Testing basato sui programmi

Angelo Gargantini

Definizioni di base del testing

Un programma P è una funzione da un dominio D ad un codominio R:

$$P: D \rightarrow R$$

può essere parziale – non definita per qualche d in D

introduciamo un predicato OK:

OK(P,d) con d in D, se P è corretto per l'input d, cioè se produce P(d) corretto

OK(P) se P è corretto, se per ogni d in D **OK**(P,d)

Definizioni (2)

Rivediamo alcune definizioni:

P sia il programma corretto

FAILURE o malfunzionamento:

- Eseguo P con d e non ottengo P(d): o ottengo un altro valore P'(d) oppure il programma si blocca, …
- cioè not OK(P,d)

FAULT o difetto o bug:

- P è stato implementato in P'
- Se P' ≠ P, la diversità è il difetto

ERRORE:

- Il motivo del difetto

Definizioni (3)

Un caso di test (test case) o test è un elemento di D

- **esempio**: se D è gli interi, un test è un intero, se D è sequenze di interi, un test sarà un sequenza (possibilmente finita) di interi, ecc ...

Un test set T o test suite è un sotto insieme (finito) di D

- **esempio**: D una coppia di interi (= int x int), T sarà un sottoinsieme di D, cioè un insieme di coppie di interi: $T \subseteq$ int x int

Applico il test set T al programma P:

OK(P,T) se per ogni t in T, OK(P,t): il test set T non ha scoperto errori (negativo)

se t in T tale che not OK(P,t): test positivo

Esempio

dato il metodo – programma P

```
static void foo(String s, int x)
```

un caso di test è una coppia String e int

- esempio [“pippo”,3]

un test set è un insieme di coppie String e int

- esempi:
  - {[“pippo”,3]} : insieme con un solo caso di test
  - con tre casi di test:
    {[“pippo”,3], [“”,54657], [“ccc”,-10]}

Attenzione: non posso dire s !=”pippo” e x > 0

Parte I: grafo di flusso

Testing strutturale dei programmi

Nel testing basato sulla struttura dei programmi:

- i criteri di test sono definiti considerando il solo codice sorgente
  - Il sorgente è una importante fonte di informazione anche per il testing
  - Si ignora la specifica del programma (non si tiene conto cioè quello che deve effettivamente fare il programma)
- il sorgente viene usato per
  - generare i casi di test
  - decidere quando si ha testato abbastanza

Copertura e flusso di controllo

- I criteri di test strutturali sono definiti considerando la "copertura" del codice

> Per **copertura** di un programma si intende la parte del programma che viene eseguita dai casi di test

- In particolare intendiamo la struttura del codice come flusso di controllo del programma
  - la copertura sarà relativa al flusso di controllo

Flusso di controllo

Il comportamento di un programma può essere rappresentato visualizzando il suo grafo del flusso di controllo

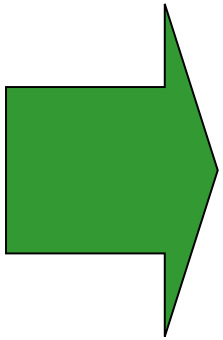
- il grafo del flusso di controllo è una rappresentazione grafica del codice sorgente
- rappresenta qualsiasi esecuzione possibile
  - ignora i particolari valori di variabili che possono variare da una esecuzione a quella successiva
- ogni istruzione è un nodo del grafo
- ogni istruzione è collegata alla sua successiva mediante una freccia

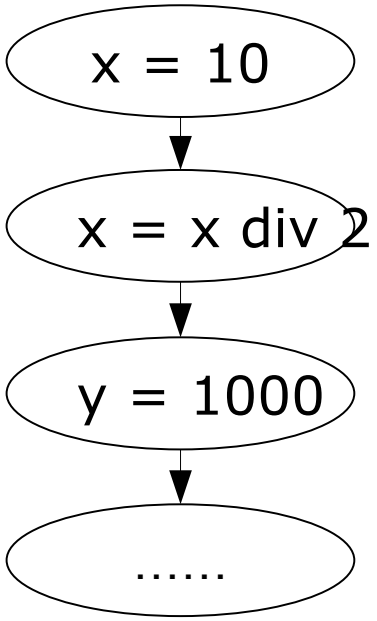
Flusso di controllo: esempio (1)

Nel caso di una semplice sequenza di istruzioni si ha un grafo semplice e intuitivo come quello dell'esempio:

```
x e y sono interi

x = 10;
x = x div 2;
y = 1000;
……
```



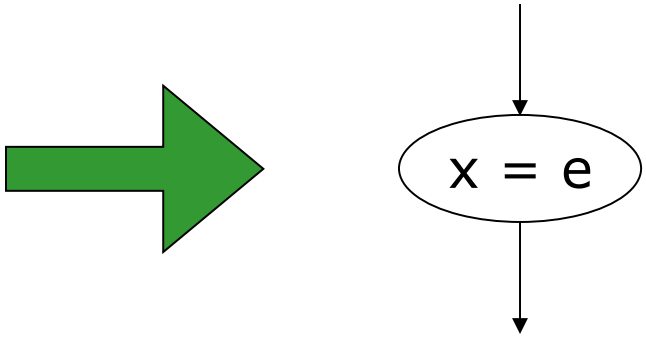


Istruzioni semplici

Le istruzioni di assegnamento, di lettura, scrittura e di return li rappresentiamo con nodi circolari

Istruzione di assegnamento

```
x = e;
```



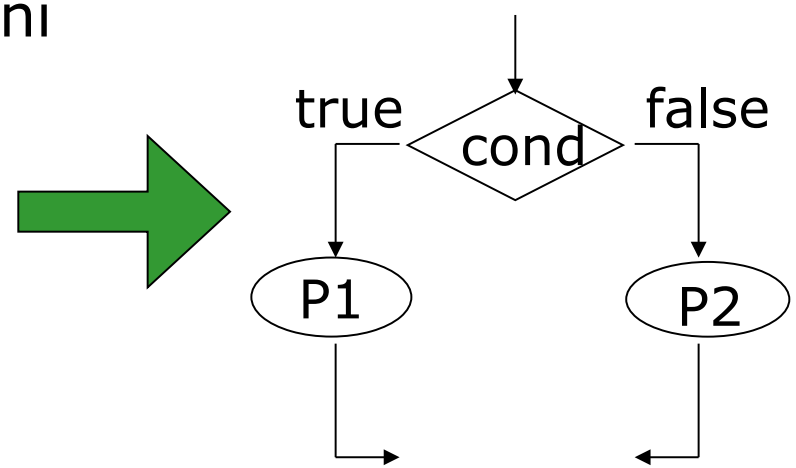
Anche chiamate di metodi esterni verranno rappresentate con semplici nodi circolari

Nodi Decisione – Istruzione If

Le istruzioni condizionali vengono rappresentate:

- da un rombo che contiene la decisione
- due cerchi con le due istruzioni (il ramo else è opzionale)
- due frecce marcate true e false che vanno dalla decisione alle due istruzioni

```
if (cond) then P1 else P2
```



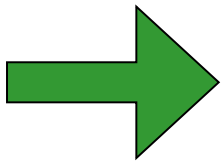
rappresenteremo le decisioni sempre con dei rombi

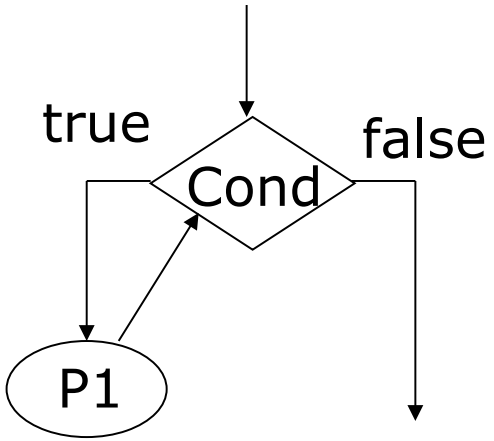
Ciclo While

I cicli while vengono rappresentati da:

- una decisione che è la condizione del while
  - con una freccia "false" che va all'istruzione successiva
  - con una freccia "true" che va alle:
- istruzioni del corpo del while
  - da eseguire se la decisione è vera
- finite le istruzioni del corpo si ritorna alla valutazione della decisione

```
while (cond) P1
```

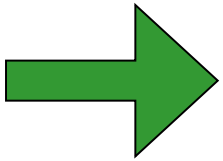


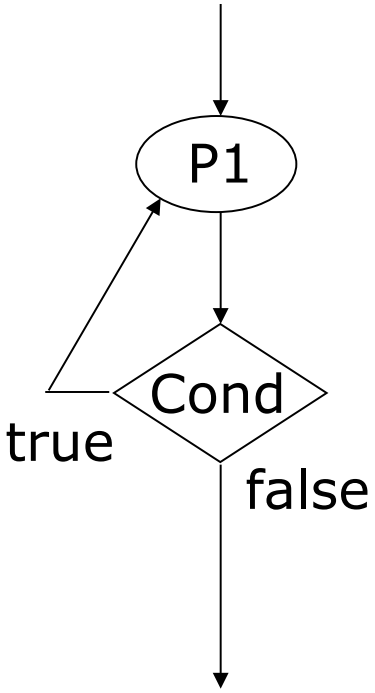


Ciclo Do-While

I cicli do-while vengono rappresentati come il while ma la decisione è posta alla fine

```
do P1
while (cond)
```



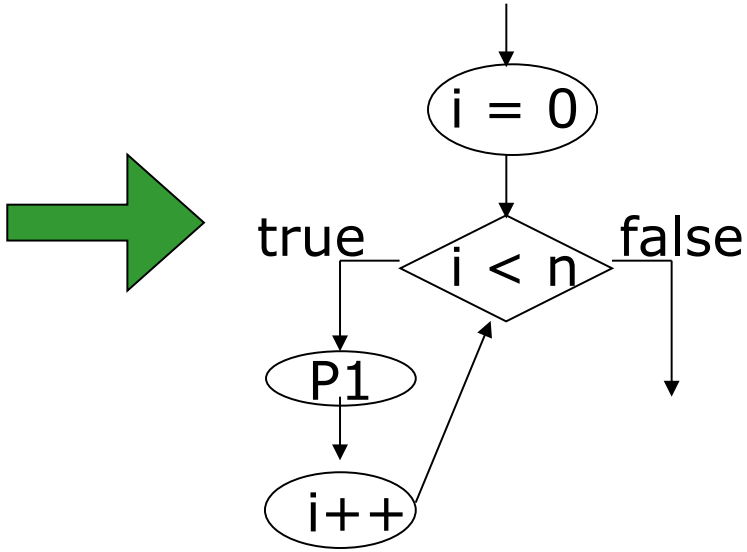


Cicli For

I cicli for sono rappresentati con il loro equivalente ciclo while

- l'istruzione di inizializzazione viene messa per prima
- la decisione è la condizione del ciclo for
- l'incremento viene eseguito come ultima istruzione del corpo
- dopo l'incremento si torna a valutare la decisione

```
for (i = 0; i < n; i++) P1
```

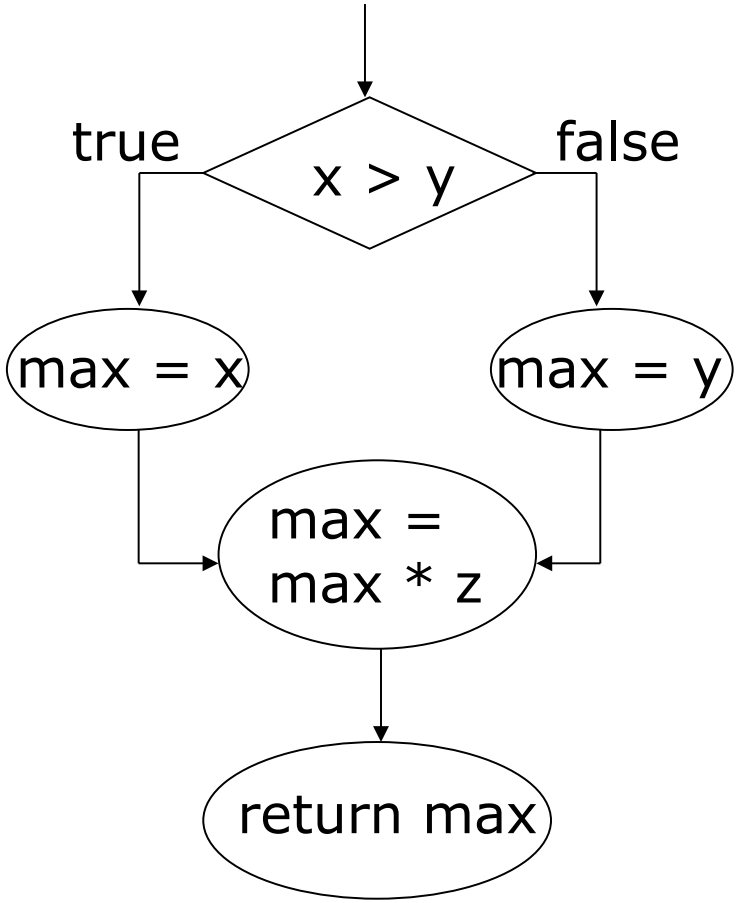


Flusso di Controllo: esempio (2)

Esempio di metodo main e suo grafo del flusso di controllo

```
int main(int x, y, z) {
    int max;

    if (x > y)
        max = x;
    else
        max = y;
    max = max * z;
    return max;
}
```

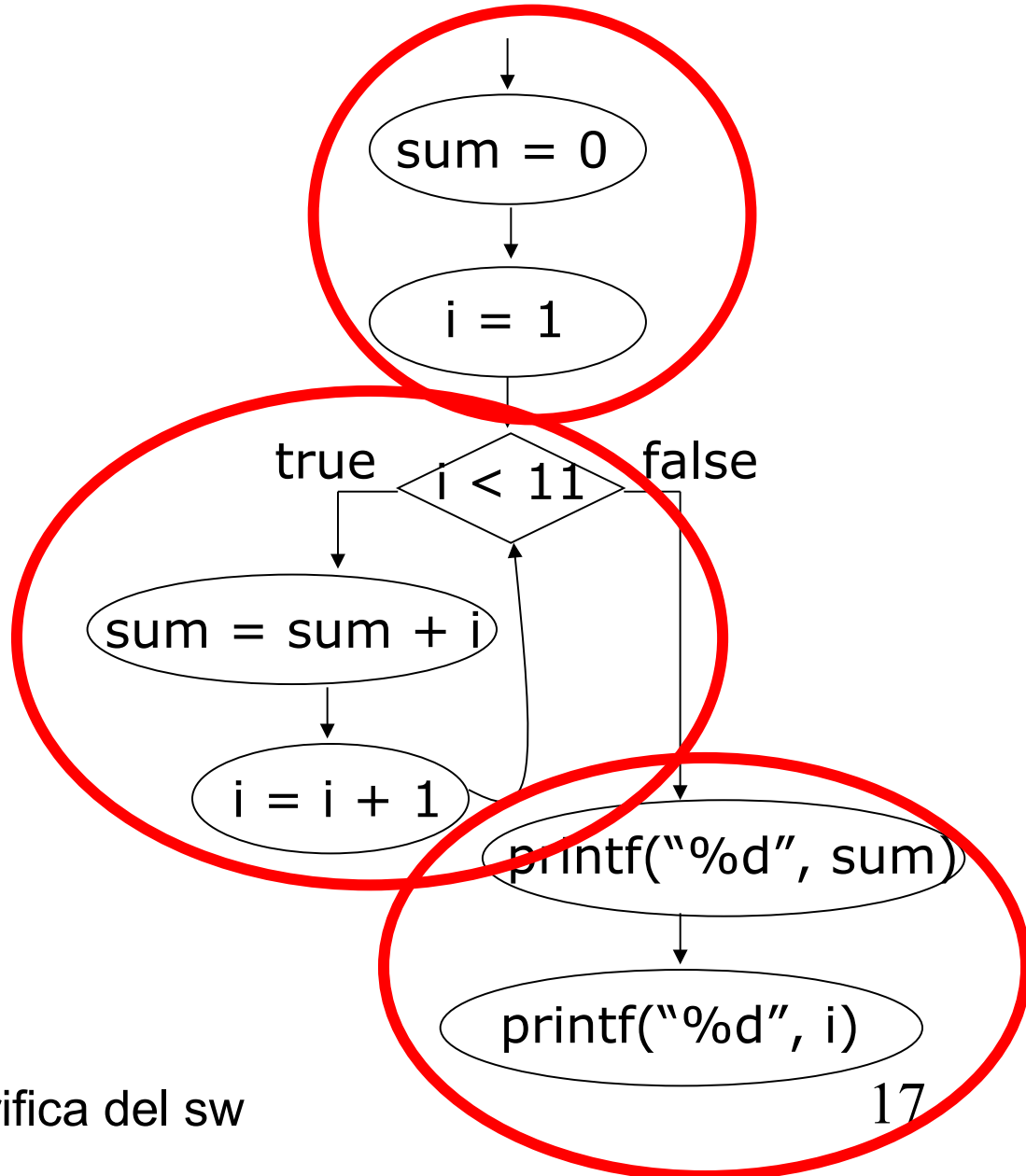


Flusso di controllo: esempio (3)

Esempio con while

```
int main() {
   int sum, I;

   sum = 0;
   i = 1;
   while (i < 11) {
         sum = sum + i;
         i = i + 1;
   }
   printf("%d",sum);
   printf("%d",i);
}
```



Il grafo di flusso di controllo come astrazione

Il flusso di controllo può essere considerato un'astrazione

- ignora il particolare cammino che verrà preso per certi valori delle variabili
- rappresenta tutte le possibili esecuzioni
- ad ogni esecuzione verrà eseguita un sequenza di nodi
  - che possiamo disegnare con una linea nel grafo
  - per ogni computazione di un programma c'è la corrispondente computazione nel grafo del flusso di controllo

Esempio di esecuzione e sua rappresentazione nel grafo

etichette

```
    int x, y;
P1:         while (even x)
P2:               x  = x div 2;
P3:         y = 1000;
P4:         ……
```

Esecuzione con x = 3, y = 1

P1 <3,1>

P3 <3,1>

P4 <3,1000>

Esecuzione con x = 6, y = 1

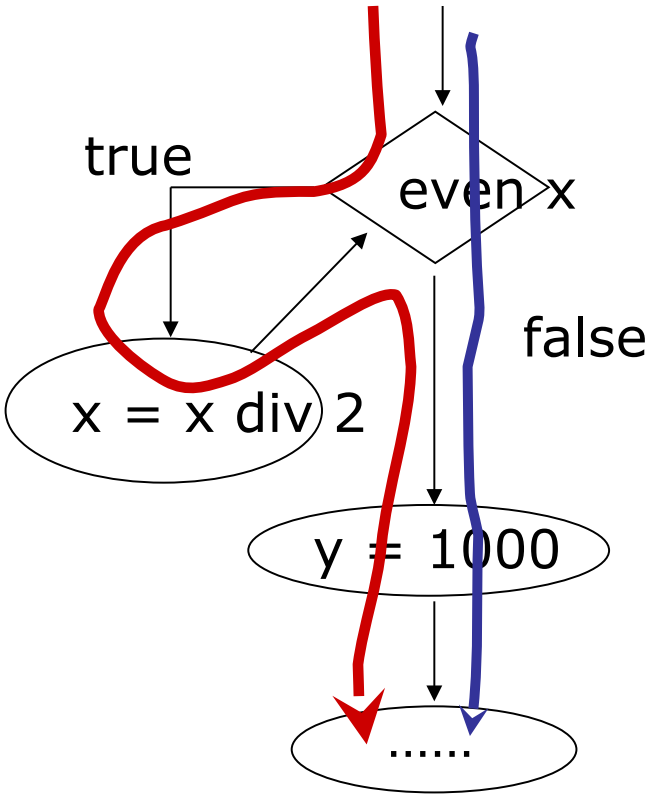
P1 <6,1>

P2 <6,1>

P1 <3,1>

P3 <3,1>

P4 <3,1000>



In sintesi

- Abbiamo visto:
  - che nel testing strutturale si considera la struttura di un programma, cioè il suo grafo di flusso
  - come costruire il grafo di flusso di programmi
- Ricordate che:
  - i nodi sono collegati da frecce (che puntano all'istruzione successiva)
    - Le istruzioni sono semplici nodi rappresentate con dei cerchi
    - Le decisioni sono rappresentate con rombi e due uscite
    - Anche i cicli possono essere rappresentati
- Ad ogni esecuzione corrisponde un cammino nel grafo

Parte II Copertura istruzioni

Criteri di test strutturali

Ricorda che criterio: P x T → {true,false}

Vedremo i seguenti criteri di test basati sulla copertura dei programmi:

1. **copertura delle istruzioni (statement coverage)**
2. **copertura degli archi (branch coverage)**

Statement coverage

Un test set T e' adeguato per testare un programma P secondo il criterio di copertura delle istruzioni, se per ogni istruzione s di P esiste un caso di test in T che esegue s

Ogni istruzione viene eseguita almeno una volta

Esempio:

- `if x != 0 then x:=x+10 else x:=x-10`

Perchè T possa soddisfare la copertura delle istruzioni deve contenere almeno due casi, uno con x = 0 e uno con x /= 0, ad esempio

T = { x = 0, x = 10}

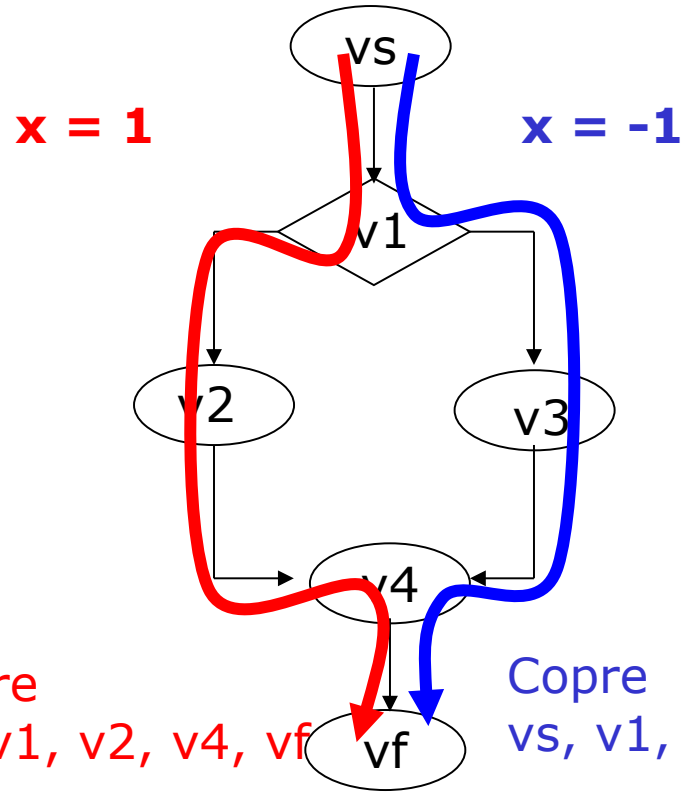
Statement Coverage – Esempio (1)

Istruzioni

- vs, v1, v2, v3, v4, vf

```
vs: int foo(int x) {
      int result;

v1:   if (x > 0)
v2:       result = x;
      else
v3:       result = 1 / x;
v4:   printf("%d", result);
vf: }
```

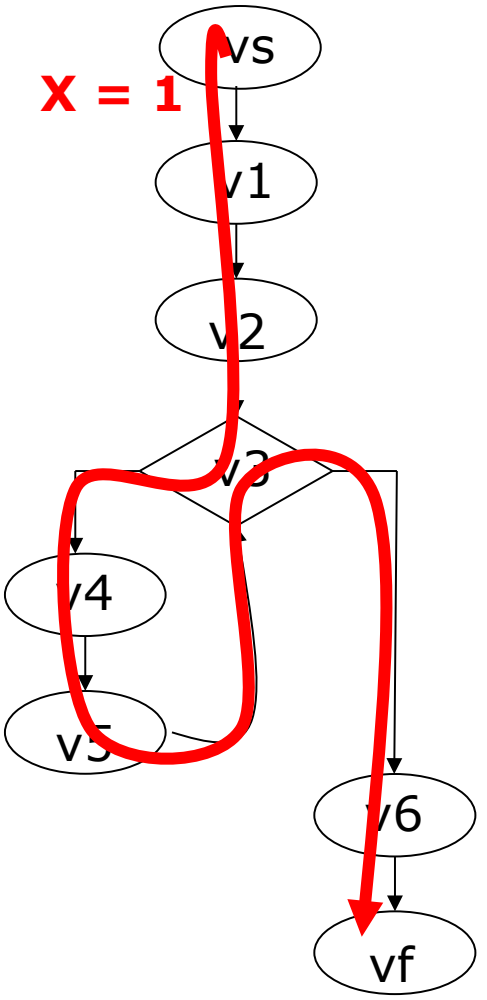


Statement coverage – Esempio (2)

Istruzioni

- vs, v1, v2, v3, v4, v5, v6, vf

```
vs:  int main(int x) {
       int sum, I;
v1:    sum = 0;
v2:    i = 1;
v3:    while (i <= x) {
v4:      sum = sum + i;
v5:      i = i + 1;
       }
v6:    printf("%d",sum);
vf:  }
```



Fault detection capability

Come valutare un criterio di copertura?

- Fault detection capability: quali errori è in grado di trovare e (più importante) quali garantisce di trovare

Statement coverage:

istruzioni (sempre) errate vengono individuate

- gli errori nelle decisioni non è detto che vengono trovati
- questo criterio e' debole
- però potrebbe anche bastare: idea di "MINIMAL CRITERION FOR COMMERCIAL SOFTWARE"
  - il vostro sw quanto è testato?

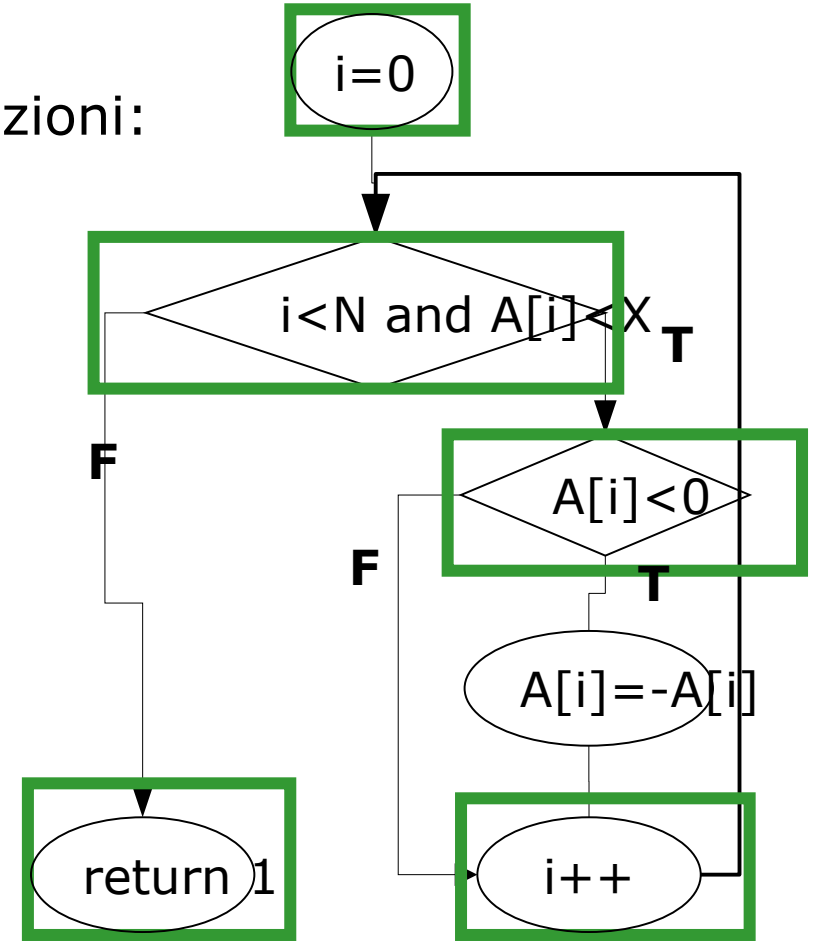
Misura di copertura dei comandi

Un caso di test (o una test suite) può soddisfare parzialmente un criterio di copertura

Il criterio si usa come **misura**

Per il criterio di copertura delle istruzioni:

$$C_0 = \frac{\text{n. di statement eseguiti}}{\text{n. di st. eseguibili}}$$



Il caso di test:
(N=1, A[0]=7, X=9)

copre 5/6

Branch coverage

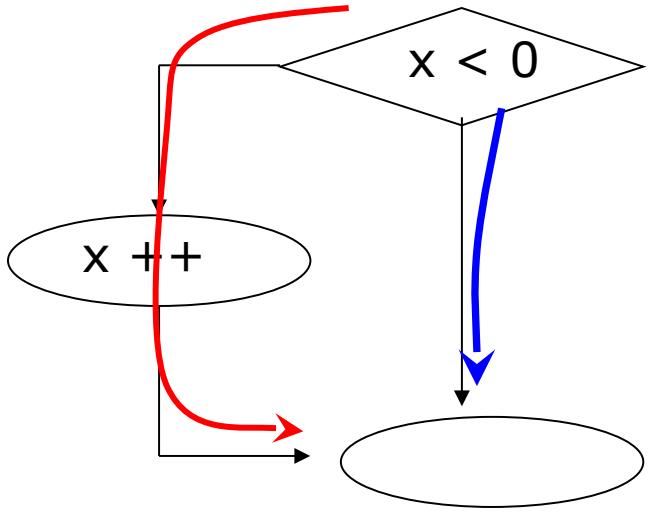
Un test set T soddisfa il criterio di copertura degli archi (o branch coverage) di P se e solo se ogni arco (branch) del grafo di P è percorso almeno una volta

Esempio:

```
if x < 0 then x ++ endif
```

Per ottenere la copertura dei branch ho bisogno di almeno due casi, uno con x < 0 e uno con x >= 0, ad esempio

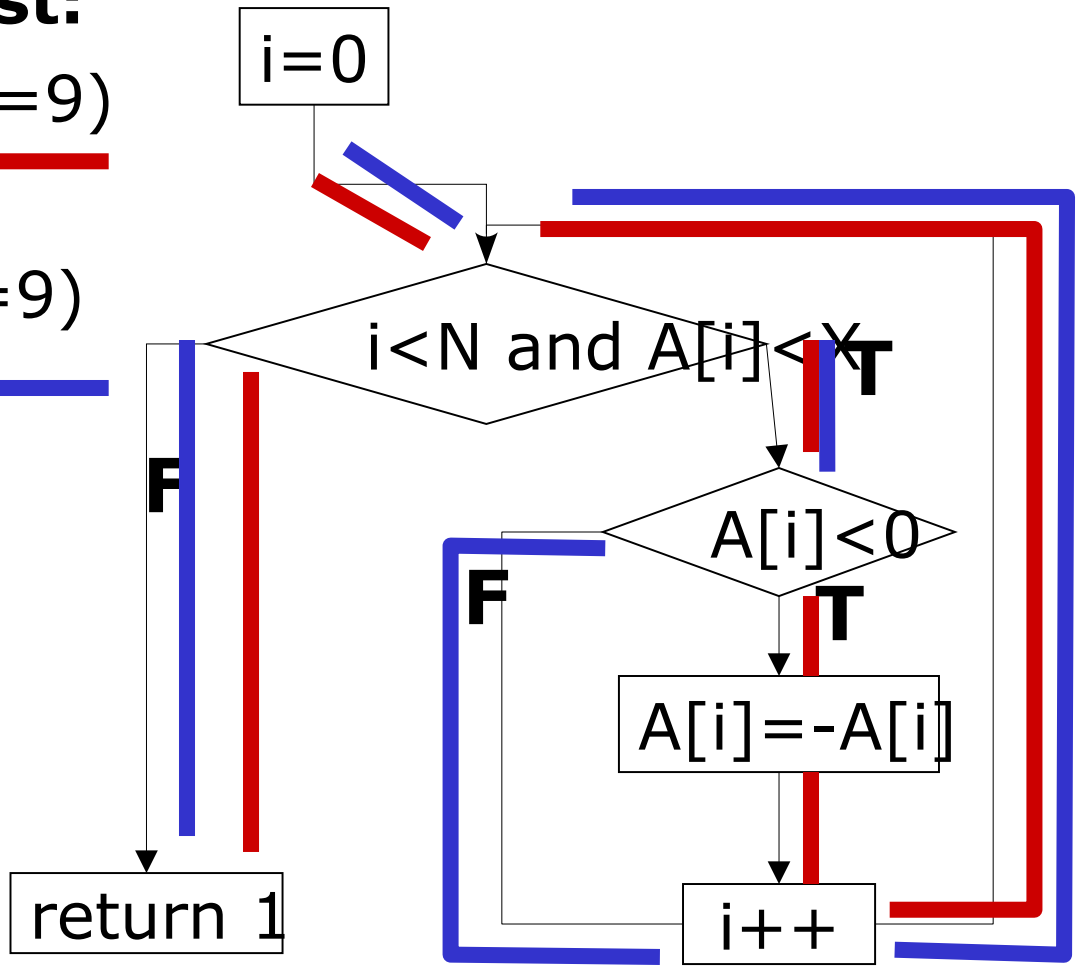
T = { x = -3, x = 10}



Branch coverage - Esempio

Abbiamo 2 casi di test:

- (N=1, A[0]=-7, X=9)
- (N=1, A[0]=7, X=9)

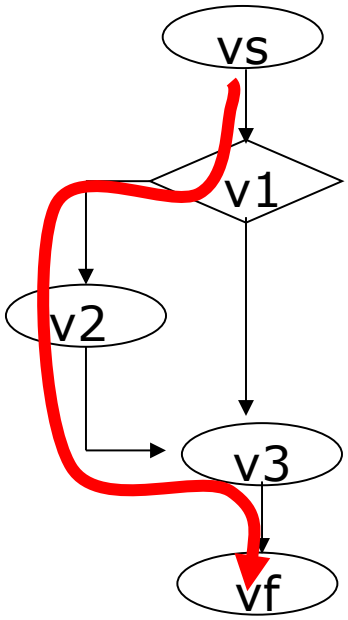


Differenza tra Statement e Branch coverage

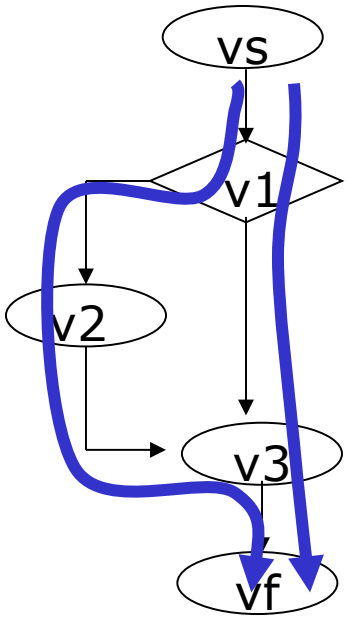
Statement coverage è più debole

Il branch coverage implica lo statement cov.

- se un test set T soddisfa il branch coverage, soddisfa anche il statement coverage
- ma non viceversa



Statement coverage

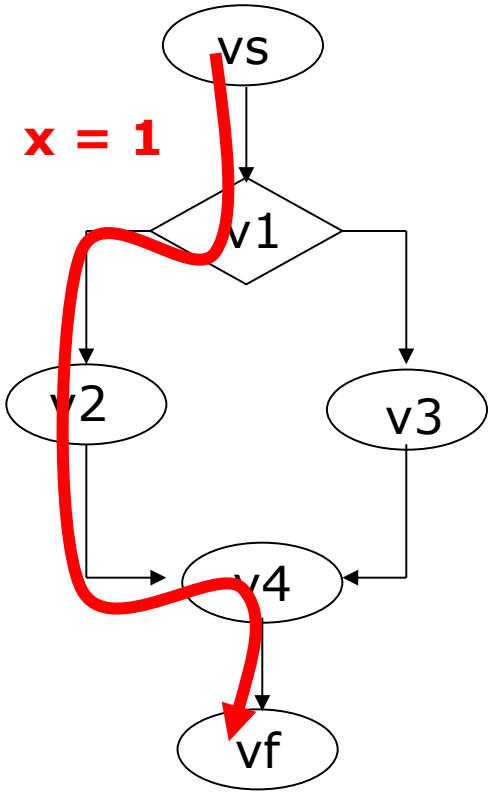


Branch coverage

Misura di copertura degli archi

Un test T può soddisfare il criterio degli archi parzialmente. In quel caso si misura la copertura:

$$C_{path} = \frac{\text{numero di branch eseguiti}}{\text{numero di branch eseguibili}}$$



Esempio: con x = 1 si copre un arco su i due eseguibili, ottenendo la copertura del 50%

Generazione dei casi di test

Come generare un test set per un certo programma e un certo criterio:

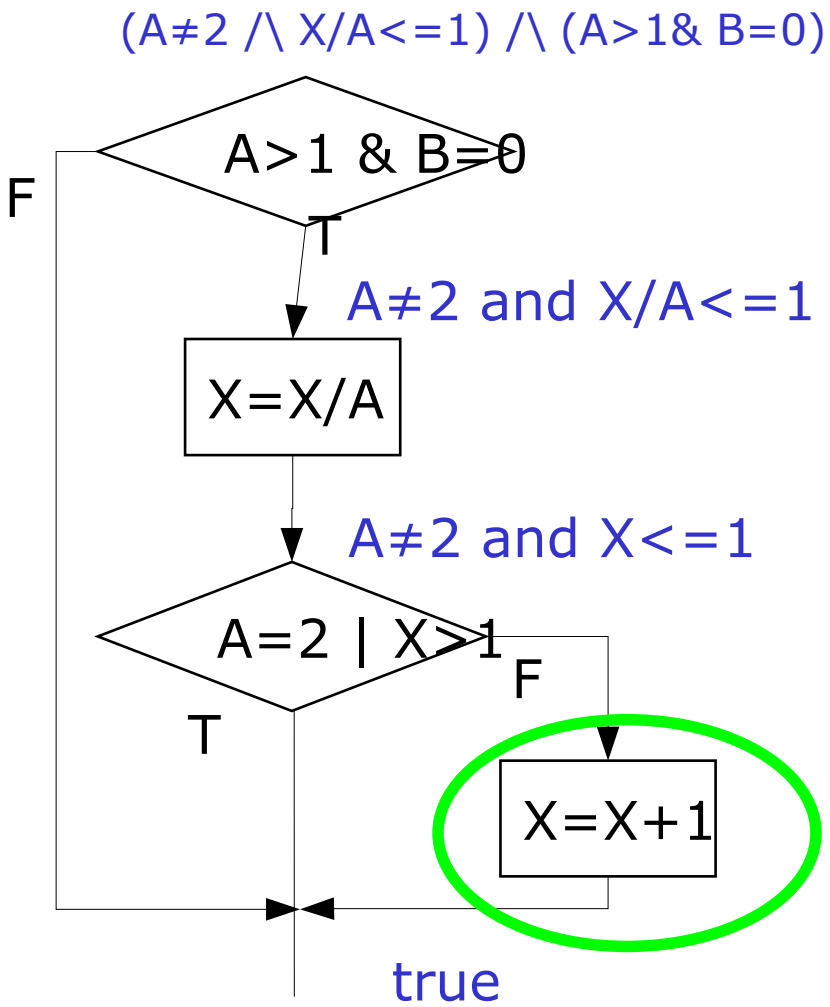
1. costruisci il grafo di flusso del programma
2. cerca i percorsi nel grafo che devono essere eseguiti per soddisfare quel criterio
3. per ogni percorso cerca i valori di input che inducono quel percorso
4. ottimizza il test suite se possibile eliminando casi di test che non siano necessari (coprono casi coperti da altri test)

Il più delle volte il passo 3 è fatto in modo intuitivo, puoi anche procedere così:

Come trovo i valori per una copertura?

- metti true alla fine del cammino che vuoi coprire
- procedi all'indietro sul cammino
- se c'è un assegnamento, sostituisci la variabile assegnata con il valore che è stato assegnato
- se c'è una ramo "T" in una decisione aggiungi la decisione alla condizione
- se c'è un "F", aggiungi la negazione

Nell'esempio voglio coprire l'istruzione x= x+1



Generazione di test

Alcune volte la condizione che si deve soddisfare per ottenere una certa copertura è complessa

- potrebbe anche non essere risolvibile algoritmicamente (OK: ricordati che è un problema non computabile)
- potrebbe anche non essere soddisfacibile
  - es.: x >0 and x <0
  - in questo caso l'istruzione o l'arco non è copribile perchè il codice non è raggiungibile: potrebbe essere un errore nel codice

In sintesi

- Abbiamo visto:
  - il criterio di copertura delle istruzioni richiede la copertura di tutte le istruzioni
  - il criterio di copertura dei branch richiede la copertura di tutti gli spigoli del grafo
- Ricordate che:
  - la copertura dei branch implica la copertura delle istruzioni
  - ma non viceversa
- E abbiamo imparato
  - a generare un caso di test per una certa copertura

Parte III: copertura Decisioni e Condizioni

Criteri di test strutturali

Abbiamo visto i criteri di copertura:

1. **delle istruzioni**
   - tutte le istruzioni vengono eseguite
1. **dei branch**
   - tutti gli archi nel grafo di flusso vengono percorsi

Vedremo ora

1. **delle decisioni**
2. **delle condizioni**

Decisioni e Condizioni (1)

Decisioni

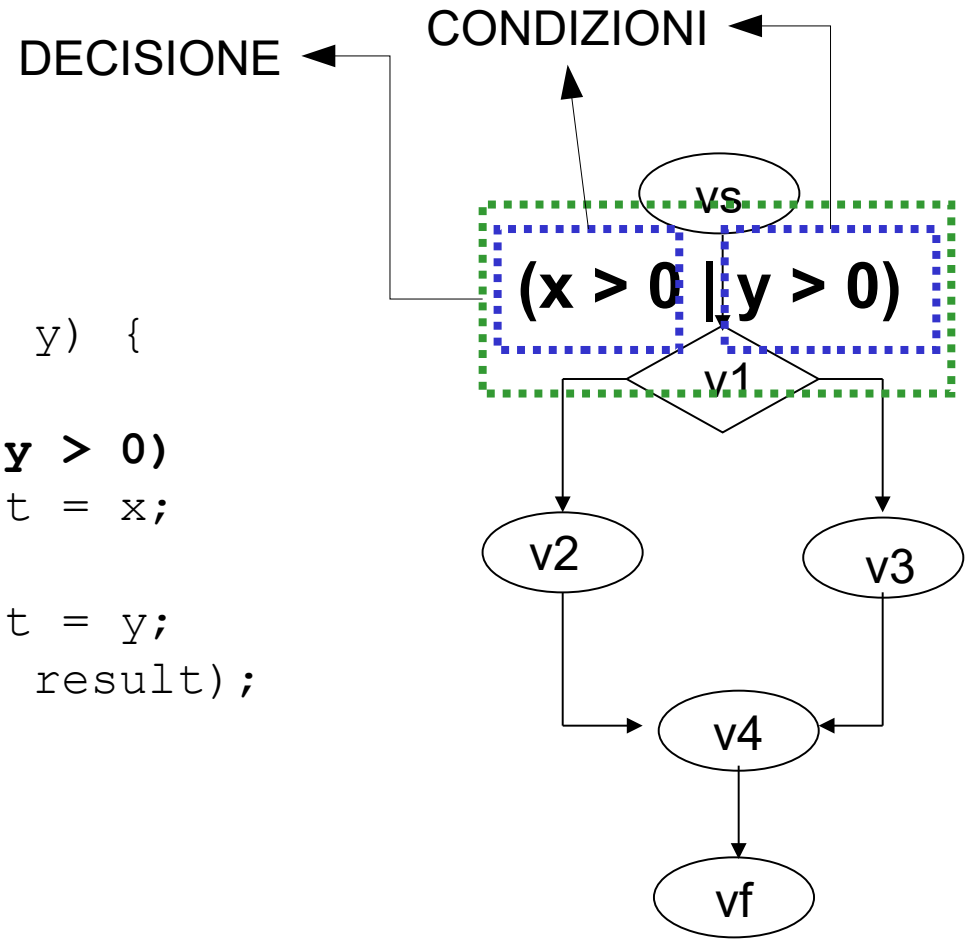
- predicato (espressione booleana) guardia di una istruzione condizionale (if) o di una iterativa (while, for, …)
- **esempio** `if (x > 0 | y > 0)` **...**

Condizioni

- una espressione booleana atomica (cioè non divisibile in altre espressioni più semplici) che appare in una decisione
- **nell'esempio sopra** `x > 0` e `y > 0`

Decisioni e Condizioni (2)

```
vs: int main(int x, y) {
        int result;
v1:     if (x > 0 | y > 0)
v2:             result = x;
        else
v3:             result = y;
v4:     printf("%d", result);
vf:   }
```



Copertura delle decisioni

Un test set T e' adeguato per testare un programma P secondo il criterio di copertura delle decisioni, se per ogni decisione di P esiste

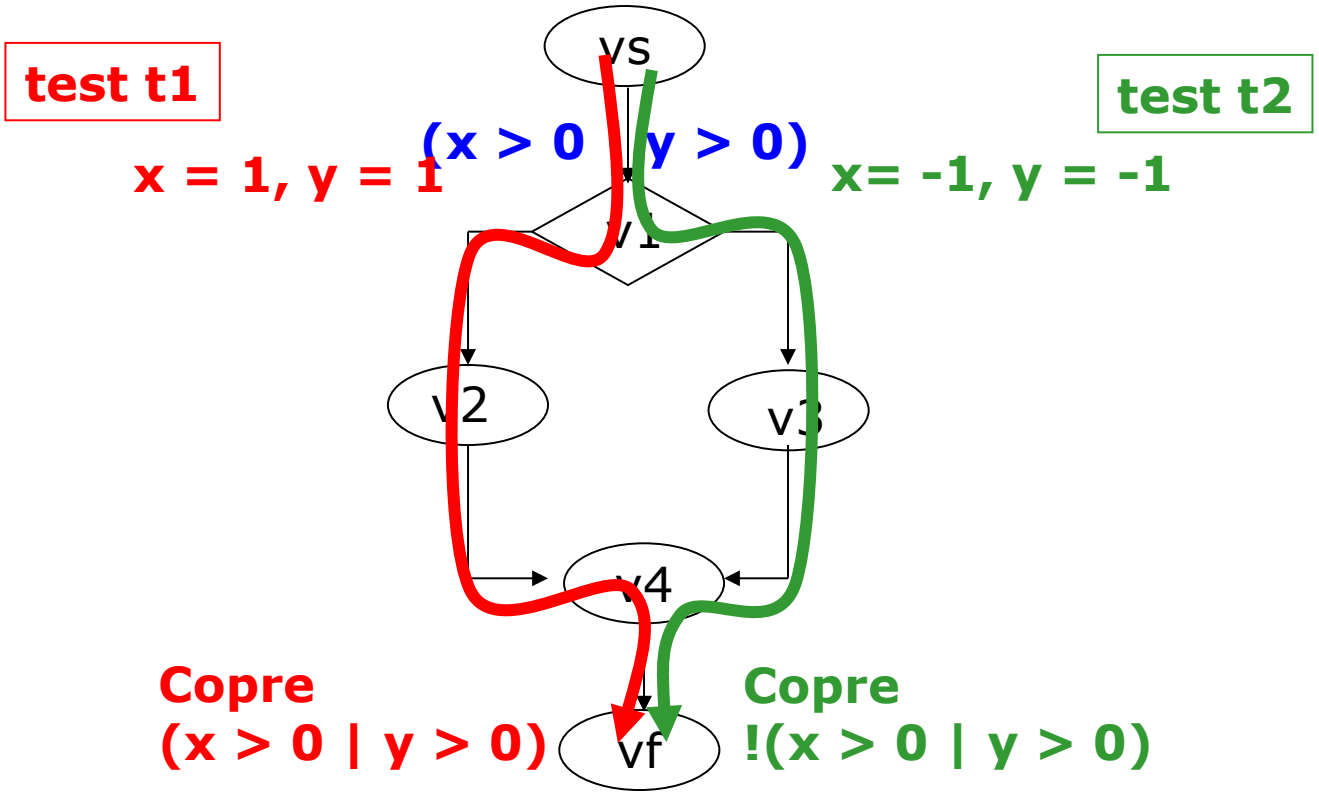
- un caso di test in T in cui la decisione è presa
- un caso di test in T in cui la decisione non è presa

Copre ogni decisione e la sua negazione

E' equivalente al branch coverage

- ogni arco nel grafo di flusso viene percorso

Esempio copertura delle decisioni (1)



Copertura delle condizioni

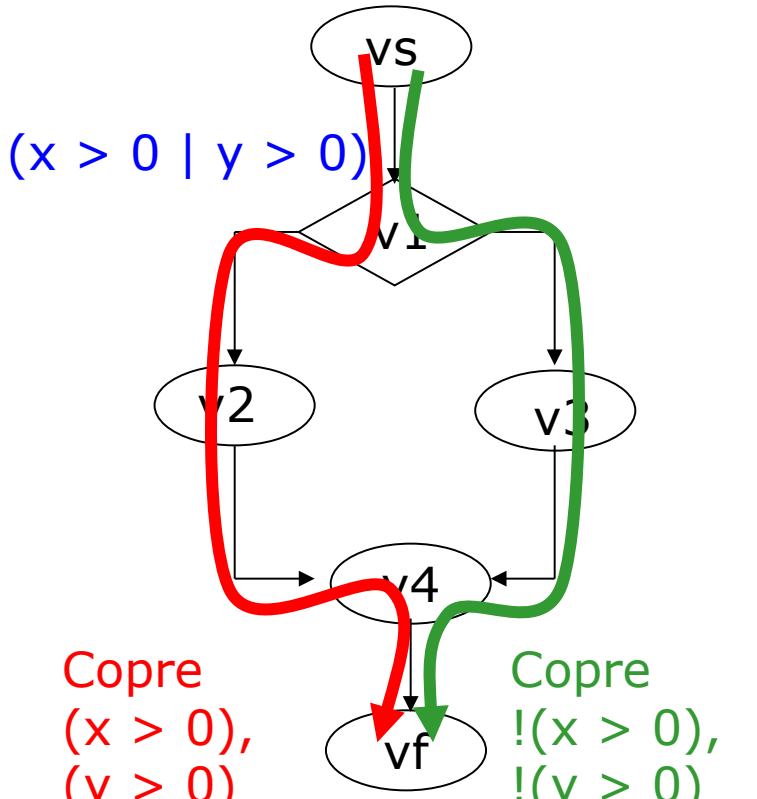
Un test set T e' adeguato per testare un programma P secondo il criterio di copertura delle condizioni, se per ogni condizione di P esiste

- un caso di test in T in cui la condizione è vera
- un caso di test in T in cui la condizione è falsa

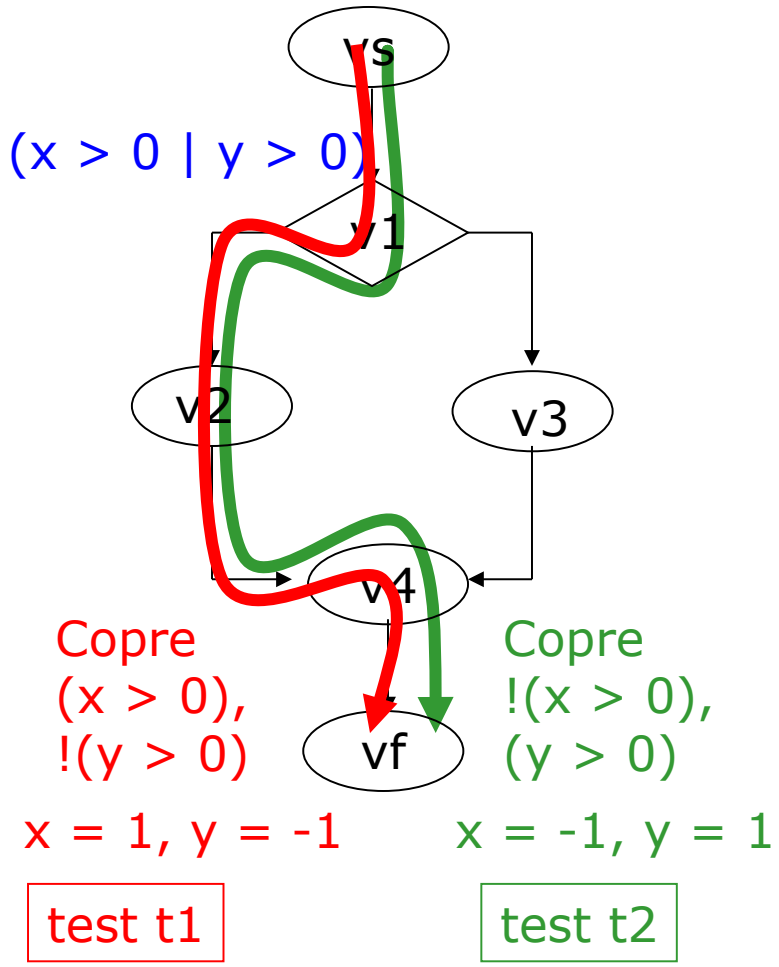
Copre ogni condizione e la sua negazione

Esempio copertura delle condizioni (1)

Test Set 1



Esempio copertura delle condizioni (2)



Anche Test Set 2 soddisfa la copertura delle condizioni

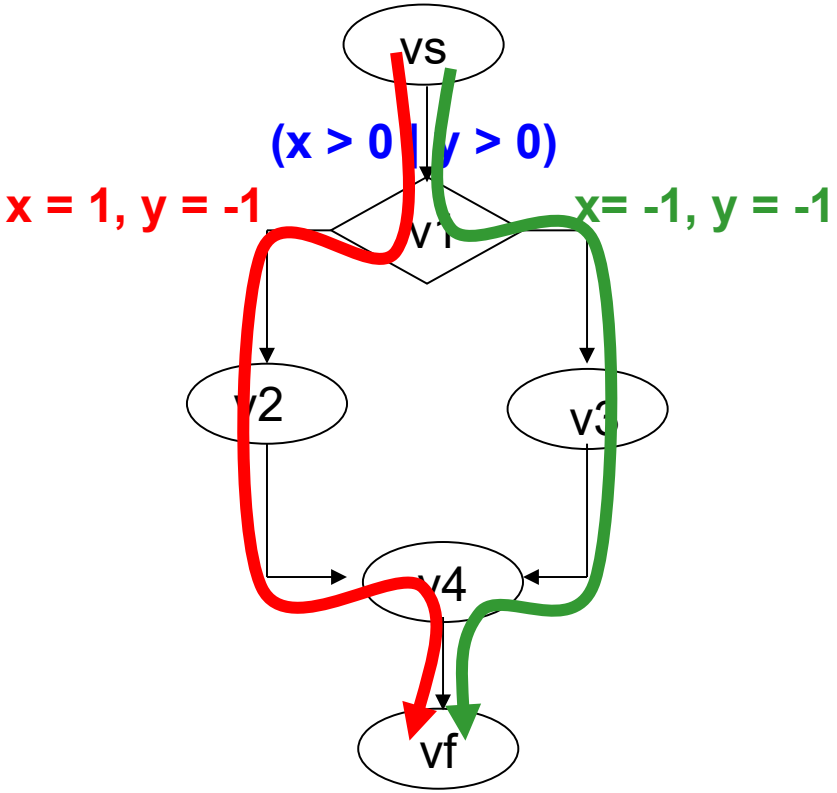
Tuttavia non copre la decisione !!

A. La copertura delle condizioni non implica la copertura delle decisioni

e neanche il branch coverage

Esempio copertura delle decisioni (2)

Test Set 3



Test set 3 copre le decisioni ma non le condizioni

- y non è mai > 0

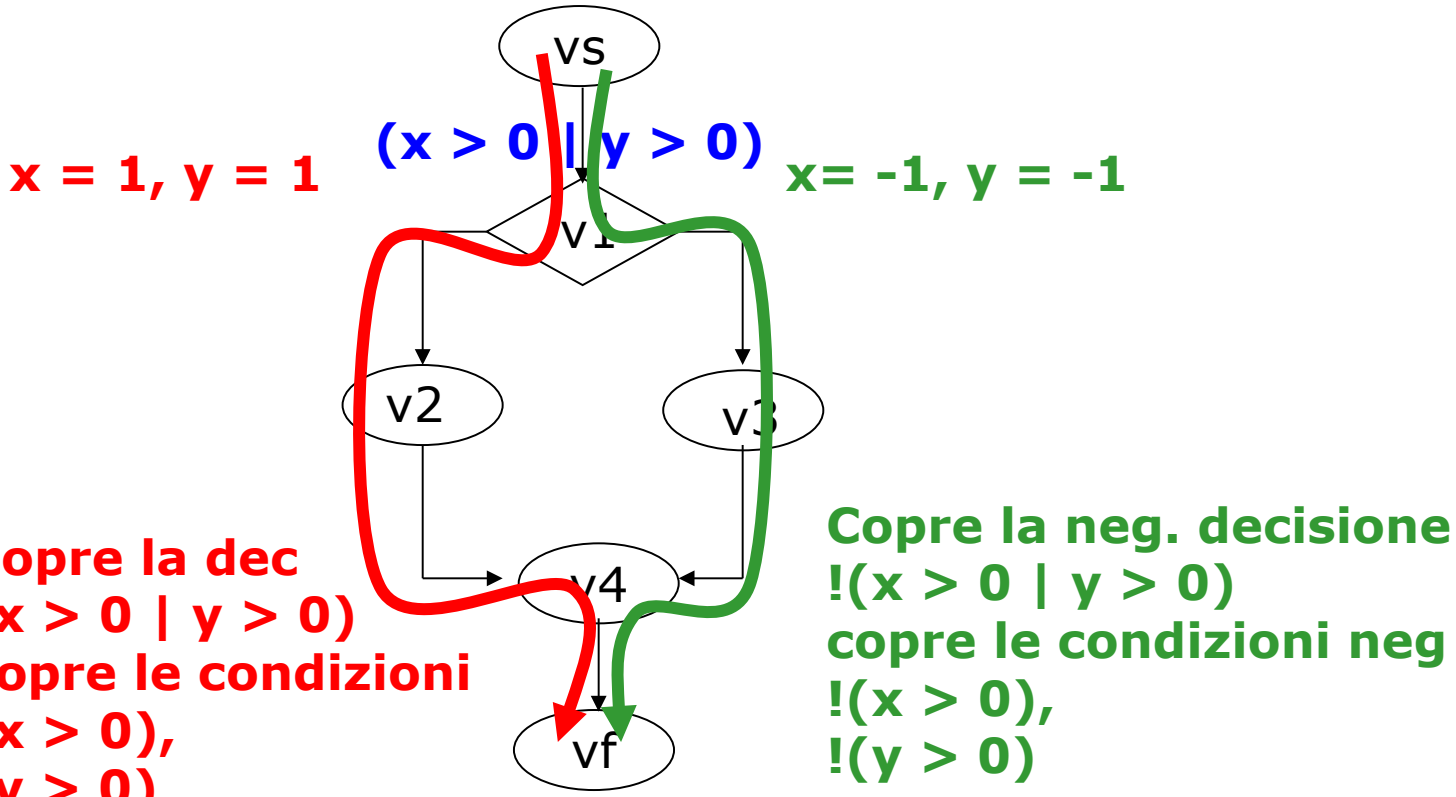
A. La copertura delle decisioni non implica la copertura delle condizioni

QUINDI A + B →

il criterio di copertura delle condizioni e quello delle decisioni sono incomparabili

Copertura decisioni e condizioni

Il criterio di copertura delle decisioni e delle condizioni richiede la copertura sia di tutte le **decisioni** che di tutte le **condizioni**



Short circuit evaluation

I compilatori spesso usano per efficienza la valutazione a corto circuito per le espressioni booleane:

Esempi:

- a && b: Se a e' falso non valuto b
- a || b: Se a e' vero non valuto b

Nota che spesso esistono operatori che evitano ciò (esempio & e | al posto di && e ||)

Questo può complicare la scelta dei casi di test

Esempio con short circuit

Testo ogni decisione/condizione per

```
( y > 0 ||  x > 0)
```

per testare x>0 la prima condizione deve essere falsa, perchè se è vera, x>0 non verrà valutata

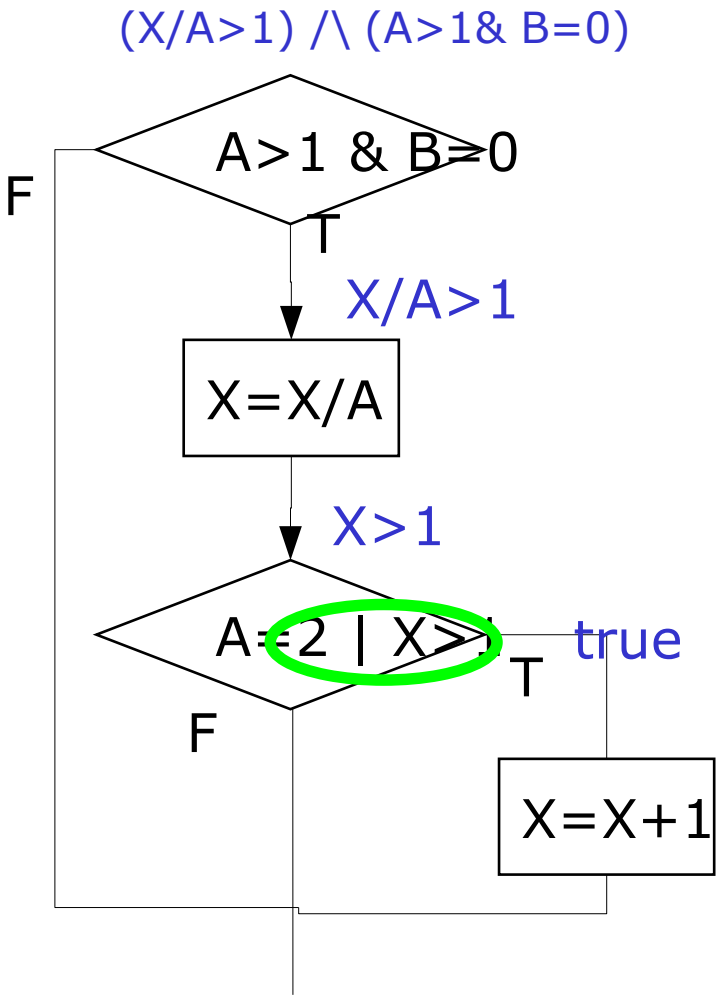
test	y>0	x>0	decisione
y = 1, x = 1	**T**	**non valutata**	**T**
y = 0, x = 1	**F**	**T**	**T**
y = 0, x = 0	**F**	**F**	**F**

Generazione

Per generare il caso di test desiderato:

se voglio coprire una certa condizione in una decisione, la includo nella costruzione della condizione sugli input

Esempio: con x>1



In sintesi

- Abbiamo visto che:
  - le decisioni sono le guardie di if, while, etc
  - le condizioni sono le espressioni atomiche all'interno delle decisioni
- Ricordati che:
  - il criterio di copertura delle decisioni richiede che ogni decisione sia valutata vera e falsa
  - quello delle condizioni richiede che ogni condizione sia valutata vera o falsa
  - i due criteri non si implicano uno con l'altro

Parte IV: MCC e MCDC

Criteri di test strutturali

Abbiamo visto i criteri di copertura:

1. delle istruzioni
2. dei branch
3. delle decisioni
4. delle condizioni
5. delle decisioni/condizioni

Vediamo ora come coprire ulteriormente le condizioni all'interno delle decisioni:

6. Multiple condition coverage (MCC)
7. Modified Condition/Decision coverage MCDC

Multiple condition coverage

Un test set soddisfa MCC se testa ogni possibile combinazione dei valori di verità delle condizioni atomiche in ogni decisione

Quindi con n condizioni in una decisione si avranno 2^n combinazioni da soddisfare (T e F per ogni condizione).

Si rappresenta in genere con una tabella

Il numero si riduce grazie alla short circuit evaluation

Per calcolare pensa alle condizioni come 1-0 …

Esempio Multiple condition cov.

(digit_high == -1 || digit_low == -1)

Test Case	digit_high == -1	digit_low == -1	
1	False	False	00
2	False	True	01
3	True	(non valutato)	10 11

il numero di test cresce esponenzialmente con il numero n di condizioni. Se n è grande l'MCC non è fattibile

Modified Condition/decision coverage

Per questo motivo si sono introdotti altri criteri più forti del semplice condition ma lineari con n.

il Modified Condition/decision coverage MCDC ha avuto successo - standard

Richiesto dalla FAA sul software dei aeromobili commerciali

Biliografia: Chilenski, J., Miller, S. *Applicability of modified condition/decision coverage to software testing.* Software Engineering Journal, September 1994, pp. 193--200.

MCDC

Il test set deve essere preso in modo che ogni condizione all'interno di una decisione deve far variare in modo indipendente il valore finale della decisione

Each atomic condition be shown to independently affect the outcome of each decision.

That is, for each atomic condition C, there are two test cases in which the truth values of all conditions except C are the same, and the compound condition as a whole evaluates to True for one of those test cases and False for the other.

Esempio MCDC (1)

per ogni condizione C in una decisione D ho due casi di test in cui C varia, le altre condizioni no e il valore finale della decisione D varia

Dec = A and B

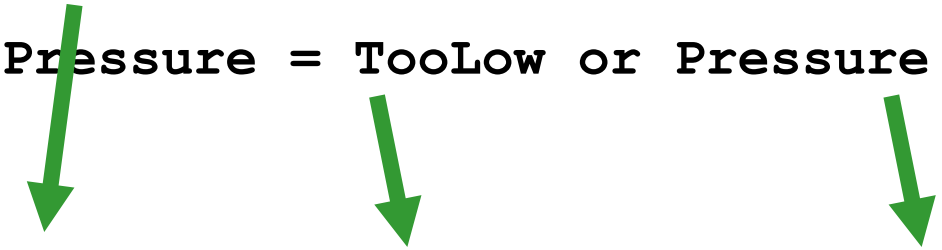
- considera A, varia solo A in modo che Dec vari (tengo B vero)
  - [A vero, B vero] (Dec) [A falso, B vero] (not Dec)
- varia B:
  - [A vero, B vero] (Dec) [A vero, B falso] (not Dec)
- dei 4 casi mi bastano 3 {[A,B],[not A,B][A,not B]}

Dec = A or B

- A: [A vero, B falso] (Dec) [A falso, B falso] (not Dec)
- B: [A falso, B vero] (Dec) [A falso, B falso] (not Dec)
- Tre casi {[not A, not B],[A, not B][not A,B]}

Esempio MCDC (2)

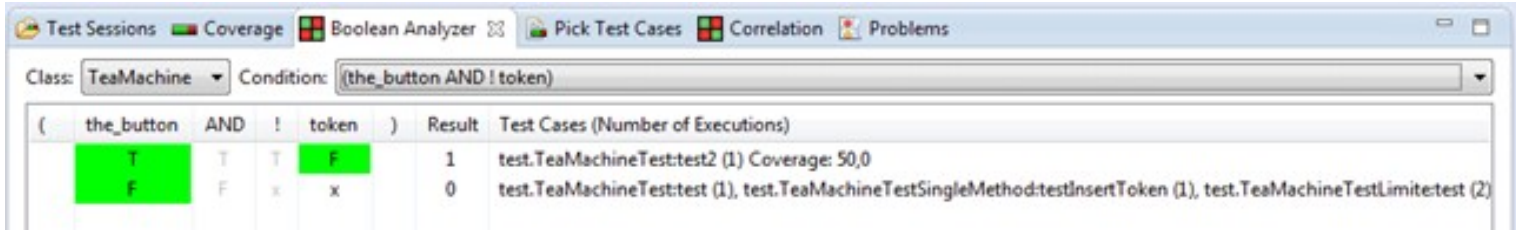
```
If Reset = on and
   (Pressure = TooLow or Pressure = Normal) then ...
```



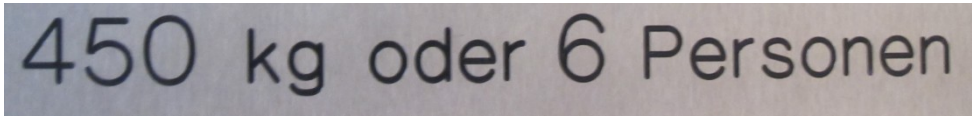
Reset = on	Pressure = TooLow	Pressure = Normal	**Valore finale**
T	T		T
F	T		F
T	T	F	T
T	F	F	F
T	F	T	T
T	F	F	F

Esempio

if(the_button && !token)



Alcune volte ci sono vincoli impliciti



Altri esempi di test per espressioni booleane

Full predicate coverage [Offutt Liu]

Missing condition coverage [Kuhn]

- come faccio a scoprire se una condizione atomica **C** e' stata dimenticata in una decisione? Scelgo i valori delle condizioni nella decisione in modo che se **C** è stata omessa il risultato finale è diverso da quello che si avrebbe che ci fosse.
- **esempio:**

```
if ( x > 3 and switchOn) then ...
```

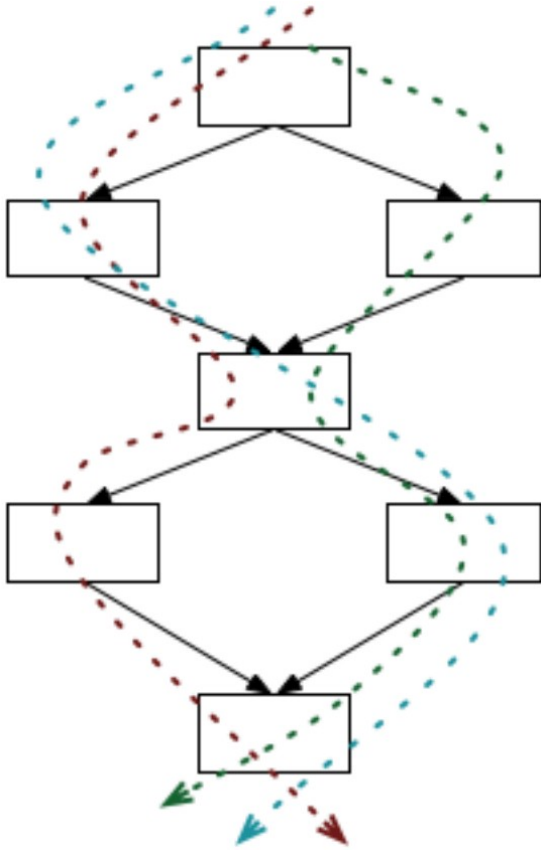
- se prendo `switchOn` vero e `x <=3` mi posso accorgere se la cond. `x>3` c'e' oppure no
- se prendo `switchOn` falso invece no

In sintesi

- Abbiamo visto altri criteri per testare le condizioni all'interno delle decisioni
- In particolare:
  - il multiple condition coverage testa tutte le possibili combinazioni di condizioni
  - il numero di test è esponenziale e rende inapplicabile questo criterio
  - il modified condition decision coverage richiede che i test mostrino come ogni condizione influisca sul valore finale delle decisione
  - cioè ogni condizione cambi in due casi di test in modo che il valore finale della decisione cambi tenendo ferme tutte le altre condizioni

Dal Cap 12

Paths? (Beyond individual branches)



- Should we explore sequences of branches (paths) in the control flow?
- Many more paths than branches
  - A pragmatic compromise will be needed

Path adequacy

- Decision and condition adequacy criteria consider individual program decisions
- Path testing focuses consider combinations of decisions along paths
- Adequacy criterion: each path must be executed at least once
- Coverage:

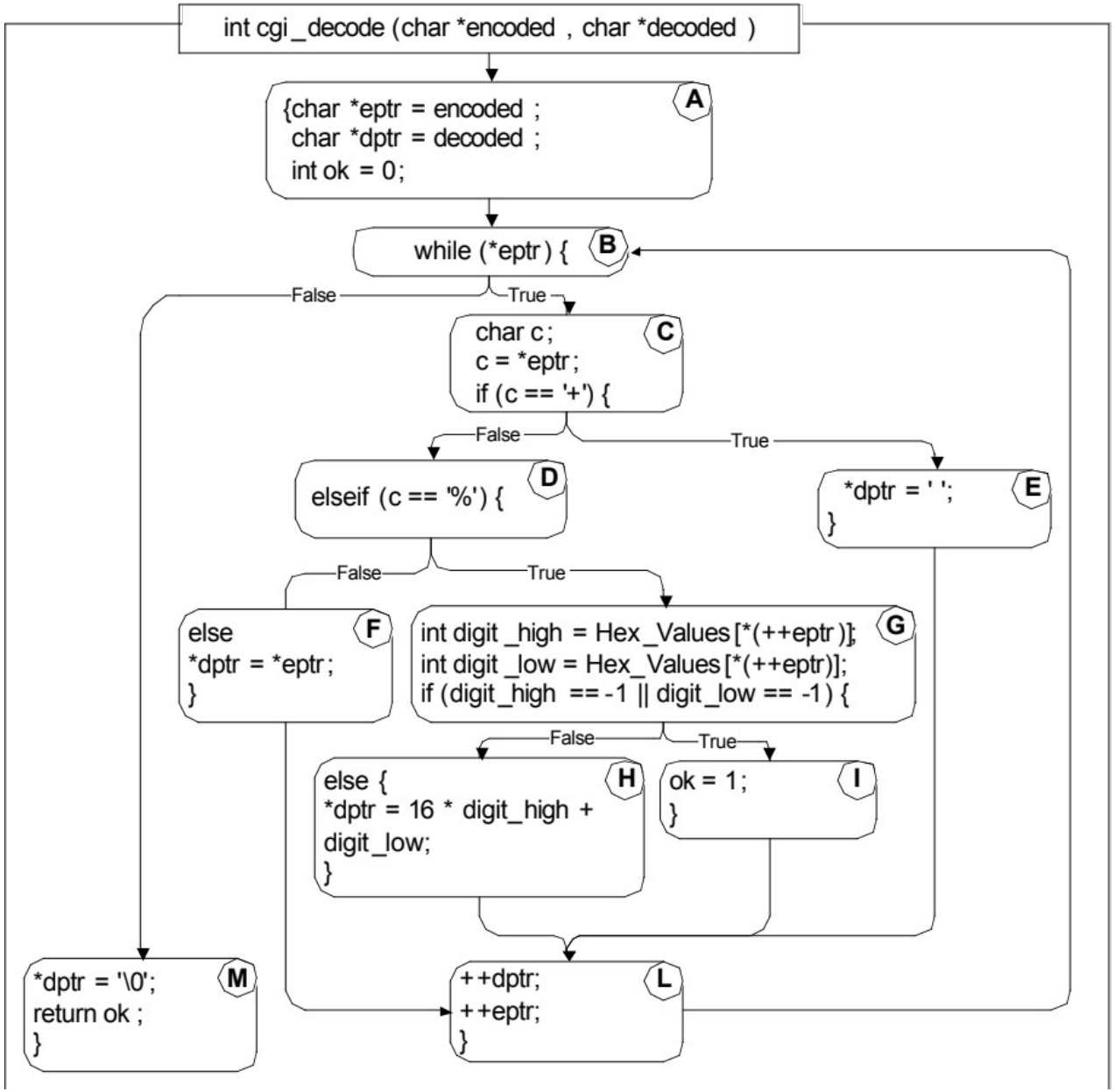
$$\frac{\text{\# executed paths}}{\text{\# paths}}$$

Practical path coverage criteria

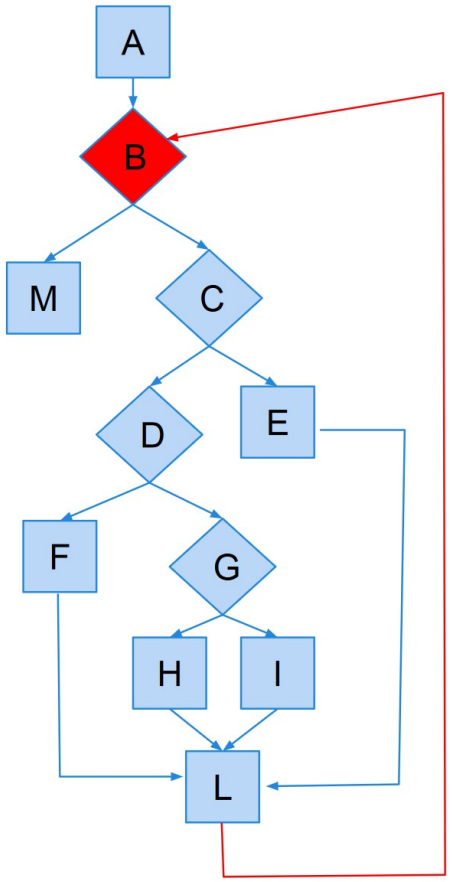
- The number of paths in a program with loops is unbounded
  - the simple criterion is usually impossible to satisfy
- For a feasible criterion: Partition infinite set of paths into a finite number of classes
- Useful criteria can be obtained by limiting
  - the number of traversals of loops
  - the length of the paths to be traversed
  - the dependencies among selected paths

Boundary interior path testing

- Group together paths that differ only in the subpath they follow when repeating the body of a loop
  - Follow each path in the control flow graph up to the first repeated node
  - The set of paths from the root of the tree to each leaf is the required set of subpaths for boundary/interior coverage


int cgi_decode (char *encoded , char *decoded)
{char *eptr = encoded ;
char *dptr = decoded ;
int ok = 0;
A
while (*eptr) {
B
False
True
char c;
c = *eptr;
if (c == '+') {
C
False
True
elseif (c == '%') {
D
*dptr = ' ';
}
E
False
True
else
*dptr = *eptr;
}
F
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
G
False
True
else {
*dptr = 16 * digit_high +
digit_low;
}
H
ok = 1;
}
I
*dptr = '\0';
return ok ;
}
M
++dptr;
++eptr;
}
L

Boundary interior adequacy for cgi-decode



B -> M
B -> C -> E -> L -> B
B -> C -> D -> F -> L -> B
B -> C -> D -> G -> H -> L -> B
B -> C -> D -> G -> I -> L -> B

Limitations of boundary interior adequacy

- The number of paths can still grow exponentially

```
if (a) {
   S1;
}
if (b) {
   S2;
}
if (c) {
   S3;
}
...
if (x) {
   Sn;
}
```

- The subpaths through this control flow can include or exclude each of the statements Si, so that in total N branches result in 2^N paths that must be traversed
- Choosing input data to force execution of one particular path may be very difficult, or even impossible if the conditions are not independent

Loop boundary adequacy

- Variant of the boundary/interior criterion that treats loop boundaries similarly but is less stringent with respect to other differences among paths
- Criterion: A test suite satisfies the loop boundary adequacy criterion iff for every loop:
  - In at least one test case, the loop body is iterated zero times
  - In at least one test case, the loop body is iterated once
  - In at least one test case, the loop body is iterated more than once
- Corresponds to the cases that would be considered in a formal correctness proof for the loop

LCSAJ adequacy

- Linear Code Sequence And Jumps: sequential subpath in the CFG starting and ending in a branch
  - TER_1 = statement coverage
  - TER_2 = branch coverage
  - TER_{n+2} = coverage of n consecutive LCSAJs

Cyclomatic adequacy

NO

- Cyclomatic number:
  number of independent paths in the CFG
  - A path is representable as a bit vector, where each component of the vector represents an edge
  - "Dependence" is ordinary linear dependence between (bit) vectors
- If e = #edges, n = #nodes, c = #connected components of a graph, it is:
  - e - n + c for an arbitrary graph
  - e - n + 2 for a CFG
- Cyclomatic coverage counts the number of independent paths that have been exercised, relative to cyclomatic complexity

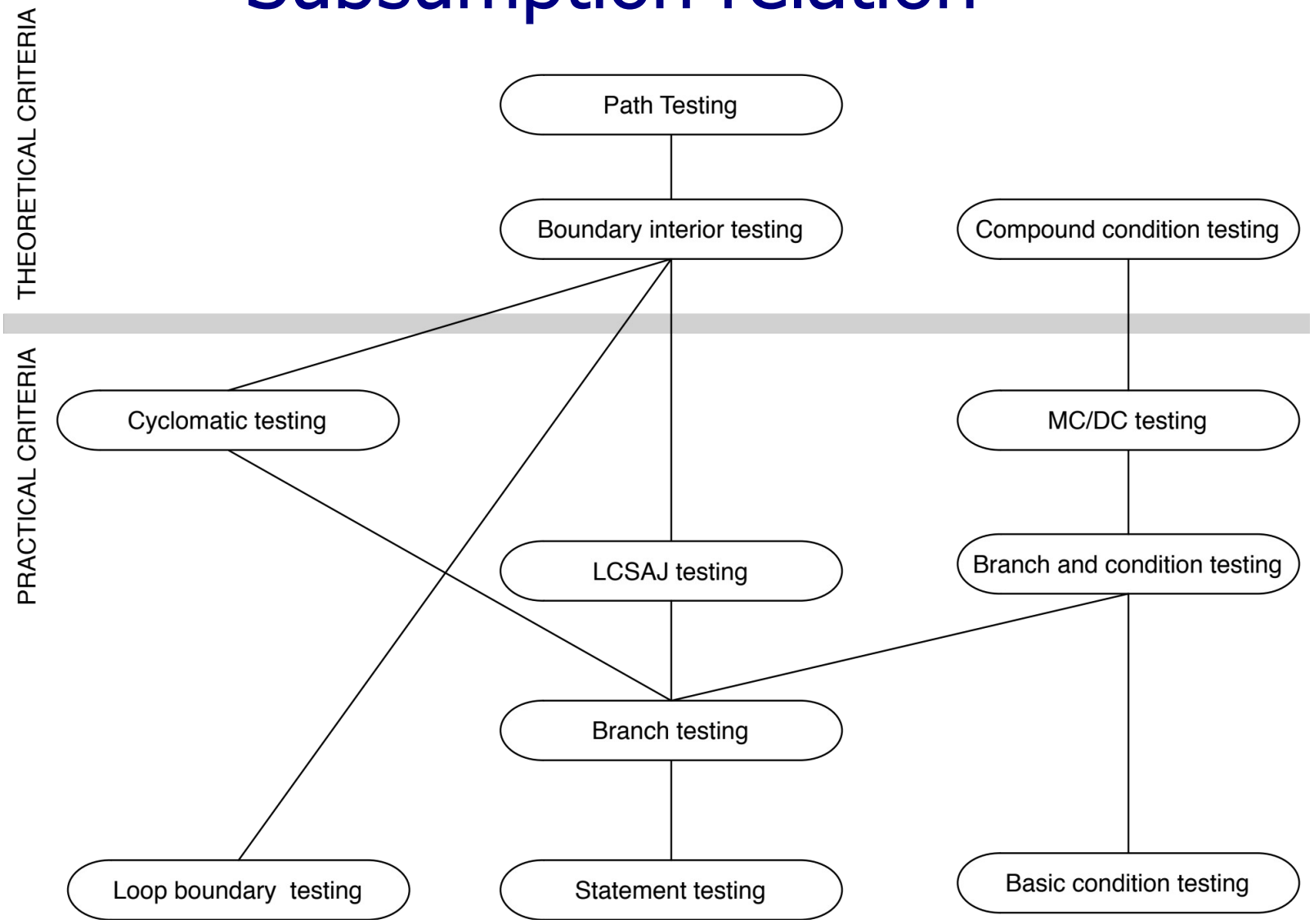
Towards procedure call testing

- The criteria considered to this point measure coverage of control flow within individual procedures.
  - not well suited to integration or system testing
- Choose a coverage granularity commensurate with the granularity of testing
  - if unit testing has been effective, then faults that remain to be found in integration testing will be primarily interface faults, and testing effort should focus on interfaces between units rather than their internal details

Procedure call testing

- Procedure entry and exit testing
  - procedure may have multiple entry points (e.g., Fortran) and multiple exit points
- Call coverage
  - The same entry point may be called from many points

Subsumption relation



Satisfying structural criteria

- Sometimes criteria may not be satisfiable
  - The criterion requires execution of
    - statements that cannot be executed as a result of
      - defensive programming
      - code reuse (reusing code that is more general than strictly required for the application)
    - conditions that cannot be satisfied as a result of
      - interdependent conditions
    - paths that cannot be executed as a result of
      - interdependent decisions

Satisfying structural criteria

- Large amounts of *fossil* code may indicate serious maintainability problems
  - But some unreachable code is common even in well-designed, well-maintained systems
- Solutions:
  - make allowances by setting a coverage goal less than 100%
  - require justification of elements left uncovered
    - RTCA-DO-178B and EUROCAE ED-12B for modified MC/DC

Summary

- We defined a number of adequacy criteria
  - NOT test design techniques!
- Different criteria address different classes of errors
- Full coverage is usually unattainable
  - Remember that attainability is an undecidable problem!
- …and when attainable, "inversion" is usually hard
  - How do I find program inputs allowing to cover something buried deeply in the CFG?
  - Automated support (e.g., symbolic execution) may be necessary
- Therefore, rather than requiring full adequacy, the "degree of adequacy" of a test suite is estimated by coverage measures
  - May drive test improvement

Generazione automatica dei casi di test

Tool per la generazione di casi di test

Ci sono molto tecniche per la generazione automatica dei casi di test

- Vantaggio: non richiedono intervento umano
- Svantaggi:
  - nessuna può garantire il 100% per tutti i programmi
  - Il problema degli oracoli: come faccio a dire che il programma si è comportato in modo corretto?
    - Facile generare gli "input"

Vedremo due approcci (in laboratorio)

- Random: randoop
- evosuite – più analitico