5.3 Korrektheit und Verifikation

Korrektheit bedeutet, dass ein Algorithmus oder ein Programm das in der Spezifikation beschriebene Problem für beliebige Eingabedaten korrekt löst.

Die Korrektheit kann immer nur in Bezug auf eine Spezifikation gezeigt werden!

Definitionen

- Ein Programm **terminiert**, wenn es für alle Eingaben letztendlich anhält.
- Ein Programm ist **partiell korrekt**, wenn das Ergebnis im Falle des Terminierens immer korrekt ist.
- Ein Programm ist **total korrekt**, wenn es terminiert und partiell korrekt ist.

Es gibt zwei **Techniken** zum Prüfen der Korrektheit:

- Testen
- Verifikation (formaler Beweis der Korrektheit)

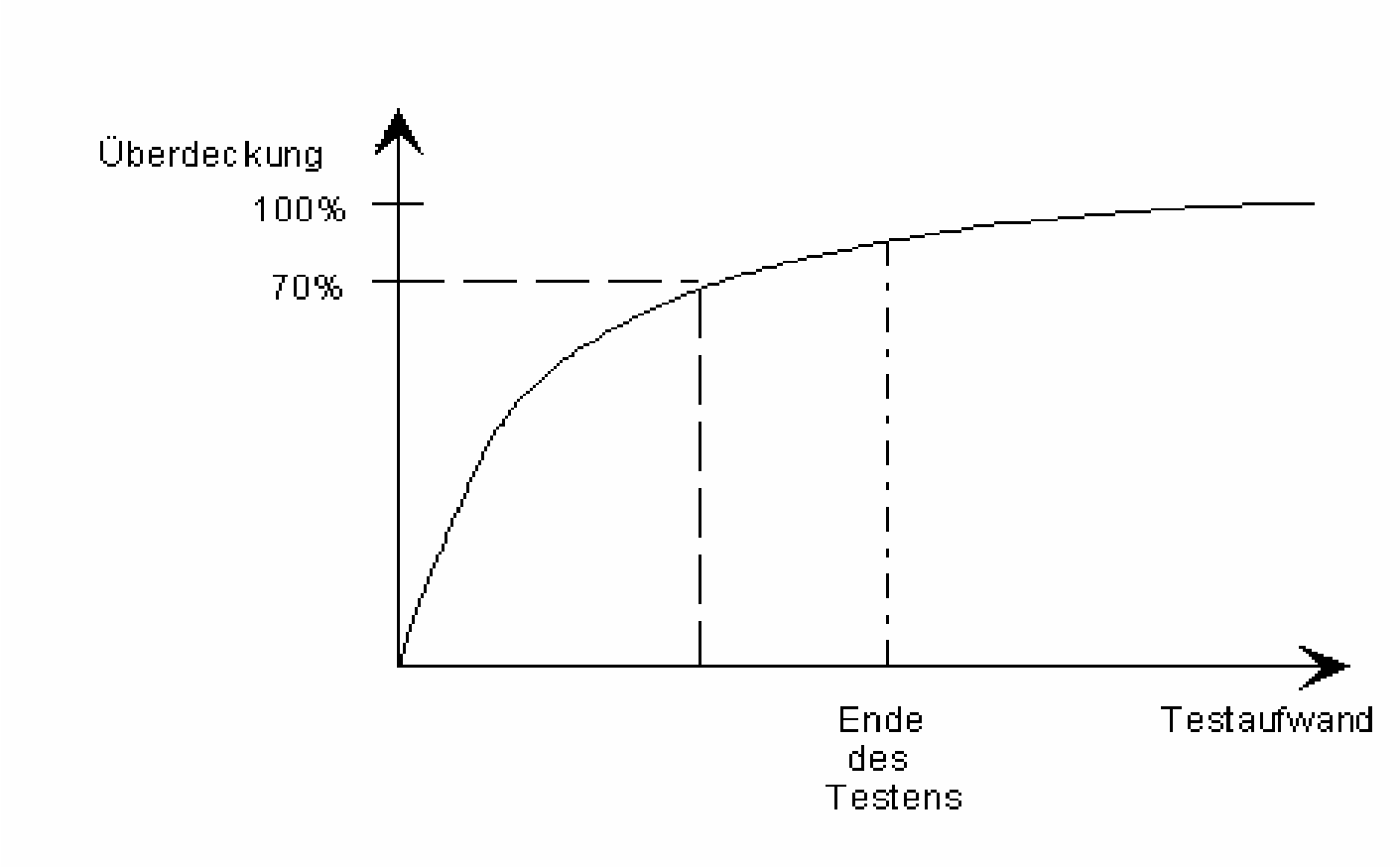
Testen

Als **Testen** (engl.: debugging) bezeichnet man die Ausführung eines Programms für eine Menge von Testdaten.

Die Menge der Testdaten ist endlich und in fast allen praktischen Fällen sehr viel kleiner als die Menge der möglichen Eingabedaten. Deshalb kann Testen nur das Vertrauen in die Korrektheit eines Programms erhöhen, nicht aber die Korrektheit beweisen.

In der Praxis erweist sich die Auswahl einer angemessenen Testdatenmenge als sehr schwierig. Ebenso schwierig ist die Analyse der Überdeckung eines Programms durch gegebene Testdaten. Hierzu hat man im Software Engineering Hilfsmittel entwickelt.

Ablauf des Testens



Beispiel für das Testen (1)

Es ist ein Programm zu schreiben, das zwei ganze Zahlen multipliziert. Als Operationen sollen nur Addition, Subtraktion, Multiplikation mit 2 und Division durch 2 zulässig sein.

Beispiel für das Testen (2)

```
1   public int mult(int x1, int y1) {
2     int x, y; long z;
3     z = 0;
4     if ((x1 > 0) && (y1 > 0)) {
5        x = x1;
6        y = y1;
7        while (x != 0) {
8          if (x % 2 == 1)
9              z = z + y;
10         y = 2 * y;
11         x = x/2;
12       }
13    }
14    return z;
15  }
```

Erster Ansatz: Testen als Schwarzer Kasten

(engl: black box testing)

Testen für alle x1, y1 < 10^{12} ohne Kenntnis der Programmstruktur ergibt

$10^{12} * 10^{12} = 10^{24}$

Testfälle.

Bei 1ms pro Ausführung sind das ca. 3*10^{13} Jahre Rechenzeit.

Fazit

Es kann selbst beim Testen mit Zufallszahlen nur ein sehr kleiner Prozentsatz der möglichen Eingabedaten getestet werden.

Zweiter Ansatz: Testen als Weißer Kasten

(engl: white box testing)

Testen unter Kenntnis der Programmstruktur. Die Auswahl der Testdaten erfolgt so, dass eine möglichst gute Überdeckung des Programmcodes erreicht wird.

Anweisungstest

Die Auswahl der Testdaten erfolgt so, dass jede Anweisung im Programm mindestens einmal ausgeführt wird.

Vorteil: unnötige Anweisungen werden entdeckt (solche, die durch keinen Testdatenfall erreicht werden können). Es sind nur relativ wenige Testfälle nötig.

Nachteil: Viele wichtige Fälle werden überhaupt nicht getestet.

Beispiel für den Anweisungstest

```
if (b) a;
```

Wenn die Bedingung **b** erfüllt ist, wird die Anweisung **a** ausgeführt. Damit ist sowohl die **if**-Anweisung als auch **a** einmal durchlaufen worden. Der Fall "**b == false**" wird nicht getestet!

Verzweigungstest

Gegeben sei das Flussdiagramm des Programms. Dann werden die Testdaten so ausgewählt, dass jede Verzweigung (Verbindungslinie zwischen Elementen des Diagramms) mindestens einmal durchlaufen wird.

Vorteil: bessere Überdeckung als mit dem Anweisungstest.

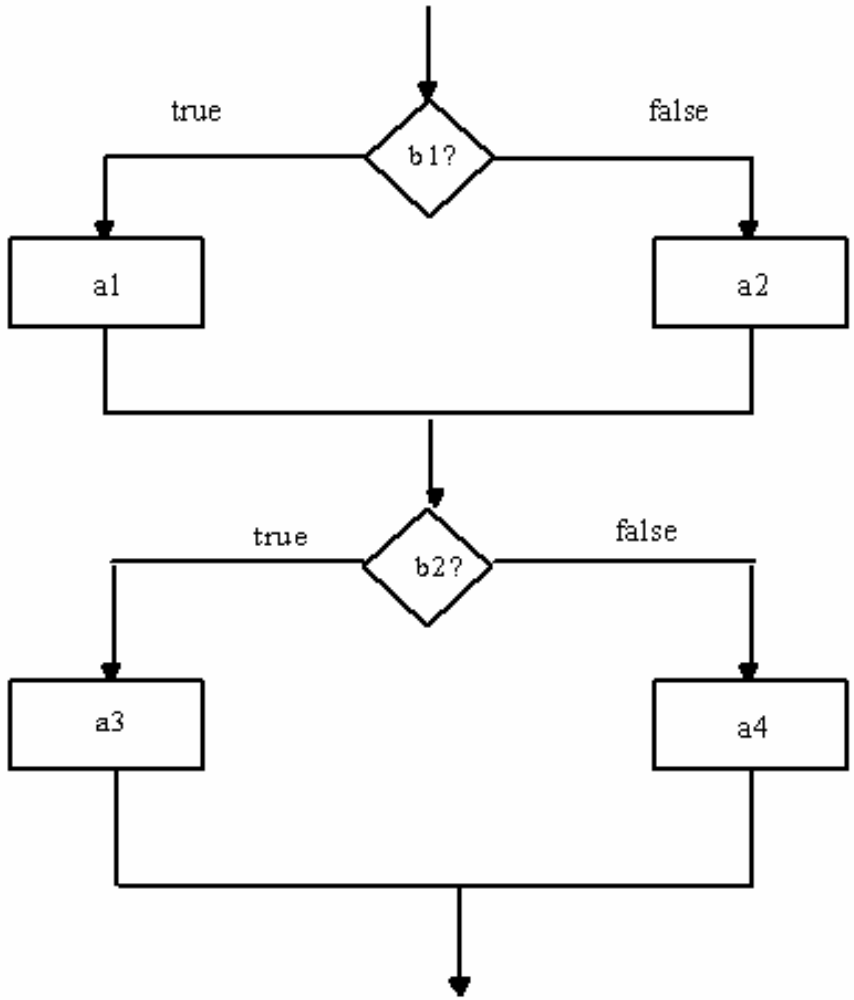
Nachteil: es werden immer noch nicht alle relevanten Fälle erfasst.

Beispiel für den Verzweigungstest (1)

```
if (b1) a1; else a2;
if (b2) a3; else a4;
```

Es gibt vier mögliche Anweisungsfolgen für a1 bis a4:

```
a1;a3
a1;a4
a2;a3
a2;a4
```



Beispiel für den Verzweigungstest (2)

Gilt nun für alle Testdatenfälle, dass sie entweder die Kombination

`b1 true  und b2 false` oder

`b1 false und b2 true`

enthalten, so sind die Anforderungen an den Verzweigungstest erfüllt (alle Pfade im Flussdiagramm werden mindestens einmal durchlaufen), aber die Ablauffolgen

`a1;a3`

`a2;a4`

werden nicht getestet.

Wegetest (1)

Im **Wegetest** werden alle möglichen Ablauffolgen von Anweisungen mindestens einmal durchlaufen. Für das Beispiel **`mult`** sehen wir:

x1	y1	durchlaufene Anweisungen
0	beliebig	3, 4, 14
beliebig	0	3, 4, 14
1	1	3, 4, 5, 6, 7, 8, 9, 10, 11, 14
2	1	3, 4, 5, 6, 7, 8, 10, 11, 7, 8, 9, 10, 11, 7, 14
3	1	…

Es gibt zu viele mögliche Ablauffolgen wegen der Schleife!

Wegetest (2)

Reduktion in der Praxis:

a) Schleife 0-mal durchlaufen

b) Schleife 1-mal durchlaufen

c) Schleife k-mal durchlaufen, $k \geq 2$, fest.

Suche nach typischen Fehlern

Mit etwas Erfahrung stellt man fest, dass Programmierer immer wieder Fehler machen. Man entwirft daher die Testdaten so, dass sie nach diesen typischen Fehlern suchen.

Beispiele:

- Feldindex außerhalb der Grenzen des Feldes (arrays).
- Anzahl der Schleifendurchgänge +/- 1.
- Namenskollision zwischen lokalen und globalen Variablen und Parametern.
- Rundungsfehler bei Gleitkommaoperationen.

Durch Testen können auch Compilerfehler oder Fehler in den Laufzeitroutinen gefunden werden (im Gegensatz zur Verifikation!).

Code-Inspektion (Code Review)

Programmcode kann nicht nur durch den Rechner, sondern auch durch den Programmierer getestet werden. Ein in der Praxis häufig eingesetztes und erstaunlich wirksames Verfahren ist die **Code-Inspektion** (code review, "Schreibtischtest"). Dabei wird der Programmcode eines Programmierers von einem anderen Programmierer auf mögliche Fehler untersucht.

Diese Methode erzieht zugleich zur strukturierten Programmierung und zur Dokumentation im Code (durch Kommentare).

Verifikation

Unter **Verifikation** versteht man den **formalen Beweis** der Korrektheit eines Programms durch mathematische Methoden.

Dazu muss bewiesen werden, dass das Programm **für beliebige Eingabedaten** terminiert und jeweils korrekte Ergebnisse liefert.

Beweis für das Terminieren

Programme ohne Schleifen terminieren immer.

Für Schleifen gilt: Durch mindestens eine Anweisung im Rumpf der Schleife muss eine Variable derart verändert werden, dass nach einer endlichen Anzahl von Durchläufen die Endebedingung erfüllt ist.

Beweistechnik

- Man suche einen Ausdruck A mit Variablen aus der Schleifenbedingung, dessen Werte in einer endlichen geordneten Menge **M** liegen (z. B. in einem endlichen Intervall aus den ganzen Zahlen).
- Man zeige, dass A bei jedem Schleifendurchgang streng monoton wächst (oder abnimmt).

Beispiel für einen Beweis des Terminierens

In Zählschleifen ist die Zählvariable der Ausdruck A.

Anderes Beispiel

Für die **`while`**-Schleife im Algorithmus **`mult`** gilt:

1. `0 ≤ x ≤ x1`
2. Die Anweisung

   `x = x/2`

   lässt **`x`** in jedem Durchlauf monoton abnehmen.

Die Variable **`x`** wird also mit Sicherheit den Wert 0 erreichen und damit die Schleife terminieren lassen.

Ähnlich kann man das Terminieren von rekursiven Prozeduren oder Funktionen beweisen.

Terminierung rekursiver Algorithmen

Ähnlich wie Terminierung der Schleifen

Beispiel:

```
public int fakultaet (int k) {
 if (k == 0)
  return 1;
 else
 return (k * fakultaet (k - 1));
}
```

A ist hier **k**, der Wert nimmt monoton ab, untere Grenze ist 0.

Schwieriger zu zeigen bei indirekter Rekursion: A ruft B, B ruft A

Beweis der Korrektheit von Programmstücken

Der formale Beweis der Korrektheit eines vollständigen Programms für alle möglichen Eingabedaten ist in der Praxis meist sehr schwierig oder unmöglich. Man kann aber manchmal die Korrektheit von wichtigen Programmstücken, insbesondere Schleifen, beweisen.

Beweis durch Induktion

Analog zur Mathematik beweist man:

1. Die Berechnungen in der Schleife sind beim ersten Durchlauf korrekt.
2. Angenommen sie sind beim n-ten Durchlauf korrekt, dann sind sie auch beim n+1-ten Durchlauf korrekt.

Beispiel (1)

```
public long potenziere(int x) {
// gibt 2x zurück, wobei x als nicht-negative Ganzzahl angenommen wird
int i; long summe;
summe = 1;
for (i = 0; i < x; i++) {
  summe += summe;
  // ***
}
  return summe;
}
```

Beispiel (2)

1. Das erste Mal, wenn *** erreicht wird, ist

   summe = 2^n (also 2^1)

2. Nehmen wir an, dass *** zum **n**-ten Male erreicht wird, dann ist

   summe = 2^n.

   Beim (n + 1)-ten Male, ist dann

   summe = $2^n + 2^n = 2^{n+1}$.

3. Deshalb gilt für jedes n, dass **summe = 2^n** ist, sobald *** zum **n**-ten Male erreicht wird.

Beispiel (3)

4. Wird die **`return`**- Anweisung je erreicht, dann gibt es zwei Möglichkeiten:
   - Entweder wurde die Schleife nicht durchlaufen, so dass in diesem Fall **`x = 0`** ist und deshalb der Anfangswert von **`summe`** nicht geändert wird. Dann ist **`summe = 1`**, was gleichbedeutend mit 2^0 ist.
   - Oder die Schleife wurde durchlaufen, so dass in diesem Falle die mit *** markierte Stelle **x**-mal erreicht wurde. Somit ist **`summe = 2`**$^{\mathbf{x}}$.

Es wird also auf jeden Fall 2^x ausgegeben, sofern nur die Ausgabeanweisung erreicht wird.

Zusicherungen (assertions) (1)

Beim Beweisen (Verifizieren) von Programmstücken muss man sich klar machen, welche Voraussetzungen an einer bestimmten Programmstelle gelten.

Zusicherungen (assertions) sind logische Ausdrücke über Programmvariablen. Sie beschreiben, welche logischen Annahmen an einer bestimmten Stelle im Programm gemacht werden können.

Zusicherungen (assertions) (2)

Gegeben sei folgende Situation:

...

assertion1;

    statement1;

    statement2;

    statement3;

assertion2;

...

Wenn es nun gelingt, unter der Annahme von assertion1 und unter Verwendung der statements den Ausdruck assertion2 herzuleiten, hat man die Korrektheit des Programmstücks formal bewiesen.

assertion1 heißt auch **pre**-condition, assertion2 heißt **post**-condition.

Algorithmische Ableitungsregeln

Gilt die Aussage P vor Ausführung von Algorithmus A und beschreibt die Aussage Q die funktionale Abhängigkeit der Ausgabegrößen von den Eingabegrößen, so kann aus der Gültigkeit von P und der Ausführung von A die Gültigkeit von Q nach der Ausführung abgeleitet werden.

In Zeichen: [P] A [Q].

[P] A [Q] bedeutet nichts anderes als

$$[P] \wedge A \rightarrow [Q]$$ (logische Implikation)

Zerlegung in Unteralgorithmen

Ist eine Zerlegung von A in Unteralgorithmen $A_1,...,A_n$ gegeben, für die jeweils

$$[P_i]A_i[Q_i], 1 \leq i \leq n$$

gilt, so muss

$$P \rightarrow P_1, \quad Q_n \rightarrow Q, \quad Q_{i-1} \rightarrow P_i, 1 \prec i \leq n$$

gelten.

Wenn darüber hinaus alle A_i terminieren, ist A (total) korrekt.

Beweis des Terminierens der Zerlegung

1. Zunächst macht man die Induktionsannahme, dass alle Unteralgorithmen terminieren.
2. Dann zeige man, dass die Zerlegung terminiert, d.h. keine unendliche Schleife eintritt, und schließlich,
3. dass bei der Zusammensetzung der Unteralgorithmen nur endlich viele Wiederholungen möglich sind.
4. Dann wendet man die Regel auf die nicht-elementaren Unteralgorithmen an.

Ableitungsregel für die if-Anweisung

[P]

if (B) A_1

else A_2

[Q]

Ableitungsregel

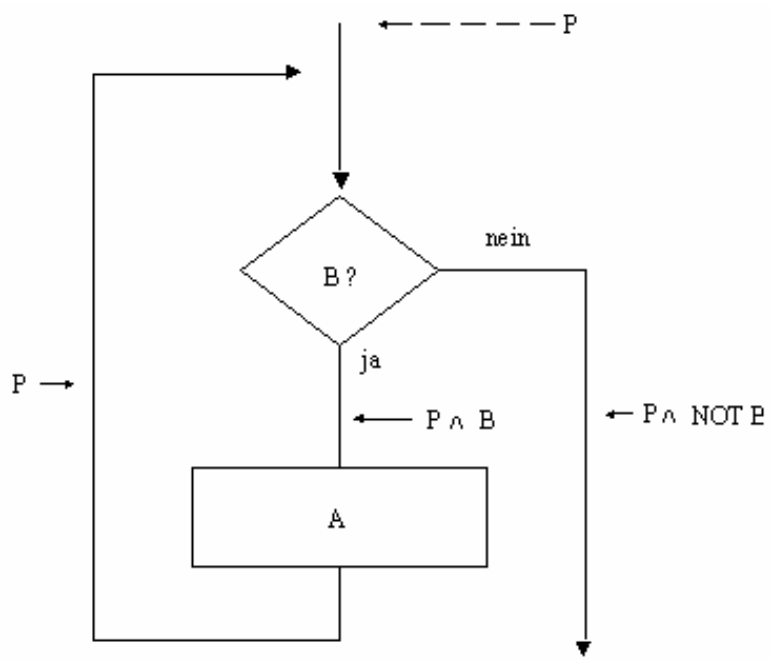
Aus	$[P \wedge B]$	A_1	[Q]
und	$[P \wedge \neg B]$	A_2	[Q]
folgt	---	---	---
	[P] **if** (B) A_1 **else** A_2 [Q]		

Ableitungsregeln für Schleifen

Schleifeninvariante

Eine Schleifeninvariante ist ein logischer Ausdruck, der sich von Durchlauf zu Durchlauf der Schleife nicht ändert. Die Schleifeninvariante ist somit auch nach dem letzten Durchlauf, beim Schleifenende, gültig und kann deshalb zum Beweis der Korrektheit der Schleife herangezogen werden.

```
while (B) A;
```



Ableitungsregel für die while-Schleife

Aus [P $\wedge$ B] A [P]

folgt [P] **while** (B) A [P $\wedge \neg$ B]

P ist die Schleifeninvariante. Sie muss sich auf Variable aus dem Schleifenrumpf A beziehen.

Die Gültigkeit von $\neg$ B beim Schleifenende ist wichtig für den Beweis der Korrektheit!

Beispiel für eine Schleifeninvariante (1)

Berechnung der Summe der ersten n Zahlen

```
public int summe(int n) {
  int i,s;
  s = 0;
  i = 0;
  while (i < n) {
    i++;
    s += i;
    // ***
  }
return s;
}
```

Beispiel für eine Schleifeninvariante (2)

An der Stelle *** gilt die Schleifeninvariante P

`summe = 0+1+2+3+...+i`

bei jedem **`i`**-ten Durchlauf der Schleife.

Mit der Endebedingung der Schleife **`¬(i < n)`** ist wegen des monotonen Wachstums von **`i`** dann **`i = n`**. Somit gilt:

`summe = 0+1+2+3+...+n`

Die Schleife ist also korrekt.

“Verification is a social process“

- Das Überzeugen durch einen Beweis in der Mathematik oder genauso auch eine Programmverifikation ist ein sozialer Prozess!
- Verifikationen sind oft lang und schwer lesbar.
- Die Spezifikation ist selbst informal; der Schritt vom Informalen zum Formalen ist niemals verifizierbar.
- Die Spezifikation ist außerdem oft sehr umfangreich (z. B. für “Einkommenssteuerberechnung”, “Betriebssystem”, “Compiler”) und in der Regel nicht vollständig.
- Programme interagieren mit der realen (informalen) Umwelt, z. B. mit I/O-Treibern, Benutzerschnittstellen, A/D-Wandlern usw., die nicht verifizierbar sind.