folgen).
Nach Voraussetzung 3(a) existiert ein \( t \in T \) mit \( b'(t) \neq b(t) \). Also gilt \( f(t) = f_1(t) \) und \( f'(t) = f_2(t) \) (oder umgekehrt). Wegen 3 (b) gilt also \( f(t) = f_1(t) \neq f_2(t) = f'(t) \), ein Widerspruch zur Annahme.

q.e.d.

Voraussetzung 3(b) von Satz 4.3.1 ist problematisch. Wenn diese Voraussetzung nicht erfüllt ist, kann trotz einer falschen Entscheidungsfunktion \( b \) die falsche Teilfunktion \( f_1 \) (bzw. \( f_2 \)) dennoch „zufälligerweise“ das richtige Ergebnis berechnen. In solchen Fällen ist der Test also nicht zuverlässig. Zur Erläuterung dieses Problems dient folgendes Beispiel:

Beispiel 4.3.2

\( f(x) = \text{if } x \geq 3 \text{ then } x \text{ else } 1 \)

Für Prädikat \( b = (x \geq 3) \) gebe es nur einen Fehler: \( B_b = \{ x \geq 0 \} \). Zur Unterscheidung der Booleschen Funktionen \( b' = (x \geq 0) \) und \( b = (x \geq 3) \) reicht der Test mit \( x = 1 \) aus, da \( b'(1) = \text{true} \) und \( b(1) = \text{false} \). Dieser Test entdeckt aber nicht den Unterschied von \( f \) und \( f' \) mit \( f'(x) = \text{if } x \geq 0 \text{ then } x \text{ else } 1 \) (Konstante im else-Zweig), \( f'(1) = 1 \) (\( x = 1 \) im then-Zweig), d. h. \( f(1) = f'(1) \).
Um diese Testfehler auszuschließen, muß man die Wertebereiche bestimmen, für die beide Teilfunktionen \( f_1 \) und \( f_2 \) identische Werte berechnen, d. h. wo ihre Differenz \( f_1(x) - f_2(x) \) gleich 0 ist. Für beliebige Funktionen ist dies leider unmöglich. Im vorliegenden Beispiel 4.3.2 ist es aber einfach: Die Teilfunktionen \( f_1(x) = x \) und \( f_2(x) = 1 \) sind gleich, wenn \( f_1(x) - f_2(x) = x - 1 = 0 \) gilt, d. h. \( x = 1 \).
Mit \( x = 1 \) darf also nicht getestet werden.

**Kompositionsart 2:**
Eine Boolesche Kontrollfunktion \( b \) steuert einen *iterativen Prozeß*, der sich durch eine Funktion \( g \) beschreiben läßt.
In Abhängigkeit davon, ob die Kontrollfunktion \( b \) vor oder erst nach dem ersten Ausführen der Funktion \( g \) überprüft, ob \( g \) zu iterieren ist, werden zwei Fälle unterschieden (siehe Abbildung 4.9).

![Iterative repeat-Funktion und Iterative while-Funktion](image)

**Abb. 4.9:** Kompositionsart 2: Kontrollierte Iteration

**Beispiel 4.3.3**
Ein Programm soll eine Nullstelle einer Funktion \( h \) mit hinreichender Genauigkeit \( \epsilon \) berechnen. Es ist also ein Intervall \((a, b)\) mit \(|a - b| \leq \epsilon\) zu bestimmen, so daß \( h(x) = 0 \) für ein \( x \) mit \( a \leq x \leq b \) gilt.

Die Berechnung der Funktionswerte von \( h \) geschieht durch eine Funktion \( g \). Das (iterative) Verfahren muß durch eine Kontrollfunktion \( b \) gesteuert werden, welche die Intervallänge für benachbarte Funktionswerte bestimmt.
Dabei ist folgendes zu beachten:

2. Kontrollfunktionen erzeugen keine Werte, die von anderen Funktionen weiterverarbeitet werden (wie bei den Kompositionsarten 1 und 3).
In Programmiersprachenotation lassen sich die beiden Fälle von iterativen Funktionen folgendermaßen darstellen. Dabei sei \( v \) der Vektor aller Variablen, die von den Funktionen \( g \) oder \( b \) referenziert oder definiert werden.

1. \( f = \textbf{repeat } v \leftarrow g(v) \textbf{ until } b(v) \)

2. \( f = \textbf{while } b(v) \textbf{ do } v \leftarrow g(v) \)

Die Funktion \( f \) kann aus folgenden Gründen fehlerhaft sein:

i. \( g \) ist fehlerhaft,

ii. \( b \) ist fehlerhaft,

iii. \( f \) wurde mit \textbf{repeat} (Fall 1) statt mit \textbf{while} (Fall 2) realisiert (oder umgekehrt).

In den ersten beiden Fällen gilt entsprechendes wie bei der Kompositionsart 1 (\textit{if-then-else} bzw. \textit{case}).
Falls durch hinreichendes Testen ausgeschlossen werden kann, daß \( g \) fehlerhaft ist, beschreibt der folgende Satz 4.3.2 die Verläßlichkeit des Tests einer iterativen while-Funktion \( f \) in Abhängigkeit von der Verläßlichkeit des Tests für die Boolesche Funktion \( b \).

**Satz 4.3.2**

**Voraussetzungen:**

1. \( f = \textbf{while } b(v) \textbf{ do } v \leftarrow g(v) \).

2. \( B_k \) ist eine Menge von Booleschen Funktionen, die denselben Eingabebereich wie \( b \) haben.

3. \( T \) ist eine Testdatenmenge, für die gilt:

   a) \( T \) unterscheidet jede Boolesche Funktion \( b' \) aus \( B_k \) von \( b \) (wenn \( b' \neq b \)),

   b) für alle \( t \) aus \( T \) und alle \( i, j \) mit \( i > j \geq 0 \) gilt:

\[
g^i(t) \neq g^j(t),
\]

wobei \( g^i \) die \( i \)-fache Iteration von \( g \) ist und \( g^0 \) die Identität ist.

4. \( F_f \) ist die Menge der Funktionen, die sich aus \( f \) ergibt, indem \( b \) durch ein \( b' \) aus \( B_k \) ersetzt wird.

**Unter den Voraussetzungen 1 bis 4 gilt:**

\( T \) kann jedes \( f' \) aus \( F_f \) von \( f \) unterscheiden (falls \( f' \neq f \)).
Beweis:
Sei $f' = \textbf{while } b'(v) \textbf{ do } v \leftarrow g(v) \textbf{ und } b' \neq b$. Dann gibt es wegen 3(a) ein Testdatum $t$ aus $T$ mit $b'(t) \neq b(t)$. Sei $b(t) = b(g^0(t)) = \text{true}$ und $b'(t) = \text{false}$. (Für $b(t) = \text{false}, b'(t) = \text{true}$ ist der Beweis analog.) Dann berechnet $f'(t)$ den Wert $t = g^0(t)$. $f(t)$ ist entweder undefiniert, (falls die while-Schleife unendlich oft durchlaufen wird), also verschieden von $f'(t)$, oder $f$ terminiert nach $n > 0$ Iterationen. Dann berechnet $f(t)$ den Wert $g^n(t)$. Also gilt wegen 3(b): $f(t) = g^n(t) \neq g^0(t) = f'(t)$.
q.e.d.

Im Unterschied zu Satz 4.3.1 hat $g$ eine neue Voraussetzung zu erfüllen. Diese Voraussetzung 3(b) bedeutet, daß bei jeder Iteration von $g$ ein neuer Wert berechnet wird, die Iteration also nicht nutzlos ist.

Für iterative repeat-Funktionen oder Versachlichungen von iterativen repeat- und while-Funktionen ist die Testbedingung komplizierter. Dies liegt daran, daß bei repeat-Funktionen der Effekt der mindestens einmaligen Ausführung der Funktion $g$ einzubeziehen ist (genauerer siehe [How 87], Theorem 4.15).

Kompositionsart 3:
Eine Funktion $f$ wird realisiert durch die Sequenz (Hintereinanderausführung) von Funktionen $f_1, \ldots, f_n$ (siehe Abbildung 4.10).

![Diagramm](image)

**Abb. 4.10:** Kompositionsart 3: Sequenz

Beispiel 4.3.4
Ein Programm für die numerische Analyse hat eine Zahlenfolge als Eingabe. Die Zahlenfolge wird zuerst von einer Funktion $f_1$ bearbeitet, die die Werte normalisiert, und dann von weiteren Funktionen $f_2, f_3, \ldots$ bearbeitet.

Die Funktion $f$ kann aus folgenden Gründen fehlerhaft sein:

1. Mindestens eine der Funktionen $f_i, i = 1, \ldots, n$, ist fehlerhaft.
2. In der Sequenz $f_1, \ldots, f_n$ fehlt eine Funktion $f'$.
3. In der Sequenz $f_1, \ldots, f_n$ ist eine Funktion $f_i$ zu viel eingebaut worden, d. h. $f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_n$ ist die korrekte Sequenz.
4. In der Sequenz \( f_1, \ldots, f_n \) ist die Reihenfolge der Funktionen falsch, d. h. für eine
Permutation \( p : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\} \) gilt:
\( f_{p(1)}, \ldots, f_{p(n)} \) ist die korrekte Sequenz.

Im ersten Fall gilt Entsprechendes wie bei den anderen Kompositionsarten.
In den Fällen 2 und 3 sollte dieser Fehler durch statische Überprüfung des Entwurfs
festgestellt werden können (siehe Kapitel 12) oder durch Testen der Gesamtfunktion
\( f \) beim Integrationstest (siehe Kapitel 13.3). Der Fall 4 ist ein Spezialfall (der
Verletzung) von Reihenfolgebedingungen, die in Kapitel 5.2 behandelt werden.

Zu den Testmethoden kann noch folgendes angemerkt werden:

1. Fehler in den Kontrollfunktionen bei der kontrollierten Iteration (Kompositi-
onsart 2) können durch funktionsorientiertes Testen auf der Systemebene nicht
immer gefunden werden. Wenn beispielsweise die Kontrollfunktion die Itera-

tion nicht zum frühestmöglichen Zeitpunkt abbricht, ist das Ergebnis bei einem
konvergierenden Verfahren trotzdem richtig. Die Funktion ist lediglich ineffizient
realisiert worden.

2. Interne — im Entwurf erzeugte — Variablen und Datenstrukturen können Feh-
lerquellen sein, die bei einem groben funktionsorientierten Testen (auf der Syste-
mebene) nicht erfasst werden.

4.3.2 Entwurf orientiertes Testen

Die Bemerkungen 1 und 2 sind eine erste Begründung für die Notwendigkeit des
entwurf orientierten funktionsbezogenen Testens. Dieses Vorgehen hat einen wei-

tereren Vorteil: Funktionen werden im Kontext von Einschränkungen getestet und
daher werden die Wertebereiche bzw. Kombinationen eingeschränkt. Der Testauf-

wand ist also geringer als bei einem „blinden“ Durchprobieren aller Möglichkeiten.
(Wenn einzelne Module allerdings in einem anderen Zusammenhang wiederverwen-
det werden sollen, dürfen diese Einschränkungen beim Testen nicht vorgenommen
werden. Wiederverwendbarkeit hat also ihren Preis.)

Beispiel 4.3.5

Die Entwurfsfunktion \( f \) sei durch ein Codestück \( P \) realisiert, welches nur ausgeführt
wird, wenn \( x \leq 2 \) ist. Der Wertebereich der Eingabvariable \( x \) von \( P \) sei das Intervall
\([-MAXINT, MAXINT]\).

Würde \( P \) unabhängig vom Kontext (\( x \leq 2 \)) getestet, so wären \( x = +MAXINT \) und \( x = -MAXINT \) die extremen Testeingabewerte. Im Kontext \( x \leq 2 \) sind die

extremen (gültigen) Werte dagegen \( x = -MAXINT \) und \( x = 2 \) und es gibt einen
ungültigen Grenzwert \( x = 2 + \epsilon \), wobei \( \epsilon \) der kleinste positive Wert ist.

Folgende Hinweise sind beim entwurf orientierten funktionsbezogenen Testen zu be-
achten:
1. Voraussetzung für diese Testart ist, daß für die Eingabe- und AusgabevARIABLEN jeder Entwurfsfunktion Wertebereich, Länge, Dimension etc. vollständig spezifiziert sind.

2. Damit die Entwurfsfunktionen getestet werden können, müssen sie i. allg. aus ihrer Umgebung im Programm herausgelöst und mit passenden Werten für ihre Eingabedatenstrukturen versorgt werden.


Für die Aufgaben 2 und 3 sind entsprechende Methoden und Werkzeuge wie beim Modultest nützlich (genaueres siehe Kapitel 13.3, insbesondere Beispiel 13.3.1).

4.3.3 Testen auf der Basis von Datenflußdiagrammen, Petri-Netzen und endlichen Automaten

Eine alternative Vorgehensweise beim absteigenden (top-down) Entwurf bzw. bei der schrittweisen Verfeinerung von Systemen besteht in der Verwendung von SADT-Diagrammen, SA-Diagrammen, Petri-Netzen, endlichen Automaten oder daraus weiter entwickelten Modellen\(^\text{15}\).

Die SA-Diagramme geben in Form von Graphen das Zusammenspiel von Funktionen (Aktivitäten genannt) an, die auf Datenflüsse operieren. Die Aktivitäten werden durch Knoten des Graphen dargestellt, die den Namen der Aktivität tragen, die Datenflüsse werden durch die Kanten des Graphen dargestellt und haben ebenfalls Namen.


**Beispiel 4.3.6**

Bei der Flugsicherung fallen u. a. die in Abbildung 4.11 dargestellten Daten an. Die Kante mit Namen „Flughöhe“ verzweigt sich in zwei Kanten mit den Namen „Flughöhe“ und „Horizontalposition“.

---

\(^{15}\)SADT = Structured Analysis and Design Technique, siehe z. B. [Ros 77]; SA = Structured Analysis, siehe z. B. [DeM 79]
Um die Bedeutung einer Verzweigung genau festzulegen, muß etwa folgendes angegeben werden:

1. der Wertebereich der Daten (d. h. für jede Kante bzw. jeden Zweig ein Wertebereich),
2. eine Vorschrift, wie ein Wert auf einer Kante aufgeteilt bzw. transformiert wird zu Werten auf den einzelnen Zweigen.

**Beispiel 4.3.7**
Für die Verzweigung des Flugstreifens aus Abbildung 4.11 sei angegeben:

1. ein Wertebereich durch Aufzählung seiner Elemente:

   **Wertebereich von Flugstreifen:**
   inaktiv,
   Flugkurs\_ohne\_Konflikt/Flughöhe\_nicht\_besetzt,
   Flugkurs\_ohne\_Konflikt/Horizontalposition,
   Flugkurs\_im\_Konflikt\_mit\_anderem\_Flugkurs

2. eine Vorschrift für die Werteweiterleitung:

   ```
   case
   Flugstreifen = Flugkurs\_ohne\_Konflikt/Flughöhe\_nicht\_besetzt
   then
   Horizontalposition := indifferent;
   Flughöhe := nicht\_besetzt
   case ... (etc.)
   ```
Um die Bedeutung der Aktivitäten (Funktionen) festzulegen, muß folgendes angegeben werden (jedenfalls für die Aktivitäten auf der untersten Verfeinerungsstufe):

1. **Vorbedingungen** für die Werte von allen relevanten Eingabedaten, Kontrolldaten und Mechanismen,

2. **Nachbedingungen** für die Werte von Ausgabedaten, d. h. die Werte auf ausgehenden Kanten der Aktivität.

Aus diesen Angaben läßt sich das Gesamtverhalten des spezifizierten Systems folgendermaßen als *(endlicher)* **Automat** angeben.


Die Verhaltensbeschreibung setzt zusätzlich die Angabe eines Anfangszustands oder mehrerer möglicher Anfangszustände voraus. Ausgehend von einem Zustand \( z \) ergibt sich aufgrund externer Ereignisse \( e \) eine Veränderung der Vorbedingungen von gewissen Aktivitäten. Dies ergibt aufgrund der Bedeutung der Aktivitäten eine Veränderung der Werte auf den Ausgangskanten der einzelnen Aktivitäten — spezifiziert durch die Nachbedingungen zu den erfüllten Vorbedingungen der Aktivität.

Diese Veränderungen innerhalb des Gesamtsystems können nacheinander oder parallel für verschiedene Aktivitäten erfolgen, wobei sich eventuell ein nichtdeterministisches Verhalten ergibt. Insgesamt führen diese Veränderungen zu einem neuen Systemzustand \( z' \) und neuen Werten auf den externen Ausgangskanten des Systems (das Tupel dieser Werte sei \( a \)).

Diese Veränderung des Systemzustands läßt sich im Automat notieren: ausgehend vom Systemzustand \( z \) ergibt sich aufgrund der externen Ereignisse \( e \) eine „Transition“ zum neuen Systemzustand \( z' \) mit der Ausgabe \( a \).

Diese Beschreibung als Automat kann nun als Ausgangspunkt für das Testen des Systemverhaltens verwendet werden. Dazu werden Tests verwendet, die gewisse „Wege“ durch den Automaten ausführen, d. h. eine Folge von Transitionen des Automaten. Da die Menge aller Wege durch den Automaten i. allg. unendlich ist, sind endlich viele Fälle auszuwählen. Das sinnvolle Auswählen solcher Fälle wird in Kapitel 5 im Zusammenhang mit den (endlichen) Automaten, die aus Pfadausdrücken abgeleitet werden, behandelt.

Im Unterschied zu SADT- oder SA-Diagrammen werden bei einem **Petri-Netz** die Datenflüsse durch **Stellen** und die Aktivitäten durch **Transitionen** beschrieben. Durch Modifikationen der ursprünglichen Petri-Netz-Definition können komplexer...
Datenstrukturen und Aktivitäten beschrieben werden. Vorteilhaft ist, daß alle Petri-Netz-Definitionen ein exaktes Verhalten der Petri-Netze vorgeben, welches entsprechend gegen die Anforderungen getestet werden kann. (Zu den Testkriterien sei auf die Literatur, z. B. [MoP 90], verwiesen.)

Beispiel 4.3.8
Das Datenflußdiagramm aus Abbildung 4.12 wird durch das Petri-Netz aus Abbildung 4.13 dargestellt.

Abb. 4.12: Datenflußdiagramm

Abb. 4.13: Petri-Netz
4.4 Übungen

Übung 4.1:
Gegeben sei ein Programm, welches die Lösungen der quadratischen Gleichung \( a \cdot x^2 + b \cdot x + c = 0 \) bestimmt, wobei \( a, b, c \) Konstanten vom Typ `Integer` sind und \( x \) die Variable. Geben Sie Äquivalenzklassen und Grenzwerte an (in Abhängigkeit von den Werten \( a, b \) und \( c \)) und entsprechende Testdaten.
Hinweis: Wird z. B. \( a = 0 \) gewählt, ergibt sich eine lineare Gleichung. Für andere Wertekombinationen von \( a, b \) und \( c \) gibt es statt der normalen zwei Lösungen (für \( x \)) evtl. nur eine oder keine reellwertige Lösung.

Übung 4.2:
Gegeben sei eine Prozedur `Zaehle_Zeichen`, die Zeichen von der Tastatur einliest, und zwar solange bis ein Zeichen erkannt wird, das kein Großbuchstabe ist, oder `Gesamtzahl` den Wert `Max` erreicht, d. h. den größten Wert, der durch den Datentyp `Integer` darstellbar ist.
Identifizieren Sie drei Ursachen und vier Wirkungen für `Zaehle_Zeichen` und stellen Sie den UWG und die Wertetabelle für die Testdaten auf.

Übung 4.3:
Geben Sie für das Dreiecksklassifikationsprogramm von Seite 17 einen Ursache/Wirkungsgraphen an. Orientieren Sie sich dabei an den Bedingungen (Ursachen) und Ausgaben (Wirkungen) von Tabelle A.1 im Anhang A.1.
Leiten Sie entsprechende Testdaten ab und vergleichen Sie die Lösung mit den Ergebnissen im Anhang A.

Übung 4.4:
Mit der UWG-Methode wurden acht Testfälle F1 bis F8 für den Textformatierer bestimmt (s. Tabelle 4.15 auf S. 98) und somit vier Testfälle eingespart. Zeigen Sie, daß bei den Wahlen \( B_1 = B_2 = B_3 = B_4 = 0 \) für Testfall F1 (d. h. \( B_4 = 0 \)), und \( t_{eb} \) für F3 (d. h. \( B_1 = 0, B_2 = B_3 = B_4 = 1 \)) die vier Testfälle \( u_1, u_2, u_3, u_4 \) überflüssig sind:

\[
\begin{align*}
  u_1: & \quad B_1 = B_2 = B_3 = 0, \quad B_4 = 1 \\
  u_2: & \quad B_1 = 0, \quad B_2 = 1, \quad B_3 = B_4 = 0 \\
  u_3: & \quad B_1 = 0, \quad B_2 = B_3 = 1, \quad B_4 = 0 \\
  u_4: & \quad B_1 = B_2 = B_3 = 1, \quad B_4 = 0
\end{align*}
\]

Es ist dabei zu zeigen, daß ein Fehler in einer Eingabebedingung \( B_i, i = 1, 2, 3, 4 \), der von einem der Testfälle \( u_1 \) bis \( u_4 \) aufgedeckt wird, auch von einem der acht Testfälle

19 Aug 2002 22:14
F1 bis F8 aufgedeckt wird.
(Beispiel: Für \( w_1 \) ergibt sich im korrekten Fall \( w_1 = 1, w_2 = 0, w_3 = 0, w_4 = 0 \). Mit dem Fehler \( D \) bei B2 (korrekt, falsch 0) ergibt sich (vgl. Beispiel 4.2.11 ab S. 90): \( w_1 = D, w_2 = D, w_3 = 0, w_4 = 0 \). Der Fehler wird also bezüglich \( w_1 \) und \( w_2 \) erkannt. Dasselbe passiert aber mit Testfall F1 mit B1 = B2 = B3 = B4 = 0, da \( w_1 \) und \( w_2 \) nur von B1 und B2 abhängen).

Übung 4.5:
Erweitern Sie den UWG aus Abb. 4.6 um die Wirkungen zu den Punkten 2, 3 und 6 bei der Lösung zu Testaufgabe 3 (siehe Anhang A.3). Welche zusätzlichen Testfälle sind dafür erforderlich?

Übung 4.6:
Identifizieren Sie die Funktionen in diversen Programmen von ca. 100 Zeilen, geben Sie die Kompositionsart(en) für die Funktionen an und planen Sie Tests auf der Grundlage der Sätze 4.3.1 (S. 102) und 4.3.2 (S. 104). Sie dürfen dabei auf anzuwendende Testverfahren für einfache (Boolesche) Ausdrücke und Anweisungen verweisen, die erst in Kapitel 9 vorgestellt werden.

4.5 Verwendete Quellen und weiterführende Literatur

Das Vorgehen bei der Fehlererwartungsmethode und bei der Methode der Äquivalenzklassenbildung wurde von Myers vorgeschlagen (s. [Mye 79]). Vorschläge zur Grenzwertanalyse und zum Testen spezieller Werte finden sich z. B. bei Myers, Howden und Adiron et al. (s. [Mye 79], S. 50-55; [How 78a], S. 187 f.; [ABC 82], S. 170; [How 78b], S. 302). Die Ursache/Wirkungsgraph-Methode stammt ursprünglich von Elmendorf, der sich an dem Konzept der Pfadsensitivierung und dem entsprechenden D-Algorithmus von Roth zum Hardwaretesten orientiert (s. [Rot 66], [Elm 73]). Die Methode wurde von Myers modifiziert, z. T. vereinfacht und durch Beispiele illustriert (s. [Mye 76], [Mye 79]). Sechs Varianten dieser Methode, welche die Anzahl der fehlersensitiven Testfälle variieren bzw. optimieren, werden von Weyaker et al. vorgestellt und verglichen (s. [WGS 94]).

Eine Vielzahl von Definitionen, Sätzen und Bemerkungen zum funktionsbezogenen Testen stammt aus den diversen Artikeln und dem Buch von Howden. Die drei Kompositionsarten für Funktionen findet man in [How 80], S. 165 f.; Satz 4.3.1 und Satz 4.3.2 entsprechen den Theoremen 4.14 und 4.15 aus dem Buch [How 87]. Ein umfangreicheres Beispiel zum entwurfsoorientierten Testen („Sortieren durch Mischen“) wird in [How 78a], S. 190 ff. vorgestellt. Die Vorschläge zur Prüfzisierung des Verhaltens eines Systems, welches durch SADT-Diagramme beschrieben ist, stammen von Morin (s. [Mor 86]). Dort wird auch erläutert, wie sich das Verhalten als Automat angeben läßt, aus dem eine geeignete Menge von Testsequenzen abgeleitet werden kann.

5 Testen von Reihenfolgebedingungen und algebraischen Spezifikationen

5.1 Testen von Reihenfolgebedingungen


2. Ein Modul stellt mehrere (Zugriffs-)Funktionen zur Verfügung. Um die Integrität der gespeicherten Daten nicht zu gefährden, sind nur bestimmte Reihenfolgen der Zugriffsfunktionen erlaubt. Wenn das Gesamtsystem robust ist, werden unerlaubte Zugriffe abgefangen und entsprechende Fehlermeldungen ausgegeben. Es ist also zu testen, ob die zulässigen Reihenfolgen fehlerfrei ausführbar sind und die unzulässigen Reihenfolgen zu entsprechenden Fehlern (mit passenden Fehlermeldungen) führen.

In beiden Fällen kann es drei Arten der Reihenfolgebedingungen geben:

(a) spezifizierte Reihenfolgebedingungen,
(b) entworfene Reihenfolgebedingungen,
(c) implementierte Reihenfolgebedingungen.

Die spezifizierten oder entworfenen Reihenfolgebedingungen sind aus der Spezifikation oder dem Entwurf i. allg. direkt ableitbar. Die implementierten Reihenfolgebedingungen lassen sich aus dem Kontrollfluß des Programmcodes ableiten, wenn die Funktionen im Programmcode identifizierbar sind.

Die im folgenden vorgestellten Techniken können dazu benutzt werden, die Übereinstimmung einer gegebenen Reihenfolgebedingung mit einer der anderen Reihenfolgebedingungen festzustellen.
Die Reihenfolgebedingungen werden durch Pfadausdrücke angegeben. Sie werden (neutral) über einer Menge $F$ von „Elementen“ definiert, die bei der hier vorliegenden Anwendung eine Menge von Funktionsnamen ist.

**Definition 5.1.1 (Pfadausdruck)**

1. **Die Syntax der Pfadausdrücke** über einer Menge $F$ ist folgendermaßen definiert:

   (a) Ein leerer Ausdruck ist ein Pfadausdruck, ein Funktionsname aus $F$ ist ein Pfadausdruck.

   (b) Wenn $P$ und $Q$ Pfadausdrücke sind, so auch

   i. die Sequenz von $P$ und $Q$: $(P; Q)$

   ii. die Alternative von $P$ und $Q$: $(P|Q)$

   iii. die Wiederholung von $P$: $P^*$

   iv. die Option von $P$: $[P]$

   (c) Nur was mit (a) oder (b) gebildet werden kann, ist ein Pfadausdruck.

   (d) Zur Einsparung von Klammern gilt, daß Sequenz $(; )$ stärker bindet als Alternative $(|)$.  

2. **Die Semantik der Pfadausdrücke** entspricht der Namenswahl für die verschiedenen Ausdrücke:

   (a) Bei einer Sequenz werden die Elemente (Funktionen) des ersten Ausdrucks vor den Elementen (Funktionen) des zweiten Ausdrucks ausgeführt.

   (b) Bei einer Alternative werden die Elemente (Funktionen) des ersten Ausdrucks oder die Elemente (Funktionen) des zweiten Ausdrucks ausgeführt (aber nicht beide).

   (c) Bei einer Wiederholung werden die Elemente (Funktionen) des Ausdrucks mindestens ein mal ausgeführt$^1$.

   (d) Bei einer Option werden die Elemente (Funktionen) des Ausdrucks kein mal oder ein mal ausgeführt.

   (e) Bei einem leeren Ausdruck wird nichts ausgeführt.


4. Pfadausdrücke mit obigen Eigenschaften werden auch **sequentielle Pfadausdrücke** genannt$^2$.

---

$^1$ Alternativ könnte man — wie bei regulären Ausdrücken — die Wiederholung als Iteration $P^*$ von $P$ definieren, die auch die 0-malige Wiederholung (also keine Ausführung) von $P$ umfaßt. Dann könnte die Option $[P]$ entfallen.

$^2$ Im Unterschied zu nebeneinigen Pfadausdrücken gemäß Definition 14.2.1
BEISPIEL 5.1.1

\[ P = (b \mid c)^\ast \cdot (d \mid e \mid f) \]

Dieser Pfadausdruck \( P \) läßt sich folgendermaßen strukturieren:

\[ Teil_1 = (b \mid c)^\ast \]
\[ Teil_2 = (d \mid e \mid f) \]
\[ P = Teil_1 \cdot Teil_2 \]

\( P \) ist eine Sequenz aus zwei Teilen:
\( Teil_1 \) ist eine Wiederholung der Alternative aus \( b \) oder \( c \). \( Teil_2 \) ist eine Option der Alternative aus einer Sequenz und \( f \), wobei die Sequenz aus \( d \) und \( e \) besteht. Bei \( Teil_2 \) wird also entweder nichts ausgeführt oder die Sequenz von \( d \) und \( e \) ausgeführt oder nur \( f \) ausgeführt.


BEISPIEL 5.1.2

\[ Q = (b \mid c) \cdot (d \mid e \mid f) \]

Zu diesem Pfadausdruck \( Q \) gibt es genau sechs mögliche (Funktions-)Sequenzen:

\[ Q_1 : b, \quad Q_4 : bf, \]
\[ Q_2 : c, \quad Q_5 : cde, \]
\[ Q_3 : bde, \quad Q_6 : ef. \]

Ersetzt man jedoch in \( Q \) den Ausdruck \((b \mid c)\) durch \((b \mid c)^\ast\), so erhält man den Ausdruck \( P \) aus Beispiel 5.1.1 mit der Wiederholung \((b \mid c)^\ast\). Für diese Wiederholung gibt es unendlich viele (Funktions-)Sequenzen: \( b, c, bb, bc, cb, cc, bbb, bbe, bec, bee, cbb, \ldots \), etc. Die vollständigen Sequenzen für \( Q \) ergeben sich durch Anhängen von \( de, f \) oder nichts (dem leeren Ausdruck).

Für Pfadausdrücke mit Wiederholungen müssen die zu testenden (Funktions-)Sequenzen eingeschränkt werden, um in endlicher Zeit fertig zu werden. Dazu wird eine andere Darstellung der Reihenfolgebedingungen verwendet, die in manchen Fällen auch direkt als Spezifikation oder Entwurf vorliegen kann.

Da Pfadausdrücke sich als sogenannte reguläre Ausdrücke interpretieren lassen, kann man auch einen endlichen Automaten angeben, der dieselben zulässigen (Funktions-)Sequenzen beschreibt. Das Verfahren wird an einem Beispiel erläutert, da eine vollständige Einführung in die benötigte Theorie der endlichen Automaten\(^3\) den Rahmen dieses Buches sprengen würde.

\(^3\)siehe z. B. [Bra 84] oder [Weg 93], Kapitel 4
Beispiel 5.1.3

\[ P = (b \mid c)^+ \cdot [d \mid e \mid f] \]

Zu \( P \) kann direkt ein nichtdeterministischer \( \varepsilon \)-Automat\(^4\) angegeben werden (siehe Abbildung 5.1).

![Diagramm zu \( P \)]

\[ \text{Abb. 5.1: } \varepsilon \text{-Automat zu } P \]

Dabei wurde folgendes beachtet:

Regeln für die Darstellung eines Pfadausdruckes als endlicher Automat:

\[ \begin{align*}
1 & \quad P \quad 2 \quad Q \quad 3 \\
1 & \quad Q \quad 2 \quad P
\end{align*} \]

Abb. 5.2: Darstellung von Pfadausdrücken

Ein Pfadausdruck (über einer Menge \( F \)), der nur aus einem Element \( f \) aus \( F \) besteht, wird durch zwei Zustände mit einem Übergang mit Symbol \( f \) beschrieben. Bei zusammengesetzten Pfadausdrücken (gemäß 1(b) von Definition 5.1.1) werden die Automaten so komponiert, wie in Abbildung 5.2 angegeben. Dabei ist zu beachten, daß bei der Sequenz von \( P \) und \( Q \) der Zustand 2 nicht (mehr) Endzustand ist, auch wenn 2 Endzustand des Automaten zu Ausdruck \( P \) war.

Der Automat aus Abbildung 5.1 hat in jedem der vier Zustände 1 bis 4 für jedes (Funktions-)Symbol \( b, c, d, e \) oder \( f \) höchstens einen Übergang (eine „Transition“ zu einem anderen Zustand), was durch eine Kante mit dem entsprechenden Symbol bezeichnet wird. Der Automat enthält aber noch sogenannte \( \varepsilon \)-Übergänge: von

\(^4\)Bei einem \( \varepsilon \)-Automaten sind spontane Übergänge von einem Zustand zu einem anderen erlaubt, ohne daß es ein verursachendes Ereignis geben muß. Diese Übergänge werden als Kante mit Symbol \( \varepsilon \) dargestellt (vgl. „spontane Transitionen“ in [Bra 84], Seite 224).
Zustand 2 nach Zustand 1, um die Wiederholung von b oder c zu beschreiben; von Zustand 2 nach 4, um die Option zu beschreiben. Zustand 1 ist Anfangszustand, was durch einen Pfeil ohne Ursprungszustand gekennzeichnet wird. Zustand 4 ist Endzustand und daher doppelt eingekeilt.


Der Automat aus Abbildung 5.3 beschreibt alle erlaubten (Funktions-)Sequenzen von P. Jede solche Sequenz entspricht gerade einem Weg von dem Anfangszustand 1 zu dem Endzustand 2 oder 4. Im allgemeinen könnte der konstruierte Automat nichtdeterministisch sein und noch überflüssige (äquivalente) Zustände enthalten, das ist aber im Beispiel nicht der Fall. Der Automat aus Abbildung 5.3 ist schon reduziert, d. h. er enthält die Minimalzahl von Zuständen, die nötig sind, um P mit einem deterministischen endlichen Automaten zu beschreiben. Dieser Automat wird mit \( A(P) \) bezeichnet.

Um leicht feststellen zu können, was eine unerlaubte (Funktions-)Sequenz ist, wird der Automat \( A(P) \) noch vervollständigt. Dazu wird ein Fehlerzustand \( E \) definiert mit folgender Eigenschaft:

1. für jedes Funktionssymbol gibt es einen Übergang von \( E \) nach \( E \),
2. für jeden Zustand \( z \) und jedes Funktionssymbol \( f \), für das es noch keinen Übergang von \( z \) zu einem anderen Zustand gibt, wird ein Übergang von \( z \) mit \( f \) zum Fehlerzustand \( E \) definiert.

Unter Verwendung dieser Regeln ergibt sich der in Abbildung 5.4 dargestellte vollständige Automat \( A_e(P) \). Nicht erlaubte (Funktions-)Sequenzen führen also vom Anfangszustand 1 zum Zustand 1, 3 oder \( E \), d. h. einem Zustand, der nicht Endzustand ist.5 (Man beachte, daß in Abbildung 5.4 zum Zwecke der Vereinfachung der

5Von Zustand 1 nach 1 führt nur die leere (Funktions-)Sequenz.
Zeichnung *mehrere* Transitionen durch *eine* Kante mit Mehrfachbeschriftung dargestellt werden; z. B. die Transitionen \( b \) und \( c \) von 2 nach 2 durch eine Kante mit Beschriftung „\( b, c \)“.)

Wenn der Pfadausdruck die erlaubten Reihenfolgen laut Entwurf beschreibt, müßte jetzt getestet werden, ob der Pfadausdruck dieselbe Menge von erlaubten (Funktions-)Sequenzen beschreibt wie die Spezifikation oder das Programm (je nach Vergleichsabsicht). Wenn angenommen werden darf oder bekannt ist, daß der vollständige endliche Automat zum Pfadausdruck der Spezifikation (oder des Programms) höchstens \( k \) Zustände hat, kann eine endliche Menge von (Funktions-)Sequenzen konstruiert werden, die diese Übereinstimmung zuverlässig feststellen kann (*Automatenäquivalenztest*, siehe [Cho 78]). Da diese Voraussetzung oft nicht erfüllt ist, bietet sich eher das folgende Verfahren an, welches zwar nicht alle Abweichungen feststellen kann, aber dafür sehr viel weniger Aufwand erfordert. Dabei wird der Automat \( A_v(P) \) als Graph betrachtet, bei dem die Kantenbeschriftungen den Transitionen von einem Zustand zu einem anderen Zustand (unter einer bestimmten Eingabe) entsprechen.

**Definition 5.1.2**

Sei \( P \) ein Pfadausdruck über einer Menge \( F \) von Funktionssymbolen, sei \( A_v(P) \) der zu \( P \) gehörige reduzierte, vollständige endliche Automat. Sei \( T \) eine endliche Menge von endlichen Sequenzen von Funktionssymbolen aus \( F \). \( T \) erfüllt das Kriterium *alle Transitionen* (für \( P \)) g. d. w. \( T \) für jede Transition

\footnote{Dieser Automat ist bis auf die Umbenennung der Zustände eindeutig bestimmt.}

\footnote{\( T \) ist Teilmenge von \( F^* \).}
$t$ des Automaten $A_t(P)$ eine Sequenz enthält, die [ausgehend vom Anfangszustand von $A_0(P)$] die Transition $t$ ausführt.

Das **Ausführen einer Transition** sollte anschaulich klar sein. In Abbildung 5.4 werden z. B. durch die Sequenz $b b f e d$ die folgenden Transitionen ausgeführt: $b$ von 1 nach 2, $b$ von 2 nach 2, $f$ von 2 nach 4, $e$ von 4 nach $E$, $d$ von $E$ nach $E$. Die Teilsequenz $b b f$ ist eine erlaubte Sequenz für $P$ (aus Beispiel 5.1.3), die Verlängerung um $e d$ bedingt den Fehler. (Man beachte, daß Kanten mit Mehrfachbeschriftung — z. B. die Kante von 1 nach $E$ mit „$d, e, f$“ — mehrere Transitionen beschreiben.)

Das Kriterium **alle Transitionen** für endliche Automaten ist mit dem Kriterium **Zweigübereinkunft** für Kontrollflußgraphen vergleichbar (vgl. Def. 7.2.4, Seite 197). Dies liegt daran, daß endliche Automaten und Kontrollflußgraphen als Graphen definiert sind. Daher lassen sich bei Bedarf auch die stärkeren Kriterien von Kapitel 7 auf endliche Automaten (und damit auf die Pfadausdrücke) anwenden.

**Beispiel 5.1.4**

Zu dem Pfadausdruck aus Beispiel 5.1.3 und dem zugehörigen endlichen Automaten aus Abbildung 5.4 wird folgende Menge von Tests, d. h. eine Menge von Sequenzen von Funktionsaufrufen, konstruiert.

$t_1 : b b c e b b$

$t_2 : e f e c$

Mit diesen beiden Testsequenzen werden alle Transitionen (Kantenbeschrifungen) ausgeführt, die auf einem Weg zu einem erlaubten Endzustand liegen, und schon einige der Transitionen, die in den Fehlerzustand führen. In einen Endzustand führen bei $t_1$ die Anfangsstücke $b, b b, b b e$ und $b b c e$, bei $t_2$ die Anfangsstücke $e$ und $e f$ (da 2 und 4 Endzustände sind).

Um alle Transitionen auszuführen — auch solche, die nur in den Fehlerzustand $E$ führen — sind z. B. noch folgende Sequenzen zu testen:

$t_3 : d e,$

$t_4 : e f,$

$t_5 : f d ;$

$t_6 : b f d,$

$t_7 : b f e,$

$t_8 : b d e f,$

$t_9 : b e c ;$

$t_{10} : c d b,$

$t_{11} : c d e,$

$t_{12} : b d d,$

$t_{13} : b d f .$

$t_3$ bis $t_5$ testen, ob die Sequenzen mit einem falschen Symbol (ungleich $b$ und $e$) anfängen; $t_6$ bis $t_9$ testen, ob die Sequenzen nach einer korrekten Teilsequenz ($b, b f$

---

8Bei einem Vergleich des Pfadausdrucks mit einem implementierten System wird folgendes vorausgesetzt: nach der Abarbeitung von Anfangsstücken von Testsequenzen ist an der Reaktion des implementierten Systems erkennbar, ob es diese Anfangsstücke als korrekt oder nicht korrekt behandelt. Entsprechendes muß bei einem Vergleich mit einer informellen Spezifikation gelten: ein Reihenfolgeprinzip muß für die Anfangsstücke angeben, ob sie korrekt sind oder nicht. Falls diese Voraussetzung nicht gegeben ist, müssen nicht nur die Testsequenzen, sondern auch alle Anfangsstücke der Testsequenzen (hier: $t_1$ und $t_2$) eingegeben und auf die (abschließende) Reaktion des Systems bzw. des Reihenfolgeprinzips gewartet werden.
oder bde) noch weitere (falsche) Funktionssymbole enthalten; \( t_{10} \) bis \( t_{13} \) testen, ob die Anfangsstücke bd oder cd der korrekten Sequenzen bde bzw. cde mit einem der nicht erlaubten Symbole b, c, d oder f fortgesetzt werden.


**Beispiel 5.1.4**

In Beispiel 5.1.4 ist nur jeweils eine Testsequenz aus den Sequenzen \( t_3 \) bis \( t_9 \), \( t_6 \) bis \( t_9 \) und \( t_{10} \) bis \( t_{13} \) auszuwählen. Dabei testen \( t_3 \) bis \( t_5 \) den Fall „ungültiger Anfang der Sequenz“, \( t_6 \) bis \( t_8 \) den Fall „zu lange Sequenz (über Zustand 2 oder 4 hinaus)“ und \( t_{10} \) bis \( t_{13} \) den Fall „ungültige Fortsetzung einer Teilsequenz (über Zustand 3 hinaus)“.

Für Module mit Zugriffsfunctionen\(^9\) läßt sich ein Treibermodul schreiben, welches die Zugriffsfunctionen in der zum Testen notwendigen Reihenfolge aufruft. Normalerweise haben die Zugriffsfunctionen noch Parameter, die bei den einzelnen Funktionsaufrufen geeignet mit Werten versehen werden müssen. Wenn die Wahl dieser Parameterwerte keinen Einfluß auf die möglichen oder erlaubten Funktionssequenzen hat, sind das Modell (Pfadausdrücke und endliche Automaten) und die zugehörige Teststrategie angemessen. Wenn die Wahl der Parameterwerte aber Einfluß darauf hat, ob die Funktionssequenzen möglich oder erlaubt sind, ist das Modell nicht angemessen. Dann sind die Pfadausdrücke mit Zusatzbedingungen zu versehen, die diese Abhängigkeit ausdrücken. Entsprechendes gilt für Gesamtfunktionen, die durch alternative Reihenfolgen von Teilfunctionen berechnet werden\(^10\). Die Auswahl zwischen den Alternativen ist durch (globale) Variablen oder Parameter der Funktionen zu steuern.

Die bisher vorgestellten Testkriterien sind nur **notwendige** Testkriterien: die Transitionen in den zu den Pfadausdrücken gehörenden endlichen Automaten müssen ausgeführt werden, weil für jede Transition (bzw. die entsprechende Funktion, welche zu der Transition gehört) ein fehlerhaftes Verhalten auftreten kann. Die Art der Fehler wird dabei aber nicht berücksichtigt und es ist auch unklar, ob jeder Fehler wirklich aufgedeckt werden kann.

Ein anderes **fehlorientiertes** Kriterium für das Testen von Pfadausdrücken orientiert sich daher direkt an möglichen Fehlern im Pfadausdruck \( P \) selbst [und nicht im zugehörigen Automaten \( A(P) \)]. Solche Fehler können als **Mutationen** formuliert werden, d. h. Veränderungen eines Pfadausdrucks in einen anderen, syntaktisch korrekten Pfadausdruck.

---

\(^9\) Siehe Bedeutung 2 der Reihenfolgebedingungen zu Beginn dieses Kapitels 5.1

\(^10\) Siehe Bedeutung 1 der Reihenfolgebedingungen am Anfang dieses Kapitels 5.1
Beispiel 5.1.6

Beim Pfadausdruck $P_1 = ( [a | b ] )^+; c$ sei durch einen Fehler die Option weggefallen, d. h. die Mutation ergibt $P_2 = ( a | b )^+; c$.

Die zugehörigen Automaten sind in Abbildung 5.5 dargestellt. Die „alle Transitionen“-Folge „abc“ für $A(P_1)$ stellt keinen Fehler fest, da „abc“ auch für $A(P_2)$ eine zulässige Folge darstellt. Nur die Folge „c“ stellt den Unterschied zwischen $P_1$ und $P_2$ fest, obwohl sie nicht alle Transitionen ausführt: „c“ ist bei $P_1$ erlaubt, nicht aber bei $P_2$.

5.2 Testen auf der Basis von algebraischen Spezifikationen

Mit algebraischen Spezifikationen wird beschrieben,

1. welche Funktionen bei der Benutzung eines Moduls verwendet werden können,
2. in welchen Reihenfolgen die Funktionen ausgeführt werden können,
3. welche Wirkung diese Ausführungen haben.

Bei Punkt 2 müssen aus der Spezifikation Pfadausdrücke abgeleitet werden, damit auf dieser Basis Testsequenzen definiert werden können. Punkt 3 kann als Testorakel für den Vergleich der Ausgaben von Spezifikation und Implementation genutzt werden.
Definition 5.2.1 (algebraische Spezifikation) \(^{11}\)
Eine algebraische Spezifikation hat folgende Form (Syntax):

1. Name der Spezifikation,
2. Eventuell eine Folge von Namen anderer Spezifikationen, die von dieser Spezifikation mitbenutzt (importiert) werden,
3. Namen für Datenbereiche (Sorten), auf denen die Funktionen operieren,
4. Deklaration der Funktionen (Operationen genannt) in der Form \( f : a \rightarrow w \), wobei \( a \) eine Folge von Sorten ist, welche die Argumente der Funktion \( f \) beschreiben, und \( w \) eine Sorte ist, die den Wertebereich von \( f \) angibt. Fehlt die Angabe \( a \), so handelt es sich um eine konstante Funktion, eine Konstante.

Die Teile 1 und 2 dienen der Strukturierung von Spezifikationen. Die Teile 3 und 4 beschreiben die verwendeten Funktionen. Die folgenden Teile beschreiben in kombiniertener Form die erlaubten Reihenfolgen und Wirkungen der Funktionen.

5. Für Teil 6 benötigte Variablen werden in der Form \( v : s \) deklariert, wobei \( v \) der Name der Variablen und \( s \) ihr Wertebereich (die Sorte) ist.
6. Durch endlich viele Gleichungen der Form \( L = R \) wird für ein Zusammenspiel von Funktionen\(^{12}\) (in der Form \( L \)) angegeben, welche Wirkung dies hat (in der Form \( R \)). Dabei kann \( R \) selbst wieder ein Zusammenspiel von Funktionen (wie oben) sein.

Anstelle einer exakten Definition der Komponenten einer algebraischen Spezifikation wird ein Beispiel für eine solche Spezifikation vorgestellt, wobei die Teile wie oben angegeben numeriert und mit (* . . . *) die Kommentare dazu eingeklammert werden.

Beispiel 5.2.1

1. Binärbäume

2. Bool, Nat, Alphabet

(* Bool spezifiziert die Sorte Bool mit den booleschen Konstanten WAHR und FALSCH und den logischen Operationen darauf.

Nat spezifiziert die Sorte Nat der natürlichen Zahlen mit der Operation NACH, welche den Nachfolger \( n + 1 \) einer Zahl \( n \) berechnet, sowie den Operationen MAX und MIN, welche das Maximum bzw. Minimum zweier Zahlen berechnen.

Alphabet stellt Bezeichner über einem Alphabet zur Verfügung. Mit diesen Bezeichnern werden die inneren Knoten in einem Binärbäumen markiert. *)

\(^{11}\) Bei dieser Definition ist Teil 6 keine exakte Definition.
\(^{12}\) beschrieben als Term über den Variablen und Funktionsnamen, wobei auch Fallunterscheidungen erlaubt sind (genaueres zu Terminen siehe S. 124)
3. Binärbaum

4. LEER :→ Binärbaum
   (* LEER definiert den leeren Binärbaum *)

BAUM : Binärbaum × Alphabet × Binärbaum → Binärbaum
   (* Aus zwei Binärbäumen und einem Element aus Alphabet wird ein neuer
    Binärbaum konstruiert.*)

LEER? : Binärbaum → Bool
HÖHE : Binärbaum → Nat
   (* LEER und BAUM sind Konstruktoren, d. h. sie erzeugen Elemente der Sorte
    Binärbaum. LEER? und HÖHE sind Abfragen, d. h. sie erzeugen Elemente der
    anderen Sorten, hier Bool und Nat.*)

5. L, R : Binärbaum
   a : Alphabet
   (* Statt „x : Sorte“ und „y : Sorte“ darf abgekürzt „x, y : Sorte“
    geschrieben werden, also z. B. „a, b : Alphabet“.*)

6. (a) LEER? (LEER) = WAHR
   (b) LEER? (BAUM (L, a, R)) = FALSCH
   (c) HÖHE (LEER) = 0
   (d) HÖHE (BAUM (L, a, R)) =
       NACH (MAX (HÖHE (L), HÖHE (R)))
       (* Die Verwendung von Variablen L, a, R in den Gleichungen 6(b) und 6(d)
         bedeutet, daß diese Gleichungen für jeden Wert aus dem Wertebereich der Variablen
         gelten sollen.*)

Verschiedene Gleichungen mit derselben Funktion auf der linken Seite können auch
dauf eine Gleichung mit einer Fallunterscheidung auf der rechten Seite zusammenge-
faßt werden.

Beispiel 5.2.2
Die Gleichungen 6(c) und 6(d) von Beispiel 5.2.1 sind äquivalent zu folgender Glei-
chung, wobei B eine neue Variable der Sorte Binärbaum und eq ein Infixoperator
zum Test der Gleichheit von Binärbäumen ist\[13\].

\[\text{In der folgenden Fallunterscheidung ist im else-Fall der Term } B \text{ stets gleich BAUM}(L, a, R) \text{ mit}
\text{passendem } L, a, R, \text{ da ein nichtleerer Baum immer eine solche Form hat. Ansonsten liegt ein Fehler}
\text{— kein Baum — vor.}\]
\[ HÖHE(B) = \begin{cases} 0 & \text{if } B = \text{LEER} \\ \text{MAX}(HÖHE(L), HÖHE(R)) & \text{else if } B = \text{BAUM}(L, a, R) \end{cases} \]

Gleichungen mit Fallunterscheidungen auf der rechten Seite können meistens in mehrere Gleichungen ohne Fallunterscheidung zerlegt werden, indem das Vorgehen von Beispiel 5.2.2 umgekehrt wird. Im folgenden werden Gleichungen ohne Fallunterscheidung benutzt, da sich die Teststrategien für einfacher formulieren lassen. Mit obiger Umkehrregel können diese Strategien dann auch auf andere Gleichungen angewandt werden.

Mit den Operationen LEER und BAUM aus Beispiel 5.2.1 kann jeder Binärbaum (eindeutig) konstruiert werden. Mit den Operationen LEER? und HÖHE können Eigenschaften der Binärbäume abgefragt werden, ohne die Bäume zu verändern. Dazu müssen die Gleichungen von Teil 6 auf einen gegebenen Binärbaum angewandt werden. Das bedeutet, daß in den Gleichungen die Variablen passend durch einen Ausdruck (genannt Term), der einen Binärbaum beschreibt, ersetzt werden müssen.

Erlaubte Terme zu einer algebraischen Spezifikation sind dabei:

T1. alle Terme, die sich für die importierten Spezifikationen bilden lassen,

T2. alle Konstanten und Variablen zu den Sorten der vorliegenden Spezifikation,

T3. zusammengesetzte Terme der Form \( f(t_1, \ldots, t_n) \), wobei \( f : a_1 \times \cdots \times a_n \to w \) im Teil 4 der Spezifikation als Operationssklausel vorkommt und \( t_i \) ein schon gebildeter Term mit Wertebereichsorte \( a_i \) ist, \( i = 1, \ldots, n \),

T4. Terme, die sich dadurch ergeben, daß in einem Term eine Variable durch einen Term passender Sorte ersetzt wird.

Beispiel 5.2.3

\( C, D, E \) seien konstante Werte der von Binärbaum importierten Spezifikation Alphabet. Nach T1 sind diese Terme der Sorte Alphabet. LEER ist ein konstanter Binärbaum, nach T2 ist also ein Term. Damit sind folgende Zeichenfolgen Terme mit Wertebereichsorte Binärbaum (nach T3 und wegen Teil 4 von Beispiel 5.2.1):

\[
B_1 = \text{BAUM(LEER, D, LEER)}
\]
\[
B_2 = \text{BAUM(LEER, C, BAUM(LEER, D, LEER))}
\]
\[
B_3 = \text{BAUM(BAUM(LEER, C, BAUM(LEER, D, LEER)), E, LEER)}
\]

Mit dem Term \( B_3 \) ist auch \( HÖHE(B_3) \) mit den Regeln T2, T3 und T4 ein Term mit Wertebereichsorte Nat, da \( L \) eine Variable und \( B_3 \) ein Term der Sorte Binärbaum sind und \( HÖHE(L) \) ein Term mit Wertebereichsorte Nat ist (siehe Teil 4 und 5 von Beispiel 5.2.1).
Mit Hilfe der Gleichungen können Terme umgeformt werden, indem der Term auf der linken Seite einer Gleichung durch die rechte Seite dieser Gleichung ersetzt wird. Der Term HöHé(B₃) sollte z. B. solange umgeformt werden, bis sich eine Konstante¹⁴ der Sorte Nat (eine natürliche Zahl) ergibt, da HöHé als letzte (äußere) Operation dieses Terms die Wertebereichsorte Nat hat. Eine solche Umformung wird Reduzierung¹⁵ des Terms genannt.

Beispiel 5.2.4
Zur Veranschaulichung wird ein Binärbaum (L, a, R) durch einen Baum mit Wurzel a, linkem Teilbaum L und rechtem Teilbaum R dargestellt.
Binärbaum B₃ aus Beispiel 5.2.3 hat also die Darstellung in Abbildung 5.6, wenn LEER durch 0 dargestellt wird.

Die HöHé des Binärbaums B₃ läßt sich folgendermaßen mit den Gleichungen der Spezifikation (Teil 6 aus Beispiel 5.2.1) ausrechnen:

Wegen 6(d) gilt:

\[ \text{HöHé} (B₃) = \text{NACH} (\text{MAX} (\text{HöHé} (L), \text{HöHé} (R)), \]

wobei

L = \text{BAUM} (\text{LEER}, C, \text{BAUM} (\text{LEER}, D, \text{LEER}),
R = \text{LEER}

gilt. Bei der inneren Auswertung ergibt sich wegen 6(d):

\[ \text{HöHé} (L) = \text{NACH} (\text{MAX} (\text{HöHé} (\text{LEER}),
\text{HöHé} (\text{BAUM} (\text{LEER}, D, \text{LEER})))), \]

¹⁴Dies ist der Idealfall. Eventuell gelangt man nur zu einem Term der Wertebereichsorte Nat, auf den keine Gleichung mehr anwendbar ist. Dann ist die Spezifikation nicht vollständig.
¹⁵Dies ist ein Spezialfall der allgemeinen Termersetzung, bei der auch die rechte Seite einer Gleichung durch die linke Seite ersetzt werden darf.
(wegen 6(c), 6(d))
\[\text{NACH} \left(\text{MAX}(0, \text{NACH} \left(\text{MAX} (\text{HÖHE}(\text{LEER}), \text{HÖHE}(\text{LEER})))\right)\right)\]

(wegen 6(c))
\[\text{NACH} \left(\text{MAX} \left(0, \text{NACH} \left(\text{MAX}(0, 0))\right)\right)\right)\]
\[= \text{NACH} \left(\text{MAX} \left(0, \text{NACH}(0)\right)\right)\]
\[= \text{NACH} \left(\text{MAX} \left(0, 1\right)\right) = \text{NACH}(1) = 2\]

(wegen der Eigenschaften von NACH und MAX, die in \textbf{Nat} definiert sind).
Also ergibt sich durch Einsetzen von \text{HÖHE}(L) = 2 und wegen 6(c):
\[\text{HÖHE}(B_3) = \text{NACH} \left(\text{MAX} \left(2, \text{HÖHE}(\text{LEER})\right)\right)\]
\[= \text{NACH} \left(\text{MAX} \left(2, 0\right)\right) = \text{NACH}(2) = 3\]

Dies entspricht der anschaulichen Bedeutung: In dem Baum aus Abbildung 5.6 hat der längste Weg von der Wurzel \(E\) zu einem Blatt (der Weg von \(E\) über \(C\) und \(D\) nach 0) die Länge 3.

Wenn die Operationen der algebraischen Spezifikation implementiert sind, kann die Implementation folgendermaßen gegen die Spezifikation getestet werden:

**Teststrategie für algebraische Spezifikationen:**

1. Zu allen Gleichungen der vorliegenden Spezifikation sind durch geeignete Ersetzungen (siehe Regel T4 für Terme auf S. 124) Paare von Termen \((t_l, t_r)\) zu bilden, so dass die Gleichung \(t_l = t_r\) laut Spezifikation richtig ist.

2. Zu jeder Gleichung \(t_l = t_r\) von Schritt 1 sind Folgen \(f_l\) und \(f_r\) von Operationen der Implementation aufzurufen, so dass \(f_l\) dem Term \(t_l\) und \(f_r\) dem Term \(t_r\) entspricht. Mit einer Gleichheitsfunktion \(f_l\) für die Wertebereichsorte von \(t_l\) und \(t_r\) ist zu prüfen, ob \(f_l\) und \(f_r\) dasselbe Ergebnis liefern. Bei Ungleichheit liegt ein Implementierungsfehler vor.

Diese Teststrategie ist natürlich keine konkrete Testmethode, da bei Schritt 1 offen gelassen wurde, welche und wieviel Paare von Termen gebildet werden sollten. Diese Frage wird nach den folgenden Beispielen 5.2.5, 5.2.6 und 5.2.7 geklärt.

**Beispiel 5.2.5**
Für Beispiel 5.2.1 (Binärbaum) seien \texttt{leer}, \texttt{leer?}, \texttt{höhe}, \texttt{baum}, \texttt{nach} und \texttt{max} die implementierte Form der entsprechenden Operationen und Konstanten. Dann ist für die Gleichung 6(d) aus Beispiel 5.2.1 etwa folgendes zu testen, wenn für \(L\), \(a\) und \(R\) dieselben Substitutionen wie in Beispiel 5.2.3 bzw. 5.2.4 vorgenommen werden.
Für die linke Seite ist aufzurufen:

\[ \text{höhe (baum (baum (leer, C), baum (leer, D, leer)), E, leer))} \]

Für die rechte Seite von Gleichung 6(d) ist aufzurufen:

\[ \text{nach(max( \text{höhe(baum(leer,C,baum(leer, D, leer))), \text{höhe(leer)))}} \]

Sei \( n \) das Ergebnis des Aufrufs für die linke Seite, \( m \) das Ergebnis für die rechte Seite. Dann ist mit einer geeigneten Prozedur \text{Nat.gleich?} zu prüfen, ob \( n \) und \( m \) die gleiche natürliche Zahl sind, d. h. \text{Nat.gleich?}(n,m) = true, wobei true die Implementation der Konstanten WAHR aus Bool ist.

Die Aufrufe für die linke und rechte Seite einer Gleichung lassen sich schematisieren:

\text{Beispiel 5.2.6}

Für Gleichung 6(d) aus Beispiel 5.2.1 reichen etwa folgende Aufrufe aus:

für die linke Seite: \( \text{höhe (baum (1, a, r)),} \)

für die rechte Seite: \( \text{nach (max (höhe (1), höhe (r))))} \)

wobei \( l, r \) und \( a \) Parameter vom (implementierten) Typ \text{binärbaum bzw. alphabet} sind.

Die nach der Teststrategie erforderlichen Aufrufe und der Vergleich der linken und rechten Seite lassen sich in einer Testprozedur zusammenfassen.

\text{Beispiel 5.2.7}

Für Beispiel 5.2.6 ergibt sich etwa folgendes:

\begin{verbatim}
procedure Axiom_Höhe_Baum
   (var l, r: binärbaum; a: alphabet);
begin
   if höhe (baum (l, a, r)) =
      nach (max (höhe (l), höhe (r)))
   then OK_Ausgabe ("Axiom_Höhe_Baum", l, a, r)
   else Fehler_Ausgabe ("Axiom_Höhe_Baum", l, a, r)
end
\end{verbatim}

Dabei sind \text{höhe, baum, nach und max} wieder die implementierte Form der entsprechenden Operationen der Spezifikation (vgl. Beispiel 5.2.5).

\text{OK_Ausgabe und Fehler_Ausgabe sind geeignete Prozeduren zum Protokollieren des Erfolgs oder Mißerfolgs des Tests;} \text{"=\text{" ist eine Vergleichsoperation auf natürlichen (oder ganzen) Zahlen, die gerade die logischen Werte true und false liefert und damit \text{Nat.gleich?} aus Beispiel 5.2.5 implementiert.} (Bei komplizierteren Wertebereichen ist dafür eine eigene Vergleichsprozedur zu schreiben.)}

Die Testprozeduren zu den Gleichungen der algebraischen Spezifikation sind dann nur noch mit geeigneten Werten für die Parameter aufzurufen. Daher muß noch die Frage geklärt werden, welche und wie viele Tests pro Gleichung einer algebraischen Spezifikation durchgeführt werden sollen. Dabei gibt es verschiedene Möglichkeiten:
1. Die **Uniformitäts-Hypothese** geht davon aus, daß sich das System bei allen Werten richtig (oder bei allen Werten falsch) verhält. Dann ist irgendein Wert für eine Variable in einer Gleichung als Testwert ausreichend. Die Uniformitätshypothese ist i. allg. bei importierten Daten angebracht.


3. Die **Hypothese der endlichen Dekomposition** geht davon aus, daß endlich viele Alternativen für die Bildung von Termen ausreichen. Diese Termen sind dann wieder als Testwerte für die entsprechende Variable zu verwenden.

Folgender Ansatz wird im weiteren Verlauf verwendet:

Aus der Gleichung sind reguläre Ausdrücke abzuleiten, auf die die Verfahren von Kapitel 5.1 kombiniert mit obigen Hypothesen anzuwenden sind. Die regulären Ausdrücke beschreiben dabei gerade die Möglichkeiten, mit denen die variablen Größen in den Gleichungen (vgl. Beispiel 5.2.6 und 5.2.7) erzeugt werden können. Dazu muß bekannt sein, welche Operationen die zu den Variablen gehörenden Wertebereiche bzw. Sorten konstruieren.

**Beispiel 5.2.8**

Die Gleichungen 6(a) und 6(c) des Binärbäums aus Beispiel 5.2.1 enthalten nur Konstanzen. Die möglichen Funktionsaufruffolgen für die linken Seiten der Gleichungen werden durch die folgenden regulären Ausdrücke beschrieben:

\[
(R6a) = (leer; leer?) \\
(R6c) = (leer; höhe)
\]

Dazu gibt es jeweils nur eine Testmöglichkeit.

Die Gleichungen 6(b) und 6(d) enthalten dagegen die Variablen $L$, $R$ und $a$.

Aus Vereinfachungsgründen wird für $a$ nur ein Testwert $A$ gewählt, da nichts genaueres über die Struktur von Alphabet bekannt ist. Hier wird also die Regularitäts-Hypothese benutzt.

Die verschiedenen Möglichkeiten, Terme der zu $L$ und $R$ gehörenden Sorte Binärbäume zu erzeugen, werden durch die Angaben in Teil 4 von Beispiel 5.2.1 beschrieben: LEEE ist ein Binärbäum; mit zwei Binärbäumen $L$ und $R$ und einem Alphabetelement $A$ ist auch $BAUM(L, A, R)$ ein Binärbäum. In Postfixschreibweise läßt sich dies als „$L A R Baum$“ beschreiben. Wenn stets $A$ verwendet wird, kann $A$ aus

\[16\]

Hier soll nur Binärbäume getestet werden. Die Korrektheit des Imports Alphabet wird vorausgesetzt.
5.2 Testen auf der Basis von algebraischen Spezifikationen

Einsparungsgründen weggelassen werden und also „$R$ Baum“ geschrieben werden. Damit ergeben sich folgende mögliche Terme in Postfixschreibweise, wenn zuerst $L$ und $R$ durch LEER ersetzt werden und später abwechselnd der rechte und linke Teilbaum ($R$ bzw. $L$) expandiert wird:

\[
T_1 = \text{LEER} \\
T_2 = \text{LEER LEER Baum} \\
T_3 = \text{LEER LEER LEER Baum Baum} \\
\text{(dies entspricht BAUM(LEER, A, BAUM(LEER, A, LEER))}) \\
T_4 = \text{LEER LEER Baum LEER Baum} \\
T_5 = \text{LEER LEER Baum LEER LEER Baum Baum} \\
\text{etc.}
\]

Diese Ausdrücke lassen sich mit der folgenden Grammatik erzeugen, die das Startsymbol $B$ und die beiden folgenden Ersetzungsregeln (E1) und (E2) hat:

\[
\begin{align*}
\text{(E1): } & B \to \text{LEER} \\
\text{(E2): } & B \to B B \text{ Baum}
\end{align*}
\]

Die Menge dieser Ausdrücke läßt sich leider nicht exakt mit regulären Ausdrücken beschreiben, da reguläre Ausdrücke nicht mächtig genug sind, um alle endlich beschreibbaren Folgen darzustellen\(^{17}\). Da aber nur endlich viele Tests durchgeführt werden können, muß sowieso eine Einschränkung vorgenommen werden, weil obige (exakte) Grammatik unendlich viele Ausdrücke erzeugt. Als Darstellung werden daher obige fünf Terme $T_1$ bis $T_5$ verwendet (was der Anwendung der Regularitäts-Hypothese entspricht) und Terme der folgenden Art, die degenerierte Binärbäume beschreiben, d. h. „linkslastige“ Bäume, bei denen der rechte Teilbaum stets nur der leere Baum ist:

\[T_R = \text{LEER; (LEER, Baum)}^+\]

Mit der Hypothese der endlichen Dekomposition werden die regulären Ausdrücke $T_1$ bis $T_5$ nun mit „ODER“ verknüpft, was wieder einen regulären Ausdruck ergibt. Der dazugehörige reduzierte endliche Automat ist in Abbildung 5.7 dargestellt, wobei $l$ für LEER und $b$ für Baum steht.

Im Unterschied zu der Vorgehensweise aus Kapitel 5.1 beschreiben die Funktionssequenzen, die zu dem endlichen Automaten aus Abbildung 5.7 gehören, keine einfach geschachtelten Funktionsaufrufe: \(Ilb\) beschreibt nicht die Schachtelung \(b(l(I))\) bzw. \(\text{BAUM(LEER(LEER))}\), sondern entsprechend der unterschiedlichen Stelligkeit von \(\text{BAUM}\) und \(\text{LEER}\) die Schachtelung \(\text{BAUM(LEER, LEER)}\). Daher können nur

\(^{17}\)Die Menge dieser Folgen (die binäre Bäume beschreiben) ist eine kontextfreie (nicht reguläre) Sprache (vgl. Kapitel 5 und 6 in [Weg 83] zu diesen Sprachklassen).
die Funktionsssequenzen gewählt werden, die vom Anfangszustand 1 zu einem der Endzustände 2, 6, 7, 11 oder 12 führen, da nur sie zu einer syntaktisch korrekten Funktionsschachtelung gehören.

Das Kriterium alle Transitionen (von Definition 5.1.2 auf S. 118) muß also modifiziert werden. Dazu ist zuerst anzugeben, wie ein Test für eine Gleichung einer algebraischen Spezifikation aussehen muß.

**Definition 5.2.2 (Test für eine Gleichung G)**

Sei $S$ eine algebraische Spezifikation, $G$ eine Gleichung der Spezifikation. Seien $x_1, \ldots, x_n$ die Variablen in Gleichung $G$. Sei $R_i$ ein regulärer Ausdruck, der alle Terme beschreibt, die Variable $x_i$ ersetzen können. Sei $A_i$ der reduzierte, deterministische, endliche Automat (ohne Fehlerzustand $E$), der die zulässigen Folgen aus $R_i$ beschreibt.

$t$ heißt ein **Test (für eine korrekte Kombination) für Gleichung $G$** genau dann, wenn $t$ paarweise die zur linken und rechten Seite von $G$ gehörigen Funktionskombinationen aufruft (und die Ergebnisse vergleicht), wobei in $G$ die Variablen $x_1, \ldots, x_n$ jeweils durch eine Funktionskombination $f_1, \ldots, f_n$ ersetzt werden und die Postfixnotation von $f_i$, $i = 1, \ldots, n$, im Automaten $A_i$ von einem Anfangszustand zu einem Endzustand führt.
BEISPIEL 5.2.9
Für den Binärbaum aus Beispiel 5.2.1 sind folgende Gleichungen zu behandeln:

(a) \( \text{LEER? (LEER)} = \text{WAHR} \)
(b) \( \text{LEER? (BAUM (L, a, R))} = \text{FALSCH} \)
(c) \( \text{HÖHE (LEER)} = 0 \)
(d) \( \text{HÖHE (BAUM (L, a, R))} = \text{NACH (MAX (HÖHE (L), HÖHE (R))}} \)

Da die Gleichungen (a) und (c) keine Variablen enthalten, kann für diese beiden Gleichungen nur jeweils ein Test durchgeführt werden. Mit den Notationen von Beispiel 5.2.5 auf S. 126 also:

für (a): „leer?(leer)“ ist zu vergleichen mit „true“,  
für (c): „höhe(leer)“ ist zu vergleichen mit „0“.

Die Gleichungen (b) und (d) enthalten die Variablen \( x_1 = L, x_2 = a \) und \( x_3 = R \). Dabei wird \( a \) nur durch die Konstante \( A \) (vgl. Beispiel 5.2.8), aber \( L \) und \( R \) etwa durch folgende Funktionskombinationen ersetzt:

- \( L \) durch \( \text{leer}; \) dies führt im Automaten aus Abbildung 5.7 zum Endzustand 2,
- \( R \) durch \( \text{baum(leer, leer)}; \) die Postfixnotation „leer leer baum“ führt zum Endzustand 7.

Also ist „leer? (baum(leer, A, baum(leer, leer)))“ mit „false“ zu vergleichen, um Gleichung (b) zu testen.

Mit dem Begriff „Test für eine korrekte Kombination für Gleichung G“ aus Definition 5.2.2 kann nun das gewünschte Testkriterium definiert werden.

DEFINITION 5.2.3 ([fast] ALLE KORREKTEN KANTEN)
Seien \( S, G, x_1, \ldots, x_n \) und \( A_i \) wie in Definition 5.2.2 definiert.

1. Eine Menge \( T \) von Tests \( t \) erfüllt das Kriterium alle korrekten Kanten g. d. w. für alle Gleichungen \( G \) von \( S \) und für alle Variablen \( x_i \) in \( G, i = 1, \ldots, n \), und für alle Kanten \( k \) des zu \( x_i \) gehörigen Automaten \( A_i \) ein Test \( t \) in \( T \) existiert, der ein Test für eine korrekte Kombination für Gleichung \( G \) ist und für die linken und rechten Seiten der Gleichung \( G \) Funktionskombinationen aufruft, bei denen die Postfixnotation zu der jeweiligen Funktionskombination \( f_i \) (die \( x_i \) in \( G \) ersetzt hat) in \( A_i \) die Kante \( k \) durchläuft. (Dabei wird angenommen, daß jeder Kante nur genau eine Transition entspricht.)
2. Falls es bei Definition 5.2.2 keinen regulären Ausdruck gibt, der die erlaubten Terme exakt beschreibt, ist ein regulärer Ausdruck zu wählen, der eine möglichst große Teilmenge dieser Termen beschreibt. In diesem Fall wird das Kriterium fast alle korrekten Kanten genannt.

Beispiel 5.2.10
Für den Binärbbaum sind die Gleichungen (a) bis (d) aus Beispiel 5.2.9 zu behandeln. Die beiden ersten Testfälle aus Beispiel 5.2.9 testen die Gleichungen (a) und (c) vollständig, da sie nur Konstanten enthalten. Bei den Gleichungen (b) und (d) wird wieder a durch die Konstante $A$ ersetzt, aber $L$ und $R$ durch alle Funktionenkombinationen, die in Postfixform alle Kanten des Automaten aus Abbildung 5.7 durchlaufen, der einige (fast alle) Funktionenkombinationen beschreibt. Dies sind also die in Tabelle 5.1 aufgeführten Folgen bzw. Terme. Wird in den

<table>
<thead>
<tr>
<th>Name</th>
<th>Folge</th>
<th>Endzustand</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>l</td>
<td>2</td>
<td>l</td>
</tr>
<tr>
<td>$t_2$</td>
<td>lllbb</td>
<td>6</td>
<td>$b(l, b(l, l))$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>llb</td>
<td>7</td>
<td>$b(l, l)$</td>
</tr>
<tr>
<td>$t_4$</td>
<td>lllbb</td>
<td>11</td>
<td>$b(b(l, l), b(l, l))$</td>
</tr>
<tr>
<td>$t_5$</td>
<td>llllbb</td>
<td>12</td>
<td>$b(b(b(l, l), l), l)$</td>
</tr>
</tbody>
</table>

| Tab. 5.1 Folgen und Terme zum Testen der Gleichungen (b) und (d) |

Gleichungen der erfordernliche Term durch den Namen $t_i, i = 1, \ldots, 5$, ersetzt, so erfüllt etwa folgende Tests das Kriterium fast alle korrekten Kanten"; für (b): "ler? (baum($t_i, A, t_j)$)" ist für $i = 1, \ldots, 5$ mit "false" zu vergleichen; für (d): "höhe (baum($t_i, A, t_j$))" ist für $i = 1, \ldots, 5$ mit "nach (max (höhe ($t_i$), höhe ($t_j$)))" zu vergleichen.

Bei den Tests für die Gleichungen (b) und (d) werden in Beispiel 5.2.10 die Variablen $L$ und $R$ gleichzeitig durch den Term $t_i$ ersetzt. Bei den jeweils fünf Tests für (b) und (d) hätten auch andere Kombinationen gewählt werden können, z. B. $L = t_1, R = t_5$ und die anderen vier Kombinationen $L = t_{i+1}, R = t_i, i = 1, \ldots, 4$. Für $L = t_1, R = t_5$ ergibt sich also für (b):

leer? (baum (leer, A, baum(baum(leer, leer), leer), leer))) zu vergleichen mit "false".

Anstelle von fünf Tests für (b) und (d) könnten auch jeweils 25 Tests durchgeführt werden: für alle fünf Terme, die für Variable $L$ eingesetzt werden, können alle Kombinationen mit den fünf Termen, die für Variable $R$ eingesetzt werden, geprüft werden. Ein solches schärferes Kriterium wird (fast) alle Kombinationen von korrekten Kanten genannt.

wobei auch in den Termen abgekürzt wird: LEER durch l und BAUM durch b.
5.2 Testen auf der Basis von algebraischen Spezifikationen

Alternativ zu dem gewählten Ansatz mit regulären Ausdrücken und endlichen Automaten kann auch eine direkte Orientierung an der Grammatik erfolgen, welche die Menge der erlaubten Terme für die Variablen in den Gleichungen der algebraischen Spezifikation beschreibt. Um endlich viele Testfälle zu erhalten, können folgende Einschränkungen im Sinne der Regularitäts-Hypothese und der Hypothese der endlichen Dekomposition gemacht werden, wobei die natürliche Zahl \( k \) vorgegeben wird:

1. alle Ableitungen der Grammatik mit beschränkter Länge \( l \leq k \) erzeugen,
2. für kontextfreie Grammatiken alle Ableitungsbäume mit Höhe \( h \leq k \) generieren,
3. spezielle Ableitungen erzeugen.

Beispiel 5.2.11
Für Binärbäume in Postfixnotation gibt es eine Grammatik mit Startsymbol \( B \) und folgenden Regeln (vgl. Beispiel 5.2.8):

\[
\begin{align*}
(E1): & \quad B \rightarrow \text{LEER} \\
(E2): & \quad B \rightarrow B \ B \ \text{Baum}
\end{align*}
\]

Eine Ableitung der Länge 1 ist \( B \Rightarrow \text{LEER}, \) die mit Regel \( E1 \) den leeren Baum \( \text{LEER} \) erzeugt. Eine Ableitung der Länge 3 ist \( B \Rightarrow B \ B \ \text{Baum} \Rightarrow \text{LEER} \ B \ \text{Baum} \Rightarrow \text{LEER} \ \text{LEER} \ \text{Baum} \), wobei beim ersten Schritt \( E2 \) und dann zweimal \( E1 \) angewandt wird. Da obige Grammatik kontextfrei ist, können Ableitungsbäume erzeugt werden. Der in Abbildung 5.8 dargestellte Baum hat die Höhe 3 und erzeugt den Term \( \text{LEER} \ \text{LEER} \ \text{LEER} \ \text{Baum} \ \text{Baum} \ \text{LEER} \ \text{Baum} \). Spezielle Ableitungen sind z. B.

![Abb. 5.8: Ableitungsbäume der Höhe 3 für einen Binärbäum](image)

alle Ableitungen, bei denen in der rechten Seite von Regel \( E2 \) (\( B \ B \ \text{Baum} \)) stets nur die linke Variable \( B \) mit Regel \( E2 \) ersetzt wird (und erst zum Schluß alle Variablen \( B \) mit Regel \( E1 \) durch \( \text{LEER} \) ersetzt werden). Damit werden „linkslastige“ Bäume erzeugt, die dem Term \( T_R \) aus Beispiel 5.2.8 entsprechen.

\[19\] d. h. auf den linken Seiten der Regeln \( E1 \) und \( E2 \) steht nur ein Symbol, hier: \( B \)
Soll bei der Softwareentwicklung frühzeitig die Spezifikation gegen die Anforderungen „getestet“ werden, kann dies natürlich „per Hand“ gemacht werden, z. B. durch Inspektion (s. Kapitel 12.1). Da dies aber fehleranfällig ist, sollte die Spezifikation wenigstens ausführbar und damit automatisch testbar sein. Für algebraische Spezifikationen bietet sich daher eine (fast) automatische Implementierer der Gleichungen in Form eines Prototypen an, in dem die Terme selbst (in der Form von Bäumen) als Realisierung verwendet werden. Gleichungen, bei denen auf der linken Seite dieselbe Funktion \( f \) zuletzt („außen“) angewendet wird, sind bei der Implementierung von \( f \) alternativ zusammenzufassen, wobei durch eine Entscheidungsfunktion entschieden wird, welche Gleichung im Einzelfall zur Anwendung kommt.

Beispiel 5.2.12
Bei dem Binärbäumen aus Beispiel 5.2.1 bilden mit LEER und BAUM zusammenge- setzte Terme die Implementation des Binärbäums. Für die Funktion HÖHE sind die Gleichungen (c) und (d) von Beispiel 5.2.9 zuständig. Daher ist folgende Entschei- dungsfunktion für einen Term \( t \) zu implementieren:

\[
\text{HÖHE}(t) =
\begin{align*}
\text{if} & \quad t = \text{LEER} \\
\text{then} & \quad 0 \quad (* \text{Gleichung (c) } *) \\
\text{else if} & \quad t = \text{BAUM}(L, a, R) \quad (* \text{„letzte“ Funktion ist BAUM } *) \\
& \quad \text{then } \text{NACH}(\text{MAX}(\text{HÖHE}(L), \text{HÖHE}(R))) \\
& \quad \text{else } \text{ERROR}(* \text{„HÖHE undeﬁniert für“}, \ t)
\end{align*}
\]

Falls die algebraische Spezifikation nicht eindeutig ist, fällt dies beim Konstruieren der Entscheidungsfunktion auf: Es gibt mehrere Gleichungen mit gleicher linker Seite und verschiedener rechter Seite.

Falls die algebraische Spezifikation unvollständig ist, gibt es einen ERROR-Zweig in der Entscheidungsfunktion, der für einen Term \( t \) ausgeführt wird.

Falls die algebraische Spezifikation den Anforderungen widerspricht, kann dies dem Ergebnis für einen Term \( t \) angesehen werden.

Diese drei (kursiv geschriebenen) Fälle weisen also auf Fehler in der Spezifikation hin. Fehlende Eindeutigkeit kann leicht erkannt werden, Unvollständigkeit und Widersprüchlichkeit allerdings nur, wenn man einen geeigneten Testfall findet; dies ist i. allg. nicht entscheidbar. (Genaueres zu dem Vorgehen und den Voraussetzungen, welche die algebraische Spezifikation erfüllen muß, siehe [Jal 87].)
5.3 Übungen

Übung 5.1:
(a) Geben Sie einen regulären Ausdruck bzw. einen endlichen Automaten an, der die erlaubten Zugriffe auf eine Datei angibt:
Die erste Operation muß *open* sein, die letzte *close*. Dazwischen sind beliebig viele *read* oder *write* erlaubt.

(b) Bestimmen Sie Testfälle für die Dateizugriffe nach dem Kriterium *alle Transitionen* aus Definition 5.1.2 auf S. 118.

Übung 5.2:
Die Eingabe eines Kommentardruckprogramms besteht aus einer Folge der Zeichen *; / und c (c repräsentiert alle anderen Zeichen außer * und /). Das Programm soll die Kommentare ausdrucken. Ein Kommentar wird von */* und */* eingeschlossen und kann noch weitere Zeichenfolgen */* enthalten, aber nicht die Zeichenfolge */.

(a) Geben Sie einen minimalen endlichen Automaten an, der die Aufgabe löst. Dabei ist für jedes erkannte Zeichen eine entsprechende Aktion anzugeben. Erlaubte Aktionen sind:

- *ignore* (*tue nichts*)
- *flush_buffer* (*der Ausgabepuffer wird geleert*)
- *add_buffer* (*füge gelesene Zeichen an Pufferinhalt an*)
- *del_buffer* (*löse letztes Zeichen im Puffer*)
- *print_buffer* (*drucke den Inhalt des Puffers*)

(b) Geben Sie eine Menge von Testfällen für das Kommentardruckprogramm an, welche das Kriterium *alle Transitionen* aus Definition 5.1.2 erfüllt.

Übung 5.3:

(a) Ergänzen Sie die algebraische Spezifikation *Binärbaum* aus Beispiel 5.2.1 und 5.2.9 um folgendes:

i. eine Funktion *BAL(b)*, die die *Balance* eines Binärbaums *b* berechnet, d. h. die Länge des kürzesten Weges von der Wurzel bis zu einem Blatt. (*BAL(LEER) = 0; BAL(b) = 1, falls der linke oder rechte Teilbaum von *b* leer ist.*)

ii. eine (Boolesche) Funktion *BAL?(b)*, die angibt, ob ein Binärbaum *b* *balanciert* ist oder nicht, d. h. ob alle Wege von der Wurzel bis zu den Blättern dieselbe Länge haben. (*BAL? ist leicht mit *BAL*, HÖHE und der (importierten) Gleichheitsabfrage „=" für natürliche Zahlen zu spezifizieren.*)
(b) Bestimmen Sie geeignete Testfälle für die neuen Gleichungen mit $BAL$ und $BAL^*$, indem Sie ähnlich wie bei den Beispielen 5.2.8 bis 5.2.10 vorgehen.

Übung 5.4:

(a) Geben Sie eine algebraische Spezifikation für einen Keller (Stack) mit folgenden Funktionen an:

- $empty$ (* definiert einen leeren Keller *)
- $push(k, e)$ (* legt ein Element $e$ auf den Keller $k$ *)
- $pop(k)$ (* entfernt das oberste Element des Kellers $k$ *)
- $top(k)$ (* liefert das oberste Element des Kellers $k$ *)
- $empty?(k)$ (* gibt an, ob der Keller $k$ leer ist oder nicht *)

(b) Bestimmen Sie geeignete Testfälle für den obigen Keller mit den Methoden von Kapitel 5.2.

Übung 5.5:

(a) Geben Sie eine algebraische Spezifikation für eine (einfache!) Schlanke (queue) mit folgenden Funktionen an:

- $empty$ (* definiert eine leere Schlanke *)
- $add(s, e)$ (* fügt ein neues Element $e$ an das Ende der Schlanke $s$ an *)
- $del(s)$ (* entfernt das Element am Anfang der Schlanke $s$ *)
- $first(s)$ (* liefert das Element am Anfang der Schlanke $s$ *)
- $empty?(s)$ (* gibt an, ob die Schlanke $s$ leer ist oder nicht *)

(b) Bestimmen Sie geeignete Testfälle für die obige Schlanke mit den Methoden von Kapitel 5.2.
5.4 Verwendete Quellen und weiterführende Literatur

Die hier vorgestellten Pfadausdrücke sind an die Notation von Seehusen angelehnt (s. [See 87]). Sie gehen auf Campbell und Habermann bzw. Andler zurück (s. [CaH 74], [And 79]). Die hier nur angedeutete Theorie der regulären Ausdrücke und der zugehörigen endlichen Automaten findet man in zahlreichen Büchern, z. B. von Brauer und Wegener (s. [Bra 84]; [Weg 93], Kap. 4 und 5).

Die Technik der algebraischen Spezifikation ist in der Praxis noch wenig verbreitet, hat aber schon eine lange Tradition. Frühe Beispiele für algebraische Spezifikationen und entsprechende Tests findet man bei McMullin, Gannon (und Hamlet), die Keller (stacks), die Bearbeitung sequentieller Dateien und Mustererkennung (pattern matching) als Beispiele angeben (s. [GMH 81], [McG 83]). Eine grundlegende Darstellung findet man z. B. bei Ehrich et al. (s. [EGL 89]). Die für dieses Buch gewählte Darstellung orientiert sich an Kreowski, der auch eine (erweiterte) Spezifikation eines Binärbaumes angibt (s. [Kre 92a]).

Die vorgestellte Teststrategie ähnelt der Strategie von Gannon, wurde aber durch die Ansätze von Gmeiner/Voges und Chow erweitert (s. [Gan 86], [GMH 81], [GmV 86], [Cho 78]). Die Uniformitäts- und Regularitäts-Hypothese und die Hypothese der endlichen Dekomposition stammen von Gaudel/Marre (s. [GaM 88]), die Idee, die Schachtelungstiefe von Termen als Komplexitätsmaß zu verwenden, geht auf Neumann/Lang zurück (s. [NeL 89]).

Die Idee der automatischen, prototypischen Implementierung von algebraischen Spezifikationen stammt von Jalote und wurde von Neumann/Lang aufgegriffen, implementiert und am Beispiel eines Kellers (stack) erläutert (s. [Jal 87], [NeL 89]).

Für „große“ algebraische Spezifikationen, die durch den Import anderer Spezifikationen eine „modulare“ Struktur haben, bietet sich ein „verteilter“ Testen der gesamten Spezifikation an. Dies entspricht den Strategien, die beim Integrationstest von Programmen angewendet werden (genauer siehe Kapitel 13). Ein Vorschlag für ein verteiltes Testen findet sich bei Kreowski (s. [Kre 92a]).

Ein komplexes Beispiel für die algebraische Spezifikation und entsprechende Tests (automatische Türüberwachung bei der U-Bahn) findet man bei Dauchy/Marre (s. [DaM 91]). Das Übungsbeispiel „Kommentarstrukturprogramm“ stammt von Chow (siehe [Cho 78]).
6 Bewertung des spezifikationsorientierten Testens

In den Kapiteln 4 und 5 wurden eine Fülle von Testmethoden vorgestellt. Da beim Testen aus Aufwandsgründen nicht alle Methoden angewendet werden können, ist ein Vergleich und eine Bewertung dringend nötig, damit eine angemessene Menge von Methoden ausgewählt werden kann.

Folgende Vergleichsmaßstäbe können an die Testkriterien angelegt werden, die den Testmethoden zugrunde liegen¹:

1. Ein Vergleich der Art „Testkriterium $K_1$ enthält Testkriterium $K_2$“ führt eine (partielle) Ordnung unter den Testkriterien ein (siehe Kapitel 6.1).
   Dies soll bedeuten, daß jede Testdatenmenge, die Kriterium $K_1$ erfüllt, auch Kriterium $K_2$ erfüllt. Von Interesse ist auch, daß zwei Testkriterien unvergleichbar sind, d. h. daß keines das andere enthält.

2. Die Anzahl der Testdaten, die notwendig sind, um ein bestimmtes Kriterium zu erfüllen, ist von großem (ökonomischem) Interesse.
   Wenn Kriterium $K_1$ Kriterium $K_2$ enthält, ist nur ableitbar, daß man für die Erfüllung von $K_2$ höchstens so viele Testdaten braucht wie für die Erfüllung von $K_1$. Die Anzahl hängt natürlich von der Größe der vorliegenden Spezifikation ab, d. h. von der Anzahl der Äquivalenzklassen bzw. der Ursachen und Wirkungen oder von der Struktur der Pfadausdrücke, endlichen Automaten oder algebraischen Spezifikationen. Daher können nur Abschätzungen für den schlechtesten Fall oder Durchschnittswerte angegeben werden (siehe Kapitel 6.2).

3. Eine Testdatenmenge, die einem Testkriterium genügt, welches ein anderes enthält, wird i. allg. mehr Fehler aufdecken können. Eine solche Aussage sollte aber durch formale Analysen oder empirische Ergebnisse für eine Menge von Spezifikationen und Programmen bestätigt oder widerlegt werden (siehe Kapitel 6.3).


¹Eine Testmethode umfaßt Testkriterien, die Anforderungen an die Testdatenmenge stellen, und Verfahren zur Generierung einer Testdatenmenge, die diesen Anforderungen genügt.
6.1  Enthaltsenein und Unvergleichbarkeit von Testkriterien

Eine Ursache für die oben konstatierte Fälle von Testkriterien ist die Verschieden-
artigkeit der Spezifikationsmethoden bzw. der Spezifikationsmodelle, die ihnen zu-
gründe liegen. Daher können Testkriterien i. allg. nur verglichen werden, wenn das
Spezifikationsmodell gleich oder ähnlich ist.

6.1.1  Vergleich der datenbereichsbezogenen und
entwurfsoorientierten Testkriterien

Für die datenbereichsbezogenen Testkriterien von Kapitel 4.2 und ihre entwurfso-
orientierte Verfeinerung wird die partielle Ordnung dieser Testkriterien durch den
gerichteten Graphen in Abbildung 6.1 dargestellt. Dabei bedeutet ein Pfeil von Kri-
terium $K_1$ nach $K_2$, daß $K_1$ das Kriterium $K_2$ strikt enthält. Dies soll heißen, daß
$K_1$ Kriterium $K_2$ enthält, aber $K_2$ nicht $K_1$ enthält. Falls kein Weg von $K_1$ nach
$K_2$ existiert, sind die Kriterien unvergleichbar.

\[
\text{Ursache/Wirkungsgraph-Methode}
\]
\[
\begin{array}{c}
\text{entwurfsoorientierte Grenzwertanalyse} \\
\downarrow \\
\text{entwurfsoorientierte Äquivalenzklassenbildung}
\end{array} \rightarrow 
\begin{array}{c}
\text{Grenzwertanalyse} \\
\downarrow \\
\text{Äquivalenzklassen-} \\
\text{bildung}
\end{array}
\]

Abb. 6.1: Partielle Ordnung der datenbereichsbezogenen Testkriterien

Die Beziehungen gelten aus folgenden Gründen:

1. Die Grenzwertanalyse ist unter zwei Voraussetzungen eine Verfeinerung der Äqui-
valenzklassenbildung:

   (a) Zu jeder Äquivalenzklasse muß mindestens ein Grenzwert existieren bzw.
gewählt werden. (Bei unkonventionellen Werten — z. B. den Farben rot, gelb,  
blau — ist dies problematisch.)

   (b) Zu jedem Testdatum, das genau eine ungültige Äquivalenzklasse abdeckt,  
uß es auch ein Testdatum geben, das genau einen ungültigen Grenzwert  
enthält (vgl. Übung 6.1).
2. Entwurfsoorientierte Kriterien sind eine Verfeinerung der Kriterien, die sich auf die (nicht so detaillierte) Spezifikation beziehen. Bei der Kompositionsart 1 des funktionsbezogenen Testens (Fallunterscheidung von Funktionen $f_1$ bis $f_n$) gilt diese Beziehung zwischen den Kriterien allerdings nur, wenn die Äquivalenzklassenbildung bei den Teilfunktionen $f_1$ bis $f_n$ die Grenzen der Äquivalenzklassen der Gesamtfunktion und die neuen Grenzen der Fallunterscheidung respektiert.

**Beispiel 6.1.1**

Für eine Rentenberechnung $f$ sei die gültige Äquivalenzklasse:

$$18 \leq \text{Alter} \leq 65$$

und die beiden ungültigen Äquivalenzklassen seien:

$$\text{Alter} < 18, \text{Alter} > 65.$$

Für die besondere Behandlung des Vorruststands sei die Rentenberechnungsfunktion $f$ folgendermaßen aufgeteilt:

$$f = \begin{cases} f_1 & \text{if } \text{Alter} \leq 58 \text{ then } f_1 \text{ else} \\ f_2 & \text{if } \text{Alter} \leq 62 \text{ then } f_2 \text{ else } f_3. \end{cases}$$

In diesem Fall sind als Äquivalenzklassen zu testen:

- für $f_1$: gültig: $18 \leq \text{Alter} \leq 58$,
  ungültig: $\text{Alter} < 18$
  (Alter $> 58$ ist überflüssig; vgl. $f_2$, $f_3$);
- für $f_2$: gültig: $58 < \text{Alter} \leq 62$,
  ungültig: $\text{Alter} > 62$ und $\text{Alter} \leq 58$ sind überflüssig;
  (vgl. $f_1$, $f_3$);
- für $f_3$: gültig: $62 < \text{Alter} \leq 65$,
  ungültig: $\text{Alter} < 62$
  (Alter $\leq 62$ ist überflüssig; vgl. $f_2$, $f_1$).

Für die Kompositionsarten 2 und 3 des funktionsbezogenen Testens (Iteration und Sequenz) gilt analoges wie bei der Kompositionsart 1. In diesen Fällen ist es allerdings schwieriger, Bedingungen für die Beziehungen der Äquivalenzklassen der Gesamtfunktion und der Teilfunktion(en) anzugeben und abzuprüfen:

Bei der *Iteration* entspricht jeder Schleifendurchlauf einem besonderen Fall, einer Äquivalenzklasse, deren Grenzen bestimmt werden müßten.

Bei der *Sequenz* überschneiden sich die Äquivalenzklassen auf eine Weise, die von der Berechnung in den einzelnen Teilfunktionen abhängt. (Daher sollten Sequenzen besser mit den Methoden für Pfadausdrücke getestet werden.)

---

2 Falls kein detaillierter Entwurf vorliegt, entfällt natürlich dieser Vergleich.
3. Entwurfsoorientierte Äquivalenzklassenbildung ist unvergleichbar mit der (spezifikationsorientierten) Grenzwertanalyse: die erste Methode bezieht sich auf mehr Äquivalenzklassen, enthält dafür aber nicht unbedingt Grenzwerte (s. Übung 6.2).

4. Die Ursache/Wirkungsgraph-Methode (UWG-Methode) ist mit den anderen Methoden unvergleichbar, selbst wenn man die Eingabebedingungen (die Ursachen) als Äquivalenzklassen auffaßt.

**Beispiel 6.1.2**

Die Spezifikation — etwa die Einstellungsbedingungen bei der Fluggesellschaft Light-Hansa — enthalte drei Eingabebedingungen:

- E1: 18 \( \leq \) Alter (in Jahren) \( \leq \) 60,
- E2: 50 \( \leq \) Gewicht (in Kilogramm) \( \leq \) 80,
- E3: 105 \( \leq \) Intelligenzquotient (IQ).

Eine Einstellung erfülle, wenn mindestens eine der drei Bedingungen erfüllt ist (ODER-Verknüpfung).

Nach der UWG-Methode sind somit vier Testdaten von folgender Art zu bilden:

- t1: E1 und E2 und E3 nicht erfüllt,
- t2: E1 erfüllt (aber E2 und E3 nicht),
- t3: E2 erfüllt (aber E1 und E3 nicht),
- t4: E3 erfüllt (aber E1 und E2 nicht).

Da es keine Testdaten gibt, bei denen genau eine Bedingung nicht erfüllt ist, wird Schritt 3-3 (siehe Seite 77) der Äquivalenzklassenbildung („genau eine ungültige Äquivalenzklasse abdecken“) nicht erfüllt.

Wegen des Gegenbeispiels 6.1.2 enthält die UWG-Methode nicht die Äquivalenzklassenbildung und — wegen der Beziehungen zwischen den anderen Kriterien und der Äquivalenzklassenbildung — auch nicht die anderen Testkriterien.

Die entwurfsoorientierte Grenzwertanalyse enthält nicht die UWG-Methode, da für Beispiel 6.1.2 folgende Testdaten für die entwurfsoorientierte Grenzwertanalyse (hier gleichbedeutend mit der Grenzwertanalyse) ausreichend sind:

(a) Kombinationen von Grenzwerten, die alle einen gültigen Wert darstellen; z.B. Alter = 18, Gewicht = 80, IQ = 105,

(b) Kombinationen von Grenzwerten, bei denen genau ein Grenzwert ungültig ist; z. B. Alter = 17, Gewicht = 50, IQ = 105.

Bei der Grenzwertanalyse fehlt also z. B. ein Test der Art t2 oder t3 oder t4, bei dem zwei Werte ungültig sind.
Damit ist gezeigt, daß die UWG-Methode mit den anderen Methoden unvergleichbar ist; somit gelten die Beziehungen von Abbildung 6.1. q. e. d.

Das Beispiel 6.1.2 der Einstellungsbedingungen der Light-Hansa zeigt, daß die Forderung von Schritt 3-3 der Äquivalenzklassenbildung, genau eine ungültige Äquivalenzklasse mit einem Testdatum abzudecken, in manchen Fällen unsinnig ist. Die obige ODER-Verknüpfung der zu den Äquivalenzklassen gehörigen Bedingungen ist ein solcher Fall; es ist unsinnig, mit einem Testdatum eine ungültige (und somit zwei gültige) Äquivalenzklassen abzudecken, da schon bei der Abdeckung einer gültigen Äquivalenzklasse die positive Systemreaktion — Einstellung der Person — erfolgen muß. Als Ausweg müßte im vorliegenden Fall die Bedingung „E1 oder E2 oder E3“ als einzige gültige Klasse definiert werden. Die Bedingung „nicht(E1) und nicht(E2) und nicht(E3)“ beschreibt dann die einzige ungültige Klasse.

### 6.1.2 Vergleich der Testkriterien für Reihenfolgebedingungen und algebraische Spezifikationen

Die Testkriterien von Kapitel 5 lassen sich untereinander vergleichen, da sie alle auf dem Modell der regulären Ausdrücke bzw. endlichen Automaten basieren. Allerdings gibt es bei den algebraischen Spezifikationen nur im folgenden Fall keine Schwierigkeiten:

1. wenn endliche Automaten zur Beschreibung ausreichen (vgl. [Gegen-] Beispiel 5.2.8 auf Seite 128);

2. wenn nur eine Variable in einer Gleichung vorkommt, da sonst mehrere Automaten zur Beschreibung der Testdaten herangezogen werden (vgl. Definition 5.2.3 auf Seite 131).

Die partielle Ordnung der Testkriterien wird daher durch zwei unabhängige gerichtete Graphen beschrieben (siehe Abbildungen 6.2 und 6.3, wobei die Bedeutung der Pfeile dieselbe wie bei Abbildung 6.1 ist).

Die Ordnung der Testkriterien aus Abbildung 6.2 ergibt sich direkt aus den Definitionen; das Kriterium „alle korrekten/einige nicht korrekte Transitionen“ ist eine Einschränkung von „alle Transitionen“ (s. Definition 5.1.2 auf Seite 118), welches wiederum nur einen Teil des „Automatenäquivalenztests“ darstellt. Die Relationen zwischen den Begriffen aus Abbildung 6.3 ergeben sich direkt aus den Definitionen 5.2.3 (1) und (2) und der Bemerkung zu den entsprechenden Kombinationen nach Beispiel 5.2.10 (siehe S. 132).

---

3Bei dieser — etwas komplizierter — Äquivalenzklassenbildung enthält die UWG-Methode auch die Äquivalenzklassenbildung, da Testdatum t1 aus Beispiel 6.1.2 die ungültige Klasse und jedes der Testdaten t2 bis t4 die gültige Klasse abdeckt.
6.1 Enthaltsenein und Unvergleichbarkeit von Testkriterien

Abb. 6.2: Ordnung der Testkriterien auf der Basis von Pfadausdrücken

Die Relation zwischen einem Kriterium und der „fast“-Variante gilt allerdings nur, wenn beide Kriterien gebildet werden dürfen, was nach Definition 5.2.3 (2) eigentlich ausgeschlossen ist: nur wenn es keinen exakten regulären Ausdruck gibt, soll näherungsweise ein regulärer Ausdruck gebildet werden. Wird diese Nähерung auch zugelassen, wenn es einen exakten Ausdruck gibt, kann auch die Unvergleichbarkeit von fast alle Kombinationen von Kanten und alle korrekten Kanten gezeigt werden (siehe Übung 6.3).

Abb. 6.3: Partielle Ordnung der Testkriterien auf der Basis von algebraischen Spezifikationen

Abbildung 6.3 kann noch verfeinert werden, da die Begriffe fast alle Kombinationen von korrekten Kanten und fast alle korrekten Kanten nichts darüber aussagen, wie stark die „fast“-Einschränkung gegenüber der korrekten Beschreibung aller möglichen Terme ist. Die dabei verwendbaren Hypothesen (s. Kapitel 5.2) lassen sich dabei ebenfalls anordnen: die Regularitäts-Hypothese und die Hypothese der endlichen Dekomposition enthalten jeweils die (sehr eingeschränkte) Uniformitäts-Hypothese und sind nicht vergleichbar.

Die vorgestellte Enthaltseneinrelation scheint ein plausibles Kriterium für den Vergleich der Testkriterien zu sein. Sie hat aber einen Haken, da alle Testkriterien laut

Im folgenden werden daher die minimalen oder nahezu minimalen Testmengen zu einem Testkriterium betrachtet. Dabei stellt sich die Frage nach dem Kosten-Nutzen-Verhältnis:

- Wie viel Aufwand muss für das Testen getrieben werden, um einem Testkriterium zu genügen?
- Wie viele und welche Fehler werden mit Tests, die ein bestimmtes Kriterium erfüllen, gefunden?

Diese Fragen werden in den folgenden Abschnitten beantwortet.

### 6.2 Anzahl der Testdaten pro Testkriterium

#### 6.2.1 Anzahl der Testdaten bei den datenbereichsbezogenen und funktionsbezogenen Testkriterien

Für die datenbereichsbezogenen Testkriterien von Kapitel 4.2 kann die Anzahl der notwendigen Testdaten folgendermaßen abgeschätzt werden.

Für die Methode der Äquivalenzklassenbildung gilt folgendes für die Anzahl \( t_{g,a} \) der Testdaten für gültige (Äquivalenz-)Klassen, wenn \( EB \) die Menge aller Eingabebedingungen \( b \) ist:

\[
\begin{align*}
t_{g,a} & \geq \max\{\text{Anzahl gültiger Klassen für Eingabebedingung } b \mid b \in EB\} \\
t_{g,a} & \leq \text{Anzahl aller gültigen Klassen}
\end{align*}
\]

(6.1)  
(6.2)

Für die Anzahl \( t_{u,a} \) der Testdaten für ungültige (Äquivalenz-)Klassen und somit für die Anzahl \( t_{a} \) der Testdaten für alle Äquivalenzklassen gilt, wenn \( k_{u,a} \) die Anzahl aller ungültigen Äquivalenzklassen ist:

\[
\begin{align*}
t_{u,a} & = k_{u,a} \\
t_{a} & = t_{g,a} + t_{u,a}
\end{align*}
\]

(6.3)  
(6.4)

Die Beziehung 6.3 folgt direkt aus der Definition von Schritt 3-3 der Methode (s. S. 77). Die Beziehungen 6.1 und 6.2 beschreiben gerade den günstigsten und
6.2 Anzahl der Testdaten pro Testkriterium

ungünstigsten Fall bei Schritt 3-2 der Äquivalenzklassenbildung. Die untere Grenze bei 6.1 wird erreicht, wenn sich alle Eingabe-Bedingungen gleichzeitig durch ein Testdatum erfüllen lassen, wobei allerdings die evtl. verschiedenen Äquivalenzklassen pro Bedingung nicht gleichzeitig abdeckbar sind. Die obere Grenze bei 6.2 wird erreicht, wenn sich alle Bedingungen gegenseitig ausschließen.

Falls es jeweils nur eine gültige (Äquivalenz-)Klasse pro Eingabebedingung gibt, gilt also für die Anzahl $t_{ar{u}}$ der Testdaten für gültige Klassen:

$$1 \leq t_{ar{u}} \leq b_c$$  \hspace{1cm} (6.5)

wobei $b_c$ die Anzahl der Eingabebedingungen ist.

Für die Grenzwertanalyse müssen folgende Bezeichnungen eingeführt werden, um eine Abschätzung angeben zu können: Sei $B$ die Menge aller Bedingungen, $EB$ die Menge der Eingabebedingungen, $AB$ die Menge der Ausgabebedingungen, also $B = EB \cup AB$; $g(b)$ sei die Anzahl der Grenzwerte pro Eingabe- oder Ausgabebedingung $b$.

Bei Bedingungen vom Typ $u \leq x \leq o^*$, die durch ein zweiseitig abgeschlossenes Intervall mit gültigen und ungültigen Grenzwerten dargestellt werden, gilt also $g(b) = 4$. Beispielsweise hat die Bedingung $18 \leq A \leq 65$ die vier Grenzwerte 17, 18, 65, 66, wenn $A$ vom Typ integer ist. Bei einseitig abgeschlossenen Intervallen (z. B. $18 < A$) gilt entsprechend $g(b) = 2$.

Damit ergibt sich für die Anzahl $t_2$ der Tests, wenn alle möglichen Kombinationen von Grenzwerten getestet werden:

$$\sum_{b \in B} g(b) \leq t_2 \leq \prod_{b \in B} g(b) = \prod_{b \in EB} g(b) \cdot \prod_{b \in AB} g(b)$$  \hspace{1cm} (6.6)

Bei 6.6 gilt die obere Abschätzung, wenn alle Kombinationen von Grenzwerten möglich und gewählt worden sind.

Die untere Abschätzung gilt für den Fall, daß die Grenzwerte nicht kombiniert werden (können) und somit einzeln getestet werden. Dabei wird angenommen, daß alle Grenzwerte (pro Variable) verschieden sind.

Für den häufigen Fall $g(b) = 4$ wird 6.6 also zu folgender Abschätzung:

$$4 \cdot |B| \leq t_2 \leq 4^{|B|}$$  \hspace{1cm} (6.7)

Für geordnete Wertebereiche existiert stets mindestens ein Grenzwert pro Äquivalenzklasse. Daher gilt wegen 6.6, 6.2, 6.3 mit $k_{\bar{u}} :=$ Anzahl aller Eingabe-Äquivalenzklassen:

$$t_2 \geq \sum_{b \in B} g(b) \geq \sum_{b \in EB} g(b) \geq k_{\bar{u}} \geq t_{\bar{u}} + t_{u}$$
Wegen 6.4 gilt also:

\[ t_\varrho \geq t_\bar{a} \] (6.8)

Die Grenzwertanalyse erfordert also mindestens so viele Tests \( t_\varrho \) wie die Methode der Äquivalenzklassenbildung \( t_\bar{a} \).

Für den häufigen Fall einer Bedingung \( b \) vom Typ „\( u \ rel_1 x \ rel_2 o \)“ (wobei \( \rel_1 \) und \( \rel_2 \) die Relationen \( \leq \) oder \( < \) sind) gilt \( g(b) = 4 \). In diesem Falle ist die Anzahl der Tests für die Grenzwertanalyse \( t_\varrho \) im Vergleich zur Methode der Äquivalenzklassebildung \( t_\bar{a} \) im Minimalfall etwa das Doppelte, im Maximalfall aber fast eine Zweierpotenz dieser Anzahl (vgl. Übung 6.4a).

Für die Ursache/Wirkungsgraph-Methode (UWG-Methode) ist die Anzahl \( t_{uvw} \) der Tests mit den Anzahlen bei der Grenzwertanalyse oder der Methode der Äquivalenzklassenbildung nur zu vergleichen, wenn folgende Annahmen gemacht werden:

1. Der Ursache/Wirkungsgraph hat nur eine Wirkung.
2. Die Wirkung hängt auf einfache Weise von den Eingabebedingungen ab, d. h. konjunktiv („und“) oder disjunktiv („oder“).
3. Die Eingabebedingungen sind von folgender Art:
   (a) „\( x \ rel k \)“ mit einer Variablen \( x \), einer Konstanten \( k \) und einer Relation \( \rel \) (z. B. \( Y \geq 3 \)) oder
   (b) „\( u \ rel_1 x \ rel_2 o \)“ mit einer Variablen \( x \) und zwei Konstanten \( u \) und \( o \), wobei \( \rel_1 \) und \( \rel_2 \) „\( \leq \)“ oder „\( < \)“ sind und \( u < o \) gilt (z. B. „\( 3 \leq Y < 8 \)“).

Sei wieder \( b_c \) die Anzahl der Eingabe-Bedingungen, die mit der Anzahl \( k_{\bar{a}} \) der gültigen Äquivalenzklassen übereinstimmt. Dann gilt für die Anzahl \( t_{uvw} \) der Tests bei der UWG-Methode:

\[ t_{uvw} \leq b_c + 1 = k_{\bar{a}} + 1 \] (6.9)

Der Grenzwert \( b_c + 1 \) wird angenommen, wenn alle bei der UWG-Methode geforderten Kombinationen von gültigen und ungültigen Bedingungen möglich sind.\(^4\)

Vergleicht man den Maximalwert von \( t_{uvw} \) für den Fall 3(b) mit den Minimalwerten \( t_\varrho \) und \( t_\bar{a} \) bei der Äquivalenzklassen- und Grenzwertmethode (vgl. Übung 6.4b), so ergibt sich sogar bei diesem ungünstigen Vergleich folgendes:

\[ t_{uvw} = \frac{t_\varrho}{4} + 1 \] (6.10)

\[ t_{uvw} = \frac{t_{\bar{a}}}{2} + \frac{1}{2} \] (6.11)

\(^4\)bei Verknüpfung mit „oder“ [„und“] eine Kombination: alle Bedingungen ungültig [gültig];
\( b_c \) Kombinationen: jeweils eine Bedingung gültig [ungültig], alle anderen ungültig [gültig].
6.2 Anzahl der Testdaten pro Testkriterium

Die Anzahl der Tests bei der UWG-Methode ist nur etwa die Hälfte bzw. ein Viertel der Anzahl der Tests bei der Äquivalenzklassen- bzw. Grenzwertmethode. Vergleicht man im Fall 3(b) die Maximalwerte von \( t_{uwg} \), \( t_{bg} \) und \( t_{g} \) (vgl. Übung 6.4b), so ergibt sich folgendes: die Anzahl der Tests bei der Äquivalenzklassenmethode ist in etwa dreimal so groß wie die Anzahl \( t_{uwg} \) der Tests bei der UWG-Methode, und die Anzahl der Tests bei der Grenzwertmethode steigt sogar exponentiell mit \( t_{uwg} \).

\[
\begin{align*}
t_{g} &= 2^{2 \cdot (t_{uwg} - 1)} \\
t_{bg} &= 3 \cdot (t_{uwg} - 1)
\end{align*}
\] (6.12) (6.13)

Insbesondere gegenüber den vollständigen Kombinationen bei der Grenzwertanalyse spart die UWG-Methode also deutlich Tests ein.

Beim funktionsbezogenen Test lassen sich die Anzahlen \( t_{ij} \) und \( t_{while} \) der benötigten Tests für die funktionalen Formen „if \( b \) then \( f_{1} \) else \( f_{2} \)“ und „while \( b(v) = \text{true} \) do \( v \leftarrow f(v) \)“ mit der Anzahl \( t_{b} \) der Tests für das Prädikat \( b \) bzw. \( b(v) \) (als unterer Grenze) abschätzen (siehe Übung 6.5, vgl. Kapitel 4.3):

\[
\begin{align*}
t_{ij} &\geq t_{b} \\
t_{while} &\geq t_{b}
\end{align*}
\] (6.14) (6.15)


6.2.2 Anzahl der Testdaten bei den Testkriterien für Reihenfolgebedingungen und algebraische Spezifikationen

Bei den Testkriterien von Kapitel 5 hängt die Anzahl der Tests natürlich von der Komplexität der Pfadausdrücke bzw. Gleichungen der algebraischen Spezifikation ab. Es gibt aber leider keinen einfachen Zusammenhang zwischen der Struktur dieser Ausdrücke bzw. Gleichungen und der Struktur der daraus abgeleiteten endlichen Automaten, die als Grundlage für die Testermittlung dienen.

Bei Pfadausdrücken und den Testkriterien alle Transitionen und alle korrekten/einige nicht korrekte Transitionen ist der Zusammenhang zwischen der Struktur eines Ausdrucks \( P \) und der Anzahl der Tests aus zwei Gründen kompliziert:

1. Es müssen auch Transitionen ausgeführt werden, die nur auf Wegen in den Fehlerrzustand des Automaten \( A_{f}(P) \) liegen. Diese Transitionen haben keine Entsprechung in dem Pfadausdruck \( P \).
Beispiel 6.2.1

Nur die Transitionen des Automaten \( A(P) \) aus Abbildung 5.3 auf S. 117 haben eine Entsprechung zu dem Pfadausdruck \( P = (b|c) + [d; e|f] \). Die zusätzlichen „Fehler“-Transitionen des vollständigen Automaten \( A_v(P) \) aus Abbildung 5.4 auf S. 118 — d. h. die drei Transitionen von 1 nach \( E \), die Transition von 2 nach \( E \), die vier Transitionen von 3 nach \( E \), die fünf Transitionen von 4 nach \( E \) und die fünf Transitionen von \( E \) nach \( E \) — haben keine direkte Entsprechung zu dem Ausdruck \( P \).


Seien \( t(p) \) und \( t(q) \) die Anzahl der Tests für den Pfadausdruck \( p \) und \( q \). Dann gilt für die Anzahl \( t(r) \) der Tests für einen zusammengesetzten Ausdruck \( r \):

(a) \( t(p) = 1 \), falls \( p \) ein einzelnes Zeichen ist,
(b) \( t(p|q) \leq t(p) + t(q) \).

Beispiel 6.2.2 (Ungleichheit bei Beziehung 2(b))

\[
t(b|bc) = 1 \neq 2 = t(b) + t(bc).
\]

Diese Ungleichheit ergibt sich aus der Tatsache, daß \( b \) ein Anfangsstück von \( bc \) ist und daher nur ein Test für die komplette Folge \( bc \) nötig ist. Die Ungleichung 2(b) kann also nicht durch die entsprechende Gleichung ersetzt werden.

Die folgenden einfachen und naheliegenden Beziehungen 2(c) bis 2(e) gelten zwar in einigen Fällen (s. Beispiel 6.2.3), allerdings nicht in jedem Fall (s. Übung 6.6):

(c) \( t(p; q) = \max(t(p), t(q)) \),
(d) \( t([p]) = t(p) \),
(e) \( t(p^+) = t(p) \).

Beispiel 6.2.3 („Zufällige“ Gültigkeit der Beziehungen (c) bis (e))

\[
t((b|c)^+) = t(b|c) = 2.
\]

\[
t([d; e|f]) = t(d; e|f) = t(d; e) + t(f) = \max(t(d), t(e)) + t(f) = \max(1, 1) + 1 = 2.
\]

\[
t((b|c)^+; [d; e|f]) = 2 = \max(2, 2) = \max(t((b|c)^+), t([d; e|f])).
\]

(Dies wird durch die beiden Testfälle \( t_1 \) und \( t_2 \) aus Beispiel 5.1.4 demonstriert.)
Der Zusammenhang zwischen der Struktur des vollständigen Automaten $A_v(P)$ (zu einem Pfadausdruck $P$) und der Anzahl der nötigen Tests bei den Kriterien alle Transitionen bzw. alle korrekten/einige nicht korrekte Transitionen ist ebenfalls nicht einfach. Klar ist nur, daß eine Schleife bzw. eine starke Zusammenhangskomponente des Automaten nur einen Test erfordert, da mit einer entsprechend langen Testsequenz alle Transitionen durchlaufen werden können. Solche Teilgraphen können also weggelassen werden, wenn die Anzahl der Tests bestimmt werden soll. Außerdem kann ein Weg (eine Folge von Transitionen) von einem Zustand $s$ nach einem Zustand $t$ durch eine Transition von $s$ nach $t$ ersetzt werden, wenn die Zwischenzustände auf dem Weg mit keinen weiteren Transitionen zusammenhängen. Für den so entstandenen Automatengraphen $A_v(P)$ kann die genaue Anzahl der Tests leicht bestimmt werden, da keine Schleifen mehr vorhanden sind. Die Anzahl der Tests $t_{af}$ für das Kriterium alle Transitionen kann z. B. durch die Anzahlen $k$ und $n$ der verbundenen Transitionen (Kanten) und Knoten in dem Automaten $A_v(P)$ nach oben abgeschätzt werden, wobei für $n \geq 2$ gilt:

$$t_{af}(P) = t_{af}(A_v(P)) \leq k - n + 2 \leq k \quad (6.16)$$

**Beispiel 6.2.4**


Beim kompletten Automatenäquivalenztest (vgl. Kapitel 5.1, S. 118) kann die Anzahl der notwendigen Tests $t_{a\tilde{a}}(A_v(P))$ folgendermaßen abgeschätzt werden:

$$t_{a\tilde{a}}(A_v(P)) = z^2 \ast e^{z-z+1} \quad (6.17)$$

Dabei ist $z$ die Anzahl der Zustände des Automaten $A_v(P)$, $s$ die Anzahl der Zustände des korrekten spezifizierten Automaten und $e$ die Anzahl der verschiedenen Eingabesymbole (auf den Kanten bei der graphischen Darstellung des Automaten). Nimmt man $s = z$ an, gilt also:

$$t_{a\tilde{a}}(A_v(P)) = z^2 \ast e \quad (6.18)$$

Also ist die Anzahl der Tests mindestens um den Faktor $z$ größer als die Anzahl der Tests $t_{af}(P)$ bei dem Kriterium alle Transitionen (vgl. 6.16, siehe Übung 6.7):

$$t_{a\tilde{a}}(A_v(P)) \geq z \ast t_{af}(P) \quad (6.19)$$

5$k - n + 2$ ist die zyklomatische Zahl nach McCabe. Sie entspricht der Anzahl der linear unabhängigen Wege im (Automaten-)Graphen (genaueres siehe Kapitel 16.1).

6Der hier weggelassene Beweis stammt laut [Cho 78], S. 181, von M. P. Vasilewski.
Für die Testkriterien auf der Basis von algebraischen Spezifikationen ergeben sich ähnliche Abschätzungen wie bei den Pfadausdrücken und endlichen Automaten. Nach Definition 5.2.3 (1) auf S. 131 geht allerdings die Anzahl der Gleichungen der algebraischen Spezifikation in die Anzahl der Tests ein.

Für die Anzahl der Tests \( t_{ak} \) beim Kriterium (fast) alle korrekten Kanten gilt somit:

\[
t_{ak} = \sum_{G \in Gl} \max \{ t(A_i) | x_i \in G \} \tag{6.20}
\]

Dabei sei \( Gl \) die Menge aller Gleichungen der algebraischen Spezifikation. Für eine Variable \( x_i \) in der Gleichung \( G \) sei \( t(A_i) \) die Anzahl der notwendigen Tests für den zu \( x_i \) gehörigen Automaten (s. Definition 5.2.3 (1)). Die Bestimmung von \( t(A_i) \) ist dabei wieder — wie bei den Pfadausdrücken — von einer Fülle von Einflüssen abhängig, die hier nicht weiter untersucht werden können. Mit \( t_{max} \), dem Maximalwert über alle \( t(A_i) \)-Werte für alle Gleichungen \( G \) aus \( Gl \), läßt sich \( t_{ak} \) nach oben abschätzen, wobei \( g = |Gl| \) die Anzahl der Gleichungen sei:

\[
t_{ak} \leq g \cdot t_{max} \tag{6.21}
\]

Bei dem Kriterium (fast) alle Kombinationen von korrekten Kanten steigt die Anzahl \( t_{akk} \) der Tests gegenüber der obigen Berechnung von \( t_{ak} \), wenn mehr als eine Variable pro Gleichung vorhanden ist und mehr als ein Test bei \( t(A_i) \) verlangt wird. Es gilt:

\[
t_{akk} = \sum_{G \in Gl} \prod_{x_i \in G} t(A_i) \tag{6.22}
\]

Das Maximum über die \( t(A_i) \)-Werte wird also durch ihr Produkt ersetzt.

Mit \( t_{max} \), dem Maximalwert über alle \( t(A_i) \)-Werte, mit \( g = |Gl| \) und \( v_{max} \), der Maximalzahl der Variablen pro Gleichung, erhält man also folgende Abschätzung:

\[
t_{akk} \leq g \cdot (t_{max})^{v_{max}} \tag{6.23}
\]

Beim Vergleich der Maximalwerte von \( t_{ak} \) und \( t_{akk} \) erhält man wegen 6.21 und 6.23:

\[
t_{akk} = g \cdot (t_{max})^{v_{max}} = t_{ak} \cdot (t_{max})^{v_{max}-1} \tag{6.24}
\]

Die maximale Anzahl der Tests beim Kriterium (fast) alle Kombinationen von korrekten Kanten ist also um den Faktor \( (t_{max})^{v_{max}-1} \) größer als die maximale Anzahl der Tests beim Kriterium (fast) alle korrekten Kanten.

6.3 Aufgedeckte Fehler pro Testkriterium

Bei dem Bäumebau aus Kapitel 5.2 sind nur die Gleichungen 6(b) und 6(d) interessant, die Variablen enthalten (siehe Beispiel 5.2.9 auf S. 131). Für die Variablen L und R gilt jeweils \( t(A_L) = t(A_R) = 5 \), für Variable a wurde \( t(A_a) = 1 \) gewählt. Also gilt stets \( \max\{t(A_i) \mid x_i \in G\} = 5 \) für \( G = 6(b) \) und \( G = 6(d) \). Für die Gleichungen 6(a) und 6(c) ist je ein Test erforderlich, da sie keine Variablen enthalten. Da Variable a praktisch wie eine Konstante behandelt wird, kann die Anzahl der Tests mit maximal \( v_{max} = 2 \) Variablen pro Gleichung folgendermaßen abgeschätzt werden:

\[
t_{ak} = 1 + 5 + 1 + 5 = 12 \leq 20 = 4 \cdot 5 = g \cdot t_{max} \quad \text{(vgl. 6.21)},
\]

\[
t_{akk} = 1 + 5^2 + 1 + 5^2 = 52 \leq 100 = 4 \cdot 5^2 = g \cdot (t_{max})^{v_{max}} \quad \text{(vgl. 6.23)}.
\]

Die obere Abschätzung für \( t_{akk} (100) \) ist um den Faktor \( (t_{max})^{v_{max} - 1} = 5 \) größer als die obere Abschätzung für \( t_{ak} (20) \). Für die exakten Werte von \( t_{ak} \) und \( t_{akk} (12 \text{ und } 52) \) gilt ein ähnlicher Faktor, nämlich \( 4 \frac{1}{2} \).

Beispiel 6.2.6

McMullen und Gannon haben vier algebraisch spezifizierte Module getestet. Diese Spezifikation hat insgesamt 211 Gleichungen\(^7\) mit 1310 Zeilen. Die zugehörige Implementierung hat insgesamt 1558 Zeilen Code (s. [McG 83], Table 1). Zum Testen wurden 381 Tests\(^8\) verwendet, also durchschnittlich 1,8 Tests pro Gleichung. Das vorgegebene Testkriterium ist allerdings eine Kombination zweier Testkriterien:

1. Ausführung aller Gleichungen bzw. Gleichungszweige (bei if-then-else-Gleichungen),

2. Ausführung aller Anweisungen der Implementierung, wobei jeder Ausdruck mindestens zwei verschiedene Werte annehmen sollte — was zu 95% erreicht wurde.

Mehrere Werte pro Variable in einer Gleichung werden nicht verlangt, daher reichen 381 Tests für 211 Gleichungen.

6.3 Aufgedeckte Fehler pro Testkriterium

Die vorhergehenden Kapitel beschäftigten sich mit dem Aufwand, den die Erfüllung der Testkriterien bedingt. Der Aufwand wurde in Kapitel 6.1 durch einen Vergleich der Kriterien nur relativ angegeben und in Kapitel 6.2 als Anzahl der notwendigen Testdaten absolut angegeben.

\(^7\) Die Gleichungen enthalten auch Fallunterscheidungen (vgl. Beispiel 5.2.2 auf S. 123) und werden Axiome genannt.

\(^8\) genauer gesagt: 381 Zeilen mit Testdaten (test data lines)

Die diversen Untersuchungen unterscheiden sich allerdings vom Ansatz her. Es gibt folgende Untersuchungsarten: Fallstudien, formale Analysen und statistische Experimente. Für einige wenige Methoden gibt es auch theoretische Untersuchungen, die allerdings von einschränkenden Annahmen über die Spezifikationen bzw. Programme ausgehen.\(^9\)


### 6.3.1 Prozentzahlen der aufgedeckten Fehler

Es gibt eine Reihe von Untersuchungen, die Aussagen darüber machen, wie hoch der Prozentsatz der gefundenen Fehler ist, wenn ein bestimmtes Testkriterium zugrunde liegt.

\(^9\)Ein Beispiel dafür sind die Sätze 4.3.1 auf S. 102 und 4.3.2 auf S. 104 zum funktionsbezogenen Testen.
6.3.1.1 Untersuchungen zum Zufallstest

Für künstlich erzeugte Fehler („Mutationen“) in ausgewählten Programmen wurde von Ntafos bestimmt, wieviel Prozent der Fehler gefunden werden. In einem Fall wurden in 7 Programmen 79,5% der Fehler mit Testdaten entdeckt (s. [Nta 84b]); in einem anderen Fall wurden in 14 FORTRAN-Programmen 93,6% der Fehler mit Testdaten entdeckt (s. [Nta 84a]). Da es beim Zufallstest kein natürliches Beendignskriterium gibt und gegenüber einem strukturierten Verfahren draisib neunmal soviel Testdaten verwendet wurden, ist dieses Ergebnis wenig aussagekräftig.

6.3.1.2 Vergleichende Untersuchungen zwischen dem Zufallstest und der Äquivalenzklassenmethode

Vergleichende Untersuchungen sind relevanter als die Untersuchungen aus Abschnitt 6.3.1.1. Dabei wird i. allg. mit der gleichen Anzahl von Testdaten getestet. Außerdem wird das Problem, daß bei der Äquivalenzklassenmethode nichts darüber ausgesagt wird, welche Daten aus den Äquivalenzklassen auszuwählen sind, durch folgenden Ansatz umgangen: Man abstrahiert völlig von der Spezifikation und betrachtet nur den gesamten Eingabebereich des Programms, der irgendwie in $k$ (Äquivalenz-)Klassen eingeteilt ist. Für den gesamten Eingabebereich und für die einzelnen Klassen werden nun Fehlerarten festgelegt.

Von Duran und Ntafos wurden jeweils 25 Klassen (in einer Menge von Experimenten) so erzeugt, daß mit 2% Wahrscheinlichkeit die Fehlerrate über 98% lag und mit 98% Wahrscheinlichkeit unter 4,9%. Pro Klasse wurde jeweils ein Testdatum zufällig ausgewählt. In 14 von 50 Experimenten war der Zufallstestmethode mindestens so gut wie die (Äquivalenz-)Klassennmethode, im Durchschnitt war die Zufallstestmethode etwas schlechter, d. h. das Verhältnis der Fehleraufdeckungswahrscheinlichkeiten lag bei 93 Prozent (siehe [DuN 84], S. 439 f.).

Dieses Ergebnis widerspricht der Intuition und der tradierten Meinung, daß die Orientierung an Äquivalenzklassen das Testen deutlich effektiver macht als zufälliges Testen. Diese Intuition geht aber von der idealen Annahme aus, daß Äquivalenzklassen nahezu homogen sind. Homogenität bedeutet, daß für jedes Eingabedatum der Klasse ein Fehler auftritt oder für jedes Eingabedatum der Klasse kein Fehler auftritt. In diesem Fall ist ein Test mit je einem Testdatum pro Äquivalenzklasse

---

10) genaueres siehe Kapitel 9.3
11) dem Zweigtest (genaueres siehe Kapitel 7)
12) das Verhältnis von Eingaben, die zu einem Fehler führen, zu allen Eingaben
13) „Die Ermittlung von [Testdaten mittels ] Äquivalenzklassen ist einer Zufallsauswahl von Testfällen weit überlegen.“ (Zitat aus [WB 84], S. 339): „Die wahrscheinlich schwächste Methode ist das Testen mit Zufallsdaten ... Die zufällige Auswahl ... besitzt sicher eine geringe Chance, eine optimale oder fast optimale Untermenge (aller möglichen Eingabedaten) zu sein.“ (Zitat aus [Mye 79], Kapitel 4)
natürlich gleichbedeutend mit einem idealen Test (vgl. Kap. 2.4) und damit einem Zufallstest hauptsächlich überlegen (vgl. [HaT 88], S. 211). In der Praxis ist dies aber nicht zu erreichen, da es keine algorithmisch-konstruktiven Verfahren zum Erzeugen von vollständig homogenen Äquivalenzklassen gibt.

Experimentell konnten Hamlet und Taylor sogar zeigen, daß der Homogenitätssgrad der Äquivalenzklassen wenig Einfluß auf die Testgüte\(^{14}\) hat. Der Test mit Klassen niedriger Homogenität war nur um 22\% schlechter als der Test mit Klassen hoher Homogenität. Wenn nur wenige Klassen (0,1\% oder weniger) einen hohen Anteil von Fehlern enthielten, war die mangelnde Homogenität sogar ohne Einfluß auf die Testgüte (siehe [HaT 88], S. 212).

Durch Gedankenexperimente konnten auch die Bedingungen heraustestet werden, unter denen die Äquivalenzklassenmethode deutlich besser als die Zufallsmethode abschneidet: Bei einer generellen Fehlerrate von 0,001\% (d. h. 1 Fehler pro 100,000 Eingaben) und 25 Testdaten findet die Zufallsmethode einen Fehler mit 0,025\% Wahrscheinlichkeit. Bei insgesamt 15 relativ kleinen und 10 relativ großen Äquivalenzklassen ergeben sich dort, wo man die in Tabelle 6.1 aufgeführten Resultate bei der Äquivalenzklassenmethode mit dem Testdatum pro Klasse. Dabei wurde die Fehlerrate der großen Äquivalenzklassen so gewählt, daß insgesamt die Fehlerrate von 0,001\% vorlag (vgl. [HaT 88], S. 211).

Bei einer Fehlerrate der kleinen Äquivalenzklassen von 0,001\% ist kein Unterschied zur Leistung des Zufallstests zu verzeichnen. Dies liegt daran, daß dann auch die großen Äquivalenzklassen diese Fehlerrate aufweisen müssen. Bei höheren Fehlerraten der kleinen Äquivalenzklassen ist die Testgüte dagegen besser, bei kleineren Fehlerraten dagegen schlechter als beim Zufallstest.

\(^{14}\)Die Testgüte ist die Wahrscheinlichkeit zur Aufdeckung eines Fehlers.

<table>
<thead>
<tr>
<th>Fehlerrate der kleinen Äquivalenzklassen (in %)</th>
<th>Wahrscheinlichkeit, irgendeinem Fehler zu finden (in %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>10</td>
<td>79</td>
</tr>
<tr>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>0,1</td>
<td>1,5</td>
</tr>
<tr>
<td>0,01</td>
<td>0,16</td>
</tr>
<tr>
<td>0,001</td>
<td>0,25</td>
</tr>
<tr>
<td>0,0001</td>
<td>0,011</td>
</tr>
<tr>
<td>0</td>
<td>0,010</td>
</tr>
</tbody>
</table>

**Tab. 6.1** Fehleraufdeckungswahrscheinlichkeit eines Tests je nach Fehlerrate
6.3.1.3 Folgerungen für das data
debereichsbezogene Testen

Als Quintessenz läßt sich also folgendes festhalten: Man muß versuchen, kleine Äquivalenzklassen zu finden, deren Fehlerrate um Größenordnungen größer ist als die durchschnittliche Fehlerrate. Die Äquivalenzklassen sind also sorgfältig an den zu erwartenden Fehlern auszurichten. Dies ist besser als die Suche nach vielen Äquivalenzklassen, die alle eine etwa gleiche Fehlerrate haben, die nahe bei 0 liegt (vgl. [HaT 88], S. 212).


6.3.1.4 Untersuchung zur Ursache/Wirkungsgraph-Methode

Weyuker et al. haben sechs Varianten der UWG-Methode bezüglich Anzahl und Art der aufgedeckten Fehler untersucht (durch Betrachten von 20 UWGs bzw. logischen Formeln einer realen Spezifikation mit 5 bis 14 Eingabebedingungen bzw.-variablen). Dabei wurden — je nach Variante — zwischen 97,9% und 99,7% aller betrachteten Fehler (z. B. Variablen-Negierung) gefunden (s. [WGS 94]).

6.3.1.5 Untersuchungen zum Test spezieller Werte

Von Howden stammt eine Untersuchung zum Test spezieller Werte. In sechs Programmen mit zusammen 712 Zeilen Code wurden mit dieser Methode 17 von 28

\[15\] Beim Test von fehlerertragter Software wurde dieselbe Beobachtung gemacht: Komponenten mit hoher Zuverlässigkeit haben Fehler, die nur unter speziellen Bedingungen auftreten, die beim Zufallstest mit sehr geringer Wahrscheinlichkeit erfüllt werden. Daher ist mit speziellen Werten zu testen, d. h. mit kleinen Äquivalenzklassen mit hoher Fehlerrate (siehe [VMT 86], S. 80 ff.)

\[16\] Aber auch bei der Bildung von Äquivalenzklassen können wichtige Typen von Testfällen übersehen werden. Durch die Methoden „Grenzwertanalyse“ und „Ursache/Wirkungsanalyse“ kann dieser Nachteil der Äquivalenzklassen reduziert werden. (Zitat aus [WIB 84], S. 339.)
156  6  Bewertung des spezifikationsorientierten Testens

bekannten Fehlern, d. h. 61% der Fehler, gefunden. Allerdings wurde der Test spezieller Werte als Verfeinerung eines kombinierten funktions- und strukturorientierten Testverfahrens eingesetzt (siehe [How 78b])\(^\text{17}\).

6.3.1.6 Untersuchungen zum funktionalen Testansatz

Weitere Ergebnisse liegen nur für den funktionalen Testansatz vor, wobei die genaue Methodik des Testens sowie die Untersuchungsart (formale Analyse, statistisches Experiment) von Untersuchung zu Untersuchung differiert. Die günstigsten Werte in Tabelle 6.2 (46% bis 49%) kommen wohl vor allem dadurch zustande, daß Ergebnisse verschiedener Personen (s. [Mye 78]) oder verschiedene Verfahren (s. [How 80]) kombiniert werden. Die schlechteren Ergebnisse (30% bis 24%) basieren meist auf strengeren Untersuchungen, sind aber z. B. bei Howden immer noch fast doppelt so gut wie die Ergebnisse des strukturoorientierten Zweigtestens\(^\text{18}\).

<table>
<thead>
<tr>
<th>aufgedeckte Fehler</th>
<th>Vorgehen</th>
<th>Quelle</th>
</tr>
</thead>
<tbody>
<tr>
<td>49% (7,3 von 15)</td>
<td>Black-Box-Test (2-Personen-Gruppen)</td>
<td>[Mye 78], S. 767</td>
</tr>
<tr>
<td>46% (38 von 83)</td>
<td>Black-Box- und Entwurfstest</td>
<td>[How 80], S. 167</td>
</tr>
<tr>
<td>30% (4,5 von 15)</td>
<td>Black-Box-Test (1-Personen-Gruppen)</td>
<td>[Mye 78], S. 767</td>
</tr>
<tr>
<td>28% (23 von 83)</td>
<td>Entwurfsgestützter Test</td>
<td>[How 80], S. 167</td>
</tr>
<tr>
<td>24% (20 von 83)</td>
<td>Black-Box-Auszahlungstest</td>
<td>[How 80], S. 167</td>
</tr>
</tbody>
</table>

Tab. 6.2 Prozentzahlen aufgedeckter Fehler beim funktionalen Testansatz

6.3.1.7 Untersuchungen zum Testen auf der Basis von Pfadausdrücken oder algebraischen Spezifikationen

Für die Testkriterien von Kapitel 5 liegen praktisch keine entsprechenden Untersuchungen vor. Nur McMullin/Gannon berichten von Testfahrungen mit ihrer Methode, die von algebraischen Spezifikationen ausgeht, aber die Testverfahren mit strukturorientierten Verfahren kombiniert. Damit wurden beim Modultest viele Fehler aufgedeckt, so daß beim Integrationstest nur noch zwei Fehler gefunden wurden (s. [McG 83]).

\(^{17}\) Zum Vergleich: Der Zweigtest (vgl. Kapitel 7.2) deckte nur 6 von 28, also 21%, der Fehler auf.

\(^{18}\) Zum Vergleich: nur 16% beim Zweigtesten bei [How 80]
6.3.2 Art der aufgedeckten Fehler

Die Aussagen über die Prozentsätze der aufgedeckten Fehler hängen von vielen Randbedingungen ab und die Prozentsätze liegen — wie zuvor gezeigt — deutlich unter 100%, meist sogar unter 50%. Daher ist es wichtig, die Fehlerarten zu kennen, die mit den verschiedenen Testmethoden jeweils gut oder schlecht aufgedeckt werden können. Diese Kenntnis kann dann für eine geeignete Kombination der Testmethoden verwendet werden (genaueres s. Kapitel 16.3).

6.3.2.1 Aufgedeckte Fehler beim datenbereichsbezogenen Testen

Ein statistisches Experiment mit 74 Personen und 4 Programmen mit 826 Quellcodeteilen und 34 bekannten Fehlern ergab folgendes: datenbereichsbezogenes Testen, welches die Äquivalenzklassenbildung und die Grenzwertanalyse kombiniert, entdeckt die in Tabelle 6.3 aufgeführten Fehler — im Vergleich zum strukturorientierten Testen (s. Kapitel 7 bis 9) und zur manuellen statischen Analyse (s. Kapitel 12.1).

<table>
<thead>
<tr>
<th>Art/Ort des Fehlers</th>
<th>Anzahl der Fehler</th>
<th>Prozentzahl der durchschnittlich entdeckten Fehler pro Methode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M1</td>
<td>M2</td>
</tr>
<tr>
<td>Initialisierung</td>
<td>2</td>
<td>75%</td>
</tr>
<tr>
<td>Ablauflogik</td>
<td>7</td>
<td>67%</td>
</tr>
<tr>
<td>Berechnung</td>
<td>8</td>
<td>64%</td>
</tr>
<tr>
<td>Schnittstellen</td>
<td>13</td>
<td>31%</td>
</tr>
<tr>
<td>Daten</td>
<td>3</td>
<td>28%</td>
</tr>
<tr>
<td>Meldungstexte</td>
<td>1</td>
<td>8%</td>
</tr>
</tbody>
</table>

Legende:
M1, M2, und M3 sind Durchschnittswerte der Testpersonen bei der kombinierten Äquivalenzklassen- und Grenzwertmethode (M1), beim strukturorientierten Testen (Anweisungsüberdeckung = M2) und bei manueller statischer Analyse (Code Reading = M3).

Tab. 6.3 Entdeckte Fehlerarten im Vergleich (nach [BaS 87], S. 1291)

Bei einer Fehlerklassifizierung nach den Begriffen ausgelassener oder veränderter (fehlerhafter) Code ergab sich, daß 61%, 39% bzw. 56% der 10 Auslassungsfehler und 54%, 44% bzw. 54% der 24 fehlerhaften Codestücke entdeckt werden (jeweils für die Methoden M1, M2 und M3 — siehe Legende von Tabelle 6.3). Das datenbereichsbezogene Testen scheint also besonders geeignet für Auslassungs-, Initialisierungs- und Ablauflogikfehler; für Berechnungs- und Schnittstellenfehler ist dagegen die statische Analyse besser. Alle drei Methoden finden Datenfehler und Fehler in Meldungstexten gleich gut bzw. gleich schlecht (etwa ein Viertel der Datenfehler, praktisch nie einen Fehler in Meldungstexten).

19 Bereichsfehler als spezielle Ablauflogikfehler sind mit der Grenzwertanalyse gut zu entdecken.
Die Ursache-Wirkungsgraph-Methode (UWG-Methode), bei der die Anzahl der Testfälle \emph{minimiert} wird (vgl. Beispiel 4.2.12 ab Seite 95) findet folgende Fehler (Änderungen im UWG) besonders gut: 99,3\% aller Negierungen von Eingabebedingungen, 97,4\% aller Fehler durch Ersetzung einer Eingabebedingung (durch eine andere oder durch einen konstanten logischen Wert) und sogar 100\% aller Negierungen von Teilausdrücken im UWG bzw. dem entsprechenden logischen Ausdruck (bei der Untersuchung von 20 UWGs bzw. entsprechenden logischen Formeln; s. \cite{WGS94}, Table VIII, IX, X).

6.3.2.2 Aufgedeckte Fehler mit speziellen Werten

Folgendes Beispiel zeigt den Fall eines Auslassungsfehlers, der mit der Methode „spezielle Werte“ aufgedeckt werden kann.

\textbf{Beispiel 6.3.1 (Teil eines Sortier-Programms)}

\begin{verbatim}
for r1 := 0 to n begin
  r0 := a[r1];
  for r2 := r1 + 1 to n begin
    if a[r2] > r0 then begin
      r0 := a[r2]; (* r0 = bisher größtes Element *)
      r3 := r2; (* r3 = Index von r0 *)
    end;
    r2 := a[r1]; (* damit soll a[r1] mit a[r3], dem bisher größten Element, vertauscht werden *)
    a[r1] := r0;
    a[r3] := r2;
  end;
end;
\end{verbatim}

Das Programm findet in der inneren for-Schleife das größte Element in der Liste \(a[1], \ldots, a[n]\) und speichert es unter \(r0\) und seinen Index unter \(r3\) ab. In der äußeren for-Schleife sollen \(a[r1]\) und \(a[r3]\) vertauscht werden und \(r1\) inkrementiert werden. Falls in der inneren Schleife nie der then-Zweig durchlaufen wird (weil \(a[r1] = r0\) größer als alle folgenden Elemente ist) behält \(r3\) den Wert aus dem vorherigen Schleifendurchlauf, obwohl korrekterweise dann \(r3 = r1\) gelten müßte. Die äußere Schleife unterläßt also fehlerhafterweise diese Re-Initialisierung \(r3 := r1\).

Wenn die „fehlerhaft“ vertauschten Elemente denselben Wert haben (z. B. \(a[0] = 1; a[1] = a[2] = 3\)) wird der Fehler beim strukturorientierten Testen nicht erkannt, falls eine automatische Initialisierung von Variablen erfolgt. Mit der Regel „alle Array-Elemente müssen verschieden sein“, die zum Testen spezieller Werte gehört, wird obiger Fehler aber zuverlässig erkannt (s. \cite{How78b}, S. 305).
Howden gibt zwei weitere Beispiele von Fehlern in einem Programmmpaket für statistische und numerische Analyse an. Dabei decken spezielle Werte bzw. Grenzwerte von Entwurfsfunktionen die Fehler auf, die mit strukturendorientierten Tests nicht gefunden werden (s. [How 80], S. 168).

### 6.3.2.3 Aufgedeckte Fehler beim Testen von Reihenfolgebedingungen

Für den Automatenäquivalenztest\(^{20}\) gibt es ein theoretisches Ergebnis (s. [Cho 78], S. 181):

**Satz 6.3.1**

*Mit dem Automatenäquivalenztest werden alle Sequenzfehler gefunden, wenn folgende Annahmen gelten:

1. der vorliegende und der korrekte Automat haben dasselbe Eingabealphabet,
2. die Abschätzung der maximalen Anzahl der Zustände des korrekten Automaten ist zutreffend.*

Dabei wird unter einem Sequenzfehler folgendes verstanden:

**Definition 6.3.1**

*Sei A ein Automat, der mit einem (korrekten) Automaten A' verglichen wird.*

1. *A hat einen Sequenzfehler* g. d. w. A einen Ausgabefehler, einen Transferfehler, einen zusätzlichen Zustand oder einen fehlenden Zustand hat.
2. *A hat einen Ausgabefehler* g. d. w. A nicht äquivalent zu A' ist, aber mit einer Änderung der Ausgabefunktion von A (ohne Veränderung der Zustände) äquivalent zu A' gemacht werden kann.
3. *A hat einen Transferfehler* g. d. w. A nicht äquivalent zu A' ist, aber mit einer Änderung der Zustandsübergangsfunktion (ohne Veränderung der Zustände) äquivalent zu A' gemacht werden kann.
4. *A hat einen zusätzlichen (bzw. fehlenden) Zustand* g. d. w. die Zahl der Zustände in A erniedrigt (bzw. erhöht) werden muß, damit A äquivalent zu A' sein kann.

Für das Testkriterium *alle Transitionen* gibt es eine positive und eine negative Feststellung bzgl. der Möglichkeit, gewisse Sequenzfehler aufzudecken. Die negative Feststellung gilt natürlich auch für das schwächere Kriterium *alle korrekten/einige nicht korrekten Transitionen* (s. [Cho 78], S. 182):

\(^{20}\)*s. Kapitel 5.1, Seite 118
Satz 6.3.2
Wenn ein Test das Kriterium „alle Transitionen“ erfüllt, dann werden alle Ausgabefehler, aber nicht unbedingt die anderen Sequenzfehler aufgedeckt.

Die positive Feststellung von Satz 6.3.2 wird durch folgendes Beispiel demonstriert.

Beispiel 6.3.2
Der Automat A aus Abbildung 6.4 a) hat einen Ausgabefehler im Zustand 2 bei Eingabe von b. Jede „alle Transitionen“-Folge muß die Eingabe b im Zustand 2 enthalten und deckt damit den Ausgabefehler auf. Beispielsweise erzeugt die „alle Transitionen“-Folge aabb die Ausgabefolge 0111 (statt korrekterweise 0110).

Abb. 6.4: Automat mit/ohne Ausgabefehler (vgl. [Cho 78], Fig. 5)

Die negative Feststellung von Satz 6.3.2 wird durch das folgende (Gegen-)Beispiel bewiesen.

Beispiel 6.3.3
Der Automat B aus Abbildung 6.5 a) hat einen Transferfehler im Zustand 2 bei Eingabe von a. Dieser Fehler kann mit der „alle Transitionen“-Folge aabb nicht aufgedeckt werden: in beiden Automaten ist die Ausgabefolge unverändert 0111.

Abb. 6.5: Automat mit/ohne Transferfehler (vgl. [Cho 78], Fig. 6)

Diese Ergebnisse lassen sich auf Pfaddarstellungen und zugehörige endliche Automaten nur anwenden, wenn den Transitionen in den Automaten $A(P)$ und $A_v(P)$ (siehe Kapitel 5.1, Abb. 5.3 und Abb. 5.4) passende Ausgabesymbole zugeordnet werden: Alle Transitionen, die in einen gültigen Endzustand führen, erhalten die Ausgabe 1, alle anderen die Ausgabe 0. Eine gültige Sequenz muß also mit der Ausgabe 1 enden.
6.4 Testdatenerzeugung und Messung der Testwirksamkeit

Beispiel 6.3.4
Für den Automaten $A_0(P)$ aus Abbildung 5.4 auf S. 118 erhalten die Transitionen, die in den Zustand 2 oder 4 führen, die Ausgabe 1, alle anderen die Ausgabe 0.

Den gültigen Eingabefolgen $b$, $bc$ und $bde$ sind also die Ausgabefolgen 1, 11 und 101 zugeordnet, den ungültigen Eingabefolgen $bd$, $be$, $bdf$ und $bdef$ dagegen die Ausgabefolgen 10, 10, 100 und 1010.

6.3.2.4 Aufgedeckte Fehler bei den Testkriterien auf der Basis von algebraischen Spezifikationen

Für die Testkriterien auf der Basis von algebraischen Spezifikationen liegen praktisch keine Untersuchungsergebnisse über gut oder schlecht aufzudeckende Fehler vor. Die Ergebnisse bei den Pfadausdrücken und entsprechenden endlichen Automaten lassen sich leider nicht anwenden: Bei algebraischen Spezifikationen werden nur gültige Sequenzen getestet und es sollen nicht Sequenzfehler, sondern abweichende Ergebnisse für die linke und rechte Seite einer Gleichung aufgedeckt werden.

6.3.2.5 Zusammenfassung

Zusammenfassend läßt sich folgendes über das spezifikationsorientierte Testen sagen: Mit diesem Vorgehen lassen sich fehlende Fallunterscheidungen, fehlende (nicht realisierte) Teilfunktionen, fehlende Ablauflogik zum Behandeln falscher Eingabedaten sowie fehlerhaft realisierte Teilfunktionen (Berechnungsfehler oder Bereichsfehler) aufdecken (vgl. [Lig 90], S. 298, [ShL 88], S. 185).

6.4 Testdatenerzeugung und Messung der Testwirksamkeit

Die Kosten für das Erzeugen von Testdaten hängen nicht nur von der Anzahl der zu erzeugenden Testdaten ab (s. Kapitel 6.2), sondern auch von dem Aufwand für das Erzeugen von Testdaten. Dieser Kostenanteil soll in diesem Kapitel quantitativ oder zumindest qualitativ bestimmt werden. Dabei werden zwei alternative Vorgehensweisen bei der Testdatenerzeugung betrachtet:

---

2\textsuperscript{1}\textsuperscript{21}McMullin und Gannon berichten, daß in einem Fall alle Fehler bis auf zwei beim Modultest gefunden wurden (s. [McG 83], S. 334). Ein nicht gefundenen Fehler war ein Bereichsfehler: fehlerhafterweise wurde die Relation $\geq$ statt $>$ verwendet. Der andere nicht entdeckte Fehler beruhte darauf, daß Modulspezifikation und Modularimplementierung einander entsprachen, aber beides gleichermaßen falsch war. Ein solcher Fehler läßt sich natürlich nur beim Integrations- oder Systemtest finden (genaueres siehe Kapitel 13).

19 Aug 2002 22:14
1. zielgerichtetes Erzeugen von Testdaten, so daß ein Testkriterium vollständig erfüllt wird;

2. zielloses Erzeugen von Testdaten mit irgendeiner Methode (oder zufällig).


Ob ein Konstrukt bei einem Test „abgedeckt“ wird, läßt sich i. allg. durch Instrumentierung des Programms oder der (ausführbaren) Spezifikation bestimmen. Dazu sind an geeigneten Stellen Meßanweisungen einzufügen, die das Ausführen des Konstrakts protokollieren.

Im folgenden wird das konkrete Vorgehen für die einzelnen Testkriterien diskutiert und der notwendige Aufwand und die auftretenden Schwierigkeiten verglichen.

### 6.4.1 Testdatenerzeugung für datenbereichsbezogene Testkriterien

#### 6.4.1.1 Testdatenerzeugung für den Zufallstest


Der Aufwand für das zufällige Erzeugen eines Wertes hängt davon ab, wie kompliziert die Beschreibung des einzuhaltenden Wertebereichs ist, da davon die beiden durchzuführenden Berechnungen abhängen:

1. **Bestimmung der Grenzen** des Zahlenbereichs, aus dem zufällig eine Zahl zu erzeugen ist, evtl. in Abhängigkeit von vorher erzeugten Zufallswerten.

---

\(^{22}\) (engl.) test effectiveness ratio. Der Begriff ist etwas irreführend. Das Maß mißt die Gründlichkeit bzw. Vollstän digkeit, mit der ein Testkriterium erfüllt wird. Die Wirksamkeit in Bezug auf die Fehlerrate (s. Kap. 6.3) wird damit aber nur indirekt erfaßt.
2. **Kodierung** der in Schritt 1 erzeugten Zahl in einen Wert des geforderten Wertebereichs.

Der Aufwand für den kompletten Zufallstest hängt dann nur noch multiplikativ von der Anzahl der zu erzeugenden Testdaten ab. Diese Anzahl ist aber beim Zufallstest willkürlich. Es gibt kein vernünftiges und praktikables Kriterium, wann das Erzeugen von Zufallsraten beendet werden sollte. Die notwendige Anzahl $t$ von Zufallstestdaten kann nun bestimmt, wenn folgendes gilt:

1. die Fehllrate $f$ des Programms ist bekannt,
2. man begnügt sich mit einer Chance $x$, einen Fehler zu finden.

In diesem Falle muß $t$ so groß sein, daß \( (1 - f)^t \leq 1 - x \) gilt (vgl. [HaT 88], S. 210).

**Beispiel 6.4.1**
Für eine Fehllrate von 1 pro 1000 ($f = 0,001$) und eine Chance von 99\% ($x = 0,99$) erhält man als minimale Anzahl $t$ von Testdaten: $t = 4603$. Für eine Chance von nur 66,67\% ($x = 0,6667$) braucht man immer noch $t = 1099$ Testdaten.

Gegen die praktische Anwendbarkeit dieses Vorgehens spricht folgendes:

1. Schätzungen der Fehllrate sind meist unmöglich und zumindest schwierig und ungenau.
2. Die Wahl des Vertrauensmaßes $x$ ist willkürlich. Was nützt eine 99\%-Chance, muß man nicht eine 99,99\%-Chance fordern? (Eine 100\%-Chance zum Aufdecken eines Fehlers ist nur erreichbar, wenn alle möglichen Eingabekombinationen gewählt werden.)
3. Bei realistischen Werten für $f$ und $x$ ergibt sich eine sehr hohe Anzahl von Tests (s. obiges Beispiel 6.4.1).

### 6.4.1.2 Testdatenerzeugung für die Äquivalenzklassenmethode

Für die Testdatenerzeugung bei der Methode *Äquivalenzklassenbildung* gilt ähnliches wie bei dem Zufallstest. Wenn mit einem Testdatum $n$ gültige (oder ungültige\(^{23}\)) Äquivalenzklassen $A_1, A_2, \ldots, A_n$ „abgedeckt“ werden sollen, so ist ein Testdatum zu erzeugen, welches die zu den Äquivalenzklassen gehörenden Eingabebedingungen $E_1$ bis $E_n$ gleichzeitig erfüllt. Wenn sich die Eingabebedingungen auf verschiedene Variablen bzw. Parameter beziehen und die Eingabebedingungen „einfach“ sind, ist diese Erzeugung ebenfalls „einfach“, andernfalls kann die Erzeugung schwierig oder sogar unmöglich sein.

---

\(^{23}\)Bei der Äquivalenzklassenbildung soll *höchstens eine* ungültige Äquivalenzklasse mit einem Testfall abgedeckt werden (s. Abschnitt 4.2.1).
Beispiel 6.4.2

1. Das folgende ist ein einfacher Fall. Die Eingabeparameter seien X und Y. Es gelte:
   Eingabebedingung \( E_1: 1 \leq X \leq 10, \) X vom Typ Integer,
   Eingabebedingung \( E_2: Y \in \{ \text{rot, gelb, blau} \}. \)
   Es ist „zufällig“ für X ein Wert zwischen 1 und 10 zu erzeugen und für Y ein Wert zwischen 1 und 3, der als Farbe kodiert wird.

2. Ein schwieriger Fall liegt vor, wenn für die Eingabe X vom Typ Integer folgendes gilt:
   Eingabebedingung \( E_1: 1 \leq X \leq 10, \)
   Eingabebedingung \( E_2: X^2 - 20, 1 \times X + 101 \leq 0. \)
   In diesem Fall muß erst die zu \( E_2 \) gehörige Gleichung \( X^2 - 20, 1 \times X + 101 = 0 \)
   gelöst werden. Mit den Lösungen \( X = 10 \) und \( X = 10, \) ergibt sich, daß \( X = 10 \)
   der einzige Wert ist, der \( E_1 \) und \( E_2 \) erfüllt.

3. Essetzt man im Fall 2 die Bedingung \( E_1 \) durch die Bedingung \( 1 \leq X \leq 9, \) gibt es keinen Wert für X, der \( E_1 \) und \( E_2 \) gleichzeitig erfüllt.

Im Extremfall können die Eingabebedingungen so kompliziert sein, daß nicht zu entscheiden ist, ob die Bedingungen gleichzeitig (oder überhaupt) erfüllt werden können. Wenn die Eingabebedingungen nicht nur von Eingabebvariablen, sondern auch von Bearbeitungszuständen des Programms abhängen, wird die Testdatenerzeugung ebenfalls kompliziert (vgl. das Beispiel 4.2.6 des Textformatierers). Im allgemeinen sind die Eingabebedingungen aber einfach. Treten die Eingabebvariablen nur linear in den Bedingungen auf, ist (leicht) entscheidbar, ob die Bedingungen gleichzeitig erfüllbar sind.

6.4.1.3 Testdatenerzeugung für die Grenzwertanalyse


---

\(^{24}\)Es ist nicht entscheidbar, ob ein beliebiges System von Ungleichungen und Ungleichungen eine Lösung besitzt (s. [Dav 73]).

\(^{25}\)Dies kann mit Methoden der linearen Optimierung geschehen (siehe z. B. [Zin 90], Kap. 2).

\(^{26}\)Sollen alle Eingabewerte (Testwerte) Grenzwerte sein, degeneriert das lineare Optimierungsproblem zu dem Problem, ein lineares Gleichungssystem zu lösen.
6.4 Testdatenerzeugung und Messung der Testwirksamkeit

Beispiel 6.4.3
Für Schritt 2 der Grenzwerteranalyse bei Beispiel 4.2.8 auf Seite 83 gilt folgender Zusammenhang zwischen der Zeilenzahl Z der zu druckenden Datei, den bedruckten Seiten X und den Zeilen Y auf der letzten Seite:

1. \( X = \text{aufrunden}(Z_{\frac{4}{3}}) \),
2. \( Y = 45, \text{ falls } (Z \mod 45) = 0 \text{ und } Z \neq 0 \),
3. \( Y = Z \mod 45, \text{ falls } (Z \mod 45) > 0 \text{ oder } Z = 0 \).

Dabei runde die Funktion „aufrunden“ auf die nächste ganze Zahl nach oben auf, d. h. \( \text{aufrunden}(0) = 0 \), \( \text{aufrunden}(0.01) = 1 \).

Bei gewünschten Werten von \( X \) und \( Y \) ist nun rückwärts zu rechnen, was hier in beiden Fällen nicht eindeutig ist, da modulo (mod) und aufrunden keine injektiven\(^{27}\) Abbildungen darstellen. Für einen Wert \( y \) für \( Y(0 \leq y \leq 45) \) erhält man als Lösungsschar die folgenden Werte \( z \) für die Größe \( Z \):

\[
z = n * 45 + y, \quad n = 0, 1, 2, \ldots, \text{ falls } y > 0 \text{ gilt;}
z = 0, \quad \text{ falls } y = 0 \text{ gilt.}
\]

Für einen gewünschten Wert \( x \) für die Größe \( X \) erhält man als Lösungsschar die folgenden Werte \( z \) für die Größe \( Z \):

\[
z = 45 * (x - 1) + k, \quad k = 1, 2, \ldots, 45, \text{ für } x > 0;
z = 0 \quad \text{ für } x = 0.
\]

Wegen der Lösungsscharen für \( z \) können die gewünschten Grenzwerte für \( X \) und \( Y \) gleichzeitig mit einem Wert von \( z \) erzeugt werden (vgl. Schritt 2 bei Beispiel 4.2.8 in Kapitel 4.2.2).

1. \( x = 0, y = 0 \text{ impliziert } z = 0 \),
2. \( x = 1, y = 45 \text{ impliziert } z = 45 \),
3. \( x = 20, y = 45 \text{ impliziert } z = 20 * 45 = 900 \),
4. \( x = 21, y = 1 \text{ impliziert } z = 20 * 45 + 1 = 901 \),
5. \( x = 2, y = 1 \text{ impliziert } z = 45 + 1 = 46 \).

\(^{27}\)Beispielsweise gilt: \( 47 \mod 45 = 2 \mod 45 = 2; \text{aufrunden}(6.01) = \text{aufrunden}(6.91) = 7 \).
Im obigen Beispiel ist die Rückwärtsrechnung schon nicht trivial. Bei umfangreichen oder komplexen Spezifikationen kann die Rückwärtsrechnung sehr kompliziert werden. Erschwerend kommt hinzu, daß ungültige Ausgabegrenzwerte von einem korrekten Programm meist nicht erzeugt werden können. Nur unter der Annahme, daß das Programm einen bestimmten Fehler hat, kann ein Testdatum erzeugt werden, welches eventuell den ungültigen Grenzwert erzeugt. (Ob das tatsächlich der Fall ist, kann nur durch Ausführen des (fehlerhaften) Programms festgestellt werden.)

**Beispiel 6.4.4**

Bei korrekter Arbeitsweise des Programms zu Beispiel 6.4.3 kann der Wert \( y = 46 \) nicht erzeugt werden (siehe Gleichungen 2 und 3). Der Test \( 5 \) mit \( z = 46 \) ist aber geeignet, diese fehlerhafte Ausgabe zu erzeugen — falls das Programm beim Formatieren nicht modulo 45 (sondern modulo \( k \) mit \( k > 45 \) ) „rechnet“.

### 6.4.1.4 Testdatenerzeugung für die UWG-Methode

Bei der Ursache/Wirkungsgraph-Methode muß erst in einem kreativen Akt der Ursache/Wirkungsgraph (UWG) konstruiert werden. Dies ist noch schwieriger als bei der Äquivalenzklassenbildung, da die Eingabeeinschränkungen und die Beziehungen zwischen Eingaben (Ursachen) und Ausgaben (Wirkungen) ermittelt werden müssen. Die Ableitung der Wertetabelle aus dem UWG ist dann allerdings automatisierbar, da die Regeln für das Aufstellen der sensiblen Testfälle mathematisch präzise definiert sind (vgl. Abschnitt 4.2.3). Damit die Testfälle auch durch konkrete Testdaten erfüllbar sein können, ist allerdings entscheidend, daß die Einschränkungskommentare genau die Eingabeeinschränkungen widerspiegeln. Andernfalls könnten Testfälle mit widersprüchlichen, also unerfüllbaren Anforderungen erzeugt werden.

Da die Eingabe- und Ausgabebedingungen (Ursachen und Wirkungen) im allgemeinen wie bei der Äquivalenzklassenbildung und Grenzwertanalyse formuliert sind, treten beim Erzeugen der konkreten Testdaten die gleichen Probleme wie bei jenen Methoden auf (s. Abschnitt 6.4.1.2 und 6.4.1.3).


G. J. Myers behält also recht: „Der Leser sollte... einsehen, daß die Bestimmung von Grenzwerten äußerst schwierig sein kann und daher einen beträchtlichen Aufwand an geistiger Arbeit verlangt.“ (Zitat aus [Mye 79], Kapitel 4, Ende des Abschnitts „Grenzwertanalyse“).
6.4.2 Messung der Testwirksamkeit für datenbereichsbezogene Testkriterien

Für die datenbereichsbezogenen Testkriterien werden nun die Probleme bei der folgenden Teststrategie betrachtet: Gegeben sei eine Menge von konkreten Testdaten; wie leicht läßt sich feststellen, wie hoch die Testwirksamkeit der Testdaten bzgl. eines der datenbereichsbezogenen Testkriterien ist?

6.4.2.1 Messung der Testwirksamkeit für den Zufallstest

Für den Zufallstest läßt sich kein Testwirksamkeitsmaß bestimmen, da es — wie in Abschnitt 6.4.1.1 erwähnt — kein praktikables Verfahren gibt, eine hinreichende Anzahl von Tests zu berechnen. Bei bekannter Fehlerrate $f$ und gegebener Anzahl $t$ von Zufallstests könnte höchstens die Wahrscheinlichkeit $x$ ausgerechnet werden, mit der ein Fehler aufgedeckt werden kann:

$$x = 1 - (1 - f)^t$$

Beispiel 6.4.5

Für $f = 0,001$ und $t = 100$ Tests hat man nur eine Chance $x$ von etwa 9,5\%, einen Fehler zu finden\(^{28}\) ($x = 1 - 0,999^{100} = 1 - 0,9048 = 0,0952$).

Für $t = 100$ und $f = 0,01$ beträgt die Chance dagegen schon 63,4\%.

6.4.2.2 Messung der Testwirksamkeit bei der Äquivalenzklassenmethode

Für die Äquivalenzklassenbildung ist die Bestimmung des Testwirksamkeitsmaßes einfach, wenn die Eingabebedingungen sich direkt auf die EingabevARIABLEN beziehen\(^{29}\). Für jedes Testdatum $t$ muß lediglich berechnet werden, in welche gültigen oder ungültigen Äquivalenzklassen die Eingabewerte fallen. Dazu ist nur zu überprüfen, ob die entsprechenden Bedingungen erfüllt sind.

Beispiel 6.4.6

Die beiden gültigen Äquivalenzklassen $G_1$ und $G_2$ seien durch folgende Eingabebedingungen charakterisiert (vgl. Beispiel 6.4.2.2 auf S. 164):

- $G_1: 1 \leq x \leq 10$,
- $G_2: x^2 - 20, 1\star x + 101 \leq 0$ bzw. $10 \leq x \leq 10, 1$.

---

\(^{28}\) $f = 0,001$ bedeutet: Einer von 1000 Eingabewerten erzeugt ein Programmfehlerverhalten.

\(^{29}\) Das Beispiel 4.2.6 des Textformatierers stellt wieder einen komplizierteren Fall dar, da Bearbeitungszustände in die Eingabebedingungen eingehen.
Die ungültigen Äquivalenzklassen sind dann:

\[ U_1 : x < 1, \ U_2 : x > 10, \]
\[ U_3 : x < 10, \ U_4 : x > 10, 1. \]

Für die folgenden vier Testdaten \( t_1 \) bis \( t_4 \) ergeben sich die in Tabelle 6.4 dargestellten „Abdeckungen“ von Äquivalenzklassen. Mit \( t_1 \) werden \( U_1 \) und \( U_3 \) abgedeckt, da

<table>
<thead>
<tr>
<th>Test</th>
<th>Wert von ( x )</th>
<th>abgedeckte Äquivalenzklassen</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>0</td>
<td>( U_1, U_3 )</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>1</td>
<td>( G_1, U_3 )</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>10</td>
<td>( G_1, G_2 )</td>
</tr>
<tr>
<td>( t_4 )</td>
<td>11</td>
<td>( U_5, U_4 )</td>
</tr>
</tbody>
</table>

Tab. 6.4 Testdaten und abgedeckte Äquivalenzklassen

\( x = 0 < 1 \) und \( x = 0 < 10 \) gilt. Daher werden die alternativen Klassen \( U_2 \) und \( G_1 \) sowie \( G_2 \) und \( U_4 \) nicht abgedeckt. Die Angaben für \( t_2 \) bis \( t_4 \) lassen sich entsprechend begründen.

Die beiden gültigen Äquivalenzklassen werden also gleichzeitig mit \( t_3 \) abgedeckt; die ungültige Klasse \( U_3 \) wird mit \( t_2 \) getestet. Die ungültigen Klassen \( U_1 \), \( U_2 \) und \( U_4 \) werden mit den Tests \( t_1 \) und \( t_4 \) nicht richtig getestet, da gleichzeitig noch eine andere ungültige Klasse (\( U_5 \), \( U_4 \) oder \( U_2 \)) abgedeckt wird. Von den sechs abzudeckenden Äquivalenzklassen (das sind hier die „Konstrukte“) werden also nur drei „richtig“ getestet (vgl. Kapitel 4.2.1). Das Testwirksamkeitsmaß

\[
\text{TWM}_A := \frac{\text{Anzahl der richtig abgedeckten Äquivalenzklassen}}{\text{Anzahl aller Äquivalenzklassen}}
\]

hat also den Wert \( \text{TWM}_A = \frac{3}{6} = 50\% \). Als Testwirksamkeitsmaß läßt sich allerdings höchstens \( \frac{3}{6} = 66,6\% \) erreichen, wenn \( x \) eine Variable vom Typ Real ist. Dies liegt daran, daß mit \( U_1 \) stets auch \( U_3 \) und mit \( U_4 \) stets auch \( U_3 \) abgedeckt wird, da \( x < 1 \) impliziert \( x < 10 \) und \( x > 10, 1 \) impliziert \( x > 10 \). Andererseits erfüllt \( x = 10, 05 \) sowohl \( U_2 \) als auch \( G_2 \), da \( 10 < 10, 05 < 10, 1 \) gilt. Also lassen sich nur vier der sechs Äquivalenzklassen „richtig“ abdecken (vgl. Abschnitt 4.2.1).

6.4.2.3 Messung der Testwirksamkeit bei der Grenzwertanalyse

Bei der Grenzwertanalyse stellt sich bei den Grenzwerten der Eingabe-Äquivalenzklassen das entsprechende Problem wie bei der Äquivalenzklassenbildung. Anstelle der Ungleichungen müssen lediglich die entsprechenden Gleichungen herangezogen werden.
Beispiel 6.4.7
Für die Äquivalenzklassen $G_1, G_2, U_1, U_2, U_3$ und die Testdaten $t_1$ bis $t_4$ aus Beispiel 6.4.6 ergibt sich Tabelle 6.5, wenn nur ganzzahlige Werte zugelassen werden.

Von den acht zu bildenden Grenzwerten$^{30}$ fallen die beiden Grenzwerte von $G_2$ zusammen (Wert 10) und die ungültigen Grenzwerte von $U_2$ und $U_4$ sind identisch (Wert 11), da nur ganzzahlige Werte erlaubt sind. Von diesen fünf Werten werden vier mit den Tests $t_1$ bis $t_4$ getestet. Betrachtet man die fünf Werte jedoch als acht Grenzwerte, so werden sieben davon mit $t_1$ bis $t_4$ getestet. In beiden Rechnungen fehlt nur ein Test für den (oberen) Grenzwert 9 von $U_3$.

Je nachdem, wie das Testwirksamkeitsmaß $TWM_G$ für die Grenzwertanalyse definiert wird, ergibt sich also $TWM_G = \frac{4}{10} = 40\%$ oder $TWM_G = \frac{7}{11} = 63,64\%$.

Ein anderes Ergebnis erhält man, wenn man von den Tests folgendes verlangt:

(K) Nur gültige Werte oder höchstens ein ungültiger Wert dürfen pro Test vorkommen.

In diesem Falle sind nur die Tests $t_2$ und $t_3$ erlaubt. Test $t_1$ ist nicht erlaubt, da er die ungültigen Äquivalenzklassen $U_1$ und $U_3$ abdeckt; Test $t_4$ ist nicht erlaubt, da er die ungültigen Klassen $U_2$ und $U_4$ abdeckt. Die erlaubten Tests $t_2$ und $t_3$ testen nur zwei Werte bzw. vier Grenzwerte. Also ist das Testwirksamkeitsmaß nur $TWM_G = \frac{4}{9} = 44,44\%$ oder $TWM_G = \frac{7}{11} = 63,64\%$.\(^{31}\)

<table>
<thead>
<tr>
<th>Test</th>
<th>Wert von $X$</th>
<th>abgedeckte Grenzwerte</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>0</td>
<td>(obere) Grenze von $U_1$</td>
</tr>
<tr>
<td>$t_2$</td>
<td>1</td>
<td>untere Grenze von $G_1$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>10</td>
<td>obere Grenze von $G_1$ und untere (und obere) Grenze von $G_2$</td>
</tr>
<tr>
<td>$t_4$</td>
<td>11</td>
<td>(untere) Grenze von $U_2$ und $U_3$</td>
</tr>
</tbody>
</table>

Tab. 6.5 Ganzzahlige Werte als Testdaten

Die alternativen Berechnungen bei Beispiel 6.4.7 zeigen, daß die Grenzwertanalyse nach verschiedenen Kriterien betrieben werden kann. Vor dem Testen sollte klar sein, welches Kriterium erfüllt werden soll.

Die Bestimmung der Testwirksamkeit der Grenzwertanalyse in Bezug auf Ausgabe- grenzwerte von aufgestellten Äquivalenzklassen wirft zusätzliche Probleme auf. Es muß für einen Test $t$ nicht festgestellt werden, ob die zugehörigen Eingabewerte

\(^{30}\) je vier bei den doppelseitigen Intervallen [1; 10] und [10; 20], wobei 10 doppelt vorkommt.

\(^{31}\) Allerdings ist wieder zu beachten, daß die Grenzen von $U_1$, $U_2$ und $U_3$ (0 und 11) mit ganzzahligen Werten nicht gemäß Forderung (K) getestet werden können, da stets $U_2$, $U_4$ oder $U_5$ gleichzeitig abgedeckt werden. Also läßt sich als Testwirksamkeitsmaß höchstens $TWM_G = \frac{4}{9} = 44,44\%$ oder $TWM_G = \frac{7}{11} = 63,64\%$ erreichen. (Es fehlt nur ein Test mit $x = 9$, der noch den Grenzwert von $U_3$ abdeckt.)

19 Aug 2002 22:14
Grenzwerte sind. Vielmehr müssen zu den Testeingaben die zugehörigen Ausgabewerte mit Hilfe der Spezifikation oder des Programms berechnet werden und es muß geprüft werden, ob diese Ausgabewerte Grenzwerte sind.

**Beispiel 6.4.8**
Für das Druckprogramm aus Beispiel 6.4.3 bzw. Kapitel 4.2.2 seien vier Testdaten \( t_1 \) bis \( t_4 \) gegeben, d. h. Eingabedateien mit verschiedenen Zeilenzahlen \( Z \), nämlich 0, 37, 45 und 900.

<table>
<thead>
<tr>
<th>Test</th>
<th>( Z )</th>
<th>( X )</th>
<th>( Y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>37</td>
<td>1</td>
<td>37</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>45</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>( t_4 )</td>
<td>900</td>
<td>20</td>
<td>45</td>
</tr>
</tbody>
</table>

**Tab. 6.6** Korrekte Werte für \( X \) und \( Y \) bei vier Testdaten des Druckprogramms

Dann ergeben sich bei einem korrekten Programm die in Tabelle 6.6 dargestellten Werte für die Anzahl \( X \) von bedruckten Seiten und die Anzahl \( Y \) von Zeilen auf der letzten Seite. Dabei werden \( X \) und \( Y \) folgendermaßen berechnet (vgl. Beispiel 6.4.3):

1. \( X = \text{aufrunden}(\frac{Z}{45}) \),
2. \( Y = 45 \), falls \( Z \mod 45 = 0 \) und \( Z \neq 0 \),
3. \( Y = Z \mod 45 \), falls \( Z \mod 45 > 0 \) oder \( Z = 0 \).

\( t_1 \) erzeugt die ungültigen Grenzwerte \( X = 0 \) und \( Y = 0 \), \( t_2 \) den gültigen Grenzwert \( X = 1 \), \( t_3 \) erzeugt die gültigen Grenzwerte \( X = 1 \) und \( Y = 45 \) und \( t_4 \) die gültigen Grenzwerte \( X = 20 \) und \( Y = 45 \) (vgl. Abschnitt 4.2.2). Es fehlen also nur die ungültigen Grenzwerte \( X = 21 \) und \( Y = 46 \) und der gültige Grenzwert \( Y = 1 \). Das Testwirksamkeitsmaß beträgt also \( \text{TWM}_G = \frac{1(3+2)}{4(1+4)} = \frac{5}{8} = 62,5\% \).

Wenn ein implementiertes Programm vorliegt, kann man die Ausgaben natürlich von Programmen erzeugen lassen und diese (automatisch) mit den aus den Ausgabêteilklassen ermittelten Grenzwerter vergleichen. In diesem Fall erzeugt ein falsches Druckprogramm (vgl. Beispiel 6.4.8) bei einer Datei mit \( Z = 46 \) Zeilen vielleicht den ungültigen Ausgabewert \( Y = 46 \).
6.4 Testdatenerzeugung und Messung der Testwirksamkeit

6.4.2.4 Messung der Testwirksamkeit bei der Ursache/Wirkungsgraph-Methode

Bei der UWG-Methode muß die Wertetabelle mit den zu erzeugenden Testfällen vorliegen (vgl. Abschnitt 4.2.3, Schritte 5 und 6), damit für generierte Testdaten die Testwirksamkeit bestimmt werden kann. Für ein Testdatum ist dabei zu bestimmen, welche Testfälle der Wertetabelle es abdeckt. Dazu muß für jede Eingabebedingung (Ursache), die bei dem Testfall erfüllt (bzw. nicht erfüllt) sein muß, festgestellt werden, ob das konkrete Testdatum diese Eingabebedingung erfüllt (bzw. nicht erfüllt). Das ist ebenso leicht möglich wie bei der Äquivalenzklassenbildung.

Das Testwirkungsmaß \( TWM_{UWG} \) ist dann:

\[
TWM_{UWG} := \frac{\text{Anzahl der abgedeckten Spalten der Wertetabelle}}{\text{Anzahl aller Spalten der Wertetabelle}}
\]

Bei der Testwirksamkeitsmessung für eine irgendwie ermittelte Testdatenmenge wird also (nur) Schritt 7 der UWG-Methode durch obiges Verfahren ersetzt. Die Schritte 1 bis 6 müssen vorher wie vorgesehen durchgeführt werden (vgl. Abschnitt 4.2.3).

6.4.3 Testdatenerzeugung und Messung der Testwirksamkeit für funktionsbezogene Testkriterien

För Funktionen, die durch alternative Auswahl oder kontrollierte Iteration aus einfachen Teilfunktionen komponiert sind, lassen sich folgendermaßen Testdaten erzeugen (vgl. Kapitel 4.3).

6.4.3.1 Alternative Auswahl

Bei der alternativen Auswahl der Art \( f = \text{if } b \text{ then } f_1 \text{ else } f_2 \) sind nach Satz 4.3.1-3(a) auf S. 102 alle Tests zu generieren, die mögliche Fehler im Entscheidungsprädikat aufdecken. Ein Verfahren dazu wird in Kapitel 9 vorgestellt. Um die Anforderungen von Satz 4.3.1-3(b) zu erfüllen, ist dann zu prüfen, ob für ein Testdatum \( t \) die Ungleichung \( f_1(t) \neq f_2(t) \) gilt. Dazu, und um die Fehlerfreiheit von \( f_1 \) und \( f_2 \) zu testen, muß man die Teilfunktionen aus ihrem Zusammenhang lösen und einzeln testen. Wenn man sich diesen Aufwand erspart, ist unklar, ob

a. die Funktion \( f \) trotz eines Fehlers von \( b \) "zufällig" einen richtigen Wert liefert, weil \( f_1(t) = f_2(t) \) gilt;

b. die Funktion \( f \) trotz eines Fehlers von \( f_1 \) oder \( f_2 \) einen richtigen Wert liefert, weil "zufällig" \( f_1(t) \) (oder \( f_2(t) \)) den richtigen Wert berechnet.

Falls für ein (für den Test von \( b \) geeignetes) Testdatum \( t \) der Fall \( f_1(t) = f_2(t) \) eintritt, ist ein anderes Testdatum \( t' \) zu generieren, welches
1. die Forderung \( f_1(t) \neq f_2(t) \) erfüllt und gleichzeitig

2. die bestehende Forderung an den Test von \( b \) erfüllt.

Diese Testdatenerzeugung kann äußerst schwierig sein, da ein System von (Un-)Gleichungen zu lösen ist. Dieses System ergibt sich bei der Berechnung der Bedingungen für \( t' \) durch symbolische (Rückwärts-)Rechnung (genauer siehe Kapitel 11 bzw. 12.3), um Forderung 1 zu erfüllen und den zusätzlichen Forderungen von 2 zu genügen (genauer s. Kapitel 9).

Da die Testdatenerzeugung schwierig ist, bietet sich die Testdatenerzeugung per Hand oder Zufall an mit anschließender Überprüfung des Erfolgs, d. h. der Testwirksamkeit. Gemäß Satz 4.3.1 muß dann für jedes Testdatum \( t \) nur ermittelt werden:

1. welche Forderung an das Testen von \( b \) durch den Test \( t \) erfüllt wird,

2. ob \( f_1(t) \neq f_2(t) \) gilt.

Für die Fälle, bei denen \( f_1(t) \neq f_2(t) \) gilt, können alle erfüllten Forderungen an den Test von \( b \) gezählt werden und alle Forderungen an den Test von \( b \) gegenübergestellt werden. Dieses Verhältnis ergibt das Testwirksamkeitsmaß \( TWM_f \) für die alternative Auswahl. \( TWM_f \) sagt nur aus, wie gut die Integration von \( f_1 \) und \( f_2 \) zur Gesamtintegration \( f \) getestet wurde; über die Korrektheit der Teilverfahren \( f_1 \) und \( f_2 \) wird dabei nichts ausgesagt. (Diese Teilverfahren sind vorab nach ähnlichem Muster zu testen, falls sie sich ebenfalls aus einfachen Funktionen komponieren lassen.)

**Beispiel 6.4.9**

Es sei angenommen, daß zum Test von \( b \) (gemäß Kapitel 9.1) 16 Tests benötigt werden. Es liegen 13 Tests vor, von denen 12 die Forderungen \( f_1(t) \neq f_2(t) \) erfüllen. Dann ist \( TWM_f = \frac{12}{15} = 73\% \). Prüft man die Bedingung \( f_1(t) \neq f_2(t) \) nicht ab, dann kann man nur eine obere Abschätzung für die Testwirksamkeit angeben: \( TWM_f \leq \frac{12}{16} = 81,25\% \).

**6.4.3.2 Kontrollierte Iteration**

Bei der **kontrollierten Iteration** der Art

\[
    f = \textbf{while } b(v) \textbf{ do } v \leftarrow g(v)
\]

sind die Testdaten ähnlich wie bei der alternativen Auswahl zu erzeugen (vgl. Satz 4.3.2 auf S. 104). Es sind wieder alle Tests zu generieren, die mögliche Fehler im Entscheidungsprädikat \( b(v) \) aufdecken, wobei zusätzlich gelten muß:

\[
    g_i(t) \neq g_j(t) \text{ für alle } i > j \geq 0.
\]
Wenn eine Obergrenze \( k \) für die Zahl der Iterationen bekannt ist oder berechnet werden kann, ist diese Prüfung durchführbar, andernfalls nicht. Wenn der Fall \( g_i(t) = g_j(t) \) eintritt, ist wieder mit einer aufwendigen symbolischen Rechnung ein anderes Testdatum zu berechnen, das die entsprechenden Forderungen erfüllt (vgl. Satz 4.3.2).

Die Berechnung eines Testwirksamkeitsmaßes \( TWM_{\text{while}} \) ist wiederum leichter möglich, da nur die Erfüllung der Forderungen von Satz 4.3.2 geprüft werden muß. Das ist allerdings auch nur möglich, wenn die oben aufgestellte Forderung nur für Werte von \( i \) und \( j \), die kleiner als eine Obergrenze \( k \) sind, geprüft werden muß. Das Testwirksamkeitsmaß \( TWM_{\text{while}} \) sagt (analog zu \( TWM_{ij} \)) nur etwas aus über die Güte des Integrationstests, d. h. welche Fehler bei der Integration von \( g \) in die Iteration (zur Berechnung von \( f \)) gemacht worden sein können. Über die Güte des Tests von \( g \) wird keine Aussage gemacht. (Die Teilfunktion \( g \) ist vorab nach ähnlichen Muster zu testen, falls \( g \) ebenfalls aus einfachen Funktionen aufgebaut ist.)

### 6.4.3.3 SADT- und SA-Diagramme


### 6.4.4 Testkriterien für Reihenfolgebedingungen und algebraische Spezifikationen

#### 6.4.4.1 Testdatenerzeugung für Reihenfolgebedingungen

Für Reihenfolgebedingungen (gegeben durch Pfadausdrücke oder endliche Automaten) ist die Testdatenerzeugung relativ einfach, wenn etwa das Kriterium *alle Transitionen* (s. Definition 5.1.2 auf Seite 118) erfüllt werden soll und bei einem Modul alle Operationsfolgen frei wählbar sind. In diesem Falle sind folgende Schritte durchzuführen:

1. den vollständigen, reduzierten, endlichen Automaten \( A_\ast(P) \) zu dem gegebenen Pfadausdruck \( P \) bestimmen bzw. den gegebenen endlichen Automaten reduzieren und vervollständigen (vgl. Kapitel 5.1),

\(^{32}\)Problematisch ist dies nur, wenn die Markierungen der Stellen nicht beschränkt sind und der modellierende Automat somit nicht endlich ist.
2. den Graph des Automaten $A_n(P)$ mit vollständigen Wegen (vom Anfangszustand zu einem Endzustand) überdecken, so daß jede Transition (Kantenbeschreibung) von $A_n(P)$ zu einem Weg gehört\footnote{Kanten mit Mehrfachbeschreibung sollen dazu vorher in mehrere Kanten mit Einfachbeschreibung aufgefüllt werden, damit die Kanten eindeutig den Transitionen entsprechen. Mit den genannten Schritten 1 und 2 ist das Kriterium alle Transitionen erfüllt. Für ein anderes Kriterium (z. B. Automatendichtigkeitstest oder alle korrekten/alle nicht korrekten Transitionen) ist eine andere Menge von Wegen zu bestimmen, was ebenfalls automatisch möglich ist.}


Schritt 2 erfordert die Lösung eines graphentheoretischen Problems (vgl. [GmV 86], S. 48). Als Vereinfachung des Problems kann der schließenfreie Automat $A_n(P)$ aus $A_n(P)$ konstruiert werden (siehe Beispiel 6.2.4 auf Seite 149). Dazu sind die starken Zusammenhangskomponenten von $A_n(P)$ zu bestimmen, ebenfalls eine graphentheoretische Aufgabe mit bekannten Lösungsverfahren (siehe z. B. Kapitel IV.6 in [Meh 84]). Für $A_n(P)$ sind dann nur geeignete Wege vom Anfangs- zum Endzustand zu bestimmen, was i. allg. durch einfaches Markieren der gewählten Transitionen (Kantenbeschriftungen) möglich ist.

**Beispiel 6.4.10**

Aus dem Automaten $A_n(P)$ aus Abbildung 5.4 wird der Automat $A_n(P)$, bei dem gegenüber $A_n(P)$ nur die zyklischen Kanten an den Zuständen 2 und 3 wegfallen (vgl. Beispiel 6.2.4 auf S. 149). Die vom Anfangszustand 1 ausgehenden Kanten $d$, $e$ und $f$ sind drei vollständige Wege $w_1$, $w_2$, $w_3$ in den Endzustand 3. Der von Zustand 1 mit Kante $b$ beginnende Weg läßt sich etwa zum vollständigen Weg $w_4 = be$ verlängern. Die von Zustand 1 ausgehende Kante $c$ läßt sich zum vollständigen Weg $w_5 = cde$ verlängern. Dann sind noch folgende Kanten nicht markiert (abgedeckt worden) (es wird jeweils der ausgehenden Zustand und die Kantenbezeichnung angegeben): $(2, f), (3, b), (3, e), (3, d), (3, f), (4, b), (4, e), (4, d), (4, c)$. Für die Kante $(2, f)$ wird durch Rückwärtsverfolgung der einlaufenden Kanten $(1, b)$ oder $(1, c)$ der Teilweg $(1, b)2, f)$ oder $(1, c)2, f)$ ermittelt, der Kante $(2, f)$ abdeckt. Mit einer Verlängerung zu einem vollständigen Weg $w_6$ kann dann etwa noch $(4, b)$ abgedeckt werden. Alle anderen Kanten erfordern einen zusätzlichen Weg, da sie direkt in den Endzustand 3 führen. Also werden noch sieben weitere Wege $w_7$ bis $w_{13}$ benötigt, um alle Kanten bzw. Transitionen abzudecken. Es werden also 13 Wege bzw. Tests benötigt (vgl. Beispiele 5.1.4 und 6.2.4 auf S. 119 und S. 149). Diese Wege sind noch an passenden Stellen um die Kanten zu ergänzen, welche die Schließung in Zustand 2 bzw. 3 ausführen. Beispielsweise ist $w_4 = be$ zu ergänzen zu $w'_4 = blee$ bedef (die Ergänzungen sind unterstrichen).

Bei obigem Verfahren und der gegebenen Reihenfolge der betrachteten, nicht überdeckten Kanten erhält man eine Minimalzahl von benötigten Tests, um alle Kanten bedeckt zu überdecken.
auszuführen. Verzichtet man auf die Konstruktion des schleifenfreien Automaten \( A_e(P) \) und wendet das Markierungsverfahren auf \( A_e(P) \) an, wird eventuell eine nicht minimale Anzahl von Wegen bzw. Tests generiert. Das passiert immer dann, wenn die Kanten in Schleifen als letztes betrachtet werden. In diesem Fall werden zusätzliche Wege konstruiert, um diese Kanten abzudecken, obwohl dies nicht notwendig ist, weil bestehende Wege passend ergänzt werden können. (Dieses Phänomen kann leider auch bei obiger Konstruktion für den Automaten \( A_e(P) \) auftreten, siehe Übung 6.8.) Es ist also abzuwägen zwischen dem Aufwand für die Konstruktion des schleifenfreien Automaten \( A_e(P) \) und dem Mehraufwand durch die Verwendung überflüssiger Tests. Dieses Aufwandsverhältnis hängt von der Schleifenstruktur des Automaten \( A_e(P) \) ab und ist daher leider nicht generell abschätzbar.

Falls die Folge der Operationen für das gegebene Modul nicht frei wählbar ist, sind die Parameterwerte zu bestimmen, von denen die Entscheidung für bestimmte Operationsfolgen abhängt. Die Bedingungen an diese Parameterwerte stellen ein System von Gleichungen oder Ungleichungen dar, welches zu lösen ist. Je nach Komplexität dieses (Un-)Gleichungssystems liegt also ein leicht lösbares oder sogar ein unlösbares Problem vor (vgl. „Erfüllbarkeit der Eingabebedingungen“ in Abschnitt 6.4.1.2, Seite 164).

**Beispiel 6.4.11**

Die Operationsfolgen und die Parameterwerte seien durch den in Abbildung 6.6 dargestellten Graphen bestimmt. Die Entscheidungsprädikate sind \( b_1, b_2 \) und \( b_3 \).

![Diagram](image)

**Abb. 6.6: Bedingte Operationsfolgen**

Die Werte der Parameter, von denen die Entscheidungen abhängen, werden im Zustandsbegriff zusammengefaßt. Zu Beginn sei dieser Zustand \( z \), nach Ausführung von Funktion \( f \) sei der Zustand \( f(z) \) etc. Damit die Folge \( fhi \) ausgeführt wird, muß also gelten:

\( (b_1(z) = \text{true}) \) und \( (b_2(f(z)) = \text{true}) \) und \( (b_2(h(f(z))) = \text{false}) \).

19 Aug 2002 22:14
Dieses Gleichungssystem ist zu lösen, d. h., es müssen Werte für den Anfangszustand \( z \) bestimmt werden, die dieses Gleichungssystem erfüllen.

Im konkreten Fall sei \( b_1 \) die Abfrage „\( A > 0 \)“, \( b_2 \) die Abfrage „\( B = 0 \)“; der Zustand \( z \) bestehe zu Beginn aus den Anfangswerten \( a \) und \( b \) der Variablen \( A \) und \( B \), d. h. \( z := (a, b) \); \( f \) ersetze den Wert von \( A \) durch sein Quadrat; \( h \) ersetze den Wert von \( B \) durch die Summe der Werte von \( A \) und \( B \). Dann gilt:

\[
\begin{align*}
b_1(z) &= true & g. \ d. \ w. & a > 0, \\
b_2(f(z)) &= true & g. \ d. \ w. & b = 0, \\
b_2(h(z)) &= false & g. \ d. \ w. & a^2 + b \neq 0.
\end{align*}
\]

Diese drei Bedingungen sind erfüllt g. d. w. wenn zu Beginn \( a > 0 \) und \( b = 0 \) gilt.

**6.4.4.2 Testdatenerzeugung für algebraische Spezifikationen**

Für die Testkriterien auf der Basis von algebraischen Spezifikationen ist die Testdatenerzeugung dem Vorgehen bei Pfadausdrücken vergleichbar. Für Axiome ohne Variablen ist das Erzeugungsproblem trivial; es ist genau der eine Fall zu testen, der durch die Gleichung spezifiziert ist (vgl. [GMH 81], S. 218). Für Axiome mit Variablen stellt sich das Problem der Erzeugung von Werten für die Variablen. Dazu muß aus den Gleichungen für die Konstruktoren ein Ausdruck abgeleitet werden, der die entsprechenden Terme beschreibt. Dies ist nicht automatisch möglich, da die Konstruktoren, die entsprechenden Gleichungen und der entsprechende Ausdruck nicht immer formal bestimmbar sind.

<table>
<thead>
<tr>
<th>Länge der Ableitung</th>
<th>Erzeugte Binärbäume</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LEER</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>BAUM (LEER, A, LEER)(^{34})</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>BAUM (LEER, A, BAUM (LEER, A, LEER))</td>
</tr>
<tr>
<td></td>
<td>BAUM (BAUM (LEER, A, LEER), A, LEER)</td>
</tr>
</tbody>
</table>

**Tab. 6.7** Erzeugung von Binärbäumen durch Ableitungen bestimmter Länge

**Beispiel 6.4.12**

Für den Binärbaum aus Beispiel 5.2.1 und die Gleichungen 6(b) und 6(d) sind LEER und BAUM als die Konstruktoren für die Term von Sorte Binärbaum zu identifizieren. Daraus (siehe Beispiel 5.2.1-4 auf S. 122) läßt sich (automatisch) nur eine kontextfreie Grammatik ableiten mit den beiden Regeln

\[^{34}\text{Eine zugehörige Ableitungsfolge}
\]

\[\text{Binärbaum} \Rightarrow \text{BAUM(Binärbaum, A, Binärbaum)} \Rightarrow \text{BAUM(LEER, A, Binärbaum)} \Rightarrow \text{BAUM(LEER, A, LEER)}\] besteht aus drei Schritten, hat also die Länge 3.

19 Aug 2002 22:14
6.4 Testdatenerzeugung und Messung der Testwirksamkeit

\[ \text{Binärbaum} \rightarrow \text{LEEP} \]
\[ \text{Binärbaum} \rightarrow \text{BAUM}(\text{Binärbaum}, \text{Alphabet}, \text{Binärbaum})^{35} \]

Alle Ableitungen bis zu einer bestimmten Länge, z. B. fünf, lassen sich daraus automatisch erzeugen (siehe Tabelle 6.7).


Damit sind die Tests für das Kriterium (fast) alle korrekten Kanten erzeugbar, indem für jede Variable einer Gleichung (eines algebraischen Axioms) die entsprechenden Terme in Postfixnotation angegeben werden und passend kombiniert werden (vgl. Beispiel 5.2.10).

Beim Kriterium (fast) alle Kombinationen von korrekten Kanten müssen dann nur alle Testwerte (Terme) für jede Variable einer Gleichung gespeichert werden und alle Kombinationen dieser Terme für alle Variablen einer Gleichung gebildet werden.

6.4.4.3 Messung der Testwirksamkeit

Die Schwierigkeit der Testdatenerzeugung liegt bei den Teststrategien für Reihenfolgebedingungen oder algebraische Spezifikationen vor allem in der Bestimmung der geeigneten Pfadausdrücke. Die Ableitung von Tests (d. h. von Wegen in dem entsprechenden endlichen Automaten) ist dann automatisch möglich, wie in Abschnitt 6.4.4.1 und 6.4.4.2 gezeigt wurde.

Bei der Bestimmung der Testwirksamkeit einer gegebenen Menge von Tests wird aber nur dieser letzte Ableitungsschritt vermieden. Die vorbereitende Ermittlung der Pfadausdrücke und endlichen Automaten ist weiterhin nötig, um das Testwirksamkeitsmaß überhaupt definieren zu können. Dieser Ansatz ist also nur zu empfehlen, wenn keine Algorithmen und Werkzeuge zur Ermittlung der notwendigerweise auszuführenden Wege (in den entsprechenden Automaten) vorhanden sind.

Für die Kriterien alle Transitionen, alle korrekten/einige nicht korrekte Transitionen und (fast) alle korrekten Kanten läßt sich das Testwirksamkeitsmaß TWM dann jeweils folgendermaßen definieren:

\[ \text{TWM} \]

\[ \text{ist der Quotient aus den ausgeführten Transitionen bzw. Kanten zu allen Transitionen bzw. Kanten (die zu den entsprechenden Automaten gehören)^{36}}. \]

\[ ^{35} \text{Die Ersetzung von „Alphabet“ durch einen konstanten Wert ist eine Entscheidung (Anwendung der Uniformitätshypothese), die nicht automatisch getroffen werden kann. Im folgenden sei dies geschehen, d. h. „Alphabet“ sei durch „A“ ersetzt.} \]

\[ ^{36} \text{Bei dem Kriterium alle korrekten/einige nicht korrekte Transitionen sind Klassen von Transitionen (statt einzelner Transitionen), die in den Fehlerzustand } F \text{ führen, zu betrachten.} \]
Bei dem Kriterium *Automatenäquivalenztest* ist auf alle Fälle die Menge der auszuführenden Wege im Automaten zu bestimmen und mit der tatsächlich ausgeführten Menge von Wegen zu vergleichen, um ein Testwirksamkeitsmaß zu bestimmen. Dieses Vorgehen bringt also überhaupt keinen Vorteil gegenüber der direkten, zielgerichteten Testdatenerzeugung.

Bei dem Testkriterium *(fast) alle Kombinationen von korrekten Kanten* ist eine Testwirksamkeismessung ebenfalls nicht zu empfehlen. Bei gegebenen Automaten zur Beschreibung der erlaubten Termen für die Variablen einer Gleichung ist nämlich nicht ohne weiteres klar, welche Kombinationen von Termen bereits getestet wurden und welche noch fehlen.

**Beispiel 6.4.13**

Es werde folgende Gleichung betrachtet:

*LEER? (BAUM(L,A,R)) = FALSCH*

mit dem konstanten Alphabetszeichen $A$ und zwei Variablen $L$ und $R$ vom Typ Binärbbaum. Für $L$ und $R$ beschreibe der endliche Automat aus Abbildung 5.7 auf S. 130 die erlaubten Termen in Postfixnotation (ohne die Konstante $A$). Es sei angenommen, daß für $L$ und $R$ die folgenden Termen bei den Tests vorkommen:

$$t_1 : l, t_2 : llbb, t_3 : llb, t_4 : lbbb.$$

Für $L$ komme noch der Term $t_5 : lbbb$ vor und für $R$ der Term $t'_5 : lbbb$, $t_5$ ist also nicht identisch mit $t_5$ (sondern eine Verlängerung um $lb$), deckt aber dieselben Kanten im Automaten aus Abbildung 5.7 auf S. 130 ab.

Kombiniert man also bei einem Test alle Terme $t_1$ bis $t_5$ für $L$ mit allen Termen $t_1$ bis $t_4$ und $t'_5$ für $R$, so werden durch diese Testmenge $T$ zwar alle Kanten des Automaten aus Abbildung 5.7 jeweils durch $t_1$ bis $t_5$ (bzw. $t_1$ bis $t_4$ und $t'_5$) abgedeckt, aber es gibt keine Menge $U$ von Termen mit der Eigenschaft, daß durch $T$ genau "alle Kombinationen von Kanten" bzgl. $U$ ausgeführt werden. Dies würde gerade bedeuten:

1. alle Terme aus $U$ decken zusammen alle Kanten des Automaten ab,

2. für alle $t$ aus $U$ und alle $u$ aus $U$ gibt es einen Test in obiger Testmenge $T$, der $L$ durch $t$ ersetzt und $R$ durch $u$ ersetzt.

Dieses negative Ergebnis liegt gerade daran, daß $t_5$ und $t'_5$ nicht identisch sind.

Wählt man aber als $U$ die Menge mit $t_1$ bis $t_5$ und $t'_5$, so fehlen in der Testmenge $T$ wiederum die Kombinationen mit $L = t'_5$ und $R = t_5$, obwohl sie im Automaten aus Abbildung 5.7 keine weiteren Kanten (gegenüber $L = t_5$ bzw. $R = t'_5$) abdecken.

Obiges Beispiel legt also folgenden Schluß nahe:

Das Testkriterium *(fast) alle Kombinationen von korrekten Kanten* läßt sich nur vernünftig erfüllen, wenn gezielt eine geeignete Menge von Termen erzeugt wird und dann alle Kombinationen gebildet werden (wie bei der Testdatengenerierung vorgesehen).

**Beispiel 6.4.1.4**
Beim Test „\(baum(baum(\text{leer}, A, \text{leer}), A, baum(\text{leer}, A, \text{leer}))\)“ sind die Aufrufe von „\(baum\)“ und „\(\text{leer}\)“ zu **instrumentieren**, nicht jedoch die Konstante „\(A\)“. Schreibt man jeweils „\(b\)“ beim „\(Baum\)-Aufruf, und „\(l\)“ beim „\(leer\)-Aufruf in eine Protokolleitei, ergibt sich folgende Postfixsequenz: „lbbllbb“ (vgl. Term \(t_4\) aus Beispiel 5.2.10). Diese Postfixsequenz führt also die folgenden Transitionen (Zustand, Eingabesymbol) im Automaten aus Abbildung 5.7 aus: (1, l), (2, l), (3, b), (7, l), (8, l), (9, b), (10, b). Damit werden schon 7 der 13 Transitionen des Automaten abgedeckt.

### 6.5 Übungen

**Übung 6.1:**

(a) Konstruieren Sie einen Fall, bei dem die Grenzwertanalyse die Äquivalenzklassenbildung **nicht** (strikt) enthält (vgl. Begründung 1 zu Abbildung 6.1).

Hinweis: In dem konstruierten Fall muß die Wahl eines ungültigen Grenzwer tes einen ungültigen Wert für eine andere Äquivalenzklasse implizieren, obwohl dies für beliebige Werte aus den beiden Äquivalenzklassen **nicht** gilt.

(b) Diskutieren Sie, ob ein solcher Fall „konstruiert“ ist oder einer sinnvollen, realistischen Äquivalenzklassenbildung entspricht.

**Übung 6.2:**

Zeigen Sie durch zwei entsprechende (Gegen-)Beispiele, daß entwurfsoorientierte Äquivalenzklassenbildung **unvergleichbar** ist mit der (spezifikationsorientierten) Grenzwertanalyse (vgl. Begründung 3 zur Aussage von Abbildung 6.1).

**Übung 6.3:**

Zeigen Sie die Unvergleichbarkeit der Kriterien fast alle Kombinationen von Kanten und alle korrekten Kanten an folgendem Beispiel, indem Sie entsprechende Tests angeben. Eine Gleichung \(G\) lautet: \(atom2(moleküle(x1, x2)) = false\).
Für $x_1$ und $x_2$ seien die erlaubten Termen (in Postfixnotation) exakt mit den regulären Ausdrücken $R_1$ und $R_2$ [mit $R_1 = R_2 = (a|b)^+; c$] beschrieben (vgl. Definition 5.2.3 auf S. 131).

Zeigen Sie zuerst, daß der reguläre Ausdruck $R' = (a; b)^+; c$ eine echte Teilmenge davon beschreibt.

Hinweis: Die reduzierten endlichen Automaten $A_1$ (zu $R_1 = R_2$) und $A'$ (zu $R'$) haben die in Abbildung 6.7 dargestellte Struktur.

**Abb. 6.7: Automaten zu regulären Ausdrücken**

**Übung 6.4:**

(a) Beweisen Sie die Beziehung zwischen der Anzahl der Tests für die Grenzwertanalyse ($t_g$) und für die Äquivalenzklassenbildung ($t_\Xi$) (s. Seite 146). Vergleichen Sie jeweils die minimalen Werte von $t_g$ und $t_\Xi$ und die maximalen Werte von $t_g$ und $t_\Xi$.

Setzen Sie dabei folgendes voraus:

i. Alle Bedingungen $b$ seien vom Typ „$u \ rel_1 x \ rel_2 o^n$, wobei $\ rel_1$ und $\ rel_2$ die Relationen $\leq$ oder $<$ sind.

ii. Die Menge der Ausgabebedingungen $AB$ sei leer.

(b) Beweisen Sie die Beziehungen 6.10 bis 6.13 zwischen dem Maximalwert von $t_{uwg}$ (Anzahl der Tests bei der UWG-Methode) einerseits und den Minimalwerten und den Maximalwerten von $t_g$ und $t_\Xi$ andererseits (s. Abschnitt 6.2.1). Machen Sie dabei die gleichen Voraussetzungen wie bei Teil (a).

**Übung 6.5:**

Beweisen Sie die Beziehungen 6.14 und 6.15 zwischen den Anzahlen $t_{i;j}$, $t_{white}$ und $t_b$ der benötigten Tests für die entsprechenden funktionalen Formen (siehe Abschnitt 6.2.1).
Übung 6.6:
In Kapitel 6.2 wurde versucht, die Anzahl der Tests eines zusammengesetzten Pfadausdrucks aus der Anzahl der Tests seiner Komponenten zu berechnen. Gebe Sie Gegenbeispiele zu den dort angegebenen Gleichungen (c), (d) und (e) an:

(c) $t(p; q) = \max(t(p), t(q))$,
(d) $t(p) = t(p)$,
(e) $t(p^+) = t(p)$.

Hinweis:
• Beachten Sie, daß die zu den Pfadausdrücken gehörigen Automaten mehrere Endzustände haben können (z. B. bei Alternativen).
• Sie können (d) und (e) mit einem gemeinsamen Gegenbeispiel widerlegen.

Übung 6.7:
 Beweisen Sie die Beziehung $t_{\text{a}a}(A_{\text{a}}(P)) \geq z \ast t_{\text{a}d}(P)$ zwischen den Testanzahlen $t_{\text{a}a}(A_{\text{a}}(P))$ und $t_{\text{a}d}(P)$ für den Automatenäquivalenztest und das Kriterium alle Transitionen und die Anzahl der Zustände des Automaten $A_{\text{a}}(P)$ (siehe Gleichung 6.19 in Abschnitt 6.2.2). Nehmen Sie dabei an, daß der korrekte Automat die gleiche Anzahl von Zuständen hat.
 Hinweis: $A_{\text{a}}(P)$ ist ein vollständiger, deterministischer Automat.

Übung 6.8:
Ziehen Sie: Bei einer einfachen Strategie zur Überdeckung aller Transitionen eines schleifenfreien Automaten wird nicht unbedingt die minimale Anzahl von Wegen (Tests) erzeugt.
Wählen Sie den Automaten $A_{\text{a}}(P)$ aus Beispiel 6.4.10 (zum Automaten $A_{\text{a}}(P)$ aus Abb. 5.4 auf S. 118) und als Strategie den Tiefendurchlauf (DFS = depth first search), wobei die Transitionen alphabetisch angeordnet werden: $b, c, d, e, f$. (Der erste erzeugte Weg ist demnach $bde$, der zweite $bdc$.)

Übung 6.9:
Nehmen Sie für den Pfadausdruck $P = (b|e)^+; [d; e|f]$ aus Beispiel 5.1.1 verschiedene Fehler an, z. B. das Weglassen der Klammerung für „$e|f$“, d. h. der korrekte Ausdruck ist $P' = (b|e)^+; [d; (e|f)]$.
Welche dieser Fehler werden durch die 13 Testfolgen aus Beispiel 5.1.4 auf S. 119, die das Kriterium alle Transitionen erfüllen, entdeckt? (Setzen Sie dabei voraus, daß ein Reihenfolgeorakel für Anfangsstücke von Testfolgen angibt, ob sie für den entsprechenden Automaten korrekt sind oder nicht.)
Übung 6.10:
Erzeugen Sie (zum Testen der Gleichungen für den Typ Binärbaum)

(a) alle Binärbäume mit bis zu sieben Ableitungsschritten aus der Grammatik mit den folgenden Regeln. (Dabei ist B das einzige zu ersetzende Nichtterminalzeichen, a ist eine konstante Wurzelbezeichnung.)

\[
(R1) : B \to l
\]
\[
(R2) : B \to b(B, a, B)
\]

(* l steht für „leer“ *)

(* b steht für „Baum“ *)

(b) Erzeugen Sie alle Binärbäume mit einer Höhe \( \leq 3 \).

Bei welchem Verfahren werden mehr Binärbäume erzeugt?

Übung 6.11:
Für das (Light-Hansa-)Beispiel 6.1.2 mit den Eingabebedingungen \( E1, E2, E3 \), die sich auf die Variablen Alter, Gewicht und IQ (Intelligenzquotient) beziehen, seien die in Tabelle 6.8 dargestellten konkreten Testdaten \( T1 \) bis \( T4 \) gewählt.

<table>
<thead>
<tr>
<th>Alter</th>
<th>Gewicht</th>
<th>IQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>20</td>
<td>60</td>
</tr>
<tr>
<td>T2</td>
<td>18</td>
<td>49</td>
</tr>
<tr>
<td>T3</td>
<td>65</td>
<td>80</td>
</tr>
<tr>
<td>T4</td>
<td>30</td>
<td>50</td>
</tr>
</tbody>
</table>

Tab. 6.8 Konkrete Testdaten für die Light-Hansa

Bestimmen Sie die auszuführenden Testfälle nach der UWG-Methode und ermitteln Sie die Testwirksamkeit \( TWM_{UWG} \) (siehe S. 171) der Testdaten \( T1 \) bis \( T4 \).

6.6 Verwendete Quellen und weiterführende Literatur

Wegen der vielen Quellen sind die Literaturreferenzen in Kapitel 6 jeweils bei der entsprechenden Textpassage angegeben worden oder die Quellen wurden schon in Kapitel 4 oder 5 angegeben.

Zusätzliche sei auf folgendes verwiesen: Vor den Untersuchungen von Duran/Ntafos und Hamlet/Taylor gab es schon von Duran/Workowsky Evaluierungen des Zufallstests (im Vergleich zu anderen Testarten), siehe [DuW 79]. Eine genauere Charakterisierung der Bedingungen, von denen es abhängt, ob die Äquivalenzklassenmethode besser oder schlechter als die Zufallsmethode ist, geben z. B. Chen/Yuan (s. [ChY 96]). Dort wird das Erfolgsmäß „mindestens einen Fehler finden“ auch mit dem Erfolgsmäß „Anzahl der gefundenen Fehler“ verglichen.

Die Idee, zur Testfallgenerierung bei Pfadausdrücken und endlichen Automaten den schließenfreien Automaten \( A_\epsilon(P) \) heranziehen, entspricht dem Konzept des reduzierten Graphen (reduced graph) bei Gabow et al. (s. [GMO 76], S. 228).
Zusammenfassung von Teil II

Teil II hat sich dem spezifikationsorientierten Testen gewidmet. Die Testmethoden unterscheiden sich dabei aufgrund der Spezifikationsart und des Vorgehens. Folgende Spezifikationsarten wurden betrachtet:

- Fallunterscheidung durch Datenbereiche (Kap. 4.2)
- funktionale Zerlegung (Abschnitte 4.3.1 und 4.3.2)
- Datenflußspezifikation (SA, SADT, s. Abschnitt 4.3.3)
- Petri-Netze und endliche Automaten (Abschnitt 4.3.3)
- Reihenfolgebedingungen (Pfadeausdrücke, endliche Automaten, s. Kap. 5.1)
- algebraische Spezifikationen (Kap. 5.2)

Das Vorgehen beim Testen ist vor allem durch die Spezifikationsart festgelegt, läßt sich aber teilweise noch variieren.

Beim datenbereichsbezogenen Testen kann unsystematisch vorgegangen werden (Zufallstest und Fehlererwartungsmethode) oder systematisch, durch Orientierung an den Eingabe- (oder Ausgabe-)Bedingungen und den zugehörigen Äquivalenzklassen von Testdaten. Die Frage der Testdatenauswahl aus Äquivalenzklassen führt zur Grenzwertanalyse und zur Verwendung spezieller Werte; die Frage der Kombination verschiedener Eingabebedingungen führt zur Ursache/Wirkungsgraph-Methode (UWG-Methode). Bei der Spezifikationsart funktionale Zerlegung (funktionale Formen) bietet sich nur ein aufsteigendes (bottom up) Testen an: bei gegebenen Tests für $b$, $f_1$ und $f_2$ kann anschließend die alternative Auswahl [$f = if b$ then $f_1$ else $f_2$], die kontrollierte Iteration [$g = repeat v ← f_i(v)$ until $b(v)$ oder $g = while b(v) do v ← f_i(v)$] und die Sequenz [$h = f_1; f_2$] getestet werden.

Bei den datenbereichsbezogenen Testmethoden schneidet die UWG-Methode unter Aufwandsge-589
sichtspunkten (Anzahl der Testdaten) besser ab als die Äquivalenzklas-590
sen- und Grenzwertmethode (Abschnitt 6.2.1, Gleichungen 6.10 bis 6.13). Bei einem591
Vergleich der Testmethoden ist aber auch der Aufwand für das (automatische) Er-592
zeugen der Testdaten zu beachten bzw. bei einer manuellen Testdatenerzeugung der593
Aufwand für die Messung der Testwirksamkeit (Kap. 6.4). Dabei zeigt sich, daß eine594
Automatisierung des Testens nur in Teilaspekten möglich ist, insbesondere, wenn595
der Nutzen des Testens (Art und Anzahl der aufgedeckten Fehler) ausreichend sein596
soll. Ein formaler Vergleich des Zufallstests mit der Äquivalenzklassenmethode zeigt597
z. B., daß – mit entsprechendem Gespür – kleine Äquivalenzklassen mit einer hohen598
Fehlerrate gebildet werden müssen, damit die Äquivalenzklassenmethode effektiv ist599
(s. Abschnitt 6.3.1.2 f.).

Insgesamt zeigt sich, daß mit spezifikationsorientiertem Testen nur 24% bis 30%600
der Fehler zuverlässig gefunden werden, bei der Kombination von Verfahren etwa601
46% bis 49% 37. Dabei werden vor allem Auslassungsfehler (fehlende Fallunterschei-602
dungen oder Teilfunktionen), Initialisierungsfehler, Ablauflogikfehler und fehlerhaft603
realisierte Teilfunktionen entdeckt.

Die restlichen Fehler können nur gefunden werden, wenn die Feinheiten der (fehler-604
haften) Implementation des Programmsystems betrachtet werden. Das ist der Inhalt605
von Teil III.

37 Nur in kleinen Programmen mit einfachen Datenbereichen konnten in zwei Fallstudien ca. 80%606
bis 90% der künstlich erzeugten Fehler gefunden werden.