

Folien zur Vorlesung **Softwaretechnik**

Abschnitt 4.4: Funktionsorientierter Test (Black-Box-Test)

Bildung von Äquivalenzklassen für Testfälle

- ❑ Ein **Grenzwert** für die Eingabe
ist ein Wert, dessen minimale Änderung
zu einem **veränderten algorithmischen Verhalten** des getesteten Systems führt.
- ❑ Ein **Extremwert** für die Eingabe
ist ein Wert, dessen minimale Änderung
zu einer ungültigen Eingabe führt.
- ❑ Eine **Äquivalenzklasse**
wird durch die Menge von Eingabewerten gebildet, die zwischen
 - zwei Grenzwerten,
 - einem Grenz- und einem Extremwert,
 - zwei Extremwerten oder
 - jenseits eines Extremwertesliegen
und zu einem **gleichen algorithmischen Verhalten** des getesteten Systems führen.
- ❑ Der Begriff *Äquivalenzklasse* wird hier nicht im Sinne der gleichnamigen mathematische Definition verwendet.

Äquivalenzklassenbasierter Test

Ablauf:

- ❑ Bestimme alle Äquivalenzklassen für die möglichen Eingaben.
- ❑ Wähle **einen** Repräsentanten aus jeder Äquivalenzklasse als Testfall aus, der nicht Grenz- oder Extremwert ist.
- ❑ Teste alle so bestimmten Testfälle.
- ❑ Teste mit **allen** Grenz- und Extremwerten.

- ❑ Probleme bei der Bestimmung der Äquivalenzklassen:
 - Bestimmung setzt geordnete Mengen von Eingabewerten voraus .
 - Über Intervallen dieser Werte wird ein gleiches Verhalten erwartet.
 - Eine geeignet interpretierbare Beschreibung des algorithmischen Verhaltens des Systems wird für jede mögliche Eingabe benötigt.

Beispiel *Rabattsystem einer Boutique*

Beschreibung der erwarteten Funktionalität:

Bei einem Warenwert von mehr als 25 € wird ein Rabatt von 15% gewährt.

Ab einem Warenwert von 150 € ein Rabatt von 25% gewährt.

Äquivalenzklassen:

Äquivalenzklasse	Wertemenge	Repräsentant	Sollergebnis
1	$0 \leq \text{Warenwert} \leq 25$	22,50	22,50
2	$25 < \text{Warenwert} < 150$	30,00	25,50
3	$150 \leq \text{Warenwert}$	200,00	150,00

Beispiel *Rabattsystem einer Boutique*

(Fortsetzung)

Beschreibung der erwarteten Funktionalität:

Bei einem Warenwert von mehr als 25 € wird ein Rabatt von 15% gewährt.

Ab einem Warenwert von 150 € ein Rabatt von 25% gewährt.

Äquivalenzklassen:

Äquivalenzklasse	Wertemenge	Repräsentant	Sollergebnis
1	$0 \leq \text{Warenwert} \leq 25$	22,50	22,50
2	$25 < \text{Warenwert} < 150$	30,00	25,50
3	$150 \leq \text{Warenwert} \leq \text{maximaler Wert}$	200,00	150,00
4 (ungültig)	$\text{minimaler Wert} \leq \text{Warenwert} < 0$	-15,00	Fehler

- Die Äquivalenzklasse 3 gibt möglicherweise einen Hinweis auf einen Fehler in der Spezifikation:
Die funktionale Beschreibung enthält keine Angabe zu einer oberen Grenze des Warenwerts, bis zu der Rabatt gewährt werden soll.

Beispiel *Rabattsystem einer Boutique*

(Fortsetzung)

Funktionale Spezifikation:

Beschreibung der erwarteten Funktionalität:

Bei einem Warenwert von mehr als 25 € wird ein Rabatt von 15% gewährt.

Ab einem Warenwert von 150 € ein Rabatt von 25% gewährt.

Äquivalenzklassen:

Äquivalenzklasse	Wertemenge	Repräsentant	Sollergebnis
1	$0 \leq \text{Warenwert} \leq 25$	22,50	22,50
2	$25 < \text{Warenwert} < 150$	30,00	25,50
3	$150 \leq \text{Warenwert} \leq \text{maximaler Wert}$	200,00	150,00
4 (ungültig)	$\text{minimaler Wert} \leq \text{Warenwert} < 0$	-15,00	Fehler
5 (ungültig)	$\text{Warenwert} < \text{minimaler Wert}$		
6 (ungültig)	$\text{Warenwert} > \text{maximaler Wert}$		

– Eventuell können für die Äquivalenzklassen 5 und 6 keine Werte eingegeben werden.

Beispiel *Rabattsystem einer Boutique*

(Fortsetzung)

Beschreibung der erwarteten Funktionalität:

Bei einem Warenwert von mehr als 25 € wird ein Rabatt von 15% gewährt.

Ab einem Warenwert von 150 € ein Rabatt von 25% gewährt.

Äquivalenzklassen:

Äquivalenzklasse	Wertemenge	Repräsentant	Sollergebnis
1	$0 \leq \text{Warenwert} \leq 25$	22,50	22,50
2	$25 < \text{Warenwert} < 150$	30,00	25,50
3	$150 \leq \text{Warenwert} \leq \text{maximaler Wert}$	200,00	150,00
4 (ungültig)	$\text{minimaler Wert} \leq \text{Warenwert} < 0$	-15,00	Fehler

Testfälle für Grenzwerte

Grenzwert	Repräsentanten für Testfälle
0	-0,01; 0,00
25	25,00; 25,01
150	149,99; 150,00

Beispiel *Rabattsystem einer Boutique*

(Fortsetzung)

Beschreibung der erwarteten Funktionalität:

Bei einem Warenwert von mehr als 25 € wird ein Rabatt von 15% gewährt.

Ab einem Warenwert von 150 € ein Rabatt von 25% gewährt.

Äquivalenzklassen:

Äquivalenzklasse	Wertemenge	Repräsentant	Sollergebnis
1	$0 \leq \text{Warenwert} \leq 25$	22,50	22,50
2	$25 < \text{Warenwert} < 150$	30,00	25,50
3	$150 \leq \text{Warenwert} \leq \text{maximaler Wert}$	200,00	150,00
4 (ungültig)	$\text{minimaler Wert} \leq \text{Warenwert} < 0$	-15,00	Fehler

Testfälle für Extremwerte

Extremwert	Repräsentanten für Testfälle (Eingabe eventuell nicht möglich)
minimale Zahl	minimale Zahl; minimale Zahl-0,01
maximale Zahl	maximale Zahl; maximale Zahl+0,01

Analyse des vorangehenden Beispiels

- ❑ Die Zahl der Äquivalenzklassen hängt im Beispiel nur von **einem** Parameter ab:
Bei mehr als einem Parameter wird die Bildung der Äquivalenzklassen aufwändig.
- ❑ Hängt das Ergebnis der Ausführung einer Methode nicht nur von Parametern sondern auch vom Zustand (also den Werten der Attribute) des ausführenden Objekts ab, so muss der Zustand bei der Bildung von Äquivalenzklassen ebenfalls berücksichtigt werden:

zustandsbasierter Test

Bei einem äquivalenzklassenbasierter Test
mit mehreren Parametern oder Zuständen,
die voneinander abhängen,
müssen die Testfälle aus **Entscheidungstabellen** abgeleitet werden.

Beispiel *Happy Hour für Kunden der Boutique*

Beschreibung der erwarteten Funktionalität:

Zur Gewinnung von Kunden werden Gutscheine ausgegeben, für die ein Rabatt von 7% gewährt wird. Durch einen zeitlich befristeten Rabatt von 5% zwischen 8 und 10 Uhr soll der morgendliche Verkauf gesteigert werden.

Analyse:

Es müssen zuerst die möglichen Kombinationen von Bedingungen und die zugehörigen Aktionen ermittelt und aufgelistet werden.

Beispiel *Happy Hour für Kunden der Boutique*

(Fortsetzung)

Beschreibung der erwarteten Funktionalität:

Zur Gewinnung von Kunden werden Gutscheine ausgegeben, für die ein Rabatt von 7% gewährt wird. Durch einen zeitlich befristeten Rabatt von 5% zwischen 8 und 10 Uhr soll der morgendliche Verkauf gesteigert werden.

Entscheidungstabelle:

Bedingungen				
Gutschein	ja	ja	nein	nein
zwischen 8 und 10 Uhr	ja	nein	ja	nein

Beispiel *Happy Hour für Kunden der Boutique*

(Fortsetzung)

Beschreibung der erwarteten Funktionalität:

Zur Gewinnung von Kunden werden Gutscheine ausgegeben, für die ein Rabatt von 7% gewährt wird. Durch einen zeitlich befristeten Rabatt von 5% zwischen 8 und 10 Uhr soll der morgendliche Verkauf gesteigert werden.

Entscheidungstabelle:

Bedingungen				
Gutschein	ja	ja	nein	nein
zwischen 8 und 10 Uhr	ja	nein	ja	nein
Aktionen				
7% Rabatt				
5% Rabatt				
12% (=5%+7%) Rabatt				
regulärer Preis				

Beispiel *Happy Hour für Kunden der Boutique*

(Fortsetzung)

Beschreibung der erwarteten Funktionalität:

Zur Gewinnung von Kunden werden Gutscheine ausgegeben, für die ein Rabatt von 7% gewährt wird. Durch einen zeitlich befristeten Rabatt von 5% zwischen 8 und 10 Uhr soll der morgendliche Verkauf gesteigert werden.

Entscheidungstabelle:

Bedingungen				
Gutschein	ja	ja	nein	nein
zwischen 8 und 10 Uhr	ja	nein	ja	nein
Aktionen				
7% Rabatt		X		
5% Rabatt				
12% (=5%+7%) Rabatt				
regulärer Preis				

Beispiel *Happy Hour für Kunden der Boutique*

(Fortsetzung)

Beschreibung der erwarteten Funktionalität:

Zur Gewinnung von Kunden werden Gutscheine ausgegeben, für die ein Rabatt von 7% gewährt wird. Durch einen zeitlich befristeten Rabatt von 5% zwischen 8 und 10 Uhr soll der morgendliche Verkauf gesteigert werden.

Entscheidungstabelle:

Bedingungen				
Gutschein	ja	ja	nein	nein
zwischen 8 und 10 Uhr	ja	nein	ja	nein
Aktionen				
7% Rabatt		X		
5% Rabatt			X	
12% (=5%+7%) Rabatt	X			
regulärer Preis				X

Beispiel *Happy Hour* für Kunden der Boutique

(Fortsetzung)

Beschreibung der erwarteten Funktionalität:

Zur Gewinnung von Kunden werden Gutscheine ausgegeben, für die ein Rabatt von 7% gewährt wird. Durch einen zeitlich befristeten Rabatt von 5% zwischen 8 und 10 Uhr soll der morgendliche Verkauf gesteigert werden.

Jede Spalte beschreibt eine Äquivalenzklasse!

Entscheidungstabelle:



Bedingungen				
Gutschein	ja	ja	nein	nein
zwischen 8 und 10 Uhr	ja	nein	ja	nein
Aktionen				
7% Rabatt		X		
5% Rabatt			X	
12% (=5%+7%) Rabatt	X			
regulärer Preis				X

Beispiel *Happy Hour für Kunden der Boutique*

(Fortsetzung)

Anmerkung:

Entscheidungstabellen mit vielen Bedingungen werden schnell unübersichtlich,
häufig können aber **irrelevante** Kombinationen gestrichen werden!

Das nächste Beispiel zeigt, wie vier Äquivalenzklassen zusammenfallen.

Beispiel *Happy Hour* für Kunden der Boutique (Modifikation)

Erweiterung der funktionalen Beschreibung:

Zusätzlich gibt es Kundenkarten mit 15% Rabatt ohne die Möglichkeit von Zusatzrabatten.

Entscheidungstabelle:

Bedingungen					
Kundenkarte	ja	nein	nein	nein	nein
Gutschein	egal	ja	ja	nein	nein
zwischen 8 und 10 Uhr	egal	ja	nein	ja	nein
Aktionen					
7% Rabatt			X		
5% Rabatt				X	
12% (=5%+7%) Rabatt		X			
regulärer Preis					X
15% Rabatt	X				

weiteres Beispiel – Berechnung der Binominalkoeffizienten

Die Methode `long bin(int n, int k)` soll den Binominalkoeffizienten $\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$ berechnen!

Dabei gilt:

$0! = 1$, $1! = 1$, $n! = (n-1)! \cdot n$, für negative Werte n ist $n!$ nicht definiert.

Ist die Berechnung nicht möglich, soll von `bin` die Meldung "error" ausgegeben und der Wert 0 zurückgegeben werden.

- ❑ Wie würde ein äquivalenzklassenbasierter Test von `bin` aussehen?
- ❑ Bestimmen Sie Äquivalenzklassen und daraus Testfälle mit Kombinationen von Werten für `n` und `k` an.

weiteres Beispiel – Berechnung der Binominalkoeffizienten

(Fortsetzung)

Die Methode `long bin(int n, int k)` soll den Binominalkoeffizienten $\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$ berechnen!

Dabei gilt:

$0! = 1$, $1! = 1$, $n! = (n-1)! \cdot n$, für negative Werte n ist $n!$ nicht definiert.

Ist die Berechnung nicht möglich, soll von `bin` die Meldung "error" ausgegeben und der Wert 0 zurückgegeben werden.

- ❑ Wie würde ein äquivalenzklassenbasierter Test von `bin` aussehen?
- ❑ Bestimmen Sie Äquivalenzklassen und daraus Testfälle mit Kombinationen von Werten für n und k an.

Äquivalenzklassen:

- ❑ für gültige Werte muss gelten: $n \geq 0$, $k \geq 0$, $n \geq k$
- ❑ $n == 0$, $k == 0$ und $n \geq k$ begrenzen die gültigen Wertekombinationen
- ❑ Vorschläge für Testfälle:
(0,0), (7,0), (7,3), (7,7) für Ergebnisse ohne "error"-Meldung
(3,7), (-1,7), (7,-1), (-1,-1) für Ergebnisse mit "error"-Meldung
(MIN_Value, MIN_Value), (MIN_Value, MAX_Value), (MAX_Value, MIN_Value), (MAX_Value, MAX_Value)

Einordnung von Tests

Geht die Ermittlung von Testfällen

– wie in den vorangehenden Beispielen –

von der Beschreibung der erwarteten Funktionalität aus:

funktionsorientierter Test

(auch **black-box-Test**, da die konkrete Implementierung nicht betrachtet wird)

Eigenschaften des funktionsorientierten Tests:

- Die Übereinstimmung von Spezifikation und Implementierung wird überprüft.

Folien zur Vorlesung **Softwaretechnik**

Abschnitt 4.5: Strukturorientierter Test (White-Box-Test)

Einordnung von Tests

Geht die Ermittlung von Testfällen

– wie in den vorangehenden Beispielen –

von der Beschreibung der erwarteten Funktionalität aus:

funktionsorientierter Test

(auch **black-box-Test**, da die konkrete Implementierung nicht betrachtet wird)

Eigenschaften des funktionsorientierten Tests:

- ❑ Die Übereinstimmung von Spezifikation und Implementierung wird überprüft.

aber:

- ❑ Bei unzureichender Beschreibung ist ein funktionsorientierter Test nicht möglich.
Beispiel: falsche Uhrzeit für morgendlichen Rabatt am Tag der Sommerzeitumstellung
- ❑ Selten auftretende Fehler werden nur zufällig aufgedeckt.
Beispiel: 20% Rabatt für eine bestimmte Kundenkarte
- ❑ Zusätzlich in die Implementierung eingefügte Funktionen werden nur zufällig aufgedeckt.
Beispiel: 20% Rabatt für eine bestimmte Kundenkarte

Ein funktionsorientierter Test reicht also nicht aus, um Vertrauen in die Software zu schaffen!

Beispiel – Berechnung der Binominalkoeffizienten

(Fortsetzung)

Die Methode `long bin(int n, int k)` soll den Binominalkoeffizienten $\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$ berechnen!

```
public static long bin (int n, int k) {
    if (n < 0 || k < 0 || n > 50) { System.out.print(" error "); return 0; }
    // Optimierungen
    if (k == 0 || k == n) return 1;
    if (k == 1 || k == n - 1) return k;
    if (2 * k > n) k = n - k;
    // Berechnung
    if (n < k) { System.out.print(" error "); return 0; }
    long c=1, d=1;
    for (int i=0; i < k; ++i) {
        c *= n - i;
        d *= i + 1;
    }
    return c / d;
}
```

Werden alle Fehler dieser Implementierung
durch die geplanten Testfälle erkannt?
(siehe Folie 406)

Beispiel – Berechnung der Binominalkoeffizienten

(Fortsetzung)

Die Methode `long bin(int n, int k)` soll den Binominalkoeffizienten $\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$ berechnen!

```
public static long bin (int n, int k) {
    if (n < 0 || k < 0 || n > 50) { System.out.print(" error "); return 0; }
    // Optimierungen
    if (k == 0 || k == n) return 1;
    if (k == 1 || k == n - 1) return k;
    if (2 * k > n) k = n - k;
    // Berechnung
    if (n < k) { System.out.print(" error "); return 0; }
    long c=1, d=1;
    for (int i=0; i < k; ++i) {
        c *= n - i;
        d *= i + 1;
    }
    return c / d;
}
```

Werden alle Fehler dieser Implementierung
durch die geplanten Testfälle erkannt?
(siehe Folie 406)

nein

Beispiel – Berechnung der Binominalkoeffizienten

(Fortsetzung)

Die Methode **long** bin(**int** n, **int** k) soll den Binominalkoeffizienten $\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$ berechnen!

```
public static long bin (int n, int k) {
    if (n < 0 || k < 0 || n > 50) { System.out.print(" error "); return 0; }
    // Optimierungen
    if (k == 0 || k == n) return 1;
    if (k == 1 || k == n - 1) return k;
    if (2 * k > n) k = n - k;
    // Berechnung
    if (n < k) { System.out.print(" error "); return 0; }
    long c=1, d=1;
    for (int i=0; i < k; ++i) {
        c *= n - i;
        d *= i + 1;
    }
    return c / d;
}
```

Wertebereich von long wird bei der Berechnung überschritten

in diesem Sonderfall muss der Wert von n zurückgegeben werden

Abfrage kommt zu spät:
so können in Optimierung negative
Werte für k entstehen

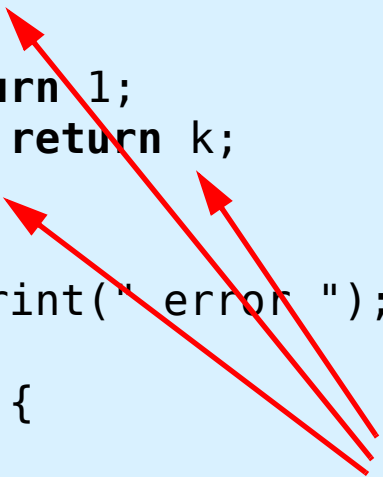
3 Fehler

Beispiel – Berechnung der Binominalkoeffizienten

(Fortsetzung)

Die Methode `long bin(int n, int k)` soll den Binominalkoeffizienten $\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$ berechnen!

```
public static long bin (int n, int k) {
    if (n < 0 || k < 0 || n > 50) { System.out.print(" error "); return 0; }
    // Optimierungen
    if (k == 0 || k == n) return 1;
    if (k == 1 || k == n - 1) return k;
    if (2 * k > n) k = n - k;
    // Berechnung
    if (n < k) { System.out.print(" error "); return 0; }
    long c=1, d=1;
    for (int i=0; i < k; ++i) {
        c *= n - i;
        d *= i + 1;
    }
    return c / d;
}
```



Probleme entstehen durch
spezifische Implementierung

3 Fehler

Einordnung von Tests

(Fortsetzung)

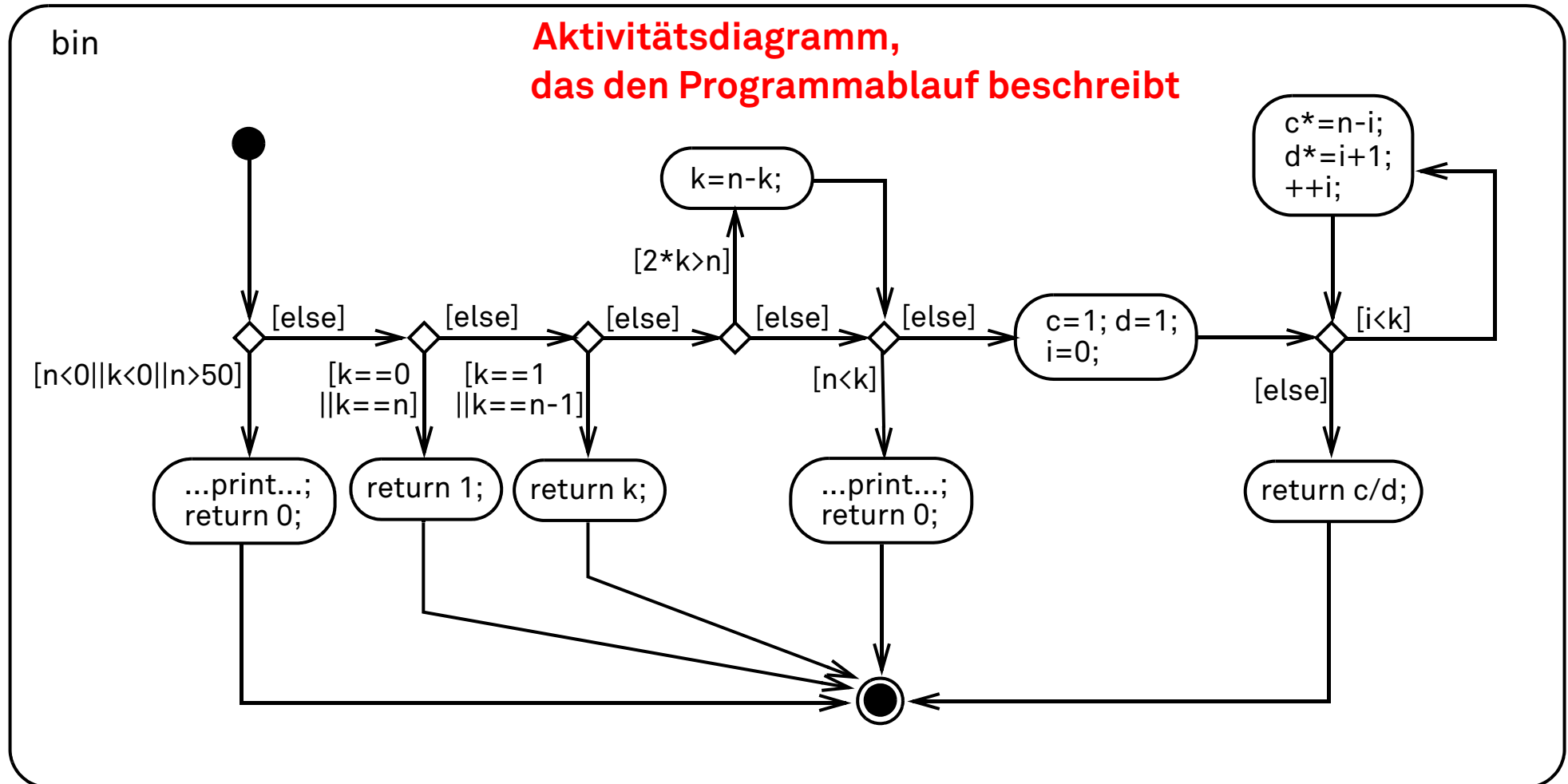
Geht die Ermittlung von Testfällen
vom **Quelltext** der Implementierung aus:
strukturorientierter Test (auch **white-box-Test**)

Die Testfälle werden so angelegt, dass bestimmte Code-Sequenzen ausgeführt werden.

Eigenschaften des strukturorientierten Tests

- ❑ Stellt Überprüfung des (gesamten) implementierten Codes sicher.
- aber:**
- ❑ Die Beschreibung wird nicht zur Auswahl von Testfällen genutzt, sondern nur zur Bestimmung der Soll-Ergebnisse,
 - es wird daher nicht versucht, die funktionale Richtigkeit gemäß einer vorher angegebenen Zielvorstellung nachzuweisen.
 - ❑ Strukturorientierte Tests sind häufig aufwändig,
 - da zunächst der Quelltext analysiert werden muss,
 - es eventuell sehr viele alternative Programmabläufe gibt,
 - zur Bestimmung der Soll-Ergebnisse der Quelltext den entsprechenden Teilen der Spezifikation zugeordnet werden muss.

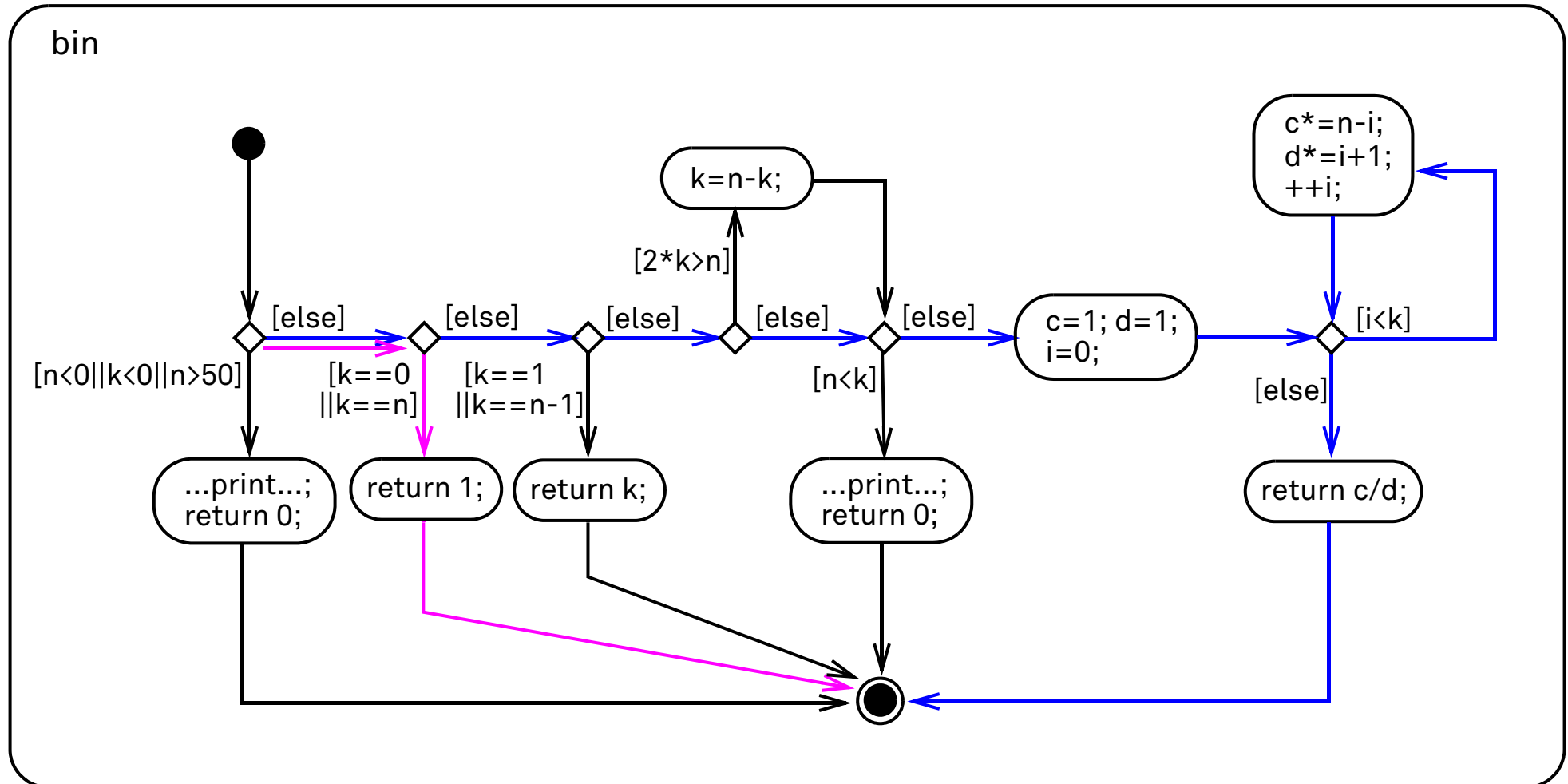
Darstellung der Struktur von bin(int n, int k)



Darstellung der Struktur von bin(int n, int k)

(Fortsetzung)

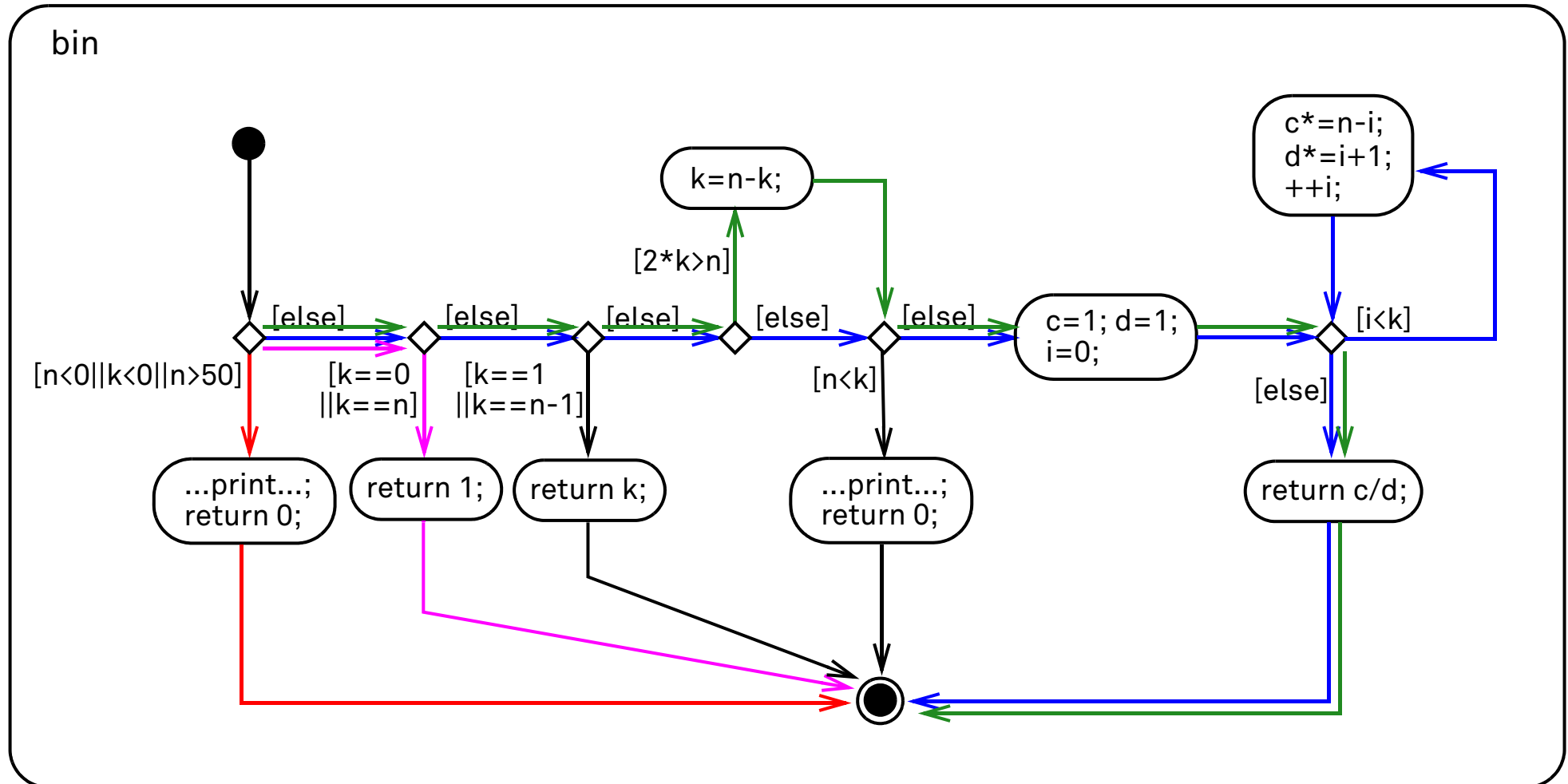
Testfälle aus black-box-Test (Folie 406): (0,0), (7,0), (7,3), (7,7)



Darstellung der Struktur von bin(int n, int k)

(Fortsetzung)

Testfälle aus black-box-Test (Folie 406): (3,7), (-1,7), (7,-1), (-1,-1)



Darstellung der Struktur von `bin(int n, int k)`

(Fortsetzung)

Analyse der von den Testfällen benutzten Pfade durch die Methode `bin`:

- ❑ Der (verborgene und fehlerhafte) Teil der Bedingung `... || n < 50` wird nicht getestet.
Der Fehler wird nicht gefunden.
- ❑ Die (verborgene und fehlerhafte) Optimierung `k == 1 || k == n - 1` wird nicht getestet.
Der Fehler wird nicht gefunden.
- ❑ Ein ungültiger Testfall (3,7) führt nicht zum erwünschten Ergebnis.
Dieser Fehler wird erkannt.
- ❑ Es gibt keinen ungültigen Testfall mit `n < k`.
Diese Situation wird nicht getestet.
- ❑ Es gibt keinen gültigen Testfall mit `2 * k > n`.
Diese Situation wird nicht getestet.

- ❑ Die Methode `bin` wird nicht ausreichend getestet.
- ❑ Aus der Betrachtung der Struktur der Methode lässt sich auf die Qualität des Tests schließen.

Strukturorientiertes Testen

Geht die Ermittlung von Testfällen
vom Quelltext der Implementierung aus:
strukturorientierter Test (auch **white-box-Test**)

Die Testfälle werden so angelegt, dass bestimmte Code-Sequenzen ausgeführt werden.

Eigenschaften des strukturorientierten Tests

- ❑ Stellt Überprüfung des (gesamten) implementierten Codes sicher.
- aber:**
- ❑ Die Spezifikation wird nicht zur Auswahl von Testfällen genutzt, sondern nur zur Bestimmung der Soll-Ergebnisse,
 - es wird daher nicht versucht, die funktionale Richtigkeit gemäß einer vorher angegebenen Zielvorstellung nachzuweisen.
 - ❑ Strukturorientierte Tests sind häufig aufwändig,
 - da zunächst der Quelltext analysiert werden muss,
 - es eventuell sehr viele alternative Programmabläufe gibt,
 - zur Bestimmung der Soll-Ergebnisse der Quelltext den entsprechenden Teilen der Spezifikation zugeordnet werden muss.

Strukturorientiertes Testen

(Fortsetzung)

Strukturorientiertes Testen wird nun mit Hilfe von UML-Aktivitätsdiagrammen präzisiert

Literatur: Hoffmann, Dirk W.: Software-Qualität, S. 157-216
http://link.springer.com/chapter/10.1007/978-3-540-76323-9_4

Liggesmeyer, Peter: Software-Qualität – Testen, Analysieren und Verifizieren von Software, S. 49-117, S136-138
http://link.springer.com/chapter/10.1007/978-3-8274-2203-3_2

Beispiel für eine Strukturanalyse (siehe Folie 349)

```
int calculate (int end, int init, int lim, int bon) {  
    int sum = 0;  
    if (end > 0) {  
        sum = init;  
        for (int i=0; i < end; i++) {  
            sum += bon;  
            if (sum > lim) {  
                sum += bon;  
            }  
        }  
        if (sum > 2*lim) {  
            sum = 2*lim;  
        }  
    }  
    return sum;  
}
```

Beispiel für eine Strukturanalyse (siehe Folie 349)

```
int calculate (int end, int init, int lim, int bon) {
```

```
    int sum = 0;
```

```
    if (end > 0) {
```

```
        sum = init;
```

```
        for (int i=0; i < end; i++) {
```

```
            sum += bon;
```

```
            if (sum > lim) {
```

```
                sum = init;
```

```
                i = 0;
```

```
            }
        }
    }
    if (sum > 2*lim) {
```

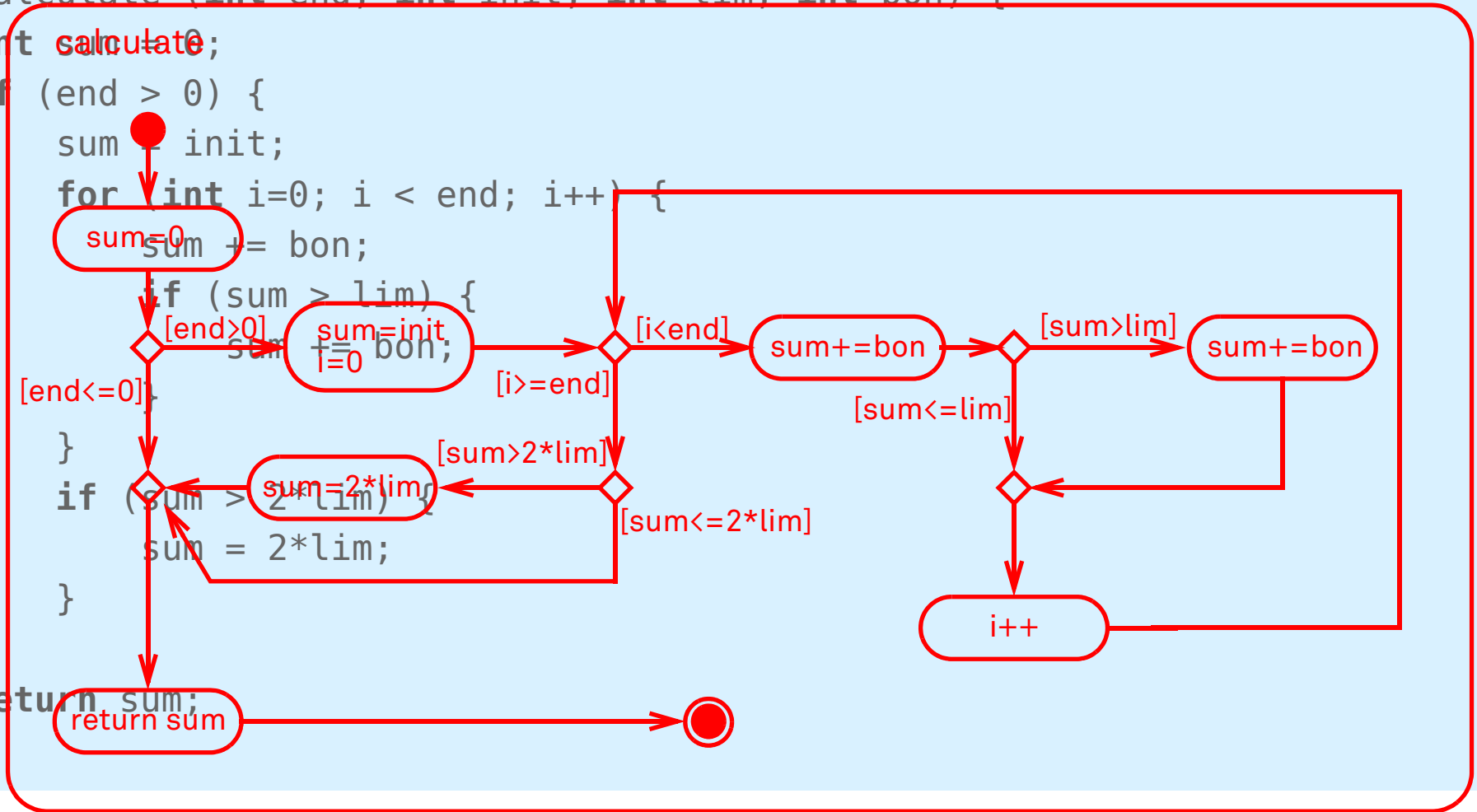
```
        sum = 2*lim;
```

```
        sum = 2*lim;
```

```
    }
}
```

```
return sum;
```

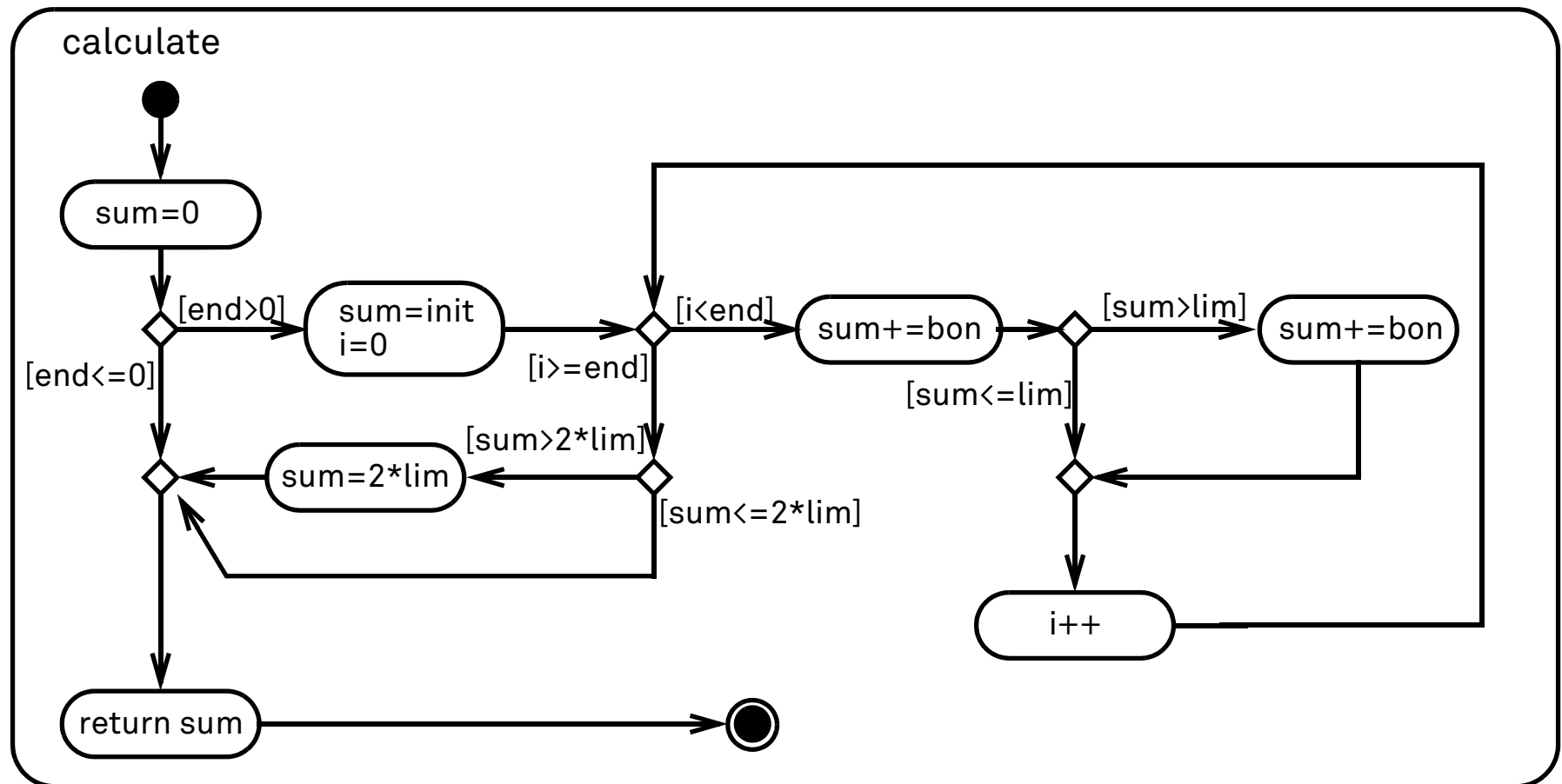
```
}
```



Ermittlung von Testfällen: Anweisungsüberdeckung (C₀-Test)

Vorgabe für **C₀-Test**: Jede Anweisung im Quelltext **muss mindestens einmal** ausgeführt werden.

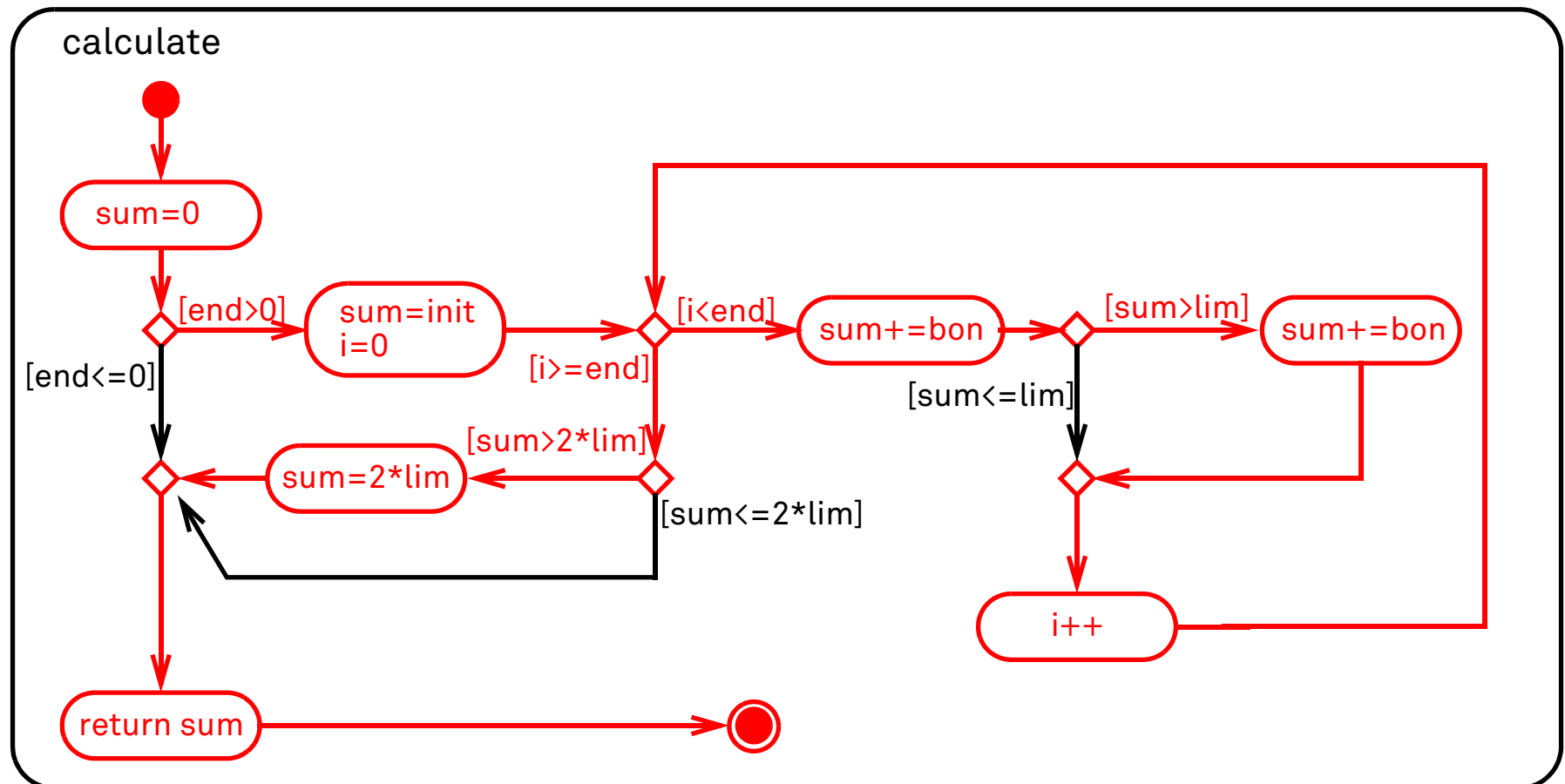
= Jede Aktion des Diagramms muss von mindestens einem Testfall erreicht werden.



Ermittlung von Testfällen: Anweisungsüberdeckung (C₀-Test)

(Fortsetzung)

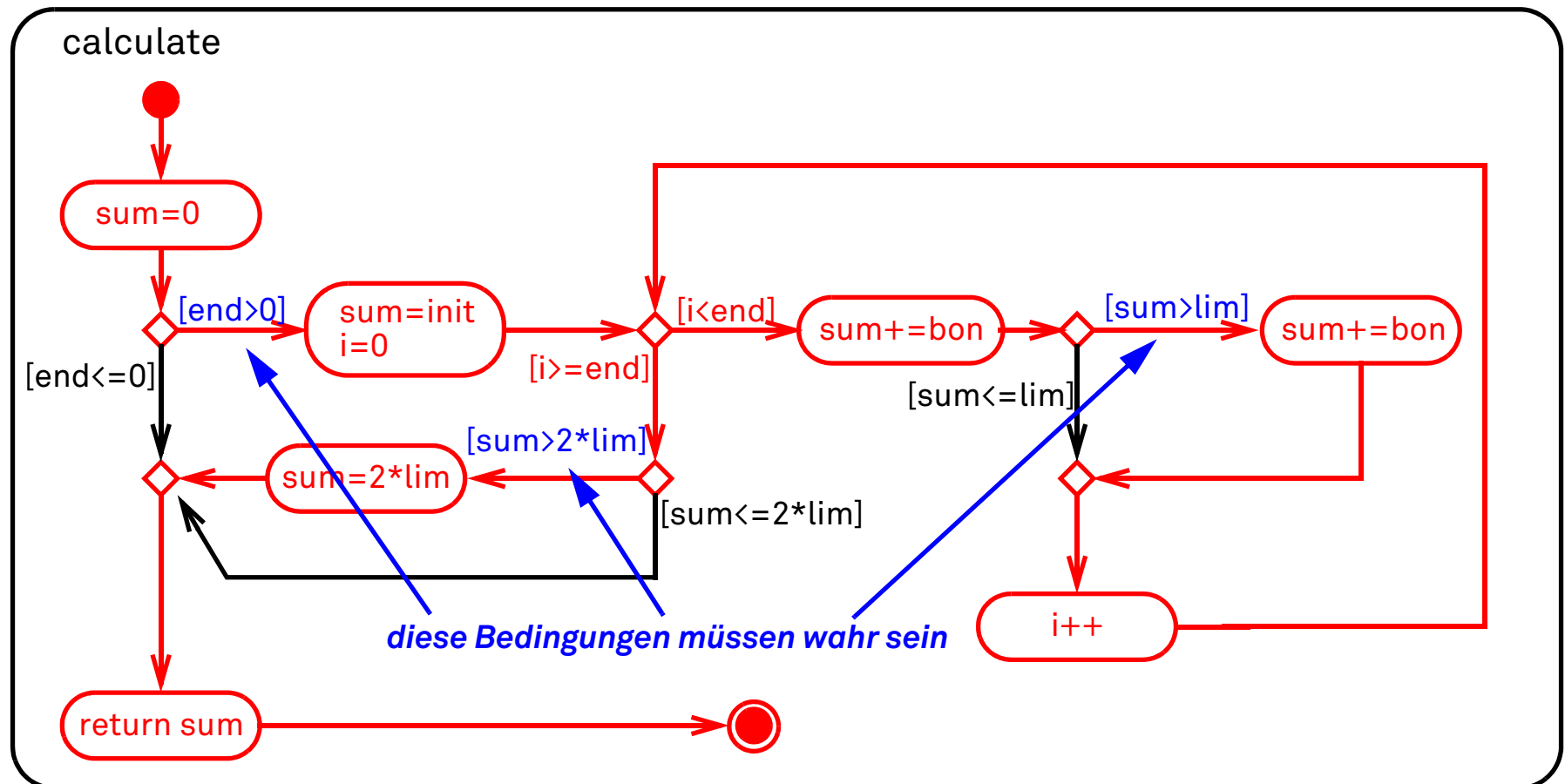
Für das Beispiel reicht ein Testfall aus, um alle Aktionen zu erreichen:
Die Parameter können geeignet gesetzt werden,
so dass alle Aktionen auf einem Pfad liegen.



Ermittlung von Testfällen: Anweisungsüberdeckung (C₀-Test)

(Fortsetzung)

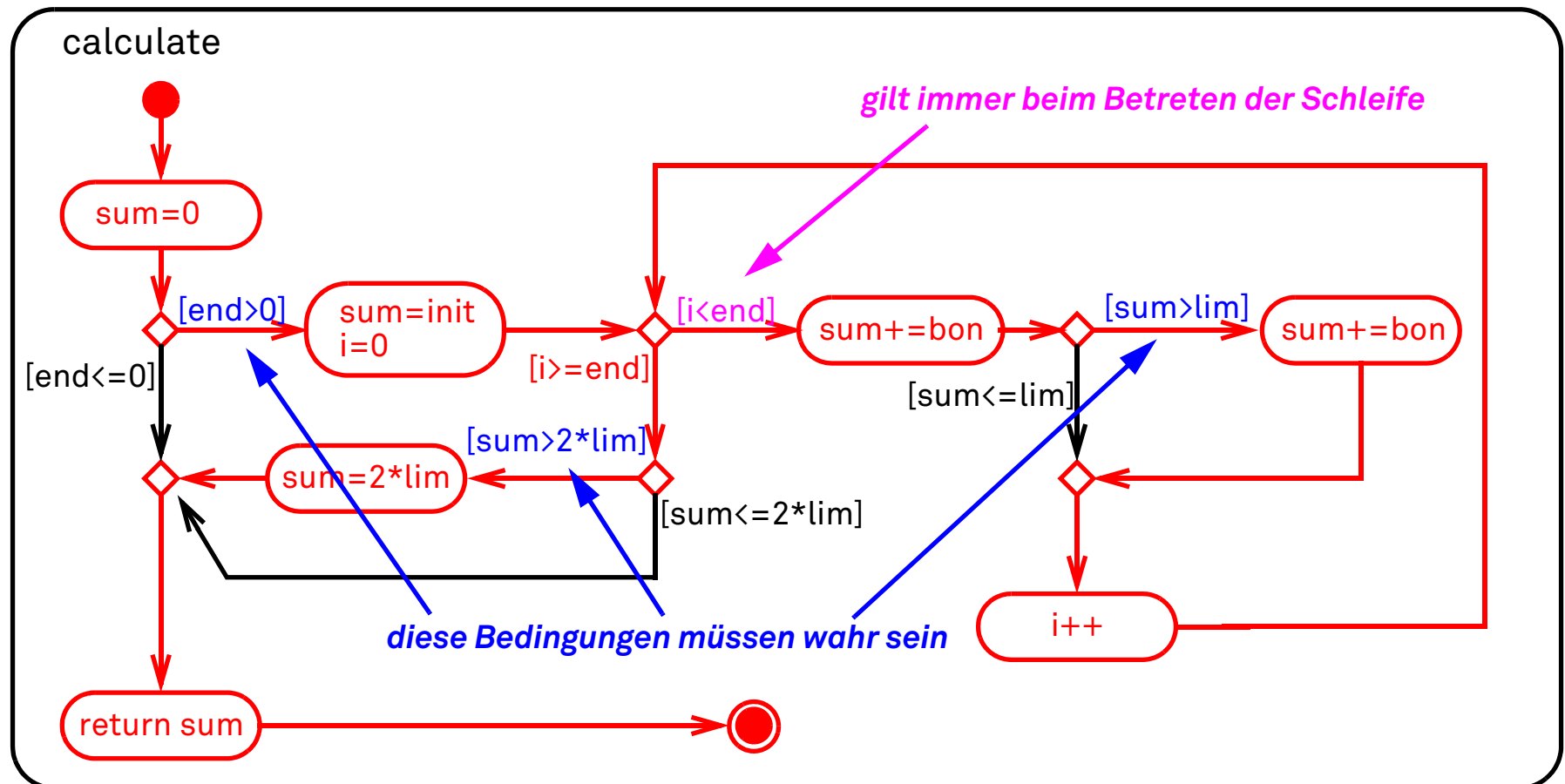
Für das Beispiel reicht ein Testfall aus, um alle Aktionen zu erreichen:
Die Parameter können geeignet gesetzt werden,
so dass alle Aktionen auf einem Pfad liegen.



Ermittlung von Testfällen: Anweisungsüberdeckung (C₀-Test)

(Fortsetzung)

Für das Beispiel reicht ein Testfall aus, um alle Aktionen zu erreichen:
Die Parameter können geeignet gesetzt werden,
so dass alle Aktionen auf einem Pfad liegen.



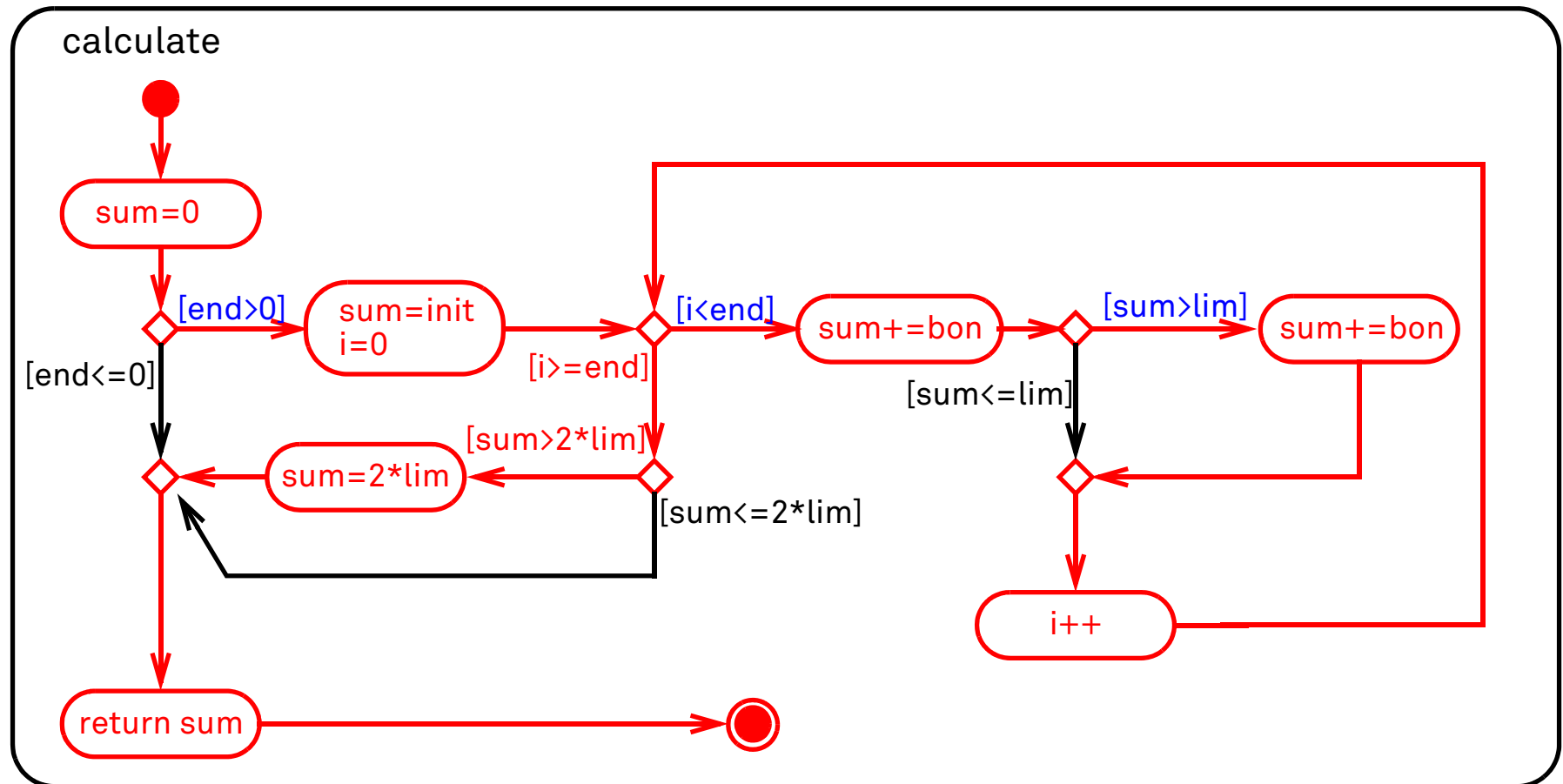
Ermittlung von Testfällen: Anweisungsüberdeckung (C₀-Test)

(Fortsetzung)

wird für `calculate(int end, int init, int lim, int bon)`

z.B. erreicht mit: `end=1, init=0, lim=1, bon=2`

Aufruf also z.B.: `calculate(1, 0, 1, 2)`



Ermittlung von Testfällen: Anweisungsüberdeckung (C_0 -Test)

(Fortsetzung)

Analyse der Leistungsfähigkeit der Anweisungsüberdeckung:

- ❑ Die Anweisungsüberdeckung erkennt nicht erreichbare Anweisungen.
- ❑ Das Einhalten bestimmter Reihenfolgen bei der Ausführung von Aktionen in Abhängigkeit von Bedingungen wird aber nicht berücksichtigt.
- ❑ Es werden nur wenige Ausführungsfolgen getestet und es wird so nur ein Teil der algorithmisch möglichen Abläufe geprüft.

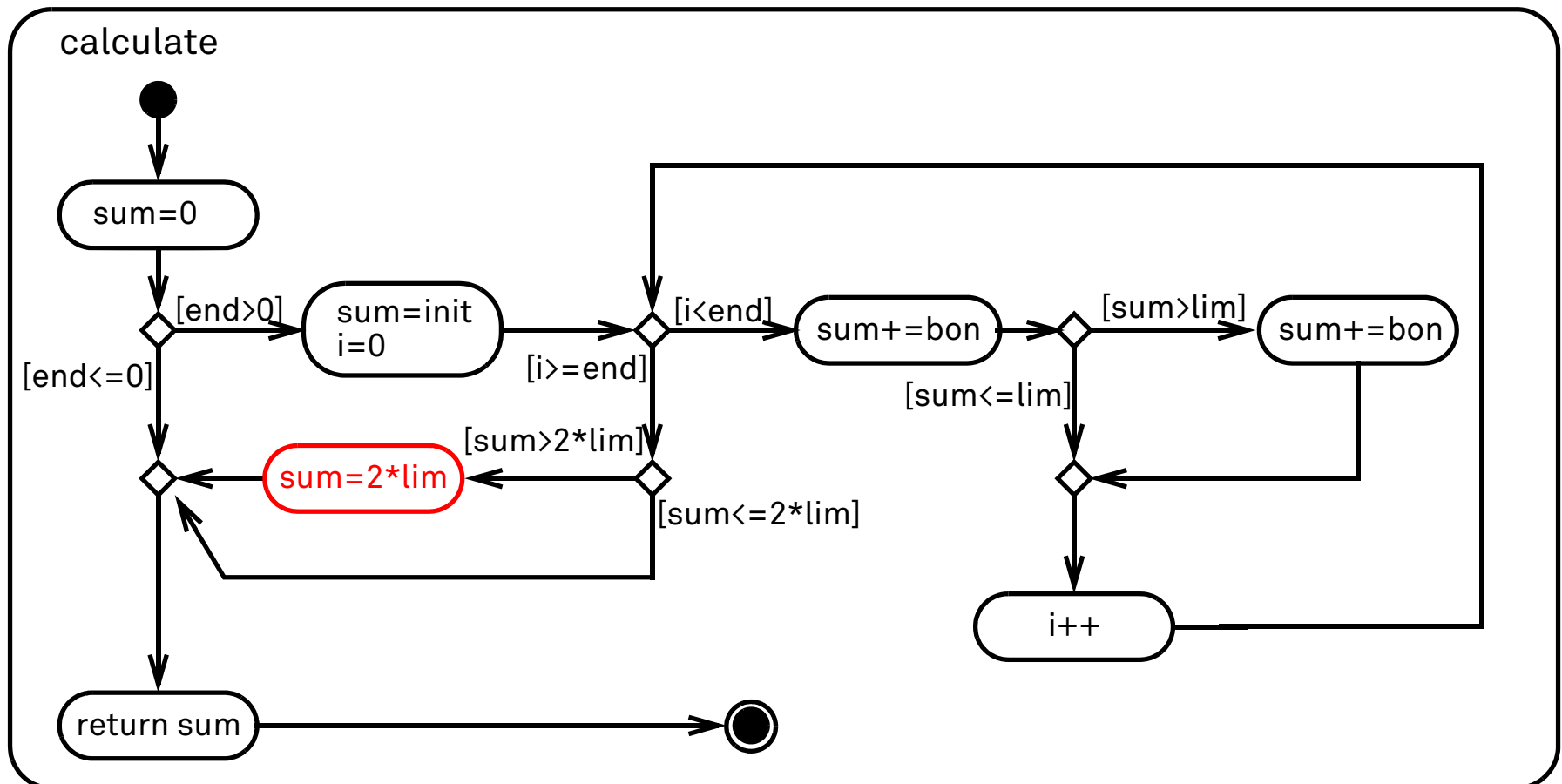
Fehler werden daher nur zufällig erkannt.

Güte eines Überdeckungstests T

Überdeckungsgrad $G = (\text{unter } T \text{ überdeckte Anweisungen}) / (\text{alle Anweisungen})$

für das Beispiel: `calculate(1, 0, 1, 2)` führt zu $G = 1.0$

`calculate(1, 3, 3, 1)` führt zu $G = 0.875$



Ermittlung von Testfällen: Zweigüberdeckung (C_1 -Test)

Da die Anweisungsüberdeckung nur zu unzureichenden Aussagen führt:

Zweigüberdeckung (C_1 -Test)

Vorgabe für **C_1 -Test**:

Jeder alternative Zweig im Programmtext wird mindestens einmal durchlaufen.

= Jede Kante des Diagramms wird mindestens einmal durchlaufen.

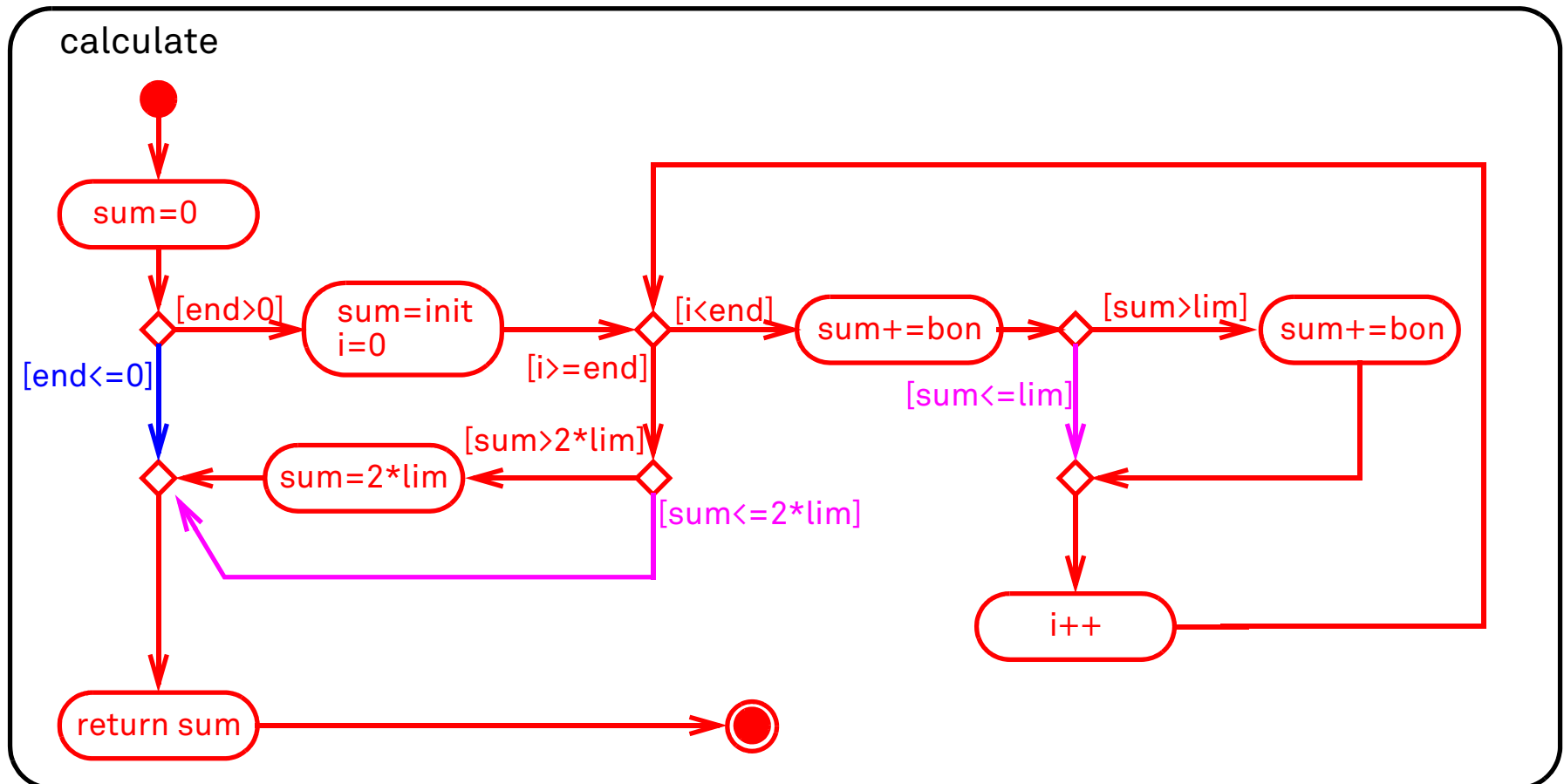
Anmerkungen:

- ❑ Die Zweigüberdeckung umfasst immer die Anweisungsüberdeckung.
- ❑ Es gibt möglicherweise schwer erreichbare Zweige, wenn die notwendigen Bedingungen nur schwer gezielt hergestellt werden können.
- ❑ Bei einfachen bedingten Anweisungen gibt es *leere Zweige*, die bei der Zweigüberdeckung durchlaufen werden müssen.
- ❑ Bei Schleifen genügt ein einziger Durchlauf. ==> **unzureichende Prüfung**
- ❑ Für das Beispiel gilt:
Zusätzlich zum Testfall aus der Anweisungsüberdeckung müssen Testfälle für die *drei leeren Zweige* ergänzt werden.

Ermittlung von Testfällen: Zweigüberdeckung (C₁-Test)

(Fortsetzung)

wird z.B. erreicht mit: `end=1, init=0, lim=1, bon=2` (aus C₀-Überdeckung bekannt),
 und `end=0` (*leerer Zweig*)
 und `end=1, init=0, lim=1, bon=0` (*2 leere Zweige*)



Ermittlung von Testfällen: Zweigüberdeckung (C₁-Test)

(Fortsetzung)

Analyse der Leistungsfähigkeit der Zweigüberdeckung:

- ❑ Die Zweigüberdeckung erkennt alle nicht erreichbaren Zweige und alle nicht erreichbaren Anweisungen.
- ❑ Die Zweigüberdeckung berücksichtigt keine Abhängigkeiten zwischen den Zweigen.
- ❑ Komplexe Bedingungen werden nur rudimentär getestet:
Bei Disjunktionen reicht ein *wahrer*,
bei Konjunktionen ein *falscher* Teilausdruck,
um den Wert der gesamten Bedingung zu setzen.

Beispiele, die mit nur zwei Belegungen für *a, b, c, d* beide Zweige abdecken:

```
if ( a | b | c | d ) A1 else A2;  
if ( a & b & c & d ) A1 else A2;
```

- ❑ Eine **vollständige Zweigüberdeckung** (Überdeckungsgrad $G = 1.0$) ist für Software in bestimmten Anwendungsbereichen der Mindeststandard:
z.B. für kritische Software im Bereich der Luftfahrt (Standard RTCA DO-178B).

Ermittlung von Testfällen: Mehrfachbedingungsüberdeckung

Da auch die Zweigüberdeckung nur zu unzureichenden Aussagen führt:

Mehrfachbedingungsüberdeckung

Vorgabe für die Mehrfachbedingungsüberdeckung:

Jede mögliche Kombination von Wahrheitswerten der atomaren Prädikate wird ausgeführt.

Anmerkungen:

- ❑ Die Mehrfachbedingungsüberdeckung umfasst die Anweisungsüberdeckung.
- ❑ Die Mehrfachbedingungsüberdeckung umfasst die Zweigüberdeckung.
- ❑ Es kann nicht-realisierte Kombinationen von Wahrheitswerten geben.
Diese müssen keine Fehler sein, z.B. bei Short-Cut-Operatoren wie && oder ||.
- ❑ Beispiel:
Der Ausdruck $(nr > 1 \ \& \ fehler < 0) \mid abbruch$ hat drei atomare Prädikate,
also müssen 8 Testfälle erstellt werden.
- ❑ **Problem:** bei n Teilausdrücken entstehen 2^n Testfälle
==> Die Mehrfachbedingungsüberdeckung ist aufgrund der kombinatorischen
Komplexität in der Praxis häufig unbrauchbar.

Ermittlung von Testfällen: modifizierte Bedingungs-/Entscheidungsüberdeckung

Pragmatische Variante der Mehrfachbedingungsüberdeckung:
modifizierte Bedingungs-/Entscheidungsüberdeckung

Vorgabe für die modifizierte Bedingungs-/Entscheidungsüberdeckung:

Für jedes atomare Prädikat muss im Test nachgewiesen werden,
dass es das Ergebnis der Auswertung der Bedingung unabhängig
von den anderen atomaren Prädikaten beeinflussen kann.

Präzisierung:

Für jedes atomare Prädikat müssen zwei Testfälle vorliegen,
die sich **nur genau** im Wert dieses Prädikats unterscheiden
und die zu unterschiedlichen Ergebnissen bei Auswertung der Bedingung führen.

Anmerkungen:

- ❑ Die modifizierte Bedingungs-/Entscheidungsüberdeckung umfasst die Anweisungs- und die Zweigüberdeckung.
- ❑ Auch die modifizierte Bedingungs-/Entscheidungsüberdeckung ist sehr aufwändig.

Ermittlung von Testfällen: modifizierte Bedingungs-/Entscheidungsüberdeckung (Fortsetzung)

Für jedes atomare Prädikat müssen zwei Testfälle vorliegen,
die sich nur genau im Wert dieses Prädikats unterscheiden
und die zu unterschiedlichen Ergebnissen bei Auswertung der Bedingung führen.

- Ein erstes, einfaches Beispiel.

Der Ausdruck $a \mid b \mid c$ benötigt vier Testfälle:

- a bestimmt nur dann das Ergebnis, wenn $b \mid c == \text{false}$ gilt,
also $b == \text{false}$ und $c == \text{false}$,
die Testfälle sind also $\text{true}, \text{false}, \text{false}$ und $\text{false}, \text{false}, \text{false}$
- b bestimmt nur dann das Ergebnis, wenn $a \mid c == \text{false}$ gilt,
die Testfälle sind also $\text{false}, \text{true}, \text{false}$ und $\text{false}, \text{false}, \text{false}$
- c bestimmt nur dann das Ergebnis, wenn $a \mid b == \text{false}$ gilt,
die Testfälle sind also $\text{false}, \text{false}, \text{true}$ und $\text{false}, \text{false}, \text{false}$

- Anmerkung:

Das Beispiel zeigt, dass die Testfälle für verschiedene Prädikate identisch sein können.
Bei n Prädikaten werden also höchstens – aber nicht immer – $2n$ Testfälle benötigt.

Ermittlung von Testfällen: modifizierte Bedingungs-/Entscheidungsüberdeckung (Fortsetzung)

Für jedes atomare Prädikat müssen zwei Testfälle vorliegen, die sich nur genau im Wert dieses Prädikats unterscheiden und die zu unterschiedlichen Ergebnissen bei Auswertung der Bedingung führen.

- ❑ weiteres Beispiel: $(nr > 1 \ \& \ fehler > 0) \mid abbruch$
- ❑ Das Prädikat $nr > 1$ hat nur dann eine Wirkung, wenn
 - $fehler > 0$ den Wert `true` hat (denn sonst liefert die Konjunktion `false`) und
 - $abbruch$ den Wert `false` hat (denn sonst liefert die Disjunktion immer `true`).
- ❑ $nr > 1$ muss selbst die beiden Werte `true` und `false` annehmen.

$nr > 1$	$fehler > 0$	$abbruch$	Ergebnis
true	true	false	true
false	true	false	false



Ermittlung von Testfällen: modifizierte Bedingungs-/Entscheidungsüberdeckung (Fortsetzung)

Für jedes atomare Prädikat müssen zwei Testfälle vorliegen, die sich nur genau im Wert dieses Prädikats unterscheiden und die zu unterschiedlichen Ergebnissen bei Auswertung der Bedingung führen.

- ❑ weiteres Beispiel: $(nr > 1 \ \& \ fehler > 0) \mid abbruch$
- ❑ Das Prädikat $fehler > 0$ hat nur dann eine Wirkung, wenn
 - $nr > 1$ den Wert `true` hat (denn sonst liefert die Konjunktion `false`) und
 - $abbruch$ den Wert `false` hat (denn sonst liefert die Disjunktion immer `true`).
- ❑ $fehler > 0$ muss selbst die beiden Werte `true` und `false` annehmen.

$nr > 1$	$fehler > 0$	$abbruch$	Ergebnis
true	true	false	true
false	true	false	false
true	false	false	false

bekannter Testfall

bekannter Testfall

neuer Testfall



Ermittlung von Testfällen: modifizierte Bedingungs-/Entscheidungsüberdeckung (Fortsetzung)

Für jedes atomare Prädikat müssen zwei Testfälle vorliegen, die sich nur genau im Wert dieses Prädikats unterscheiden und die zu unterschiedlichen Ergebnissen bei Auswertung der Bedingung führen.

- ❑ weiteres Beispiel: weiteres Beispiel: $(nr > 1 \ \& \ fehler > 0) \mid abbruch$
- ❑ Das Prädikat `abbruch` hat nur dann eine Wirkung, wenn
 - die Konjunktion den Wert `false` liefert,
also `nr > 1` **oder** `fehler > 0` den Wert `false` besitzt
- ❑ `abbruch` muss selbst die beiden Werte `true` und `false` annehmen.

nr>1	fehler>0	abbruch	Ergebnis
true	true	false	true
false	true	false	false
true	false	false	false
false	true	true	true

bekannter Testfall

bekannter Testfall

bekannter Testfall

neuer Testfall

- ❑ Für den Test des Beispiels werden also 4 Testfälle benötigt.

Ermittlung von Testfällen: modifizierte Bedingungs-/Entscheidungsüberdeckung (Fortsetzung)

Analyse der modifizierten Bedingungs-/Entscheidungsüberdeckung:

- ❑ Für n atomare Prädikate werden mindestens $n+1$ Testfälle benötigt.
- ❑ Für n atomare Prädikate werden höchstens $2n$ Testfälle benötigt.
- ❑ Es gibt in der Regel verschiedene Kombinationen von Testfällen, die eine modifizierte Bedingungs-/Entscheidungsüberdeckung mit $G=1.0$ herstellen.
- ❑ Sind atomare Prädikate voneinander abhängig, so kann eine modifizierte Bedingungs-/Entscheidungsüberdeckung mit $G=1.0$ nicht sichergestellt werden.
- ❑ Eine modifizierten Bedingungs-/Entscheidungsüberdeckung mit $G=1.0$ ist ebenfalls ein für Software in bestimmten Anwendungsbereichen (z.B. Luftfahrt) vorgeschriebener Standard.

Ermittlung von Testfällen: Pfadüberdeckung

Vorgabe für die **Pfadüberdeckung**:

Jeder mögliche Pfad wird mindestens einmal durchlaufen.

- ❑ In dieser allgemeinen Form nur von theoretischer Bedeutung:
Schleifen können zu sehr vielen Pfaden führen, die nicht alle getestet werden können.
- ❑ pragmatische Alternative:
 k -begrenzte strukturierte Pfadüberdeckung
Jede Anweisung wird **höchstens** k -mal durchlaufen.
Insbesondere werden also die Rümpfe von Schleifen nur bis maximal k -mal durchlaufen.

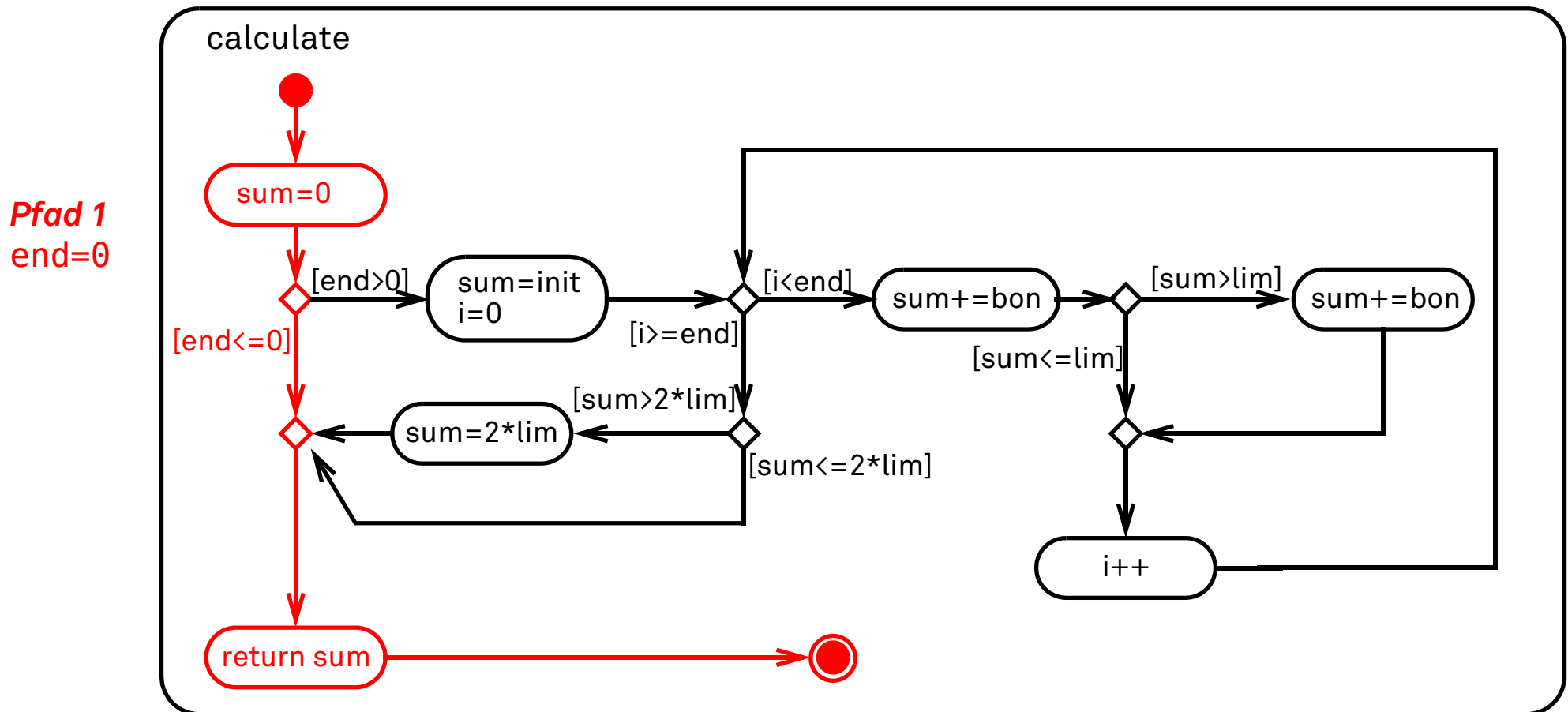
Anmerkungen:

- ❑ Für k wird typischerweise ein kleiner Wert, z.B. $k=2$, gewählt.
- ❑ Eine **2-begrenzte** strukturierte Pfadüberdeckung schließt also immer die **1-begrenzte** strukturierte Pfadüberdeckung ein.

Ermittlung von Testfällen: strukturierte Pfadüberdeckung

1-begrenzte strukturierte Pfadüberdeckung ($k=1$):

Jeder mögliche Pfad ohne Wiederholung einer Aktion wird durchlaufen.

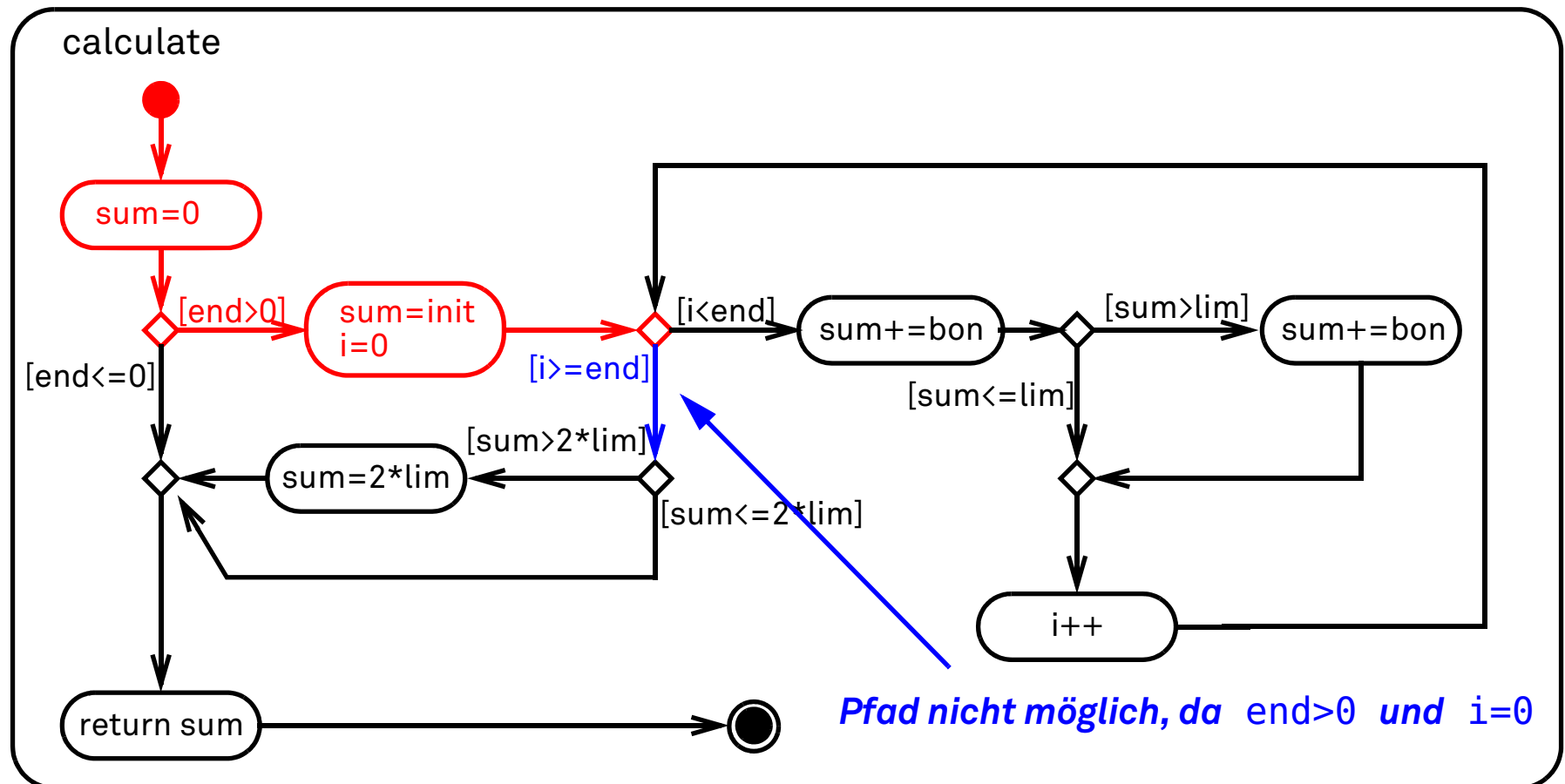


Ermittlung von Testfällen: strukturierte Pfadüberdeckung

(Fortsetzung)

1-begrenzte strukturierte Pfadüberdeckung ($k=1$):

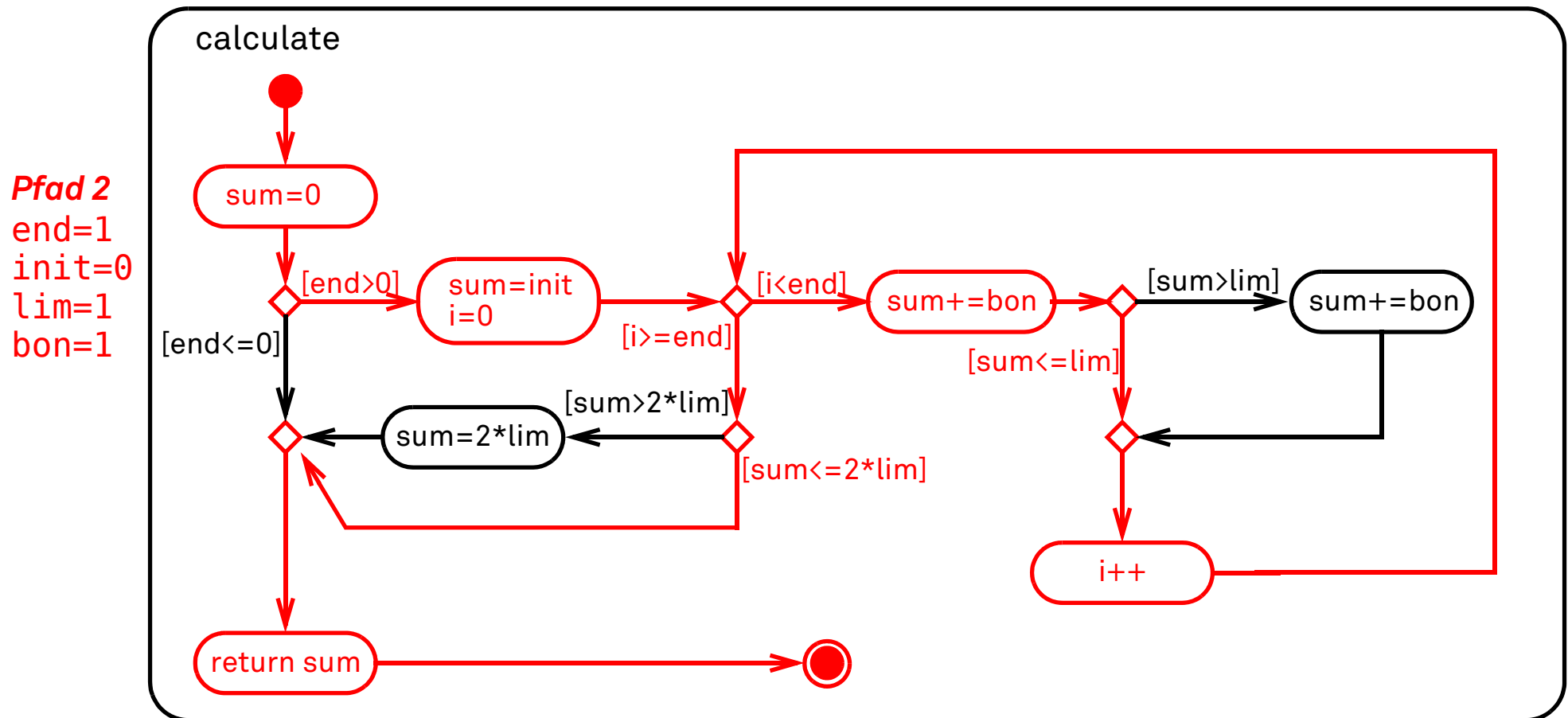
Jeder mögliche Pfad ohne Wiederholung einer Aktion wird durchlaufen.



Ermittlung von Testfällen: strukturierte Pfadüberdeckung

(Fortsetzung)

1-begrenzte strukturierte Pfadüberdeckung ($k=1$):
Jeder mögliche Pfad ohne Wiederholung einer Aktion wird durchlaufen.



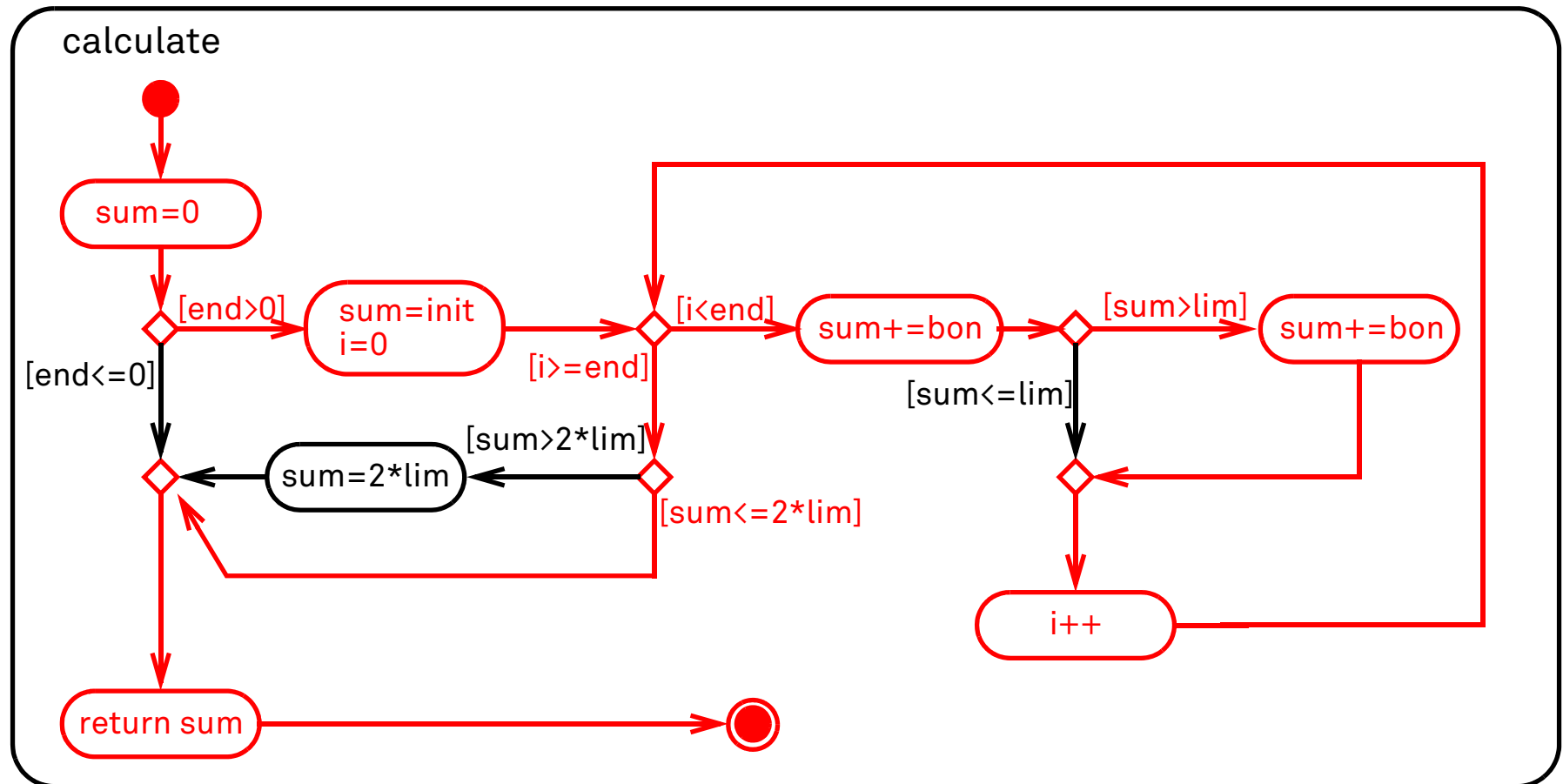
Ermittlung von Testfällen: strukturierte Pfadüberdeckung

(Fortsetzung)

1-begrenzte strukturierte Pfadüberdeckung ($k=1$):

Jeder mögliche Pfad ohne Wiederholung einer Aktion wird durchlaufen.

Pfad 3
end=1
init=2
lim=2
bon=1

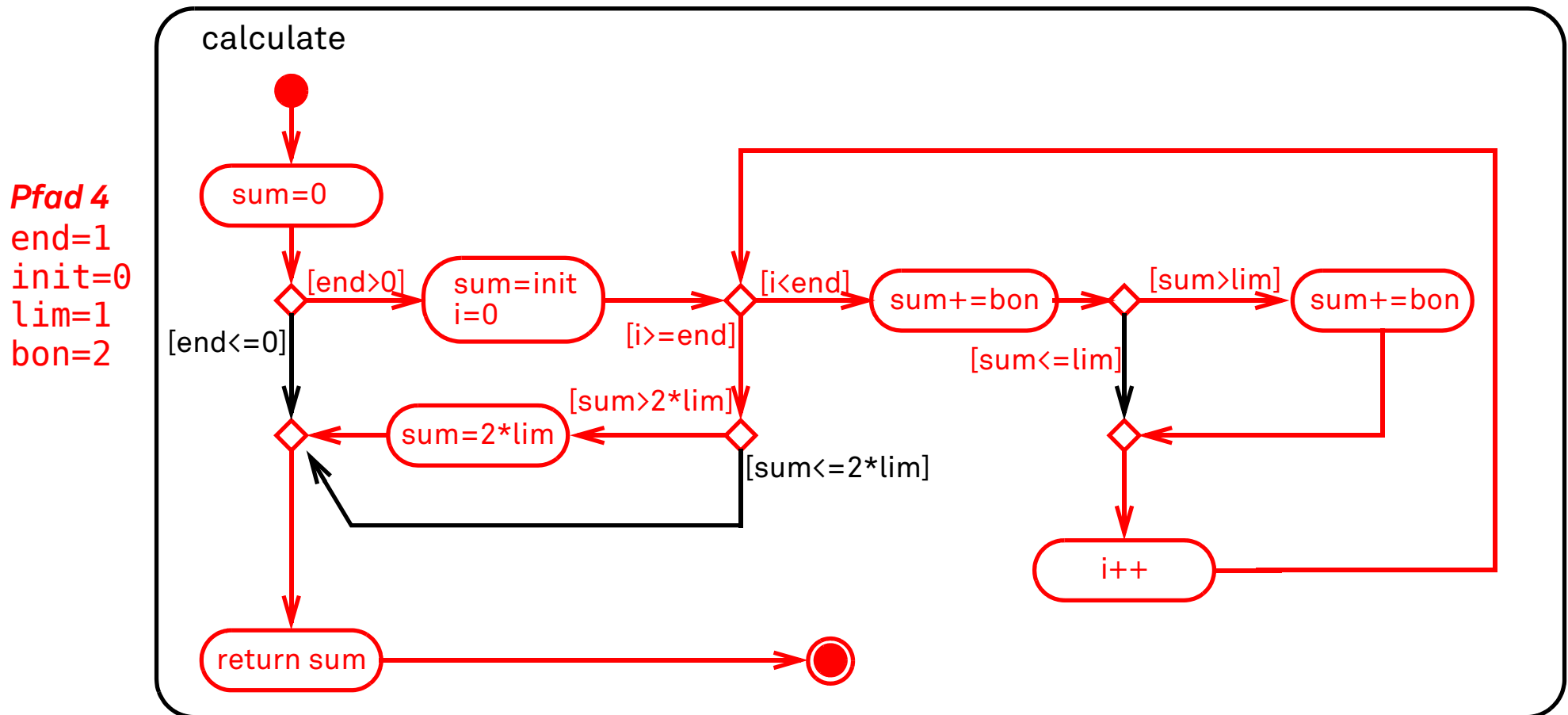


Ermittlung von Testfällen: strukturierte Pfadüberdeckung

(Fortsetzung)

1-begrenzte strukturierte Pfadüberdeckung ($k=1$):

Jeder mögliche Pfad ohne Wiederholung einer Aktion wird durchlaufen.

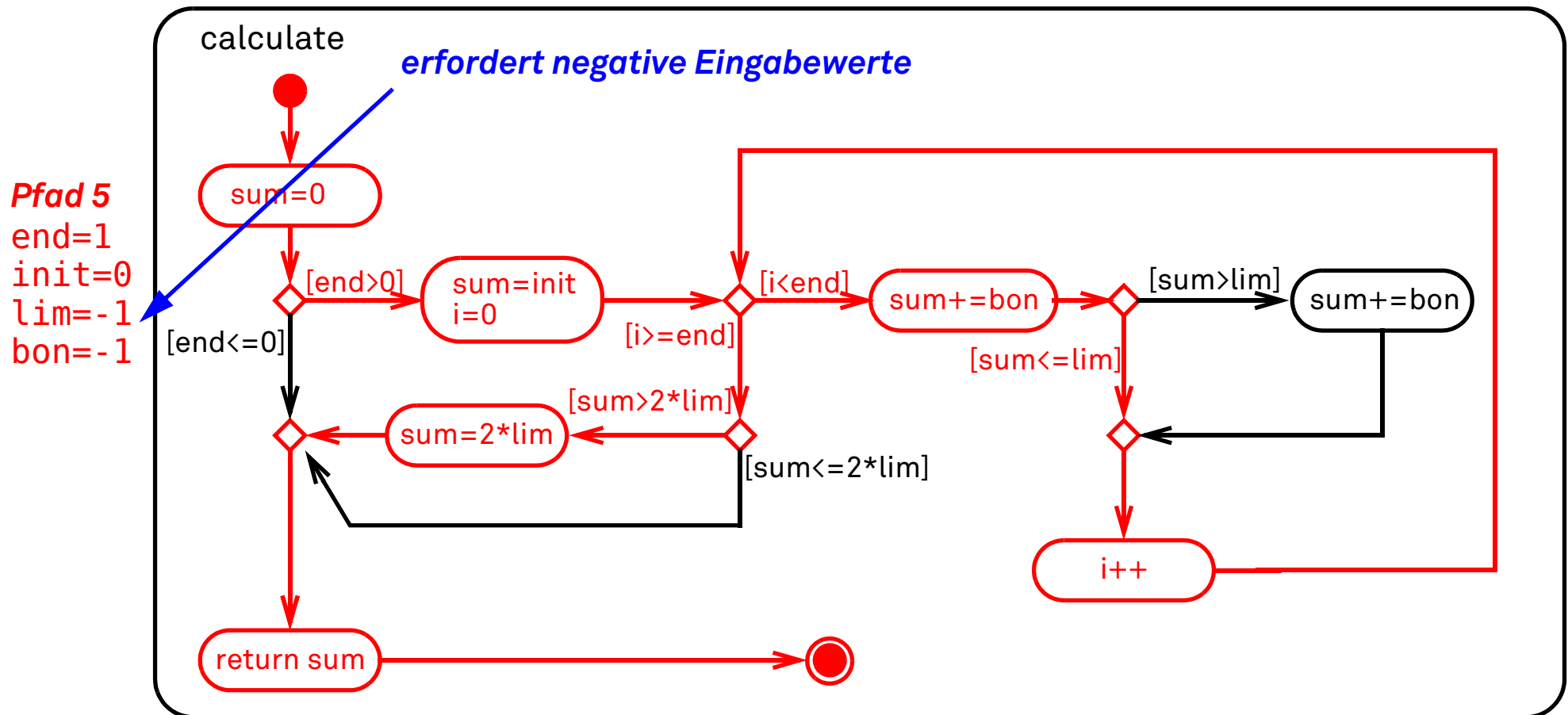


Ermittlung von Testfällen: strukturierte Pfadüberdeckung

(Fortsetzung)

1-begrenzte strukturierte Pfadüberdeckung (k=1):

Jeder mögliche Pfad ohne Wiederholung einer Aktion wird durchlaufen.



Ermittlung von Testfällen: strukturierte Pfadüberdeckung

(Fortsetzung)

Analyse der Leistungsfähigkeit der Pfadüberdeckung

- ❑ Die Pfadüberdeckung ist wegen der kombinatorischen Komplexität in der Praxis kaum anwendbar.
- ❑ Eine – ebenfalls aufwändige – Alternative ist die k -begrenzte strukturierte Pfadüberdeckung

Güte eines Tests T :

- ❑ Überdeckungsgrad $G = (\text{unter } T \text{ überdeckte Pfade}) / (\text{alle Pfade})$
- ❑ **aber:** in der Regel können nicht alle theoretisch möglichen Pfade während der Ausführung tatsächlich abgedeckt werden.
Daher kann die Güte nur schwer festgestellt werden.

Vereinfachte Schleifenüberdeckung

Eine Überprüfung von Schleifen kann zusätzlich nach folgendem Schema erfolgen:

Vorgaben für die **vereinfachte Schleifenüberdeckung**:

Die folgenden fünf Testfälle müssen berücksichtigt werden, sofern sie auftreten können:

- ❑ Schleife wird nicht betreten – prüft den Misserfolg der Eintrittsbedingung
- ❑ Es erfolgt genau 1 Durchlauf durch die Schleife – prüft Initialisierungen
- ❑ Es erfolgen genau 2 Durchläufe durch die Schleife – prüft Initialisierungen
- ❑ Es erfolgt eine typische Anzahl von Durchläufen – prüft Abbruchkriterien
- ❑ Die maximale Anzahl von Durchläufen wird erreicht – prüft Abbruchkriterien

- ❑ Bei zwei geschachtelten Schleifen kombinieren sich diese Tests bereits zu 21 Fällen:
 - Die äußere Schleife wird nicht betreten.
 - Jeder der anderen 4 Testfälle für die äußere Schleife erfordert 5 Testfälle für die innere Schleife.

weitere strukturorientierte Tests

bisher vorgestellt:

Die Testfälle für strukturorientierte Tests werden so bestimmt, dass eine bestimmte Form der Überdeckung des Quelltextes von Methoden erreicht wird.

Für *größere* Programmstrukturen

können die Testfälle für strukturorientierte Tests zusätzlich anhand der Überdeckung der Bestandteile dieser Strukturen ermittelt werden:

- ☐ Überdeckung der von einer Klasse angebotenen Methoden
- ☐ Überdeckung der in einer Klasse aufgerufenen Methoden
- ☐ Überdeckung der ausgelösten/behandelten Ausnahmen
- ☐ ...

Strukturorientierte Tests (Zusammenfassung)

- ❑ Anweisungsüberdeckung (C_0)
- ❑ Zweigüberdeckung (C_1)
- ❑ Mehrfachbedingungsüberdeckung
- ❑ modifizierte Bedingungs-/Entscheidungsüberdeckung
- ❑ Pfadüberdeckung
- ❑ k-begrenzte strukturierte Pfadüberdeckung
- ❑ vereinfachte Schleifenüberdeckung

Beispiel: Apple IOS Implementierung des Secure Sockets Layer (SSL)

Das Beispiel zeigt, dass an einem kritischen Teil des Apple IOS-Betriebssystems auf sorgfältige strukturierte Tests verzichtet wurde.

Die entsprechende Implementierung wurde erstellt: September 2012

Der Fehler wurde erkannt: Februar 2014

- ❑ Ziel der Software u.a.:
Prüfung, dass der Schlüssel für eine gesicherte Verbindung auch vom gewünschten Partner bereitgestellt wird
- ❑ fehlerhafte Implementierung:
 - Prüfung entfällt völlig
 - jeder Schlüssel wird akzeptiert

Beispiel: Apple IOS Implementierung des Secure Sockets Layer (SSL)

(Fortsetzung)

IOS – Codefragment:

```
if ((err = ReadyHash(&SSLHashSHA1, &hash(tx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams) != 0)
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut) != 0)
    goto fail;
err = sslRawVerify(ctx, ...);
...
fail: ...
```

Aufbau der Software:

Viele aufeinander folgende Methodenaufrufe, die bei korrekter Ausführung jeweils den Wert 0 zurückgeben. Bei nicht-korrektter Ausführung wird der Wert gespeichert und mit einer Fehlerbehandlung an der Marke `fail` fortgefahren.

Beispiel: Apple IOS Implementierung des Secure Sockets Layer (SSL)

(Fortsetzung)

IOS – Codefragment:

```
if ((err = ReadyHash(&SSLHashSHA1, &hash(tx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams) != 0)
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut) != 0)
    goto fail;
err = sslRawVerify(ctx, ...);
...
fail: ...
```

Fortsetzung bei fail:

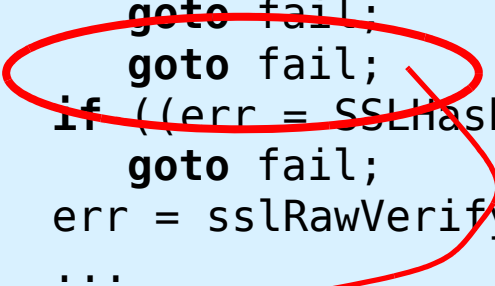
An der Marke `fail` wird *nicht* abgebrochen, da vom Scheitern einer der vorangehend aufgerufenen Methoden ausgegangen wird.

Beispiel: Apple IOS Implementierung des Secure Sockets Layer (SSL)

(Fortsetzung)

IOS – Codefragment:

```
if ((err = ReadyHash(&SSLHashSHA1, &hash(tx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams) != 0)
    goto fail;
goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut) != 0)
    goto fail;
err = sslRawVerify(ctx, ...);
...
fail: ...
```



Fehler: unkontrollierter Sprung, die nachfolgenden Anweisungen sind nicht erreichbar.

Beispiel: Apple IOS Implementierung des Secure Sockets Layer (SSL)

(Fortsetzung)

IOS – Codefragment:

```
if ((err = ReadyHash(&SSLHashSHA1, &hash(tx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams) != 0)
    goto fail;
goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut) != 0)
    goto fail;
err = sslRawVerify(ctx, ...);
...
fail: ...
```

nicht erreichbarer Aufruf

Es kann unmittelbar festgestellt werden:

- zusammenhängendes Aktivitätsdiagramm kann nicht gezeichnet werden
- C_0 -Überdeckung mit $G=1,0$ kann nicht erreicht werden
- ein ausreichender strukturorientierter Test kann nicht erfolgt sein

Beispiel: Apple IOS Implementierung des Secure Sockets Layer (SSL)

(Fortsetzung)

IOS – Codefragment:

```
if ((err = ReadyHash(&SSLHashSHA1, &hash(tx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams) != 0)
    goto fail;
goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut) != 0)
    goto fail;
err = sslRawVerify(ctx, ...);
...
fail: ...
```

- ❑ unkontrollierter Sprung, die nachfolgenden Anweisungen sind nicht erreichbar
- ❑ ein Scheitern der Funktionalität von `sslRawVerify` ist nicht betrachtet worden:
 - Äquivalenzklassen für funktionales Verhalten sind nicht angelegt und getestet worden
 - ein ausreichender funktionsorientierter Test ist ebenfalls nicht erfolgt

Beispiel: Apple IOS Implementierung des Secure Sockets Layer (SSL)

(Fortsetzung)

IOS – Codefragment:

```
if ((err = ReadyHash(&SSLHashSHA1, &hash(tx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams) != 0)
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut) != 0)
    goto fail;
err = sslRawVerify(ctx, ...);
...
fail: ...
```

- ❑ unkontrollierter Sprung, die nachfolgenden Anweisungen sind nicht erreichbar
- ❑ ein Scheitern der Funktionalität von `sslRawVerify` ist nicht betrachtet worden
- ❑ eine andere Programmstruktur hätte den Fehler möglicherweise konstruktiv vermieden: Blöcke, `if-else`, `switch`, eine gemeinsame Bedingung mit short-cut-Disjunktionen

Testen objektorientierter Systeme

Objektorientierung führt zu komplexen Beziehungen zwischen den zu testenden Methoden:

- ❑ Die Ausführung einer Methode hinterläßt einen Zustand, der die nachfolgenden Ausführungen von Methoden beeinflusst.
- ❑ Vererbung schafft (verborgene) Abhängigkeiten.
- ❑ Polymorphie/dynamische Bindung führt zu einer Vervielfachung der möglichen Pfade.
- ❑ Kapselung durch Zugriffsrechte erschwert die Sicht auf die Zustände von Objekten:
 - Die Herstellung eines bestimmten Zustands ist möglicherweise schwer machbar.
 - Die Abfrage des Zustands ist möglicherweise schwer machbar.
- ❑ Klassen sind häufig funktional umfangreicher als notwendig:
=> Testen nicht benötigter Methoden verursacht zusätzlichen Aufwand.

Testen objektorientierter Systeme

(Fortsetzung)

Die Ausführung einer Methode hinterläßt einen Zustand,
der die nachfolgenden Ausführungen von Methoden beeinflusst.

Beispiel

Klasse `ArrayList<E>` mit den Methoden `add(E elem)` und `remove(E elem)`:

- ❑ Jeder Aufruf von `add` erfolgt für eine Liste, die eventuell Elemente enthält, jeder Aufruf von `add` hinterläßt immer eine veränderte Liste.
- ❑ Jeder Aufruf von `remove` erfolgt für eine Liste, die eventuell Elemente enthält, ein Aufruf von `remove` hinterläßt manchmal eine veränderte Liste.
- ❑ Neben dem Parameter `elem` bildet das `ArrayList`-Objekt selbst mit allen seinen Elementen für beide Methode eine weitere Eingabe für die Ausführung.
- ❑ Zugleich ist die geänderte Liste auch das Ergebnis dieser Ausführung.
- ❑ Der Zustand der Liste geht in die Gestaltung von Äquivalenzklassen ein und hat für die Bestimmung der Testfälle eine wesentliche Bedeutung:
zustandsbasiertes Testen

Testen objektorientierter Systeme

(Fortsetzung)

Folgen des Einsatzes von Spezialisierung/Vererbung:

- ❑ Fehler in Oberklassen werden an die Unterklassen weitergegeben.
- ❑ Das Testen der Unterklassen ist nicht ohne Einbeziehung der Oberklassen möglich.
- ❑ Änderungen an Oberklassen erfordern immer erneutes Testen aller Unterklassen.
- ❑ Objekte von Unterklassen können Objekte der Oberklassen ersetzen:
 - Wird Software mit Referenzen/Zeigern auf Oberklassen getestet, müssen eventuell Objekte aller Unterklassen in den Test eingebracht werden.
 - Wird eine Unterklasse geändert, muss eventuell die Nutzung von Referenzen auf die Oberklasse überprüft werden.

Anmerkungen: Testen objektorientierter Systeme

(Fortsetzung)

Folgen des Einsatzes von Spezialisierung/Vererbung

Beispiel

```
class ArrayList<E> {  
    public void add(E elem) { ... }  
    public void remove(E elem) { ... }  
}
```

```
class SpecialList<E> extends ArrayList<E> {  
    public void add(E elem) { ... super.add(x); ... }  
}
```

- ❑ Ein Änderung der Methode `add` in der Klasse `ArrayList` ändert unmittelbar das Verhalten der Klasse `SpecialList`.
- ❑ Ein Änderung der Methode `remove` in der Klasse `ArrayList` ändert ebenfalls unmittelbar das Verhalten der Klasse `SpecialList`.

Testen objektorientierter Systeme

(Fortsetzung)

Folgen des Einsatzes von dynamischer Bindung:

- ❑ Der Programmablauf kann nicht unmittelbar aus Programmcode abgeleitet werden.
- ❑ Alle durch dynamisches Binden mögliche Abläufe müssen getestet werden.
- ❑ Methoden aus verschiedenen Stufen der Vererbungshierarchie arbeiten in verschiedenen Kombinationen zusammen und müssen in (allen) möglichen Kombinationen getestet werden.

Testen objektorientierter Systeme

(Fortsetzung)

Folgen des Einsatzes von dynamischer Bindung

Beispiel

```
public void sort(Comparable[] elems) { ... }
```

- ❑ Die Methode `sort` soll das übergebene Feld sortieren.
- ❑ Die Methode `sort` kann mit allen Feldern aufgerufen werden, deren Elemente das Interface `Comparable` implementieren, also eine `compareTo`-Methode bereitstellen.
- ❑ Die `sort`-Methode kann also auf Felder mit sehr unterschiedlichen Inhalten angewandt werden, soll aber für alle Aufrufe korrekt arbeiten.
- ❑ In dem als Parameter übergebenen Feld könnten auch Objekte verschiedener typkompatibler Klassen abgelegt sein. Dann ist bei der Ausführung möglicherweise entscheidend, welches Objekt die `compareTo`-Methode bereitstellt.
- ❑ Beim Implementieren und Testen der Methode `sort` kann immer nur von einer korrekten Implementierung der aufgerufenen `compareTo`-Methoden ausgegangen werden.
- ❑ Bei einer Nutzung der Methode `sort` muss deren *Eignung* aber möglicherweise in jedem Einzelfall durch Tests nachgewiesen werden.

Operationalisierung der Testdurchführung:

- ❑ Um Vertrauen in die Korrektheit einer Implementierung zu gewinnen müssen funktionsorientierte Tests
und
strukturorientierte Tests
immer gemeinsam durchgeführt werden.
- ❑ Schon kleine Beispiele zeigen,
dass häufig eine (sehr) große Zahl von Testfällen benötigt wird.
- ❑ Die Durchführung von Tests muss daher systematisch erfolgen.
- ❑ Tests müssen nachvollziehbar dokumentiert werden.
- ❑ Tests müssen nach jeder Änderungen an der Software wiederholt werden (können).

daraus folgt:

- ❑ Tests müssen als Testprogramme entwickelt werden,
die jederzeit erneut ausgeführt werden können.
- ❑ Hierfür gibt es Hilfsmittel (= Software-Werkzeuge), z.B. das Framework **JUnit**.

Folien zur Vorlesung **Softwaretechnik**

Abschnitt 4.6: Testunterstützung durch JUnit

JUnit

- ❑ Java-Framework
 - Menge von Klassen, die durch Erben und Benutzen verwendet werden können
 - feste Ausführungskomponenten für die benutzten Klassen
 - aktuelle Version: 4.12
 - Download: <http://junit.org/>
- ❑ grundlegende Ideen:
 - Implementierung und Test werden in getrennten Klassen verwaltet
 - eigenes Testprogramm
 - standardisierte Implementierung der Tests
 - standardisierte Überprüfung der Testergebnisse

- ❑ Beispiel:

```
public class Simple {  
    private double value;  
    public Simple(double v) { value = v; }  
    public double getValue() { return value; }  
}
```

Implementierung:

```
public class Simple {  
    private double value;  
    public Simple(double v) {...}  
    public double getValue() {...}  
}
```

Test:

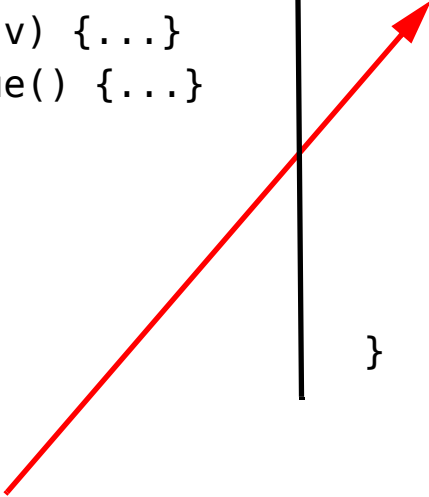
```
import org.junit.Test;  
import static org.junit.Assert.*;  
import org.junit.runner.JUnitCore;  
  
public class SimpleTest {  
    @Test public void testValue() {  
        Simple two = new Simple(2.0);  
        assertEquals(2.0,two.getValue(),0.01);  
    }  
    public static void main(String[] args) {  
        JUnitCore.run(SimpleTest.class);  
    }  
}
```


Implementierung:

```
public class Simple {  
    private double value;  
    public Simple(double v) {...}  
    public double getValue() {...}  
}
```

Test:

```
import org.junit.Test;  
import static org.junit.Assert.*;  
import org.junit.runner.RunWith;  
  
public class SimpleTest {  
    @Test public void testValue() {  
        Simple two = new Simple(2.0);  
        assertEquals(2.0,two.getValue(),0.01);  
    }  
    public static void main(String[] args) {  
        JUnitCore.run(SimpleTest.class);  
    }  
}
```



durch Annotation gekennzeichnete Methode, die einen Testfall implementiert:
`@Test public void test...()`
(So annotierte Methoden werden im Rahmen der Testausführung erkannt und ausgeführt.)

Exkurs: Annotationen in Java

- ❑ Annotationen sind eine Möglichkeit, Programmtexte mit zusätzlichen Informationen (Metadaten) anzureichern.
- ❑ Annotationen haben einen fest vorgegebenen syntaktischen Aufbau.
- ❑ Annotationen können
 - während der Übersetzung oder
 - während der Ausführung ausgewertet werden.
- ❑ Beispiele für vordefinierte Annotationen:
 - `@Override` markierte Methode überschreibt Methode der Oberklasse
 - `@Deprecated` «veraltete» Methode, sollte nicht genutzt werden
 - `@SuppressWarnings` unterdrückt Warnungen des Compilers

Implementierung:

```
public class Simple {
    private double value;
    public Simple(double v) {...}
    public double getValue() {...}
}
```

Test:

```
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.runner.RunWith;
```

```
public class SimpleTest {
    @Test public void testValue() {
        Simple two = new Simple(2.0);
        assertEquals(2.0, two.getValue(), 0.01);
    }
    public static void main(String[] args) {
        JUnitCore.run(SimpleTest.class);
    }
}
```

importiert statische Methoden so,
dass sie ohne Klassenprefix
verwendet werden können

Methode zur Überprüfung einer Bedingung:
assertEquals erwartet, dass Werte von erstem und zweitem
Parameter um höchstens den Wert des dritten Parameters
voneinander abweichen.

Implementierung:

```
public class Simple {  
    private double value;  
    public Simple(double v) {...}  
    public double getValue() {...}  
}
```

Test:

```
import org.junit.Test;  
import static org.junit.Assert.*;  
import org.junit.runner.RunWith;  
  
public class SimpleTest {  
    @Test public void testValue() {  
        Simple two = new Simple(2.0);  
        assertEquals(2.0,two.getValue(),0.01);  
    }  
    public static void main(String[] args) {  
        JUnitCore.run(SimpleTest.class);  
    }  
}
```



Ausführung der Testfälle (= mit @Test gekennzeichnete Methoden)
(Der explizite Aufruf entfällt in Programmier- und Entwicklungsumgebungen.)

assert...-Methoden

- ❑ assert-Methoden melden das Ergebnis ihrer Auswertung an das Framework.
- ❑ Das Framework sammelt diese Meldungen und erstellt einen Gesamtbericht zum Testerfolg.

Beispiele für assert-Methoden:

- ❑ assertTrue(**boolean** c), assertFalse(**boolean** c)
- ❑ assertEquals(Object expected, Object actual),
assertEquals(String e, String a)
(basieren auf Vergleich mit der equals-Methode der entsprechenden Klasse)
assertEquals(**int** e, **int** a),
assertEquals(**double** e, **double** a, **double** delta), ...
- ❑ assertNull(Object o)
assertNotNull(Object o)
- ❑ assertSame(Object e, Object a)
assertNotSame(Object e, Object a)
(Diese Methoden basieren auf dem Vergleich der Referenzen.)

Erweiterung

Implementierung:

```
public class Simple {  
    private double value;  
    public Simple(double v) {...}  
    public double getValue() {...}  
    public Simple add(Simple s) {...}  
}
```

Test:

```
public class SimpleTest {  
    @Test public void testValue() {  
        Simple two = new Simple(2.0);  
        assertEquals(two.getValue(),2.0,0.01);  
    }  
    @Test public void testAdd() {  
        Simple two = new Simple(2.0);  
        Simple sum = two.add(two);  
        assertEquals(4.0,sum.getValue(),0.01);  
        assertEquals(2.0,two.getValue(),0.01);  
    }  
}
```

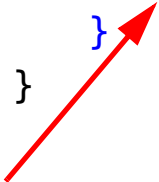
Erweiterung

Implementierung:

```
public class Simple {  
    private double value;  
    public Simple(double v) {...}  
    public double getValue() {...}  
    public Simple add(Simple s) {...}  
}
```

Test:

```
public class SimpleTest {  
    @Test public void testValue() {  
        Simple two = new Simple(2.0);  
        assertEquals(two.getValue(), 2.0, 0.01);  
    }  
    @Test public void testAdd() {  
        Simple two = new Simple(2.0);  
        Simple sum = two.add(two);  
        assertEquals(4.0, sum.getValue(), 0.01);  
        assertEquals(2.0, two.getValue(), 0.01);  
    }  
}
```



Prüfung, ob two
unverändert bleibt

Erweiterung

Implementierung:

```
public class Simple {  
    private double value;  
    public Simple(double v) {...}  
    public double getValue() {...}  
    public Simple add(Simple s) {...}  
}
```

gleiche Anweisungen zur
Vorbereitung der Testfälle



Test:

```
public class SimpleTest {  
    @Test public void testValue() {  
        Simple two = new Simple(2.0);  
        assertEquals(two.getValue(), 2.0, 0.01);  
    }  
    @Test public void testAdd() {  
        Simple two = new Simple(2.0);  
        Simple sum = two.add(two);  
        assertEquals(4.0, sum.getValue(), 0.01);  
        assertEquals(2.0, two.getValue(), 0.01);  
    }  
}
```


Erweiterung

Implementierung:

```
public class Simple {
    private double value;
    public Simple(double v) {...}
    public double getValue() {...}
    public Simple add(Simple s) {...}
}
```

@Before: Die Methode wird
vor jedem Testfall aufgerufen
und fasst Initialisierungen
zusammen.

analog:
@After: Die Methode wird
nach jedem Testfall aufgerufen.

Test:

```
public class SimpleTest {
    private Simple two;
    @Before public void setUp() {
        two = new Simple(2.0);
    }
    @Test public void testValue() {
        Simple two = new Simple(2.0);
        assertEquals(two.getValue(), 2.0, 0.01);
    }
    @Test public void testAdd() {
        Simple two = new Simple(2.0);
        Simple sum = two.add(two);
        assertEquals(4.0, sum.getValue(), 0.01);
        assertEquals(2.0, two.getValue(), 0.01);
    }
}
```

auch möglich in JUnit:

- ❑ Zusammenfassen von Testklassen zu *Testsuiten*, die dann gemeinsam ausgeführt werden
- ❑ Unterstützung des Testens von Ausnahmen
- ❑ Einschränkung:
Da die Testfälle in einer eigenen Klasse programmiert werden, ist **kein** Zugriff zu **privaten** Eigenschaften des getesteten Objekts möglich.

==> Eventuell müssen in der getesteten Klasse zusätzliche Methoden zur Unterstützung des Testens bereitgestellt werden.

alternativ: Build-In-Tests innerhalb der zu testenden Klasse

- ❑ Testwiederholung:
JUnit ermöglicht nach Änderungen an der Implementierung ein einfaches Wiederholen von (allen) Tests.

Debugger

Ein Debugger ist ein Software-Werkzeug, das bei der Lokalisierung von Fehlern eingesetzt wird.

- ❑ Ein Debugger ermöglicht es, die Anweisungen eines Programms einzeln oder bis zu einer bestimmten Position («Breakpoint») auszuführen.
- ❑ Nach Erreichen des Breakpoints hält die Ausführung an.
- ❑ Falls die Ausführung angehalten hat, ermöglicht der Debugger auch das Betrachten der Werte von Variablen und Attributen.
- ❑ Eventuell ermöglicht der Debugger auch ein Ändern der Werte von Variablen oder Attributen.
- ❑ Ein Debugger arbeitet immer interaktiv, erfordert eine manuelle Steuerung des Vorgangs und eine visuelle Prüfung des dargestellten Programmzustands.

- ❑ Der Debugger wird während der Implementierung oder nach dem Erkennen eines Fehlers eingesetzt.
- ❑ Der Debugger ist **nicht** für die allgemeine Überprüfung auf Korrektheit geeignet.

Zusammenfassung Testen

- ❑ funktionsorientierter Test:
Testfälle werden über Äquivalenzklassen aufgrund der Beschreibung bestimmt.
- ❑ strukturorientierter Test:
Testfälle werden aus dem Quellcode abgeleitet mit dem Ziel, bestimmte Überdeckungen zu erreichen.
- ❑ Visualisierung von Programmstrukturen:
UML-Aktivitätsdiagramme
- ❑ Testunterstützung durch Werkzeuge:
Beispiel **JUnit**

Zusammenfassung Testen

(Fortsetzung)

offensichtliche Beobachtung:

- ❑ Viele Änderungen erfordern häufiges Testen.

Lösung:

- ❑ **Regressionstest**
besteht aus der Wiederholung
aller für eine Vorversion erfolgreich ausgeführten Testfälle.
- ❑ Die Korrektheit der Testergebnisse kann teilweise durch den Vergleich
der Ergebnisse der aktuellen Version
mit den Ergebnissen der Vorgängerversion bestimmt werden.
- ❑ Regressionstests müssen automatisiert werden, da
 - viele Testfälle ständig wiederholt werden müssen und zugleich
 - in jedem Testdurchlauf nur wenige Fehler gefunden werden,
da geänderter Programmcode vor der Änderung immer bereits getestet war.