Convalida e Verifica

- Processi che controllano che il software soddisfi i requisiti del cliente e sia conforme alla sua specifica

- Il sistema dovrebbe essere validato e verificato in ciascuna fase del suo sviluppo

Convalida

"Stiamo costruendo il prodotto giusto?"
Soddisfa i requisiti del cliente?

Verifica

"Stiamo costruendo il prodotto nel modo giusto?"
Rispetta le specifiche date?

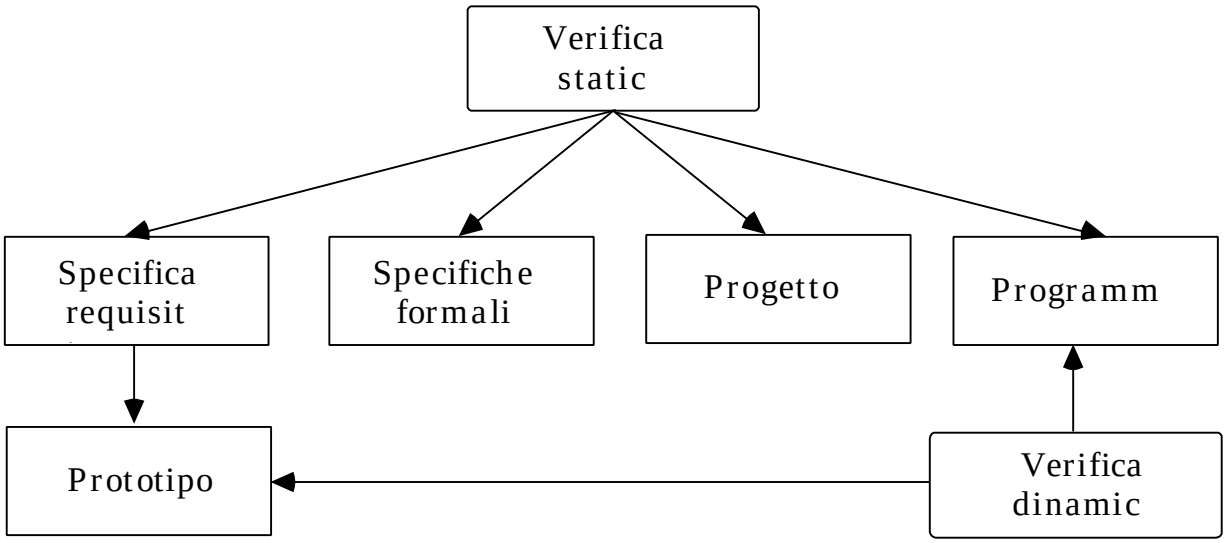
- Utilizzano tecniche sia statiche che dinamiche

- La verifica non coinvolge solo il prodotto finale ma segue passo per passo il progetto e lo sviluppo del prodotto

Tecniche statiche e dinamiche

- Tecniche statiche, si occupano di analizzare e controllare le varie rappresentazioni del sistema (ad es., il documento di specifica dei requisiti, o il codice sorgente prodotto)

- Tecniche dinamiche (o ***test***), verificano la bonta` della realizzazione



- ***Tecniche statiche***, applicabili in tutto il ciclo di vita

  - Includono analisi del programma e verifiche formali

- ***Tecniche dinamiche***, piu` utilizzate nella pratica

  - Si utilizza il programma o il prototipo utilizzando dati simili a quelli che si utilizzeranno una volta installato il sistema

- Il collaudo finale non è che l'ultima fase di un processo di verifica e convalida che si svolge parallelamente al processo di sviluppo del prodotto

- Verifica e convalida richiedono la definizione delle proprietà che caratterizzano la qualità del prodotto, lo studio di metodi per il controllo e la verifica di ciascun passo di sviluppo, la formalizzazione di tecniche per il collaudo del prodotto finale

- Spesso si considera implicitamente la ***correttezza funzionale*** come l'unica qualità che debba (o possa) essere controllata

- Verifica e convalida sono attività complesse, spesso sottovalutate e affidate quasi esclusivamente all'esperienza ed al buon senso del programmatore

Convalida e Verifica (V&V)

- Il primo passo del processo V&V (*verification and validation*) consiste nel determinare quali proprietà siano rilevanti per il prodotto in esame

- Definizione delle proprietà che caratterizzano il prodotto software
  - *correttezza funzionale*
  - *affidabilità, robustezza*
  - *risposte temporali adeguate, ...*

- La determinazione delle proprietà che devono essere garantite da un prodotto software dipende da molti fattori, quali:
  - *il tipo di applicazione*
  - *l'ambiente in cui il software viene utilizzato*
  - *il modo di integrazione ed interazione con l'ambiente in cui il software è utilizzato*
  - *la vita prevista per l'applicazione*
  - *l'evoluzione prevista per l'ambiente di utilizzo durante la vita dell'applicazione*

- Si selezionano le tecniche di sviluppo più adatte per garantire il soddisfacimento di tali proprietà e per cui esistano metodi e strumenti di verifica e convalida

- La disponibilità di tecniche generali di verifica e convalida delle varie fasi di sviluppo software è ancora molto limitata

- La verifica e convalida del ciclo di vita del software è resa difficile dalla mancanza di un linguaggio comune per le fasi che precedono la codifica

- Esistono metodi di verifica e convalida per analizzare correttezza e prestazioni del software (linguaggi di *specifica formale*)

- Lo sviluppo di tecniche di verifica e convalida delle fasi di specifica e progetto è legato alla definizione di linguaggi e strumenti per tali fasi

Convalida e Verifica

- Convalida, a partire dai requisiti (informali) espressi dall'utente

  Le tecniche ed i metodi disponibili sono necessariamente vaghi, incompleti e difficilmente formalizzabili

- Verifica, a partire dalle specifiche (rigorose, o addirittura formali) prodotte dall'analista dati i requisiti iniziali e utilizzate per il progetto e la produzione del software

  Metodi sistematici e tecniche generali, e conseguentemente strumenti di supporto adeguati, tanto più precisi quanto più formale è la specifica da cui si parte

- Convalida e verifica sono attività complementari

- Qualsiasi verifica del software può al più garantire la correttezza rispetto alle sue specifiche, lasciando comunque non risolto il problema di stabilire se le specifiche corrispondano ai requisiti formulati dall'utente (*convalida*)

Terminologia

- Consistente con gli standard proposti dall'*IEEE* (*The Institute of Electrical and Electronics Engineers*)

- Il funzionamento non corretto di un programma è detto ***malfunzionamento*** (*failure*)

- Un malfunzionamento è legato al comportamento di un programma, ma non alla sua struttura statica (il codice)

Esempio:

```
1 program raddoppia (input, output);
2 var x,y: integer;
3 begin
4    read(x);
5    y := x*x;                * anziche' +
6    write(y)
7 end.
```

Con ingresso 3, risultato 9 (malfunzionamento del programma *raddoppia*)

- Un malfunzionamento è causato dalla presenza di ***anomalie*** (*fault*) nel programma

- Un'anomalia coinvolge la *struttura statica* del programma (il codice), senza alcuna relazione con il comportamento del programma

  Nell'esempio, anomalia nella riga 5 dove anziché l'operatore + è usato l'operatore *

- Un programma può contenere una o più anomalie senza per questo presentare malfunzionamenti (ad esempio, il caso in cui l'anomalia è contenuta in un cammino non eseguibile, che non può far sorgere malfunzionamenti di sorta; un altro caso è rappresentato dalla presenza di più anomalie il cui effetto totale è nullo)

- Il manifestarsi di un malfunzionamento non sempre corrisponde ad una ed una sola anomalia

- Il termine ***errore*** (*error*) indica, nello standard proposto dall'*IEEE*, la *causa* di un anomalia

- La fonte di un errore può essere di tipo molto diverso

- Il termine ***baco*** (*bug*), di uso comune, nella terminologia standard *IEEE* è sostituito dal termine anomalia

Correttezza del software

- Metodi di ***prova formale*** si propongono come obiettivo la dimostrazione dell'assenza di anomalie nel prodotto finale

- Metodi piu` pragmatici (tecniche di ***test***) sono volti a rilevare eventuali malfunzionamenti del software

- Altri metodi (tecniche di ***debugging***) cercano di localizzare le anomalie alla base dei malfunzionamenti rilevati

- ***Analisi statica***, se le tecniche disponibili sono basate sull'esame del codice sorgente

- ***Analisi dinamica***, se basate sull'esecuzione del programma

- Alcune tecniche di analisi dinamica prevedono l'esecuzione del programma con dati numerici, altre tecniche più sofisticate permettono invece di generalizzare l'esecuzione "numerica" di un programma, utilizzando simboli come valori delle variabili durante l'esecuzione del programma (*esecuzione simbolica*)

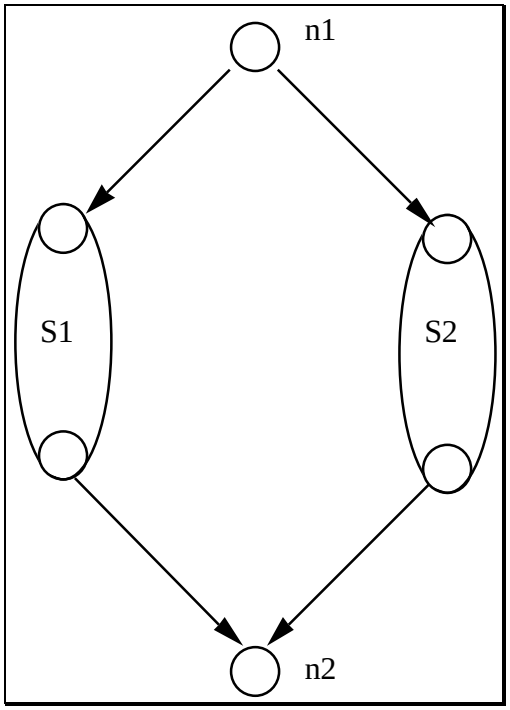
Grafo di controllo di un programma sequenziale

- Molti metodi di convalida e verifica fanno riferimento al grafo di controllo del programma in esame

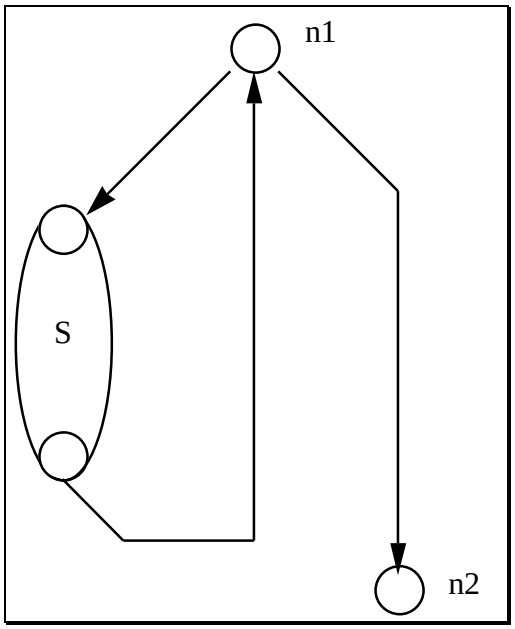
Esempio: mini-Pascal

- Un comando di *assegnamento* o di ingresso/uscita può essere rappresentato da un solo nodo nel grafo di controllo

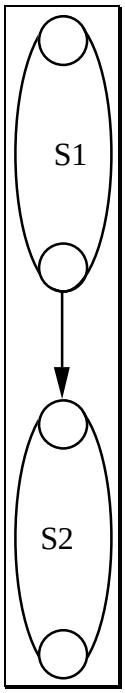
- ***if*** *cond* ***then*** *S1* ***else*** *S2*



- ***while*** *cond* ***do*** *S*:

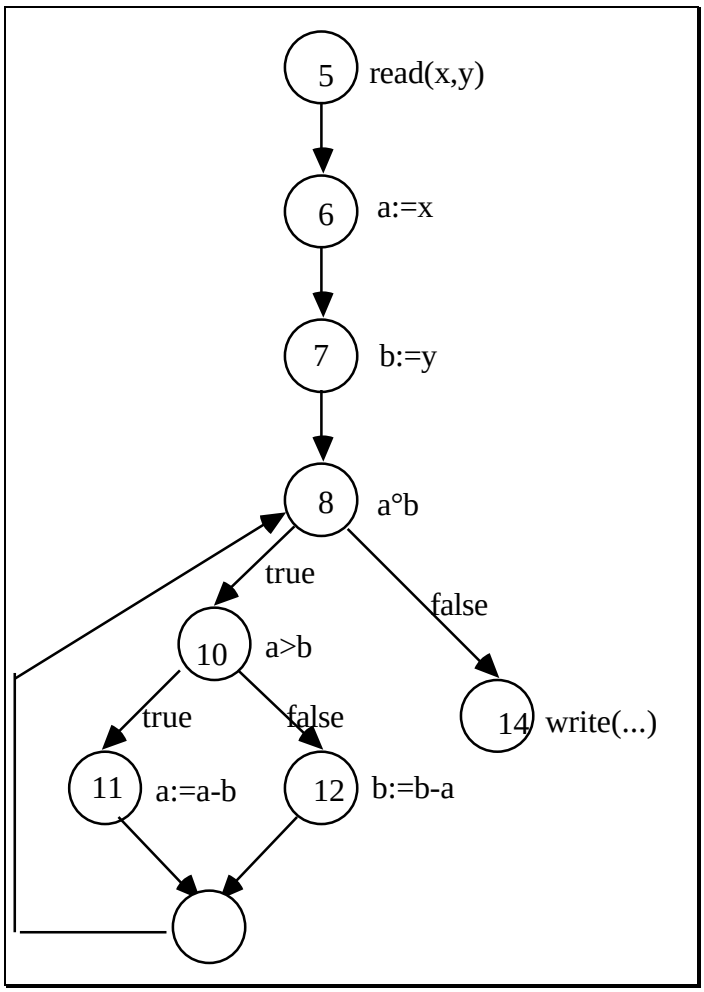


- Composizione sequenziale di due comandi: $S_1; S_2$



Esempio di grafo di un programma:

```
1    program gcd (input, output);
2        var
3            x, y, a, b: integer;
4        begin
5            read (x, y);
6            a := x;
7            b := 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 ('MDC dei dati e''', a)
15       end.
```

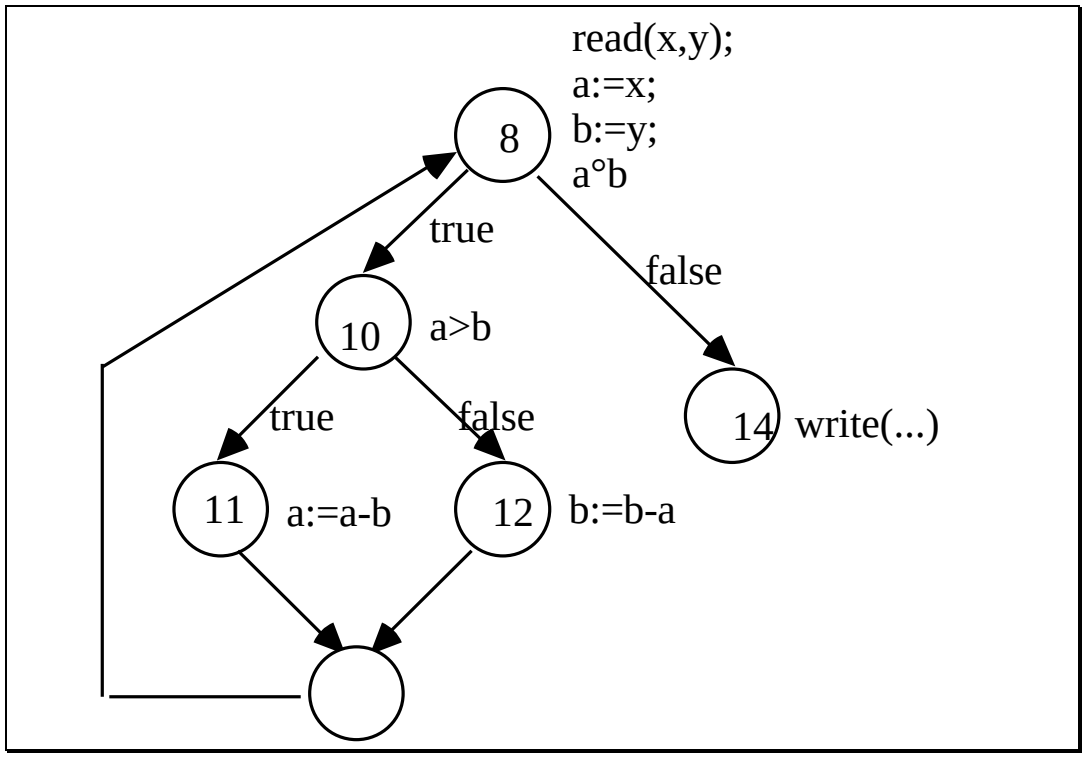


- Il grafo corrispondente ad un programma può essere modificato a seconda dell'uso che si intende fare

- Possono essere sia eliminati che aggiunti nodi ed archi

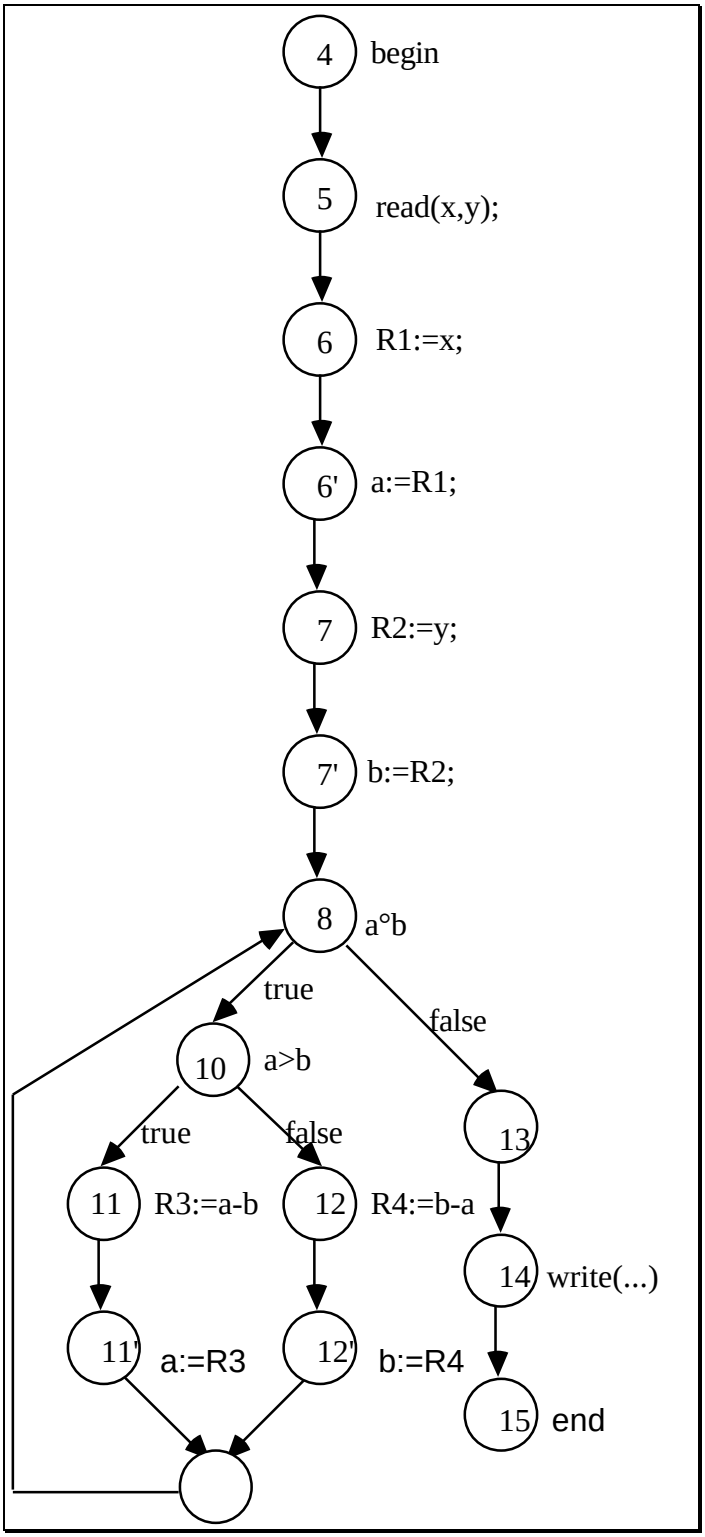
Esempio (1):

- Se si e` interessati solo ai comandi che comportano più alternative nel flusso di controllo i nodi corrispondenti a sequenze di ingressi/uscite ed assegnamenti possono essere collassati:



Esempio (2):

- Rappresentazione delle diverse azioni che corrispondono all'assegnamento (*R1*,...,*R4*, registri):



Analisi statica di un programma

- Obiettivo: determinare la presenza di anomalie in un programma

- Osservazione del codice sorgente (*analisi statica*), non richiedono l'esecuzione del programma

- Esecuzione del programma per particolari dati di ingresso, rilevando eventuali malfunzionamenti e risalendo da questi alle anomalie presenti nel programma (*analisi dinamica*, e fase di *debugging*)

- Per tecniche di analisi statica non esiste il problema di ridurre il numero di casi da esaminare (il codice e` finito), che invece è cruciale per tecniche di analisi dinamica

- La finitezza dei casi da esaminare, che è la caratteristica principale delle tecniche di analisi statica, comporta l'impossibilità di rilevare tutte le anomalie che possono causare malfunzionamenti del programma

- Le tecniche di analisi dinamica permettono quindi di rilevare un maggior numero di malfunzionamenti, e conseguentemente anomalie, rispetto alle tecniche di analisi statica

Analisi statica in compilazione

- La compilazione del codice sorgente richiede che il programma soddisfi particolari caratteristiche di correttezza statica

- Il tipo e l'estensione delle analisi eseguite dipende sia dal linguaggio in cui è scritto il programma sorgente, sia dal compilatore utilizzato

- In generale, il numero di anomalie rilevabili staticamente dipende dalle caratteristiche di dinamicità del linguaggio (ad esempio, regole di visibilità dei nomi)

- Le tecniche fondamentali di analisi statica utilizzate dai compilatori comprendono:
  - *analisi lessicale*
  - *analisi sintattica*
  - *controllo dei tipi (type checking)*

Analisi statica:
analisidi flusso dei dati

- Analisi dell'evoluzione del valore associato alle variabili durante l'esecuzione di un programma, intrinsecamente dinamica

- Alcuni aspetti di questo problema possono però essere analizzati anche staticamente

- Ad ogni comando è possibile associare ***staticamente*** il tipo di operazioni eseguite sulle variabili
  - *definizioni* (*d*),
  - *usi* (*u*)
  - *annullamenti* (*a*),

- Sequenze di comandi, corrispondenti a possibili esecuzioni, sono riducibili staticamente a sequenze di tali operazioni

Esempio:

```
1 procedure swap (x1, x2: real)
2 var x: real;
3 begin
4 x2 := x;
5 x2 := x1;
6 x1 := x;
7 end;
```

- Per la variabile *x*, la sequenza *s* (assegnamenti 4,5,6) può essere ridotta a:
  - un annullamento (il valore associato alla variabile *x* non è infatti definito al momento dell'attivazione del sottoprogramma)
  - un uso (la variabile *x* compare infatti a destra dell'operatore di assegnamento nel comando di linea 4)
  - e un secondo uso (la variabile x compare a destra dell'operatore di assegnamento anche nel comando di linea 6)

- La sequenza di operazioni sulla variabile *x* può quindi essere riassunta con la stringa *auu*

- Per la variabile *x1* la sequenza di operazioni corrispondenti può essere riassunta dalla stringa *dud* (è uno dei parametri formali della procedura e quindi il valore è definito al momento della chiamata)

- Per la variabile *x2*, la sequenza di operazioni corrispondenti e`*ddd*

- L'esame delle sequenze ottenute per ogni variabile può ***rilevare la presenza di anomalie***

- La sequenza *auu*, ad esempio, ottenuta per la variabile *x* permette di dedurre che il valore usato nei due comandi di assegnamento alle linee 4 e 6 non è definito, i due usi della variabile sono infatti preceduti da un annullamento

- In generale, ogni sequenza contenente un ***uso*** non preceduto da una ***definizione*** senza annullamenti intermedi è sintomo di una possibile anomalia dovuta all'uso di valori non definiti

- Nel programma *swap*, che dovrebbe scambiare il contenuto dei parametri *x1* e *x2* facendo uso di una variabile locale *x*, ad esempio, le variabili *x* ed *x2* nell'assegnamento di linea 4 (*x2 := x*;) sono state erroneamente invertite

- Anche la sequenza *ddd* ottenuta per la variabile *x2* permette di rilevare l'anomalia nel programma *swap*: il valore associato a *x2* all'atto della chiamata non è usato prima di essere sostituito da un nuovo valore, è quindi assegnato inutilmente alla variabile

- In generale, ogni sequenza contenente due definizioni consecutive è sintomo di una possibile anomalia

Regole generali:

- La violazione di tali regole permette di dedurre la presenza di possibili anomalie nel programma:

R1 **l'uso di una variabile *x* deve essere sempre essere preceduto in ogni sequenza da una definizione della stessa variabile *x*, senza annullamenti intermedi.**

Un uso non preceduto da una definizione corrisponde infatti al potenziale uso di un valore non determinato di una variabile; al momento dell'uso, infatti, il valore della variabile non è ancora stato definito. Allo stesso modo se tra l'uso e la precedente definizione compare un annullamento, il valore della variabile non è definito all'atto del uso.

R2 **una definizione di una variabile *x* deve sempre essere 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 prima di ulteriori definizioni o annullamenti della variabile corrisponde all'assegnamento di un valore non successivamente utilizzato e quindi potenzialmente inutile. Ciò può essere sintomo di un'anomalia

dovuta all'omissione del comando che avrebbe dovuto usare il valore assegnato alla variabile.

Esempi

aduduu

duadudu

legali secondo le regole (1) e (2)

aduddu

dauduu

duaudu

non soddisfanno invece le regole (1) e (2)

- Nella prima compaiono due definizioni consecutive, contrariamente a quanto richiesto dalla regola (2)
- Nella seconda tra il primo uso e la precedente definizione compare un annullamento, la regola (1) non è quindi soddisfatta
- Nella terza, è interposto un annullamento tra un uso (il secondo uso della sequenza) e la definizione precedente (la prima definizione della sequenza)

- Non tutte le sequenze *au* e *dd* corrispondono necessariamente ad anomalie

- La sequenza *au* può per esempio comparire in un generatore di numeri casuali, che legge il contenuto non inizializzato di una cella di memoria per determinare il seme della generazione

- La sequenza *dd* può essere dovuta ad una cattiva strutturazione del programma, per cui la prima definizione della sequenza non è usata nell'esecuzione considerata, ma lo è in un'altra esecuzione, che richiede la percorrenza di un cammino diverso:

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

- L'analisi di flusso dei dati può essere applicata in tutti i casi in cui l'esecuzione di un programma può essere ridotta a sequenze di azioni in cui l'ordine di esecuzione delle singole azioni è importante

- Azioni su file, ad esempio,: si consideri il caso in cui le operazioni ammesse su files siano *apertura*, *chiusura*, *lettura* e *scrittura*. In questo caso un programma può essere ridotto a sequenze di *aperture* (*a*), *chiusure* (*c*), *letture* (*l*) e *scritture* (*s*), per ogni file usato dal programma, con i seguenti vincoli:

  - una lettura (scrittura) deve essere preceduta da un'apertura, senza chiusure intermedie;

  - un'apertura deve essere seguita da una chiusura, prima di un'apertura successiva;

  - una lettura (scrittura) non può essere seguita da una scrittura (lettura) se non dopo una chiusura e successiva riapertura del file;

  - una chiusura deve essere sempre preceduta da un apertura senza chiusure intermedie.

- Sottosequenze di operazioni interessanti per la rilevazione di possibili anomalie:
  - *cl* e *cs* indicano la presenza di operazioni di lettura (scrittura) precedute da una chiusura;
  - le sottosequenze *ls* e *sl*, indicano la presenza operazioni di lettura e scrittura non separate da chiusura e riapertura del file;
  - le sottosequenze *cc* e *aa* indicano la presenza di chiusure non precedute da apertura e aperture non seguite da chiusure.

- **Nota Bene:** il numero di operazioni diverse considerate ({d, u, a} per le variabili, {a, c, l, s} per le operazioni su file) è sempre finito

- Lo scopo della riduzione di programmi a sequenze di eventi è infatti quello di ridurre un insieme infinito (l'insieme di tutte le possibili esecuzioni di un cammino, ad un insieme finito, e, quindi, analizzabile staticamente

Analisi di flusso (cont.)

- In presenza di decisioni e cicli in numero di cammini e di conseguenza di sequenze di operazioni non è limitabile a priori

- Occorre un linguaggio che permetta la descrizione di cammini alternativi, e cicli (operatori di scelta ed iterazione)

- Un linguaggio adatto a questo scopo è costituito dalle ***espressioni regolari***

- Il concetto di espressione regolare estende il concetto di sequenza aggiungendo all'operatore di composizione sequenziale, gli operatori di alternativa + e di iterazione *

- L'operatore di composizione sequenziale · (spesso omesso) è utilizzato per descrivere azioni sequenziali, l'operatore + è utilizzato per rappresentare azioni alternative e l'operatore * di Kleene per rappresentare cicli di azioni

Espressioni regolari:

- Formalmente le espressioni regolari possono essere definite a partire da un alfabeto finito *A* dalle seguenti regole ricorsive:

- ε, la stringa nulla è un'espressione regolare,

- ogni simbolo dell'alfabeto *A* è un'espressione regolare,

- se e_1 ed e_2 sono espressioni regolari, allora anche $e_1.e_2$, e_1+e_2, e_1^* sono espressioni regolari,

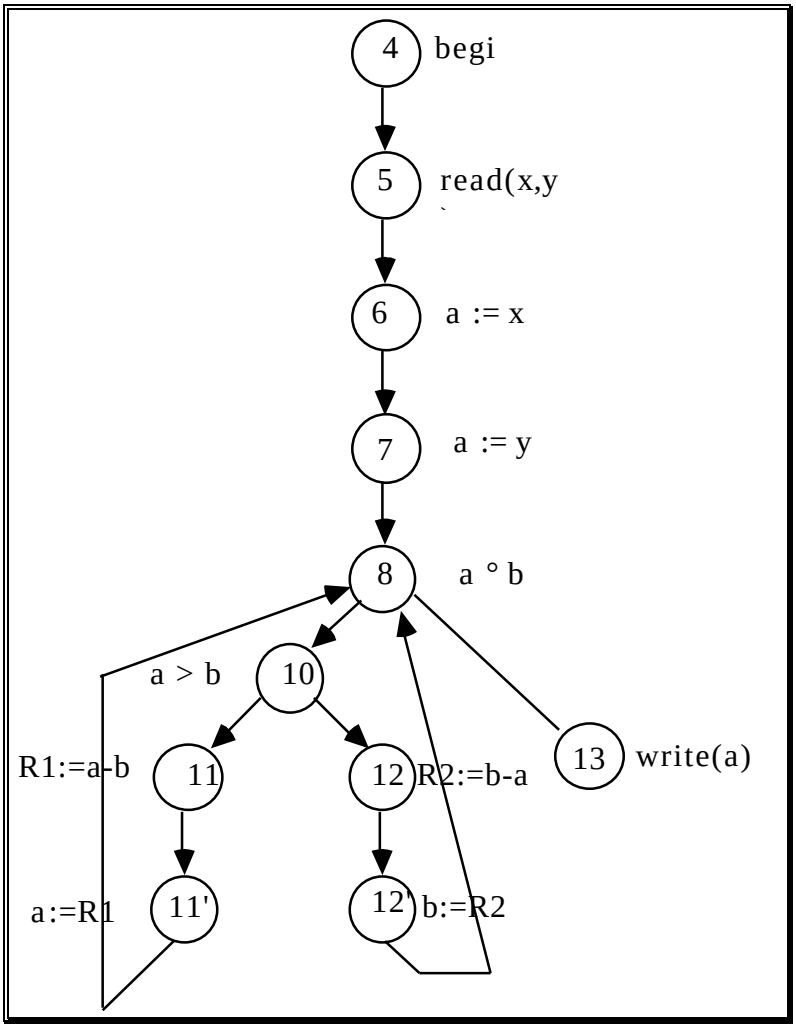
- niente altro è un'espressione regolare.

Espressioni regolari per l'analisi di flusso

```
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;
13     write ('MCD e''', a)
14     end.
```

- Per il programma *gcd* ci sono infiniti cammini, ciascuno corrispondente ad un diverso numero di iterazioni del ciclo *while*

- L'anomalia di linea 7 *(a := y)* al posto di *(b := y)* , può essere rilevata con tecniche di analisi di flusso dei dati, nonostante la presenza di un ciclo



- L'assegnamento di linea 11 (rispettivamente 12) comporta infatti due diverse operazioni (un uso ed una definizione) sulla stessa variabili *a* (rispettivamente *b*)

- A ciascun nodo del grafo di controllo si associano tre insiemi che rappresentano le variabili *definite*, *usate* e *annullate* dall'azione corrispondente nel programma

nodo	variabili definite	variabili usate	variabili 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		
13		a	

- L'espressione ottenuta percorrendo il cammino *p* per la variabile *a* è indicata con *P(p; a)*

- Percorrendo il cammino *[4, 5, 6, 7, 8, 10, 11, 11', 8, 13]* ed annotando le operazioni sulla variabile *a*, ad esempio, si ottiene l'espressione *(adduuuuduu*), che è indicata con *P([4, 5, 6, 7, 8, 10, 11, 11', 8, 13]; a)*

- Nel caso di ***analisi di flusso delle variabili***, l'alfabeto su cui costruire le *espressioni regolari* è dato dai simboli *{d, u, a}*

- Nel caso di analisi di flusso delle operazioni sui files, l'alfabeto è dato dai simboli *{a, c, l, s}*

- In generale per ogni diverso tipo di analisi va definito un alfabeto opportuno, che risulta sempre essere finito, visto che i tipi di operazioni considerabili sono sempre in numero finito

Esempio (cont.)

- Per l'esempio, l'insieme di azioni eseguite sulla variabile *a* corrispondente a tutti i cammini uscenti dal nodo 8, nodo 8 escluso, denotati con [8->], è data dall'espressione regolare, costruita sul linguaggio {*d, u, a*}:

$$P([8->]; a) = (u(ud+u)u)^* u$$

- L'espressione regolare corrispondente alla variabile *a* per tutti i cammini entranti nel nodo 8, nodo 8 escluso, denotati con *[->8]*, è data dall'espressione regolare:

$$P([->8]; a) = add$$

- L'espressione regolare corrispondente alla variabile *a* per l'intero grafo è:

$$P([4->]; a) = addu(u(ud+u)u)^* u$$

- Espressioni rappresentanti gli stessi insiemi di cammini sono tra loro equivalenti

- **Esempio:**

  $(add(u(uud+u))^*uu)$ e` equivalente all'espressione $(addu(u(ud+u)u)^*u)$ corrispondente alla variabile *a* per l'intero grafo

- L'analisi di flusso dei dati può essere eseguita esaminando la ***consistenza*** delle ***espressioni regolari*** costruite ***per ogni variabile***

- L'esame rileva eventuali violazioni delle regole *R1* e *R2* e quindi segnala possibili anomalie.

- **Esempio:**

$$P([4->]; a) = addu(u(ud+u)u)^*u$$

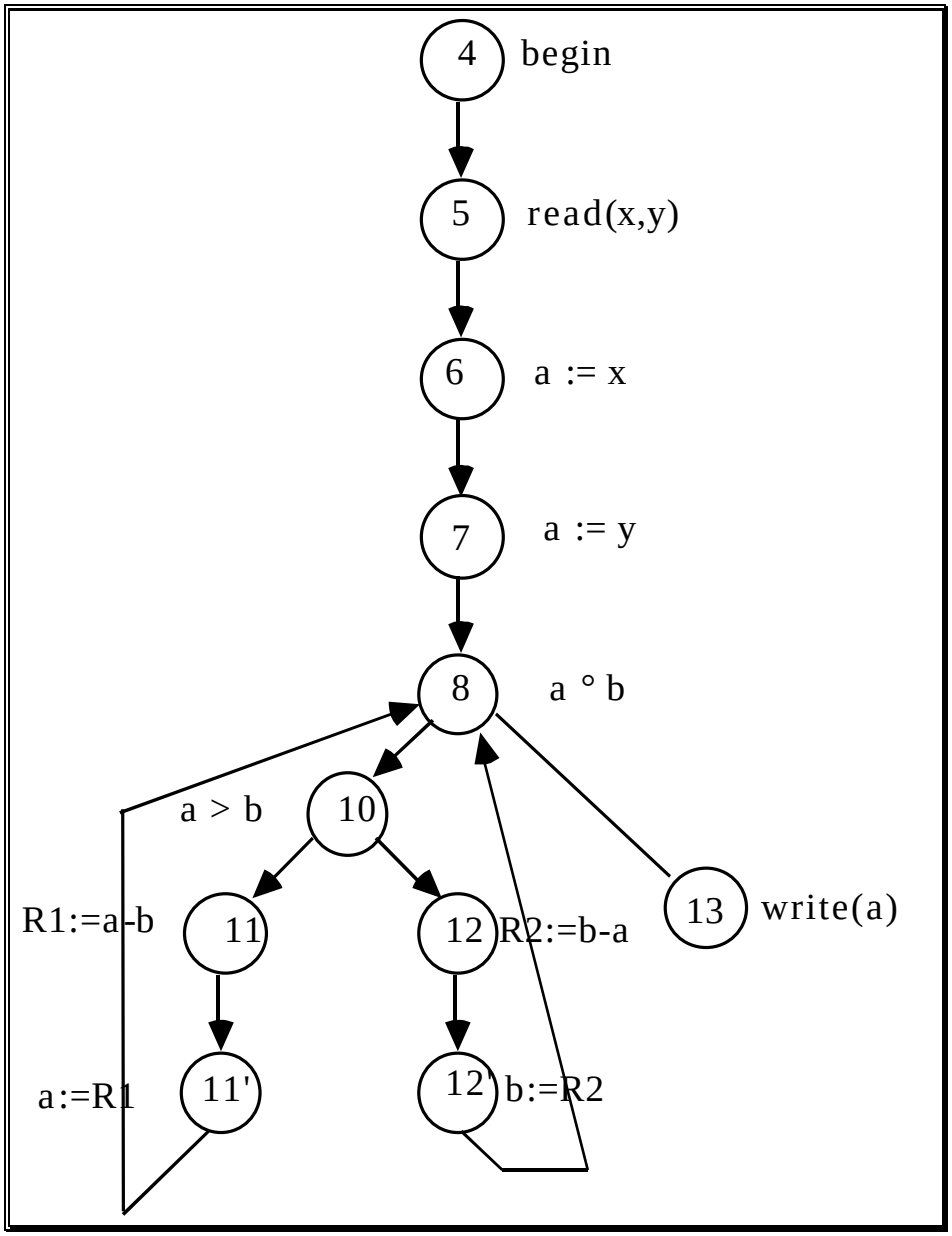
  Esiste una definizione, la prima, mai seguita da un uso

- Analisi molto costosa (percorrenza del grafo per ogni variabile per costruire una espressione regolare)

Algoritmi *live* e *avail*

- Per alcuni problemi si possono usare algoritmi di analisi più efficienti
- Algoritmi *live* e *avail* utilizzati da molti strumenti di analisi di flusso
- Ai nodi del grafo si associano tre insiemi derivabili dagli insiemi di variabili *definite*, *usate* ed *annullate*, una volta stabilito il tipo di anomalia di cui si intende verificare l'esistenza
- Problema di rilevare ***definizioni consecutive*** della stessa variabile senza usi o annullamenti intermedi (uso e annullamento equivalenti)
- Due categorie di operazioni: ***rilevanti*** (le definizioni) e ***di riallineamento*** (riferimenti ed annullamenti)
- A ciascun nodo *n* si associano gli insiemi di variabili:

- quelle su cui è stata effettuata un'operazione rilevante, *gen(n)*
- quelle su cui è effettuata un'operazione di riallineamento, *kill(n)*
- variabili non rilevanti per il nodo in esame, *null(n)*


4
begin
5
read(x,y)
6
a := x
7
a := y
8
a ° b
a > b
10
R1:=a-b
11
12
R2:=b-a
13
write(a)
a:=R1
11'
12'
b:=R2

nodo	gen	kill	null
4		x, y, a, b	
5	x, y		a, b
6	a	x	y, b
7	a	y	x, b
8		a, b	x, y
10		a, b	x, y
11		a,b	x, y
11'	a		b, x, y
12		a, b	x, y
12'	b		a, x, y
13		a	b, x, y

- Per ogni nodo *n*, i tre insiemi *gen(n)*, *kill(n)* e *null(n)* devono essere disgiunti e la loro unione uguale all'insieme delle variabili del programma

- Il problema di rilevare definizioni consecutive della stessa variabile senza usi intermedi può essere risolto confrontando l'insieme *gen* di un nodo con un ulteriore insieme opportunamente calcolato, detto insieme *live*

- Insieme *live* di un nodo n_1 e` l'insieme di tutte le variabili che appartengono all'insieme *gen* di un successore, oppure all'insieme *gen* di un nodo n_2 che sta su un cammino *c* che parte dal nodo n_1, e che non appartengono all'insieme *kill* di nessun nodo nel cammino *c* tra n_1 e n_2,

nodo	live	gen
4	x, y, a	
5	a	x, y
6	a	a
7		a
8		
10		
11	a	
11'		a
12	b	
12'		b
13		

live↔gen

- Una variabile *x* appartiene a *live(n)* se e solo se esiste almeno un cammino uscente dal nodo *n* tale che la prima azione diversa da *null* sulla variabile x sia *gen*

- Insieme *avail* per altri problemi di analisi di flusso

- Una variabile *x* appartiene ad *avail(n)* se e solo se per tutti i cammini che terminano nel nodo *n* l'ultima azione eseguita sulla variabile *x*, a meno di azioni *null* è un'azione *gen*.

- Per ogni nodo *n*, l'insieme *avail* rappresenta le variabili disponibili in quel nodo (definite e non usate o annullate)

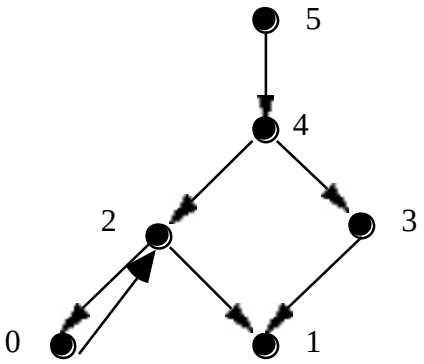
Algoritmo live:

```
1 begin
2     for j := 0 to |N| do live(j) := 0;
3     change := true;
4     while change do
5     begin
6         change := false;
7         for j := 0 to |N| do
8         begin
8         previous := live(j);
10        live(j) := ⋃_{k∈S(j)} (live(k)∩(el-kill(k)))∪gen(k)
11        if previous ? live(j) then
12            change := true;
13        end;
14    end;
15end
```

- L'algoritmo *live* inserisce negli insiemi *live* associati ai nodi dapprima solo gli elementi che appartengono ai *gen* dei successori (S)

- Aggiorna quindi i *live* inserendovi gli elementi appartenenti agli insiemi *live* parziali dei successori così come costruiti al passo precedente, escludendo elementi appartenenti anche al *kill* del successore in questione

Esempio



n	gen	kill	null
0			A,B
1		A,B	
2	A		B
3	B		A
4		A,B	
5			A,B

n \ k	1	2	3	4
0	Φ	A	A	A
1	Φ	Φ	Φ	Φ
2	Φ	Φ	A	A
3	Φ	Φ	Φ	Φ
4	Φ	A,B	A,B	A,B
5	Φ	Φ	Φ	Φ

Algoritmo avail:

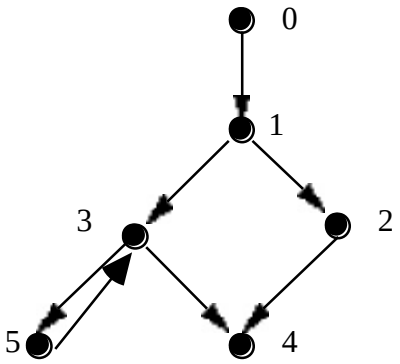
```
1 begin
2     avail(0) := 0;
3     for j := 1 to |N| do avail(j):=el;
4     change := true;
5     while change do
6     begin
7         change := false;
8         for j := 0 to |N| do
9         begin
10        previous := avail(j);
                                  ⋂ (avail(k)⋂(el-kill(k)))⋃gen(k)
                                  k∈P(j)
11        avail(j) :=

12        if previous ? avail(j)
13            then change := true;
14        end;
15     end;
16 end
```

- L'algoritmo *avail* costruisce da subito l'insieme *avail* associato al nodo iniziale: tale insieme è vuoto, essendo il nodo iniziale privo di predecessori (P) per definizione

- Costruisce quindi gli insiemi *avail* togliendo dall'insieme di tutti i possibili elementi, quelli per i quali esiste almeno un cammino entrante nel nodo in questione che termina con *kill*

Esempio



n	gen	kill	null
0	B		A
1		A,B	
2	A,B		
3	A		B
4		A,B	
5			A,B

n \ k	1	2	3
0	Φ	Φ	Φ
1	A,B	B	B
2	A,B	Φ	Φ
3	A,B	Φ	Φ
4	A,B	A	A
5	A,B	A	A

Analisi dinamica

- Le tecniche di analisi dinamica richiedono l'***esecuzione*** del codice

- Presentano problemi di ***infinitezza***: non e` possibile un'analisi esaustiva, come nel caso di tecniche di analisi statica

- Selezione di un ***insieme finito di dati di ingresso***, detti ***dati di test***, con cui esercitare il programma

- Dati di test non selezionati in maniera completamente casuale, ma sfruttando le conoscenze sul problema e/o sul programma

- Criteri di selezione di test basati sul codice (l'implementazione) e le specifiche

- Due classi di ***criteri di selezione di test***, complementari tra loro:

  - criteri di selezione dipendenti solo dalle specifiche o a scatola nera (*black box)* o funzionali

  - criteri di selezione dipendenti solo dal codice o a scatola bianca (*white box*) o strutturali

- Obiettivo, ***selezionare*** insiemi di ***dati di test*** quanto più possibile ***significativi***, in grado cioè di rilevare il maggior numero di malfunzionamenti

- Tre ***necessità*** contrastanti:
  - massimizzare la probabilità di rilevare eventuali malfunzionamenti, richiede un insieme molto grande di dati di test
  - minimizzare i costi dell'attività di test e quindi il numero di dati di test selezionati
  - usare criteri il più possibile generali, non legati all'abilità delle persone incaricate di condurre il test

- Ottenuti i ***risultati***:
  - si confrontano con le specifiche
  - oppure li si sottopone direttamente al giudizio dell'utente (convalida)
  - oppure li si confronta con quelli ottenuti eseguendo versioni precedenti del programma per gli stessi dati di test (per verificare se la manutenzione del programma non ha introdotto malfunzionamenti rispetto alle precedenti versioni, *test di regressione*)

Criteri di terminazione

- Quando considerare conclusa la fase di analisi dinamica?

- Criterio di terminazione temporale (adottato per sistemi in cui malfunzionamenti non causano grossi danni)

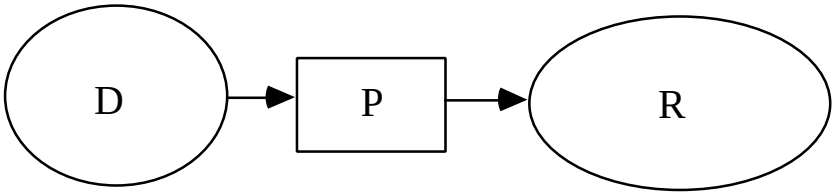
- Non tiene conto nè delle difficoltà che possono via via essere incontrate nella fase di analisi e che possono causare ritardi, nè dello stato del prodotto al termine dell'analisi

- Non fornisce alcuna misura di quanto efficacemente il programma è stato analizzato

- Altri criteri di terminazione per la fase di analisi del software possono essere derivati in maniera quasi immediata da criteri di selezione di dati di test

Certificazione del software: fondamenti teorici

- Scopo dell'attività di test è la rilevazione di malfunzionamenti



- Risultato *P(d)* ottenuto eseguendo il programma *P* sul dato di ingresso *d*, con *d* $\in$ *D*, è corretto se soddisfa le specifiche

- Un programma *P* è corretto, indicato come *ok(P)*, se è corretto per ogni elemento *d* del dominio di ingresso *D*:

ok(P) se e solo se $\forall$ *d* $\in$ *D, ok(P, d)*

- Un ***test*** *T* per un programma *P* con dominio *D* è un sottoinsieme del dominio *D*

- Un elemento *t* di un test *T* è detto ***dato di test***

- Esecuzione di un test *T* consiste nell'esecuzione del programma *P* per tutti i dati di test *t* contenuti nel test *T*

- Un programma *P* è detto ***corretto per un test*** *T*, *ok(P,T)*, se il programma *P* è corretto per tutti gli elementi di *T*: *" tÎ T ok(P,t)*

- Un test *T* ha ***successo*** per un programma *P*, indicato con *successo(T,P)*, se rileva uno o più malfunzionamenti presenti nel programma *P*

- Da un test che non rileva alcun malfunzionamento non può essere dedotta la correttezza del programma, ma piuttosto l'***inadeguatezza del test***

- Un test *T* per un programma *P* è detto ***ideale*** se l'insuccesso del test *T* implica la correttezza del programma *P*

- Uno degli scopi della ***teoria del test*** è la definizione di ***metodi*** per la ***selezione*** di ***test ideali***

Criteri di selezione dei test

- ***Criterio di selezione di test*** consente di selezionare uno o più test per lo stesso programma

Esempio:

Programma *P* con dominio di ingresso l'insieme dei numeri interi

Criterio di selezione di test *C*: almeno un numero negativo, lo zero, almeno un numero postivo

{-4, 0, 5}, {-2, -9, 0, 8, -12, 6} sono test selezionati dal criterio C, che seleziona infiniti test

- Un criterio di selezione di test *C* è ***affidabile*** per un programma *P*, *affidabile(C, P)*, se *" coppia di test T_1 e T_2 selezionati dal criterio C, se il test T_1 ha successo, allora anche il test T_2 ha successo e viceversa*

- Affinché un criterio di selezione di test sia affidabile è necessario che tutti i test selezionati dal criterio falliscano (o rispettivamente abbiano successo), ma non che tutti consentano di rilevare lo stesso malfunzionamento

- Un criterio di selezione di dati di test C è ***valido*** per un programma P, *valido(C, P)*, se, qualora il programma P non sia corretto, esiste almeno un test T selezionato da C che ha successo per il programma P

- La validità di un criterio C garantisce quindi che se esiste un errore nel programma P, tale errore può essere rilevato da ***almeno uno dei test*** T_i che soddisfano il criterio C

- Non tutti i test selezionati da un criterio valido sono in grado di rilevare i malfunzionamenti presenti nel programma

Esempio:

```
1 program raddoppia (input, output);
2 var x,y: integer;
3 begin
4    read(x);
5    y := x*x;
6    write(y)
7 end.
```

- Un criterio C_1 che seleziona solo sottoinsiemi di *{0, 2}*è affidabile, ma non valido

- Un criterio C_2 che seleziona sottoinsiemi di *{0, 1, 2, 3, 4}* è valido, ma non è affidabile: i test *{0, 4}* e *{0, 2}* danno risultati opposti

- Un criterio C_3 che seleziona insiemi che contengono almeno un valore maggiore od uguale a *3*, è sia valido sia affidabile

- L'aver dimostrato che un certo criterio di selezione di test soddisfa particolari proprietà di affidabilità e validità per il programma inizialmente in esame, non garantisce nulla sulle proprietà godute dallo stesso criterio di selezione di test una volta che il programma in esame viene modificato per rimuovere le anomalie riscontrate durante la fase di test

```
1 program raddoppia (input, output);
2 var x,y: integer;
3 begin
4    read(x);
5    y := x+5;
6    write(y)
7 end.
```

Sia C_1 che C_2 diventano test ideali

C_3 diventa valido (seleziona almeno un test che contiene un numero diverso da 5 e quindi rileva il malfunzionamento), ma non più affidabile (seleziona il test *{5}* che non ha successo per il programma, non rileva cioè malfunzionamenti)

Risultati teorici

- *Teorema di Goodenough e Gerhart*: il fallimento di un test *T* per un programma *P* selezionato da un criterio *C* affidabile e valido per il programma *P*, permette di dedurre la correttezza del programma *P*

(affidabile(C, P) Ù valido(C, P) Ù

selezionato(C, T) Ù∃ successo(T, P)) Þ ok(P)

- Il problema e` stabilire le proprietà di affidabilità e validità di un criterio di selezione di test

- Il criterio di test *C* che richiede che il test selezionato contenga tutto il dominio di ingresso è valido e affidabile; si parla in tal caso di ***test esaustivo***

- Il test esaustivo non è però praticabile nella maggior parte dei casi significativi, vista la dimensione del dominio di ingresso, spesso infinito

- Non è possibile, in generale, costruire un criterio di selezione di test valido ed affidabile, che non sia il test esaustivo (*teorema di Howden*)

> *Non esiste una **procedura effettiva** H che, dato un programma arbitrario P con dominio D, generi un test ideale finito; non esiste cioè un criterio di selezione di test C affidabile e valido (effettivo) soddisfatto da un test finito per il programma P*

- Il teorema corrisponde alla *tesi di Dijkstra*:

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

Solo con *prove formali di correttezza*.

Indecidibilità e test

Teorema di Weyuker:

- Dato un generico programma P i seguenti problemi risultano indecidibili:

  a) esiste almeno un dato di ingresso che causa l'esecuzione di un particolare cammino?

  b) esiste un particolare dato di ingresso che causa l'esecuzione di una particolare condizione (branch)?

  c) esiste un dato di ingresso che causa l'esecuzione di un particolare cammino?

  d) è possibile trovare almeno un dato di ingresso che causi l'esecuzione di ogni comando di P?

  e) è possibile trovare almeno un dato di ingresso che causi l'esecuzione di ogni condizione (branch) di P?

  f) è possibile trovare almeno un dato di ingresso che causi l'esecuzione di ogni cammino di P?

- Il teorema di Weyuker è estremamente importante per i ***metodi di test strutturale***, attualmente tra i più applicati

Test strutturale (o a scatola bianca)

- Si adottano criteri di selezione dipendenti solo dal codice (detti anche criteri a scatola bianca (*white box*) o strutturali

• Criteri di selezione generali

- *di copertura dei comandi*
- *di copertura delle decisioni*
- *di copertura delle condizioni*
- *di copertura delle decisioni e delle condizioni*
- *di copertura dei cammini*
- *di n-copertura dei cicli*

• Criteri di selezione Data Flow

- *di copertura delle definizioni*
- *di copertura di tutti gli usi*
- *di copertura dei cammini du*

Criteri di selezione generali

- **Criterio di copertura dei comandi** (*statement test)*: un test *T* soddisfa il criterio di copertura dei comandi se e solo se ogni comando eseguibile del programma è eseguito in corrispondenza di almeno un dato di test in *T*

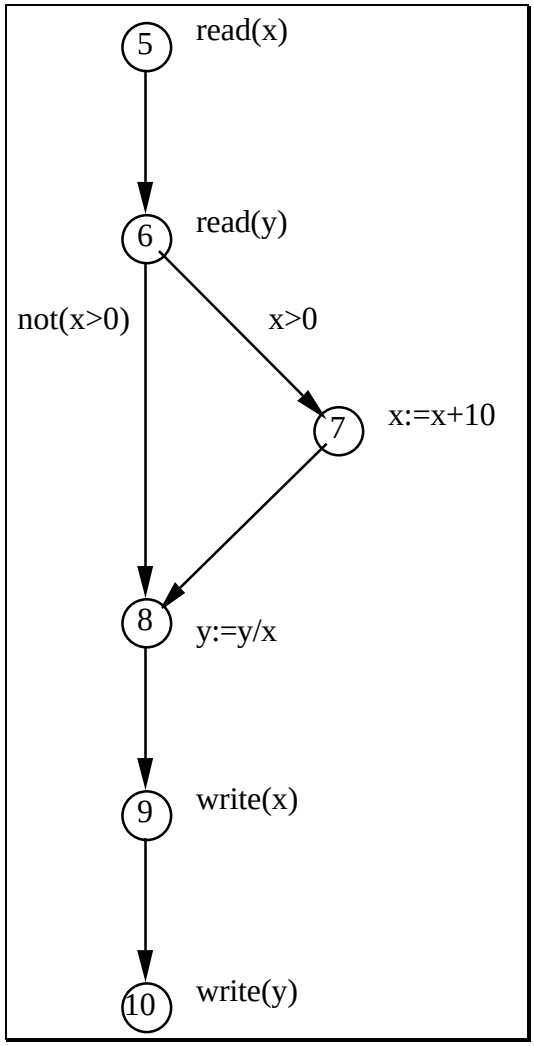
- Nel grafo di controllo del programma, ***ogni nodo*** corrispondente ad un comando eseguibile deve essere ***percorso almeno una volta***

- Misura di copertura:

$$C_0 = \frac{\text{numero di comandi eseguiti}}{\text{numero totale di comandi eseguibili}}$$

Esempio

```
1  Program statement (input, output);
2      var
3          x,y : real;
4      begin
5          read(x);
6          read(y);
7          if x > 0 then x:=x+10;
8          y:=y/x;
9          write(x);
10         write(y);
11     end.
```



- Un test *S* che contiene un dato di test con *x* ? 0 e *y* qualsiasi, ad esempio il test S = {(x = 20, y = 30)}, causa l'esecuzione di tutti i comandi del programma e quindi soddisfa il criterio di copertura dei comandi

- La copertura C_0 assicurata è uguale a 1

- Il malfunzionamento (divisione per zero), che si verifica quando il comando alla linea 8 viene eseguito con valore zero per la variabile *x* non è rilevato dal test *S*

- Il test *S* non comporta l'esecuzione del programma per tutti i valori delle decisioni: la decisione alla linea 7 *(x>0)* dà valore *vero* per tutti i dati di test contenuti in *S*

Criteri di selezione generali (cont.)

- **Copertura delle decisioni** (*branch test*): un test *T* soddisfa il criterio di copertura delle decisioni se e solo se ***ogni arco*** del grafo di controllo del programma è ***percorso almeno una volta***

- Misura di copertura:

$$C_1 = \frac{\text{numero di archi percorsi}}{\text{numero totale di archi percorribili}}$$

- Il criterio di copertura delle decisioni contiene propriamente il criterio di copertura dei comandi

Esempio (cont.)

- Il test S non soddisfa il criterio di copertura delle decisioni: l'arco *(6, 8)* non viene mai percorso

- Il test $D = \{(x = 20, y = 30), (x = 0, y = 30)\}$ soddisfa il criterio di copertura delle decisioni, in quanto causa la percorrenza di tutti gli archi almeno una volta, e permette di rilevare il malfunzionamento causato dalla divisione per zero alla linea 8

Criteri di selezione generali (cont.)

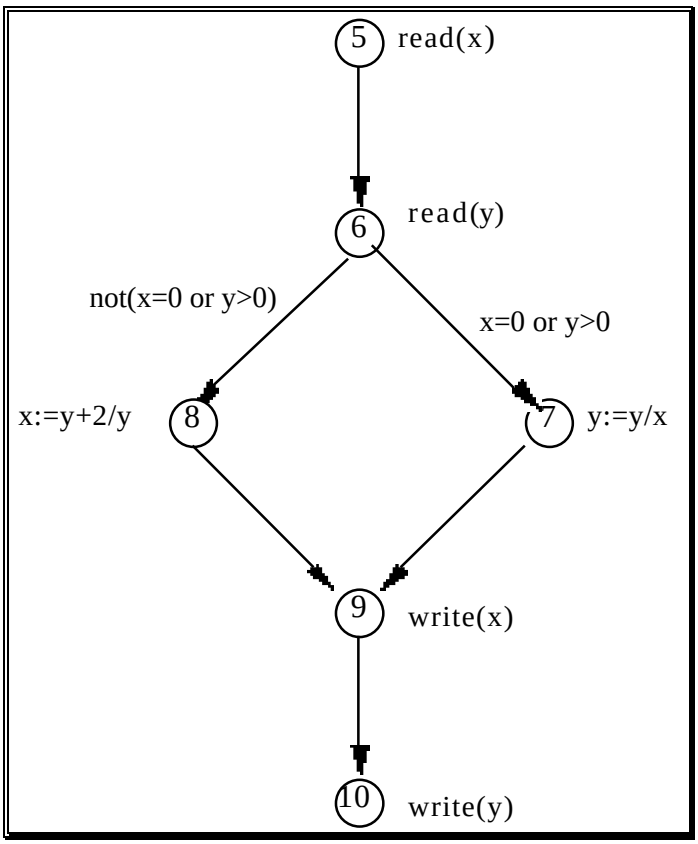
- **Copertura delle condizioni** (*condition testing*): un test T soddisfa il criterio di copertura delle condizioni se e solo se ogni singola condizione che compare nelle decisioni del programma vale sia vero che falso per diversi dati di test in T

- **Criterio di copertura delle decisioni e delle condizioni**: un test T soddisfa il criterio di copertura delle decisioni e delle condizioni se e solo se ogni decisione vale sia vero che falso ed ogni singola condizione che compare nelle decisioni del programma vale sia vero che falso per diversi dati di test in T

- Il criterio di copertura delle decisioni e delle condizioni contiene sia il criterio di copertura delle condizione che il criterio di copertura delle decisioni

Esempio

```
1  Program branch (input, output);
2  var
3      x,y:real;
4  begin
5      read(x);
6      read(y);
7      if (x = 0 or y > 0) then  y:=y/x
8      else   x:=y+2/y;
9      write(x);
10     write(y);
11 end.
```



- Il test $B = \{(x = 5, y = 5), (x = 5, y = -5)\}$ soddisfa il criterio di copertura delle decisioni, ma non rileva un malfunzionamento causato da una divisione per zero sia alla linea 7 che alla linea 8

- La decisione alla linea 7 può essere posta a *vero* ed a *falso* agendo unicamente su una sola delle due condizioni in *or* nella decisione (non soddisfa il *copertura delle condizioni*)

- Il test $C = \{(x = 0, y = -5), (x = 5, y = 5)\}$ soddisfa il criterio di copertura delle condizioni, infatti per il dato $(x = 0, y = -5)$ la condizione $(x = 0)$ vale *vero* e la condizione $(y > 0)$ vale *falso*, per il dato $(x = 5, y = 5)$ la condizione $(x = 0)$ vale *falso* e la condizione $(y > 0)$ vale *vero*

- Il test *C* rileva il malfunzionamento causato dall'anomalia alla linea 7, ma non quello causato dall'anomalia alla linea 8 (non soddisfa il criterio di copertura delle decisioni, infatti per entrambi i dati del test *C* la decisione di linea 7 vale *vero*)

Criteri di selezione generali (cont.)

- I criteri esaminati non considerano esplicitamente i cicli

- **Criterio di copertura dei cammini** (*path test*): un test T soddisfa il criterio di copertura dei cammini se e solo se ogni cammino nel programma viene percorso per almeno un dato di test in T

- Misura di copertura:

$$C_N = \frac{\text{numero di cammini percorsi}}{\text{numero totale di cammini percorribili}}$$

- Il numero di cammini eseguibili può essere infinito, rendendo tale criterio (il migliore) non applicabile

- Per limitare il numero di cammini si fissa il numero massimo di percorrenze di ciascun ciclo

- **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 di iterazioni di ogni ciclo non superiore ad n è eseguito per almeno un dato di test in T

- Determinazione automatica del numero appropriato di iterazioni per ciascun ciclo a partire da considerazioni sulla relazione tra i vari comandi che compongono il programma

Esempio

```
1    program gcd (input, output);
2        var
3            x, y, a, b: integer;
4        begin
5            read (x, y);
6            a := x;
7            b := 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 ('MDC dei dati e''', a)
15       end.
```

- Il ciclo *while* comporta definizioni ed usi delle variabili *a* e *b* (almeno 2-copertura dei cicli)

Esempio

```
a : array [1..k] of items;
el : item;
.....
i := 1;
while a[i] <> el do
.....
```

- Test che soddisfa il criterio di k-copertura dei cicli (dove *k* rappresenta il limite superiore del campo di validità degli indici dell'array *a*)

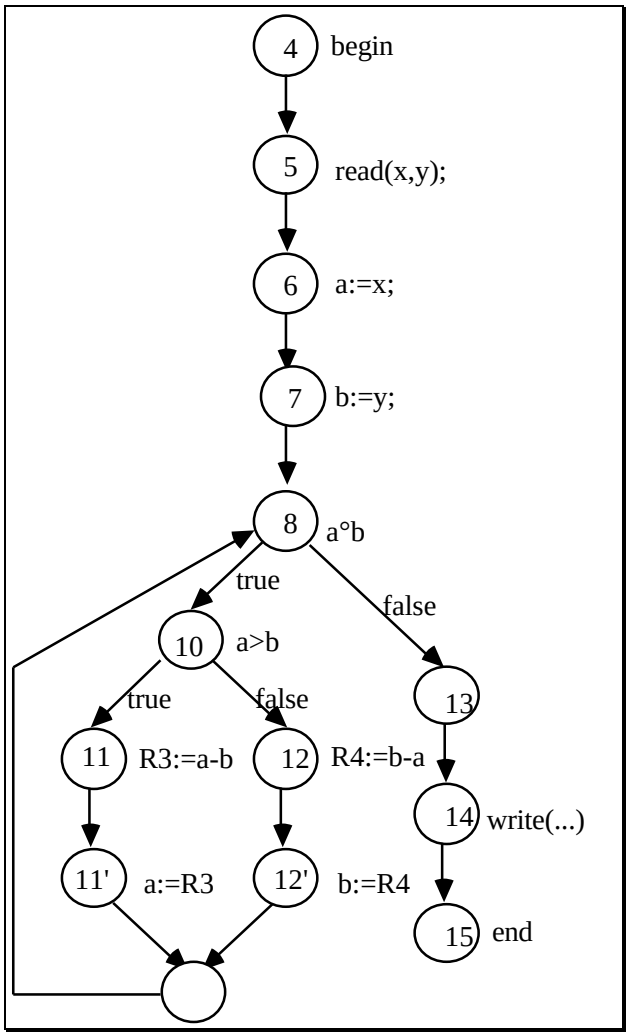
Criteri di selezione Data Flow

- Determinano la significatività dei cammini in un programma a partire dall'analisi di flusso delle variabili

- Selezione di un cammino basata sul numero di sequenze definizione/uso delle variabili

- Un ciclo viene ripetuto un numero maggiore di volte solo se ciò permette di eseguire sequenze di definizione/uso non già esaminate

Esempio: *gcd*



- variabili*x* ed *y* definite (linea 5) e usate (linee 6 e 7), sufficienti test selezionati da criteri di 0-copertura dei cammini

- variabili *a* e *b* : definizione di *a* di linea 6 usata nelle linee 8, 10, 11, 12 e 14; definizione di *a* di linea 11 (nodo 11') usata in 8, 10, 11, 12; definizione di *b* di linea 7 usata nelle linee 8, 10, 11 e 12; definizione di *b* di linea 12 (nodo 12') usata nelle linee 8, 10, 11, 12.

- ***Obiettivo***: determinare il numero minimo di iterazioni richieste affinché ***tutte le sequenze definizione/uso*** per le variabili *a* e *b* siano esercitate ***almeno una volta***

- Dati di test che non causano ***alcuna iterazione del ciclo*** *while* (ad esempio il dato *(x = 5, y = 5))* non sono sufficienti

- Dati di test che causano ***un'iterazione del ciclo*** *while* (ad esempio il dato (x = 10, y = 5)) permettono di esercitare tutti gli usi delle variabili *a* e *b* che possono seguire le definizioni alle linee 6 e 7 rispettivamente, ma non gli usi dei valori definiti dai comandi alle linee 11 (nodo 11') e 12 (nodo 12')

- Dati di test che causano ***due iterazioni del ciclo*** *while* permettono invece di testare tutte le sequenze definizione/uso delle variabili *a* e *b*, e quindi permettono un'analisi completa per il criterio scelto (ad esempio il dato *(x =15, y = 5))*

- Dati di test che richiedono ***piu` di due di iterazioni dei ciclo*** *while* non permettono di esercitare sequenze di definizione/uso non già esaminate (ad esempio il dato *(x =20, y = 5))*

Criteri di selezione DF (cont.)

- I criteri di selezione di test *DF* considerano solo usi e definizioni delle variabili, e non annullamenti

- Le anomalie dovute a usi di variabili annullate possono essere rilevate e rimosse staticamente

- A ogni nodo del grafo di controllo del programma in esame sono associati:
  - l'insieme delle ***variabili definite*** dall'esecuzione del comando corrispon-dente al nodo e
  - l'insieme delle ***variabili usate*** dallo stesso comando

- Costruzione del grafo in modo che *def* e *use* associati a ciascun nodo siano disgiunti (introdurre registri "fittizi" se necessario)

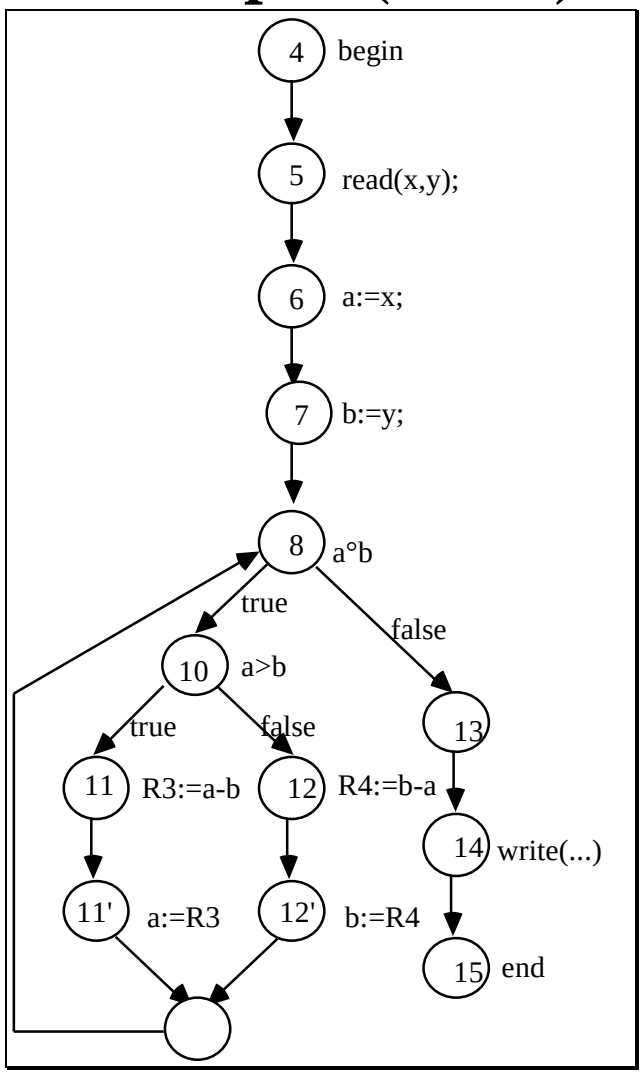
Definizioni

- Sia *x* una variabile del programma *P*

- Il cammino *(i, n_1, .., n_m, j)* nel grafo corrispondente al programma P è un ***cammino libero da definizioni rispetto alla variabile x dal nodo i al nodo j*** se e solo se la variabile *x* non appartiene a nessuno degli insiemi *def* associati ai nodi *i*, n_1, .., n_m, *j*

- Per ogni nodo *i* e ogni variabile *x* appartenente all'insieme *def* associato al nodo *i*, si definisce ***l'insieme du(x, i)*** come l'insieme di tutti i nodi *j* tali che esiste un cammino libero da definizioni rispetto alla variabile *x* dal nodo *i* al nodo *j* e *x* è usata nel nodo *j*

Esempio (cont.)



nodo	def	use	du(x)	du(y)	du(a)	du(b)
5	x, y		6	7		
6	a	x			8,10,11,12,14	
7	b	y				8,10,11,12
8		a, b				
10		a, b				
11		a, b				
11'	a				8,10,11,12,14	
12		a, b				
12'	b					8,10,11,12
14		a				

Criteri di selezione DF

- I criteri di selezione di test vengono definiti a partire dal grafo associato al programma e dagli insiemi *def*, *use* e *du* corrispondenti

- **Criterio copertura delle definizioni**: un test *T* soddisfa il criterio di copertura delle definizioni se e solo se per ogni nodo *i* ed ogni variabile *x* appartenente all'insieme *def(i), T* include un cammino libero-da-definizioni dal nodo *i* ***ad almeno un elemento*** di *du(i, x)*

- **Esempio**

  Per il programma *gcd* il criterio seleziona test che causano al più un'iterazione del ciclo *while*

- Il criterio ***non richiede*** che per ogni definizione siano raggiunti tutti gli usi che possono fare riferimento alla stessa definizione

  Per la definizione di *a* corrispondente al nodo 11', il criterio è soddisfatto (uso di *a* al nodo 8)

- Un criterio più completo considera ***tutti gli elementi*** dell'insieme *du*

- **Criterio di copertura di tutti gli usi**: un test *T* soddisfa il criterio di copertura tutti gli usi se e solo se per ogni nodo *i* ed ogni variabile *x* appartenente all'insieme *def(i), T* include un cammino libero-da-definizioni dal nodo *i* ***ad ogni elemento*** di *du(i, x)* .

- **Esempio**

  Per il programma *gcd* il criterio seleziona test adeguati

- **Criterio copertura dei cammini du**: un test *T* soddisfa il criterio di copertura dei cammini *du* se e solo se per ogni nodo *i* ed ogni variabile *x* appartenente all'insieme *def(i), T* include ***tutti i cammini*** non contenenti cicli liberi-da-definizioni dal nodo *i* ***ad ogni elemento*** di *du(i, x)* .

Analisi mutazionale

- E` un ulteriore metodo di ***analisi dinamica***
- Usato sia per selezionare test che per ***misurare la qualità*** di dati di ***test selezionati*** in precedenza
- Consiste nel generare un insieme di programmi P simili al programma P da analizzare, e provare lo stesso test T per il programma P e per tutti i programmi in P
- I programmi in P sono chiamati ***mutanti***
- Un test T è selezionato richiedendo che ciascun programma in P sia distinto dal programma P per il test T
- La ***qualità di un test*** T è data dal numero di programmi in P che si riescono a distinguere dal programma P in base al test T
- La ***misura*** di qualità di un test dipende dall'insieme di mutanti generati

Analisi mutazionale (cont.)

- Scelta di un ***insieme adeguato di mutanti***, in numero limitato, ma in grado di fornire una valutazione abbastanza accurata del test

- Dipende dal tipo di applicazione, dal programma, dal tipo di errori più probabili, dal linguaggio scelto

- E` un'analisi facilmente automatizzabile

Test funzionale

- I criteri di test funzionali permettono di derivare dati di test a partire dalle specifiche del programma, a prescindere dal codice

- Approccio *black-box*

- Permette di derivare dati di test non sintetizzabili con criteri strutturali

- La definizione di criteri di selezione di test funzionali generali dipende dal linguaggio usato per le specifiche

  - *Se espresse in modo informale, i criteri di test richiedono l'intervento dell'utente*

  - *Se espresse in termini formali, i criteri di test possono essere definiti rigorosamente e l'intervento umano può essere limitato ai casi non decidibili*

Test funzionale basato su specifiche informali

- La selezione consiste principalmente nel decomporre una specifica in un insieme di casi rilevanti e produrre per ciascun caso uno o più dati di test

- Identificazione di un ***insieme di condizioni***, e quindi, per ciascuna condizione, la produzione di uno o più ***casi***

- Per *specifiche informali*, dipende dall'esperienza e l'abilità delle persone a cui è affidata la fase di test

- Per *specifiche semiformali*, ad esempio diagrammi di flusso dei dati oppure automi a stati finiti, possono essere definite regole generali di decomposizione

- Il processo di selezione è completato dalla selezione di uno o più dati per ogni caso identificato

- Permette di derivare casi di test in grado di rilevare la presenza di ***anomalie*** dovute alla ***mancata realizzazione di un cammino di controllo*** nel programma

- Malfunzionamenti per dati particolari che non corrispondono ad un solo caso, ma rappresentano la frontiera tra più casi diversi

- Si usa in questi casi il ***criterio di selezione di test di frontiera*** (*boundary value analysis*)

- Seleziona un dato di test per ogni sottoinsieme di casi contigui, per cui cioè esiste almeno un elemento di frontiera, e non per casi singoli

- Altri dati critici sono relativi alla combinazione di più casi

- Metodo basato sulla costruzione del ***grafo causa-effetto***

- Le specifiche iniziali sono ridotte a condizioni booleane, rappresentanti fatti di ingresso o uscita

- Tra le condizioni booleane identificate è possibile definire relazioni di causa-effetto (grafo, archi *and, or, not*)

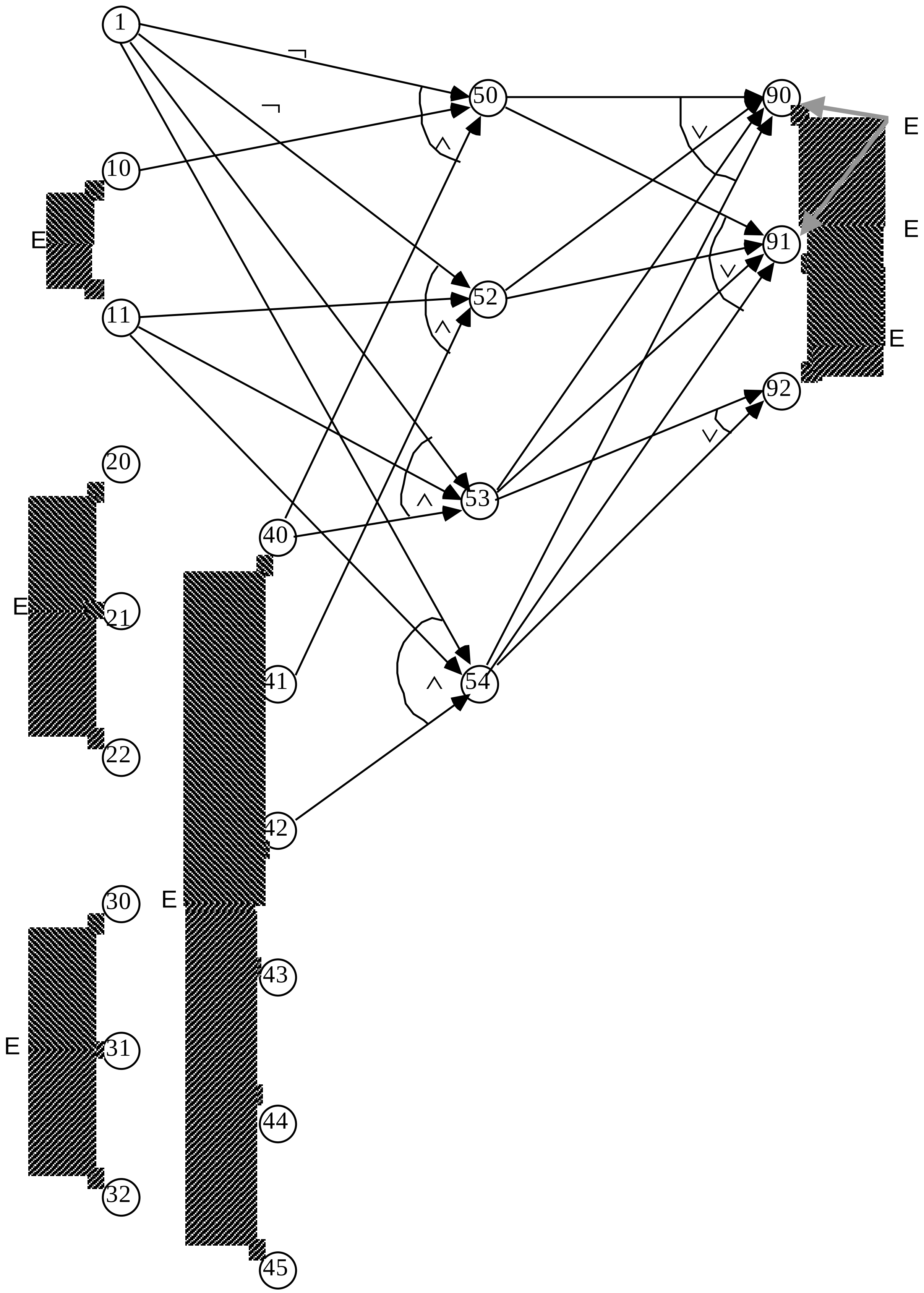
- Per derivare casi di test a partire dal grafo causa-effetto viene generata una ***tabella di decisione*** con una riga per ogni fatto ed una colonna per ogni classe di dati di test identificati

- Ogni colonna della tabella viene costruita a partire da un effetto finale, percorrendo all'indietro il grafo per determinare per quali combinazioni di cause si verifica l'effetto selezionato

Esempio

1	cliente di riguardo
10	permanenza inferiore a 15 giorni
11	permanenza superiore a 15 giorni
20	stanza singola
21	stanza doppia
22	stanza matrimoniale
30	alta stagione
31	media stagione
32	bassa stagione
40	periodo di permanenza con conto totale inferiore a 1.500.000 lire
41	periodo di permanenza con conto totale superiore a 1.500.000 lire
42	periodo di permanenza con conto totale inferiore a 1.660.000 lire
43	periodo di permanenza con conto totale superiore a 1.660.000 lire
44	periodo di permanenza con conto totale inferiore a 2.140.000 lire
45	periodo di permanenza con conto totale superiore a 2.140.000 lire
90	pagamento in contanti
91	pagamento con carta di credito
92	pagamento con assegni

- Ad un cliente non di riguardo (nodo 1 con arco ÿ) che si è fermato meno di 15 giorni (nodo 10) è richiesto il pagamento della tariffa piena (nodo 50), che deve essere effettuato in contanti o carta di credito (nodi 90 e 91)



(E, mutuamente esclusive)

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	1	1	1	1	1	1
10	1	0	1	0	0	0	0	0	0	0
11	0	1	0	1	1	1	1	1	1	1
20										
21										
22										
30										
31										
32										
40	1	0	1	0	1	0	1	0	1	0
41	0	1	0	1	0	1	0	1	0	0
42	0	0	0	0	0	0	0	0	0	1
43	0	0	0	0	0	0	0	0	0	0
44	0	0	0	0	0	0	0	0	0	0
45	0	0	0	0	0	0	0	0	0	0
90	1	1	0	0	1	1	0	0	0	0
91	0	0	1	1	0	0	1	1	0	0
92	0	0	0	0	0	0	0	0	1	1

- Fatti di ingresso (1 - 45), fatti di uscita (90, 91, 92)
- Dieci cammini identificati sul grafo parziale a partire dai nodi che rappresentano condizioni di uscita
- In tabella, 0 se per il caso (colonna) la condizione booleana (riga) è falsa, 1 se è vera, nessun valore se indifferente
- Per ogni effetto finale esistono numerose combinazioni di cause iniziali, non tutte ugualmente significative
- Eliminazione di casi ridondanti

Selezione di test: esecuzione simbolica

- I criteri di selezione di test introdotti permettono di determinare quali cammini eseguire per ottenere la copertura desiderata del programma

- Non forniscono però un metodo per determinare effettivamente dati di test che causano l'esecuzione dei cammini selezionati (*problema indecidibile*, teorema di Weyuker)

- Si utilizza la tecnica di ***esecuzione simbolica***

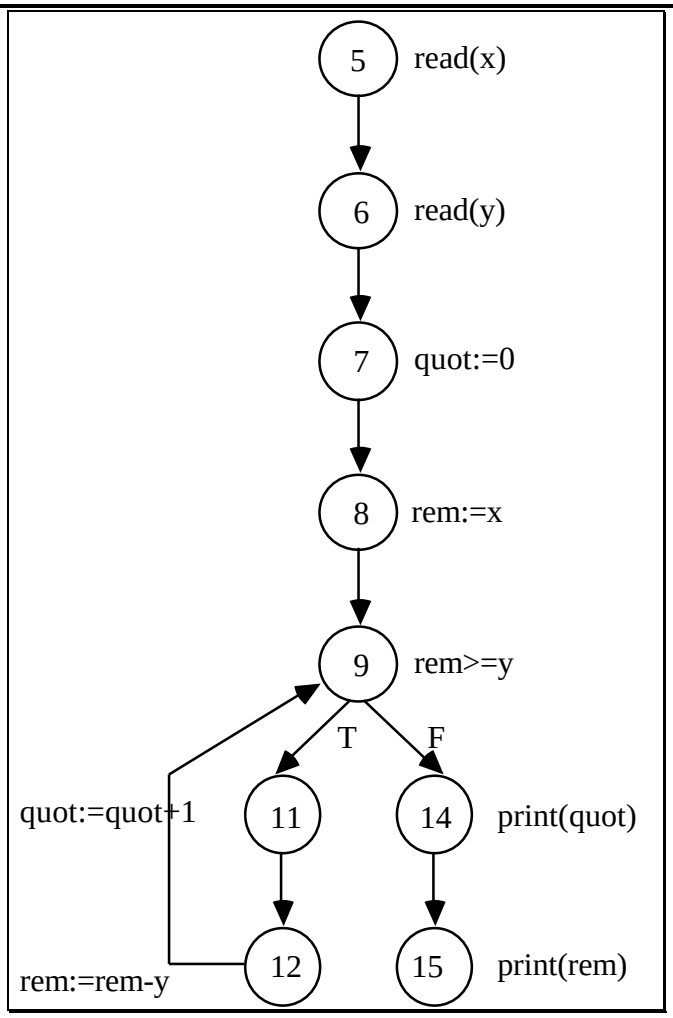
- Permette di determinare:
  - le *condizioni* che devono essere verificate dai dati in ingresso affinché un particolare *cammino* venga *eseguito*
  - la relazione tra i valori prodotti dall'esecuzione del programma ed i valori in ingresso al programma stesso

- La sintesi di dati di test a partire dalla condizione di percorrenza di un cammino, o, eventualmente, la conclusione che un cammino non è eseguibile, richiede l'intervento umano (per l'indecidibilita`)

- Utilizzata anche in fase di ottimizzazione e documentazione

Esempio:

```
1  program divide (input, output);
2  var
3  x, y, quot, rem: integer;
4  begin
5       read(x);
6       read(y);
7       quot := 0;
8       rem := x;
9       while rem >= y do
10           begin
11                quot := quot + 1;
12                rem := rem - y
13           end;
14      print(quot);
15      print(rem);
16 end.
```



- Si vuole selezionare un test che soddisfi il ***criterio di copertura dei comandi***

- Cammino *c = (5, 6, 7, 8, 9, 11, 12, 9, 14, 15)* , contiene tutti i nodi del programma

- Occorre selezionare almeno un dato che causi l'esecuzione del cammino così identificato

AMBIENTE:
variabili e loro valori simbolici +
PC = true (condizione iniziale)

ESECUZIONE:
con valutazione di espressioni che possono restringere il valore di *PC*
se nessuna e` vera

☞ In presenza di rami alternativi scelta del ramo ***determinata dallo scopo dell'esecuzi-one simbolica***

PC, Path Condition – viene ristretta aggiungendo condizioni in congiunzione durante l'esecuzione simbolica

Esecuzione simbolica: esempio

AMBIENTE INIZIALE:
Variabili *x, y, quot e rem* con valore *undef*
Condizione *PC* con valore *true*

ESECUZIONE:
read(x) e *read(y)* (linee 5 e 6)

AMBIENTE:
Variabili *x* e *y* con valori simbolici *valx* e *valy*

Assegnamento di linea 7 (*quot:=0*)

AMBIENTE:
{(x = valx), (y = valy), (quot = 0), (rem = undef), (PC = true)}

Assegnamento di linea 8 (*rem := x*)

AMBIENTE:
{(x = valx), (y = valy), (quot = 0), (rem = valx), (PC = true)}

Comando while di linea 9:
valutazione della condizione *rem >= y*,
che produce come risultato *valx >= valy*

Valutazione delle due espressioni
PC => valx >= valy e *PC => ¬(valx >= valy)*

nessuna e` vera (*PC* vale *true*),
entrambi i rami sono percorribili

☞ Scelta del ramo ***determinata dallo scopo dell'esecuzione simbolica***: selezionare un dato che causi l'esecuzione del cammino *c*

AMBIENTE:
{(x = valx), (y = valy), (quot = 0), (rem = valx),
*(**PC =** true and **valx >= valy**)}*

Assegnamenti di linea 11 e 12

AMBIENTE:
(x = valx), (y = valy), (quot = 1), (rem = valx - valy), (PC = valx >= valy)}

Valutazione della condizione *rem >= y*,
che produce come risultato *valx- valy >= valy*

Valutazione delle due espressioni
PC =>(valx - valy) >= valy e
PC => ◊ ((valx - valy) >= valy)

nessuna e` vera (*PC* vale *true*),

☞ Scelta del ramo ***determinata dallo scopo dell'esecuzione simbolica***: selezionare un dato che causi l'esecuzione del cammino *c*. Scelta di uscire dal ciclo while (una sola esecuzione del ciclo)

AMBIENTE:

{(x = valx),(y = valy),(quot=1),(rem = valx-valy),
***(PC_{new} = PC_{old}** and* **((valx - valy)>= valy)**}

ovvero:

$PC_f = (valx >= valy) \wedge (valx < 2*valy)$

- L'espressione booleana così ottenuta rappresenta la condizione di percorrenza del cammino *c*

- Ogni coppia di valori che soddisfa *PC* , ad esempio la coppia *valx = 5* e*valy = 3*, causa un'esecuzione corrispondente al cammino *c*

- L'insieme {*(valx = 5, valy = 3)*} rappresenta quindi un ***test che soddisfa il criterio di copertura dei comandi***

Sessione di test con esecuzione simbolica

- Consiste dei seguenti passi:

  1. selezione del ***criterio di copertura desiderato*** (il criterio di copertura dei comandi, nell'esempio);

  2. selezione di un ***insieme di cammini*** la cui percorrenza permette di ***soddisfare il criterio scelto*** (il cammino *c*, nell'esempio precedente);

  3. esecuzione simbolica dei cammini selezionati per determinare la condizione di percorrenza di ciascun cammino selezionato (la condizione PC_f nell'esempio precedente);

  4. selezione di un dato di test ***per ogni condizione selezionata*** al passo 3 (l'insieme *{(valx = 5, valy = 3)}*, nell'esempio precedente). L'insieme dei valori così selezionati rappresenta un test per il programma in base al criterio di selezione scelto.

- Il passo 3 ***non può*** essere svolto in modo completamente meccanico, ma richiede in generale l'intervento creativo dell'uomo

Test "in grande"

- Test di moduli software, come singoli programmie descritti in termini di singole istruzioni (*unit test*)

- Architettura software, dove i moduli sono parti di un sistema più complesso (interazioni)

- ***Test "in grande"***, distinto il base a:

  - diverso livello di astrazione a cui il programma è considerato (*test di integrazione* e *test di sistema*)

  - soggetto che esegue il test (*test di accettazione*)

  - informazioni disponibili durante la fase di test (*test di regressione*)

Test di accettazione

- Il test è eseguito dall'utente finale che ha a disposizione solo i manuali richiesti da contratto e/o ritenuti utili da parte del produttore

E` la fase principale di ***convalida*** del sistema

- Non ci sono criteri generali o tecniche formali per il test di accettazione

- Nel caso di prodotto da distribuire sul mercato il test di accettazione è spesso suddivisa in:

  - α-test (il prodotto è rilasciato all'interno dell'organizzazione del produttore, che si propone come utente finale)

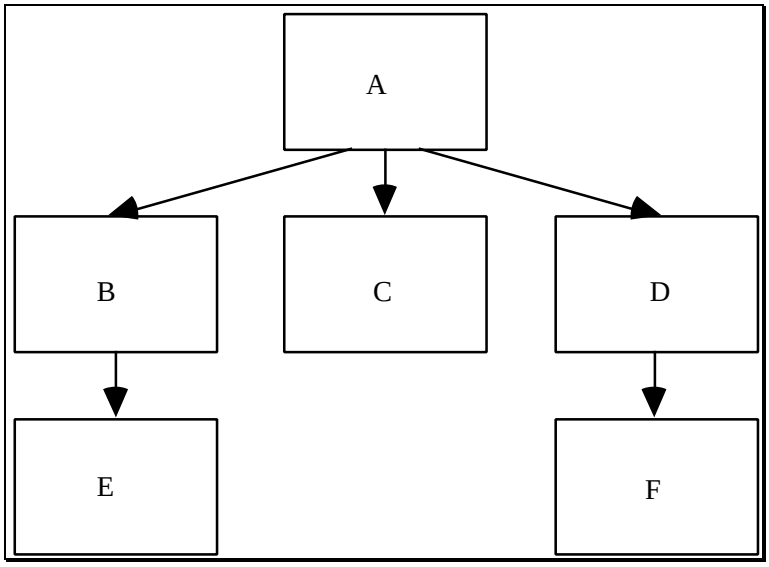
  - β-test (il prodotto è distribuito ad un numero limitato di utenti scelti come campione dell'utenza finale).

Test di regressione

- Utilizzato in fase di manutenzione
- Il programma da testare costituisce una nuova versione del prodotto
- Verifica compatibilità e differenze della nuova versione con la versione precedente
- Più facile da automatizzare, se vengono registrati in opportune basi di dati tanto i dati di test quanto i risultati ottenuti per ciascun test
- Eventuali algoritmi di traduzione se il formato dei dati di ingresso ed uscita è stato modificato passando da una versione alla versione successiva

Test di integrazione

- Il test di un programma composto da più moduli comportata sostanzialmente due passi:
  - il test dei singoli moduli
  - il test dei meccanismi di integrazione dei vari moduli
- Ambiente di esecuzione di un modulo



- Il test di un modulo M richiede la disponibilità dei moduli chiamati da M e che chiamano M
- Nell'esempio, non è possibile testare un singolo modulo senza prima aver testato gli altri moduli del sistema
- Test dei singoli moduli in un ambiente simulato e integrazione successiva, dopo il test individuale

Moduli di simulazione

- ***Moduli di simulazione*** distinti tra:
  - moduli chiamati dal modulo in esame, detti moduli fittizi (*stub*)
  - moduli che chiamano il modulo in esame, detti moduli guida (*driver*)
- Un ***modulo fittizio*** ha la stessa ***interfaccia*** del modulo simulato ed e` realizzato:
  - come modulo che restituisce un valore costante
  - oppure richiedendo l'intervento umano per calcolare il valore da restituire
  - oppre realizzando *prototipi* del modulo simulato
- Un ***modulo guida*** deve simulare l'ambiente chiamante, ovvero occuparsi dell'inizializzazione dell'am-biente non locale del modulo in esame. E` realizzato:
  - fornendo inizializzazioni costanti dell'ambiente non locale
  - oppure richiedendo l'intervento dell'utente
  - come prototipo del modulo simulato

Test di integrazione (cont.)

- Permette di verificare la correttezza dei singoli moduli e dei meccanismi di integrazione tra moduli, ma non il sistema nella sua totalità

- Due approcci fondamentali, per il test di integrazione:

Non incrementale (*big bang test*)

integrazione di tutti i moduli precedentemente testati e verifica quindi dell'intero sistema

Incrementale (*incremental test*)

integrazione di moduli via via prodotti e testati singolarmente

Richiede meno moduli fittizi e moduli guida

Rileva subito eventuali anomalie sulle interfacce

Localizza più facilmente le anomalie

Esercita più a lungo ciascun modulo

Strategie incrementali di test

Test top-down

- Consiste nello sviluppare, testare ed integrare i moduli a partire dai moduli più in alto nella gerarchia delle chiamate
- Non richiede la produzione di moduli guida
- Fornisce prototipi del sistema fin dalle prime fasi di sviluppo: il modulo principale con gli opportuni moduli fittizi costituisce già un primo prototipo dell'intero sistema
- Richiede moduli fittizi molto complessi e pospone la realizzazione dei moduli di ingresso/uscita

Test bottom-up

- Consiste nello sviluppare, testare ed integrare i moduli a partire dai moduli più in basso nella gerarchia delle chiamate
- Non richiede lo sviluppo di moduli fittizi
- Richiede moduli guida piuttosto complessi
- Permette di testare subito i moduli terminali del grafo gerarchico delle chiamate (moduli di calcolo ed interfaccia più complessi)
- Non fornisce però prototipi del sistema

Test di sistema

- Il sistema ottenuto al termine della fase di integrazione non può considerarsi testato adeguatamente

- Proprietà definibili per un sistema nella sua globalità non come proprietà dei singoli moduli (ad esempio, privatezza)

- Un test di sistema è composto di ***vari tipi di test*** applicabili a seconda del ***tipo di sistema*** e del ***tipo di requisiti*** che devono essere soddisfatti

  - ***test di stress*** (condizioni di carico maggiori del previsto)

  - ***test di privatezza*** (verifica proprietà di sicurezza sugli accessi)

  - ***test di robustezza*** (verificare che il sistema risponda adeguatamente a dati non corretti)

  - ***test di configurazione*** (verifica se il sistema funziona correttamente per tutte le configurazioni richieste)

Debugging

Attivita` della fase V&V per la localizzazione e la rimozione delle anomalie alla base dei malfunzionamenti identificati

- Spesso si usano impropriamente tecniche di debugging anziché di test per rilevare malfunzionamenti
- *Debugging*, definito per un programma ed un insieme di dati di ingresso che causano malfunzionamenti rilevati durante la fase di *test*
- Malfunzionamento rilevato in fase di convalida o di specifica
- Rilevare, localizzare e correggere le ***anomalie presenti nel codice***
- Non esiste una corrispondenza biunivoca tra malfunzionamenti ed anomalie

- ***Localizzazione delle anomalie***, nelle singole unità testate o nelle interfacce delle unità coinvolte nel malfunzionamento

- Applicazione intensiva del ***principio di riduzione della distanza tra malfunzionamenti ed anomalie***

  - produzione (parziale o totale) degli stati intermedi dell'esecuzione del programma ed esame del valore di tali stati (*tecniche delle stampe, tracing, strumenti di supporto al debugging, etc.*)

  - debugging per *dump di memoria*, risulta difficile esaminare i risultati ottenuti per la differenza tra la rappresentazione astratta dello stato, legata alle strutture dati del linguaggio di programmazione, e la rappresentazione fornita dallo strumento

- *debugging simbolico* consiste nel produrre gli stati intermedi usando una rappresentazione compatibile con il linguaggio in cui è scritto il programma in esame

- L'esame degli stati prodotti è demandata ***totalmente allo sviluppatore***