**Verifica e convalida**

Si dice che debuggare ed analizzare un codice sia 3x più difficile di scriverlo. Questo implica che se impiego il 100% del mio impegno per riuscire ad implementare qualcosa, molto probabilmente non riuscirò a debuggarlo. Vediamo metodi per facilitare questa operazione.

**Terminologia**

*NOTE: The fault tolerance discipline distinguishes between a human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error).*

* **Convalida**: confronto del software con i requisiti del committente (***requisiti informali***).
* **Verifica**: confronto del software con le specifiche prodotte dall’analista (requisiti formali, aka ***specifiche***).
* **Malfunzionamento** o guasto: funzionamento incorretto del programma, il codice è scritto correttamente, ma il risultato ottenuto non è corretto. *es. funzione “raddoppia” che moltiplica per 3.*
* **Difetto** o **anomalia**: è una sezione specifica del codice ed è condizione necessaria ma non sufficiente per il verificarsi di un malfunzionamento. *es. nella funzione precedente, la riga dove si fa n\*3.*
* **Sbaglio**: è la causa del difetto/anomalia, generalmente un errore umano.

*es. il lancio di Ariane 5, lanciato nel 1996, è esploso a causa di uno sbaglio umano, il difetto era un’eccezione di overflow dovuta alla conversione da un 64 bit float ad un 16 bit int, che ha portato al malfunzionamento dell’esplosione del razzo.*

**Tecniche**

Distinguiamo le tecniche di verifica e convalida in due tipi:

* **Statico**: basato sull’analisi del codice. Sono tecniche complesse da ideare ma veloci.
* **Dinamico**: basato sull’esecuzione del programma. Sono le tecniche più semplici da eseguire.

Immagine che contiene testo, diagramma, linea, schermata

Descrizione generata automaticamenteL’obiettivo delle tecniche di analisi del codice è quello di **verificare perfettamente una proprietà** arbitraria tramite una prova logica (dal lato statico) o una ricerca esaustiva degli stati possibili del programma (dal lato dinamico). Questo è estremamente complesso dunque spesso impossibile, ma possiamo “accontentarci”.

Approssimare un’analisi però dev’essere fatto con criterio: sebbene la tecnica ideale sia eccessivamente dispendiosa, non spendere abbastanza risorse porterebbe ad un’analisi degenerante:

* per **estrema semplificazione delle proprietà**: sono stati rilassati troppo i criteri di valutazione delle proprietà. es. se il programma deve gestire correttamente un puntatore, verificare che non sia null è una semplificazione estrema.
* per **estrema inaccuratezza pessimi**stica: “se non riesco a dimostrare l’assenza di un problema, allora assumo sia presente”.
* per **estrema inaccuratezza ottimistica**: “se non riesco a dimostrare la presenza di un problema, allora assumo non esista”.

Tra tutti i metodi tra l’ideale e questi estremismi, in questo corso vogliamo porre particolare enfasi sull’utilizzo di **metodi formali** (analisi di data flow e dimostrazione della correttezza delle specifiche logiche), il **testing** (black, grey o white box) ed il **debugging**.

**Correttezza**

**Correttezza di un programma**: un programma P può essere visto come una funzione con un dominio ed un codominio di insiemi di dati. P(d) indica l’esecuzione di P sul dato in ingresso d del dominio. La funzione ausiliaria ok(P, d) indica se il risultato è corretto.

*Un programma P è corretto ⇔ ok(P,d) ∀d ∈ D*

Un **test T** è un sottoinsieme del dominio: eseguire un test consiste nell’esecuzione del programma ∀t ∈ T.

*Un programma passa un test sse ok(P,T) ⇔ ok(P,t) ∀t∈T*

E visto che lo scopo di un test è di rilevare anomalie, un test T ha **successo** se rileva uno o più malfunzionamenti presenti nel programma P.

*successo(T,P) ⇔ ∃t ∈ T tc ⨼ok(P,t)*

**Test ideale**

*Un test è ideale per P sse ⨼successo(P,T) ⇒ ok(P,D)*

Ossia se il superamento del test senza malfunzionamenti implica la correttezza del test. In generale, è **impossibile trovare un test ideale**. Non è un risultato dimostrato, ma la tesi di Dijkstra dice che il test di un programma può dimostrare malfunzionamenti ma non può dimostrarne l’assenza.

Naturalmente non possiamo scorrere tutto il dominio D per i test: ci serve definire dei **criteri di selezione** che ci permettano di trovare un T che approssimi il test ideale. Un criterio deve essere:

* **Affidabile**: un criterio C si dice affidabile se, presi T1 e T2, allora o entrambi hanno successo o nessuno dei due ha successo;

*affidabile(C,P) ⟺ (∀T1,T2 ∈ C  successo(T1,P) ⇔ successo(T2,P))*

* **Valido**: un criterio C si dice valido se, qualora P non sia corretto, esiste almeno un T selezionato in base a C che ha successo per il programma P;

*valido(C,P) ⟺ (¬ok(P,D) ⇒ ∃T ∈ C | successo(T,,P))*

In classe abbiamo visto un esempio, preso il codice:

**static** int raddoppia(int par) {

int risultato;

risultato = (par \* par);

**return** risultato;

}

* *T = {sottoinsiemi di 0,2} è un test affidabile perché funziona sia con 0 sia con 2, ma non è valido perché chiaramente non è corretto e nessun caso di test rileva anomalie;*
* *T = {sottoinsiemi di 0,1,2,3,4} non è affidabile perché se 0 e 2 passano, gli altri no, ma è valido perché ci sono casi di test che rilevano malfunzionamenti;*
* *T = {sottoinsiemi di D con valore di almeno 18} è sia affidabile che valido;*

Naturalmente, il giudizio su questi test è molto semplice perché il codice è facile da leggere, ma per programmi complessi, questa analisi diventa molto più complessa.

**Conclusione**: possiamo dire che

*(affidabile(C,P) ∧ valido(C,P) ∧ (∃T∈C | ¬successo(T,P))) ⟹ ok(P,D)*

**NB**: un criterio può essere automaticamente affidabile se il criterio seleziona un solo test.

**Evidenziare un’anomalia**

Per notificare un malfunzionamento, un caso di test deve:

* **Eseguire il comando** che contiene l’anomalia;
* Portare il sistema in uno **stato scorretto** tramite l’anomalia;
* **Propagare lo stato scorretto** fino all’uscita del codice, in modo da produrre un output diverso dall’expected;

Come valutare quanti e quali casi di test implementare è un procedimento molto dipendente dal codice da testare, ma possiamo guidare questo ragionamento tramite delle **metriche di copertura**: maggiore è la copertura di un test, più è da preferire rispetto ad altri.

1. **Criterio di copertura dei comandi**: in base al primo punto notiamo una cosa, noi vogliamo eseguire tutti i comandi, per poter notare quale contiene un’anomalia.

*Criterio della copertura dei comandi: è rispettato sse ogni comando eseguibile dal programma è eseguito per almeno un caso di test t ∈ T.*

La metrica di questo criterio risulta essere la frazione di comandi testati rispetto a tutti i comandi del programma. A laboratorio abbiamo già visto questo criterio grazie alla funzione “run with Coverage” di Intellij.

|  |  |  |
| --- | --- | --- |
| 01 void main(){  02 float x, y;  03 read(x);  04 read(y);  05 if (x != 0)  06 x = x + 10;  07 y = y / x;  08 write(x);  09 write(y);  10 } | Esempio criterio copertura comandi | Come rule of thumb, convertendo il codice in un diagramma di flusso, un test selezionato con questo criterio cerca di visitare quanti più nodi possibili tramite uno o più casi di test.  Come si può notare, questo criterio non garantisce la correttezza del programma; infatti, se scegliessi il caso di test <3,7> coprirei tutti i comandi, ma non rileverei l’anomalia della possibile divisione per 0. |

1. **Criterio di copertura delle decisioni**: un test T soddisfa il criterio di copertura delle decisioni sse ogni decisione effettiva viene resa sia vera che falsa in corrispondenza di almeno un caso di test t di T.

|  |  |  |
| --- | --- | --- |
| 01 void main(){  02 float x, y;  03 read(x);  04 read(y);  05 if (x != 0 && y > 0)  06 x = x + 10;  07 else  08 y = y / x  09 write(x);  10 write(y);  11 } | Esempio criterio decisioni | Questo criterio implica il criterio della copertura dei comandi: come nell’esempio a fianco, è possibile visitare tutte le decisioni sse si visitano tutti gli archi, quindi anche i nodi.  La metrica di questo criterio è il rapporto tra le decisioni rese sia vere che false su tutte le decisioni possibili.  **NB**: per decisione si intende il “blocco” intero di condizioni in una clausola if, anche quando è composto da più condizioni. |

1. **Criterio di copertura delle condizioni**: un test T soddisfa il criterio di copertura delle condizioni sse ogni singola condizione viene resa sia vera che falsa in corrispondenza di almeno un caso di test t contenuto in T.

|  |  |  |
| --- | --- | --- |
| 01 void main(){  02 float x, y;  03 read(x);  04 read(y);  05 if (x != 0 || y > 0)  06 y = y / x;  07 else  08 y = (y + 2) / x  09 y = y / x;  10 write(x);  11 write(y);  12 } | Esempio criterio decisioni | La metrica di questo criterio è la quantità di condizioni che sono state rese sia vere che false da almeno un caso di test su tutte le condizioni nel codice.  Questo criterio **NON** implica i precedenti: è sufficiente prendere l’esempio a fianco con il test:  T = { <0,5>, <5,-5> }  Che soddisfa il criterio delle condizioni ma non visita tutti i comandi (la riga 8 non viene mai coperta), quindi neanche questo criterio garantisce la correttezza del programma. |

1. **Criterio di copertura delle condizioni e delle decisioni**: un test T soddisfa questo criterio quando qualsiasi decisione ed ogni singola decisione viene resa sia vera che falsa per vari casi di test t. Intuitivamente, si tratta dell’intersezione dei due criteri precedenti; quindi, implica il criterio di copertura delle condizioni, delle decisioni e conseguentemente anche quello dei comandi, ma neanch’esso garantisce la correttezza del programma.

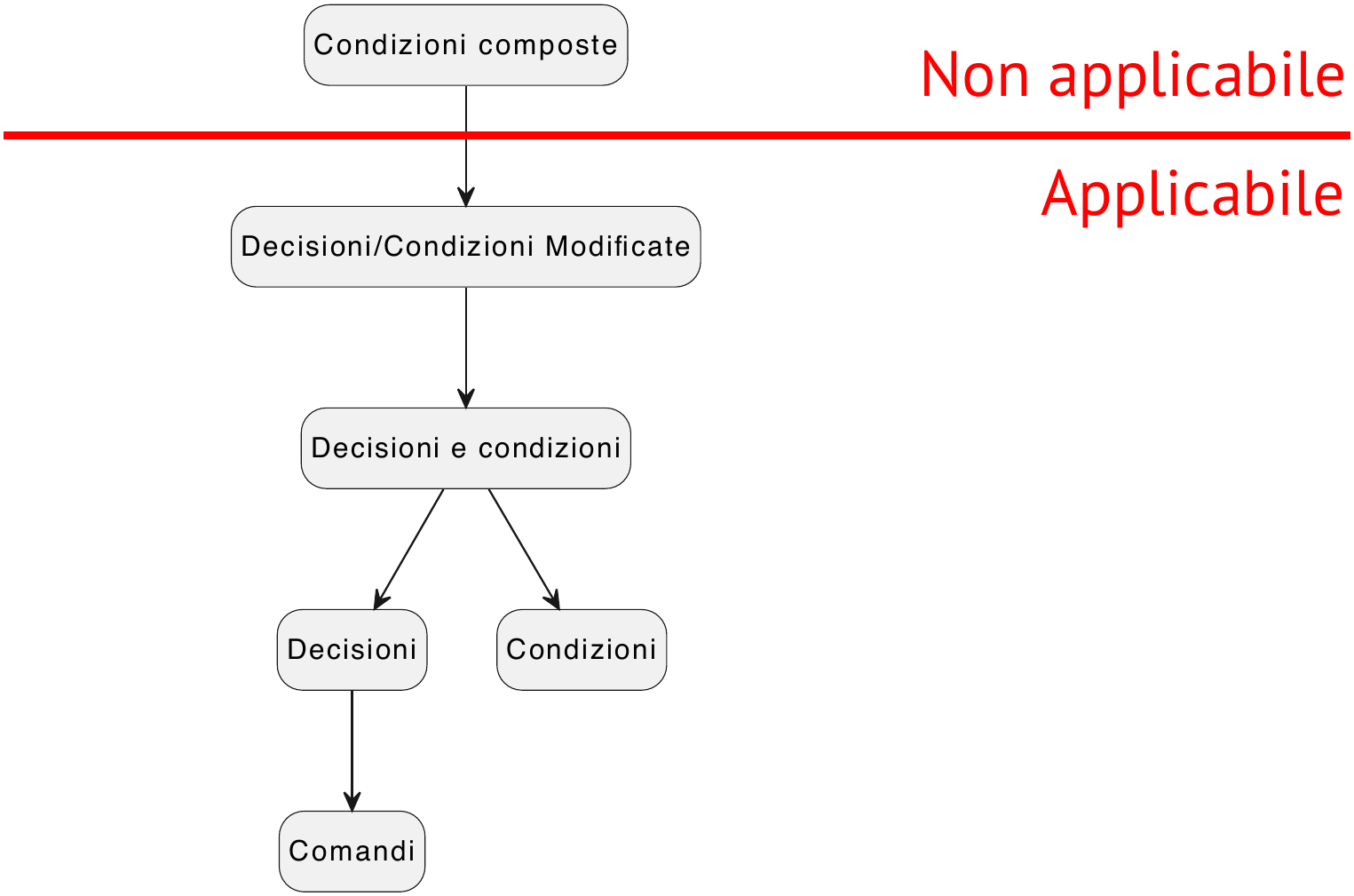
Nell’esempio precedente, T = {<0,-5>,<5,5>} lo soddisfa ma non rileva l’anomalia a riga 6.

1. **Criterio di copertura delle condizioni composte**: un test T soddisfa il criterio di copertura delle condizioni composte sse per ogni decisione composta, vengono utilizzate tutte le combinazioni possibili di verità delle condizioni componenti.

Questo criterio implica che nel caso si abbia una decisione nel formato (condizione && condizione), in T devono figurare casi di test che prevedano <V,V>, <V,F>, <F,V>, <F,F>. Inoltre, implica il criterio precedente (e quindi tutti gli altri).

La natura combinatoria delle dimensioni di un test che rispetta questo criterio lo rende poco fattibile e spesso inefficiente se ci sono combinazioni impossibili da ottenere e dunque inutili da testare.

1. **Criterio di copertura delle condizioni e delle decisioni modificate**: per ovviare al problema della dimensione esponenziale dei casi di test del criterio precedente, si possono testare solamente le condizioni base più rilevanti, ossia quelle che a parità di configurazione di tutte le altre, il valore della condizione base modifica il valore della condizione composta.



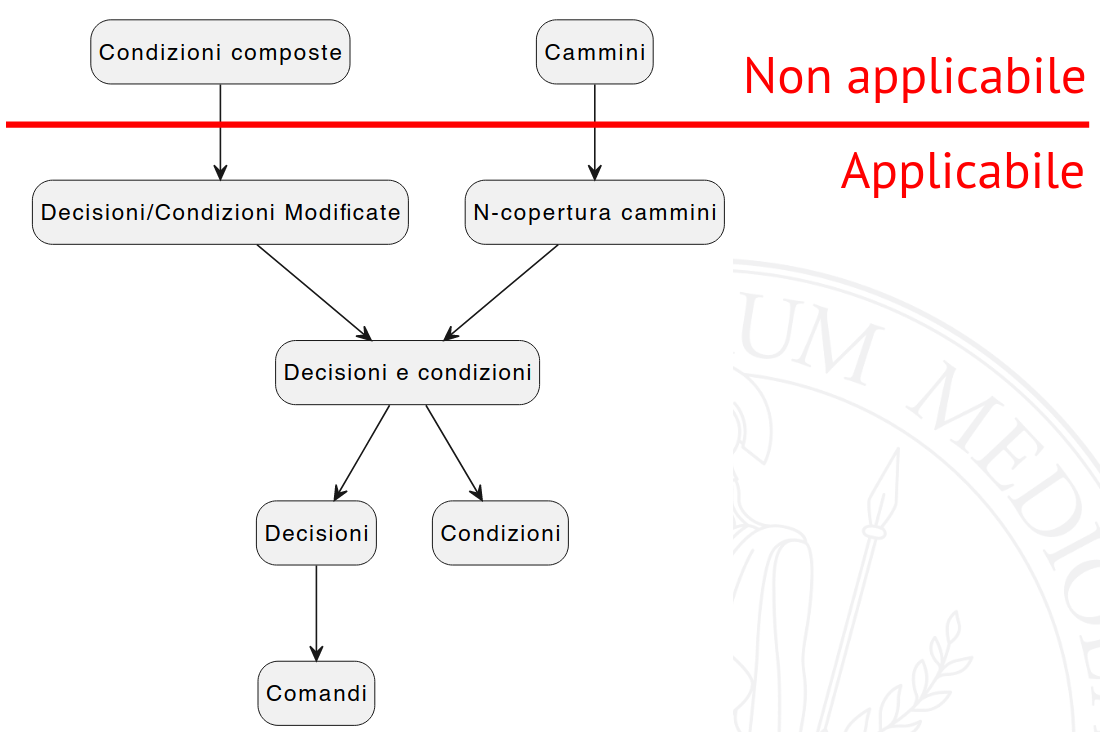
1. **Criterio di copertura dei cammini**: un test T soddisfa il criterio di copertura dei cammini sse ogni cammino del grafo di controllo del programma viene percorso per almeno un caso di test in T.

La metrica di questo criterio è il rapporto tra i cammini percorsi e tutti i cammini percorribili.

È un criterio spesso impraticabile, soprattutto nel caso di cicli perché i cammini diventano infiniti.

1. **Criterio di N-copertura dei cicli**: un test T soddisfa il criterio di n-copertura sse ogni ciclo del programma viene eseguito 0,1,…,n volte per almeno un caso t di T.

NB: il caso n=2 è il minimo, perché permette di testare zero iterazioni, una sola iterazione e molteplici iterazioni. D’altronde se n = 1, sarebbe un costrutto indistinguibile da un if.



**Analisi statica**

Vediamo un’altra analisi, differente dal testing, che ci permette di valutare il programma. Esaminiamo l’**insieme finito di istruzioni** del programma, un’operazione sensibilmente **meno costosa** dell’analisi dinamica di tutti gli stati possibili di esecuzione. Naturalmente, non rileverà errori dipendenti da componenti interne al programma, ma permette di notarne altri.

1. **Analisi Data Flow**

Si tratta di un tipo di analisi inizialmente utilizzata nel campo dell’**ottimizzazione dei compilatori** (che svolgono operazioni di controllo lessicale, sintattico, di tipo, ed *analisi di data flow*), e non si basa sul flusso dinamico dei dati ma sulle operazioni definibili staticamente:

* **Definizion**e **d**: se il comando assegna un valore ad una variabile;
* **Uso** **u**: se il comando richiede il valore di una variabile;
* **Annullamento** **a**: se al termine dell’esecuzione dell’istruzione il valore della variabile non è significativo (es. dichiarazione senza inizializzazione, termine dello scope);

Da qui definiamo, per ogni variabile, una **sequenza di operazioni**: definizione, uso, annullamento. Questa sequenza, per essere corretta, deve seguire delle regole:

1. L’***uso*** di una variabile deve sempre essere preceduto da una sua definizione senza annullamenti intermedi (ua è un errore!);
2. La ***definizione*** di una variabile deve sempre essere seguita da un uso: se fosse subito seguita da un’altra definizione, la prima ha poco senso; se fosse subito seguita da un annullamento, è un valore che non ha senso di esistere (dd, da sono errori!);
3. L’***annullamento*** di una variabile deve sempre essere seguito da una definizione (aa, au sono errori!);

*Vediamo un semplice esempio di analisi di data flow:*

|  |  |
| --- | --- |
| **void** swap(int &x1, int &x2) {  int x1;  x3 = x1;  x3 = x2;  x2 = x1;  } | * x1: …auua…; * x2: …ud…; * x3: …dd…; |

Nel caso in cui si volesse **estendere** l’analisi della sequenza di operazioni a più variabili, definiamo come P([0,…,n],a) la sequenza di operazioni per la variabile a, nelle righe da 0 ad n del programma P, ad esempio:

|  |  |
| --- | --- |
| 01 **void** main() {  02 float a, b, x, y;  03 read(x);  04 read(y);  05 a = x;  06 b = y;  07 **while** (a != b)  08 **if** (a > b)  09 a = a - b;  10 **else**  11 b = b - a;  12 write(a);  13 } | P([0,1,2,3,4,5,6,7,8,9,10,11,12,13], a) = a2, d5, u7, u8, u9, d9, u11, u12, a13 |
| La notazione precedente ***non è particolarmente espressiva*** in caso di cicli e condizioni, perché i cicli sono potenzialmente infiniti e non tutte le sottosequenze vengono eseguite in caso di condizionali. Introduciamo:   * P(**[1→]**, a): per indicare tutte le operazioni dalla 1; * La pipe **|** per indicare operazioni mutualmente esclusive; * L’asterisco **\*** per indicare operazioni eseguibili da 0 a n volte; |

1. **Testing e criteri di copertura**

Torniamo al **testing**, applicando ciò che abbiamo appena visto riguardo l’analisi di data flow:

* affinchè si presenti un malfunzionamento dovuto ad un’anomalia in una **definizione**, è necessario un uso del valore assegnato nella definizione;
* un **ciclo** dovrebbe essere ripetuto di nuovo se verrà usato un valore definito nell’iterazione precedente.

Introduciamo della terminologia utile:

* : insieme delle variabili definite nel nodo i del diagramma di flusso (corrisponde alla riga di codice);
* : insieme dei nodi j tali che:
  + x appartiene a ;
  + x è usato in j;
  + esiste un cammino da i a j, senza definizioni di x;

Sostanzialmente sono tutti i nodi j che potrebbero utilizzare il valore definito in i.

* 1. **Criterio di copertura delle definizioni**: un test T soddisfa il criterio di copertura delle definizioni sse per ogni nodo i e variabile x apppartenente a def(i), T include un caso di test che esegue almeno un cammino di du(x,i).

*T∈Cdef ⟺ ∀i∈P, x∈def(i)  ∃j∈du(i,x), t ∈ T  che esegue un cammino da i a j senza ulteriori definizioni di x*

* 1. **Criterio di copertura degli usi**: un test T soddisfa il criterio di copertura degli usi sse per ogni nodo i e variabile x ∈ def(i), per ogni elemento j ∈ du(x,i), T include un caso di test che esegue un cammino libero da definizioni da i a j.

*T∈Cdef ⟺ ∀i∈P, x∈def(i), j∈du(i,x), t ∈ T  che esegue un cammino da i a j senza ulteriori definizioni di x*

**Attenzione**: non include il criterio di copertura delle definizioni perché nel caso in cui non ci fosse nessuna definizione questo vale ma quello di prima no.

* 1. **Criterio di copertura dei cammini DU**: questo criterio richiede la copertura di tutti i cammini du.

*T∈CpathDU ⟺ ∀i ∈ P, ∀x ∈ def(i), ∀j ∈ du(i,x),*

*∀ cammino da i a j senza ulteriori definizioni di x*

*∃t∈T che lo esegue*

Infine, notiamo che non è necessario che l’analisi del flusso avvenga necessariamente sulle variabili, ma esistono anche **altri oggetti** verificabili. A lezione abbiamo fatto un esempio con i File, con le operazioni di apertura, chiusura, lettura, e scrittura.

Come ci sono altri oggetti, abbiamo anche altri criteri, come la **copertura del budget**: utilizzare dei test che utilizzano tutte le risorse disponibili.

**Processo di review**

Fin’ora abbiamo visto tecniche formali per l’approccio alla verifica ed alla convalida del codice, largamente operazioni effettuabili tramite dei tool. Ma non è sempre l’approccio corretto: se chi scrive il codice non ha compreso a pieno l’obiettivo del programma, il suo codice corretto presenterà comunque dei problemi. Per questo, vediamo alcune tecniche di ***review*** del codice, dove un utente umano esamina il codice.

**Bebugging**

Nei rari casi in cui serva testare un codice similmente corretto, si può utilizzare un metodo particolare, dove si incarica il team di sviluppo di **inserire n errori** dentro il codice prima di mandare il programma a chi lo deve testare. Questa aggiunta incentiva il team di testing a trovarli tutti, ed è probabile che nel mentre troveranno altri errori non facenti parte di quelli “artificiali”.

La metrica utilizzata per valutare questo metodo di verifica è la percentuale trovata di errori artificiali. È una buona tecnica, ma non va presa troppo rigorosamente perché potrebbe risultare difficile valutare quanto rendere complessi i bug artificiali.

**Analisi mutazionale**

Vediamo un’evoluzione del bebugging: dato un programma P ed un test T, generiamo un **insieme di programmi**  simili al nostro programma P, ed eseguire lo stesso test T su tutti. Gli elementi di sono detti **mutanti**.

Ci si aspetta che **se P è corretto, allora i mutanti sono sbagliati**; dunque, per ognuno di essi almeno un caso di test t deve produrre un risultato diverso. Se così non fosse, vorrebbe dire che la parte in cui P ed il mutante differiscono non viene testata adeguatamente.

*Un test T soddisfa il criterio di copertura dei mutanti sse per ogni mutante , esiste almeno un caso di test t T la cui esecuzione produca un risultato diverso per da P.*

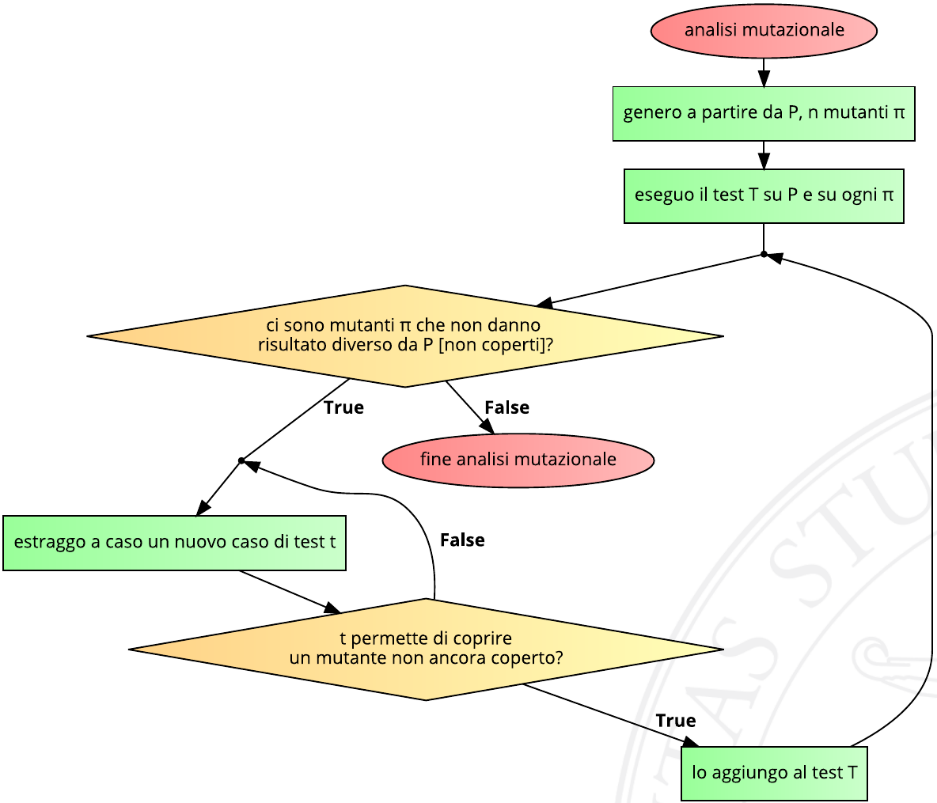
La metrica di valutazione per questo criterio è la quantità di riconosciuta come diversa da P su tutti i mutanti.

Qui però sorgono dei dubbi: i mutanti, quanto devono essere simili a P? E quanti dobbiamo produrne? Il caso ideale utilizza mutazioni **poco differenti** ed in **quantità abbondanti**, una per ogni possibile difetto. Questo però richiederebbe pressoché infiniti mutanti, quindi è auspicabile una quantità nell’ordine delle centinaia come minimo.

Al fine di produrli, utilizziamo gli **operatori mutanti**, ossia delle funzioni che generano uno o più mutanti di P. Gli operatori più semplici generano mutanti con differenze puramente sintattiche, in modo da modificare il programma senza creare errori di compilazione.

Distinguiamo delle **classi di operatori** in base a ciò su cui apportano modifiche:

* **costanti** e **variabili**;
* **operatori** ed **espressioni**: come la modifica di condizioni;
* **comandi**: come la modifica fa cicli a condizionali;

Tramite questi operatori è possibile utilizzare dei **tool per la generazione dei mutanti**, ma a caro prezzo, perché devono modificare il codice, compilarlo per verificare che non ci siano problemi, e verificare che la modifica non ricada nello stesso dominio di verifica di un altro mutante già generato. Per questo, alcuni tool si basano sulla modifica dell’eseguibile, con il tradeoff di generare programmi potenzialmente non ottenibili tramite compilazione.

L’algoritmo a fianco descrive l’automatizzazione dell’analisi mutazionale, ma attenzione, perché non garantisce che terminerà. Infatti, è sufficiente che sia sintatticamente diverso ma semanticamente identico a P, e sarà impossibile trovare un nuovo t.

**Object Oriented Testing**

Nel caso di testing di oggetti, i metodi visti finora **perdono di efficacia**: un programma non è più una semplice funzione con dominio di input e dominio di output (era vero solo per linguaggi procedurali, dove i programmi sono solamente metodi), bensì una manipolazione di un oggetto, che a sua volta è composto da stato e metodi. Per non parlare della complicazione aggiunta dal dynamic dispatch per l’analisi statica dei cammini.

Iniziamo da un quesito importante:

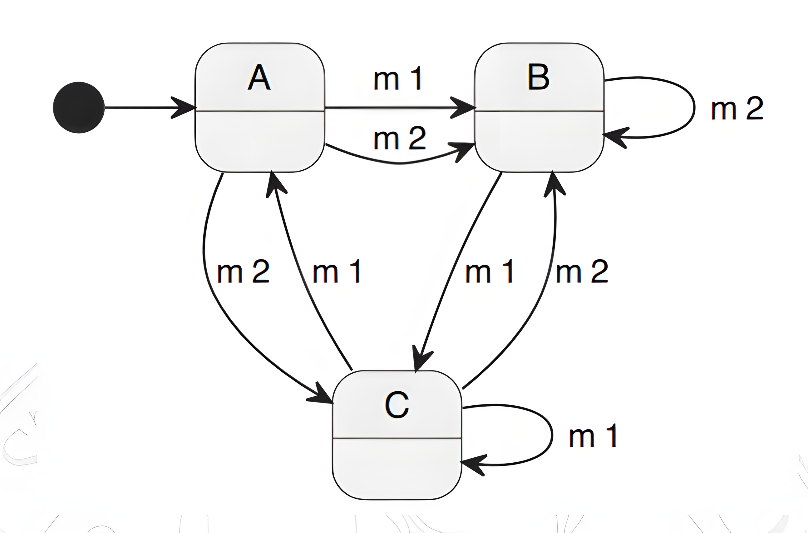
Se implemento una sottoclasse, i metodi ereditati vanno testati? Il consenso generale è **sì**, applicando questi metodi ai nuovi contesti per cui la sottoclasse è pensata, si possono trarre dei benefici nel testare anche questi metodi. Quindi, si può dire che non si eredita il testing, ma si possono ereditare i casi di test dalla superclasse, qualora fossero corretti.

**Class testing**

Per testare una classe è necessario:

* **isolarla** tramite il mocking e gli stub: in questo modo possiamo accertarci della correttezza indipendentemente dal contesto;
* adottare **strumenti per l’estrazione** di campi e stato per bypassare l’incapsulamento;
* **costruire la classe di test** con istanze e chiamate a metodi necessari per soddisfare il criterio di copertura scelto;

Per quest’ultimo punto notiamo una cosa importante: è possibile definire un nuovo criterio di copertura, costruito appositamente per testare le classi.

**Copertura della classe**: avendo a disposizione una definizione statica dello stato dell’oggetto, come uno state diagram, possiamo estendere i criteri di copertura visti precedentemente pensando di voler…

…**coprire tutti i nodi**: per ogni stato, si implementa un caso di test che lo raggiunge.

…**coprire tutti gli archi**, quindi testare tutti i metodi per ogni stato.

…**coprire tutte le coppie di cammini input/output**.

…**coprire tutti i cammini identificabili** nel grafo.

Quest’ultimo è un criterio di copertura impossibile, sempre a causa della quantità eccessiva di cammini in un grafo potenzialmente molto complesso.

**White box vs black box**

Il testing può essere white box se, in fase di scrittura dei test, si ha conoscenza del codice del SUT, mentre si parla di test black box altrimenti.

**Testing funzionale**

Questo tipo di testing non si basa più sulla struttura del codice, bensì sul suo **funzionamento**, valutando il programma dal **punto di vista dell’utente finale**. Si tratta quindi di un **testing blackbox**, e può essere un metodo di testing obbligato (se si riceve dal committente una richiesta di validazione di un lavoro esterno) oppure volontario per approcci differenti al testing.

**Tecniche principali**

* **Metodi basati su grafi**: operano sui vari diagrammi già visti prima, come quello di sequenza;
* Suddivisione del dominio in **classi di equivalenza**: partizioniamo il dominio dei dati in ingresso in base a come vengono elaborati dal programma. In questo modo possiamo individuare quale categoria di input porta ad errori (es. elaborazione di numeri negativi).

Per cercare le classi di equivalenza, si può notare la base da cui partire: ***input validi*** ed ***input non validi***. Questo permette di verificare che il programma reagisca bene ad input malformati.

**NB**: *non sono classi di equivalenza rispetto all’output, ma rispetto al tipo di elaborazione che ricevono.*

**Category partition**: è una tecnica standard di suddivisione in classi di equivalenza utilizzabile a diversi livelli, dai test di unità, ai test di integrazione ed ai test di sistema. Consiste nei seguenti passi:

1. **Analizzare le specifiche**: da queste, si ricavano, per tutti i parametri, ciò che li caratterizza (contenuto, valore, etc.). Da queste caratteristiche definiamo delle **categorie**.
2. **Scegliere dei valori**: per ogni categoria, si scelgono tutti ed i soli **casi significativi**, ossia ciò che si ritiene abbia senso testare. Questa operazione richiede una certa esperienza per essere applicata con efficacia.

Spesso il mantra “*nessuno, uno, molti*” si rivela utile per questa parte (es. per un reader leggere un file vuoto, con un solo carattere o con più caratteri).

1. **Determinare vincoli** tra le scelte: trovati i valori significativi per tutti i parametri, serve testare l’interazione tra di loro. Naturalmente, testare tutte le combinazioni di valori sarebbe troppo costoso, motivo per cui si utilizza il **pairwise testing**, che genera un caso di test per ogni possibile coppia di valori.

Per ridurre ulteriormente il carico di test, si possono anche definire i **vincoli** tra i valori, per non testare coppie che sappiamo essere impossibile generare in scenari reali (es. testare Edge su MacOS quando Edge non è disponibile per quel sistema operativo).

1. **Scrivere i test**: si enumerano tutti i test utili finchè non se ne hanno un numero ragionevole.

* **Analisi dei valori limite** (chiamati anche test di frontiera): si introducono dei test specifici per valori vicini agli estremi del dominio degli input e delle categorie in cui può essere suddiviso (le classi di equivalenza menzionate prima).
* **Collaudo per confronto** (con vecchie versioni): si confronta il funzionamento del programma vecchio con il programma nuovo.

**Testing funzionale e OO**

Visto che il testing funzionale non si basa su codice, potrebbero esserci problemi: non esiste una struttura gerarchica che possa guidare l’integrazione, perché le relazioni tra classi (associazione, aggregazione, dipendenza) sono spesso caotiche e cicliche. Quindi seppur si possa applicare facilmente alle singole unità, non vale lo stesso per il sistema completo, per cui ci si dovrà avvalere dei diagrammi UML visti in passato per semplificare il processo.

**Software inspection**

Si tratta di una tecnica manuale per individuare e correggere gli errori, molto flessibile grazie all’impiego di personale umano. Consiste in un’attività di gruppo ben definita: la **Fagan Code Inspection**, definita in maniera molto rigorosa. I membri del gruppo vengono divisi in:

* **Moderatore**, che coordina i meeting, gestisce i membri del gruppo e controlla il processo;
* **Readers** e **testers**, che leggono il codice al gruppo;
* **Autore**, un partecipante passivo che risponde alle domande;

Il **processo** è diviso in fasi:

1. **Planning**: scelta dei partecipanti e pianificazione dei meeting.
2. **Overview**: assegnare gli ruoli ai partecipanti.
3. **Preparation**: attività svolte offline per la comprensione del codice o la struttura del progetto intero.
4. **Inspection**: il goal è trovare e registrare il maggior numero di difetti senza correggerli, parafrasando linea per linea. È fondamentale notare che non vanno corretti, si lascia che sia l’autore del codice a decidere come correggerli durante la fase di rework.

Per rendere questa fase efficiente, la durata dell’ispezione è limitata e si procede in base ad una **checklist** contenente errori passati, per verificare che non siano stati commessi nuovamente. Individuando un nuovo sbaglio si popolerà la checklist. Esistono anche checklist pubbliche in base al linguaggio (es. la checklist della NASA per C).

Un altro strumento utile potrebbero essere aiuti alla comprensione del codice come rappresentazioni grafiche ed evidenziazione di parti rilevanti.

1. **Rework**: l’autore corregge i difetti individuati.
2. **Follow** **up**: fase opzionale, si effettua un’altra ispezione con eventuale rework.

**NB**: il programmatore non va incentivato a nascondere difetti; quindi, non devono essere utilizzati per la valutazione del personale.

***Fagan’s Law****: inspections significantly increase productivity, quality and project stability.*

**Active Design Reviews**: un revisore non preparato potrebbe stare seduto senza dire nulla; quindi, è stata sviluppata una variante dove l’autore fa domande ai revisori, che dovendo rispondere sono costretti a partecipare.

**Vantaggi del testing di gruppo**

***Weinberg’s Law****: a developer is unsuited to test their code.* Questo perché lo sviluppatore ha già una visione del codice; dunque, si rischia che parafrasando le istruzioni, le corregga subconsciamente. Quindi torna utile avere un gruppo esterno a controllare il codice.

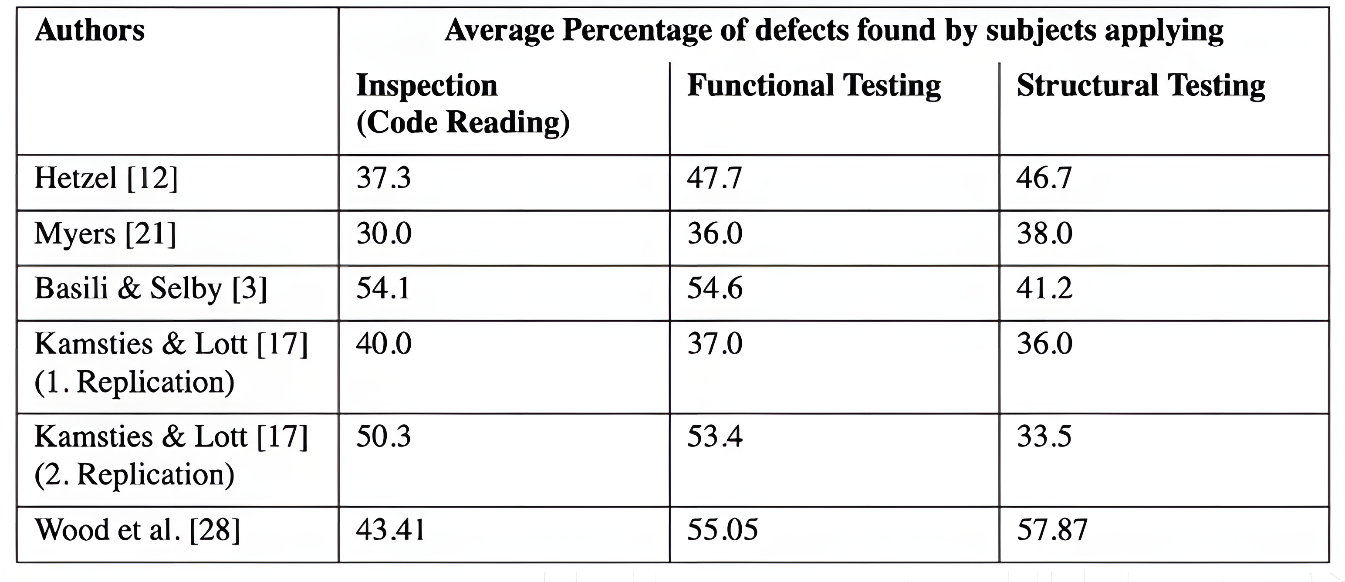
Inoltre, è una tecnica non solo **rigorosa**, ma **evolve in positivo** nel tempo grazie alla checklist che migliora man mano che si individuano ulteriori difetti.

**Svantaggi del testing di gruppo**

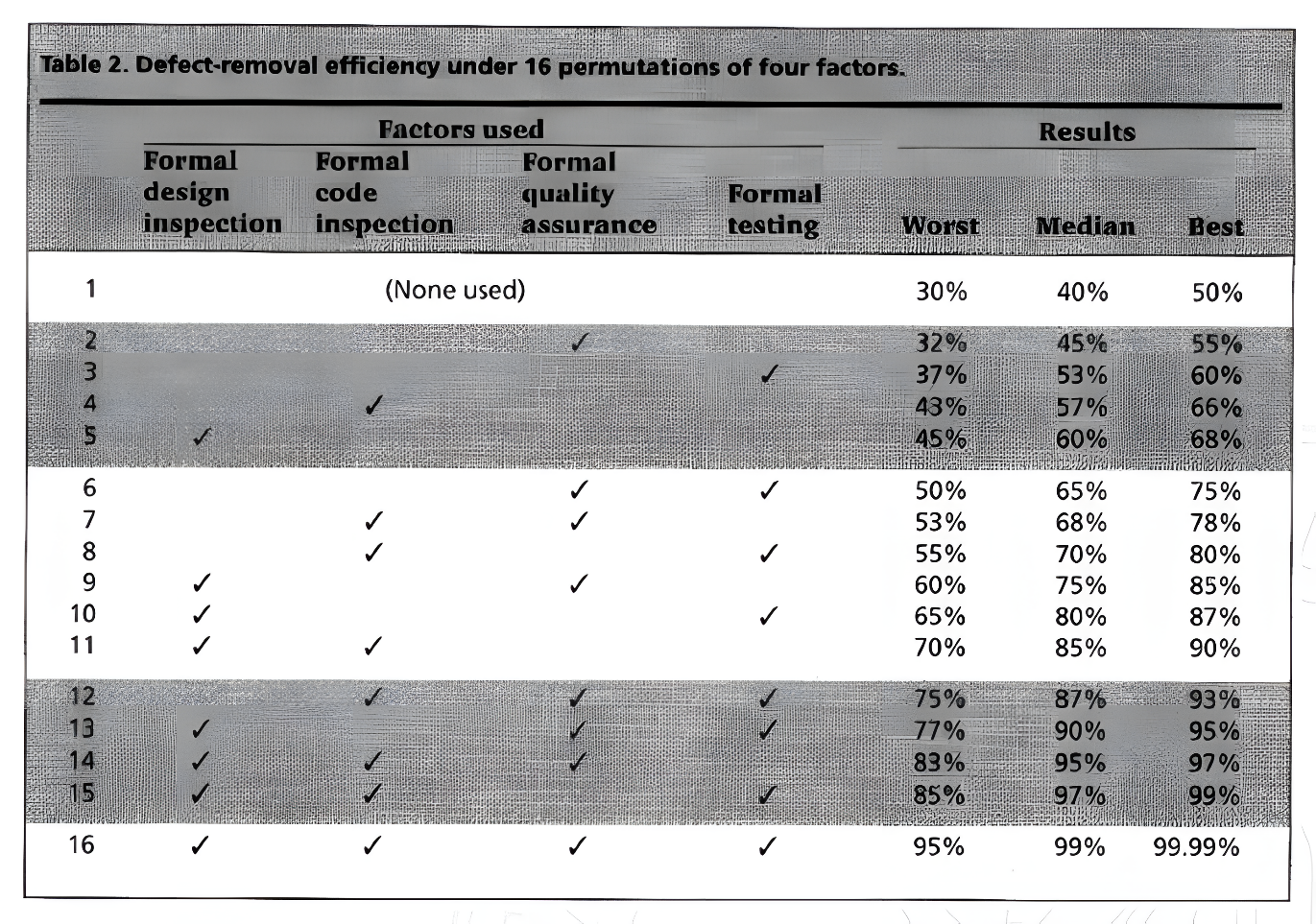
Il testing di gruppo ha dei **limiti**: innanzitutto, è limitato al testing unitario, perché la mente umana può elaborare una quantità limitata di informazioni. In aggiunta, non è un processo incrementale, perché ad ogni rework il codice potrebbe cambiare talmente tanto da necessitare di ripartire da zero.

**Confronto e combinazione delle tecniche di testing**

Ecco di seguito una tabella risultante da più studi applicati a tutti i metodi di testing, volto a **confrontare** quale sia la migliore: come si può notare, tra tutti gli studi **non c’è una tecnica che ne esce vincitrice**, e **nessuna arriva al 100%** perché **ognuna verifica aspetti differenti** e nessuna di queste è perfetta.



Proprio perché si tratta di tecniche differenti con approcci differenti, però, sarebbe corretto affermare che, non rilevando gli stessi errori, allora **combinarle potrebbe portare ad effetti ancora migliori**. E ce lo conferma un altro studio:



***Hetzel-Myer’s Law****: a combination of different V&V methods outperforms any single method alone.*

**Debugging**

Il debugging è un’operazione mirata a **localizzare e rimuovere anomalie**, basata sulla ***riproducibilità*** di un malfunzionamento. È un’attività che richiede anche una buona conoscenza del codice e del contesto del programma, quindi è molto complessa ed articolata.

Immagine che contiene testo, Carattere, bianco, linea

Descrizione generata automaticamente“*Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it*” - Brian W. Kernighan

In particolare, spieghiamo la complessità del debugging in base a varie cause:

* Esistono ***moltissimi tipi di relazioni*** anomalia-malfunzionamento;
* La relazione anomalia-malfunzionamento ***non è biunivoca***: un malfunzionamento può essere causato da più anomalie, come un’anomalia potrebbe causare più malfunzionamenti;
* La correzione di un’anomalia richiede la ***modifica del codice***, operazione che potrebbe introdurre altre anomalie;

**Tecniche di debugging**

* **Tecnica Naive**: utilizzare delle print tattiche per stampare valori intermedi assunti dalle variabili. È una tecnica molto semplice da applicare, ma richiede di modificare il codice ed è poco flessibile.
* **Tecnica Naive avanzata**: si possono sfruttare funzionalità avanzate fornite dai linguaggi, come gli #ifdef di C. Purtroppo anche questa tecnica è molto debole per gli stessi motivi della precedente, seppur in gravità ridotta.
* **Dump di memoria**: scrivere su un file il contenuto corrente della memoria, per avere un’immagine statica dello stato. Questa tecnica è più interessante, ma è un’operazione complessa a causa della rappresentazione in memoria difficile da analizzare rispetto alla rappresentazione del linguaggio, e produce una quantità di dati molto elevata dove solo pochi sono realmente utili.
* **Debugging simbolico**: tramite dei tool specifici, si producono gli stati intermedi del programma in una rappresentazione simile a quella del linguaggio usato. Esistono anche debugger simbolici che forniscono funzionalità aggiuntive per visualizzare il comportamento del programma in maniera selettiva tramite breakpoints e watchpoints.
* **Debugging per prova**: molti debugger simbolici permettono di specificare una condizione per stabilire una watch condizionale, che verifica che la condizione rimanga vera per tutta l’esecuzione del programma.
* **Funzionalità aggiuntive**:
  + Modificare valori a runtime;
  + Modificare codice a runtime;
  + Visualizzare graficamente strutture dati come alberi;