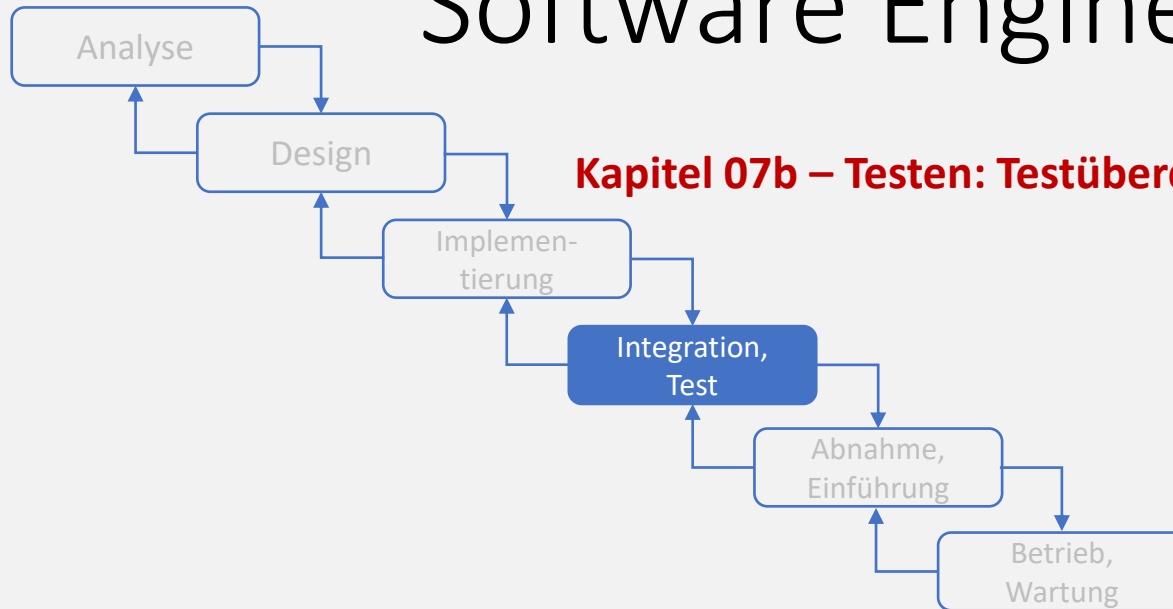




Software Engineering

Kapitel 07b – Testen: Testüberdeckungskriterien



Gliederung

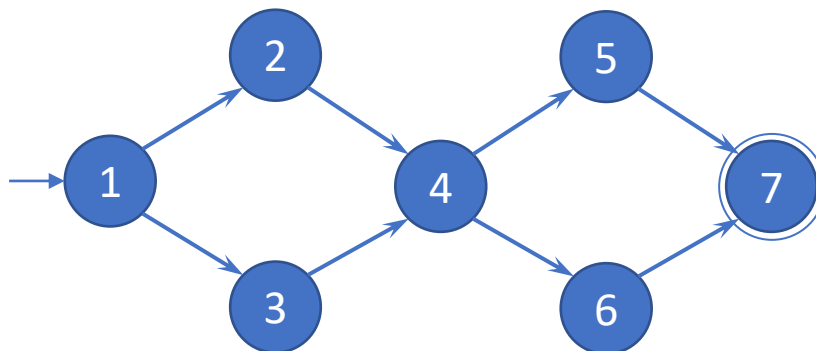
1. Grundlagen zu Graphen
2. Strukturelle Überdeckungskriterien (NC, EC, EPC, CPC)
3. Kontrollflussgraphen (von Code zu Graphen)
4. Datenfluss-Überdeckungskriterien (ADC, AUC, ADUPC)

Graphen (1)

- Graphen sind die meistgenutzte Datenstruktur für Testen
- Wir kennen Graphen in SE bereits in unterschiedlichen Kontexten:
 - (UML-)Modelle
 - Zustandsautomaten
 - Kontrollflussgraphen
 - ...
- **Tests** sollen solche **Graphen** in der Regel in irgendeiner Weise “überdecken“
- Ein Graph $G = (V, V_0, V_f, E)$ ist hier definiert als:
 - V : nicht-leere Menge von Knoten
 - $V_0 \subseteq V$: nicht-leere Menge von Startknoten
 - $V_f \subseteq V$: nicht-leere Menge von Endknoten
 - $E \subseteq V \times V$: Menge von Kanten (v_i, v_j) von Knoten v_i nach v_j
- Ein **Pfad** in einem Graphen $G = (V, V_0, V_f, E)$ ist eine **Knotenfolge**: $[v_1, v_2, \dots, v_n]$, bei der jeweils $(v_i, v_{i+1}) \in E$ ($i \in \{1, \dots, n-1\}$)
- Die **Länge eines Pfades** ist die Anzahl der Kanten, aus denen er besteht

Graphen (2)

- Ein **Testpfad** ist ein Pfad $[v_1, v_2, \dots, v_n]$ für den gilt: $v_1 \in V_0$ und $v_n \in V_f$
 - Ein **Testpfad** p **tourt einen Subpfad** q , falls q Subpfad von p ist.
 - Ein Testpfad **repräsentiert die Ausführung eines Testfalls**
 - Manche Testpfade können von mehreren Tests ausgeführt werden
 - Manche Testpfade können von keinem Test ausgeführt werden
 - **SESE-Graphen** (Single-Entry, Single-Exit)
 - Besondere Graphen, die nur einen Start- und einen Endknoten besitzen (s. unten)
- Frage: wie viele Testpfade hat der folgende Graph:



Testpfade

[1,2,4,5,7]

[1,2,4,6,7]

[1,3,4,5,7]

[1,3,4,6,7]

Graphen als Abstraktion von Software

- Wir nutzen Graphen beim Testen folgendermaßen:
 - Erzeuge ein Modell der Software als Graph
 - Verlange, dass die Tests bestimmte Knoten-/Kanten-/Subpfad-Mengen besuchen
- **Testanforderungen (TR):**
 - Beschreiben Eigenschaften von Testpfaden (z.B. „beinhaltet jeden erreichbaren Knoten von G “)
- **Strukturelle Überdeckungskriterien:**
 - Sind auf einem Graphen lediglich über Knoten und Kanten definiert
 - Beispiele:
 - Knotenüberdeckung (NC: node coverage),
 - Kantenüberdeckung (EC: edge coverage),
 - Kantenpaar-Überdeckung (EPC: edge-pair coverage), ...
- **Datenfluss-Überdeckungskriterien:**
 - Verlangen, dass ein Graph mit weiteren Informationen (Verweise auf Variablen) angereichert ist (dabei helfen uns def- und uses-Informationen; s. später)



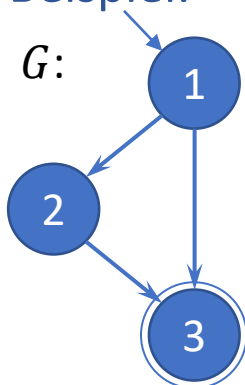
Kriterium: Knotenüberdeckung (NC)

Die erste und einfachste Überdeckungsmethode ist Knotenüberdeckung

- Definition Knotenüberdeckung (node coverage, NC):
 - Eine Menge von Tests T erfüllt das Knotenüberdeckungskriterium auf Graph G , wenn für jeden (syntaktisch) erreichbaren Knoten $v \in V$ ein Pfad $p \in \text{path}(T)$ existiert, so dass p den Knoten v besucht
- Kürzer:
 - TR enthält jeden erreichbaren Knoten von G
 - Intuitiv: “jeder Befehl muss ausführbar sein”

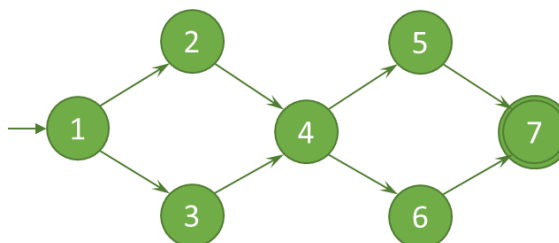
In vielen kommerziellen Testing-Tools implementiert

- Beispiel:



TR = { 1, 2, 3 }

Testpfad = [1, 2, 3] erfüllt alle Kriterien von TR



TR = {
Testpfade = }

Kriterium: Kantenüberdeckung (EC)

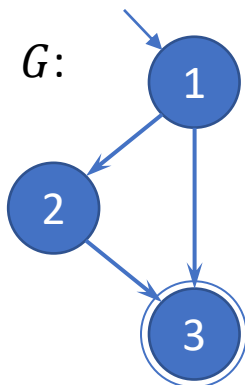
Das Kriterium der Kantenüberdeckung ist etwas ausdrucksstärker

- Definition Kantenüberdeckung (edge coverage, EC):
 - TR enthält jeden erreichbaren Pfad der Länge maximal 1, in G
 - Bemerkung: also auch Graphen erlaubt, die einen Knoten und keine Kanten besitzen



In vielen kommerziellen Testing-Tools implementiert

- Knoten- und Kantenüberdeckung sind nur dann nicht gleich, wenn zwischen zwei Knoten eine Kante und (mind.) ein weiterer Subpfad existiert (s.u.)
- Beispiel:



$TR = \{ (1,2), (2,3), (1,3) \}$
Testpfade = $\{ [1,2,3], [1,3] \}$

Wie sähe ein Code-Beispiel dafür aus, dass gilt: $NC=EC$?

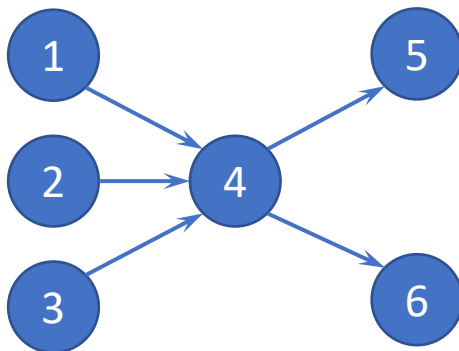
Bemerkung: Testpfad $[1,2,3]$ erfüllt Knotenüberdeckung von G (s. Folie zuvor)

Kriterium: Kantenpaar-Überdeckung (EPC)

Möchte man Paare von Kanten überdecken, muss man das Kriterium der Kantenpaar-Überdeckung nutzen

- Definition Kantenpaar-Überdeckung (edge-pair coverage, EPC):
 - TR enthält jeden erreichbaren Pfad der Länge maximal 2, in G

- Beispiel:



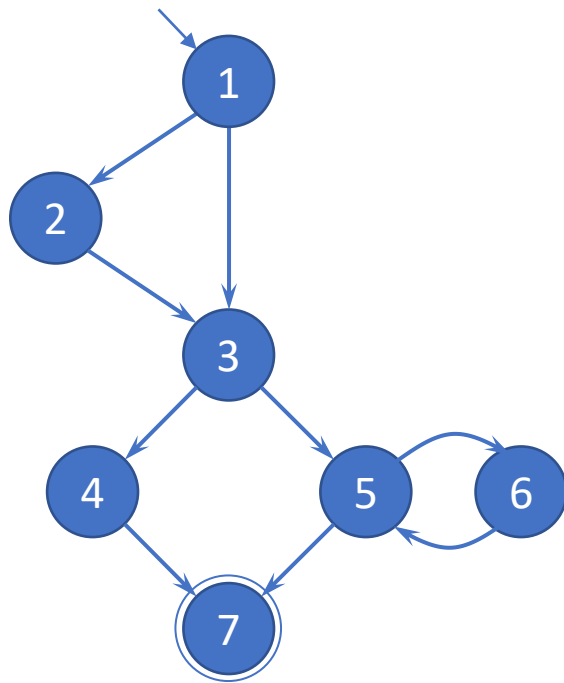
$TR = \{ [1,4,5], [1,4,6], [2,4,5], [2,4,6], [3,4,5], [3,4,6] \}$

Kriterium: Vollständige Pfadüberdeckung (CPC)

Die logische Erweiterung ist, die Überdeckung aller Pfad zu fordern

- Definition Vollständige Pfadüberdeckung (complete path coverage, CPC):
 - TR enthält jeden (erreichbaren) Pfad in G
 - Was ist das Problem an dieser Definition?
- Definition: Spezifizierte Pfadüberdeckung (specified path coverage, SPC):
 - TR enthält eine Menge S von Testpfaden, wobei S als Parameter bereitgestellt wird
 - Was ist das Problem an dieser Definition?

Zusammenfassung: Strukturelle Überdeckung



Knotenüberdeckung (NC)

TR = { 1, 2, 3, 4, 5, 6, 7 }

Testpfade: [1, 2, 3, 4, 7], [1, 2, 3, 5, 6, 5, 7]

Kantenüberdeckung (EC)

TR = { (1,2), (1,3), (2,3), (3,4), (3,5), (4,7), (5,6), (5,7), (6,5) }

Testpfade: ...

Kantenpaar-Überdeckung (EPC)

TR = { [1,2,3], [1,3,4], [1,3,5], [2,3,4], [2,3,5], [3,4,7], [3,5,6],
[3,5,7], [5,6,5], [6,5,6], [6,5,7] }

Testpfade: ...

Vollständige Pfadüberdeckung (CPC)

Testpfade: [1,2,3,4,7], [1,2,3,5,7], [1,2,3,5,6,5,7], [1,2,3,5,6,5,6,5,7], ...

Einfache Pfade und Primpfade

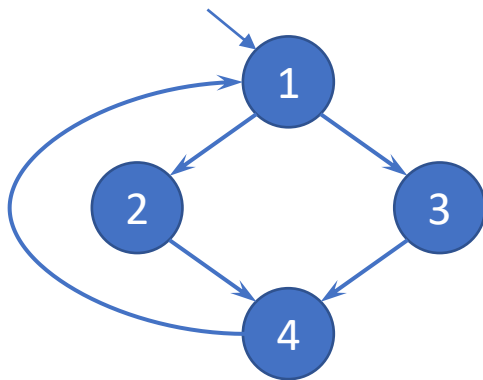
Umgehen mit Kreisen in Graphen:

- Einfacher Pfad:

- Pfad von Knoten von v_i zu v_j , auf dem kein Knoten mehr als einmal vorkommt (außer evtl., dass erster und letzter Knoten gleich sind)
- Hat also keine internen Zyklen
- Ein Zyklus ist ein einfacher Pfad

- Primpfad:

- Einfacher Pfad, der nicht als echter Teilpfad eines anderen einfachen Pfades auftritt



Einfache Pfade

[1], [2], [3], [4], [1,2], [1,3], [2,4], [3,4], [4,1],
[1,2,4], [1,3,4], [2,4,1], [3,4,1], [4,1,2], [4,1,3],
[1,2,4,1], [1,3,4,1], [2,4,1,2], [2,4,1,3], [3,4,1,2], [3,4,1,3],
[4,1,2,4], [4,1,3,4]

Primpfade

UI

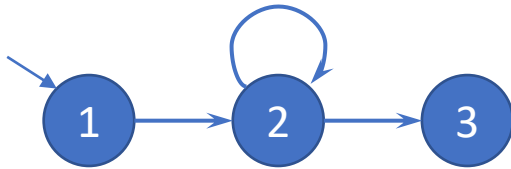
[1,2,4,1], [1,3,4,1], [2,4,1,2], [2,4,1,3], [3,4,1,2], [3,4,1,3],
[4,1,3,4], [4,1,2,4]

Primpfad-Überdeckung

Das Kriterium der Primpfad-Überdeckung

- Ein einfaches und endliches Kriterium fordert, dass Zyklen sowohl ausgeführt als auch übersprungen werden können sollen
- **Definition Primpfad-Überdeckung (prime path coverage, PPC):**
 - TR enthält jeden Primpfad in G
- **Eigenschaften:**
 - Kriterium wird jeden Pfad der Länge 0,1, ... besuchen
 - D.h., es umfasst Knoten- und Kantenüberdeckung
 - Es umfasst fast (aber nicht ganz) EPC
 - Hinweis: Hat ein Knoten einen Selfloop, so verlangt EPC Pfade mit dem Selfloop! Diese sind aber i.d.R. nicht prim! (s. Beispiel)

- **Beispiel:**



Kantenpaar-Überdeckung (EPC)

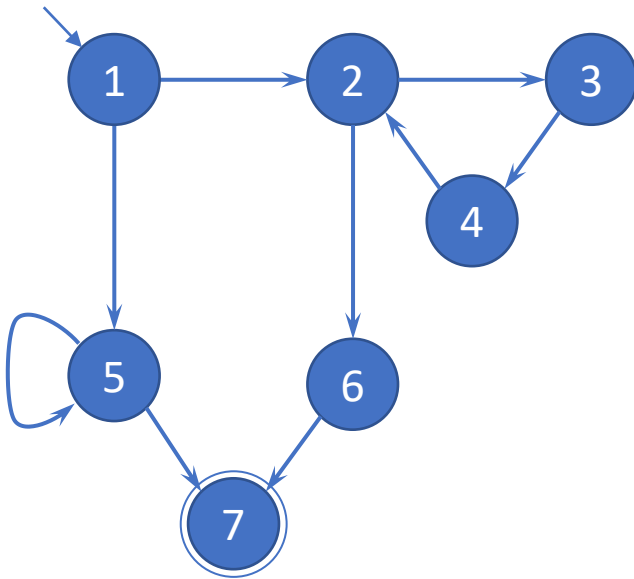
TR = { [1,2,3], [1,2,2], [2,2,3], [2,2,2] }

Primpfad-Überdeckung (PPC)

TR = { [1,2,3], [2,2] }

Algorithmische Berechnung von Primpfaden (1)

1. Berechne iterativ alle einfachen Pfade steigender Länge:

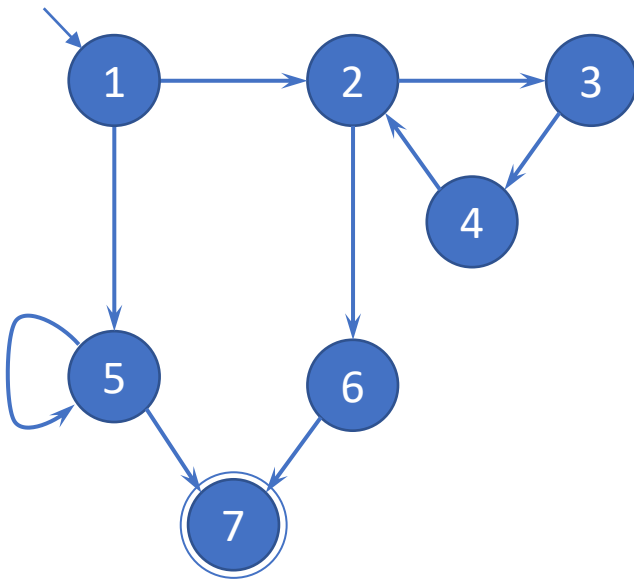


- Einfache Pfade der Länge 0:
 - [1],[2],[3],[4],[5],[6],[7]
- Einfache Pfade der Länge 1:
 - [1,2],[1,5],[2,3],[2,6],[3,4],[4,2],[5,5],[5,7],[6,7]
- Einfache Pfade der Länge 2:
 - [1,2,3],[1,2,6],[1,5,7],[2,3,4],[2,6,7],[3,4,2],[4,2,3],[4,2,6]
- Einfache Pfade der Länge 3:
 - [1,2,3,4],[1,2,6,7],[2,3,4,2],[3,4,2,3],[3,4,2,6],[4,2,3,4],[4,2,6,7]
- Einfache Pfade der Länge 4:
 - [3,4,2,6,7]

Rot: nicht mehr als einfacher Pfad erweiterbar

Grün: Zyklus

Algorithmische Berechnung von Primpfaden (2)



2. Alle einfachen Pfade, die nicht rot oder grün markiert sind, werden gelöscht (da sie erweitert werden können und damit ein echter Teilpfad ihrer Verlängerung sind)

3. Alle Teilpfade von bestehenden einfachen Pfaden werden gelöscht

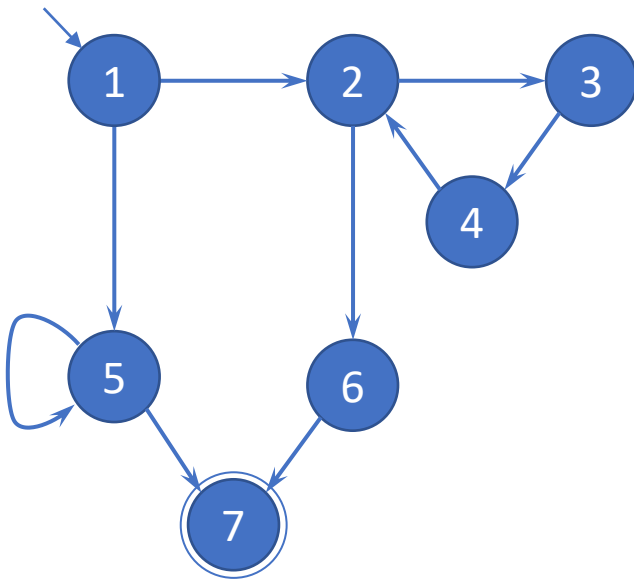
- [1],[2],[3],[4],[5],[6], ~~[7]~~
- [1,2],[1,5],[2,3],[2,6],[3,4],[4,2],[5,5], ~~[5,7]~~, ~~[6,7]~~
- [1,2,3],[1,2,6],[1,5,7], [2,3,4], ~~[2,3,7]~~, [3,4,2],[4,2,3],[4,2,6]
- [1,2,3,4],[1,2,6,7], [2,3,4,2],[3,4,2,3],[3,4,2,6], [4,2,3,4], ~~[4,2,6,7]~~
- [3,4,2,6,7]

- Algorithmus terminiert immer, da Länge des längsten Primpfades $\leq |V|$ (Anzahl Knoten) gilt.

Rot: nicht mehr als einfacher Pfad erweiterbar

Grün: Zyklus

Herleitung von Testpfaden aus Primpfaden



- Ergebnis des Algorithmus:

- [1],[2],[3],[4],[5],[6],[7]
- [1,2],[1,5],[2,3],[2,6],[3,4],[4,2],[5,5] [5,7],[6,7]
- [1,2,3],[1,2,6],[1,5,7] [2,3,4],[2,6,7],[3,4,2],[4,2,3],[4,2,6]
- [1,2,3,4],[1,2,6,7],[2,3,4,2],[3,4,2,3],[3,4,2,6],[4,2,3,4],
[4,2,6,7]
- [3,4,2,6,7]

- Testpfade:

- Z.B. [1,2,6,7], [1,2,3,4,2,3,4,2,6,7], [1,5,7], [1,5,5,7]
würden ausreichen

- Algorithmus von Hand:

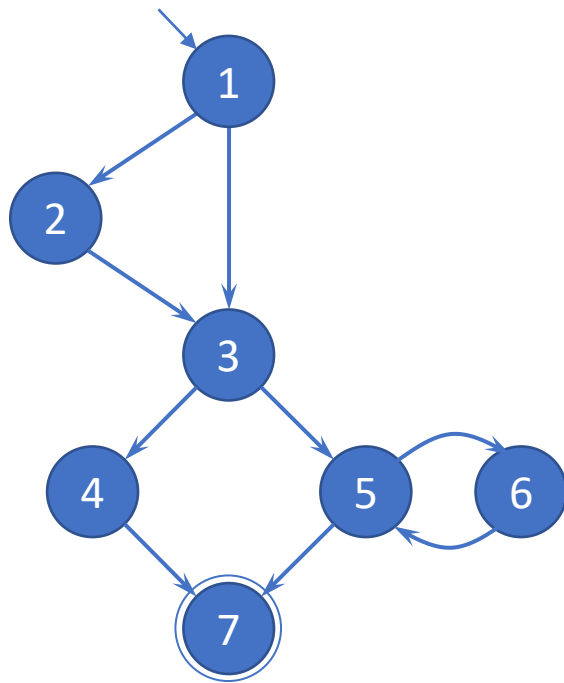
- Wiederhole:

- Verwende einen der aktuell längsten Primpfade und erweitere diese zu Start- und Endknoten:

- Bsp1: [3,4,2,6,7] → [1,2,3,4,2,6,7] (tour 3 Primpfade, s.o.)

- Bsp2: [4,2,3,4] → [1,2,3,4,2,3,4,2,6,7] (tour weitere)

Zusammenfassung: Strukturelle Überdeckung



Einfache Pfade

[1],[2], ..., [7], [1,2],...

insgesamt 38 Stück

Primpfade

[1,2,3,4,7], [1,2,3,5,7], [1,2,3,5,6]

[1,3,4,7], [1,3,5,7], [1,3,5,6]

[5,6,5], [6,5,6], [6,5,7]

insgesamt 9 Stück

Führe Zyklus 0 mal aus

Führe Zyklus 1 mal aus

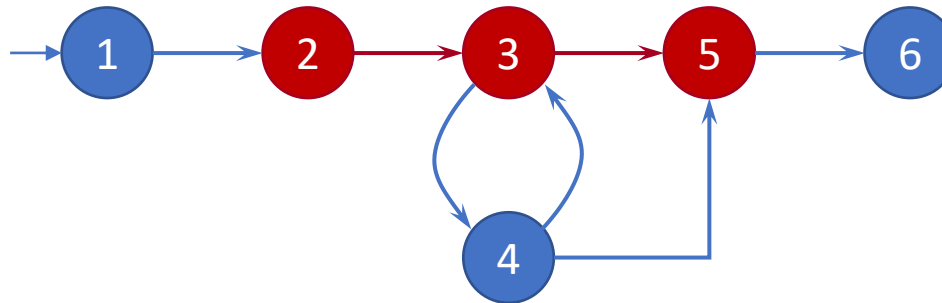
Führe Zyklus mehrfach aus

Motivation für Touren (mit Abstechern oder Umleitungen)

Die aktuelle Situation deckt nicht alle Fälle ab, die wir betrachten müssen

- Motivation:

- In Programmen kann es häufig dazu kommen, dass z.B. Pfade wie [2,3,5] nicht direkt durchlaufen werden können, da die Schleife so ausgelegt ist, dass man sie nicht 0-mal ausführen kann



- Lösung:

- Statt der strikten Definition von Pfaden und Teilpfaden ("Tour"), nutzen wir die etwas weichere von "Tour mit Abstechern" oder "Tour mit Umleitung"

Weitere Definitionen für Pfade als Erweiterungen für unsere Überdeckungsmaße



- **Tour:**

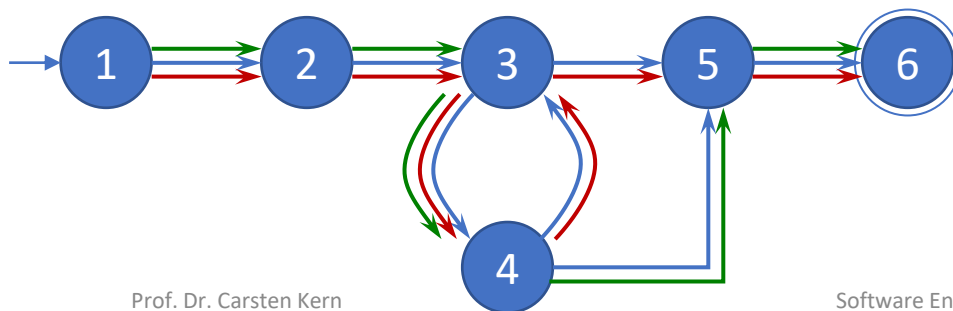
- Testpfad p *tourt* Teilpfad q , wenn q ein Teilpfad von p ist

- **Tour mit Abstechern (tour with sidetrips):**

- Testpfad p *tourt* Teilpfad q *mit Abstechern*, wenn **jede Kante** in q auch in p in der gleichen Reihenfolge vorkommt
- Die Tour kann also an Knoten Abstecher enthalten, solange diese jeweils zurück zum jeweiligen Knoten kommen

- **Tour mit Umleitungen (tour with detours):**

- Testpfad p *tourt* Teilpfad q *mit Umleitung*, wenn **jeder Knoten** in q auch in p in der gleichen Reihenfolge vorkommt
- Die Tour kann eine Umleitung ab Knoten v_i enthalten, solange diese an einem Nachfolger von v_i zurück auf den Primpfad gelangt



Beispiele bei geg. Primpfad $q = [1, 2, 3, 5, 6]$:

1. $p = [1, 2, 3, 5, 6]$ *tourt* q
2. $p = [1, 2, 3, 4, 3, 5, 6]$ *tourt* q **mit Abstecher**
3. $p = [1, 2, 3, 4, 5, 6]$ *tourt* q **mit Umleitung**
4. $[3, 4, 3]$ ist **Abstecher** von $[2, 3, 5]$
5. $[3, 4, 5]$ ist eine **Umleitung** von $[2, 3, 5]$

Ein Zoo an Überdeckungskriterien

Weitere Überdeckungskriterien und deren Zusammenhang:

– Best effort testing:

- Erfülle möglichst viele Test-Anforderungen ohne Abstecher (side trips)
- Nutze Abstecher (side trips) um übriggebliebene Test-Anforderungen zu erfüllen

– Einfache Rundreiseüberdeckung (Simple Round Trip Coverage: SRTC)

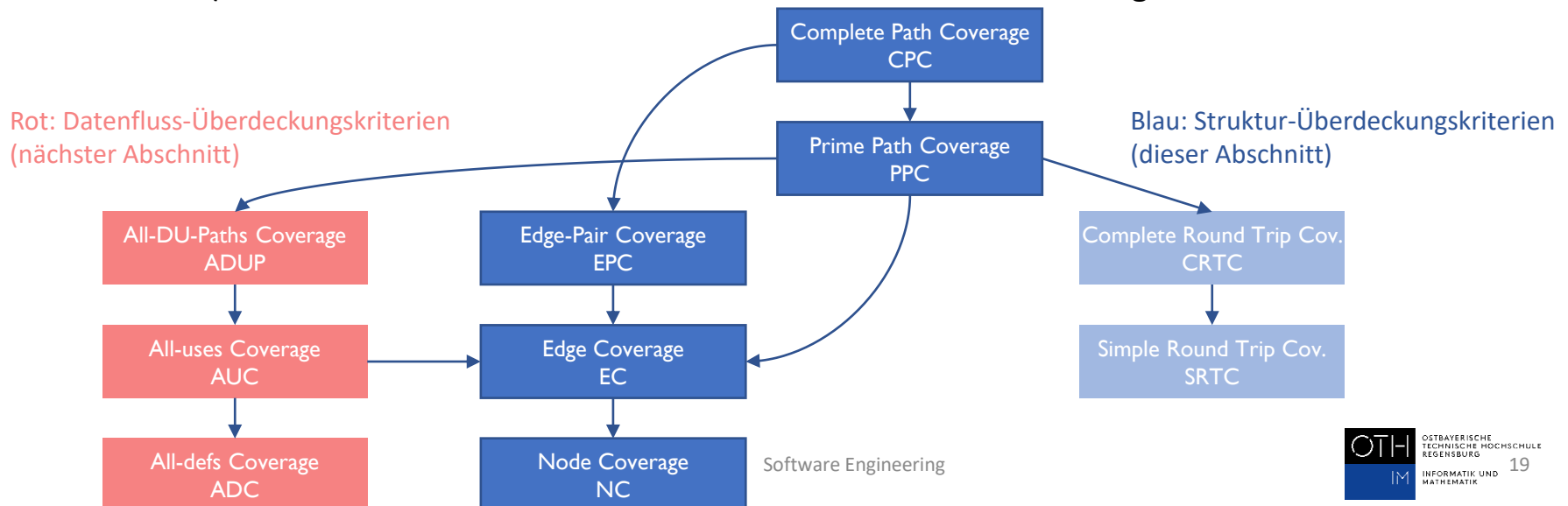
- TR enthält mindestens eine Rundreise für jeden erreichbaren Knoten in G, die an einer Rundreise beginnt und endet

Rundreise (round trip): Primpfad, der am selben Knoten startet und endet

– Vollständige Rundreiseüberdeckung (Complete Round Trip Cov.: CRTC)

- TR enthält alle Rundreise-Pfade für jeden erreichbaren Knoten in G

– SRTC & CRTC vermeiden Knoten und Kanten, die nicht auf Rundreisen liegen (daher subsumieren sie nicht EPC, EC und NC und sind deswegen alleine nicht ausreichend)



Von Code zu Tests

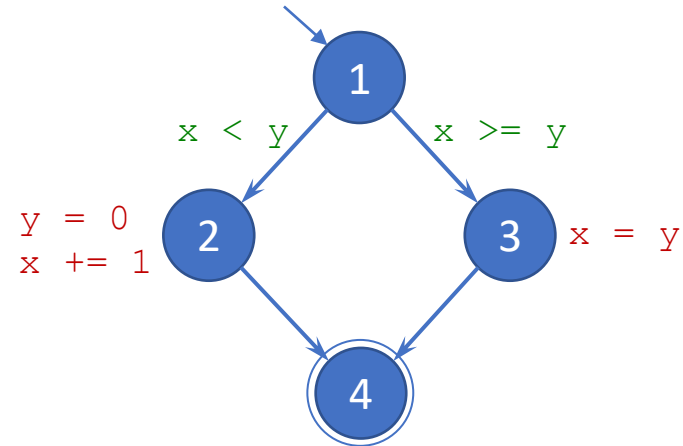
Von Code zu Tests: Kontrollflussgraphen

Wie repräsentieren wir gegebenen Code so, dass wir Tests ableiten können?

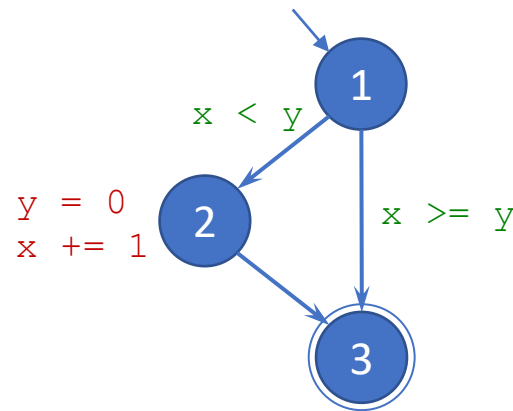
- Idee: bei gegebenem Testartefakt (z.B. Code):
 - Extrahiere einen Graphen,
 - Analysiere diesen nach bekannten Kriterien
- Definition: Kontrollflussgraph (CFG)
 - Stellt alle Ausführungsmöglichkeiten eines Programms/Methode dar
 - Knoten sind dabei Befehle oder Befehlssequenzen (Basisblöcke) (Farben: s. Bspe)
 - Kanten stellen Kontrollfluss dar (Farben: s. Bspe)
 - Wird manchmal mit weiteren Informationen angereichert:
 - Mit Verzweigungsprädikaten
 - Mit defs (Befehle, die Variablen Werte zuweisen)
 - Mit uses (Befehle, die Variablen nutzen)
- Basisblock (elementarer/atomarer Block):
 - Eine Folge von Befehlen für die gilt: wenn der erste Befehl ausgeführt wird, werden auch alle weiteren ausgeführt (keine Verzweigungen)

Übersetzung von Code in CFGs: Der if-Befehl

```
if (x < y) {  
    y = 0;  
    x += 1;  
} else {  
    x = y;  
}
```

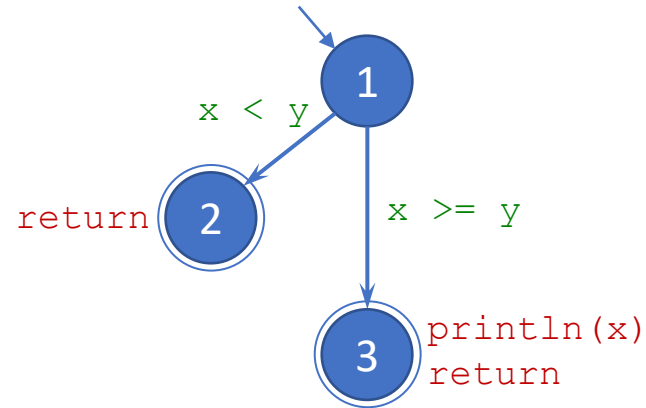


```
if (x < y) {  
    y = 0;  
    x += 1;  
}
```



Übersetzung von Code in CFGs: Der if-return-Befehl

```
if (x < y) {  
    return;  
}  
println(x);  
return;
```



Keine Kante von Knoten 2 nach 3!
Die return-Knoten müssen verschieden sein!

Übersetzung von Code in CFGs: Schleifen (while, for, do-while)



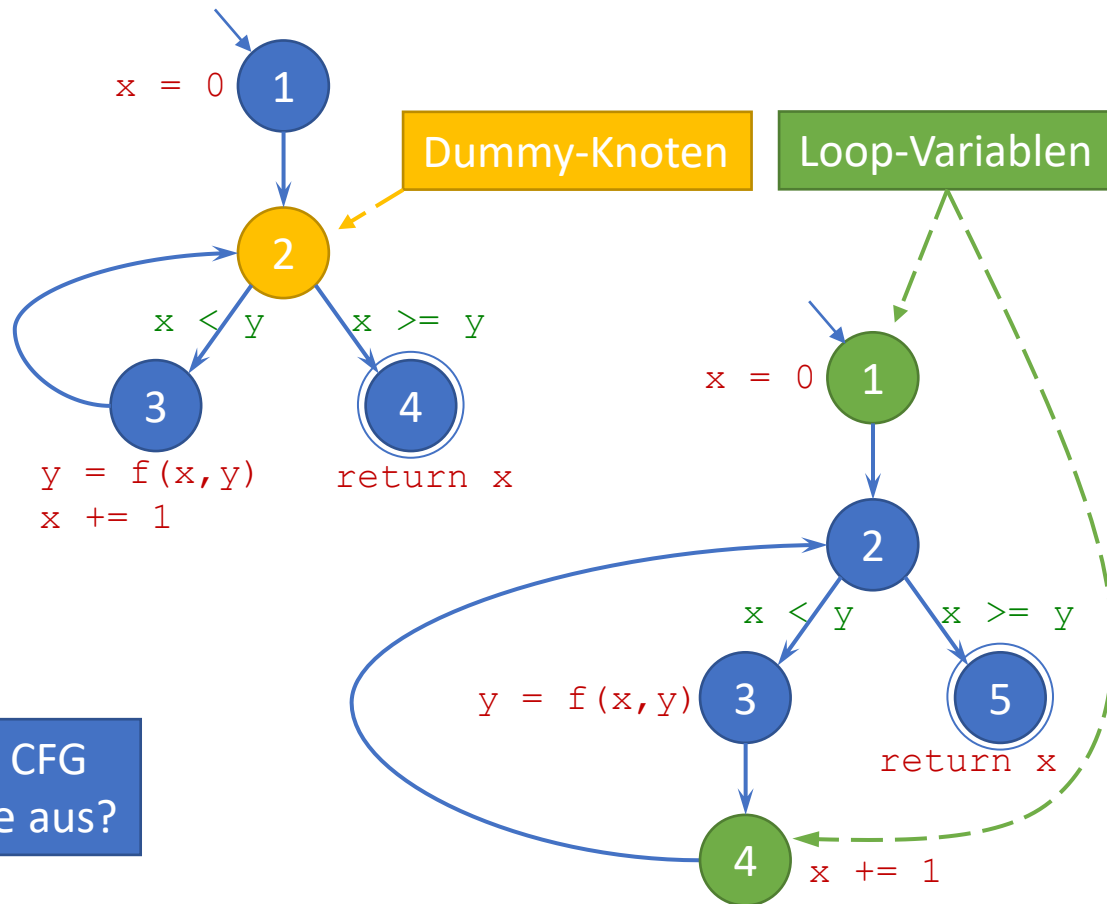
- Bei Schleifen müssen zusätzliche Knoten eingefügt werden
- Diese Knoten stellen weder Befehle noch Basisblöcke dar

```
x = 0;  
while (x < y) {  
    y = f(x, y);  
    x += 1;  
}  
return x;
```

```
for (x = 0; x < y; x++) {  
    y = f(x, y);  
}  
return x;
```

```
x = 0;  
do {  
    y = f(x, y);  
    x += 1;  
} while (x < y)  
return x;
```

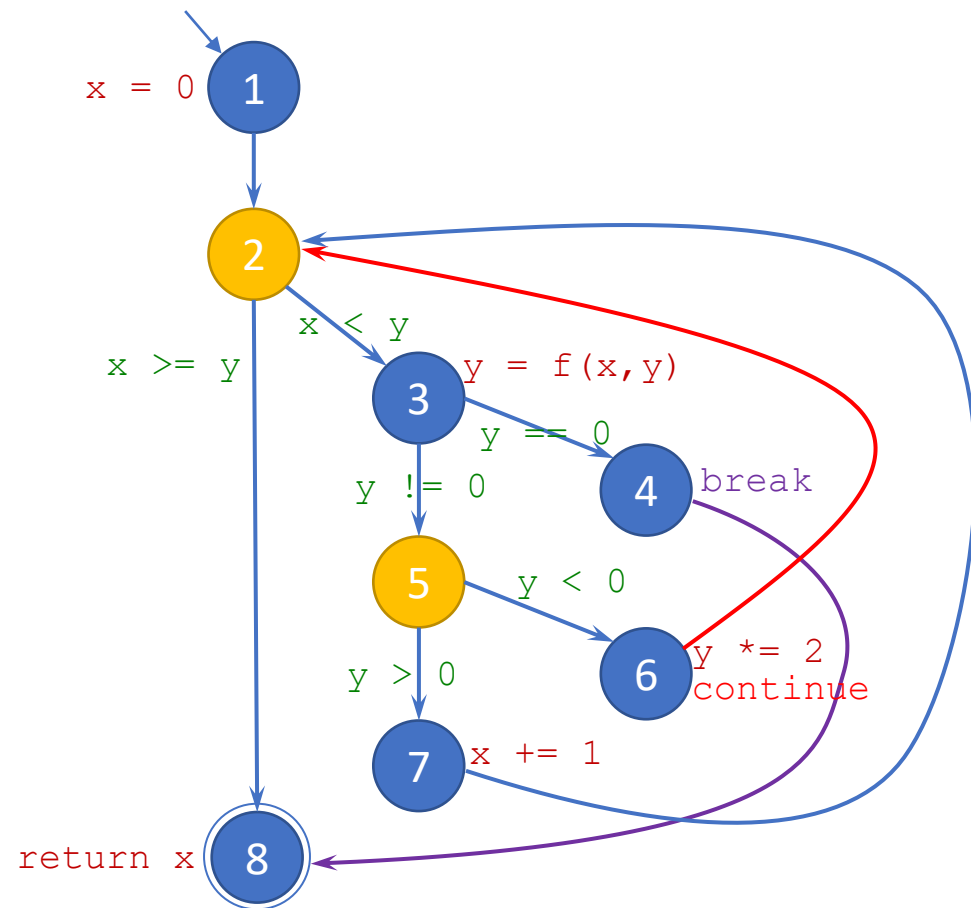
Wie sieht der CFG
für do-while aus?



Übersetzung von Code in CFGs: Schleifen (break, continue)



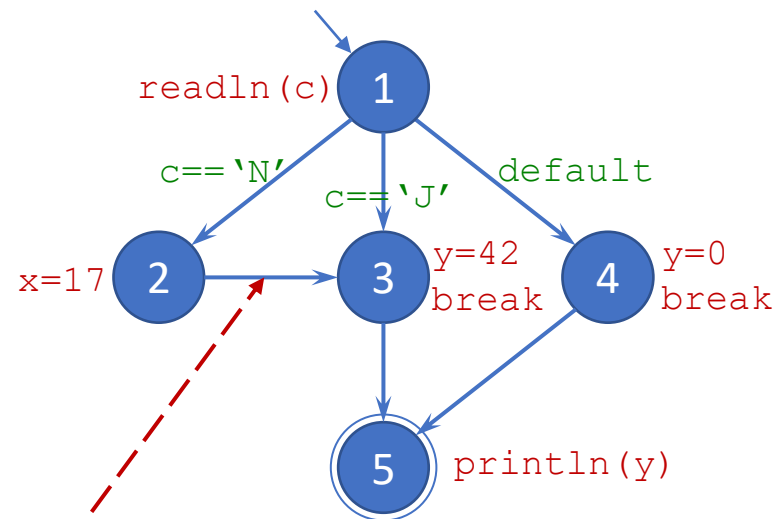
```
x = 0;  
while (x < y) {  
    y = f(x, y);  
    if (y == 0) {  
        break;  
    } else if (y < 0) {  
        y *= 2;  
        continue;  
    }  
    x += 1;  
}  
return x;
```



Übersetzung von Code in CFGs: switch-case-Bedingung



```
readln(c);  
switch c {  
  case 'N':  
    x = 17;  
  case 'J':  
    y = 42;  
    break;  
  default:  
    y = 0;  
    break;  
}  
println(y);
```



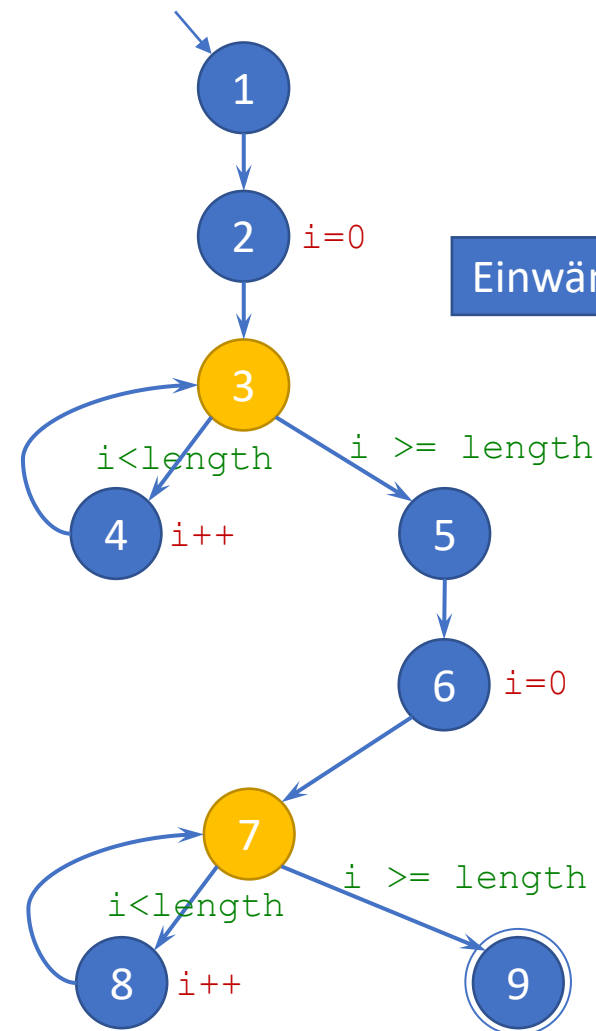
Fehlendes break!

- Weitere Programmiersprachenkonstrukte in CFGs transformierbar, z.B.:
 - Exceptions (try-catch) → s. Übung
 - ...

Erzeuge den Kontrollflussgraphen bzw. Ermittle die zugehörigen Annotationen



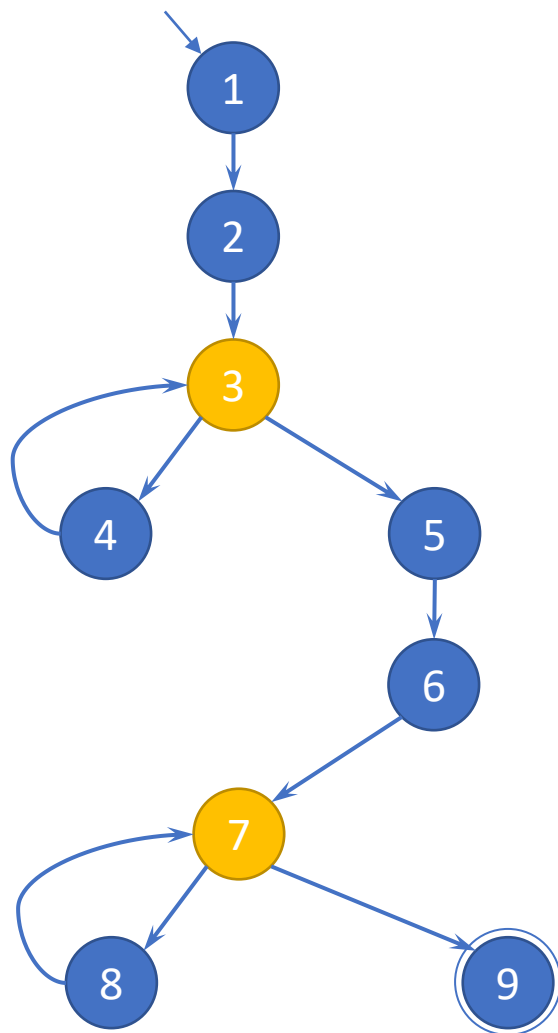
```
public static void computeStats (int[] numbers) {  
    int length = numbers.length;  
    double med, var, sd, mean, sum, varsum;  
    sum = 0;  
  
    for (int i = 0; i < length; i++) {  
        sum += numbers [i];  
    }  
  
    med = numbers[length / 2];  
    mean = sum / (double)length;  
    varsum = 0;  
  
    for (int i = 0; i < length; i++) {  
        varsum += ((numbers[i] - mean) * (numbers[i] - mean));  
    }  
    var = varsum / (length - 1.0);  
    sd = Math.sqrt(var);  
  
    System.out.println ("length:\t\t\t" + length);  
    System.out.println ("mean:\t\t\t" + mean);  
    System.out.println ("median:\t\t\t" + med);  
    System.out.println ("variance:\t\t" + var);  
    System.out.println ("standard deviation:\t" + sd);  
}
```



Einwände?



Erzeuge den Kontrollflussgraphen



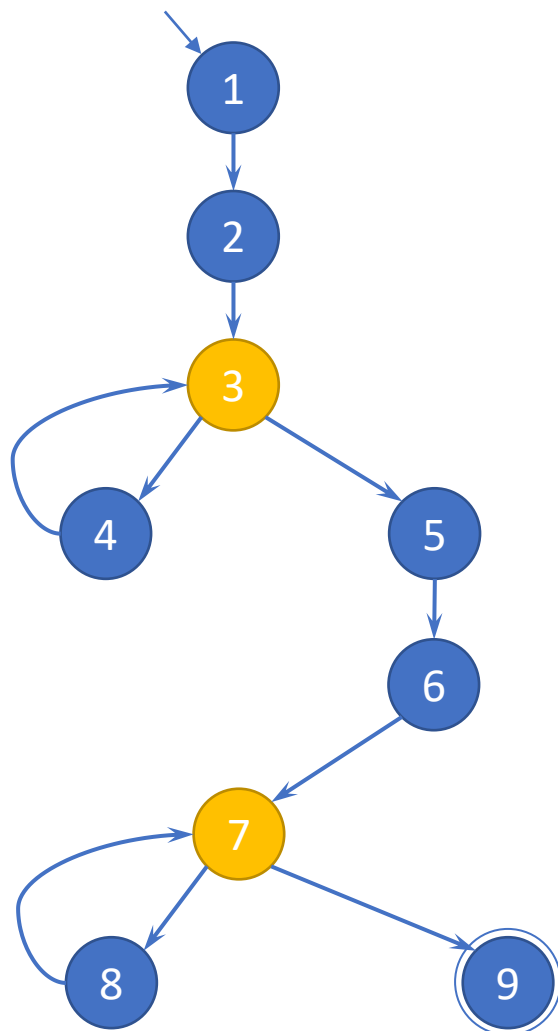
Kantenüberdeckung (EC)

$TR = \{ (1,2), (2,3), (3,4), (3,5), (4,3),$
 $(5,6), (6,7), (7,8), (7,9), (8,7) \}$

Testpfade: $\{ [1, 2, 3, 4, 3, 5, 6, 7, 8, 7, 9] \}$



Erzeuge den Kontrollflussgraphen



Kantenpaar-Überdeckung (EPC)

TR = { (a) [1,2,3], (b) [2,3,4], (c) [2,3,5], (d) [3,4,3], (e) [3,5,6],
 (f) [4,3,4], (g) [4,3,5], (h) [5,6,7], (i) [6,7,8], (j) [6,7,9],
 (k) [7,8,7], (l) [8,7,8], (m) [8,7,9] }

Testpfade: {

- i. [1,2,3,4,3,5,6,7,8,7,9], // Schleife je einmal
- ii. [1,2,3,5,6,7,9], // Schleife je keinmal
- iii. [1,2,3,4,3,4,3,5,6,7,8,7,8,7,9] // Schleife je mehrm.

}

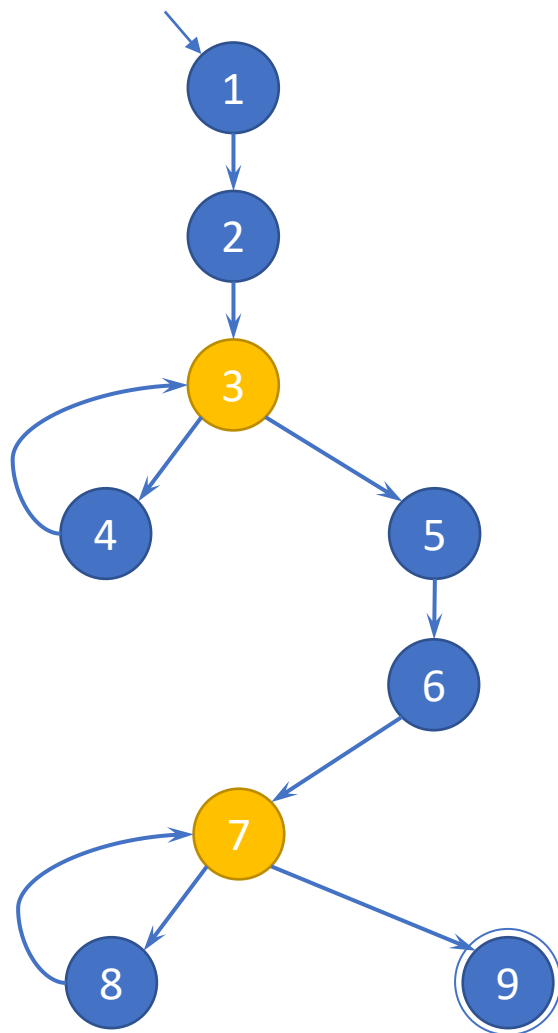
Testpfad	Getourte TRs	Abstecher
i.	(a), (b), (d), (e), (g), (h), (i), (k), (m)	(c), (j)
ii.	(a), (c), (e), (h), (j)	
iii.	(a), (b), (d), (e), (f), (g), (h), (i), (k), (l), (m)	(c), (j)

TP iii. macht TP i. überflüssig!

Eine *minimale* Menge von TPs ist günstiger



Erzeuge den Kontrollflussgraphen



Primpfad-Überdeckung (PPC)

TR = {(a) [3,4,3], (b) [4,3,4], (c) [8,7,8], (d) [8,7,9], (e) [7,8,7],
 (f) [1,2,3,4],
 (g) [4,3,5,6,7,8], (h) [4,3,5,6,7,9],
 (i) [1,2,3,5,6,7,8], (j) [1,2,3,5,6,7,9] }

Testpfade: { i. [1,2,3,4,3,5,6,7,8,7,9], // Schleife je einmal
 ii. [1,2,3,5,6,7,9], // Schleife je keinmal
 iii. [1,2,3,4,3,4,3,5,6,7,8,7,8,7,9] // Schleife je mehrm.
 iv. [1,2,3,4,3,5,6,7,9] // 1. Schleife einmal
 v. [1,2,3,5,6,7,8,7,9] // 2. Schleife einmal
 }

Testfpad	Getourte TRs	Abstecher
i.	(a), (d), (c), (f), (g)	(h), (i), (j)
ii.	(j)	
iii.	(a), (b), (c), (d), (e), (f), (g)	(h), (i), (j)
iv.	(a), (f), (h)	(j)
v.	(d), (e), (i)	(j)



Erzeuge den Kontrollflussgraphen

```
int myMethod(int a[], int k) {  
  
    // Variablen initialisieren  
    int l = 0, r=a.length-1, m, pos=-1;  
  
    // ???  
    while (l <= r && pos == -1) {  
        m = (l+r) / 2;  
        if (a[m] == k)  
            pos = m;  
        else  
            if (a[m] < k)  
                l = m+1;  
            else  
                r = m-1;  
    }  
    return pos;  
}
```

Datenfluss-Überdeckungskriterien

Die Begriffe „def“ und „use“:

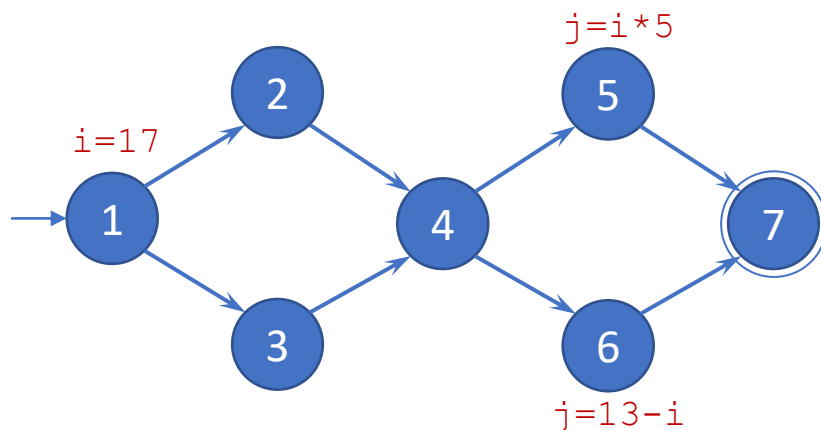
- **def**: eine Stelle, an der ein **Wert** im Speicher **gespeichert** wird
 - Variable tritt auf der **linken Seite** einer Zuweisung auf (z.B.: x=42)
 - Variable ist **aktueller Parameter** in einem Aufruf und die Methode ändert Wert
 - Variable ist ein **formaler Parameter** einer Methode (implizites def; Methodenstart)
 - Variable ist eine **Eingabe** in ein Programm
- **use**: eine Stelle, an der ein **Variablenwert genutzt** wird
 - Variable tritt auf der **rechten Seite** einer Zuweisung auf
 - Variable tritt in einer **Bedingungsprüfung** auf
 - Variable ist ein **aktueller Parameter** für eine Methode
 - Variable ist eine **Ausgabe** des Programms
 - Variable ist eine Ausgabe einer Methode in einer **return**-Anweisung
- Wenn ein def und use am **gleichen Knoten** auftreten (also in einem Basisblock), handelt es sich **nur um ein DU-Paar**, falls das **def nach dem use** auftritt und der Knoten in einer Schleife liegt

Datenfluss-Überdeckungskriterien

Die Begriffe „def“ und „use“:



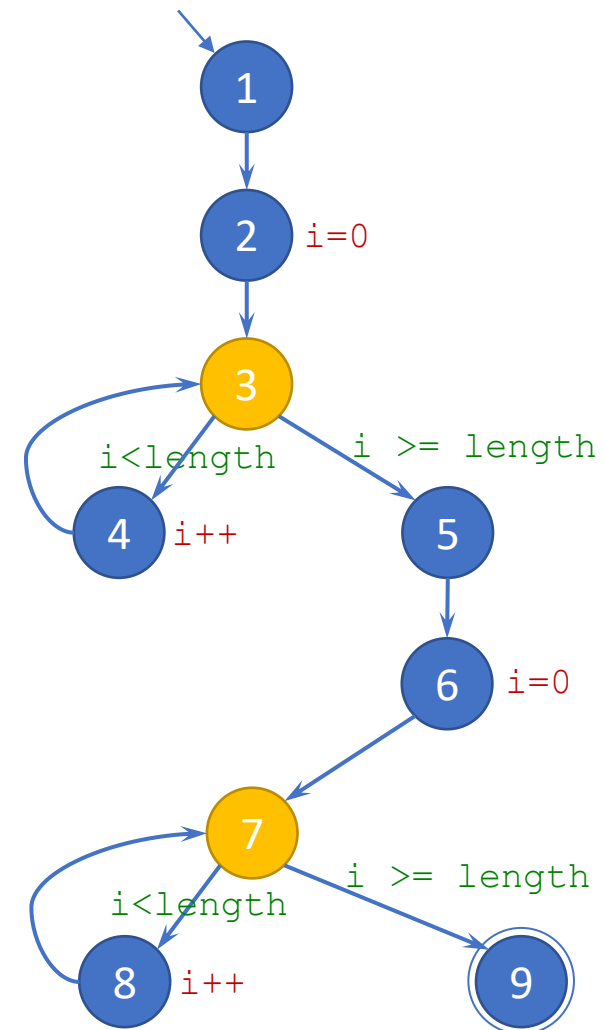
- Ziel: sicherstellen, dass Werte korrekt berechnet und genutzt werden
 - def: eine Stelle an der eine Variable in den Speicher geladen wird
 - use: eine Stelle, an der auf einen Variablenwert zugegriffen wird
- Werte in den defs sollten mindestens eine/mehrere/alle möglichen uses erreichen



Stelle	Defs	Uses
1	{ i }	--
5	{ j }	{ i }
6	{ j }	{ i }

Annotiere den Kontrollflussgraphen für computeStats

```
public static void computeStats (int[] numbers) {  
    int length = numbers.length;  
    double med, var, sd, mean, sum, varsum;  
    sum = 0; 1  
  
    for (int i = 0; i < length; i++) { 2 4  
        sum += numbers[i]; 4  
    }  
  
    med = numbers[length / 2];  
    mean = sum / (double)length; 5  
    varsum = 0;  
  
    for (int i = 0; i < length; i++) { 6 8  
        varsum += ((numbers[i] - mean) * (numbers[i] - mean)); 8  
    }  
    var = varsum / (length - 1.0);  
    sd = Math.sqrt(var);  
  
    System.out.println ("length:\t\t\t" + length); 9  
    System.out.println ("mean:\t\t\t" + mean);  
    System.out.println ("median:\t\t\t" + med);  
    System.out.println ("variance:\t\t" + var);  
    System.out.println ("standard deviation:\t" + sd);  
}
```

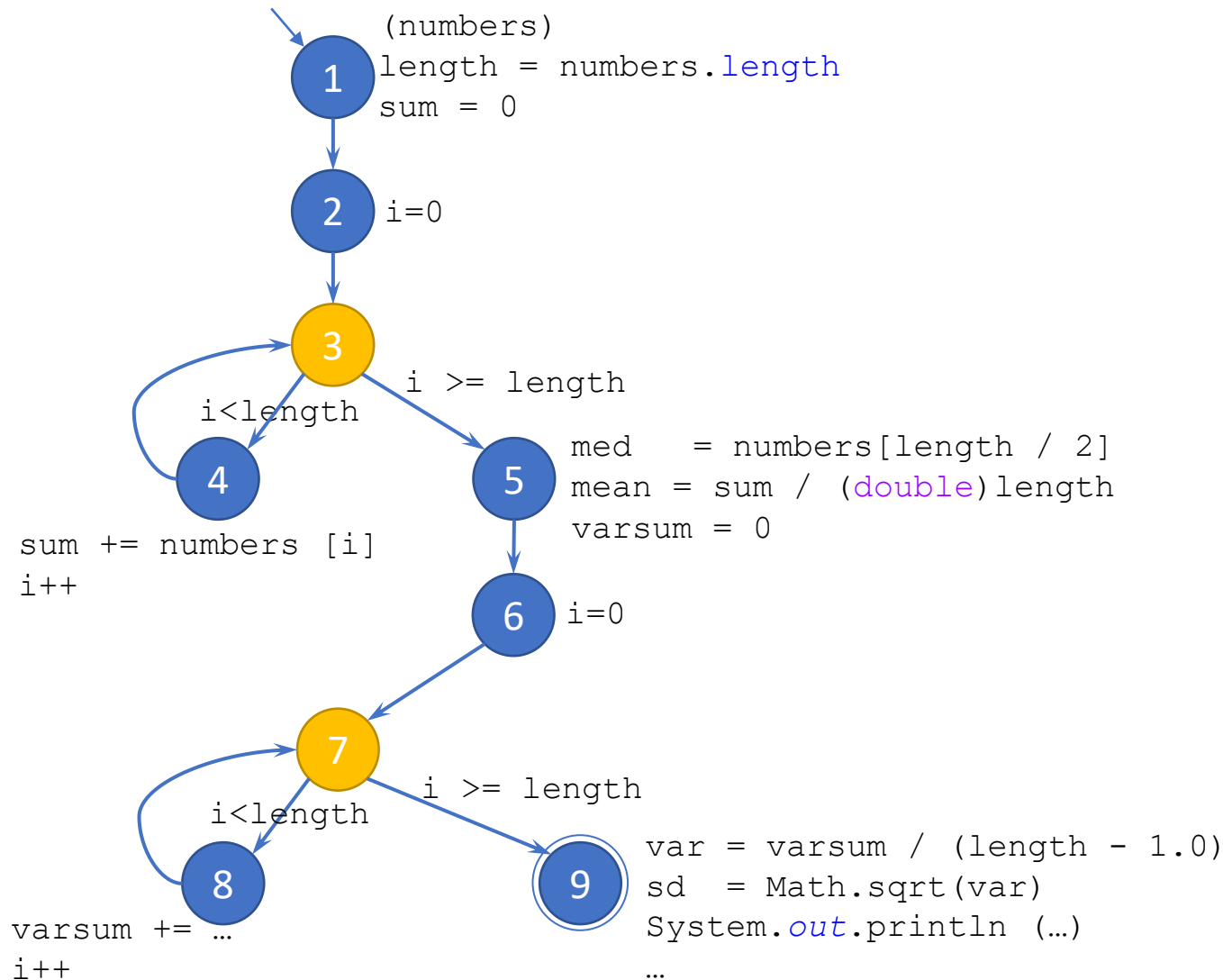


Datenfluss-Überdeckungskriterien

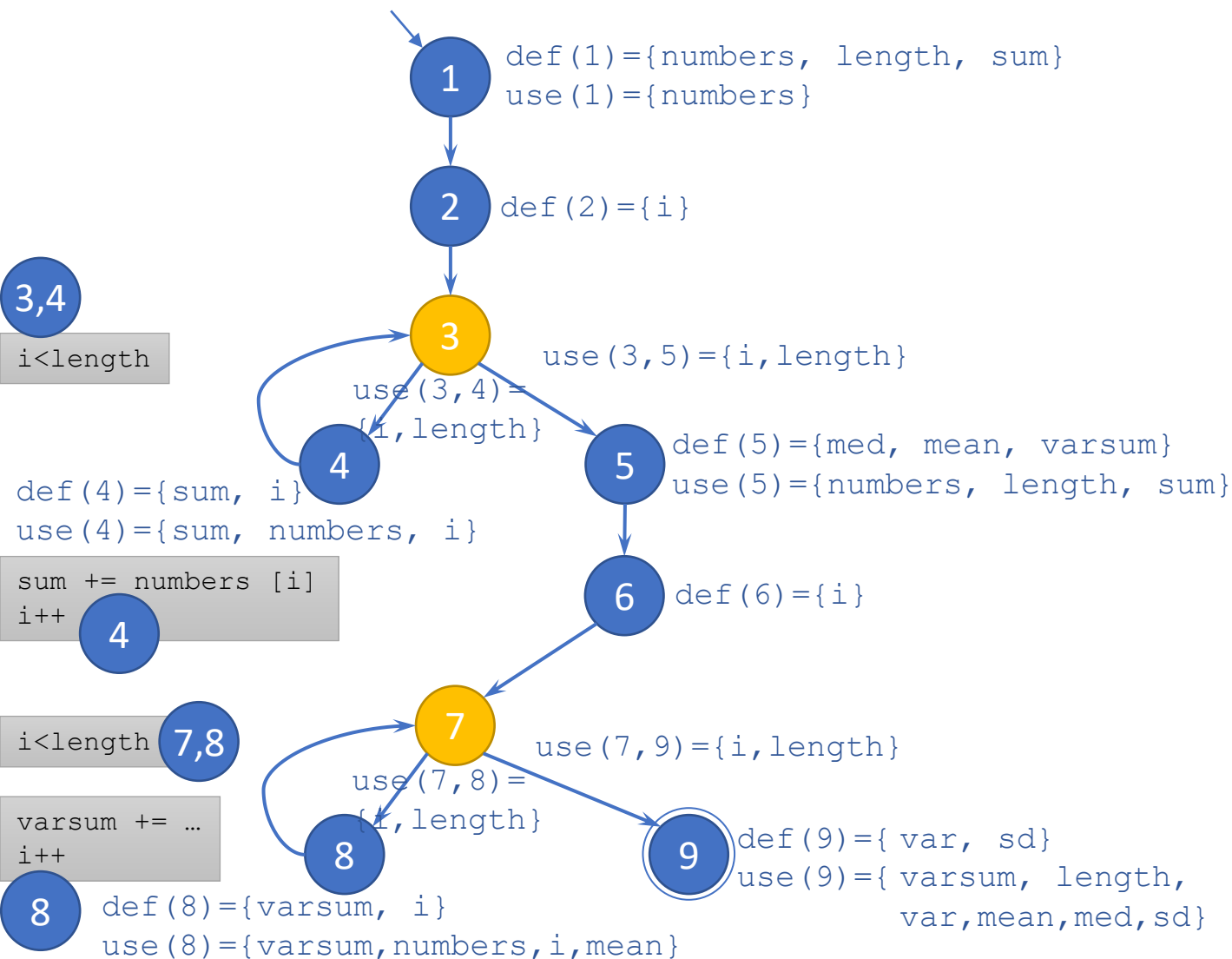
Etwas formaler: „def“ und „use“:

- **def(n), def(e):** Menge von Variablen die durch Knoten n oder Kante e definiert werden
- **use(n), use(e):** Menge von Variablen die von Knoten n oder Kante e genutzt werden
- **DU-Paar:** ein Tupel von Stellen (l_i, l_j) (Knoten oder Kanten) im CFG, bei denen eine Variable v bei l_i definiert und bei l_j genutzt wird
- **def-frei:** Ein Pfad von l_i nach l_j ist def-frei in Bezug auf Variable v , wenn v auf diesem Pfad kein weiterer Wert an irgendeinem Knoten oder einer Kante gegeben wird
- **Reichweite:** Wenn ein def-freier Pfad von l_i nach l_j in Bezug auf v existiert, dann erreicht die def von v an l_i das use von v an l_j
- **DU-Pfad:** Ein einfacher Subpfad der def-frei in Bezug auf v von einem def von v zu einem use von v gelangt
 - $du(n_i, n_j, v)$: Menge der DU-Pfade von n_i nach n_j (in Bezug auf v)
 - $du(n_i, v)$: Menge der DU-Pfade die bei n_i starten (in Bezug auf v)

Annotiere den Kontrollflussgraphen mit Code



Annotiere den Kontrollflussgraphen mit def und use



```
(numbers)
length = numbers.length
sum = 0
```

```
i=0
```

```
i >= length
```

```
med = numbers[length / 2]
mean = sum / (double)length
varsum = 0
```

```
i=0
```

```
i >= length
```

```
var = varsum / (length-1.0)
sd = Math.sqrt(var)
System.out.println (...)
...
```

def-use-Tabelle für computeStats

Knoten	def	use
1	{numbers, length, sum}	{numbers}
2	{i}	
3		
4	{sum, i}	{sum, numbers, i}
5	{med, mean, varsum}	{numbers, length, sum}
6	{i}	
7		
8	{varsum, i}	{varsum, numbers, i, mean}
9	{var, sd}	{varsum, length, var, mean, med, sd}

Kante	use
(1,2)	
(2,3)	
(3,4)	{i, length}
(4,3)	
(3,5)	{i, length}
(5,6)	
(6,7)	
(7,8)	{i, length}
(8,7)	
(7,9)	{i, length}

Bestimmung der DU-Paare für computeStats

Variable	DU-Paare
numbers	(1,4), (1,5), (1,8)
length	(1,5), (1,9), (1, (3,4)), (1, (3,5)), (1, (7,8)), (1, (7,9))
med	(5,9)
var	(9,9)
sd	(9,9)
mean	(5,8), (5,9)
sum	(1,4), (1,5), (4,4), (4,5)
varsum	(5,8), (5,9), (8,8), (8,9)
i	(2,4), (2, (3,4)), (2, (3,5)), (2,8) , (2, (7,8)) , (2, (7,9)) (4,4), (4, (3,4)), (4, (3,5)), (4,8) , (4, (7,8)) , (4, (7,9)) (5,7), (5, (7,8)), (5, (7,9)) (8,8), (8, (7,8)), (8, (7,9))

defs kommen vor uses,
Nicht als DU-Paare zu werten

```
var = varsum / (length - 1.0)
sd = Math.sqrt(var)
```









defs kommen nach uses,
Als DU-Paare zu werten (s. F. 32)









```
sum += numbers[i]
```

Kein def-freier Pfad ...
Scope von i unterschiedlich

Rest ok, da kein Pfad von (6 oder 8) nach (3 oder 4) existiert

Bestimmung der DU-Pfade für computeStats

Variable	DU-Paare	DU-Pfade
numbers	(1,4)	[1,2,3,4] 
	(1,5)	[1,2,3,5] 
	(1,8)	[1,2,3,5,6,7,8] 
length	(1,5)	[1,2,3,5]
	(1,9)	[1,2,3,5,6,7,9] 
	(1, (3,4))	[1,2,3,4]
	(1, (3,5))	[1,2,3,5]
	(1, (7,8))	[1,2,3,5,6,7,8]
	(1, (7,9))	[1,2,3,5,6,7,9]
med	(5,9)	[5,6,7,9] 
var	(9,9)	Kein Pfad notwendig
sd	(9,9)	Kein Pfad notwendig
mean	(5,8)	[5,6,7,8] 
	(5,9)	[5,6,7,9]
sum	(1,4)	[1,2,3,4]
	(1,5)	[1,2,3,5]
	(4,4)	[4,3,4] 
	(4,5)	[4,3,5] 

Variable	DU-Paare	DU-Pfade
varsum	(5,8)	[5,6,7,8]
	(5,9)	[5,6,7,9]
	(8,8)	[8,7,8] 
i	(8,9)	[8,7,9] 
	(2,4)	[2,3,4] 
	(2,(3,4))	[2,3,4]
	(2,(3,5))	[2,3,5] 
	(4,4)	[4,3,4] 
	(4,(3,4))	[4,3,4]
	(4,(3,5))	[4,3,5]
	(6,8)	[6,7,8] 
	(6,(7,8))	[6,7,8]
	(6,(7,9))	[6,7,9] 
	(8,8)	[8,7,8] 
	(8,(7,8))	[8,7,8]
	(8,(7,9))	[8,7,9]

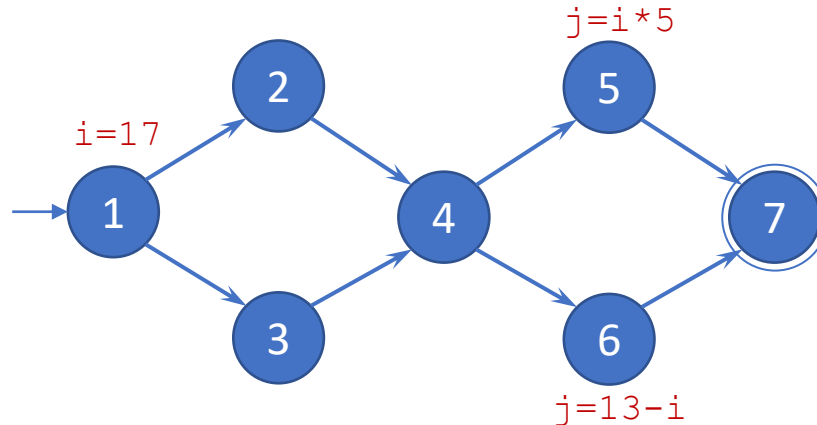
 : kein Betreten einer Schleife;
  : mindestens eine Schleifeniteration notwendig;
  : mind. Zwei Schleifeniterationen notwendig

Touren von DU-Pfaden und Datenfluss-Überdeckungskriterien

- Ein Testpfad p **DU-tourt** einen Subpfad q in Bezug auf v , wenn:
 - p tourt Subpfad q und
 - Der gewählte Subpfad ist def-frei in Bezug auf v
- Auch hier können Sidetrips wie vorher verwendet werden
- Es können drei Kriterien genutzt werden (vgl. F. 19 unten links & F. 35):
 - **Nutze jedes def (All-Defs-Coverage: ADC)**
 - **Bedeutung:** Stelle sicher, dass jedes def ein use erreicht
 - **Formal:** Für jede Menge an DU-Pfaden $S = du(n, v)$, TR beinhaltet mindestens einen Pfad d in S
 - **Gelange zu jedem use (All-Uses-Coverage: AUC)**
 - **Bedeutung:** Stelle sicher, dass jedes def alle möglichen uses erreicht
 - **Formal:** Für jede Menge von DU-Pfaden nach uses $S = du(n_i, n_j, v)$, TR beinhaltet mindestens einen Pfad d in S
 - **Verfolge alle DU-Pfade (All-DU-Paths-Coverage: ADUPC)**
 - **Bedeutung:** Decke alle Pfade zwischen defs und uses ab
 - **Formal:** Für jede Menge $S = du(n_i, n_j, v)$, TR beinhaltet jeden Pfad d aus S

Datenfluss-Überdeckungskriterien

ADC, AUC, ADUPC



Stelle sicher, dass jedes def ein use erreicht

ADC für Variable i

[1,2,4,5]

Stelle sicher, dass jedes def alle möglichen uses erreicht

AUC für Variable i

[1,2,4,5]

[1,2,4,6]

Decke alle Pfade zwischen defs und uses ab

ADUPC für Variable i

[1,2,4,5]

[1,2,4,6]

[1,3,4,5]

[1,3,4,6]

Wo ist der Fehler?



DU-Pfade

[1,2,3,4]
[1,2,3,5]
[1,2,3,5,6,7,8]
[1,2,3,5,6,7,9]

[2,3,4]
[2,3,5]

[4,3,4]
[4,3,5]

[5,6,7,8]
[5,6,7,9]

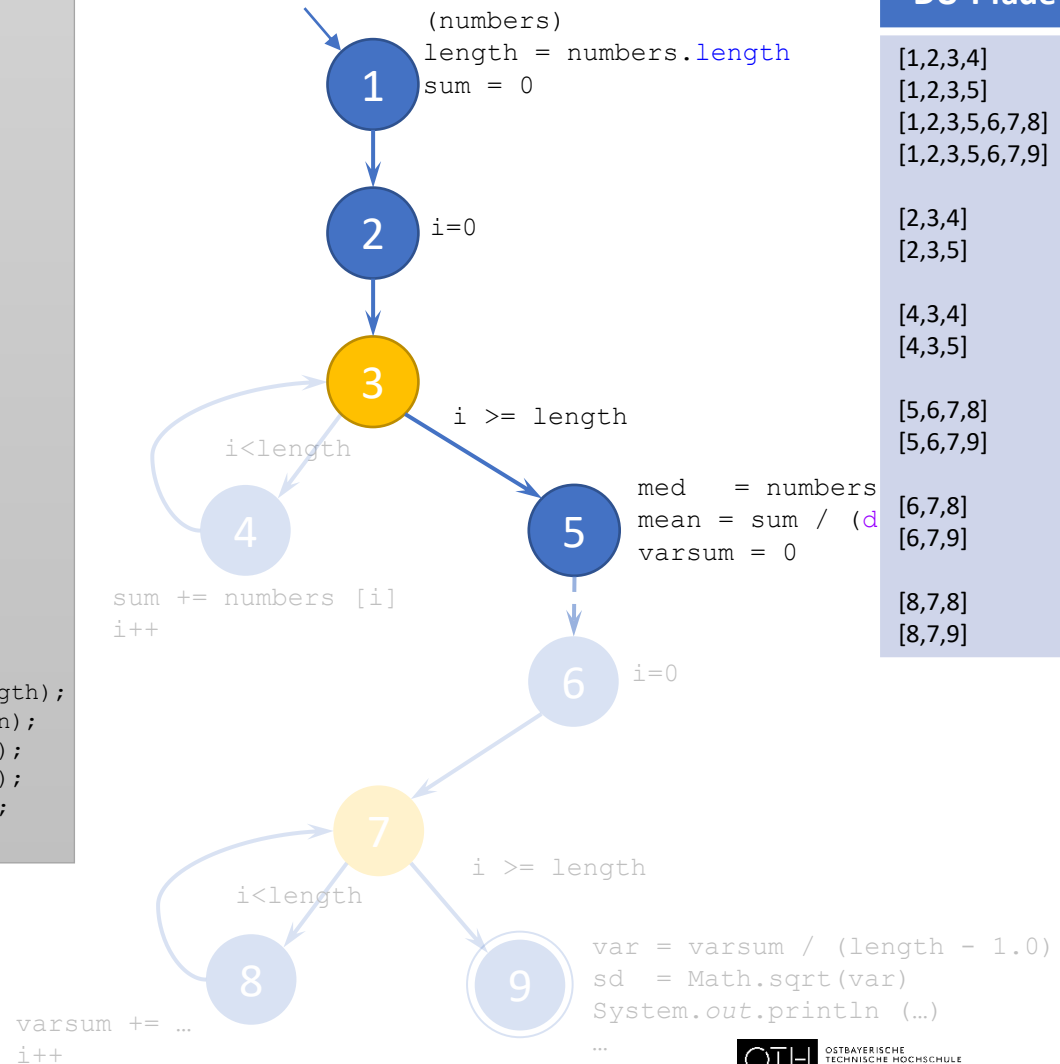
[6,7,8]
[6,7,9]

[8,7,8]
[8,7,9]

```
1 public static void computeStats (int[] numbers) {  
2     int length = numbers.length;  
3     double med, var, sd, mean, sum, varsum;  
4     sum = 0;  
5     for (int i = 0; i < length; i++) {  
6         sum += numbers [i];  
7     }  
8     med = numbers[length / 2];  
9     mean = sum / (double)length;  
10    varsum = 0;  
11    for (int i = 0; i < length; i++) {  
12        varsum += ((numbers[i] - mean)  
13                    * (numbers[i] - mean));  
14    }  
15    var = varsum / (length - 1.0);  
16    sd = Math.sqrt(var);  
17  
18    System.out.println ("length:\t\t\t" + length);  
19    System.out.println ("mean:\t\t\t" + mean);  
20    System.out.println ("median:\t\t\t" + med);  
21    System.out.println ("variance:\t\t" + var);  
22    System.out.println ("standard deviation:\t" + sd);  
23 }
```

⇒ Mithilfe der DU-Pfade haben wir einen Fehler entdeckt!

⇒ Testpfad: [1,2,3,5,6,7,9]



Literatur

- P. Ammann, J. Offutt, *“Introduction to Software Testing”*, Cambridge University Press, 2. Auflage, 2018

