



Tecniche e metodologie di testing



Tecniche di testing

- Le principali tecniche per l'effettuazione di Software Testing sono:
 - Analisi statica
 - Analisi dinamica
 - Analisi formale



Tecniche di testing

- **Analisi statica:** processo di valutazione di un sistema o di un suo componente basato sulla sua forma, struttura, contenuto, documentazione senza che esso sia eseguito. Esempi sono: revisioni, ispezioni, recensioni, analisi data flow
- **Analisi dinamica:** processo di valutazione di un sistema software o di un suo componente basato sulla osservazione del suo comportamento in esecuzione. Tipicamente con il termine testing ci si riferisce all'analisi dinamica
- **L'analisi formale** usa rigorose tecniche matematiche per l'analisi di algoritmi. E' usata soprattutto per la verifica del codice e dei requisiti specie quando questi sono specificati con linguaggi formali (es. Z, VDM).



Principali tecniche di analisi statica

- Analisi statica in compilazione
- Code reading
- Control flow analysis
- Data flow analysis
- Code inspections or reviews
- Esecuzione simbolica



Analisi statica in compilazione

- I compilatori effettuano una analisi statica del codice per verificare che un programma soddisfi particolari caratteristiche di correttezza statica, per poter generare il codice oggetto.
- Le informazioni e le anomalie che può rilevare un compilatore dipendono dalle caratteristiche del linguaggio e dalle facility di cui esso dispone.
- Es. linguaggi con regole di visibilità statica dei nomi permettono la rilevazione di un maggiore numero di anomalie di quelli con regole di visibilità dinamiche.



Analisi statica in compilazione

- La generazione di Cross Reference List risulta molto utile in successive analisi del codice per l'individuazione di anomalie non rilevabili dal compilatore.
- Tipiche anomalie identificabili: nomi di identificatori non dichiarati, incoerenza tra tipi di dati coinvolti in una istruzione, incoerenza tra parametri formali ed effettivi in chiamate a subroutine, codice non raggiungibile dal flusso di controllo



Code reading

- E' effettuata un'attenta lettura del codice per individuare errori e/o discrepanze con il progetto.
- Il lettore effettua mentalmente una pseudo-esecuzione del codice e processi di astrazione che lo conducono a verificare la correttezza del codice rispetto alle specifiche e il rispetto di standard adottati.



Code reading

- **Tipici errori identificabili:** nomi di identificatori errati, errato innesto di strutture di controllo, loop infiniti, inversione di predicati, commenti non consistenti con il codice, incorretto accesso ad array o altre strutture dati, incoerenza tra tipi di dati coinvolti in una istruzione, incoerenza tra parametri formali ed effettivi in chiamate a subroutine, inefficienza dell'algoritmo, non strutturazione del codice, codice morto, etc.
- Il code reading è talvolta indicato anche con le dizioni desk review o desk checking.



Code inspections - review

- Riunioni formali cui partecipa un gruppo di persone tra cui almeno una del gruppo di sviluppo
- Il codice è esaminato linea per linea e i partecipanti fanno commenti e/o annotazioni
- Tipicamente queste riunioni sono preannunciate ai partecipanti cui viene fornita la documentazione necessaria (codice e relativi documenti) per la revisione
- L'analisi effettuata viene discussa nella riunione dove vengono decise le azioni eventualmente da intraprendere (accettazione del codice, rigetto, annotazioni su eventuali non aderenze a specifiche, indicazioni delle modifiche da apportare).



Control Flow Analysis

- Il flusso di controllo è esaminato per verificarne la correttezza.
- Il codice è rappresentato tramite un grafo, il grafo del flusso di controllo (Control flow Graph - CfG), i cui nodi rappresentano statement (istruzioni e/o predicati) del programma e gli archi il passaggio del flusso di controllo.
- Il grafo è esaminato per identificare ramificazioni del flusso di controllo e verificare l'esistenza di eventuali anomalie quali codice irraggiungibile e non strutturazione.



Data flow analysis

- Analizza l'evoluzione del valore delle variabili durante l'esecuzione di un programma, permettendo di rilevare anomalie.
- Intrinsecamente dinamica, ma alcuni aspetti possono essere analizzati staticamente.
- L'analisi statica è legata alle operazioni eseguite su una variabile:
 - **definizione:** alla variabile è assegnato un valore
 - **uso:** il valore della variabile è usato in un'espressione o un predicato
 - **annullamento:** al termine di un'istruzione il valore associato alla variabile non è più significativo

Es. nell'espressione $a := b + c$ la variabile a è definita mentre b e c sono usate



Data flow analysis

- La definizione di una variabile, così come un annullamento, cancella l'effetto di una precedente definizione della stessa variabile, ovvero ad essa è associato il nuovo valore derivante dalla nuova definizione (o il valore nullo)
- Una corretta sequenza di operazioni prevede che:
 - L'uso di una variabile x deve essere sempre preceduto da una definizione della stessa variabile x , senza annullamenti intermedi
 - Un uso non preceduto da una definizione può corrispondere al potenziale uso di un valore non determinato



Data flow analysis

- Una definizione di una variabile x deve essere sempre seguita da un uso della variabile x , prima di un'altra definizione o di un annullamento della stessa variabile x
 - Una definizione non seguita da un uso corrisponde all'assegnamento di un valore non utilizzato e quindi potenzialmente inutile



Data flow analysis

- Sequenze di istruzioni sono riconducibili a sequenze di *definizioni* (d), *usi* (u), *annullamenti* (a) delle variabili referenziate nei comandi

Procedure swap (x1, x2: real)

var x:real

begin

x2:=x;

x2:=x1;

x1:=x;

end;

x: (auu)

x2: (ddd)

x1: (dud)

La sequenza (auu) di x e la sequenza (ddd) di x2 sono indicative di una qualche anomalia



Data flow analysis

Non sempre le sequenze ...**au**... o ...**dd**... corrispondono ad anomalie

Es.

au può comparire in un generatore di numeri casuali (è letto il contenuto di una cella di memoria non inizializzata per determinare il seme della generazione)

dd può dipendere da una cattiva strutturazione del programma (la prima definizione non è usata nella esecuzione considerata ma lo è un'altra, su un altro cammino)



Data flow analysis

```
1 .....  
2 x:= .....  
3 if .... then x:=  
  ....  
4 ... := ... x ....  
5 .....
```

Nelle linee 2 e 3 due definizioni staticamente consecutive di x; la definizione della linea 2 è usata nella linea 4 che in alcune esecuzioni può seguire direttamente la linea 2

Sequenze di azioni su una variabile per una determinata esecuzione (cammino) sono rappresentabili tramite espressioni



Data flow analysis

- L'espressione relativa ad un cammino p di un programma P per la variabile x è indicata con $P(p; x)$

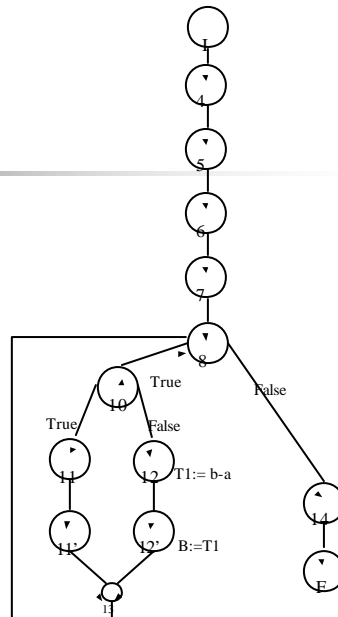
1
2	x:=
3	if then x:=
4	... := ... x ...
5

$P([\dots, 2, 4, \dots]; x) = (\dots du \dots)$

$P([\dots, 2, 3, 4, \dots]; x) = (\dots ddu \dots)$



```
1 program gcd (input, output);
2 var
3 x,y,a,b : integer;
4 begin
5 read (x,y);
6 a:=x;
7 a:=y
8 while a <> b do
9   begin
10    if a>b
11      then a:=a-b
12      else b:=b-a;
13   end;
14 write ("il massimo comune divisore è", a)
15 end.
```





Data flow analysis

A ciascun nodo del CfG è possibile associare l'insieme delle variabili definite in esso, quello delle variabili usate e quello delle variabili annullate.

Nodo	Var. definite	Var usate	Var. annullate
4			x, y, a, b
5	x, y		
6	a	x	
7	a	y	
8		a, b	
10		a, b	
11		a, b	
11'	a		
12		a, b	
12'	b		
14		a	

E' quindi possibile scrivere l'espressione $P(p; x)$ facendo riferimento a tali insiemi

Anomalia ...dd... per *a*
anomalia ...a--u... per *b*
dovuta ad errore a linea 7 $b := y;$

$$P([I,4,5,6,7,8,10,11,11',8,14,F];a) = (-a-dduuuuuu-)$$

$$P([I,4,5,6,7,8,10,11,11',8,14,F];b) = (-a--uuu-u--)$$



Espressioni regolari

- Usate per rappresentare espressioni relative ad una variabile corrispondenti a più cammini
- Un'espressione regolare è definita a partire da un alfabeto finito A (nel nostro caso $A = \{a, d, u, -\}$) e dalle seguenti regole ricorsive:
 - ϵ , stringa nulla, è un'espressione regolare
 - ogni simbolo di A è un'espressione regolare
 - se e_1 ed e_2 sono espressioni regolari, allora lo sono anche le espressioni che si formano da queste con l'uso degli operatori sequenza (\cdot), alternativa ($+$) e ciclo ($*$); quindi $e_1 \cdot e_2$, $e_1 + e_2$, e_1^* sono espressioni regolari
 - niente altro è un'espressione regolare

$$P([I \textcircled{R}];a) = a-ddu(u(ud+u)u)^*u$$

$$P([\textcircled{R} 8];a) = -a-dd$$

$[n \rightarrow]$ indica tutti i cammini uscenti dal nodo n , nodo n escluso;

$[\rightarrow n]$ indica tutti i cammini entranti nel nodo n , nodo n escluso



Esecuzione simbolica

- Il programma non è eseguito con i valori effettivi ma con valori simbolici dei dati di input
- L'esecuzione procede come una esecuzione normale ma non sono elaborati valori bensì formule formate dai valori simbolici degli input
- Gli output sono formule dei valori simbolici degli input
- L'esecuzione simbolica anche di programmi di modeste dimensioni può risultare molto difficile
- Ciò è dovuto all'esecuzione delle istruzioni condizionali: deve essere valutato ciascun caso (vero e falso); in programmi con cicli ciò può portare a situazioni difficilmente gestibili.



Esecuzione simbolica

```
1 function product (x,y,z: integer):  
integer;  
2 var tmp1, tmp2: integer;  
3 begin  
4 tmp1 := x*y;  
5 tmp2 := y*z;  
6 product := tmp1 * tmp2 / y;  
7 end;
```

Stm.	x	y	z	tmp1	tmp2	product
1	X	Y	Z	?	?	?
4	X	Y	Z	X*Y	?	?
5	X	Y	Z	X*Y	Y*Z	?
6	X	Y	Z	X*Y	Y*Z	(X*Y)*(Y*Z)/Y



Esecuzione simbolica

<code>read (x, y, z);</code>	X, Y, Z
<code>t:=x+y;</code>	$T \mapsto X+Y$
<code>x:=x+t-1;</code>	$X \mapsto 2*X+Y-1$
<code>z:=t*y;</code>	$Z \mapsto X*Y + Y * Y$
<code>y:=y+1;</code>	$Y \mapsto Y+1$
<code>if x>=0 then</code>	$X \geq 0$
<code>X<0</code>	
<code>t:=x+y+z;</code>	$T \mapsto (2+Y)*(X+Y)$
<code>T↦ X+Y</code>	
<code>write (t);</code>	



Classificazione di primo livello delle tecniche di testing dinamico

- Black Box Testing (Testing Funzionale)
- White Box Testing (Testing Strutturale)



Testing funzionale

- Il testing é fondato sull'analisi degli output generati dal sistema o da suoi componenti in risposta ad input definiti sulla base della sola conoscenza dei requisiti specificati del sistema o di suoi componenti

Testing strutturale

- Il testing é fondato sulla definizione dei casi di prova, degli input associati e dell'oracolo sulla base della conoscenza della struttura del software ed in particolare del codice

- Requisiti funzionali, specifiche, decomposizione funzionale, requisiti di performance...e tc.
- Tipi di dati, dati, strutture di controllo, control flow, data flow, call-graph, dominanze, dipendenze etc.



Tecniche di testing funzionale

... cosa testare

Funzionalità esterne: funzionalità visibili all'utente e definite dai requisiti e dalle specifiche

Funzionalità interne: funzionalità non visibili all'utente e definite dal progetto di high e low level

.... partendo da

Definizione Funzionalità: dati di ingresso, dati di uscita, precondizioni, postcondizioni



Oracolo

- Condizione necessaria per effettuare un test:
 - conoscere il comportamento atteso per poterlo confrontare con quello osservato per poter definire l'oracolo
- L'oracolo conosce il comportamento atteso per ogni caso di prova
- **Oracolo umano**
 - si basa sulle specifiche o sul giudizio
- **Oracolo automatico**
 - generato dalle specifiche (formali)
 - stesso software ma sviluppato da altri
 - versione precedente (test di regressione)



Criteri di copertura per il testing funzionale

- Ogni funzionalità deve essere eseguita almeno una volta
 - Per ogni funzionalità effettuare un numero di esecuzioni dedotte dai dati di ingresso e di uscita, da precondizioni e postcondizioni
 - Definito il dominio dei dati di I/O effettuare test case ottenuti selezionando
 - valori in posizione centrale
 - valori ai bordi
 - valori "speciali"
 - Precondizioni e postcondizioni per test case
 - positivi
 - negativi (invalid input data, invalid output data)
 - neutrali
- ... Suddivisione dei test case in classi di equivalenza



Suddivisione in classi di equivalenza

- Il dominio dei dati di ingresso è suddiviso in classi di casi di test in modo tale che, se il programma è corretto per un caso di test, si possa dedurre ragionevolmente che è corretto per ogni caso di test in quella classe
- Una classe di equivalenza rappresenta un insieme di stati validi o non validi per una condizione sulle variabili d'ingresso



Definizione delle classi di equivalenza

Se la condizione sulle variabili d'ingresso specifica:

- Intervallo di valori:
 - Una classe valida per valori interni all'intervallo, una non valida per valori inferiori al minimo, e una non valida per valori superiori al massimo; una classe per valori uguali o vicini agli estremi dell'intervallo
- Valore specifico:
 - Una classe valida per il valore specificato, una non valida per valori inferiori, e una non valida per valori superiori



Definizione delle classi di equivalenza

- Elemento di un insieme discreto
 - Una classe valida per ogni elemento dell'insieme, una non valida per un elemento non appartenente
- Valore booleano
 - Una classe valida per il valore TRUE, una classe non valida per il valore FALSE



Selezione dei casi di test dalle classi di equivalenza

- Ogni classe di equivalenza deve essere coperta da almeno un caso di test
 - Un caso di test per ogni classe non valida
 - Ciascun caso di test per le classi valide deve comprendere il maggior numero di classi valide ancora scoperte



Analisi dei valori limite

- I casi di test che esplorano condizioni limite spesso rilevano la presenza di malfunzionamenti
- Le condizioni limite:
 - Sono direttamente agli estremi
 - Immediatamente al di sopra
 - Immediatamente al di sotto degli estremi di:
 - Classi di equivalenza di ingresso
 - Classi di equivalenza di uscita
- Le condizioni limite possono essere molto sottili
 - Usare la creatività per cercare altre situazioni estreme



Documentare i test eseguiti

Produzione, archiviazione ed uso di checklist

.... Ad esempio ...

Funzionalità	T.C.	Input	Output	Pre-cond	Post-cond	Priorità	Esito



Metodo top-down di produzione delle check list

- Analisi dei documenti di definizione dei requisiti e di specifica, estrazione delle funzionalità esterne e di altre features, dati di I/O e pre e post condizioni, definizioni priorità, nomi dei test
- Analisi dei documenti di progetto di high e low level, estrazione delle funzionalità interne, dati di I/O e pre e post condizioni, definizioni priorità, nomi dei test
- Definizione dei test case e delle procedure di test
- Generazione tabella di copertura (o dei test case) e tabella delle procedure



Tecniche di testing strutturale

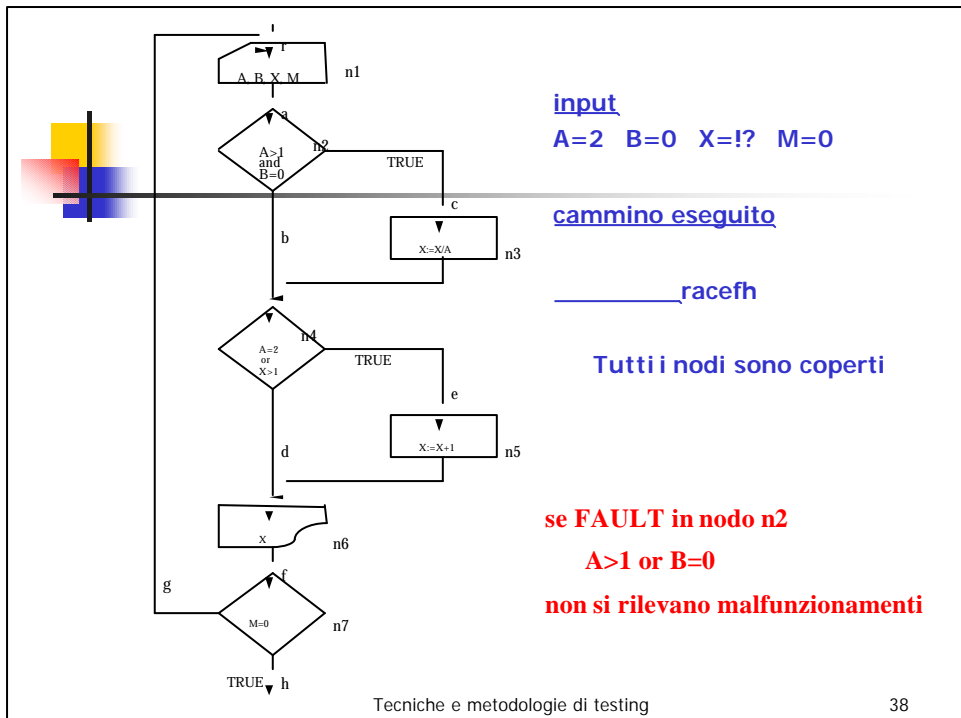
- Le tecniche di testing strutturale sono in generale fondate sull'adozione di metodi di Copertura degli oggetti che compongono la struttura dei programmi
- **COPERTURA:** definizione di un insieme di casi di test (in particolare input data) in modo tale che gli oggetti di una definita classe (es. strutture di controllo, istruzioni, archi del GFC, predicati,..etc.) siano attivati almeno una volta nell'esecuzione dei casi di test

...criteri di copertura sono definibili anche per il testing funzionale....

Node (statement) coverage

- Dato un GFC(P), ovvero P, viene definito un insieme dei 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):
$$\frac{\text{n.ro di statement eseguiti}}{\text{n.ro di statement totale}}$$



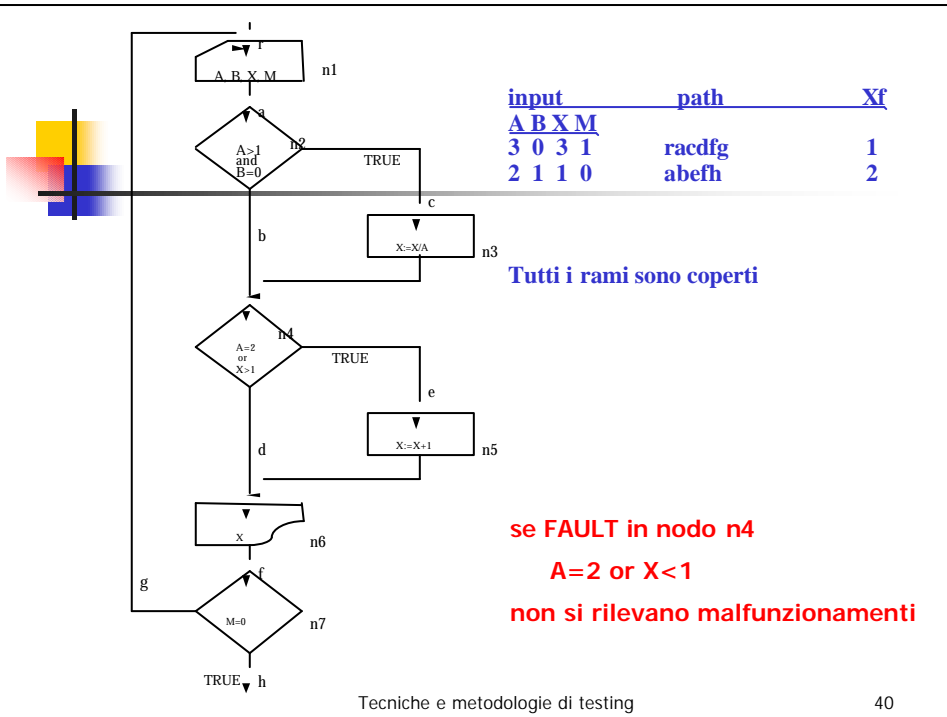


Branch (decision) coverage

- Dato un GFC(P), ovvero P, viene definito un insieme dei test case la cui esecuzione implica l'attraversamento di tutti i rami di GFC(P), ovvero l'esecuzione di tutte le decisioni di P.

$$\text{Test Effectiveness Ratio (TER)} : \frac{\text{n.ro di rami attraversati}}{\text{n.ro di totale di rami}}$$

La copertura di tutti i rami implica la copertura di tutti i nodi

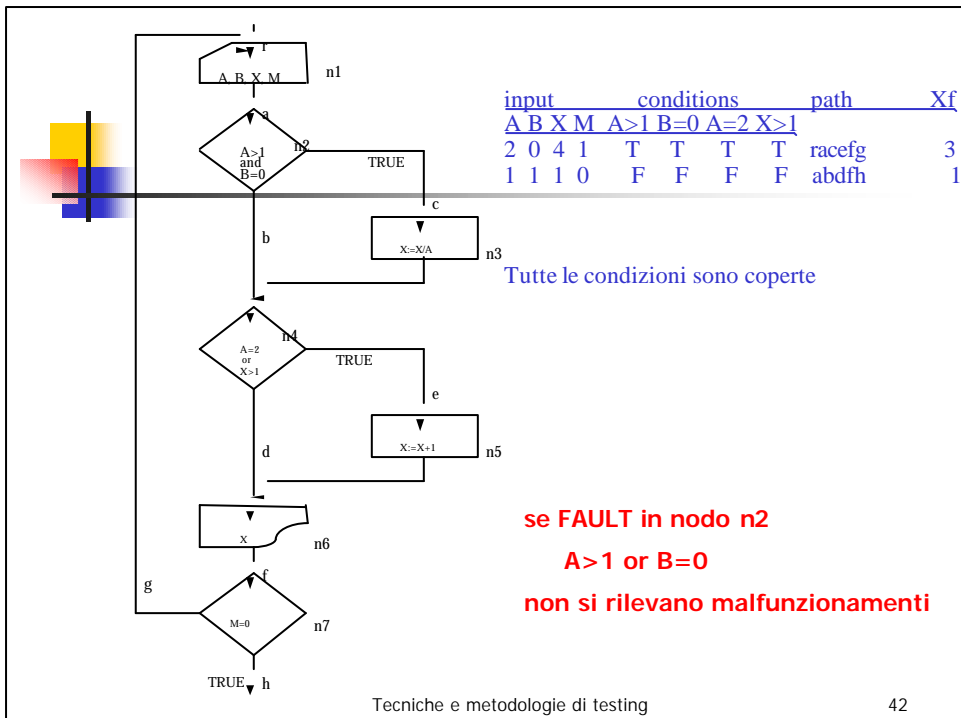




Condition coverage

- Dato un P, viene definito un insieme dei 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.

La copertura di tutte le condizioni NON implica la copertura di tutte le decisioni!

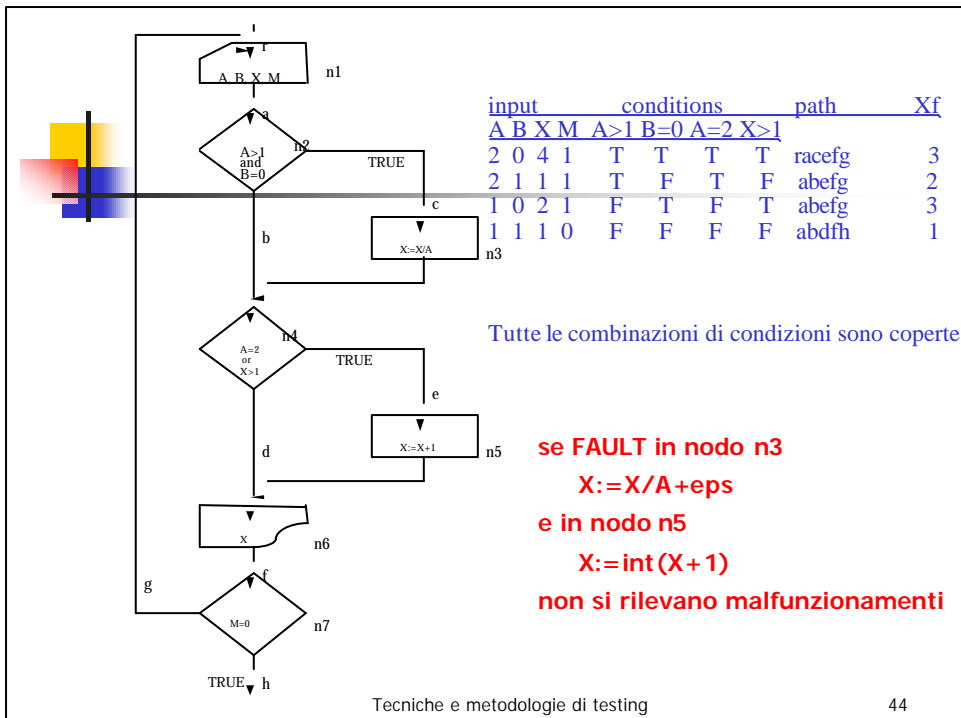




Multiple condition coverage

- Dato un P, viene definito un insieme dei test case la cui esecuzione implica l'esecuzione, per ogni decisione, di tutte le combinazioni di condizioni

La copertura di tutte le combinazioni di condizioni implica la copertura di tutte le condizioni e di tutte le decisioni





Criterio di n-copertura dei cicli

- Un test T soddisfa il criterio di n-copertura dei cicli se e solo se ogni cammino contenente un numero intero di iterazioni di ogni ciclo non superiore ad n è eseguito per almeno un dato di test T

Problema: stabilire il numero ottimale di iterazioni....



Metodi basati sulla copertura dei cammini

DEVE ESSERE ATTIVATO OGNI CAMMINO DEL GRAFO

- **I problema:** il numero dei cammino è, generalmente, infinito
- **II problema:** gli infeasible path

Soluzione: selezione di un numero finito ed eseguibile di cammini:

- metodi fondati sui grafi di controllo
- metodi data flow

(... naturalmente un buon testatore cerca le ragioni della presenza di infeasible path ...)



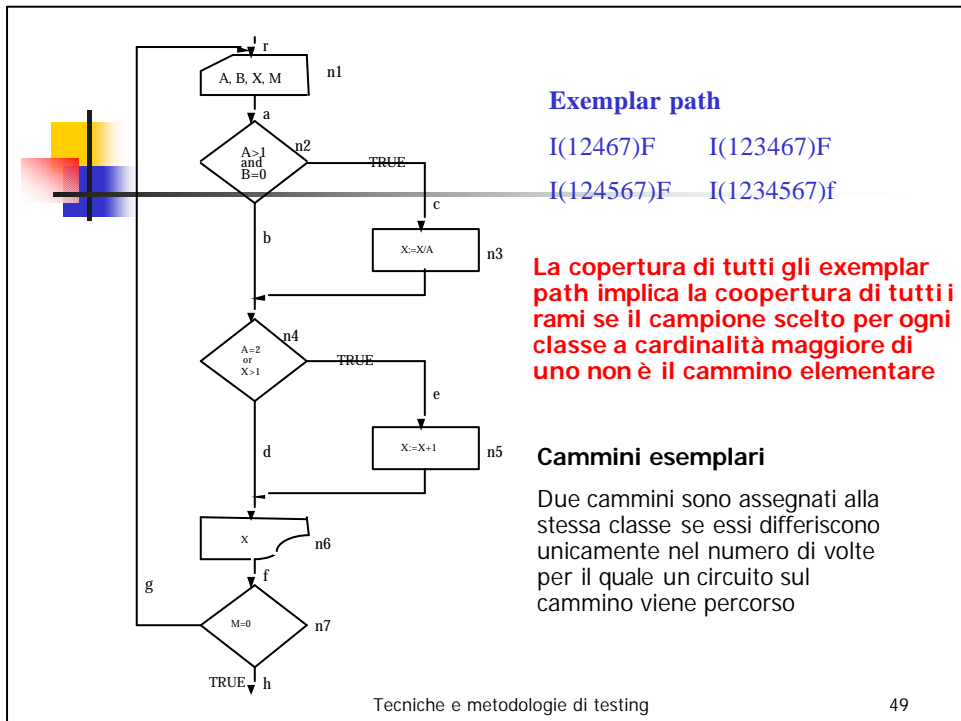
Metodi basati sul grafo di controllo

- L'insieme (o un sottoinsieme definito) 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 assicurino l'attraversamento almeno una volta di almeno un cammino per ogni classe.
 - Metodo dei cammini esemplari
 - Metodo dei cammini linearmente indipendenti (McCabe)



Metodo dei cammini esemplari

- Un cammino privo di circuiti è detto cammino elementare
- Il numero di cammini elementari in un grafo è finito
- Si consideri il numero di cammini elementari e totali di $GfC(P)$
- Classi di cammini considerate: un cammino elementare totale e tutti i cammini che da esso differiscono unicamente per il numero di attraversamenti di circuiti che sul cammino giacciono

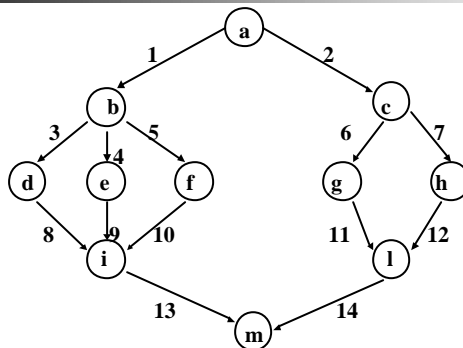


Metodo dei cammini linearmente indipendenti (McCabe)

- Un cammino si dice indipendente se introduce almeno un nuovo insieme di istruzioni o una nuova condizione; in un CfG un cammino è indipendente se attraversa almeno un arco non ancora percorso
- L'insieme di tutti i cammini linearmente indipendenti di un programma formano i 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.

Metodo dei cammini linearmente indipendenti (McCabe)

- Il numero dei cammini linearmente indipendenti di un programma è pari al numero cicломatico di McCabe:
 - $V(G) = E - N + 2$ E: n.ro di archi in G - N: n.ro di nodi in G
 - $V(G) = P + 1$ P: n.ro di predicati in G
 - $V(G) = \text{n.ro di regioni chiuse in G} + 1$
- Test case esercitanti i cammini di base garantiscono l'esecuzione di ciascuna istruzione almeno una volta



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

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

3) $V(G) = \text{N.ro regioni chiuse} + 1 = 4 + 1 = 5$

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



Metodi fondati sul data flow

- Cammini def-use: un def-clear-path rispetto ad una variabile v

- Triple def-use (i, j, v)



Data flow e dipendenze sui dati

- Data una componente di tipo procedura P , sia V l'insieme delle variabili di P (referenziate da istruzioni e predicati in P) e sia $GFC(P) = \langle N, AC, nI, nF \rangle$, il suo grafo del flusso di controllo. $\forall n \in N - \{nI, nF\}$, siano $DEF(n) \subseteq V$ e $USO(n) \subseteq V$ gli insiemi delle variabili rispettivamente definite e usate da n .
- La relazione $DEF_USO(P) \subseteq N - \{nI, nF\} \times N - \{nI, nF\} \times V$ è definita come segue:
 $(n, m, v) \in DEF_USO(P)$ se e solo se:
 - (i) $v \in DEF(n) \cap USO(m)$;
 - (ii) \exists un cammino di lunghezza $k \geq 1$, $n = p_0, p_1, \dots, p_k = m$ in $GFC(P)$ tale che $\forall i, 1 \leq i \leq k-1, v \notin DEF(p_i)$.



Data flow e dipendenze tra i dati

- **computational-use(c-use)**: si ha un c-use(v) quando la variabile v è usata in output o è usata nella espressione di una assegnazione;
- **predicate-use(p-use)**: si ha un p-use(v) quando la variabile v è usata in un predicato che condiziona il control flow;
- **def-use graph**: un GFC(p) in cui ogni nodo n è etichettato anche con gli insiemi $def(n)$, $c-use(n)$, $p-use(n)$;

una particolare classe di cammini:

- **definition clear path (def-clear)**: con riferimento ad una variabile v , un cammino $\langle i, n_1, \dots, n_K, J \rangle$ si dice def-clear(v) se per ogni nodo n_i del cammino $v \notin DEF(n_i)$.



Tecniche di testing

- **all-du-path**: ogni cammino def-use deve essere eseguito almeno una volta
- **all-uses**: almeno un def-clear-path da ogni definizione ad ogni uso deve essere eseguito
- **all-p-uses/some-c-uses**: almeno un def-clear path da ogni definizione ad ogni p-use deve essere eseguito; se vi sono definizioni senza p-use, per esse va eseguito almeno un def-clear path dalla definizione ad un c-use
- **all-c-uses/some-p-uses**: almeno un def-clear path da ogni definizione ad ogni c-use deve essere eseguito; se vi sono definizioni senza c-use, per esse va eseguito almeno un def clear path dalla definizione ad un p-use
- **all-p-uses**: almeno un def-clear path da ogni definizione ad ogni suo p-use deve essere eseguito
- **all-defs**: almeno un def-clear path da ogni definizione ad un suo uso

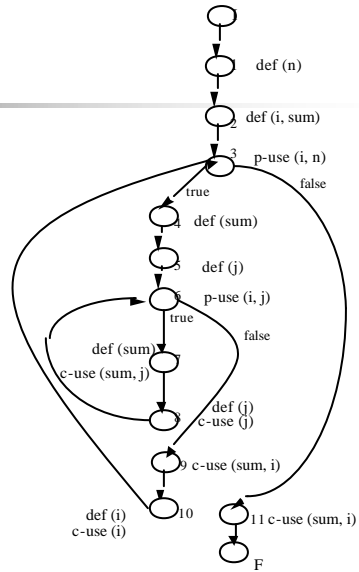


Program Sums

```

1 read(n);
2 i = 1; sum = 0;
3 while (i<=n) do
4   sum= 0;
5   j = 1;
6   while (j <= i) do
7     sum = sum +
j;
8     j = j + 1;
9   endwhile;
10  write (sum,j);
11  i = i + 1;
12 endwhile;
13 write (sum,i);
14 end Sums.

```



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

Program Sums

```

1 read(n);
2 i = 1; sum = 0;
3 while (i<=n) do
4   sum= 0;
5   j = 1;
6   while (j <= i) do
7     sum = sum + j;
8     j = j + 1;
9   endwhile;
10  write (sum,j);
11  i = i + 1;
12 endwhile;
13 write (sum,i);
14 end Sums.

```

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

* triple infeasible

Test cases per all-uses

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