Strukturelle Testverfahren
White-Box-Tests

Kontrollflussorientierte Testverfahren

- codebasierte Testverfahren: Grundlage ist Programmtext des Testobjekts
- Idee: alle Quellcodeteile des Testobjekts mindestens einmal zur Ausführung bringen
- anhand Programmlogik ablauforientierte Testfälle ermitteln und ausführen
- Testende bei zuvor festgelegter Überdeckung
  * Anweisungsüberdeckung
  * Zweigüberdeckung
  * Pfadüberdeckung
  * Bedingungsüberdeckung
- Voraussetzung: Kontrollflussgraph

Beispiel

Programmbeschreibung: Zähle in einer Textdatei die Anzahl der Zeilen, Wörter und Zeichen. Ein Wort ist definiert als eine Zeichenfolge, die nicht durch Blank getrennt wird.

```
#include <stdio.h>
#include <ctype.h>
#define N 40

int main(int argc, char *argv[]) {
    int i;
    long linec, wordc, charc;
    char l[N];
    FILE *file;
```


Beispiel (2)

```
if (argc != 2) {
    printf("usage: %s filename\n", argv[0]);
    return 1;
}

linec = 0;
wordc = 0;
charc = 0;

file = fopen(argv[1], "r");
if (file == NULL) {
    perror(argv[1]);
    return 2;
}
```

Beispiel (3)

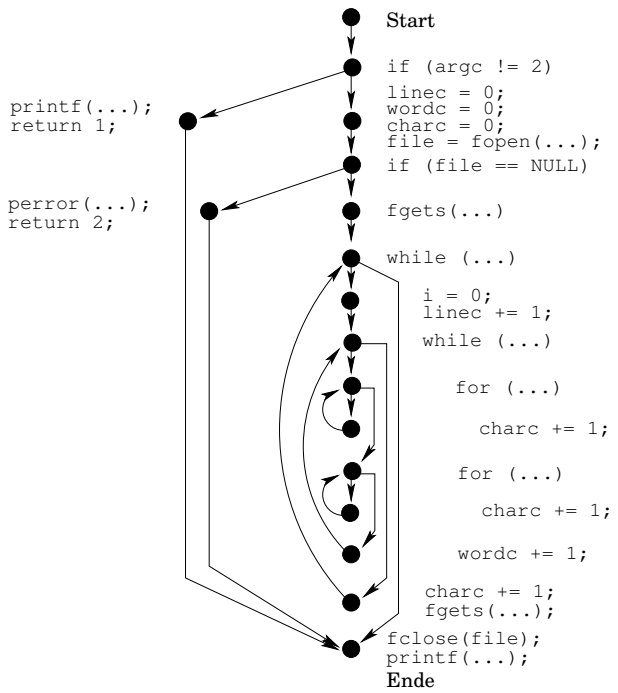
```
fgets(l, N, file);
while (!feof(file)) {
    i = 0;
    linec += 1;
    while (l[i] != '\n') {
        for (; isblank(l[i]) && l[i] != '\n'; i++)
            charc += 1;
        for (; !isblank(l[i]) && l[i] != '\n'; i++)
            charc += 1;
        wordc += 1;
    }
    fgets(l, N, file);
}
```


Beispiel (4)

```
    fclose(file);
    printf("%ld %ld %ld\n", linec, wordc, charc);
    return 0;
}
```

Kontrollflussgraph

- auch Programmablaufplan genannt
- gerichteter Graph $G = (V, E)$ mit Startknoten s und Endknoten t
- Pfad: Folge $v_1, v_2, \ldots, v_n$ von Knoten mit $v_1 = s$, $v_n = t$ und $(v_i, v_{i+1}) \in E$ für $i = 1, \ldots, n-1$



Anweisungsüberdeckungstest

Eigenschaften

- auch C_0-Test genannt
- verlangt Ausführung aller Anweisungen (Knoten) oder einer zuvor festgelegten Mindestquote

$$\text{Überdeckung} = \frac{\#\text{ durchlaufene Anweisungen}}{\#\text{ Anweisungen}} \cdot 100\%$$

- Testaufwand minimieren → möglichst wenige Testfälle
- wesentliche Aspekte werden nicht geprüft (z.B. fehlende Anweisungen in einem leeren `else`)
- niedrige Quote: nur 18% der Fehler werden entdeckt

Anweisungsüberdeckungstest (2)

in unserem Programmbeispiel:

- Aufruf: `wordcount`
  → behandeln fehlender Programmparameter
- Aufruf: `wordcount tst.txt` (`tst.txt` existiert nicht)
  → behandeln von Dateizugriffsfehlern
- Aufruf: `wordcount test.txt` (Inhalt: `Ein Test!`)
  → alle restlichen Anweisungen

⇒ Anweisungsüberdeckung = 100%

Anweisungsüberdeckungstest (3)

Bewertung

- notwendiges, aber nicht hinreichendes Testkriterium
- nur in Kombination mit anderen Verfahren geeignet
- nicht ausführbarer Code kann gefunden werden

Beispiel:

```
if (2 * x > 0) {
    x = x / 2;
    if (x < 0)      // kann nicht erfüllt sein!
        x += 10;    // wird nie ausgeführt
}
```


Zweigüberdeckungstest

Eigenschaften

- auch C_1-Test genannt
- verlangt Ausführung aller Zweige (Kanten) oder einer zuvor festgelegten Mindestquote

$$\text{Überdeckung} = \frac{\#\text{ durchlaufene Zweige}}{\#\text{ Zweige}} \cdot 100\%$$

- festhalten, welche Zweige bei welchem Testfall durchlaufen werden sollen, um Abweichungen im Ablauf feststellen zu können
- niedrige Quote: 34% der Fehler werden entdeckt
- Erfolgsquote ist höher als bei statischer Analyse
- fehlende Zweige werden nicht direkt entdeckt

Zweigüberdeckungstest (2)

in unserem Programmbeispiel:

- Aufruf `wordcount`
  → behandeln fehlender Programmparameter
- Aufruf `wordcount tst.txt` (`tst.txt` existiert nicht)
  → behandeln von Dateizugriffsfehlern
- Aufruf `wordcount test.txt` (Inhalt: `Ein kleiner\tTest!`)
  → alle restlichen Kanten

⇒ Zweigüberdeckung = 100%
(gleiche Testfälle wie bei Anweisungsüberdeckung)

Aber: ein fehlender Zweig wird nicht gefunden (Speicherzugriffsfehler bei Zeilen mit mehr als 40 Zeichen)

Zweigüberdeckungstest (3)

Bewertung

- gilt als minimales Testkriterium
- nicht ausführbare Zweige können gefunden werden
- Kontrollfluss wird an Verzweigungen kontrolliert
- Optimierung oft durchlaufender Programmteile möglich

für objektorientierte Systeme unzureichend

- die einzelnen Methoden sind normalerweise wenig umfangreich und von geringer Komplexität
- die Komplexität entsteht durch Beziehungen zwischen den Klassen

Zweigüberdeckungstest (4)

Nachteile

- unzureichend für den Test von Schleifen
- Abhängigkeiten zwischen Zweigen werden nicht berücksichtigt ⇒ **Pfadüberdeckungstest**
- ungeeignet für Test komplexer Bedingungen ⇒ **Bedingungsüberdeckungstest**

Zweigüberdeckungstest (5)

GNU COVerage tool: gcov*

- Tool zum Messen der Anweisungs-/Zweigüberdeckung von C++/C-Programmen
- das Programm ist mit den Optionen `-g -fprofile-arcs -ftest-coverage` zu übersetzen
- anschließend wird das Programm ausgeführt
- mit `gcov` bzw. `gcov -b` wird die Auswertung angezeigt

* näheres in Zeller, Krinke: Programmierwerkzeuge. dpunkt.verlag.

Zweigüberdeckungstest (6)

`gcov` in unserem Programmbeispiel:

- übersetzen mit `gcc -g -fprofile-arcs -ftest-coverage wordcount.c -o wordcount`
- Programmausführung mit `./wordcount _wc.txt`
- `gcov -b wordcount` liefert

```
Lines executed: 100.00% of 33
Branches executed: 100.00% of 14
Taken at least once: 92.86% of 14
Calls executed: 100.00% of 12
```

- in Datei `wordcount.c.gcov` weitere Informationen

Überdeckung hängt von Compiler-Optimierung ab!

- Compile ohne Optimierung, dann obige Testfälle

```
       for (; isblank(line[i]) && line[i]!='\n'; i++)
branch  0 taken 67%
branch  1 taken 100%
branch  2 never executed
branch  3 taken 100%
```

- Compile mit Optimierung `O3`, dann obige Testfälle

```
       for (; isblank(line[i]) && line[i]!='\n'; i++)
call    0 returns 100%
branch  1 taken 100%
branch  2 taken 67%
branch  3 taken 100%
```


Pfadüberdeckungstest

Eigenschaften

- verlangt Ausführen aller Programmpfade (unrealistisch)
- entwickelt zum Testen von Schleifen
- Pfadanzahl wächst bei unbestimmten Wiederholungen explosiv
- Teil der konstruierbaren Pfade nicht ausführbar, da sich Bedingungen gegenseitig ausschließen können
- gutes Testverfahren: Quote bei 64%
- besser nur in Kombination mit anderen Verfahren

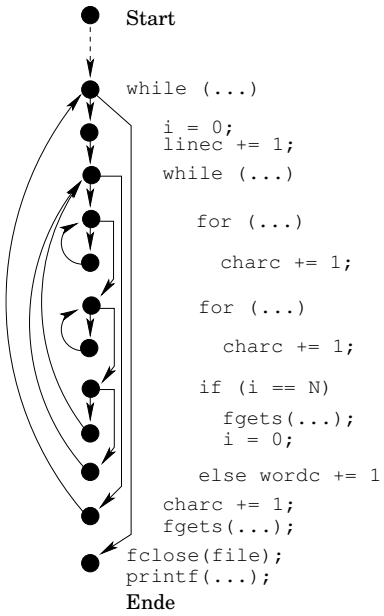
In unserem Programmbeispiel: $n_s, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, [n_9, n_8]^{40}, \ldots$ führt zu einem Speicherzugriffsfehler!

Beispiel: Korrektur

lesen einer Zeile in Blöcken von `N` Zeichen

```
while (line[i] != '\n') {
    for (; (i < N) && isblank(line[i])
                   && line[i] != '\n'; i++)
        charc += 1;
    for (; (i < N) && !isblank(line[i])
                   && line[i] != '\n'; i++)
        charc += 1;
    if (i == N) {
        fgets(line, N, file);
        i = 0;
    } else wordc += 1;
}
```


Beispiel: korrigierter Kontrollflussgraph



Pfadüberdeckungstest (2)

In unserem korrigierten Programmbeispiel:

1. n_s, n_1, n_a, n_t siehe Zweigüberdeckungstest
2. $n_s, n_1, n_2, n_3, n_b, n_t$ siehe Zweigüberdeckungstest
3. $n_s, n_1, n_2, n_3, n_4, n_5, n_t$ siehe Zweigüberdeckungstest
4. $n_s, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_{15}, n_5, n_t$
   $\rightarrow$ Datei enthält nur eine leere Zeile
5. $n_s, \ldots, n_5, n_6, n_7, n_8, n_{10}, n_{12}, n_{14}, n_7, n_{15}, n_5, n_t$
   $\rightarrow$ n_8, n_{10}, n_{12} im ersten Schleifendurchlauf nicht möglich
6. $n_s, \ldots, n_5, n_6, n_7, n_8, n_{10}, [n_{11}, n_{10}]^4, n_{12}, n_{14}, n_7, n_{15}, n_5, n_t$
   $\rightarrow$ Zeile enthält ein Wort aus 4 Zeichen
7. usw.

Pfadüberdeckungstest (3)

entdeckte Fehler:

- $n_s, \ldots, n_5, n_6, n_7, n_8, n_9, n_8, n_{10}, n_{12}, n_{14}, n_7, n_{15}, n_5, n_t$
  $\rightarrow$ Zeile enthält ein Blank: es wird ein Wort gezählt!
  allg: Anzahl Wörter falsch, wenn Blank am Zeilenende
- $n_s, \ldots, n_5, n_6, n_7, n_8, n_{10}, [n_{11}, n_{10}]^N, n_{12}, n_{13}, n_7, n_8, n_9, n_8, n_{10}, [n_{11}, n_{10}]^x, n_{12}, n_{14}, n_7, n_{15}, n_5, n_t$
  $\rightarrow$ falsche Anzahl Wörter, wenn Block endet bei Wortende und weitere Wörter in Zeile vorhanden
  $\rightarrow$ falsche Anzahl Zeichen, da `fgets` am Blockende automatisch ein `'\0'` einfügt

Pfadüberdeckungstest (4)

Bewertung

- keine praktische Bedeutung, da nur sehr eingeschränkt durchführbar
- interessant nur im Zusammenhang mit fehlerorientierten Testansätzen

Variante: Boundary Interior Test praktisch anwendbar

- Schleifen werden beim Test nur einmal überprüft
- 2 Gruppen von Pfaden für jede Schleife im Programm
  * boundary test: alle Pfade, die die Schleife betreten, aber nicht wiederholen
  * interior test: alle Pfade, die eine Schleifenwiederholung beinhalten

Beispiel: Korrektur

```
while (line[i] != '\n') {
    int found = 0;

    // White-Space lesen
    for (; (i < N) && isblank(line[i])
                   && line[i] != '\n'; i++)
        charc += 1;

    // Wort lesen
    for (; (i < N) && !isblank(line[i])
                   && line[i] != '\0'
                   && line[i] != '\n'; i++) {
        charc += 1;
        found = 1;
    }
```

Beispiel: Korrektur (2)

```
    // Blockende erreicht
    if (i == N - 1) {
        fgets(line, N, file);
        i = 0;
        if (isblank(line[0]))
            wordc += found;
    } else wordc += found;
}
```


Bedingungsüberdeckungstest

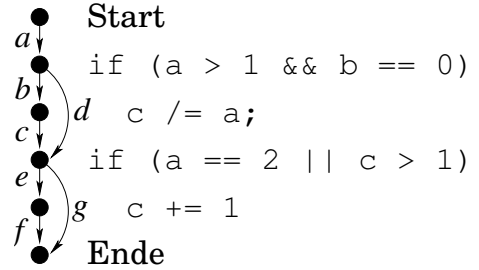
bisher: bei Zweigüberdeckung wird nur der Ergebniswert einer Bedingung berücksichtigt (true/false)

Ziel: Testen boolescher Ausdrücke bei Schleifen und bedingten Anweisungen → Testintensität ist abhängig von Komplexität der Bedingung!

Beispiel:

```
if (a > 1 && b == 0)
    c /= a;
if (a == 2 || c > 1)
    c += 1;
```

Kontrollflussgraph:



Bedingungsüberdeckungstest (2)

einfache Bedingungsüberdeckung: Im Test nimmt jeder atomare Prädikatsterm jeweils den Wert *true* und *false* an.

im Beispiel:

- atomare Prädikatsterme: `a > 1`, `b == 0`, `a == 2` und `c > 1`
- Test 1: $a = 2$, $b = 1$ und $c = 4$ → Pfad $adef$
- Test 2: $a = 1$, $b = 0$ und $c = 1$ → Pfad adg

⇒ Zweig bc wird nicht abgedeckt!

Zweig- und Anweisungsüberdeckungstest nicht enthalten!

Bedingungsüberdeckungstest (3)

Zweig-/Bedingungsüberdeckung:
Mischform aus einfacher Bedingungsüberdeckung und einer expliziten Zweigüberdeckung

im Beispiel:

- Test 1: $a = 2$, $b = 0$ und $c = 4$ → Pfad $abcef$
- Test 2: $a = 1$, $b = 1$ und $c = 1$ → Pfad adg

⇒ alle Zweige werden abgedeckt

Bedingungsüberdeckungstest (4)

Mehrfach-Bedingungsüberdeckung: Alle Variationen der atomaren Bedingungen testen.

im Beispiel: 8 Testfälle

a	b	a>1	&&	b==0
1	0	f	f	t
1	1	f	f	f
2	0	t	t	t
2	1	t	f	f

a	c	a==2	||	c>1
1	1	f	f	f
1	2	f	t	t
2	1	t	t	f
2	2	t	t	t

Probleme:

- bei n atomaren Prädikatstermen gibt es 2^n Variationen
- nicht alle Variationen sind möglich ($a < 2$ AND $a > 7$)

Bedingungsüberdeckungstest (5)

minimale Mehrfach-Bedingungsüberdeckung*:
Jede Bedingung (ob atomar oder nicht) muss mindestens einmal *true* und *false* sein.

im Beispiel: 2 Testfälle

a	b	a>1	&&	b==0
1	1	f	f	f
2	0	t	t	t

a	c	a==2	||	c>1
1	1	f	f	f
2	2	t	t	t

Unterschied zu Zweig-/Bedingungsüberdeckungstest?

---

* unterschiedliche Definitionen des Begriffs in der Literatur

Bedingungsüberdeckungstest (6)

minimale Mehrfach-Bedingungsüberdeckung:
neben den atomaren und der Gesamtentscheidung auch alle zusammengesetzten Teilentscheidungen gegen wahr und falsch prüfen. (P. Liggesmeyer)

minimale Mehrfach-Bedingungsüberdeckung:
Jede mögliche Kombination von Wahrheitswerten, bei denen die Änderung des Wahrheitswerts eines Terms den Wahrheitswert der logischen Verknüpfung ändern kann. (E.H. Riedemann)

zugrundeliegende Idee: entspricht der Testauswahl beim Ursache-Wirkungsgraphen (eine fehlerhafte Berechnung eines atomaren Wahrheitswerts wird nicht maskiert)

Bedingungsüberdeckungstest (7)

Beispiele:

a	b	a AND b
f	f	f
f	t	f
t	f	f
t	t	t

a	b	not (a EXOR b)
f	f	t
f	t	f
t	f	f
t	t	t

- **AND** der Fall a = b = *false* wird nicht getestet:
  wenn a = *false* und die Berechnung von b = *false* falsch ist $\Rightarrow$ Fehler nicht sichtbar
- **not EXOR** alle Fälle werden getestet

Bedingungsüberdeckungstest (8)

Beispiele: (Fortsetzung)

a	b	c	a OR b OR c
f	f	f	f
f	f	t	t
f	t	f	t
f	t	t	t

a	b	c	a OR b OR c
t	f	f	t
t	f	t	t
t	t	f	t
t	t	t	t

OR teste Kombination, bei der alle Werte *false* sind und alle Kombinationen, bei denen genau ein Wert *true* ist:

sind zwei der Werte *true* und eine der zugrundeliegenden Berechnungen ist falsch $\Rightarrow$ Fehler nicht sichtbar

Bedingungsüberdeckungstest (9)

in unserem Beispiel: `a>1 && b==0` und `a==2 || c>1`

- Test 1: $a = 3$, $b = 0$, $c = 3$
- Test 2: $a = 2$, $b = 1$, $c = 1$
- Test 3: $a = 1$, $b = 0$, $c = 1$

nicht zu testen sind die Fälle:

- AND: $a \leq 1$, $b \neq 0$
- OR: $a = 2$, $c > 1$

Bedingungsüberdeckungstest (10)

Bewertung:

- einfache Bedingungsüberdeckung nicht ausreichend
- Mehrfach-Bedingungsüberdeckung ist zu aufwändig
- minimale Mehrfach-Bedingungsüberdeckung ist notwendig, da komplexe Bedingungen oft fehlerhaft sind (subsummiert Anweisungs- und Zweigüberdeckung)
- komplexe, zusammengesetzte Bedingungen können in verschachtelte, einzelne Abfragen aufgeteilt werden $\rightarrow$ dann reicht Zweigüberdeckungstest
- schwer: messen der Überdeckung der Teilbedingungen (Compiler verkürzt Auswertung von booleschen Bedingungen und beeinflusst Auswertungsreihenfolge)

Datenflussorientierte Testverfahren

Überblick: defs/uses-Verfahren*

- Test der Datenbenutzungen: Definition von Variablen, lesende und schreibende Zugriffe auf Variablen
- eignen sich für Test von Datenobjekt- und Datentypmodulen sowie Klassen
- nur wenige Testwerkzeuge vorhanden
- defs/uses-Kriterien auf Basis von Variablenzugriffen
  - * *def:* Wertzuweisung oder Definition
  - * *c-use:* berechnen von Werten in einem Ausdruck
  - * *p-use:* bilden von Wahrheitswerten in Bedingungen

* näheres in E.H. Riedemann: Testmethoden für sequentielle und nebenläufige Software-Systeme. B.G.Teubner Verlag

defs/uses-Verfahren

all defs-Kriterium

- für jede Definition einer Variablen wird eine Berechnung oder Bedingung getestet
- beinhaltet weder Zweig- noch Anweisungsüberdeckung

all p-uses-Kriterium

- jede Kombination aus Definition und prädikativer Benutzung testen
- beinhaltet Zweigüberdeckung

all c-uses-Kriterium

- testet berechnende Zugriffe auf Variablen
- beinhaltet weder Zweig- noch Anweisungsüberdeckung

defs/uses-Verfahren (2)

weitere Kriterien:

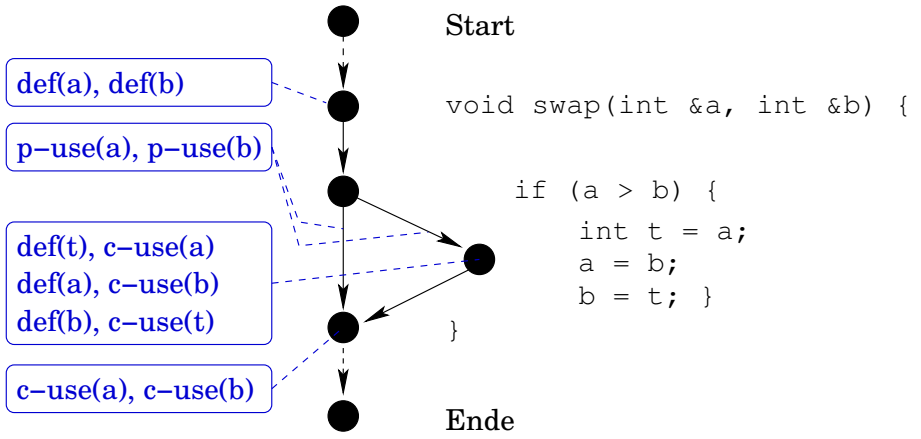
- **all c-uses/some p-uses:** Alle berechnenden Zugriffe testen. Falls keiner existiert, muss Variable in mindestens einer Bedingung benutzt werden.
- **all p-uses/some c-uses:** Alle prädikativen Zugriffe testen. Falls keiner existiert, muss Variable in mindestens einer Berechnung verwendet werden.
- **all uses:** Für jede Definition werden alle berechnenden und alle prädikativen Zugriffe auf die Variable getestet.

Datenflussgraph

- ist ein erweiterter Kontrollflussgraph
- Knoten sind attribuiert mit *def* und *c-use*:
  - *def*(n): Menge der Variablen, die in n definiert werden
  - *c-use*(n): Menge der Variablen, die in n berechnend benutzt werden
- Kanten sind attribuiert mit *p-use*:
  - *p-use*(n_i, n_j): Menge der Variablen, die in der Kante (n_i, n_j) prädikativ benutzt werden
- zusätzliche Knoten für Beginn und Ende eines Sichtbarkeitsbereiches

Datenflussgraph (2)

Beispiel:

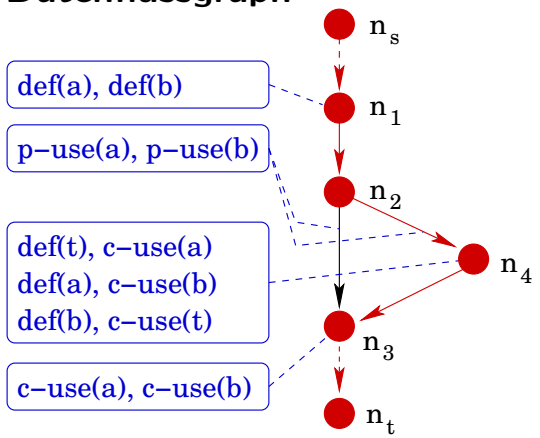


defs/uses-Verfahren (3)

all defs

- Testpfad:
  $n_s, n_1, n_2, n_4, n_3, n_t$
- *def(a)* aus n_1 wird in n_4 benutzt
- *def(a)* aus n_4 wird in n_3 benutzt
- *def(b)* analog

Datenflussgraph

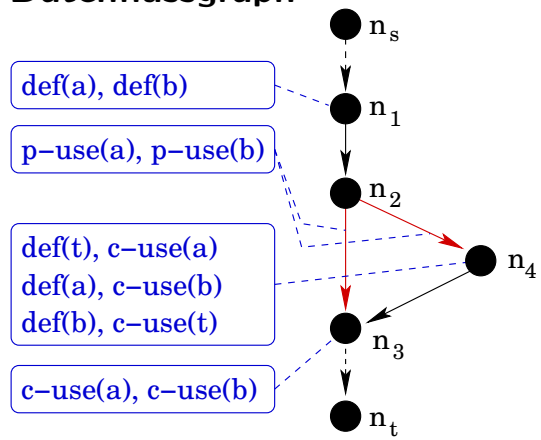


defs/uses-Verfahren (4)

all p-uses

- Testpfad 1:
  $n_s, n_1, n_2, n_4, n_3, n_t$
- *def(a)* aus n_1 wird in (n_2, n_4) benutzt
- *def(b)* analog
- Testpfad 2:
  n_s, n_1, n_2, n_3, n_t
- *def(a)* aus n_1 wird in (n_2, n_3) benutzt
- *def(b)* analog

Datenflussgraph

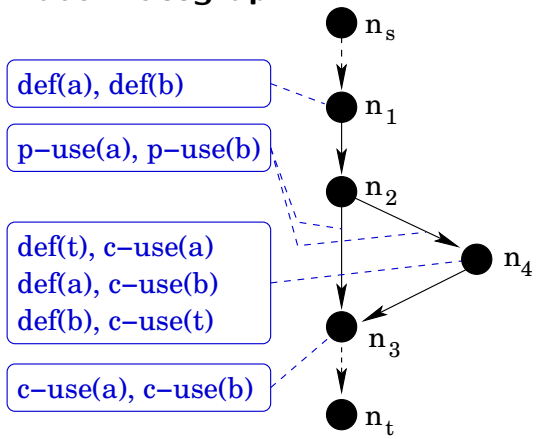


defs/uses-Verfahren (5)

all c-uses

- Testpfade:
  $n_s, n_1, n_2, n_4, n_3, n_t$
  und n_s, n_1, n_2, n_3, n_t
- *def(a)* in n_1 bedingt Test der Pfade nach n_3 und n_4
- *def(a)* in n_4 bedingt Test von n_4 nach n_3
- *def(b)* analog

Datenflussgraph



Bewertung der White-Box-Testverfahren

- nur geeignet für untere Teststufen, nicht geeignet für Systemtest
- übersehene/vergessene und daher nicht realisierte Anforderungen werden nicht abgedeckt
- werkzeugunterstützung notwendig: Testobjekt muss an strategisch wichtigen Stellen instrumentiert werden
  * Zähler einbauen und mit Null initialisieren
  * beim Durchlauf der entsprechenden Stelle Zähler inkrementieren
  * nach Test Zähler auf Null → Programmstelle wurde nicht durchlaufen
  * Instrumentierung von Hand zu teuer und fehleranfällig