Software Testing

Tecniche di testing strutturale

1

1

Testing Strutturale

- La definizione dei casi di test è fondata sulla base della conoscenza della struttura del software ed in particolare del suo codice
- Tipi e strutture di dati (classi), strutture di controllo, flusso di controllo, flusso dei dati, etc.
- Non è scalabile (usato soprattutto a livello di unità o sottosistema)
- Attività complementare al testing funzionale
- Non può rilevare difetti che dipendono dalla mancata implementazione di alcune parti della specifica

2

2

Tecniche di Testing Strutturale

- In generale fondate sull'adozione di ***criteri di copertura*** degli oggetti che compongono la struttura dei programmi
  - ✓ ***COPERTURA***: definizione di un insieme di casi di test in modo tale che gli oggetti di una definita classe (es. strutture di controllo, istruzioni, archi del GFC, predicati,..etc.) siano attivati almeno una volta nell'esecuzione dei casi di test
  - ✓ Definizione di una metrica di copertura: ***Test Effectiveness Ratio (TER)*** = # oggetti coperti / # oggetti totale
  - ✓ *NB: Copertura totale(100%) non sempre possibile (problemi di indecidibilità)*

3

3

Selezione di casi di test

- Come selezionare i casi di test per il criterio di copertura adottato ?
  - ✓Ogni caso di test corrisponde all'esecuzione di un particolare cammino sul GFC di un programma P
  - ✓Individuare i cammini che ci garantiscono il livello di copertura desiderato
  - ✓Esecuzione simbolica di ogni cammino e individuazione di input data che soddisfano la path condition (e che causano l'esecuzione del cammino)
- Troppo costoso …

4

4

Una strategia meno costosa …

- Scegliere un criterio di copertura funzionale e un criterio di copertura strutturale
  - ✓ complementarietà di test funzionale e strutturale
- Individuare i casi di prova in accordo al criterio di copertura funzionale ed eseguire il test funzionale
- Controllare la copertura rispetto al criterio di copertura strutturale scelto
- Individuare le parti non coperte e selezionare dei cammini che ci consentono di raggiungere il livello di copertura desiderato
- Individuare i casi di prova per l'esecuzione di questi cammini ed eseguire il test strutturale

5

5

Copertura dei nodi (statement)

- Dato un programma P, viene definito un insieme di test case la cui esecuzione implica l'attraversamento di tutti i nodi di GFC(P), ovvero l'esecuzione di tutte le istruzioni di P
  - ✓ ***Test Effectiveness Ratio (TER)***

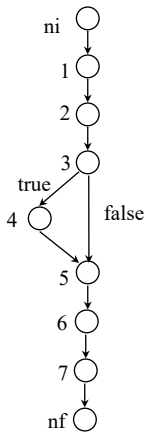
$$\text{TER} = \frac{\text{n.ro di statement eseguiti}}{\text{n.ro di statement totale}}$$

6

6

Esempio (node coverage)

```
   void statement()
   {
     double x, y;
1.   cin >> x;
2.   cin >> y;
3.   if (x != 0)
4.         x = x + 10;
5.   y = y/x;
6.   cout << x;
7.   cout << y;
   }
```



copertura del 100%:
cammino
1, 2, 3, 4, 5, 6, 7

Esecuzione simbolica:
path condition *X* != 0

input data per caso di test:
y: qualsiasi valore;
x: valore diverso da 0

NB: failure per x = 0

7

7

Copertura delle decisioni (branch)

- Dato un programma P, viene definito un insieme di test case la cui esecuzione implica l'attraversamento di tutti i rami di GFC(P), ovvero l'esecuzione di tutte le decisioni di P
  - ✓ ***Test Effectiveness Ratio (TER)***

$$TER = \frac{\text{n.ro di branch eseguiti}}{\text{n.ro di branch totali}}$$

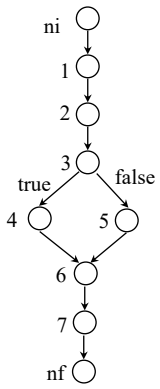
- ***NB: la copertura delle decisioni implica la copertura dei nodi***

8

8

Esempio (branch coverage)

```
   void branch()
   {
     double x, y;
1.   cin >> x;
2.   cin >> y;
3.   if (x == 0 || y > 0)
4.           y = y / x;
5.   else   x = y + 2/x;
6.   cout << x;
7.   cout << y;
    }
```



cammini:
a) 1, 2, 3, 4, 6, 7
b) 1, 2, 3, 5, 6, 7

path conditions:
a) $X == 0 \;||\; Y > 0$
b) $X \;!= 0 \;\&\&\; Y <= 0$

input data:
a) y > 0, x != 0
b) y<= 0, x != 0

NB: failure per x = 0

9

9

Copertura delle condizioni

- Dato un programma P, viene definito un insieme di test case la cui esecuzione implica l'esecuzione di tutte le condizioni (valori vero e falso delle componenti relazionali dei predicati) caratterizzanti le decisioni in P
  - ✓ ***NB: nessuna relazione tra copertura delle decisioni e copertura delle condizioni***
  - ✓ Esempio precedente, input data:
    a) y > 0, x != 0
    b) y<= 0, x == 0
    ... eseguito solo il ramo true

10

10

Copertura di decisioni e condizioni

- Dato un programma P, viene definito un insieme di test case la cui esecuzione implica l'esecuzione di tutte le decisioni e di tutte le condizioni caratterizzanti le decisioni in P
  - ✓ ***Tali criteri implicano sia copertura delle condizioni che copertura delle decisioni***
- ***Multiple condition coverage***: Per ogni decisione vengono considerate tutte le combinazioni di condizioni
- ***Modified condition coverage***: Per ogni decisione vengono considerate solo le combinazioni di valori delle condizioni per le quali una delle condizioni determina il valore di verità della decisione
  - ✓ ***Riduzione dei casi di test …***

11

11

Esempio: Modified Condition Coverage

- Esempio : $A \wedge (B \vee C)$

	ABC	Res.	Corr. False Case
1	TTT	T	A (5)
2	TTF	T	A (6), B (4)
3	TFT	T	A (7), C (4)
4	TFF	F	B (2), C (3)
5	FTT	F	A (1)
6	FTF	F	A (2)
7	FFT	F	A (3)
8	FFF	F	-

Prendere una coppia per ogni condizione:
- A : (1,5), opp. (2,6), opp. (3,7)
- B : (2,4)
- C : (3,4)

Due insiemi minimi per coprire il modified condition criterion :
- (2,3,4,6) or (2,3,4,7)

4 casi di test test invece di 8 per tutte le possibili combinazioni

12

12

Copertura dei cammini

- Dato un programma P, viene definito un insieme di test case la cui esecuzione implica l'attraversamento di tutti i cammini di GFC(P)
  - ✓ ***Test Effectiveness Ratio (TER)***

$$TER = \frac{\text{n.ro di cammini eseguiti}}{\text{n.ro di cammini totali}}$$

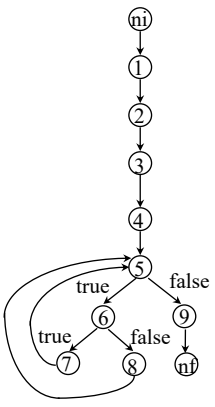
- ***Problemi:***
  - ✓ numero di cammini infinito (o comunque elevato)
  - ✓ infeasible path

13

13

Esempio

```
void gcd() {
    int x, y, a, b;
1.  cin >> x;
2.  cin >> y;
3.  a = x;
4.  b = y;
5.  while (a != b)
6.        if(a > b)
7.              a = a - b;
8.        else  b = b - a;
9.  cout << a;
}
```



I cicli possono portare a cammini infiniti:

1, 2, 3, 4, 5, 6, 7, 5, 6, 7, ...

14

14

Copertura dei cammini: soluzioni

- Un numero di cammini infinito implica la presenza di circuiti
  - ✓ NB: il numero dei cammini elementari (privi di circuiti) in un grafo è finito
- Soluzione: limitare l'insieme dei cammini
  - ✓ ***Criterio di n-copertura dei cicli***
  - ✓ ***Metodi basati su exemplar-paths***
  - ✓ ***Metodi dei cammini linearmente indipendenti (Mc Cabe)***
  - ✓ ***Metodi basati su data-flow***

15

15

Criterio di n-copertura dei cicli

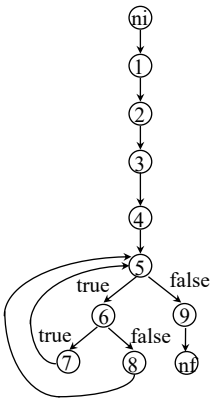
- Si seleziona un insieme di test case che garantisce l'esecuzione dei cammini contenenti un numero di iterazioni di ogni ciclo non superiore ad n (ogni ciclo deve essere eseguito da 0 ad n volte)
- ***Al crescere di n può diventare molto costoso***
  - ✓ Caso pratico n = 2 (ogni ciclo viene eseguito 0 volte, 1 volta , 2 volte)
- NB: il criterio di 1-copertura dei cicli (n = 1) implica il criterio di copertura dei branch

16

16

Esempio: 2-copertura dei cicli

```
void gcd() {
    int x, y, a, b;
1.  cin >> x;
2.  cin >> y;
3.  a = x;
4.  b = y;
5.  while (a != b)
6.        if(a > b)
7.              a = a - b;
8.        else  b = b - a;
9.  cout << y;
}
```



0 volte:
1, 2, 3, 4, 5, 9

1 volta:
1, 2, 3, 4, 5, 6, 7, 5, 9
1, 2, 3, 4, 5, 6, 8, 5, 9

2 volte:
1, 2, 3, 4, 5, 6, 7, 5, 6, 7, 5, 9
1, 2, 3, 4, 5, 6, 7, 5, 6, 8, 5, 9
1, 2, 3, 4, 5, 6, 8, 5, 6, 8, 5, 9
1, 2, 3, 4, 5, 6, 8, 5, 6, 7, 5, 9

fault non individuato con il cammino

17

17

Metodo degli exemplar path

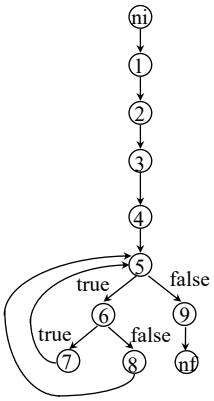
- L'insieme dei cammini del grafo di controllo viene partizionato in un numero finito di classi di equivalenza
  - ✓ Criterio di copertura: un insieme di test case che assicuri l'attraversamento almeno una volta di almeno un cammino per ogni classe
- Metodo degli ***exemplar path*** (cammini esemplari).
  - ✓ Due cammini sono assegnati alla stessa classe se essi differiscono unicamente nel numero di volte per il quale un circuito sul cammino viene percorso

18

18

Metodo degli exemplar-path: esempio

```
void gcd() {
    int x, y, a, b;
1.  cin >> x;
2.  cin >> y;
3.  a = x;
4.  b = y;
5.  while (a != b)
6.        if(a > b)
7.              a = a - b;
8.        else  b = b - a;
9.  cout << y;
}
```



Exemplar Paths

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

NB: Il metodo degli exemplar path implica il metodo di copertura dei branch

19

19

Metodo dei cammini linearmente indipendenti (McCabe)

- Siano n_N ed n_E il numero di nodi e archi di un grafo del flusso di controllo G; un cammino può essere rappresentato come un vettore di n_E elementi (uno per ogni arco); ogni elemento rappresenta il numero di occorrenze (0 o più) dell'arco sul cammino
- Possibile stabilire se un insieme di cammini sono linearmente indipendenti
  - ✓ Un cammino è linearmente indipendente rispetto ad un insieme di cammini se attraversa almeno un arco non ancora percorso
  - ✓ NB: Il metodo di McCabe implica il criterio di copertura dei branch

20

20

Metodo dei cammini linearmente indipendenti (McCabe)

- Un insieme massimale di cammini linearmente indipendenti è detto insieme di cammini di base
  - ✓Tutti gli altri cammini sono generati da una combinazione lineare di quelli di base.
- Dato un programma l'insieme dei cammini di base non è unico.
- Criterio di copertura: un insieme di test case che garantisce l'esecuzione almeno una volta di ogni cammino in un insieme di cammini di base

21

21

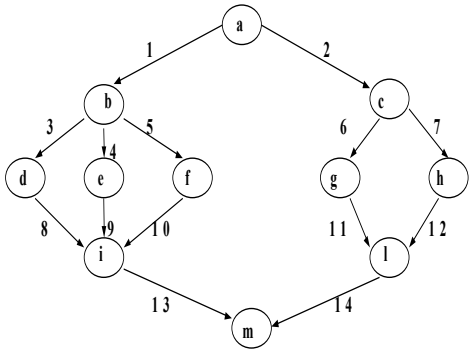
Metodo dei cammini linearmente indipendenti (McCabe)

- Il numero dei cammini linearmente indipendenti di un programma è pari al numero ciclomatico di McCabe
- Sia G il grafo del flusso di controllo di un programma
  - ✓ ***Numero di archi di G: n_E***
  - ✓ ***Numero di nodi di G: n_N***
  - ✓ ***Numero di nodi predicato (decisioni) di G: n_D***
    - » ***Il numero di predicati in G è pari al numero di regioni chiuse in G***
- *Metrica di McCabe (numero ciclomatico):*
  - ✓ ***$v(G) = n_E - n_N + 2 = n_D + 1$***
- N.B. Per nodi predicati a n vie, ogni nodo vale come n-1 nodi a due vie

22

22

Esempio



C1: 1,3,8,13
C2: 1,4,9,13
C3: 1,5,10,13
C4: 2,6,11,14
C5: 2,7,12,14

1) V(G) = E - N + 2 = 14 - 11 + 2 = 5

2) V(G) = P + 1 = 4 + 1 = 5

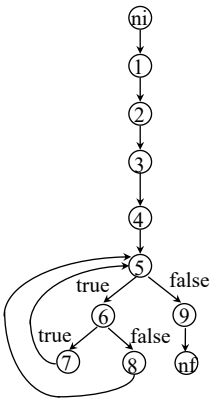
3) V(G) = N.ro regioni chiuse + 1 = 4 +1 = 5

23

23

Esempio: cammini linearmente indipendenti

```
void gcd() {
    int x, y, a, b;
1.  cin >> x;
2.  cin >> y;
3.  a = x;
4.  b = y;
5.  while (a != b)
6.        if(a > b)
7.              a = a - b;
8.        else  b = b - a;
9.  cout << a;
}
```



v(G) = 3

a. 1, 2, 3, 4, 5, 9
b. 1, 2, 3, 4, 5, 6, 7, 5, 9
c. 1, 2, 3, 4, 5, 6, 8, 5, 9

d. 1, 2, 3, 4, 5, 6, 7, 5, 6, 8, 5, 9

d = b + c - a

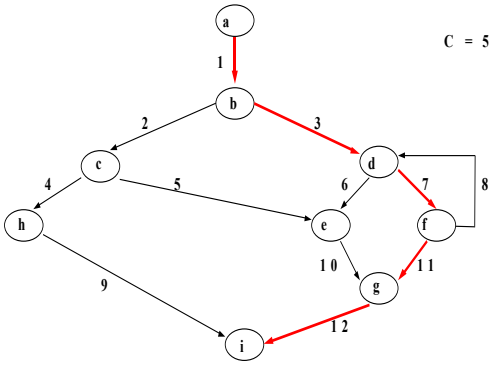
a. (1, 2), (2, 3), (3, 4), (4, 5), (5, 9)
b. (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 5), (5, 9)
c. (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 8), (8, 5), (5, 9)

24

24

Ricerca dei percorsi indipendenti (1)

Passo 1



La ricerca dei cammini di base è effettuata attraverso la tecnica delle ***baseline*** *successive*

Individuare il percorso che contiene il maggior numero di punti di decisione.

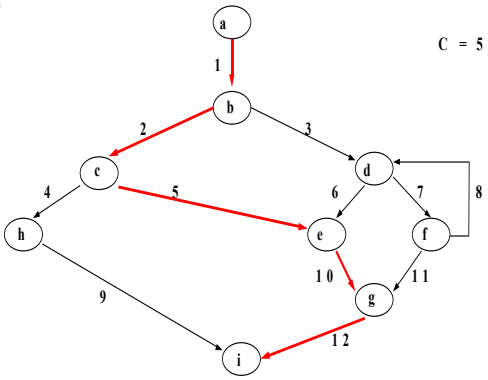
Tale percorso è detto: BASELINE

25

25

Ricerca dei percorsi indipendenti (2)

Passo 2



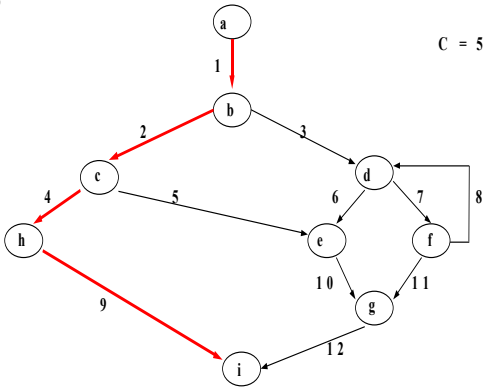
Il secondo percorso è uno di quelli alternativi al primo, secondo il primo punto di decisione della baseline, scegliendo sempre quello con il maggior numero di punti di decisione.

26

26

Ricerca dei percorsi indipendenti (3)

Passo 3



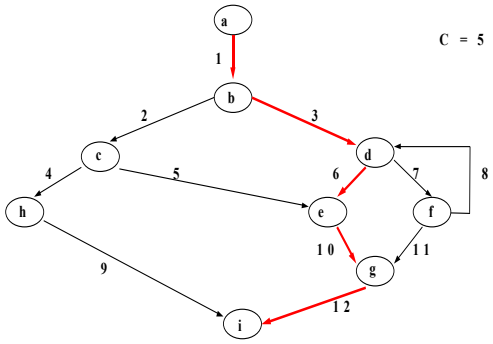
Individuare altri percorsi alternativi, cambiando man mano i punti di decisione appartenenti all'ultimo percorso ricavato.

27

27

Ricerca dei percorsi indipendenti (4)

Passo 4



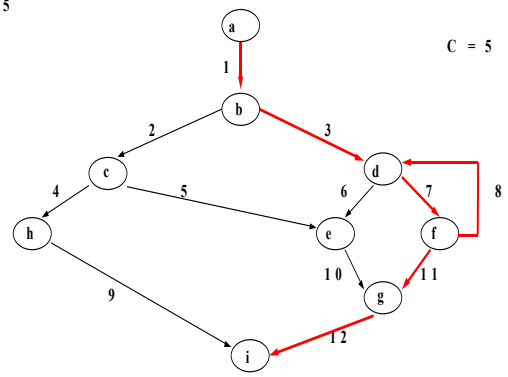
Quando sono terminati i percorsi alternativi per esaurimento dei punti di decisione, si ritorna alla BASELINE, si cambia il percorso nel successivo punto di decisione, mantenendo il maggior numero di punti di decisione.

28

28

Ricerca dei percorsi indipendenti (5)

Passo 5



Iterare il passo 3 e il passo 4 finché tutti i punti di decisione non sono stati esaminati in ogni via o si è raggiunti il numero di percorsi indipendenti calcolato

29

29

Metodi fondati sul data-flow

- Dato un programma P, sia GFC(P) il suo grafo del flusso di controllo.
- un cammino x, n1, n2, ..., nk, y dal nodo x al nodo y è un ***definition clear path*** (*cammino libero da definizioni*) rispetto ad una variabile v, se non ci sono definizioni di v su nessuno dei nodi ni (NB: possono esserci definizioni di v sui nodi x e y)
- un definition clear path x, n1, n2, ..., nk, y dal nodo x al nodo y è un ***definition-use path*** (***du-path*** in breve) rispetto ad una variabile v, se c'è una definizione di v ad x ed un uso di v ad y

30

30

Usi di una variabile v

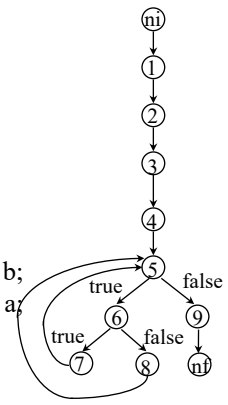
- *Computational use* di v o ***c-use***(v) si ha quando la variabile v è usata in espressioni per il calcolo di un'altra variabile o in istruzioni di output
- *Predicate use* di v o ***p-use***(v) si ha quando la variabile v è usata in un predicato che condiziona il flusso di controllo

31

31

Esempio

```
void gcd() {
    int x, y, a, b;
1.  cin >> x;
2.  cin >> y;
3.  a = x;
4.  b = y;
5.  while (a != b)
6.      if(a > b)
7.          a = a - b;
8.      else  b = b - a;
9.  cout << a;
}
```



Obiettivo: eseguire classi di du-path

def(x) = {1}
c-use(x): {3}

def(y) = {2}
c-use(y) = {4}

def(b) = {4, 8}
p-use(b) = {5, 6}
c-use(b) = {7, 8}

def(a) = {3, 7}
p-use(a) = {5, 6}
c-use(a) = {7, 8}

du-paths:
1, 2, 3
2, 3, 4
3, 4, 5
3, 4, 5, 6
3, 4, 5, 6, 7
…

32

32

Criteri di copertura data-flow (1/2)

all-du-path: deve essere eseguito ogni du-path da ogni definizione ad ogni uso

all-uses: deve essere eseguito almeno un du-path da ogni definizione ad ogni uso

all-p-uses/some-c-uses: deve essere eseguito almeno un du-path da ogni definizione ad ogni p-use; se una definizione non raggiunge nessun p-use, allora va eseguito almeno un du-path dalla definizione ad un c-use

all-c-uses/some p-uses: deve essere eseguito almeno un du-path da ogni definizione ad ogni c-use; se una definizione non raggiunge nessun c-use, allora va eseguito almeno un du-path dalla definizione ad un p-use

33

33

Criteri di copertura dat-flow (2/2)

all-c-uses: deve essere eseguito almeno un du-path da ogni definizione ad ogni suo c-use

all-p-uses: deve essere eseguito almeno un du-path da ogni definizione ad ogni suo p-use

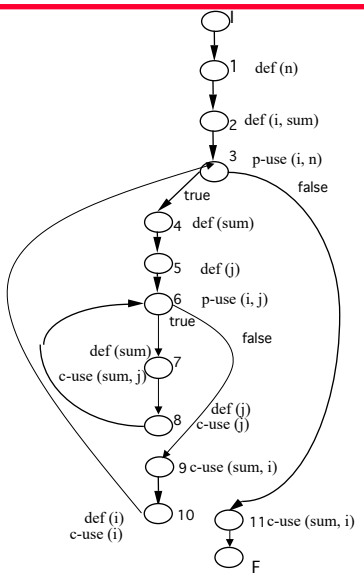
all-defs: almeno un du-path da ogni definizione ad un suo uso (p-use o c-use)

34

34

Esempio (1)

```
void sums() {
   int n, i, sum,
1   cin >> n;
2   i = 1; sum = 0;
3   while (i<=n) {
4     sum= 0;
5     j = 1;
6     while (j <= i) {
7         sum = sum + j;
8         j = j + 1;
      }
9     cout << sum << i;
10    i = i + 1;
    }
11 cout << sum << i;
}
```



Esempio (2)

Node	def	c-use
1	n	-
2	i,sum	-
3	-	-
4	sum	-
5	j	-
6	-	-
7	sum	(sum, j)
8	j	j
9	-	(sum,i)
10	i	i
11	-	(sum,i)

edge	p-use
(3,4)	(i,n)
(3,11)	(i,n)
(6,7)	(j,i)
(6,9)	(i,j)

```
void sums() {
   int n, i, sum,
1   cin >> n;
2   i = 1; sum = 0;
3   while (i<=n) {
4     sum= 0;
5     j = 1;
6     while (j <= i) {
7         sum = sum + j;
8         j = j + 1;
      }
9     cout << sum << i;
10    i = i + 1;
    }
11 cout << sum << i;
}
```

Esempio (3)

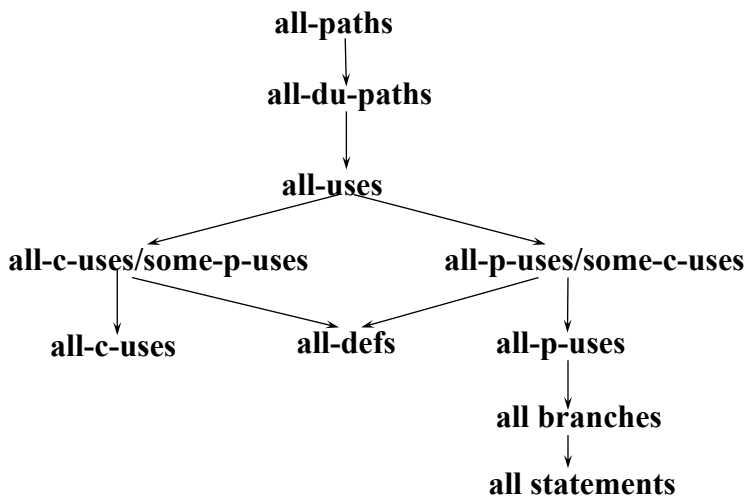
Node	def-c-use triples	def-p-use triples
1		(1,(3,4),n) (1,(3,11),n)
2	(2,9,sum) (2,10,i) (2,11,i) (2,11,sum)	(2,(3,4),i) (2,(3,11),i) (2,(6,7),i) (2,(6,9),i)
4	(4,7,sum) (4,9,sum)* (4,11,sum)*	
5	(5,7,j) (5,8,j)	(5,(6,7),j) (5,(6,9),j)*
7	(7,9,sum) (7,11,sum) (7,7,sum)	
8	(8,8,j) (8,7,j)	(8,(6,7),j) (8,(6,9),j)
10	(10,9,i) (10,10,i) (10,11,i)	(10,(3,4),i) (10,(3,11),i) (10,(6,7),i) (10,(6,9),i)

* ***infeasible triple***

Esempio: Test-cases per all-uses

Test	n	sum	i	execution trace
1	0	0	1	I,1,2,3,11,F
2	1	1	1	I,1,2,3,4,5,6,7,8,6,9,10,3,11,F
		1	2	
3	3	1	1	I,1,2,3,4,5,6,7,8,6,9,10,3,4,5,$(6,7,8)^2$,6,9,10,3, 4,5,$(6,7,8)^3$,6,9,10,3,11,F
		3	2	
		6	3	
		6	4	

Gerarchia di implicazioni tra criteri di copertura



39

39

Criteri strutturali e livelli di testing

- La selezione di casi di test mediante criteri di copertura strutturali è principalmente indicata per il testing di unità
- Criteri basati sui cammini sono poco scalabili
- Nella pratica, test selezionati con criteri funzionali vengono usati per controllare su scala più larga livelli di copertura strutturale
  - ✓principalmente statement e branch coverage

40

40

gcov

- **gcov** è uno strumento per il controllo della copertura di branch e statements in un programma C
- Più in generale, può essere considerato uno strumento di code profiling
  - ✓ Quanto spesso una linea di codice è stata eseguita
  - ✓ Quale il tempo di esecuzione (cumulativo) per quella linea
- Esso è fornito a corredo del compilatore **gcc**

41

41

gcov – Istruzioni per l'uso (1)

- Compilare il programma utilizzando i switch -fprofile-arcs -ftest-coverage di gcc
  - ✓ gcc -fprofile-arcs -ftest-coverage myprog.c –o myprog
- Eseguire il programma passandogli in input i casi di test prodotti
  - ✓ ./myprog

42

42

gcov – Istruzioni per l'uso (2)

- Eseguire gcov
  - ✓gcov myprog.c

    90.91% of 11 source lines executed in file myprog.c
    Creating myprog.c.gcov.

- Esaminare l'output contenuto in myprog.c.gcov

43

43

Esaminiamo il file

```
-:
        #include<stdio.h>
                int main()
            1    {
            1      int j,x,y,out;
            1      out=0;
            1      scanf("%d",&x);
            1      scanf("%d",&y);
            4      for(j=0;j<x;j++)
                    {
            3         if(j>y)
       ######          printf("Inside\n");
                      else
            3           out++;
                    }
            1      if(out)
            1        printf("Out is %d\n",out);
                }
```

44

44

gcov – branch coverage

- Quanto visto in precedenza è utile per esaminare lo statement coverage
- Per visualizzare il branch coverage, occorre eseguire gcov con l'opzione –b
- gcov -b myprog.c

```
90.91% of 11 source lines executed in file myprog.c
83.33% of 6 branches executed in file myprog.c
66.67% of 6 branches taken at least once in file myprog.c
80.00% of 5 calls executed in file myprog.c
Creating myprog.c.gcov.
```

45

45

branch coverage – gcov output (1)

```
       #include<stdio.h>
       int main()
            1    {
call 0 returns = 100%
            1      int j,x,y,out;
            1      out=0;
            1      scanf("%d",&x);
call 0 returns = 100%
            1      scanf("%d",&y);
call 0 returns = 100%
            4      for(j=0;j<x;j++)
branch 0 taken = 75%
branch 1 taken = 100%
branch 2 taken = 100%
```

46

46

branch coverage – gcov output (2)

```
            {
            3           if(j>y)
branch 0 taken = 100%
      ######        printf("Inside\n");
call 0 never executed
branch 1 never executed
              else
            3         out++;
            }
            1       if(out)
branch 0 taken = 0%
            1         printf("Out is %d\n",out);
call 0 returns = 100%
       }
```

47

47

Note

- Il conteggio delle esecuzioni e il computo della copertura è cumulativo rispetto alle successive esecuzioni del programma
- Le tracce di esecuzione sono mantenute nel file **.da**

48

48