Testarea sistemelor software

8.1. Intoducere

Un test constă în execuţia programului pentru un set de date de intrare convenabil ales, pentru a verifica dacă rezultatul obţinut este corect.

Un caz de test este un set de date de intrare împreună cu datele de ieşire pe care programul ar trebui să le producă.

Cazurile de test se aleg astfel încât să fie puse în evidenţă, dacă este posibil, situaţiile de funcţionare necorespunzătoare.

Testarea este activitatea de concepţie a cazurilor de test, de execuţie a testelor şi de evaluare a rezultatelor testelor, în diferite etape ale ciclului de viaţă al programelor.

Tehnicile de testare sunt mult mai costisitoare decât metodele statice (inspectările şi demonstrările de corectitudine), de aceea în prezent se apreciază că tehnicile statice pot fi folosite pentru a le completa pe cele dinamice, obţinîndu-se astfel o reducere a costului total al procesului de verificare şi validare.

Metodele traditionale de verificare/validare presupun realizarea verificării/validării prin inspectări şi teste.

Aceste activităţi ocupă circa 30÷50% din efortul total de dezvoltare, în funcţie de natura aplicaţiei.

Prin testare nu se poate demonstra corectitudinea unui program.

Aceasta deoarece în cele mai multe cazuri este practic imposibil să se testeze programul pentru toate seturile de date de intrare care pot conduce la execuţii diferite ale programului.

Testarea poate doar să demonstreze prezenţa erorilor într-un program.

Într-un sens, testarea este un proces distructiv, deoarece urmăreşte să determine o comportare a programului neintenţionată de proiectanţi sau implementatori.

Din acest punct de vedere testarea nu trebuie să fie făcută de persoanele care au contribuit la dezvoltarea programului.

Pe de altă parte conoaşterea structurii programului si a codului poate fi foarte utilă pentru alegerea unor date de test relevante.

Testarea în vederea verificării programului foloseşte date de test alese de participanţii la procesul de dezvoltare a programului.

Ea este efectuată la mai multe nivele: la nivel de unitate funcţională, de modul, de subsistem, de sistem.

Testarea în vederea validării programului, numită şi ***testare de acceptare,*** are drept scop stabilirea faptului că programul satisface cerinţele viitorilor utilizatori.

Ea se efectuează în mediul în care urmează să funcţioneze programul, deci folosindu-se *date reale*.

Prin testarea de acceptare pot fi descoperite şi erori, deci se efectuează şi o verificare a programului.

8.2. Testarea pe parcursul ciclului de viaţă al unui program.

8.2.1. Testele "unitare"

Testarea unui modul (o funcţie, o clasă, unitate) este realizată de programatorul care implementează modulul.

Toate celelalte teste sunt efectuate, în general, de persoane care nu au participat la dezvoltarea programului.

Scopul testarii unui modul este de a se stabili că modulul este o implementare corectă a specificaţiei sale (conforma cu specificatia sa). Specificatia poate fi neformala sau formala.

De exemplu:

- o specificaţie de pre şi post condiţii pentru o funcţie sau procedură;
- un invariant al clasei, care specifică mulţimea stărilor posibile ale fiecărui obiect din clasa respectivă, împreună cu specificaţii de pre şi post condiţii la nivelul funcţiilor membru;
- o specificaţie algebrică a clasei;
- o specificaţie Z/Z++ etc.

În cursul testarii unui modul, modulul este tratat ca o entitate independentă, care nu necesită prezenţa altor componente ale programului.

Testarea izolată a unui modul ridică două probleme:

- simularea modulelor apelate de cel testat;
- simularea modulelor apelante.

Modulele prin care se simulează modulele apelate de modulul testat se numesc module "ciot" (în engleză, "stub") . Un modul "ciot" are aceeaşi interfaţă cu modulul testat şi realizează în mod simplificat funcţia sa. De exemplu, dacă modulul testat apelează o funcţie de sortare a unui vector, cu antetul:

```
void sortare(int n, int *lista);
```

se poate folosi următoarea funcţie "ciot":

```
void sortare(int n, int *lista)
{ int i;
  printf(" \n Lista de sortat este:");
  for(i=0; i<n; i++)  printf("%d", lista[i]);
  // se citeste lista sortata, furnizata de testor
   for(i=0; i<n; i++)  scanf("%d", lista[i]);
 }
```

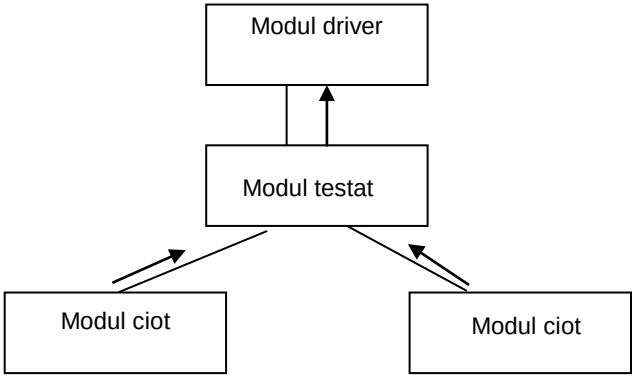

Fig.8.1. Structura programului executabil pentru testarea izolată a unui modul

Un modul "ciot" se poate reduce eventual la o tabelă de perechi de forma: "valori ale parametrilor de intrare la un apel - rezultatul prevăzut".

Cazurile de apel ale modulului testat de către celelalte module ale programului sunt simulate în cadrul unui "modul driver".

Modulul driver apelează modulul testat furnizndu-i ca date de intrare datele de test ale modulului.

Datele de test pot fi generate de modulul driver, pot fi preluate dintr-un fişier sau furnizate de testor într-o manieră interactivă.

8.2.2. Testele de integrare

Sunt dedicate verificării interacţiunilor dintre module, grupuri de module, subsisteme, până la nivel de sistem. Există mai multe metode de realizare a testelor de integrare.

Testele de integrare presupun, ca şi testele unitare, realizarea de module "ciot" şi module "driver".

Numărul de module "driver" şi de module "ciot" necesare în testele de integrare depinde de ordinea în care sunt testate modulele (metoda de integrare).

Testele de integrare necesită de asemenea instrumente de gestiune a versiunilor şi a configuraţiilor.

8.2.2.1. Metoda "big-bang"

Sunt integrate într-un program executabil toate modulele existente la un moment dat.

Modulele "driver" şi "ciot" necesare sunt de asemenea integrate.

Metoda este periculoasă căci toate erorile apar în acelaşi timp şi localizarea lor este dificilă.

8.2.2.2. Integrare progresiva

Metodele de *integrare progresivă* sunt mult mai eficiente.

În fiecare moment se adaugă ansamblului de module integrate numai un singur modul.

Astfel, erorile care apar la un test provin din modulul care a fost ultimul integrat.

8.2.2.2.1. Integrarea ascendentă (bottom-up)

Se începe prin testarea modulelor care nu apelează alte module, apoi se adaugă progresiv module care apelează numai modulele deja testate, până când este asamblat întregul sistem.

Metoda necesită implementarea câte unui modul "driver" pentru fiecare modul al programului (şi nici un modul "ciot").

Avantajele testării de jos în sus

Nu sunt necesare module "ciot". Modulele "driver" se implementează mult mai uşor decât modulele "ciot". Există chiar instrumente care produc automat module "driver".

Dezavantajele testării de jos în sus

1. Programul pe baza căruia se efectuează validarea cerinţelor este disponibil numai după testarea ultimului modul.

2. Corectarea erorilor descoperite pe parcursul integrării necesită repetarea procesului de proiectare, codificare şi testare a modulelor.

Principalele erori de proiectare sunt descoperite de abia la sfârşit, când sunt testate modulele principale ale programului. ceea ce, în general, conduce la reproiectare şi reimplementare.

8.2.2.2.2. Integrarea descendentă (top-down)

Se începe prin testarea modulului principal, apoi se testează programul obţinut prin integrarea modulului principal şi a modulelor direct apelate de el, şi aşa mai departe.

Metoda presupune implementarea unui singur modul "driver" (pentru modulul principal) şi a câte unui modul "ciot" pentru fiecare alt modul al programului.

Integrarea descendentă poate avea loc pe parcursul unei implementări descendente a programului.

Modulul principal este testat imediat ce a fost implementat, moment în care nu au fost încă implementate modulele apelate de el.

De aceea, pentru testare este necesar să se implementeze module "ciot".

În continuare, pe măsură ce se implementează modulele de pe nivelul ierarhic inferior, se trece la testarea lor folosind alte module “ciot”, ş.a.m.d.

În fiecare pas este înlocuit un singur modul "ciot" cu cel real.

Avantajele testării de sus în jos

1. Erorile de proiectare sunt descoperite timpuriu, la inceputul procesului de integrare, atunci când sunt testate modulele principale ale programului.

Aceste erori fiind corectate la început, se evită reproiectarea şi reimplementarea majorităţii componentelor de nivel mai coborât, aşa cum se întâmplă când erorile respective sunt descoperite la sfârşitul procesului de integrare.

2. Programul obtinut este mai fiabil căci principalele module sunt cel mai mult testate.

3. Prin testarea modulelor de nivel superior se poate considera că sistemul în ansamblul său există dintr-o fază timpurie a dezvoltării şi deci se poate exersa cu el în vederea validării cerinţelor; acest aspect este de asemenea, foarte important în privinţa costului dezvoltării sistemului.

Dezavantajele testării de sus în jos

1. Este necesar să se implementeze câte un modul "ciot" pentru fiecare modul al programului, cu excepţia modulului principal.

2. Este dificil de simulat prin module "ciot" componente complexe şi componente care conţin în interfaţă structuri de date.

3. În testarea componentelor de pe primele nivele, care de regulă nu afişează rezultate, este necesar să se introducă instrucţiuni de afişare, care apoi sunt extrase, ceea ce presupune o nouă testare a modulelor.

Aceste dezavantaje pot fi reduse aplicand tehnici hibride, de exemplu, folosind în locul unor module "ciot", direct modulele reale testate.

Integrarea nu trebuie sa fie strict descendenta.

De exemplu, experienta arata ca este foarte util sa se înceapă prin integrarea modulelor de interfaţă utilizator.

Aceasta permite continuarea integrării în condiţii mai bune de observare a comportării programului.

8.3. Testele de sistem

Acestea sunt teste ale sistemului de programe şi echipamente complet.

Sistemul este instalat şi apoi testat în mediul său real de funcţionare.

Sunt teste de conformitate cu specificaţia cerintelor de sistem (software) :

- *teste functionale*, prin care se verifica satisfacerea cerintelor functionale
- *teste prin care se verifica satisfacerea cerintelor ne-functionale :*
  - de performanţă,
  - de fiabilitate,
  - de securitate, etc.

Adesea, testele de sistem ocupă cel mai mult timp din întreaga perioadă de testare.

8.3.1. Testele de acceptare (validare)

Sunt teste de conformitate cu produsul solicitat, conform contractului cu clientul (->Specificatia cerintelor utilizatorilor).

Aceste teste sunt uneori conduse de client.

Pentru unele produse software, testarea de acceptare are loc în două etape:

1. **Testarea alfa**: se efectuează folosindu-se specificaţia cerinţelor utilizatorilor, până când cele două părţi cad de acord că programul este o reprezentare satisfăcătoare a cerinţelor.
2. **Testarea beta**: programul este distribuit unor utilizatori selecţionaţi, realizându-se astfel testarea lui în condiţii reale de utilizare.

8.3.2. Testele regresive

Se numesc astfel testele executate după corectarea erorilor, pentru a se verifica dacă în cursul corectării nu au fost introduse alte erori.

Aceste teste sunt efectuate de regulă în timpul mentenanţei sistemului.

Pentru uşurarea lor este necesar să se arhiveze toate testele efectuate în timpul dezvoltării programului, ceea ce permite, în plus, verificarea automată a rezultatelor testelor regresive

8.4. Determinarea cazurilor de test

Testarea unui modul, a unui subsistem sau chiar a întregului program presupune stabilirea unui set de cazuri de test.

Un *caz de test* cuprinde:

- un set de date de intrare;
- funcţia / funcţiile exersate prin datele respective;
- rezultatele (datele de ieşire) aşteptate;

În principiu (teoretic) testarea ar trebui să fie exhaustivă, adică să asigure exersarea tuturor căilor posibile din program.

Adesea o astfel de testare este imposibilă, de aceea trebuie să se opteze pentru anumite cazuri de test.

Prin acestea **trebuie să se verifice răspunsul programului atât la intrări valide cât şi la intrări nevalide.**

Sunt două metode de generare a cazurilor de test, care nu se exclud, de multe ori fiind folosite împreună.

Ambele metode pot sta la baza unor instrumente de generare automată a datelor (cazurilor) de test:

1. Cazurile de test se determină pe baza specificaţiei componentei testate, fără cunoaşterea realizării ei; acest tip de testare se numeşte **testare "cutie neagra"** ("black box").
2. Cazurile de test se determină prin analiza codului componentei testate. Acest tip de testare se mai numeşte şi testare "cutie transparentă", sau **testare structurală.**

8.4.1. Testarea "cutie neagră".

Cazurile de test trebuie să asigure următoarele verificări:

- reacţia componentei testate la intrări valide;
- reacţia componentei la intrări nevalide;
- existenţa efectelor laterale la execuţia componentei, adică a unor efecte care nu rezultă din specificaţie;
- performanţele componentei (dacă sunt specificate).

Deoarece în marea majoritate a cazurilor testarea nu poate fi efectuată pentru toate seturile de date de intrare (testare exhaustivă), în alegerea datelor de test plecând de la specificaţii se aplică unele metode fundamentate teoretic precum şi o serie de euristici.

Alegerea cazurilor de test folosind clasele de echivalenţă

Cazurile de test pot fi alese partiţionând atât datele de intrare cât şi cele de ieşire într-un număr finit de clase de echivalenţă.

Se grupează într-o aceeaşi clasă datele care, conform specificaţiei, conduc la o aceeaşi comportare a programului.

O dată stabilite clasele de echivalenţă ale datelor de intrare, se alege câte un eşantion (de date de test) din fiecare clasă.

De exemplu, dacă o dată de intrare trebuie să fie cuprinsă între 10 şi 19, atunci clasele de echivalenţă sunt:

1) valori < 10

2) valori între 10 şi 19

3) valori > 19

Se pot alege ca date de test: 9, 15, 20.

Experienţa arată că este util să se aleagă date de test care sunt la frontiera claselor de echivalenţă. Astfel, pentru exemplul de mai sus ar fi util să se testeze programul pentru valorile de intrare 10 şi 19.

Alte cazuri de test se aleg astfel încât la folosirea lor să se obţină eşantioane din clasele de echivalenţă ale datelor de ieşire (din interiorul si de la frontierele lor).

Exemplu:

Fie un program care trebuie să genereze între 3 şi 6 numere cuprinse între 1000 şi 2500. Atunci se vor alege intrări astfel încât ieşirea programului să fie:

- 3 numere egale cu 1000
- 3 numere egale cu 2500
- 6 numere egale cu 1000
- 6 numere egale cu 2500
- rezultat eronat: mai puţin de 3 numere sau mai mult de 6 numere cu valori în afara intervalului [1000..2500]
- între 3 şi 6 numere cuprinse între 1000 şi 2500

În alegerea datelor de test trebuie să se elimine redundanţele rezultate din considerarea atât a claselor de echivalenţă de intrare cât şi a celor de ieşire.

Unele programe tratează datele de intrare în mod secvenţial.

În aceste cazuri se pot detecta erori în program testându-l cu diverse combinaţii ale intrărilor în secvenţă.

Metoda de partiţionare în clase de echivalenţă nu ajută în alegerea unor astfel de combinaţii.

Numărul de combinaţii posibile este foarte mare chiar şi pentru programe mici.

De aceea nu pot fi efectuate teste pentru toate combinaţiile.

În aceste cazuri este esenţială experienţa celui care alcătuieşte setul de cazuri de test.

Testarea "cutie neagră" este favorizată de existenţa unei specificaţii formale a componentei testate.

Exemplu: Fie o funcţie de căutare a unui număr întreg într-un tablou de numere întregi, specificată astfel:

int cauta (int x[], int nrelem, int numar);

pre : nrelem > 0 **and exist** i **in** [0..nrelem-1] : x[i] = numar

post : x “[cauta(x,nrelem,numar)] = numar **and** x' = x’’

error : cauta(x,nrelem,numar) = -1 **and** x' = x’’

Intrările valide sunt cele care satisfac pre-condiţia.

Efectele laterale ale funcţiei "cauta" s-ar putea reflecta în modificarea unor variabile globale.

Pentru evidenţierea lor ar trebui ca la fiecare execuţie de test a funcţiei să se vizualizeze valorile variabilelor globale înainte şi după apelul funcţiei.

Aceste verificări pot fi limitate, înlocuindu-se cu inspectarea_codului funcţiei, din care rezultă evantualitatea modificării unor variabile globale.

Setul minim de date de test trebuie să verifice funcţia în următoarele situaţii:

1. tablou vid (nrelem=0)
2. tablou cu un singur element (nrelem=1)
   a). valoarea parametrului "numar" este în tablou
   b). valoarea parametrului "numar" nu este în tablou
3. tablou cu număr par de elemente ("nrelem" este un număr par)
   a). "numar" este primul în tablou
   b). "numar" este ultimul în tablou
   c). "numar" este într-o poziţie oarecare a tabloului
   d). "numar" nu este în tablou

4. tablou cu număr impar de elemente ("nrelem" este impar)
şi a,b,c,d ca la punctul 3.

Din specificaţie rezultă că funcţia nu trebuie să modifice tabloul; de aceea, apelul său în programul de test trebuie să fie precedat şi urmat de vizualizarea parametrului tablou.

Setul cazurilor de test ar putea fi:

1) intrări: orice tablou
nrelem=0
orice număr
ieşiri : valoarea funcţiei = -1
tabloul nemodificat

2) intrări: x[0] = 10
nrelem=1
număr = 10
ieşiri : valoarea funcţiei = 0
tabloul nemodificat: x[0]= 10

3) intrări: x[0]= 10
nrelem=1
număr = 15
ieşiri : valoarea funcţiei = -1
tabloul nemodificat: x[0] = 10

4) intrări: x[0] = 10, x[1] = 20
nrelem=2
număr = 10
ieşiri : valoarea funcţiei = 0
tabloul nemodificat: x[0] = 10, x[1] = 20

.....

În alegerea cazurilor de test s-au folosit următoarele euristici:

- Programele de căutare prezintă erori atunci când elementul căutat este primul sau ultimul in structura de date;
- De multe ori programatorii neglijează situaţiile în care colecţia prelucrată în program are un număr de elemente neobişnuit, de exemplu zero sau unu;

- Uneori, programele de căutare se comportă diferit în cazurile: număr de elemente din colecţie par, respectiv număr de elemente din colecţie impar; de aceea sunt testate ambele situaţii

Concluzii:

- Setul de cazuri de test a fost determinat numai pe baza specificaţiei componentei şi a unor euristici (este vorba de experienţa celui care efectuează testele), deci fără să se cunoască structura internă a componentei, care este tratată ca o “cutie neagră”.

Eventuala examinare a codului sursă nu urmăreşte analiza fluxului datelor şi a căilor de execuţie, ci doar identificarea variabilelor globale cu care componenta interacţionează.

- Clasele de echivalenţă se determină pe baza specificaţiei.
- Se aleg drept cazuri de test eşantioane din fiecare clasă de echivalenţă a datelor de intrare. Experienţa arată că cele mai utile date de test sunt acelea aflate la frontierele claselor de echivalenţă.
- Cazurile de test alese verifică componenta doar pentru un număr limitat de eşantioane din clasele de echivalenţă ale datelor de intrare; faptul că din testare a rezultat că ea funcţionează corect pentru un membru al unei clase nu este o garanţie că va funcţiona corect pentru orice membru al clasei.

- Se determină de asemenea clasele de echivalenţă ale datelor de ieşire şi se aleg pentru testare datele de intrare care conduc la date de ieşire aflate la frontierele acestor clase.
- Partiţionarea în clase de echivalenţă nu ajută la depistarea erorilor datorate secvenţierii datelor de intrare. In aceste cazuri este utilă experienţa programatorului.

Reprezentarea comportării programului printr-o diagramă de stări-tranziţii poate fi folositoare în determinarea secvenţelor de date de intrare de utlizat în testarea programului.

Testele “cutie neagră”, numite şi teste funcţionale sunt utile nu numai pentru testarea programului. Ele pot fi utilizate (ca teste statice) pentru verificarea unei specificaţii intermediare faţă de o specificaţie de nivel superior.

“Slăbiciunile” testelor funcţionale:

- Nu este suficient să se verifice că programul satisface corect toate funcţiile specificate; unele proprietăţi interne, nespecificate, ca de exemplu timpul de răspuns, nu sunt verificate;
- Programul este testat pentru “ceea ce trebuie să facă” conform specificaţiilor şi nu şi pentru ceea ce face în plus, de exemplu pentru ca implementarea să fie mai eficientă sau pentru a facilita reutilizarea;

- În absenţa unui standard în privinţa specificaţiilor formale, tehnicile de testare functională sunt eterogene

8.4.2. Testarea structurală

Acest tip de testare se bazează pe analiza fluxului controlului la nivelul componentei testate.

Principalul instrument folosit în analiză este *graful program* sau *graful de control.*

Acesta este un graf orientat, ale cărui noduri sunt de două tipuri:

- noduri care corespund “blocurilor de instrucţiuni indivizibile maximale” şi
- noduri care corespund instrucţiunilor de decizie.

Fiecare bloc are un singur punct de intrare şi un singur punct de ieşire.

Arcele reprezintă transferul controlului între blocurile/instrucţiunile componentei program.

Graful program are un nod unic de intrare şi un nod unic de ieşire.

Dacă există mai multe puncte de intrare sau mai multe puncte de ieşire, acestea trebuie să fie unite într-unul singur (care se adaugă suplimentar).

Nodul de intrare are gradul de intrare zero, iar cel de ieşire are gradul de ieşire zero.

Fie următorul text de program:

```
f1=fopen(….);
f2=fopen(….);
fscanf(f1, ”%f”, x);
fscanf(f1, “%f”, y);
z=0;
while(x>=y)
{ x-=y;
  z++;
}
fprintf(f2, “%d”, z);
fclose(f1); fclose(f2);
```

Textul este decupat în următoarele blocuri:

```
1. f1=fopen(….);
   f2=fopen(….);
   fscanf(f1, ”%f, x);
   fscanf(f1, “%f, y);
```

```
   z=0
2. while(x>=y)
3. x-=y;
    z++;
4. fprintf(f2, "%d", z);
    fclose(f1);
   fclose(f2);
```

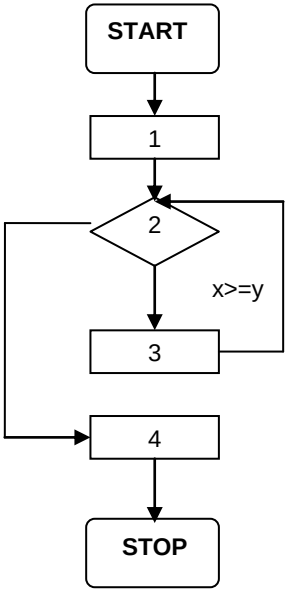

Fig.8.2. Graful de control

Graful de control este redat în Fig.8.2.

Se consideră căile care încep cu nodul de intrare şi se termină cu nodul de ieşire.

Testarea structurală constă în execuţia componentei testate pentru date de intrare alese astfel încât să fie parcurse unele dintre aceste căi.

Nu este necesar, şi în general este imposibil, să se execute toate căile de la intrare la ieşire ale unui program sau ale unei componente program.

Prezenţa ciclurilor conduce la un număr foarte mare (deseori infinit) de căi.

De asemenea, anumite căi sunt ne-executabile.

Testele structurale favorizează evidenţierea următoarelor tipuri de erori logice:

1. *Căi absente în graful program* - ca urmare a ignorării unor condiţii (de exemplu: test de împărţire la zero);
2. *Selectarea unei căi necorespunzătoare* - datorită exprimării incorecte (de multe ori incomplete) a unei condiţii;
3. *Acţiune necorespunzătoare sau absentă* (de exemplu: calculul unei valori cu o metodă incorectă, neatribuirea unei valori unei anumite variabile, apelul unei proceduri cu o listă de argumente incorectă, etc.).

Dintre acestea, cel mai simplu de depistat sunt erorile de tip 3. Pentru descoperirea lor este suficient să se asigure execuţia tuturor instrucţiunilor programului.

Alegerea căilor de testat are loc pe baza unor “criterii de acoperire” a grafului de control.

Dintre acestea cele mai cunoscute sunt:

Execuţia tuturor instrucţiunilor

Cazurile de test se aleg astfel încât să se asigure execuţia tuturor instrucţiunilor componentei testate.

Acesta este un criteriu de testare minimal, dar el nu este întotdeauna satisfăcător.

De exemplu, pentru testarea componentei cu graful din Fig.8.3, pe baza criteriului execuţiei tuturor instrucţiunilor, este suficient să se aleagă date de test care asigură execuţia conditiei si a instrucţiuilor I1 şi I2.

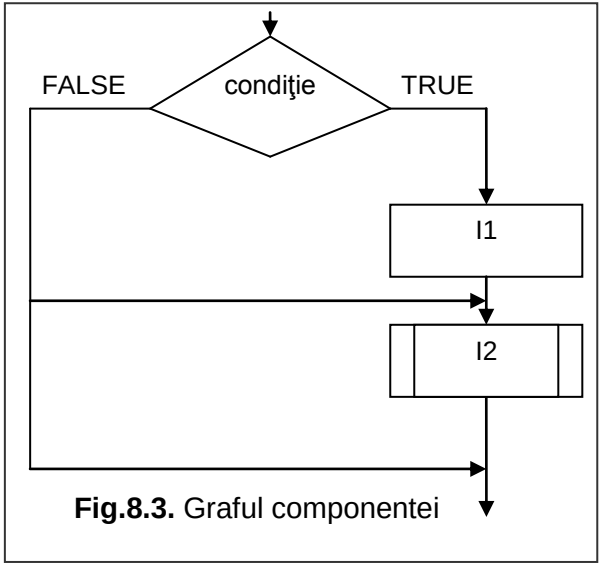

Fig.8.3. Graful componentei

Nu se testează cazurile în care condiţia are valoarea FALSE.

Transferul controlului pentru astfel de cazuri poate fi eronat.

Traversarea tuturor arcelor

Cazurile de test se aleg astfel încât să se asigure traversarea fiecărui arc al grafului program cel puţin pe o cale.

Pe baza acestui criteriu se va putea verifica dacă transferul controlului este corect pentru valoarea FALSE a condiţiei, în exemplul de mai sus.

În acelaşi timp, criteriul nu impune traversarea mai mult de o dată a unui ciclu. Anumite erori apar numai atunci când un ciclu este traversat de mai multe ori.

Traversarea tuturor căilor

Pe baza acestui criteriu ar trebui ca testele să asigure traversarea tuturor căilor componentei testate.

Pentru majoritatea programelor nu se poate aplica acest criteriu, deoarece numărul de căi de execuţie este infinit.

Criteriul poate fi restricţionat, de exemplu asigurând traversarea tuturor căilor cu o lungime mai mică sau egală cu o constantă dată.

Problema alegerii cazurilor de test constă din trei subprobleme distincte:

- selectarea căilor de testat;
- alegerea datelor de test pentru execuţia fiecărei căi selectate;
- determinarea rezultatelor care trebuie să se obţină la fiecare test.

În continuare prezentăm o metodă de alegere a datelor de test pe baza criteriului traversării tuturor arcelor grafului de control. Metoda presupune parcurgerea următoarelor etape:

1. Se determină setul căilor de testat, C, astfel încât:
   a) $\forall$ c $\in$ C este o cale de la nodul de intrare la nodul de iesire;
   b) fiecare arc din graful de control este cuprins într-o cale c $\in$C
   c) Σ lung (c) = minima
      c $\in$ C
2. Se determină predicatul fiecărei căi, c $\in$ C, $R_c(I)$, unde I = $(i_1, i_2, \ldots, i_n)$ este vectorul variabilelor de intrare;
3. Pentru fiecare $R_c(I)$, c $\in$ C, se determină un set de atribuiri X_c pentru variabilele de intrare care satisfac predicatul căii:

$R_c(X_c)$ = true

Atunci setul datelor de test este:

DT = { X_c | c $\in$ C, $X_c \in D_I$, $R_c(X_c)$ = true },

n

unde $D_I = \sum_{k=1}^{n} D_{ik}$, iar D_{ik} este domeniul de valori al variabilei de intrare $i_{k.}$

1. Pentru fiecare $X_c \in$ DT, se determina valorile corecte ale variabilelor de ieşire

Exemplu

Fie urmatorul graf program :

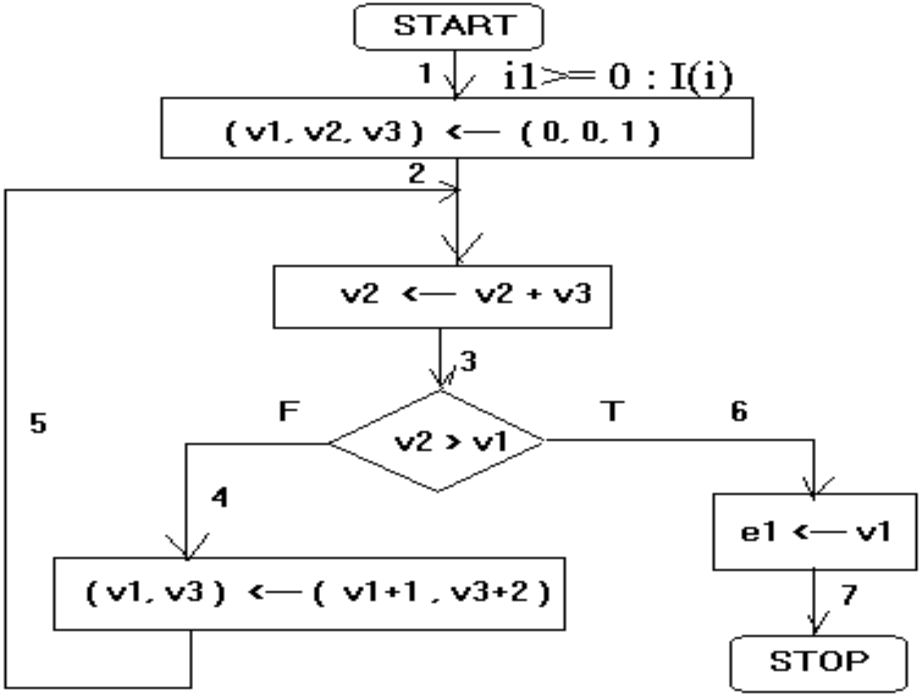

Fig. 8.4. Graf program

Obs: testul v2>v1 trebuie inlocuit cu v2>i1.

1) Alegerea cailor :

Se poate alege setul de căi C:

$$C = \{c_1, c_2\},$$

unde $c_1 = \{1,2,3,4,5,3,6,7\}$

$c_2 = \{1,2,3,6,7\}$

În afara condiţiilor menţionate s-au mai avut în vedere următoarele criterii:

- alegerea unei căi care să nu conţina ciclul
- alegerea unei căi care să conţina ciclul, parcurgându-l o singura dată, pentru ca lungimea căii să fie minimă.

2) **Determinarea predicatelor de cale**

O metodă de determinare a predicatului unei căi constă în concatenarea condiţiilor de ramificare de pe calea respectivă pe măsura ce sunt întâlnite atunci când calea este parcursa de la nodul de

ieşire pană la cel de intrare. Totodată, la întâlnirea unei atribuiri, y ← E, dacă y face parte din expresia curentă a predicatului, se substituie y cu E.

In unele cazuri, predicatul obţinut printr-o singură parcurgere a unui ciclu este fals pentru orice atribuire a variabilelor de intrare. Un astfel de caz corespunde unei căi neexecutabile. In consecintă se va alege un predicat în care ciclul este parcurs de două sau de mai multe ori.

Construirea predicatelor de cale

Rc_1 = v2 > i1

↓ 3 se substituie v_2 cu (v_2 + v_3)

Rc_1 = (v2+ v3) > i1

↓ 5

Rc_1 = (v2+ (v3+2)) > i1

↓ 4

Rc_1 = (v2+ v3+2) > i1 Λ ~(v2 > i1)

↓ 3

Rc_1 = ((v2+ v3)+v3+2) > i1 Λ ~((v2+ v3) > i1)

↓ 2

Rc_1 = (4 > i1) Λ ~ (1 > i1)

↓ 1

Rc_1 = (4 > i1) Λ (i1 >= 1) Λ (i1 >= 0)

<=>

Rc_1 = (1 <= i1 < 4) . Calea c1 este executată pentru orice valoare a lui i1 care satisface acest predicat.

Stabilirea rezultatelor execuţiei fiecărei căi pentru fiecare set de date de test ales, necesită cunoaşterea transformării realizate de componenta analizată asupra datelor de intrare. Aceasta rezultă din specificaţia componentei. In cazul de faţă trebuie să se stabilească ce valoare trebuie sa aibă iesirea e1 pentru fiecare valoare aleasă pentru i1. Programul reprezentat prin graful de control analizat calculează numarul maxim de termeni ai sumei 1+3+5+…+(2j+1), ∀ j >= 0, care pot fi adunati a.î. suma lor să fie mai mica decât i1. Astfel se poate verifica că pentru orice 1<= i1 <=4, e1 = 1, căci dacă s-ar aduna doi termeni, 1+3 = 4, 4>i1.

Cazul de test al căii c1 poate fi: i1=2, e1 = 1.

Calea c2 = {1,2,3,6,7}

Rc_2 = v2 > i1
↓ 3
Rc_2 = (v2+ v3) > i1

$$\downarrow 2$$
$$Rc_2 = 0 + 1 > i1$$
$$\downarrow 1$$
$$Rc_2 = (i1 < 1) \ \Lambda \ (i1 >= 0)$$

<=>

$$Rc_2 = (0 <= i1 < 1) \quad => \quad Rc_2 = (i1 = 0)$$

Cazul de test: i1 = 0, e1 = 0.

“Slabiciunile” testelor structurale sunt:

- testele selectionate depind numai de structura programului; ele trebuie recalculate la fiecare modificare; nu pot fi reutilizate eficient de la o versiune la alta;
- testele selecţionate acoperă cazurile legate de ceea ce “face” componenta testată şi nu de ceea ce “trebuie să facă”; astfel, dacă un caz particular a fost uitat în faza de implementare, testul structural nu releva aceasta deficienţă; el trebuie deci completat cu testul funcţional. Mai exact trebuie să se înceapă cu testarea funcţională care se completeză cu testarea structurală.

Testele statistice

Testele statistice se efectueaza in timpul testarii de sistem.

Se numesc astfel testele în care datele de test se aleg aleator, după o lege de probabilitate care poate fi:

- uniformă pe domeniul datelor de intrare al programului testat- aceste teste se mai numesc **teste statistice uniforme**;
- similară cu distribuţia datelor de intrare estimate pentru exploatarea programului – aceste teste se mai numesc **teste statistice operaţionale.**

Rezultatele experimentale ale testelor statistice uniforme sunt inegale.

Pentru unele programe s-au dovedit foarte eficiente, ele conducând la descoperirea unor erori însemnate.

Pentru altele s-au dovedit ineficiente.

O posibilă explicaţie ar consta în faptul că ele asigură o bună acoperire în cazul programelor pentru care domeniile de intrare ale căilor de execuţie au o probabilitate comparabilă.

Ele nu acoperă căile care corespund tratării excepţiilor.

De aceea trebuie completate cu teste folosind date de intrare în afara domeniului intrărilor.

Testele statistice operaţionale sunt în general **teste de fiabilitate**.

Prin ele nu se urmăreşte în general descoperirea de erori ci mai ales comportarea programului în timp.

Căderile observate în timpul acestor teste permit estimarea măsurilor de fiabilitate, cum ar fi MTBF (Mean Time Between Failures).