
Progettazione delle prove (2)

- Criteri White box
- Test mutazionale
- Oracolo

Laura Semini, Ingegneria del Software

Dipartimento di Informatica, Università di Pisa



Criteri white box

Sono criteri per l'individuazione dei casi di input che **si basano sulla struttura del codice**

- Sinonimi:
 - Criteri strutturali
 - Criteri a scatola aperta

Perché criteri basati sul codice

- I criteri strutturali che vediamo oggi devono aiutare ad aggiungere altri test
 - Oltre a quelli generati con criteri funzionali
- Rispondono alla domanda:
 - “Quali altri casi devo aggiungere per far emergere malfunzionamenti che non sono apparsi con il testing fatto con casi di prova basati su criteri black-box?”
- Per abuso di linguaggio si parla di white/black-box testing: è solo la progettazione white/black box, non il testing!

Gli elementi di un flusso di controllo

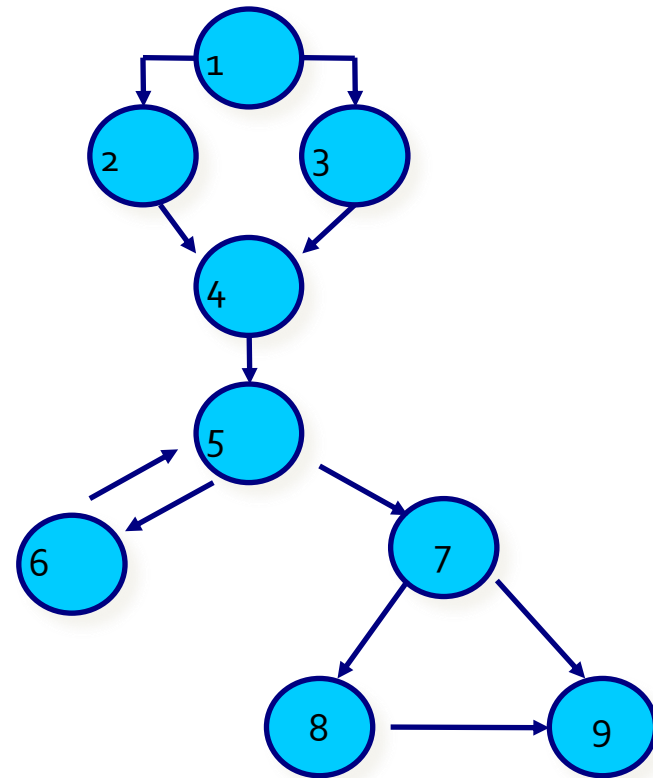
- Banalmente potremmo dire che un programma non è testato adeguatamente se alcuni suoi elementi non vengono mai esercitati dai test.
- I criteri strutturali di progettazione di casi di test (aka control flow testing) sono definiti per classi particolari di elementi e richiedono che i test esercitino **tutti** quegli elementi del programma
- Gli elementi possono essere: **comandi**, **branches** (**decisioni**), **condizioni** o **cammini**.

Grafo di flusso

- Grafo di flusso
 - definisce la **struttura del codice** identificandone le parti e come sono collegate tra loro
 - è ottenuto a partire dal codice
- I diagrammi a blocchi (detti anche diagrammi di flusso o **flow chart**) sono un linguaggio di modellazione grafico per rappresentare algoritmi (in senso lato)

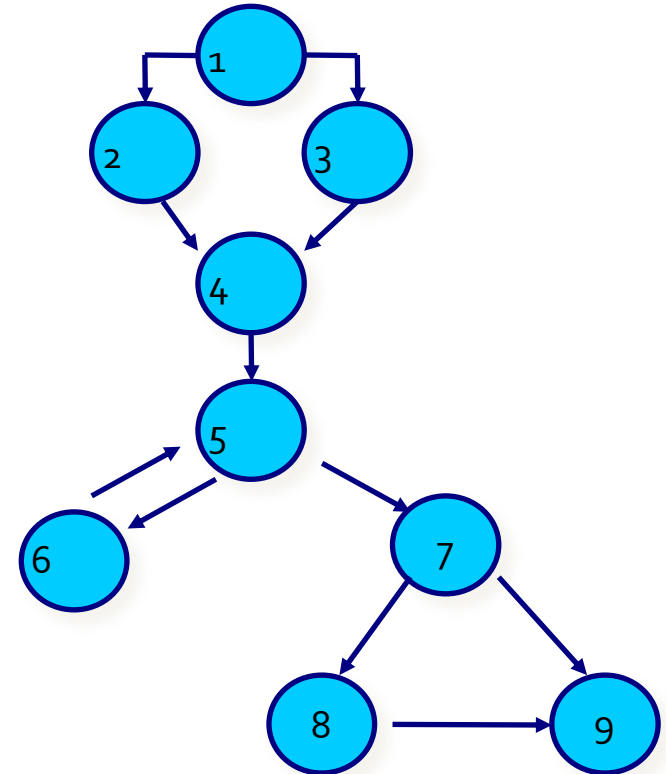
Un esempio di grafo di flusso

```
double eleva(int x, int y) {  
  1. if (y<0)  
  2.     pow = 0-y;  
  3.     else pow = y;  
  4. z = 1.0;  
  5. while (pow!=0)  
  6.     { z = z*x; pow = pow-1 }  
  7. if (y<0)  
  8.     z = 1.0 / z;  
  9. return(z);  
}
```



Criterio: copertura dei comandi

```
double eleva(int x, int y) {  
  1. if (y<0)  
  2.     pow = 0-y;  
  3.     else pow = y;  
  4. z = 1.0;  
  5. while (pow!=0)  
  6.     { z = z*x; pow = pow-1 }  
  7. if (y<0)  
  8.     z = 1.0 / z;  
  9. return(z);  
}
```



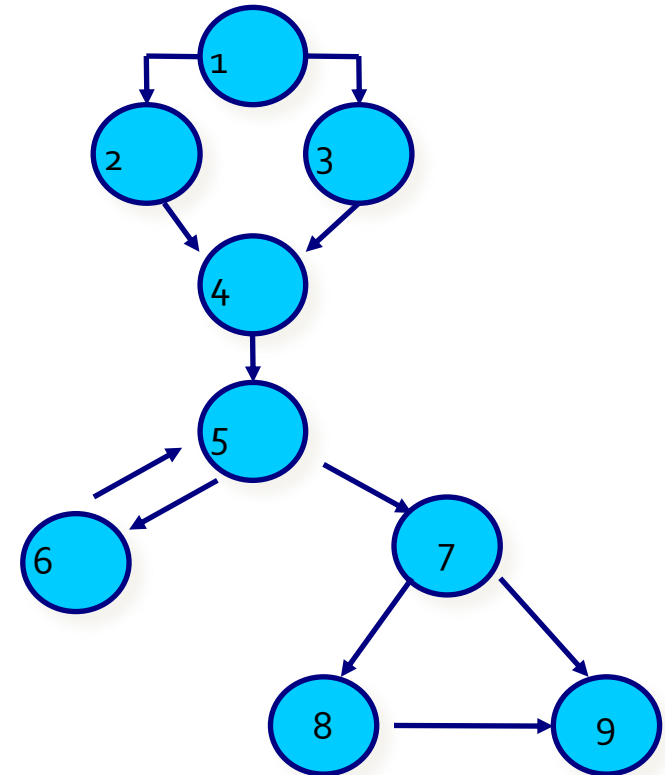
Insieme di valori per x e y che esercitino i comandi

Esempi:

- $\{(x=2,y=2), (x=0,y=0)\}$ (non esercita tutti i comandi)
- $\{(x=-2,y=3), (x=4,y=0), (x=0, y=-5)\}$ (esercita tutti i comandi)

Copertura dei comandi: misura (è una misura di adeguatezza)

```
double eleva(int x, int y) {  
  1. if (y<0)  
  2.     pow = 0-y;  
  3.     else pow = y;  
  4. z = 1.0;  
  5. while (pow!=0)  
  6.     { z = z*x; pow = pow-1 }  
  7. if (y<0)  
  8.     z = 1.0 / z;  
  9. return(z);  
}
```



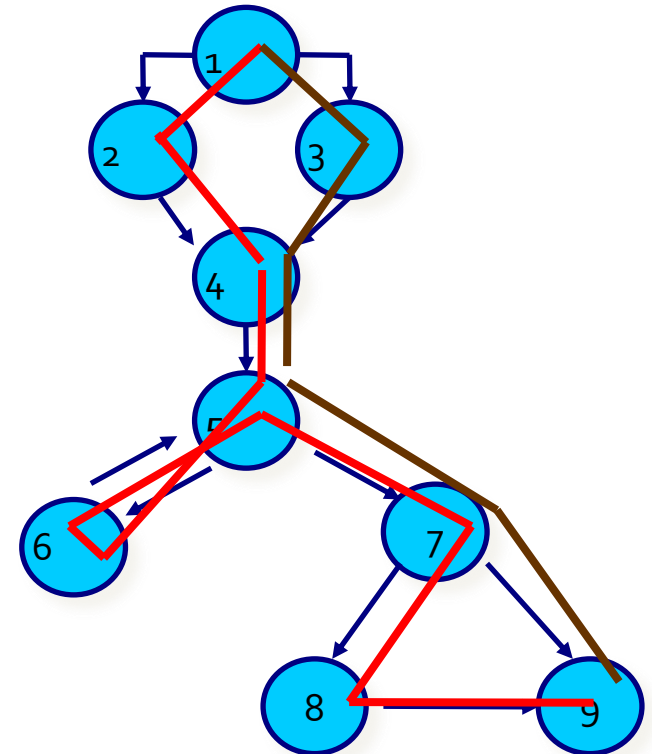
Misura di copertura = $\frac{\text{numero di comandi esercitati}}{\text{numero di comandi totali}}$

Misura di copertura dei comandi: esempio

```
double eleva(int x, int y) {  
  1. if (y<0)  
  2.     pow = 0-y;  
  3.     else pow = y;  
  4. z = 1.0;  
  5. while (pow!=0)  
  6.     { z = z*x; pow = pow-1 }  
  7. if (y<0)  
  8.     z = 1.0 / z;  
  9. return(z);  
}
```

Per avere una copertura totale servono almeno due casi di test: uno con $y < 0$ e uno con $y \geq 0$. In particolare:

- $\{(x=2, y=-2)\}$ esercita i comandi lungo il cammino rosso ed ha una copertura di $8/9=89\%$
- $\{(x=2, y=0)\}$ esercita i comandi lungo il cammino marrone ed ha una copertura di $6/9=66\%$
- $\{(x=2, y=-2), (x=2, y=0)\}$ ha una copertura di $9/9=100\%$

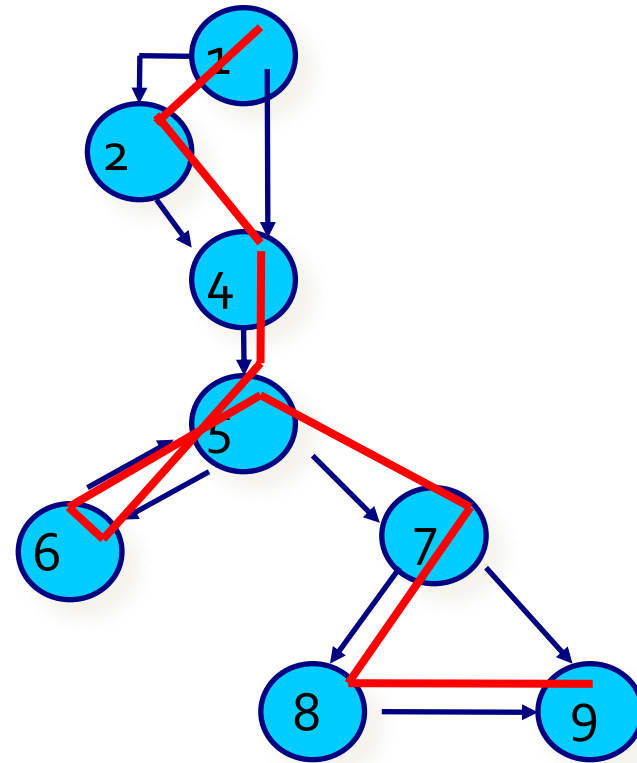


Copertura dei comandi

- La copertura non è monotona rispetto alla dimensione dell'insieme di test:
- $\{(x=4, y=-2)\}$ ha una copertura più alta rispetto a $\{(x=2, y=0), (x=-2, y=2)\}$
- Al solito, cerchiamo di minimizzare il numero di test, a parità di copertura
- Ma non sempre vale la pena cercare a tutti i costi un insieme minimale che dia copertura al 100%

Criterio: copertura delle decisioni

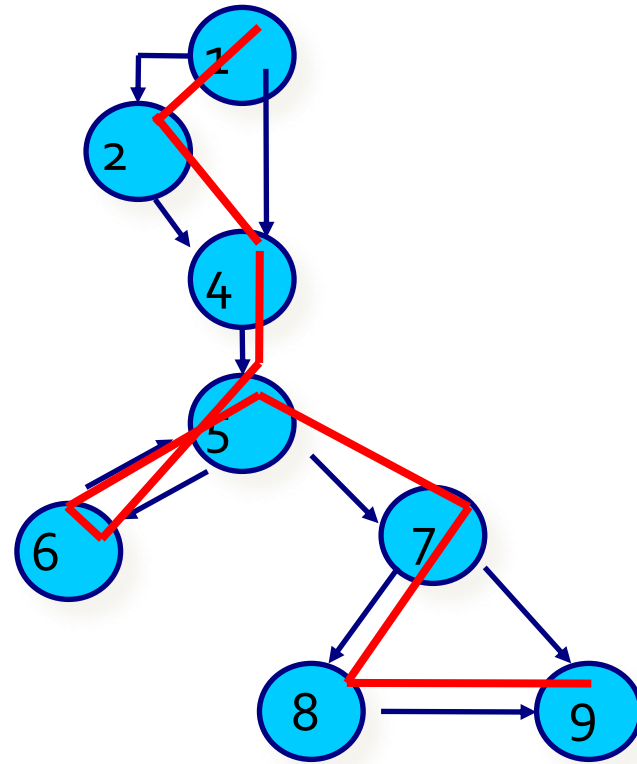
```
double eleva(int x, int y){  
    pow=y;  
    if (y<0)  
        pow = -pow;  
    z = 1.0;  
    while (pow!=0)  
        { z = z * x; pow = pow-1}  
    if (y<0)  
        z = 1.0 / z;  
    return(z);  
}
```



Con (x=2,y=-1) si esercitano tutti i comandi ma manca il ramo else
Bisogna avere casi di test che esercitino entrambi i rami di ogni condizione

Copertura delle decisioni: misura

```
double eleva(int x, int y){  
    pow=y;  
    if (y<0)  
        pow = -pow;  
    z = 1.0;  
    while (pow!=0)  
        { z = z * x; pow =  
pow-1}  
    if (y<0)  
        z = 1.0 / z;  
    return(z);  
}
```



Per avere una copertura delle decisioni devo avere almeno due casi di test uno $y < 0$ e uno $y \geq 0$. Devo coprire tutte le frecce!

Misura di copertura = $\frac{\text{numero di archi esercitati}}{\text{numero di archi totali}}$

Condizioni composte

- si consideri il codice

```
if (x>1 || y==0) {comando1}  
else {comando2}
```

- Il test $\{x=0, y=0\}$ e $\{x=0, y=1\}$ garantisce la piena copertura delle decisioni, ma non esercita tutti i valori di verità della prima condizione
 - in particolare avrei potuto voler scrivere $x<1$ e aver invertito il verso del $<$
- Il test $\{x=2, y=2\}$ e $\{x=0, y=0\}$ esercita i valori di verità delle due condizioni (ma non tutte le decisioni)
- Il test $\{x=2, y=1\}$, $\{x=0, y=0\}$, $\{x=1, y=1\}$ esercita tutti i valori di verità delle due condizioni e tutte le decisioni

Copertura di condizioni semplici

- Un insieme di test T per un programma P copre tutte le condizioni semplici (basic condition) di P se, per ogni condizione semplice CS in P , T contiene un test in cui CS vale **true** e un test in cui CS vale **false**
- Copertura delle basic condition=
n. di valori di verità assunti dalle basic conditions
 2^n . di basic conditions

Multiple condition coverage

- si consideri il codice

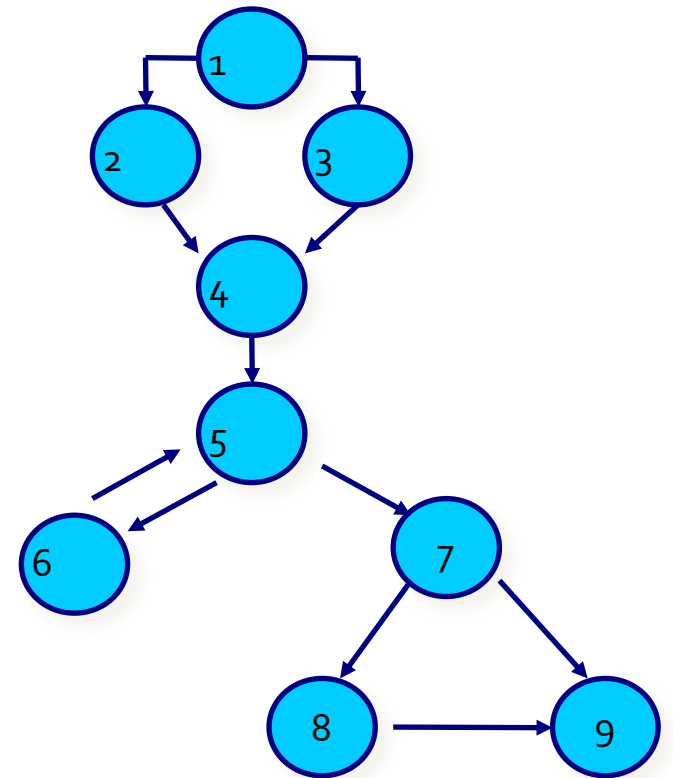
```
if (x>1 && y==0 && z>3) {comando1}
```

```
else {comando2}
```

- La multiple condition coverage richiede di testare tutte le possibili combinazioni (2^n con n condizioni semplici)
- Nell'esempio sarebbero 2^3 casi, ma (semantica Java di &&) ci si può ridurre da a 4:
 - vero, vero, vero
 - vero, vero, falso
 - vero, falso, -
 - falso, - , -

Copertura dei cammini

- Richiede di percorrere tutti i cammini
- In presenza di cicli il numero di cammini infinito
- Per limitare il numero di cammini da a casi di test che esercitino il ciclo
 - 0 volte,
 - esattamente una volta
 - più di una volta
- Alcuni cammini impossibili (1245679)



Fault based testing (e test mutazionale)



Fault based testing

- Ipotizza dei difetti potenziali nel codice sotto test
- Crea o valuta una test suite sulla base della sua capacità di rilevare i difetti ipotizzati
- La più nota tecnica di fault based testing è il **test mutazionale**
 - **Si iniettano difetti modificando il codice**

Test mutazionale per valutare la qualità di una batteria di test

Specifica: la funzione foo restituisce $x+y$ se $x \leq y$ e $x*y$ altrimenti

\\originale

```
int foo(int x, int y)
{ if(x <= y)
    return x+y;
  else return x*y; }
```

\\versione modificata (mutante)

```
int foo(int x, int y)
{ if(x < y)
    return x+y;
  else return x*y; }
```

Consideriamo la seguente batteria di test

{<(0,0), 0>, <(2,3), 5>, <(4,3), 12>}

Non fallisce né nell'originale né nella versione modificata

→ si dice che il mutante non viene ucciso

→ la batteria è poco efficace e va riprogettata

(anche se copre: criteri strutturali : tutte le decisioni, tutte le istruzioni, criteri funzionali : le classi di equivalenza e la frontiera)

Dobbiamo contare i pesci in un lago



Contiamo i pesci nel lago



Mettiamo M pesci meccanici nel lago che contiene un numero imprecisato di pesci

Osserviamo N pesci e vediamo che di questi N_1 sono quelli meccanici

Contiamo...



Ne avevamo messi M
meccanici

Ne osserviamo N

Di questi N_1 sono quelli
meccanici

$$N_1 : N = M : \text{Total}$$

$$\text{Total} = \frac{N * M}{N_1}$$

Definiamo una versione modificata del programma (mutante)

Specifica: la funzione foo restituisce $x+y$ se $x \leq y$ e $x*y$ altrimenti

\\originale con un difetto

```
int foo(int x, int y)
{ if(x < y)
    return x+y;
  else return x*y; }
```

\\mutante in cui inietto un difetto (M=1)

```
int foo(int x, int y)
{ if(x < y)
    return 3;
  else return x*y; }
```

Consideriamo la seguente batteria di test

{<(0,0), 0>, <(2,3), 5>, <(4,3), 12>}

Permette di evidenziare zero difetti nell'originale

2 difetti ($N=2$) nel mutante, di cui uno iniettato ($N_1=1$)

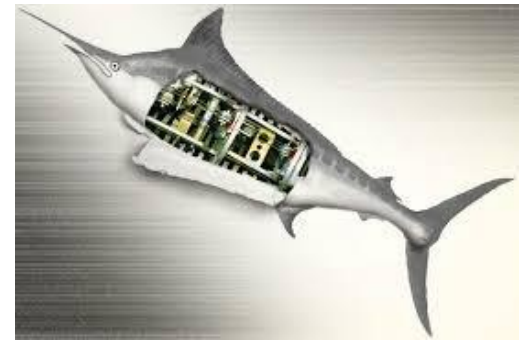
→ Total = $(N*M)/N_1 = 2$ (totale dei difetti inclusi quelli «meccanici»)

Total – $N_1 = 1$

→ Stimo che nella versione originale ci sia un difetto che non avevo trovato

Assunzione

i difetti che mettiamo sono rappresentativi di quelli che potrebbero esserci davvero



Test mutazionale

- Dopo aver esercitato un programma P su una batteria di test T , e aver verificato P corretto rispetto a T
- si vuole fare una verifica più profonda sulla correttezza di P :
 - si introducono dei difetti (piccoli, dette **mutazioni**) su P
 - si chiami P' il programma modificato: P' viene detto **mutante**
- Si eseguono su P' gli stessi test di T . Il test dovrebbe manifestare dei malfunzionamenti.
 - Se il test non rileva questi difetti, allora significa che la batteria di test non era abbastanza buona
 - Se li rivela, abbiamo una maggior fiducia sulla batteria di test.
- Questo è un metodo per valutare la capacità di un test, e valutare se è il caso di introdurre test più sofisticati.

Test mutazionale

- **mutazione**: cambiamento sintattico (un bug inserito nel codice)
 - Esempio: modifica $(i < 0)$ in $(i \leq 0)$
- Un **mutante** viene **ucciso** se fallisce almeno in un caso di test
- **efficacia di un test** = quantità di mutanti uccisi
- La tecnica si applica in congiunzione con altri criteri di test
- Nella sua formulazione è prevista infatti l'esistenza, oltre al programma da controllare, anche di un insieme di test già realizzati.

Ipotesi del programmatore Competente

I difetti reali sono **piccole variazioni sintattiche** del
programma corretto

=>

Mutanti sono modelli ragionevoli dei programmi con
difetti

Mettendo insieme tutte le ipotesi

Test che trovano semplici difetti allora trovano anche
difetti più complessi

una test suite che uccide i mutanti è capace anche di
trovare difetti reali nel programma

Esempi di mutazioni

- **crp**: sostituzione (replacement) di costante per costante
 - ad esempio: da $(x < 5)$ a $(x < 12)$
- **ror**: sostituzione dell'operatore relazionale
 - ad esempio: da $(x \leq 5)$ a $(x < 5)$
- **vie**: eliminazione dell'inizializzazione di una variabile
 - cambia `int x = 5;` a `int x;`
- **lrc**: sostituzione di un operatore logico
 - Ad esempio da `&` a `|`
- **abs**: inserimento di un valore assoluto
 - Da `x` a `|x|`

Mutazioni per il C

ID	Operator	Description	Constraint
<i>Operand Modifications</i>			
crp	constant for constant replacement	replace constant $C1$ with constant $C2$	$C1 \neq C2$
scr	scalar for constant replacement	replace constant C with scalar variable X	$C \neq X$
acr	array for constant replacement	replace constant C with array reference $A[I]$	$C \neq A[I]$
scr	struct for constant replacement	replace constant C with struct field S	$C \neq S$
svr	scalar variable replacement	replace scalar variable X with a scalar variable Y	$X \neq Y$
csr	constant for scalar variable replacement	replace scalar variable X with a constant C	$X \neq C$
asr	array for scalar variable replacement	replace scalar variable X with an array reference $A[I]$	$X \neq A[I]$
ssr	struct for scalar replacement	replace scalar variable X with struct field S	$X \neq S$
vie	scalar variable initialization elimination	remove initialization of a scalar variable	
car	constant for array replacement	replace array reference $A[I]$ with constant C	$A[I] \neq C$
sar	scalar for array replacement	replace array reference $A[I]$ with scalar variable X	$A[I] \neq X$
cnr	comparable array replacement	replace array reference with a comparable array reference	
sar	struct for array reference replacement	replace array reference $A[I]$ with a struct field S	$A[I] \neq S$
<i>Expression Modifications</i>			
abs	absolute value insertion	replace e by $\text{abs}(e)$	$e < 0$
aor	arithmetic operator replacement	replace arithmetic operator ψ with arithmetic operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
lcr	logical connector replacement	replace logical connector ψ with logical connector ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
ror	relational operator replacement	replace relational operator ψ with relational operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
uoi	unary operator insertion	insert unary operator	
cpr	constant for predicate replacement	replace predicate with a constant value	
<i>Statement Modifications</i>			
sdl	statement deletion	delete a statement	
sca	switch case replacement	replace the label of one case with another	
ses	end block shift	move } one statement earlier and later	

Figure 16.2: A sample set of mutation operators for the C language, with associated constraints to select test cases that distinguish generated mutants from the original program.

Mutanti Validi e Utili

- Un mutante è **invalido** se non è sintatticamente corretto, cioè se non passa la compilazione, è **valido** altrimenti
- Un mutante è **utile** se è **valido** e distinguerlo dal programma originale non è facile, cioè se esiste solo un piccolo sottoinsieme di test che permette di distinguerlo dal programma originale.
- Un mutante è **inutile** se è ucciso da quasi tutti i casi di test
- Trovare mutazioni che producano mutanti validi e utili non è facile e dipende dal linguaggio

Come sopravvive un mutante

- Un mutante può essere **equivalente** al programma originale
 - Cambiare $(x \leq 0)$ a $(x < 0 \text{ OR } x=0)$ non ha cambiato affatto l'output: La mutazione non è un vero difetto
 - Determinare se un mutante è equivalente al programma originale può essere facile o difficile; nel peggiore dei casi è indecidibile
- Oppure la suite di test potrebbe essere inadeguata
 - Se il mutante poteva essere stato ucciso, ma non lo era, indica una debolezza nella suite di test

Esempi

\\originale

```
int foo(int x, int y)
{ if(x < y)
    return x+y;
  else return x*y;}
```

\\invalido

```
int foo(int x, int y)
{ if(x < "a")
    return x+y;
  else return x*y;}
```

\\inutile

```
int foo(int x, int y)
{ if(x < y)
    return x*y;
  else return x*y;}
```

\\equivalente

```
int foo(int x, int y)
{ if(x < y)
    return x+y+1-1;
  else return x*y;}
```

Esercizio

```
public void insertionSort(int [] array, int min, int max) {}
    for(int i = min+1; i < max; i++) {
        temp=a[i];
        j=i-1;
        while(j>=0 && a[j]>temp) {    a[j+1]=a[j];    j--; }
        a[j+1]=temp;
    }
    for(i=0;i<10;i++){
        printf("%d \t", a[i]);
    }
}
```

TS = {TC1, TC2}

TC1 = final int[] array = { 5, 9, 0, 2, 7, 3 }; insertionSort(array, 0, 6);

TC2 = final int[] array2 = { 3, 1, 0, 2, 7, 3 }; insertionSort(array2, 2, 4);

Fornire due serie di mutazioni M1 e M2 tali che:

1. Il mutante ottenuto applicando M1 viene ucciso da TS
2. Il mutante ottenuto applicando M2 NON è ucciso da TS.

Test mutazionale: discussione

- Questa strategia è adottata con obiettivi diversi
 - favorire la scoperta di malfunzionamenti ipotizzati: intervenire sul codice può essere più conveniente rispetto alla generazione di casi di test ad hoc.
 - valutare l'efficacia dell'insieme di test, controllando se “si accorge” delle modifiche introdotte sul programma originale.
 - cercare indicazioni circa la localizzazione dei difetti la cui esistenza è stata denunciata dai test eseguiti sul programma originale
- Uso limitato dal gran numero di mutanti che possono essere definiti, dal costo della loro realizzazione, e soprattutto dal tempo e dalle risorse necessarie a eseguire i test sui mutanti e a confrontare i risultati

L'oracolo e l'individuazione degli output attesi



Motivazione

- Inutile produrre automaticamente 10.000 casi di input se l'output atteso deve essere calcolato a mano!

Come trovare l'output atteso

- Risultati ricavati dalle specifiche
 - specifiche formali
 - specifiche eseguibili
- Inversione delle funzioni
 - quando l'inversa è "più facile"
 - a volte disponibile fra le funzionalità
 - limitazioni per difetti di approssimazione
 - Partire dall'output e trovare l'input
 - Per esempio per testare un algoritmo di ordinamento prendere un array ordinato (output atteso) e rimescolarlo per ottenere un input

Come trovare l'output atteso

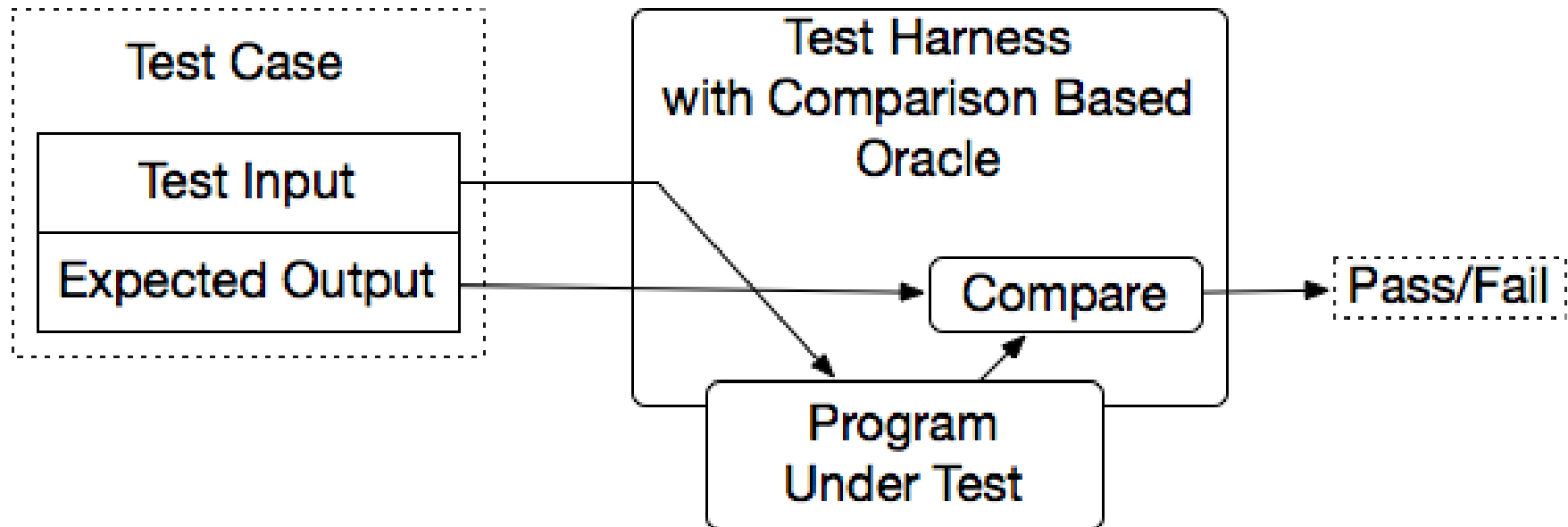
- Versioni precedenti dello stesso codice
 - disponibili (per funzionalità non modificate)
 - prove di non regressione
- Versioni multiple indipendenti
 - programmi preesistenti (back-to-back)
 - sviluppate ad hoc
 - semplificazione degli algoritmi
 - magari poco efficienti ma corrette

Come trovare l'output atteso

- Semplificazione dei dati d'ingresso
 - provare le funzionalità su dati semplici
 - risultati noti o calcolabili con altri mezzi
 - ipotesi di comportamento costante

- Semplificazione dei risultati
 - accontentarsi di risultati plausibili
 - tramite vincoli fra ingressi e uscite
 - tramite invarianti sulle uscite

Automazione dell'architettura di test



Syllabus

- Cap 12-16-17
 - Software Testing and Analysis: Process, Principles and Techniques- Mauro Pezzè e Michal Young
- In particolare:
 - Cap 12: tutto tranne 12.6
 - Cap 16: tutto tranne 16.5
 - Cap 17: in dettaglio solo 17.5