Presentazione presso Libreria FELTRINELLI - Pisa
1 dicembre 2023 ore 18.00

Letture a cura di: Clotilde Luisa Borelli

Interviene: prof.ssa Linda Pagli

Sarà presente l'autore

Hedy
(U.S. Patent n. 2.292.387)

di Silvio Melani

LA VITA FUORI DAGLI SCHEMI DI HEDY LAMARR, UN'ATTRICE HOLLYWOODIANA DI SUCCESSO E UN'INCREDIBILE INVENTRICE.

Hedy Lamarr, celebre attrice degli anni Trenta/Cinquanta, nonché inventrice della tecnologia alla base delle comunicazioni wireless attuali.

Se oggi possiamo telefonare in sicurezza con gli smartphones, lo dobbiamo alla tecnologia del “salto di frequenza” (frequency hopping) da lei ideata tra il 1941 e il 1942.

La professoressa **Linda Pagli**, ordinario di informatica all'Università di Pisa e appassionata studiosa delle storie di donne che hanno lasciato un'impronta nella scienza e nella tecnologia, condurrà la presentazione del libro "**Hedy (U.S. Patent n. 2.292.387)**".

Lo farà dialogando con il suo autore, **Silvio Melani**, filologo, storico e saggista, che ne ha ricostruito la storia, svelandola attraverso una lente letteraria e tecnologica.

Progettazione delle prove (2)
- Criteri White box
- Test mutazionale
- Oracolo

Laura Semini, Ingegneria del Software

Dipartimento di Informatica, Università di Pisa

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

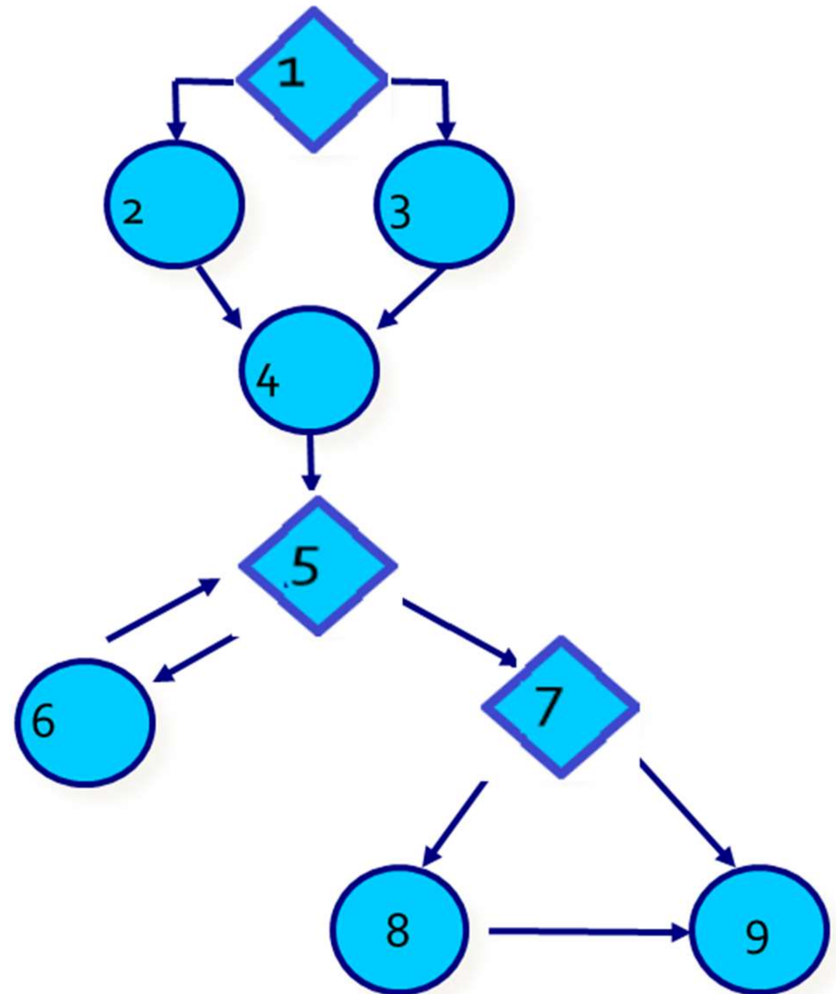
- I diagrammi a blocchi (detti anche diagrammi di flusso o flow chart) sono un linguaggio di modellazione grafico per rappresentare algoritmi (in senso lato)

Un esempio di grafo di flusso

```
double eleva(int x, int y)
```

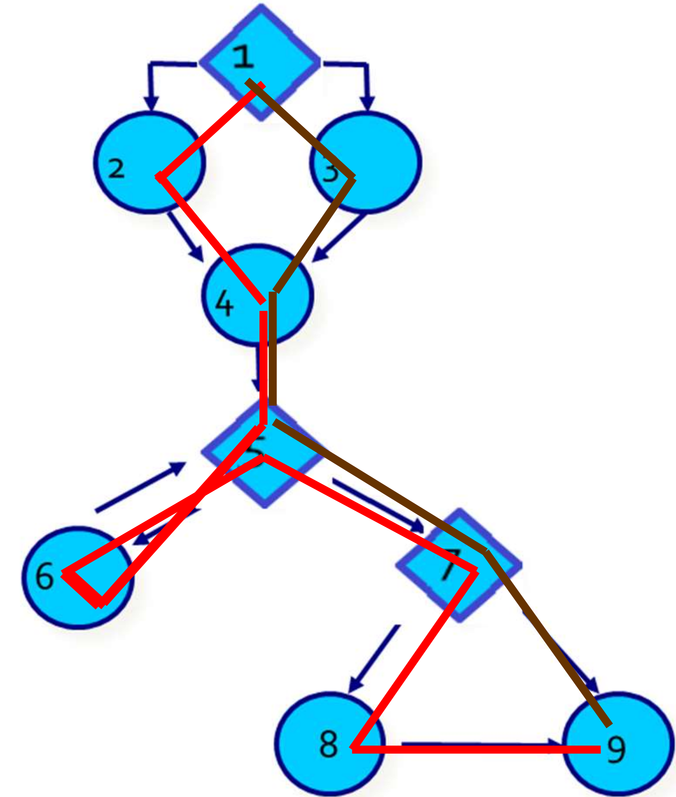
Restituisce: x^y

```
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);
}
```

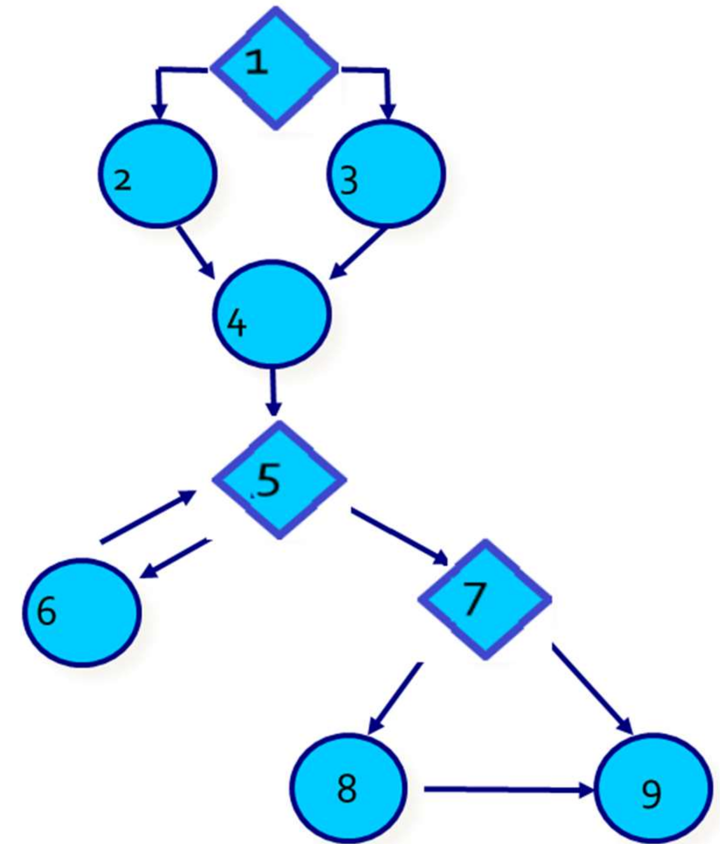


Si cercano valori per x e y che esercitino tutti i comandi

- {(x=0,y=0)} (non esercita tutti i comandi)
- { (x=0,y=0), (x=2,y=2)} (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);
}
```



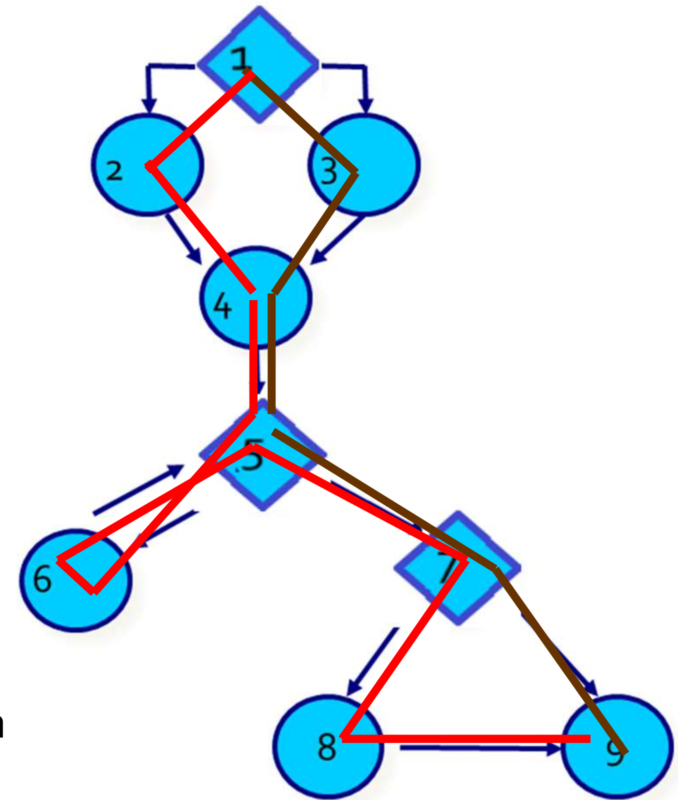
$$\text{Misura di copertura} = \frac{\text{numero di comandi esercitati}}{\text{numero di comandi totali}}$$

Misura di copertira 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<o e uno con y>=o. 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

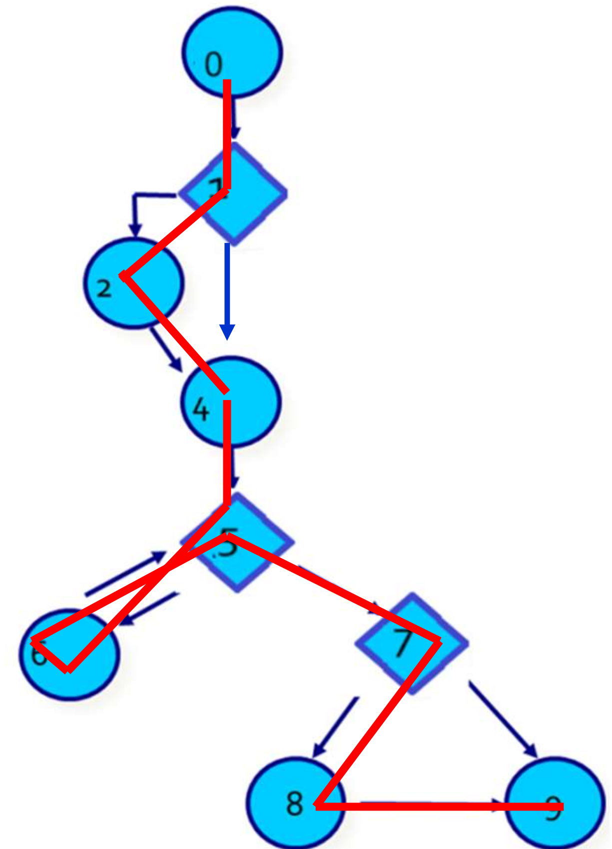
- La copertura non è monotona rispetto alla dimensione dell'insieme di test:

- {(x=4,y=-2)} ha una copertura più alta rispetto a {(x=2,y=0), (x=-2,y=2)}

- Al solito, cerchiamo di minimizzate 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

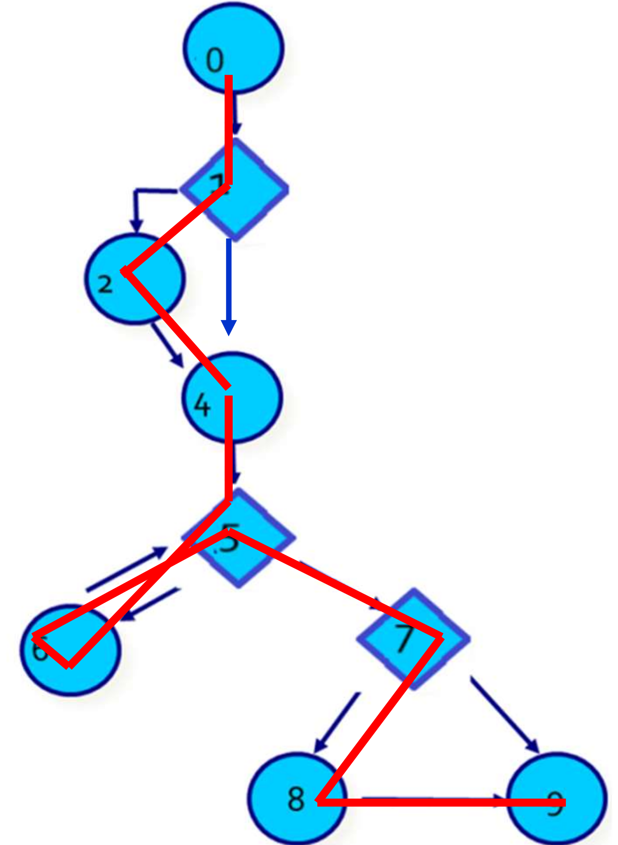
```
double eleva(int x, int y){
  pow=y;
  if (y<0)
      pow = -pow;
  z = 1.0;
  while (pow!=0)
       { z = z * x; pow = pow-1}
  if (y<0)
       z = 1.0 / z;
  return(z);
     }
```



Con (x=2,y=-1) si esercitano tutti i comandi ma…
bisogna avere casi di test che esercitino entrambi i rami di ogni condizione
Per avere una copertura delle decisioni occorre avere almeno due casi di test: uno y<0 e uno y>=0. (Per coprire tutte le frecce)

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);
     }
```



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

Con (x=2,y=-1) copertura di $\frac{9}{11}$ delle decisioni

Condizioni composte

```
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
- 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=0} , {x=0, 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=

$$\frac{\text{n. di valori di verità assunti dalle basic conditions}}{\text{2*n. di basic conditions}}$$

Condizioni composte

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

- Il test {x=0, y=0} e {x=0, y=1} ha copertura delle condizioni semplici: $\frac{3}{4}$
- Il test {x=2, y=2} e {x=0, y=0} ha copertura delle condizioni semplici: $\frac{4}{4}$
- Il test {x=2, y=0} , {x=0, y=1} ha copertura delle condizioni semplici: $\frac{4}{4}$

Multiple condition coverage

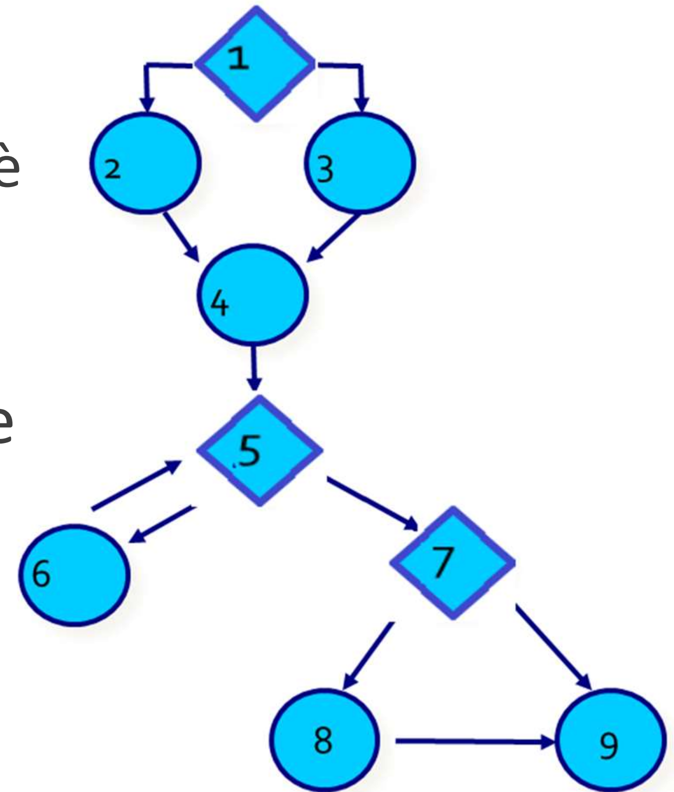
- si consideri il codice

```
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
  - Cresce in modo esponenziale con il numero di decisioni
  - In presenza di cicli il numero di cammini è potenzialmente infinito
- Per limitare il numero di cammini da attraversare si richiedano casi di test che esercitino il ciclo
  - 0 volte,
  - esattamente una volta
  - più di una volta
- Nota: alcuni cammini sono 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

- Precondizione: 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 sella 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.

Che cos'è una mutazione?

- Una **mutazione** è un piccolo cambiamento in un programma.
  - *Esempio*: si cambia (i <0) in (i <= 0)

  - *Ipotesi*: I difetti reali sono piccole variazioni sintattiche del programma corretto =>
  - Mutanti sono modelli ragionevoli dei programmi con difetti

Che cos'è il test mutazionale?

- E' un metodo di test strutturale volto a
  - **valutare/migliorare l'adeguatezza delle suite di test** e a
  - **stimare il numero di difetti nei sistemi sotto test**.
- Il processo, dato un programma P e una suite di test T, è il seguente:
  1. Applichiamo delle mutazioni a P per ottenere una sequenza P1, P2,... Pn di mutanti di P.
  2. Ogni mutante deriva dall'applicazione di una singola operazione di mutazione a P.
  3. Si esegue la suite di test T su ciascuno dei mutanti

Si dice che **T uccide** il **mutante Pj** se rileva un malfunzionamento:

- il mutante viene ucciso se fallisce almeno in un caso di test di T
- Si dice anche che il caso di test in questione ha ucciso il mutante

Test mutazionale

- ***Efficacia di un test*** = quantità di mutanti uccisi/numero mutanti
  - Se T uccide k mutanti su n, l’efficacia di T è k/n.
- Un mutante **sopravvive a una test suite** se per tutti i test case della test suite non si distingue l’esito del test se eseguito sul programma originale o su quello mutante

- 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.
- Uno dei vantaggi di questo approccio è che può essere quasi completamente automatizzato

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

Test mutazionale per valutare la qualità di una batteria di test

Specifica: la funzione foo restituisce x+y se x<=y e x*y altrimenti

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

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

- Consideriamo la batteria di test: {<(0,0), 0>, < (2,3), 5>, < (4,3), 12>}
- Il mutante non viene ucciso (sopravvive)
- la batteria è poco adeguata e va riprogettata
  - anche se copre:
    - criteri strutturali : tutte le decisioni, tutte le istruzioni,
    - criteri funzionali : le classi di equivalenza e la frontiera

Test mutazionale per stimare il numero di difetti nel sistema

Contiamo i pesci nel lago

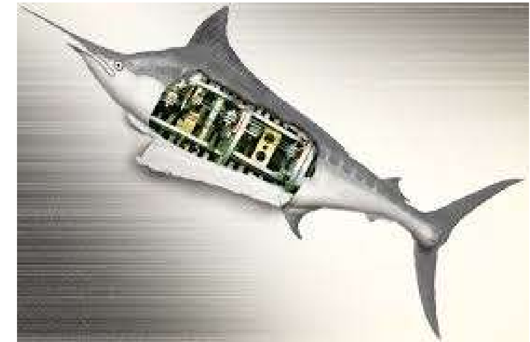
Contiamo i pesci nel lago

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

Osserviamo N pesci e vediamo che **di questi N, N1 sono meccanici**

Assunzione

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



Contiamo…

Ne avevamo messi M meccanici

Ne osserviamo N

Di questi N_1 sono meccanici

$N_1 : N = M : Total$

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

Definiamo una versione modificata del programma (mutante)

Specifica: la funzione foo restituisce x+y se x<=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 batteria di test {<(0,0), 0>, < (2,3), 5>, < (4,3), 12>}
- Evidenzia zero difetti nell'originale
- Evidenzia 2 difetti (N=2) nel mutante, di cui uno iniettato (N1=1)
  - Total = (N*M)/N1= 2 (totale dei difetti inclusi quelli «meccanici»)
  - Total – N1=1
- Stimo che nella versione originale ci sia un difetto che non avevo trovato

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 <= 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
Operand Modifications			
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	$A[I] \neq S$
Expression Modifications			
abs	absolute value insertion	replace e by 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	
Statement Modifications			
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 sintattticamente 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 esiste solo un piccolo sottoinsieme di casi 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 <=0) a (x < 0 OR x=0) non ha cambiato 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;}
```

```
\\equivalemte
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>=min && a[j]>temp)   {   a[j+1]=a[j];       j--;    }
        a[j+1]=temp;
 }
 for(i=0;i<10;i++){
    printf("%d \t", a[i]); }
}
```

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.

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);

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

Oracolo

Un "oracolo" è uno strumento o una metodologia utilizzati per generare i risultati attesi di un test case.

L'importanza dell'oracolo risiede nel fatto che fornisce **un punto di riferimento** per valutare la correttezza del sistema durante il processo di testing.

Un buon oracolo è cruciale per garantire la qualità del software e la corretta individuazione di eventuali difetti.

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

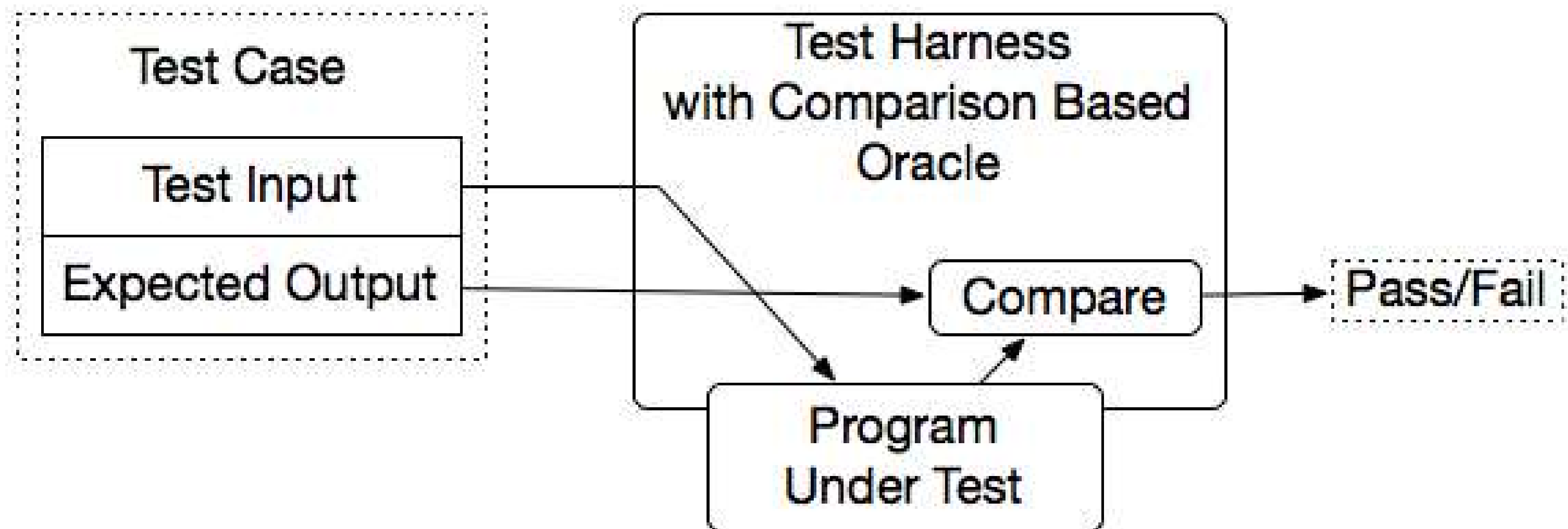
- **Versioni multiple indipendenti**
  - programmi preesistenti (back-to-back)
  - sviluppate ad hoc
  - semplificazione degli algoritmi
  - magari poco efficienti ma corrette

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



Test di sistema

Facility test (test delle funzionalità)

- È il più intuitivo dei controlli, quello cioè che mira a controllare che ogni funzionalità del prodotto stabilita nei requisiti sia stata realizzata correttamente.

Security test

- Cercando di accedere a dati o a funzionalità che dovrebbero essere riservate,si controlla l’efficacia dei meccanismi di sicurezza del sistema

Test di sistema

Usability test

si vuole valutare la facilità d'uso del prodotto da parte dell'utente finale.

- Valutazione fra le più soggettive;
- prodotto + documentazione + livello di competenza dell'utenza + caratteristiche operative dell'ambiente d'uso del prodotto.

Performance test

È un controllo mirato a valutare l'efficienza di un sistema rispetto ai tempi di elaborazione e ai tempi di risposta.

- controllo critico per esempio per i sistemi in tempo reale
- Il sistema viene testato a diversi livelli di carico, per valutarne le prestazioni

Test di sistema

Volume test (o load test, test di carico)

il sistema è sottoposto al carico di lavoro massimo previsto dai requisiti e le funzionalità sono controllate in queste condizioni.

- Lo scopo è sia individuare malfunzionamenti che non si presentano in condizioni normali, quali difetti nella gestione della memoria, buffer overflows, etc., sia garantire un'efficienza base anche in condizioni di massimo carico.
- Le tecniche e gli strumenti come per il performance test: , i due tipi di test hanno scopi molto differenti, da un lato valutare le prestazioni a vari livelli di carico, non limite, dall'altro valutare il comportamento del sistema sui valori limite.

Test di sistema

Stress test

Il sistema è sottoposto a carichi di lavoro superiori a quelli previsti dai requisiti o è portato in condizioni operative eccezionali – in genere sottraendogli risorse di memoria e di calcolo.

- esplicito superamento dei limiti operativi previsti dai requisiti.
- Lo scopo è quello di controllare la capacità di “recovery” (recupero) del sistema dopo un fallimento.

Storage use test

È ancora un controllo legato all’efficienza di un sistema, ma mirato alla richiesta di risorse – la memoria in particolare – durante il funzionamento, e ha implicazioni sull’ambiente operativo richiesto per poter installare il sistema.

Test di sistema

Configuration test

obiettivo la prova del sistema in tutte le configurazioni previste:

- piattaforme di installazione diverse per sistema operativo o dispositivi hardware installati,
- insiemi di requisiti funzionali leggermente diversi.

Compatibility test

obiettivo valutare la compatibilità del sistema con altri prodotti software.

- versioni precedenti dello stesso prodotto,
- sistemi diversi, ma funzionalmente equivalenti che il prodotto deve rimpiazzare,
- altri sistemi software con cui il prodotto deve interagire

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