Testing basato sui programmi

Definizioni

Programma

> Un **programma** P è una *funzione* da un dominio D ad un codominio R:
>
> $$P : D \mapsto R$$

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**, cioè se $\forall d \in D, OK(P, d)$

Testing

Inoltre ricordiamo la terminologia del testing, assumendo un programma P corretto:

- **Malfunzionamento:**
  Eseguo P con d e non ottengo $P(d)$ ma:
  - ottengo un altro valore $P'(d)$
  - il programma si blocca

  Quindi, avremo $\neg OK(P, d)$
- **Difetto**(***bug***):
  Il programma P ha un'implementazione P', e se $P' \neq P$, la diversità sarà il **bug**
- **Errore:**
  Il **motivo** del bug

Casi di test e test set

> Un **caso di test** (o **test**) è un elemento $d \in D$

H4

> Un **test set** T è un *sottoinsieme finito* di D

Un **test set** può essere:

- **negativo**:

  > Un test set si dice **negativo** se vale $OK(P, T)$, cioè se $\forall t \in T, OK(P, t)$

- **positivo**:

  > Un test set si dice **positivo** se vale $\neg OK(P, T)$, cioè se $\exists t \in T \mid \neg OK(P, t)$

Test ideale

Si vorrebbe dimostrare la correttezza di un programma attraverso un **test set** finito. Se tale **test set** esiste viene detto **test ideale**:

H4

> Un test set T è **ideale** se:
>
> $$OK(P, T) \implies OK(P)$$
>
> cioè se il test negativo su T implica che P sia corretto.

Test esaustivo

Un **test ideale** è un **test esaustivo**:

H5

> Un test set si dice **esaustivo** se $T = D$, quindi:
>
> $$OK(P, T) \implies OK(P, D)$$

Quindi si considera T l'insieme di tutti i possibili input D, e chiaramente se vale $OK(P,T)$, P si dimostra corretto. Questo approccio ha grossi limiti:

- non sempre D è finito (es. insieme N)
- D può essere troppo grande! Incapacità di esaurire i casi di test in tempi ragionevoli

La soluzione è quindi quella di trovare un *sottoinsieme di* D che corrisponda a qualche *criterio* che renda il test affidabile e valido.

Criterio di test

> Un **criterio di test** è una *funzione* che per un dato programma P e la sua specifica S, preso in input un **test set** T restituisce vero o falso.
>
> $$C : P \times S \times T \mapsto \{true, false\}$$
>
> $$C_{P,S} = \begin{cases} true & \text{se } T \text{ adeguato a trovare ogni bug in P} \\ & \text{rispetto a S secondo il criterio C} \\ false & \text{altrimenti} \end{cases}$$

La definizione appena data descrive un criterio di test che applica contemporaneamente testing white-box e testing black-box. Infatti vengono dati in input sia il programma P sia le specifiche S. Nel caso in cui stiamo facendo esplicitamente un solo tipo di testing avremo i due seguenti casi:

- **White box**
  C non dipende da S:
  $$C : P \times T \mapsto \{true, false\}$$
  $$C_P = \begin{cases} true & \text{se } T \text{ adeguato a trovare ogni bug in P} \\ & \text{secondo il criterio C} \\ false & \text{altrimenti} \end{cases}$$
- **Black box**
  C non dipende da P:
  $$C : S \times T \mapsto \{true, false\}$$

$$C_S = \begin{cases} true & \text{se } T \text{ adeguato a trovare ogni bug} \\ & \text{rispetto a S secondo il criterio C} \\ false & \text{altrimenti} \end{cases}$$

Infine possiamo considerare un **criterio** come un ***generatore*** *di test set*:

> Un **criterio generatore** di test set è una funzione
>
> $$C : P \times S \mapsto T$$
>
> dove T è tale che $C(P, S, T) = true$,
> cioè una funzione che dato P e S genera un **test set adeguato**

I criteri generatori possono anch'essi applicare testing white box o black box:

- **White box**
  C non dipende da S:

$$C : P \mapsto T$$

Criterio affidabile

> Un **criterio** C si dice ***affidabile*** se per ogni coppia di test set T_1 e T_2 *adeguati* secondo C, se T_1 individua un **malfunzionamento**, allora anche T_2 lo individua e viceversa.

Formalmente:

$$\forall T_1, T_2, \quad \neg OK(P, T_1) \iff \neg OK(P, T_2)$$

Con un **criterio affidabile**, $OK(P, T)$ non dipende dal T particolare, in quanto o tutti i test set riescono a trovare i bug, o nessuno di essi. Ciò offre consistenza nel testing.
L'**affidabilità** di un criterio può non essere abbastanza:

```
int raddoppia(int x){
    return x*x;
}
```

Prendo un criterio C_1 che seleziona come test set solo i sottoinsiemi di $\{0, 2\}$:

- $T_1 = \{0\} \rightarrow$ `raddoppia(0) = 0`; non riscontro malfunzionamenti
- $T_2 = \{2\} \rightarrow$ `raddoppia(2) = 4`; non riscontro malfunzionamenti

$C1$ è **affidabile**, poiché sia T_1 che T_2 non hanno osservato malfunzionamenti. Il testing è stato efficace? No! In quanto sappiamo esserci un bug.

Criterio valido

H4

> Un **criterio** C si dice ***valido*** se, qualora il programma P *non sia corretto*, esiste almeno un test set T che soddisfa C che è in grado di individuare il malfunzionamento.

Formalmente:

$$\exists T \mid C_{PS}(T) \wedge \neg OK(P, T)$$

```
int raddoppia(int x){
    return x*x;
}
```

Prendo un criterio C_2 che seleziona qualsiasi $T \subseteq \{0, 1, 2, 3, 4\}$:

- $T_3 = \{3\} \rightarrow$ `raddoppia(3) = 9`; ho individuato un malfunzionamento

C_2 è **valido**, in quanto individua un malfunzionamento. Ma non è affidabile, in quanto al suo interno vi è anche $T_2 = \{2\}$, che come abbiamo già visto non riscontra malfunzionamenti.

Teorema di Goodenough e Gerhart

H4

> Dato un criterio C e un programma P,

$$\left.\begin{cases} C \text{ valido per } P \\ C \text{ affidabile per } P \end{cases}\right\} \implies C \text{ ideale}$$

Quindi dato un test set T che rispetta C, vale che

$$OK(P,T) \implies OK(P)$$

In altre parole se P passa il test set T, allora è corretto, in quanto T è **ideale**.

```
int raddoppia(int x){
    return x*x;
}
```

Prendo il criterio C_3 che seleziona un qualsiasi sottoinsieme di $\mathbb{N}$ contenente almeno un caso di test $t \geq 3$:

- $T_1 = \{3\}$
- $T_2 = \{0, 1, 2, 3\}$
- $T_3 = \{-2, 0, 3, 56\}$
- $\dots$

C_3 è **valido**, poiché con qualsiasi $t \geq 3$ scopro il malfunzionamento, e **affidabile**, poiché tutti i T che seleziono hanno almeno un $t \geq 3$.
C_3 è infatti **ideale** per il teorema appena mostrato.

Limiti del testing

Trovare un criterio valido e affidabile è però molto difficile.
Così come applicare un test set esaustivo non è praticabile.

In generale valgono i seguenti teoremi.

Teorema di Howden

> Non esiste un *algoritmo* che dato un programma P generi un **test ideale finito**.

Teorema di Dijkstra

> Il test di un programma può rilevare la presenza di malfunzionamenti ma mai dimostrarne l'assenza

Criteri di test empirici

Data la difficoltà nel definire criteri ideali, è possibile comunque usarli per avere un certo grado di **copertura**.

> Un **criterio di test empirico** è una *funzione* che per un dato programma P e la sua specifica S, preso in input un **test set** T restituisce un ***grado di copertura***.
>
> $$C : P \times S \times T \mapsto [0, 1]$$

dove:

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

Non esiste un algoritmo che trovi un **criterio di test empirico** con risultato pari a 1 *per tutti* i programmi. Esistono però algoritmi e tecniche che sono in grado di risolverlo per "molti" programmi.

Inoltre nel momento in cui consideriamo solo la struttura del programma e non la specifica, ci addentriamo nel **testing strutturale dei programmi**.

Testing strutturale dei programmi

In questo caso i **criteri** sono scelti soltanto osservando la **struttura** del sorgente. Tali criteri sono detti criteri di test strutturali. Essi si basano sulla **copertura** del programma.
Per **struttura** del programma intendiamo il ***flusso di controllo*** del programma.
La **copertura** sarà quindi quella relativa al **flusso di controllo**.

Vediamo qui alcuni nuovi strumenti che serviranno per i criteri di copertura.

Flusso di controllo

> Rappresentazione tramite **grafo** del codice sorgente dove:
>
> - ogni istruzione è un nodo
> - ogni istruzione è collegata alla successiva mediante una freccia

Le istruzioni di:

- assegnamento
- lettura
- scrittura
- `return`

le rappresentiamo attraverso nodi *circolari*.

Le istruzioni di decisione le rappresentiamo invece con dei *rombi*.

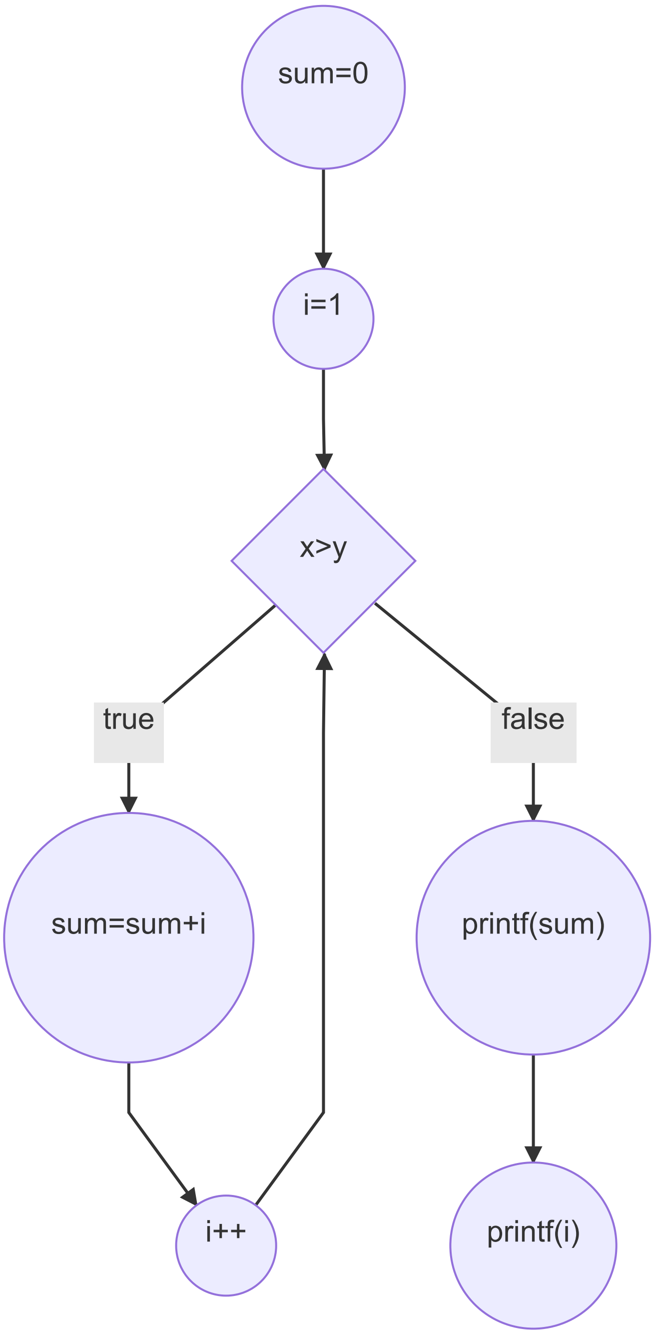
Il seguente programma:

```c
int main(){
    int sum, i;
    sum = 0;
    i = 1;

    while(i < 11){
        sum = sum + i;
        i = i + 1;
    }

    printf("%d", sum);
    printf("%d", i);
}
```

viene rappresentato dal seguente grafo di flusso:



Essendo il grafo del flusso di controllo un'*astrazione* del programma, potenzialmente percorrerlo cambierà ad ogni esecuzione con input diverso!