Software-Test: Strukturtest

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken







Welche Testfälle auswählen?

Ich kann nur eine beschränkte Zahl von Läufen testen – welche soll ich wählen?

Funktionale Verfahren Auswahl nach *Eigenschaften der Eingabe*

Strukturtests Auswahl nach *Aufbau des Programms*

Ziel im Strukturtest: hohen *Überdeckungsgrad* erreichen
⇒ Testfälle sollen möglichst viele Aspekte der Programmstruktur abdecken (Kontrollfluß, Datenfluß)







Anwendung: Zeichen zählen

Das Programm `zaehlezchn` soll

- Zeichen von der Tastatur einlesen, bis ein Zeichen erkannt wird, das kein Großbuchstabe ist
- Die Zahl der eingelesenen Zeichen und Vokale ausgeben

```
$
```







Anwendung: Zeichen zählen

Das Programm `zaehlezchn` soll

- Zeichen von der Tastatur einlesen, bis ein Zeichen erkannt wird, das kein Großbuchstabe ist
- Die Zahl der eingelesenen Zeichen und Vokale ausgeben

```
$ zaehlezchn
```







Anwendung: Zeichen zählen

Das Programm `zaehlezchn` soll

- Zeichen von der Tastatur einlesen, bis ein Zeichen erkannt wird, das kein Großbuchstabe ist
- Die Zahl der eingelesenen Zeichen und Vokale ausgeben

```
$ zaehlezchn
Bitte Zeichen eingeben:
```







Anwendung: Zeichen zählen

Das Programm `zaehlezchn` soll

- Zeichen von der Tastatur einlesen, bis ein Zeichen erkannt wird, das kein Großbuchstabe ist
- Die Zahl der eingelesenen Zeichen und Vokale ausgeben

```
$ zaehlezchn
Bitte Zeichen eingeben: HALLELUJA!
Anzahl Vokale:  4
Anzahl Zeichen: 9
$ _
```







Zeichen zählen – Benutzung


```
#include <iostream>
#include <limits.h>

void ZaehleZchn(int& VokalAnzahl, int& Gesamtzahl);

int main()
{
    int AnzahlVokale = 0;
    int AnzahlZchn   = 0;
    cout << "Bitte Zeichen eingeben: ";
    ZaehleZchn(AnzahlVokale, AnzahlZchn);
    cout << "Anzahl Vokale:  " << AnzahlVokale << endl;
    cout << "Anzahl Zeichen: " << AnzahlZchn << endl;
}
```

Zeichen zählen – Realisierung

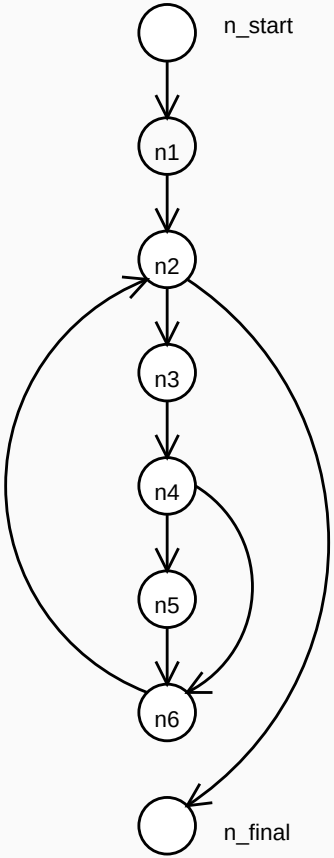

```
void ZaehleZchn(int& VokalAnzahl, int& Gesamtzahl)
{
    char Zchn;
    cin >> Zchn;
    while (Zchn >= 'A' && Zchn <= 'Z' &&
           Gesamtzahl < INT_MAX)
    {
        Gesamtzahl = Gesamtzahl + 1;
        if (Zchn == 'A' || Zchn == 'E' || Zchn == 'I' ||
            Zchn == 'O' || Zchn == 'U')
        {
            VokalAnzahl = VokalAnzahl + 1;
        }
        cin >> Zchn;
    }
}
```







Kontrollflußgraph




```
cin >> Zchn;

while (Zchn >= 'A' && Zchn <= 'Z' &&
       Gesamtzahl < INT_MAX)

Gesamtzahl = Gesamtzahl + 1;

if (Zchn == 'A' || Zchn == 'E' ||
    Zchn == 'I' || Zchn == 'O' ||
    Zchn == 'U')

VokalAnzahl = VokalAnzahl + 1;

cin >> Zchn;
```

Einfache Überdeckungsmaße

Eine Menge von Testfällen kann folgende Kriterien erfüllen:

Anweisungsüberdeckung Jeder Knoten im Kontrollflußgraph muß einmal durchlaufen werden (= jede Anweisung wird wenigstens einmal ausgeführt)

Zweigüberdeckung Jede Kante im Kontrollflußgraph muß einmal durchlaufen werden; schließt Anweisungsüberdeckung ein

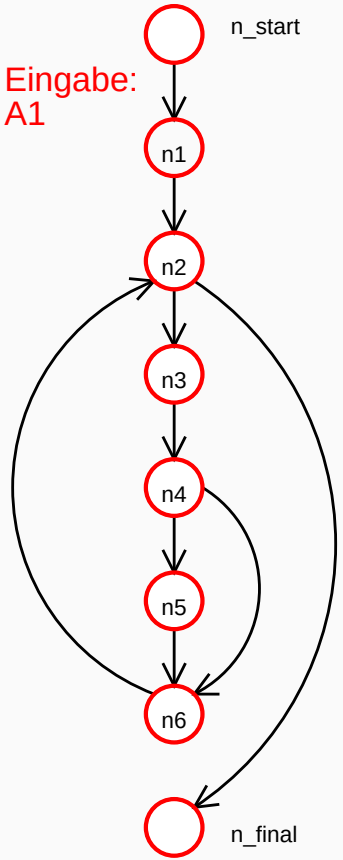
Pfadüberdeckung Jeder *Pfad* im Kontrollflußgraphen muß einmal durchlaufen werden





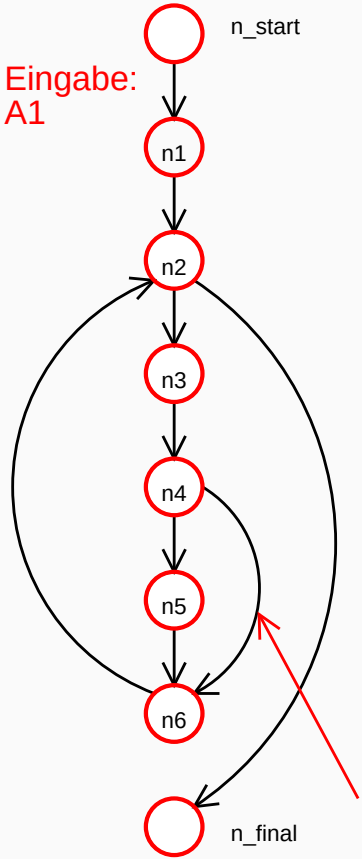


Anweisungsüberdeckung (C_0)



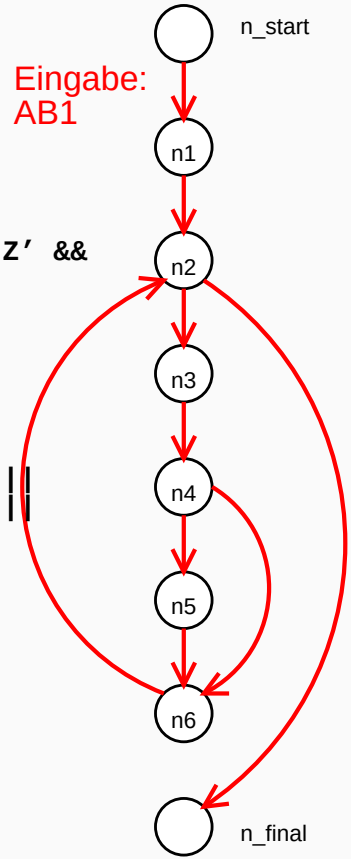

```
cin >> Zchn;


while (Zchn >= 'A' && Zchn <= 'Z' &&
       Gesamtzahl < INT_MAX)

Gesamtzahl = Gesamtzahl + 1;


if (Zchn == 'A' || Zchn == 'E' ||
    Zchn == 'I' || Zchn == 'O' ||
    Zchn == 'U')

VokalAnzahl = VokalAnzahl + 1;


cin >> Zchn;
```

Zweigüberdeckung (C_1)




```
cin >> Zchn;

while (Zchn >= 'A' && Zchn <= 'Z' &&
       Gesamtzahl < INT_MAX)

Gesamtzahl = Gesamtzahl + 1;

if (Zchn == 'A' || Zchn == 'E' ||
    Zchn == 'I' || Zchn == 'O' ||
    Zchn == 'U')

VokalAnzahl = VokalAnzahl + 1;

cin >> Zchn;
```

Zweig (n4, n6)
wird nicht notwendig ausgeführt



Anweisungs- und Zweigüberdeckung

Anweisungsüberdeckung

- Notwendiges, aber nicht hinreichendes Testkriterium
- Kann Code finden, der nicht ausführbar ist
- Als eigenständiges Testverfahren nicht geeignet
- Fehleridentifizierungsquote: 18%







Anweisungs- und Zweigüberdeckung

Anweisungsüberdeckung

- Notwendiges, aber nicht hinreichendes Testkriterium
- Kann Code finden, der nicht ausführbar ist
- Als eigenständiges Testverfahren nicht geeignet
- Fehleridentifizierungsquote: 18%

Zweigüberdeckung

- gilt als *das* minimale Testkriterium
- kann nicht ausführbare Programmzweige finden
- kann häufig durchlaufene Programmzweige finden (Optimierung)
- Fehleridentifikationsquote: 34%

Schwächen der Anweisungüberdeckung

Warum reicht Anweisungsüberdeckung nicht aus?

Wir betrachten den folgenden Code:

```
x = 1;
if (x >= 1)          // statt y >= 1
    x = x + 1;
```

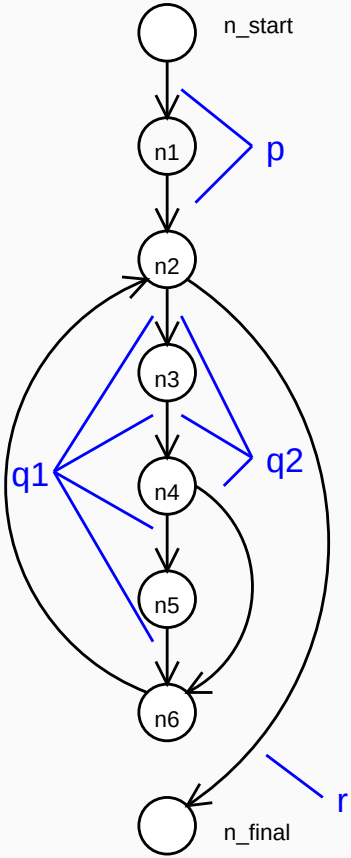
Hier wird zwar jede Anweisung einmal ausgeführt; die Zweigüberdeckung fordert aber auch die Suche nach einer Alternative (was hier schwerfällt).





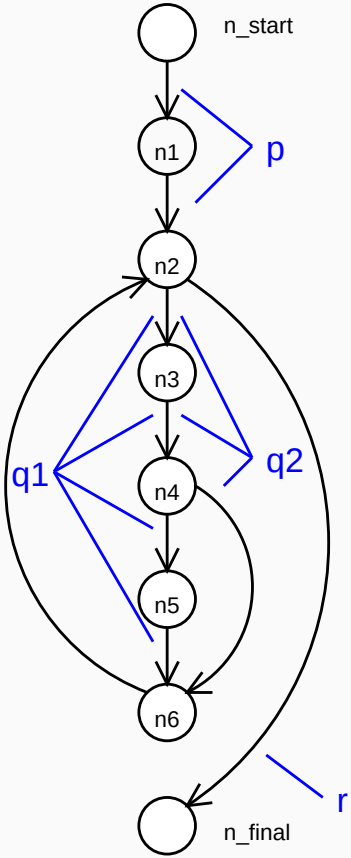


Pfadüberdeckung




```
cin >> Zchn;

while (Zchn >= 'A' && Zchn <= 'Z' &&
       Gesamtzahl < INT_MAX)

Gesamtzahl = Gesamtzahl + 1;

if (Zchn == 'A' || Zchn == 'E' ||
    Zchn == 'I' || Zchn == 'O' ||
    Zchn == 'U')

VokalAnzahl = VokalAnzahl + 1;

cin >> Zchn;
```

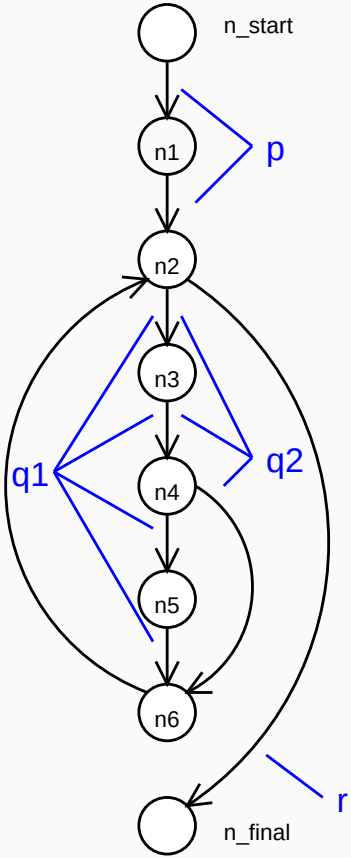
Pfadüberdeckung




```
cin >> Zchn;

while (Zchn >= 'A' && Zchn <= 'Z' &&
       Gesamtzahl < INT_MAX)

Gesamtzahl = Gesamtzahl + 1;

if (Zchn == 'A' || Zchn == 'E' ||
    Zchn == 'I' || Zchn == 'O' ||
    Zchn == 'U')

VokalAnzahl = VokalAnzahl + 1;

cin >> Zchn;
```

Pfade $P = p(q_1|q_2)^*r$

Pfadüberdeckung




```
cin >> Zchn;

while (Zchn >= 'A' && Zchn <= 'Z' &&
       Gesamtzahl < INT_MAX)

Gesamtzahl = Gesamtzahl + 1;

if (Zchn == 'A' || Zchn == 'E' ||
    Zchn == 'I' || Zchn == 'O' ||
    Zchn == 'U')

VokalAnzahl = VokalAnzahl + 1;

cin >> Zchn;
```

Pfade $P = p(q_1|q_2)^*r$

$\Rightarrow |P| = 2^{INT\_MAX} - 1$

Pfadüberdeckung (2)

Pfadüberdeckung

- mächtigstes kontrollstrukturorientiertes Testverfahren
- Höchste Fehleridentifizierungsquote
- keine praktische Bedeutung,
  da Durchführbarkeit sehr eingeschränkt







Strukturierte Verfahren

Der *Boundary Interior-Pfadtest* ist eine schwächere Version des Pfadüberdeckungstests.

Idee: Beim Test von Schleifen wird auf die Überprüfung von Pfaden verzichtet, die durch mehr als einmalige Schleifenwiederholung erzeugt werden.







Strukturierte Verfahren

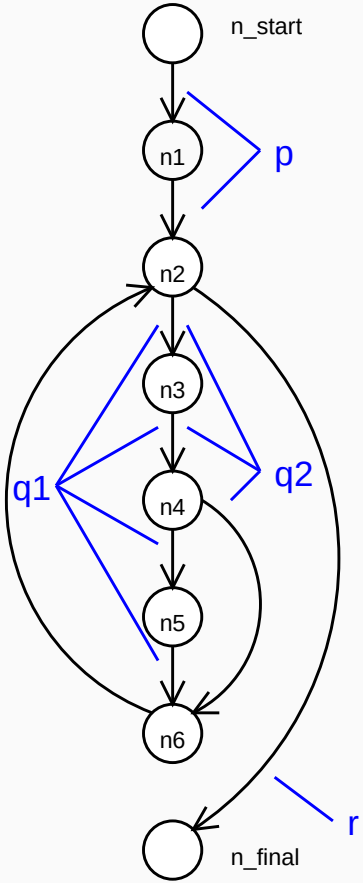
Der *Boundary Interior-Pfadtest* ist eine schwächere Version des Pfadüberdeckungstests.

Idee: Beim Test von Schleifen wird auf die Überprüfung von Pfaden verzichtet, die durch mehr als einmalige Schleifenwiederholung erzeugt werden.

Verallgemeinerung: *Strukturierter Pfadtest* – Innerste Schleifen werden maximal k-mal ausgeführt

- Erlaubt die gezielte Überprüfung von Schleifen
- Überprüft zusätzlich Zweigkombinationen
- Im Gegensatz zum Pfadüberdeckungstest praktikabel
- Fehleridentifikationsquote: um 65% (Strukturierter Pfadtest)

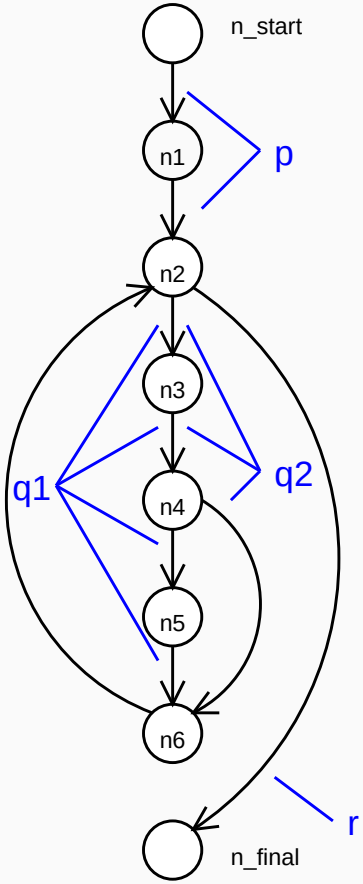
Boundary Interior-Pfadtest



Pfade außerhalb Schleife

1a) `Gesamtzahl = INT_MAX` pr

Boundary Interior-Pfadtest



Pfade außerhalb Schleife

1a) `Gesamtzahl = INT_MAX` pr

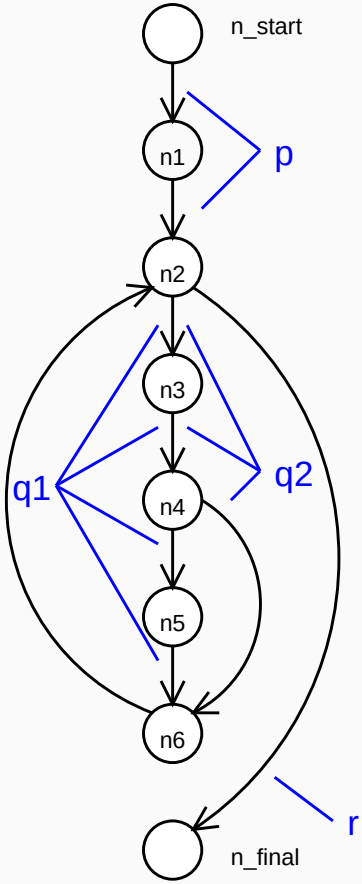
Boundary tests:

Pfade, die Schleife betreten, jedoch nicht wiederholen

2a) Zchn = 'A', '1' pq_1r

2b) Zchn = 'B', '1' pq_2r

Boundary Interior-Pfadtest



Pfade außerhalb Schleife

1a) `Gesamtzahl = INT_MAX` pr

Boundary tests:

Pfade, die Schleife betreten, jedoch nicht wiederholen

2a) Zchn = 'A', '1' pq_1r

2b) Zchn = 'B', '1' pq_2r

Interior tests:

Pfade, die mindestens eine Schleifenwiederholung enthalten

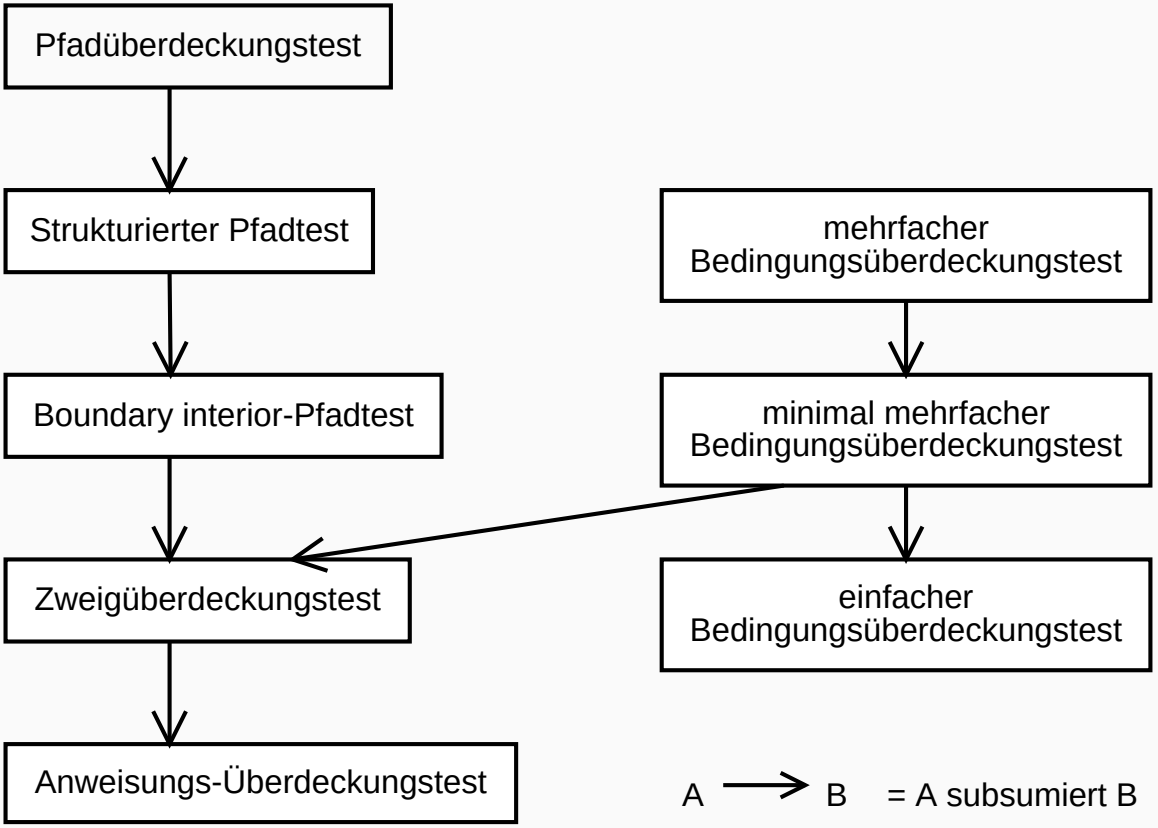
3a) Zchn = 'E', 'I', 'N', '*' pq_1q_1r

3b) Zchn = 'A', 'H', '!' pq_1q_2r

3c) Zchn = 'H', 'A', '+' pq_2q_1r

3d) Zchn = 'X', 'X', ',' pq_2q_2r

Verfahren im Überblick



Bedingungsüberdeckung

Wir betrachten die Bedingungen aus ZaehleZchn:

```
while (Zchn >= 'A' && Zchn <= 'Z' &&
       Gesamtzahl < INT_MAX)                    (A)

if (Zchn == 'A' || Zchn == 'E' || Zchn == 'I' ||
    Zchn == 'O' || Zchn == 'U')                 (B)
```

Die *Struktur der Bedingungen* wird vom Zweigüberdeckungstest nicht geeignet beachtet

⇒ *Bedingungsüberdeckungstests*

Einfache Bedingungsüberdeckung

Ziel: Jede atomare Bedingung muß wenigstens einmal `true` und `false` sein.

Beispiel: In der Bedingung (B)

```
if (Zchn == 'A' || Zchn == 'E' || Zchn == 'I' ||
    Zchn == 'O' || Zchn == 'U')
```

sorgt der Testfall

Zchn = 'A', 'E', 'I', 'O', 'U'

dafür, daß jede atomare Bedingung einmal `true` und wenigstens einmal `false` wird.

Mehrfach-Bedingungsüberdeckung

Die einfache Bedingungsüberdeckung

- Schließt weder Anweisungsüberdeckung noch Zweigüberdeckung ein
- $\Rightarrow$ als alleinige Anforderung nicht ausreichend

Ziel der *Mehrfach-Bedingungsüberdeckung*: *alle Variationen* der atomaren Bedingungen bilden!

Das klappt aber nicht immer – z.B. kann nicht gleichzeitig `Zchn == 'A'` und `Zchn == 'E'` gelten.







Minimale Mehrfach-Bedingungsüberdeckung

Wie einfache Bedingungsüberdeckung, jedoch:

Die Gesamt-Bedingung *muß wenigstens einmal* `true` *und wenigstens einmal* `false` *werden.*

- Schließt Zweigüberdeckung (und somit Anweisungsüberdeckung) ein
- $\Rightarrow$ realistischer Kompromiß







Beispiel

Jede *atomare Bedingung* und jede *Gesamt-Bedingung* muß wenigstens einmal `true` und wenigstens einmal `false` werden.

Testfall
`Gesamtzahl` `Zchn`

`Zchn >= 'A'`
`Zchn <= 'Z'`
`Gesamtzahl < INT_MAX`
Bedingung (A)

`Zchn == 'A'`
`Zchn == 'E'`
`Zchn == 'I'`
`Zchn == 'O'`
`Zchn == 'U'`
Bedingung (B)

Beispiel

Jede *atomare Bedingung* und jede *Gesamt-Bedingung* muß wenigstens einmal `true` und wenigstens einmal `false` werden.

Testfall	1
`Gesamtzahl`	0
`Zchn`	'A'
`Zchn >= 'A'`	**T**
`Zchn <= 'Z'`	**T**
`Gesamtzahl < INT_MAX`	**T**
Bedingung (A)	**T**
`Zchn == 'A'`	**T**
`Zchn == 'E'`	**F**
`Zchn == 'I'`	**F**
`Zchn == 'O'`	**F**
`Zchn == 'U'`	**F**
Bedingung (B)	**T**

Beispiel

Jede *atomare Bedingung* und jede *Gesamt-Bedingung* muß wenigstens einmal `true` und wenigstens einmal `false` werden.

Testfall	1	
`Gesamtzahl`	0	1
`Zchn`	'A'	'E'
`Zchn >= 'A'`	**T**	T
`Zchn <= 'Z'`	**T**	T
`Gesamtzahl < INT_MAX`	**T**	T
Bedingung (A)	**T**	T
`Zchn == 'A'`	**T**	**F**
`Zchn == 'E'`	**F**	**T**
`Zchn == 'I'`	**F**	F
`Zchn == 'O'`	**F**	F
`Zchn == 'U'`	**F**	F
Bedingung (B)	**T**	T

Beispiel

Jede *atomare Bedingung* und jede *Gesamt-Bedingung* muß wenigstens einmal `true` und wenigstens einmal `false` werden.

Testfall	1				
`Gesamtzahl`	0	1	2	3	4
`Zchn`	'A'	'E'	'I'	'O'	'U'
`Zchn >= 'A'`	**T**	T	T	T	T
`Zchn <= 'Z'`	**T**	T	T	T	T
`Gesamtzahl < INT_MAX`	**T**	T	T	T	T
Bedingung (A)	**T**	T	T	T	T
`Zchn == 'A'`	**T**	**F**	F	F	F
`Zchn == 'E'`	**F**	**T**	F	F	F
`Zchn == 'I'`	**F**	F	**T**	F	F
`Zchn == 'O'`	**F**	F	F	**T**	F
`Zchn == 'U'`	**F**	F	F	F	**T**
Bedingung (B)	**T**	T	T	T	T

Beispiel

Jede *atomare Bedingung* und jede *Gesamt-Bedingung* muß wenigstens einmal `true` und wenigstens einmal `false` werden.

Testfall	1					
`Gesamtzahl`	0	1	2	3	4	5
`Zchn`	'A'	'E'	'I'	'O'	'U'	'B'
`Zchn >= 'A'`	**T**	T	T	T	T	T
`Zchn <= 'Z'`	**T**	T	T	T	T	T
`Gesamtzahl < INT_MAX`	**T**	T	T	T	T	T
Bedingung (A)	**T**	T	T	T	T	T
`Zchn == 'A'`	**T**	**F**	F	F	F	F
`Zchn == 'E'`	**F**	**T**	F	F	F	F
`Zchn == 'I'`	**F**	F	**T**	F	F	F
`Zchn == 'O'`	**F**	F	F	**T**	F	F
`Zchn == 'U'`	**F**	F	F	F	**T**	F
Bedingung (B)	**T**	T	T	T	T	**F**

Beispiel

Jede *atomare Bedingung* und jede *Gesamt-Bedingung* muß wenigstens einmal `true` und wenigstens einmal `false` werden.

Testfall	1						
Gesamtzahl	0	1	2	3	4	5	6
Zchn	'A'	'E'	'I'	'O'	'U'	'B'	'1'
Zchn >= 'A'	**T**	T	T	T	T	T	**F**
Zchn <= 'Z'	**T**	T	T	T	T	T	T
Gesamtzahl < INT_MAX	**T**	T	T	T	T	T	T
Bedingung (A)	**T**	T	T	T	T	T	**F**
Zchn == 'A'	**T**	**F**	F	F	F	F	-
Zchn == 'E'	**F**	**T**	F	F	F	F	-
Zchn == 'I'	**F**	F	**T**	F	F	F	-
Zchn == 'O'	**F**	F	F	**T**	F	F	-
Zchn == 'U'	**F**	F	F	F	**T**	F	-
Bedingung (B)	**T**	T	T	T	T	**F**	-

Beispiel

Jede *atomare Bedingung* und jede *Gesamt-Bedingung* muß wenigstens einmal `true` und wenigstens einmal `false` werden.

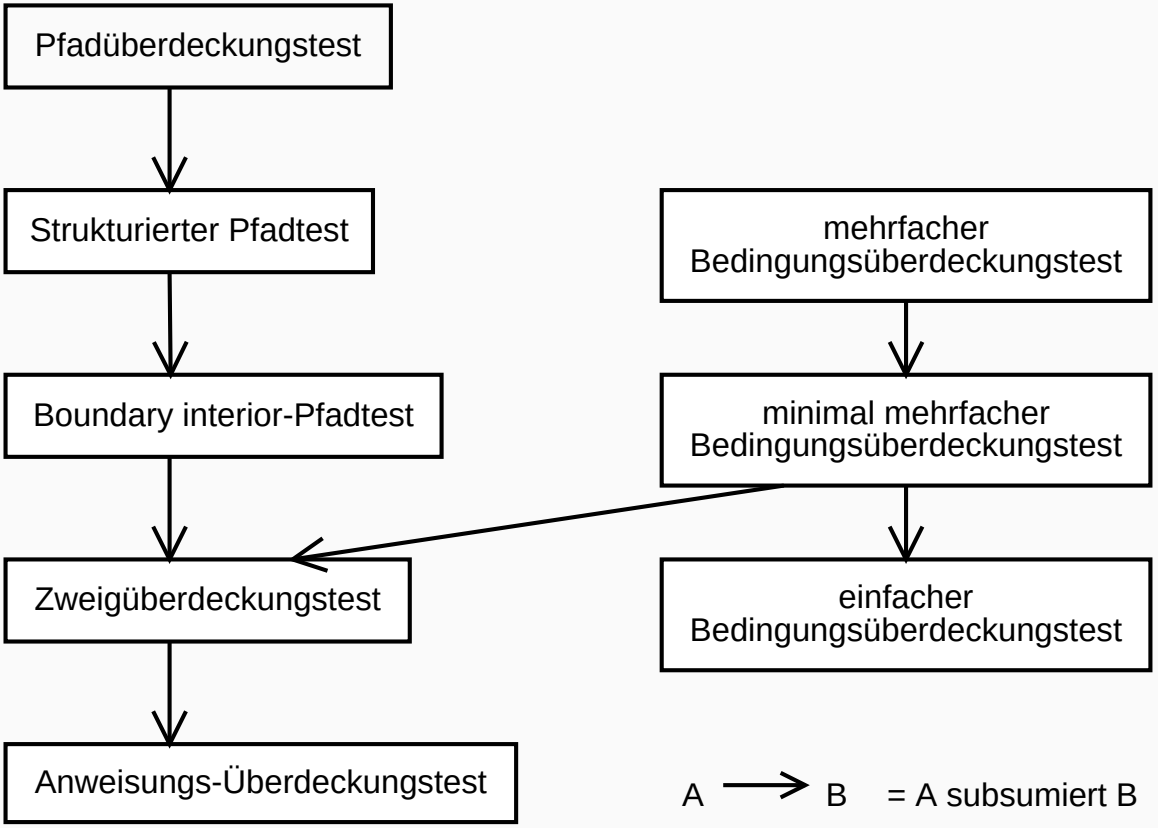
Testfall	1							2
`Gesamtzahl`	0	1	2	3	4	5	6	0
`Zchn`	'A'	'E'	'I'	'O'	'U'	'B'	'1'	'a'
`Zchn >= 'A'`	**T**	T	T	T	T	T	**F**	T
`Zchn <= 'Z'`	**T**	T	T	T	T	T	T	**F**
`Gesamtzahl < INT_MAX`	**T**	T	T	T	T	T	T	T
Bedingung (A)	**T**	T	T	T	T	T	**F**	F
`Zchn == 'A'`	**T**	**F**	F	F	F	F	-	-
`Zchn == 'E'`	**F**	**T**	F	F	F	F	-	-
`Zchn == 'I'`	**F**	F	**T**	F	F	F	-	-
`Zchn == 'O'`	**F**	F	F	**T**	F	F	-	-
`Zchn == 'U'`	**F**	F	F	F	**T**	F	-	-
Bedingung (B)	**T**	T	T	T	T	**F**	-	-

Beispiel

Jede *atomare Bedingung* und jede *Gesamt-Bedingung* muß wenigstens einmal `true` und wenigstens einmal `false` werden.

Testfall	1							2	3
`Gesamtzahl`	0	1	2	3	4	5	6	0	INT_MAX
`Zchn`	'A'	'E'	'I'	'O'	'U'	'B'	'1'	'a'	'D'
`Zchn >= 'A'`	**T**	T	T	T	T	T	**F**	T	T
`Zchn <= 'Z'`	**T**	T	T	T	T	T	T	**F**	T
`Gesamtzahl < INT_MAX`	**T**	T	T	T	T	T	T	T	**F**
Bedingung (A)	**T**	T	T	T	T	T	**F**	F	F
`Zchn == 'A'`	**T**	**F**	F	F	F	F	–	–	–
`Zchn == 'E'`	**F**	**T**	F	F	F	F	–	–	–
`Zchn == 'I'`	**F**	F	**T**	F	F	F	–	–	–
`Zchn == 'O'`	**F**	F	F	**T**	F	F	–	–	–
`Zchn == 'U'`	**F**	F	F	F	**T**	F	–	–	–
Bedingung (B)	**T**	T	T	T	T	**F**	–	–	–

Verfahren im Überblick



Wann welches Verfahren wählen?

Typischerweise *Kombination* aus Pfad- und Bedingungsüberdeckung

Genügt es, Schleifen $1\times$ zu wiederholen?

Wenn ja: *boundary interior*-Test

Sonst: *strukturierter Pfadtest*

Genügt es, atomare Bedingungen zu prüfen?

Wenn ja: *einfache Bedingungsüberdeckung*

Sonst: *minimale Mehrfach-Bedingungsüberdeckung*







Zwischenbilanz

Die Auswahl der Testfälle kann anhand des Kontrollflusses wie folgt geschehen:

- Anweisungsüberdeckung
- Zweigüberdeckung
- Pfadüberdeckung (verschiedene Ausprägungen, insbes. strukturierter Pfadtest und *Boundary Interior*-Test)
- Bedingungsüberdeckung (verschiedene Ausprägungen, insbes. *minimale Mehrfach-Bedingungsüberdeckung*)

Der *Überdeckungsgrad* gibt an, wieviele Anweisungen / Zweige / Pfade durchlaufen wurden.







Datenflußorientierte Verfahren

Neben dem Kontrollfluß kann auch der *Datenfluß* als Grundlage für die Definition von Testfällen dienen.

Grundidee: Wir betrachten den *Datenfluß* im Programm, um Testziele zu definieren.

Variablenzugriffe werden in Klassen eingeteilt

Für jede Variable muß ein Programmpfad durchlaufen werden, für den bestimmte Zugriffskriterien zutreffen (*def/use*-Kriterien).







Def/Use-Kriterien

Zugriffe auf Variablen werden unterschieden in

Zuweisung (*definition*, *def*)

Berechnende Benutzung (*computational use*, *c-use*) zur Berechnung von Werten innerhalb eines Ausdrucks

Prädikative Benutzung (*predicative use*, *p-use*) zur Bildung von Wahrheitswerten in Bedingungen (Prädikaten)







Datenflußgraph

Der *Datenflußgraph* ist ein erweiterter Kontrollflußgraph

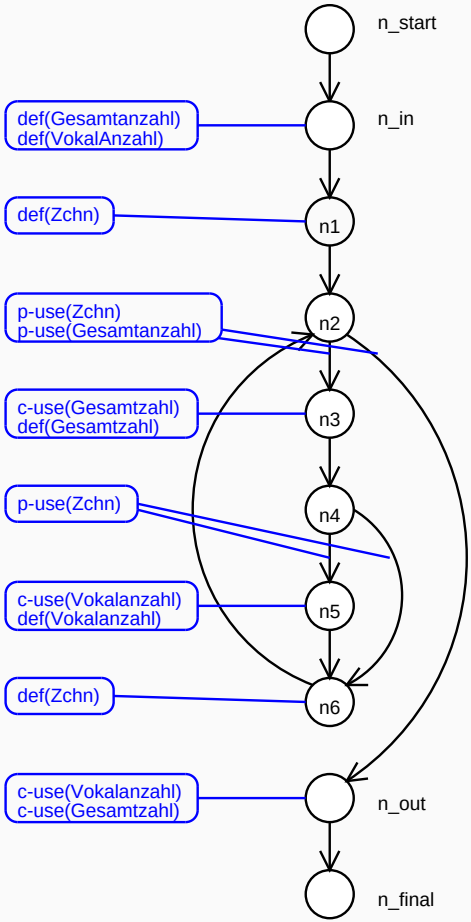
- Knoten sind attribuiert mit *def* und *c-use*:
  - $def(n)$: Menge der Variablen, die in n definiert werden
  - $c\text{-}use(n)$: Menge der Variablen, die in n berechnend benutzt werden
- Kanten sind attribuiert mit *p-use*:
  - $p\text{-}use(n_i, n_j)$: Menge der Variablen, die in der Kante (n_i, n_j) prädikativ benutzt werden
- Es gibt Extra-Knoten für Beginn und Ende eines Sichtbarkeitsbereiches







Datenflußgraph (2)




```
Import von 'VokalAnzahl'
und 'Gesamtzahl'

cin >> Zchn;

while (Zchn >= 'A' && Zchn <= 'Z' &&
       Gesamtzahl < INT_MAX)

Gesamtzahl = Gesamtzahl + 1;

if (Zchn == 'A' || Zchn == 'E' ||
    Zchn == 'I' || Zchn == 'O' ||
    Zchn == 'U')

VokalAnzahl = VokalAnzahl + 1;

cin >> Zchn;

Export von 'VokalAnzahl'
und 'Gesamtzahl'
```

Ein MinMax-Programm

Das MinMax-Programm ordnet min und max:

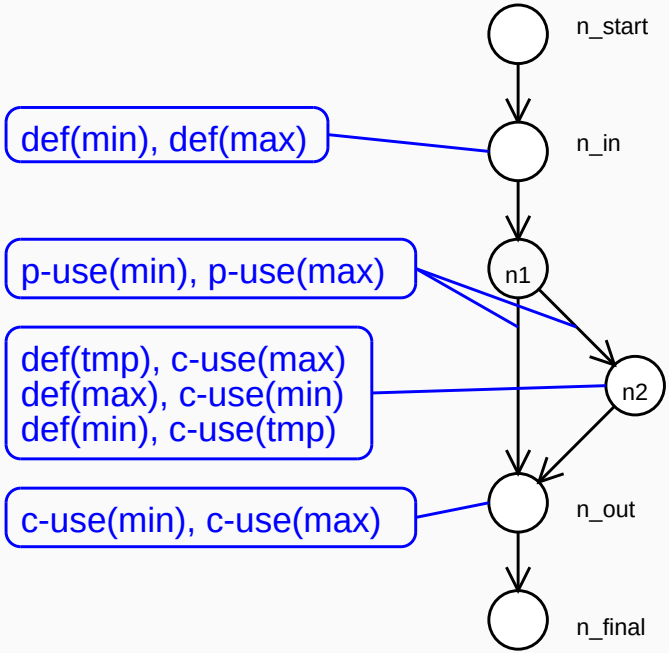
```
void MinMax(int& min, int& max)
{
    if (min > max)
    {
        int tmp = max;
        max = min;
        min = tmp;
    }
}
```







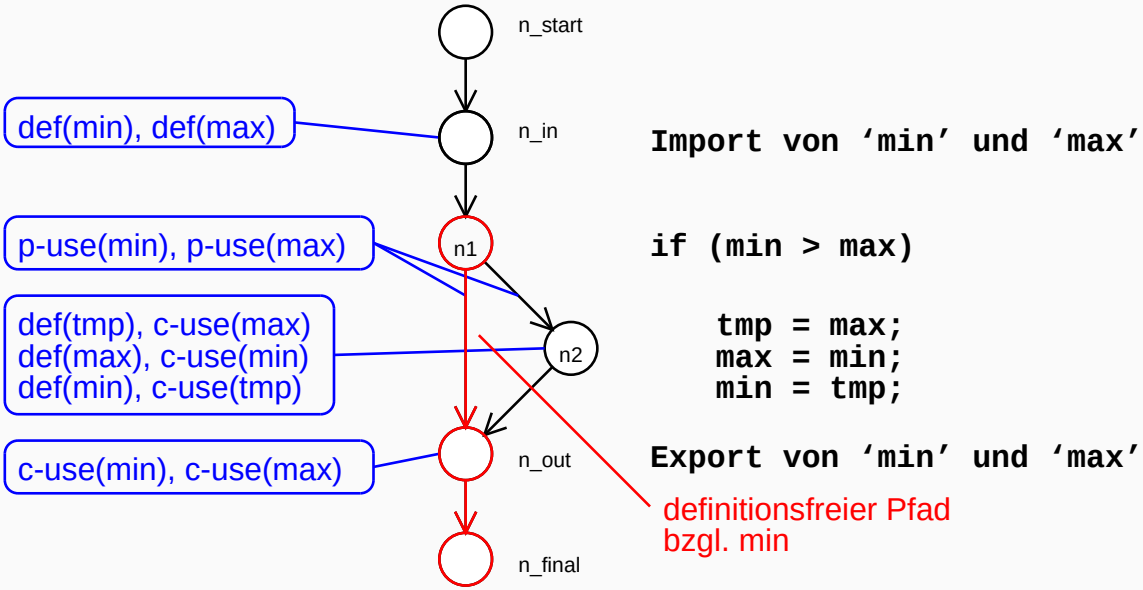
Datenflußgraph MinMax




```
Import von ‘min’ und ‘max’

if (min > max)

    tmp = max;
    max = min;
    min = tmp;

Export von ‘min’ und ‘max’
```

Definitionen

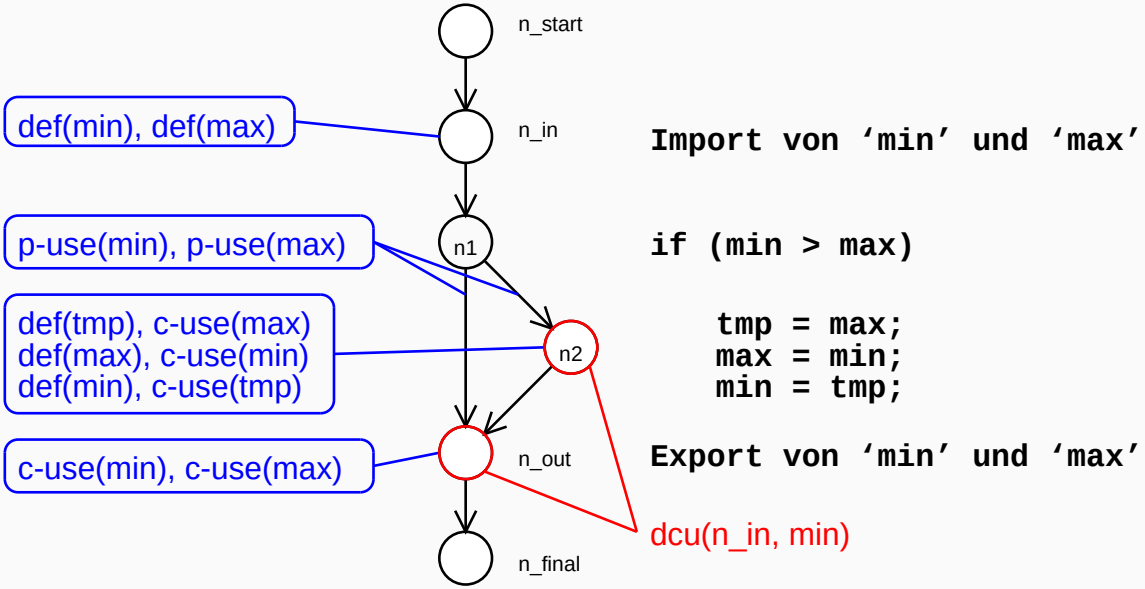
Ein *definitionsfreier Pfad* bezüglich einer Variablen x ist ein Pfad, in dem x nicht neu definiert wird.



Definitionen (2)

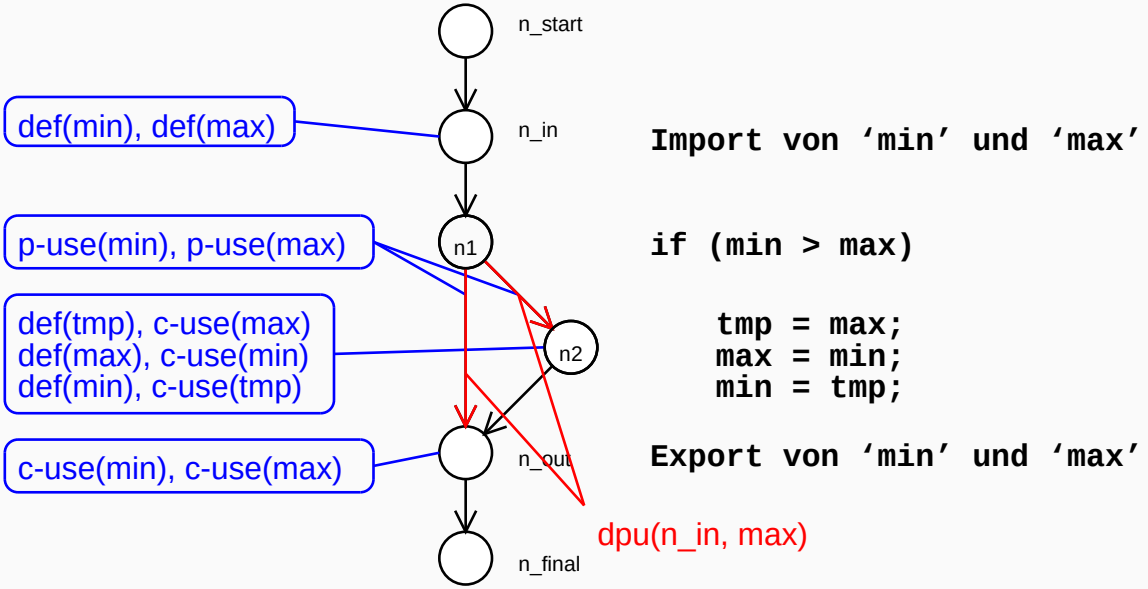
Die Menge $dcu(x, n_i)$ umfaßt alle Knoten n_j, in denen $x \in c\text{-}use(n_j)$ ist und ein definitionsfreier Pfad bezüglich x von n_i nach n_j existiert

$\Rightarrow$ „alle berechnenden Benutzungen von x aus n_i“



Definitionen (3)

Die Menge $dpu(x, n_i)$ umfaßt alle Kanten (n_j, n_k), in denen $x \in p\text{-}use(n_j, n_k)$ ist und ein definitionsfreier Pfad bezüglich x von n_i nach (n_j, n_k) existiert.
$\Rightarrow$ „alle prädikativen Benutzungen von x aus n_i“




```
Import von ‘min’ und ‘max’

if (min > max)

    tmp = max;
    max = min;
    min = tmp;

Export von ‘min’ und ‘max’
```

all defs-*Kriterium*

*Für jede Definition (*all defs*) einer Variablen wird eine Berechnung oder Bedingung getestet*

Für jeden Knoten n_i und jede Variable $x \in def(n_i)$ muß ein definitionsfreier Pfad zu *einem* Element von $dcu(x, n_i)$ oder $dpu(x, n_i)$ getestet werden.

auch statisch überprüfbar (Datenflußanalyse)

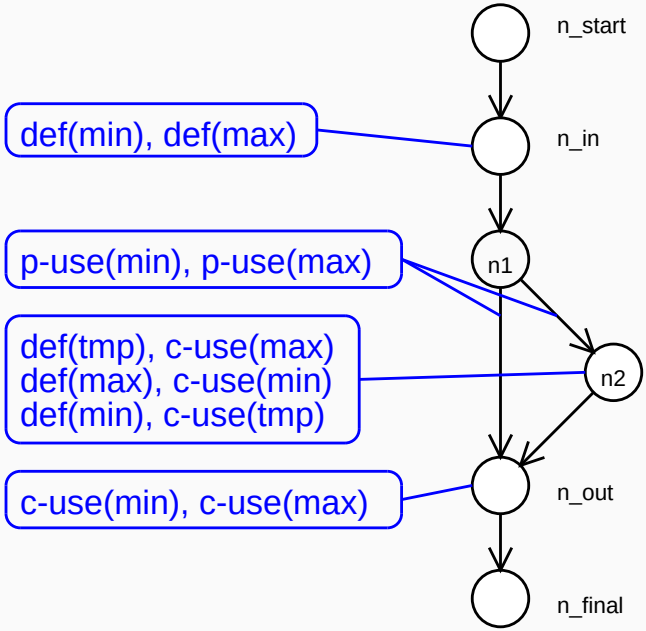
umfaßt weder Zweig- noch Anweisungsüberdeckung







all defs-*Kriterium (2)*



Testpfad
$n_{start}, n_{in}, n_1, n_2, n_{out}, n_{final}$
erfüllt *all defs*-Kriterium:

Definition min aus n_{in}
wird in n_2 benutzt

Definition min aus n_2
wird in n_{out} benutzt

max analog

all p-uses-*Kriterium*

Jede Kombination aus Definition und prädikativer Benutzung wird getestet

Für jeden Knoten n_i und jede Variable $x \in def(n_i)$ muß ein definitionsfreier Pfad zu *allen* Elementen von $dpu(x, n_i)$ getestet werden.

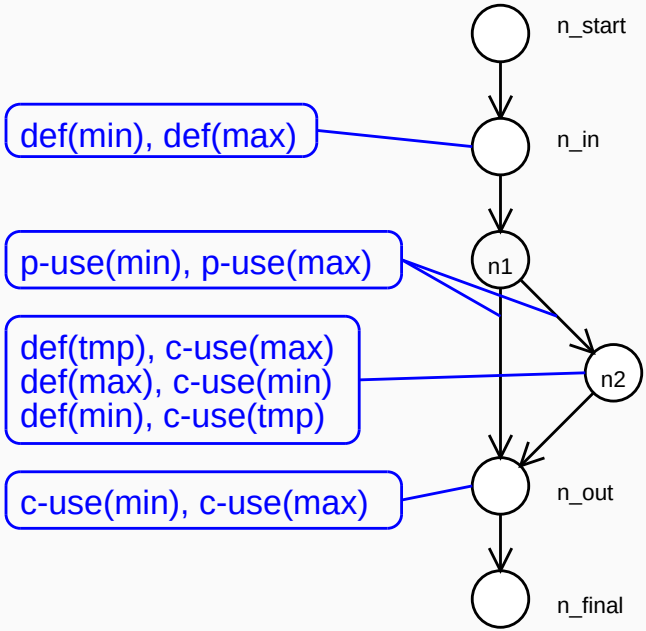
umfaßt Zweigüberdeckung







all p-uses-*Kriterium (2)*



Testpfade
$n_{start}, n_{in}, n_1, n_2, n_{out}, n_{final}$
$n_{start}, n_{in}, n_1, n_{out}, n_{final}$
erfüllen *all p-uses*-Kriterium:

prädikative Zugriffe auf *min*
sind beide abgedeckt

max analog

all c-uses-*Kriterium*

Jede Kombination aus Definition und berechnender Benutzung wird getestet

Für jeden Knoten n_i und jede Variable $x \in def(n_i)$ muß ein definitionsfreier Pfad zu *allen* Elementen von $dcu(x, n_i)$ getestet werden.

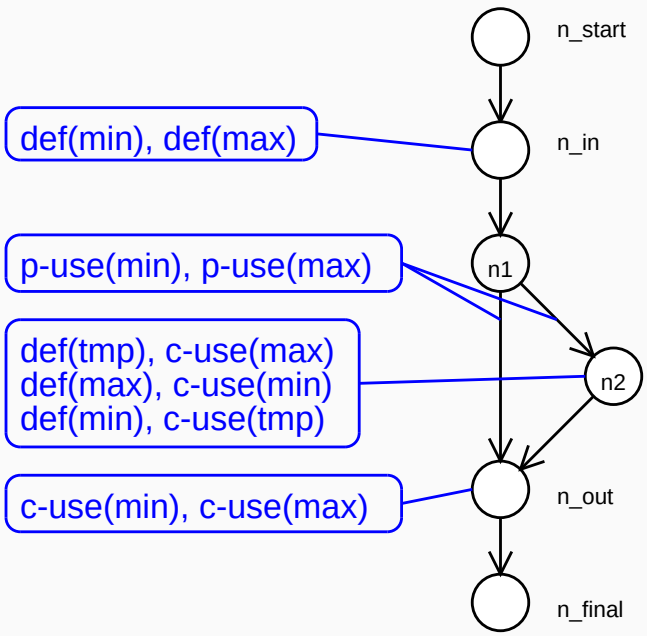
umfaßt weder Zweig- noch Anweisungsüberdeckung







all c-uses-*Kriterium (2)*



Testpfade
$n_{start}, n_{in}, n_1, n_2, n_{out}, n_{final}$
$n_{start}, n_{in}, n_1, n_{out}, n_{final}$
erfüllen *all c-uses*-Kriterium:

Definition in n_{in}
bedingt Test der Pfade
nach n_2 und n_{out}

Definition in n_2
bedingt Test des Pfades
nach n_{out}

max analog

Kombinierte Kriterien

all c-uses/some p-uses Wie *all c-uses*, aber:

Ist $dcu(x, n_i) = \emptyset$,
so muß ein definitionsfreier Pfad
zu *einem* Element von $dpu(x, n_i)$ getestet werden.

subsumiert *all defs* und *all c-uses*-Kriterium







Kombinierte Kriterien

all c-uses/some p-uses Wie *all c-uses*, aber:

Ist $dcu(x, n_i) = \emptyset$,
so muß ein definitionsfreier Pfad
zu *einem* Element von $dpu(x, n_i)$ getestet werden.

subsumiert *all defs* und *all c-uses*-Kriterium

all p-uses/some c-uses Wie *all p-uses*, aber:

Ist $dpu(x, n_i) = \emptyset$,
so muß ein definitionsfreier Pfad
zu *einem* Element von $dcu(x, n_i)$ getestet werden.

subsumiert *all defs* und *all p-uses*-Kriterium







all-uses-*Kriterium*

Jede Kombination aus Definition und Benutzung wird getestet

Für jeden Knoten n_i und jede Variable $x \in def(n_i)$ muß ein definitionsfreier Pfad zu allen Elementen von $dcu(x, n_i)$ und $dpu(x, n_i)$ getestet werden.

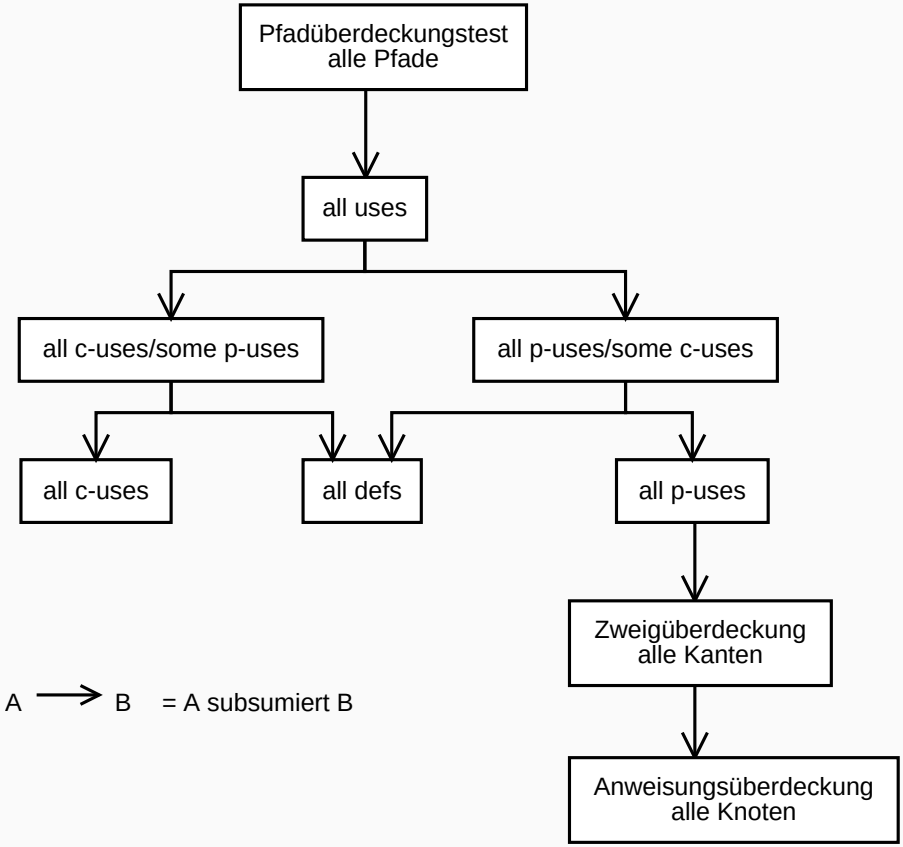
subsumiert *all p-uses* und *all c-uses*-Kriterium







Kriterien im Überblick



Zusammenfassung

Die Auswahl der Testfälle kann anhand der Programmstruktur wie folgt geschehen:

- Anweisungsüberdeckung
- Zweigüberdeckung
- Pfadüberdeckung (verschiedene Ausprägungen, insbes. strukturierter Pfadtest und *Boundary Interior*-Test)
- Bedingungsüberdeckung (verschiedene Ausprägungen, insbes. *minimale Mehrfach-Bedingungsüberdeckung*)
- Datenflußorientierte Verfahren (Def/Use-Kriterien)

Der *Überdeckungsgrad* gibt an, wieviele Anweisungen / Zweige / Pfade durchlaufen wurden.

Literatur

- **Lehrbuch der Softwaretechnik, Band 2** (Balzert)
- **Wissensbasierte Qualitätsassistenz zur Konstruktion von Prüfstrategien für Software-Komponenten** (P. Liggesmeyer, 1993)