Cuprins

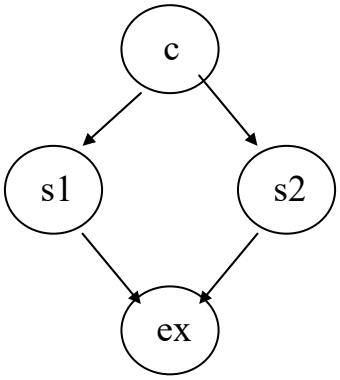
2. Testare structurală

- datele de test sunt generate pe baza implementării (programului), fără a lua în considerare specificația (cerințele) programului
- pentru a utiliza metode structurale de testare programul poate fi reprezentat sub forma unui graf orientat
- datele de test sunt alese astfel încât să parcurga toate elementele (instrucțiune, ramură sau cale) grafului măcar o singură dată. În funcție de tipul de element ales, vor fi definite diferite măsuri de acoperire a grafului: acoperire la nivel de instrucțiune, acoperire la nivel de ramură sau acoperire la nivel de cale

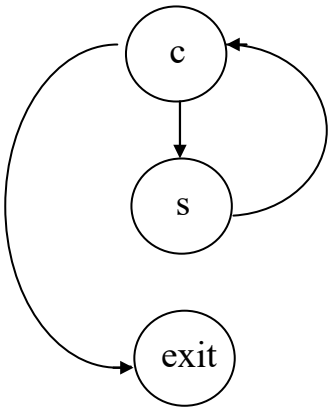
Transformarea programului într-un graf orientat

- Pentru o secvență de instrucțiuni se introduce un nod

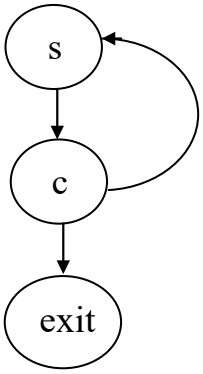
- if c then s1 else s2



- while c do s

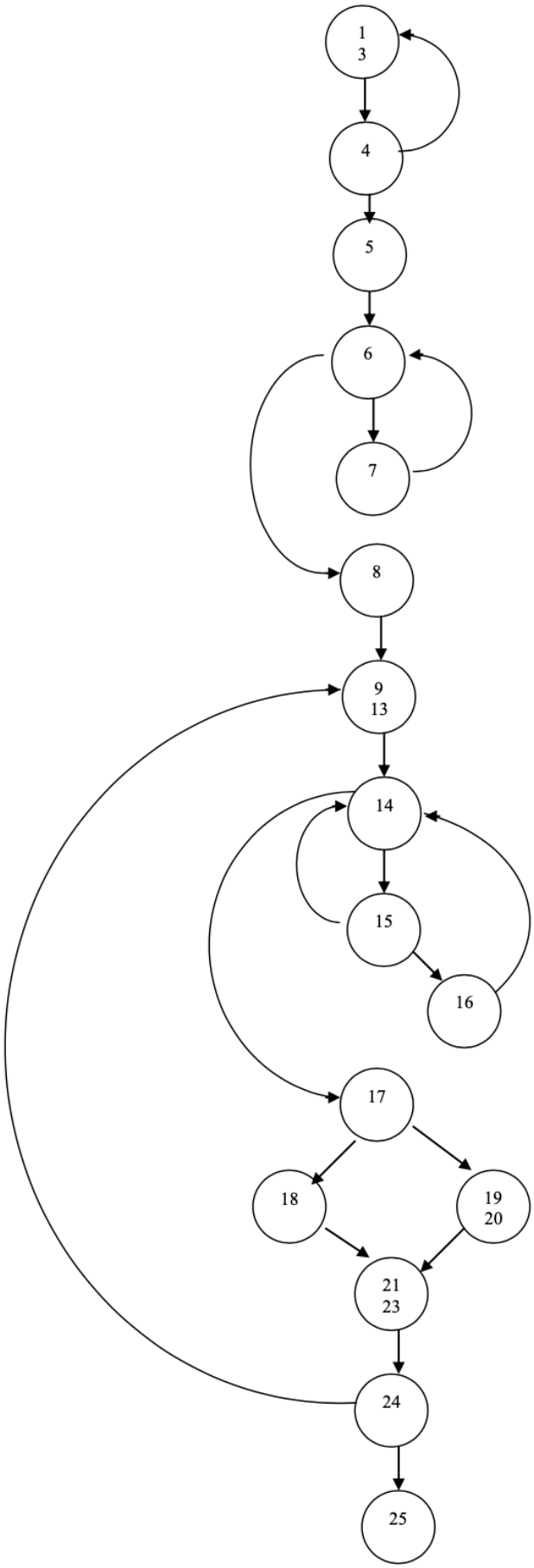


repeat s until c



```
// extras cod clasa de testat implementată în Java (obsolete)
    public class MyClass {
        public static void main(String[] arg) {

            KeyboardInput in = new KeyboardInput();
            char response,c,n1;
            boolean found;
            int n,i;
            char[]a = new char[20];
1           do {
2               System.out.println("Input an integer between 1 and
20: ");
3               n = in.readInteger();
4           } while (n < 1 || n > 20);
5           System.out.println("input "+n+" character(s)");
6           for(i=0;i<n;i++)
7               a[i] = in.readCharacter();
8           n1 = in.readCharacter();
9           do {
10              System.out.println("Input character to search for:
");
11              c = in.readCharacter();
12              n1 = in.readCharacter();
13              found = false;
14              for(i=0;!found && i<n;i++)
15                  if(a[i] == c)
16                      found = true;
17              if(found)
18                  System.out.println("character "+ c +" appears
at position "+i);
19              else
20                  System.out.println("character "+ c +" does not
appear in string");
21              System.out.println("Search for another
character?[y/n]: ");
22              response = in.readCharacter();
23              n1 = in.readCharacter();
24          } while((response == 'y')||(response == 'Y'));
25      }
    }
```


1
3
4
5
6
7
8
9
13
14
15
16
17
18
19
20
21
23
24
25

Pe baza grafului se pot defini diverse acoperiri:

- *Acoperire la nivel de instrucțiune*: fiecare instrucțiune (nod al grafului) este parcursă măcar o dată
- *Acoperire la nivel de ramură*: fiecare ramură a grafului este parcursă măcar o dată
- *Acoperire la nivel de condiție*: fiecare condiție individuală dintr-o decizie să ia atât valoarea adevărat cât și valoarea fals
- *Acoperire la nivel de cale*: fiecare cale din graf este parcursă măcar o dată

(a) Statement coverage (acoperire la nivel de instrucțiune)

Pentru a obține o acoperire la nivel de instrucțiune, trebuie să ne concentram asupra acelor instrucțiuni care sunt controlate de condiții (acestea corespund ramificațiilor din graf)

Intrari				**Rezultat afișat**	**Instrucțiuni parcurse**
N	**x**	**C**	**s**		
1	a	a	y		1..3, 4, 5, 6 7, 6, 8, 9..13 14, 15, 16, 14, 17, 18, 21..23 24, 9..13
		b	n		14, 15, 14, 17, 19..20, 21..23 24, 25

```
// extras cod teste implementate în Java cu librăria JUnit
@Test
public void statementCoverage() {
//…
}
```

Testarea la nivel de instrucțiune este privita de obicei ca nivelul minim de acoperire pe care îl poate atinge testarea structurală. Acest lucru se datorează sentimentului că este absurd să dai în funcționare un software fără a executa măcar o dată fiecare instrucțiune.

Totuși, destul de frecvent, această acoperire nu poate fi obținută, din următoarele motive:

- Existenta unei porțiuni izolate de cod, care nu poate fi niciodată atinsă. Această situație indică o eroare de design și respectiva porțiune de cod trebuie înlăturată.
- Existența unor porțiuni de cod sau subrutine care nu se pot executa decat în situații speciale (subrutine de eroare, a căror execuție poate fi dificilă sau chiar periculoasă). În astfel de situații, acoperirea acestor instrucțiuni poate fi inlocuită de o inspecție riguroasă a codului

Avantaje:

- Realizează execuția măcar o singură dată a fiecarei instrucțiuni
- În general ușor de realizat

Slăbiciuni:

Nu asigură o acoperire suficientă, mai ales în ceea ce privește condițiile:

- Nu testează fiecare condiție în parte în cazul condițiilor compuse (de exemplu, pentru a se atinge o acoperire la nivel de instrucțiune în programul folosit ca exemplu, nu este necesară introducerea unei valori mai mici ca 1 pentru *n*)
- Nu testează fiecare ramură
- Probleme suplimentare apar în cazul instrucțiunilor *if* a căror clauza *else* lipsește. În acest caz, testarea la nivel de instrucțiune va forța execuția ramurii corespunzătoare valorii adevărat, dar, deoarece nu există clauza *else*, nu va fi necesară și execuția celeilalte ramuri. Metoda poate fi extinsă pentru a rezolva această problemă.

(b) Decision coverage (acoperire la nivel de decizie) sau branch coverage (acoperire la nivel de ramură)

- Este o extindere naturală a metodei precedente.
- Genereaza date de test care testează cazurile când fiecare decizie este adevărată sau falsă.

	Decizii
(1)	while (n<1||n>20)
(2)	for (i=0; i<n; i++)
(3)	for(i=0; !found && i<n; i++)
(4)	if(a[i]==c)
(5)	if(found)
(6)	while ((response=='y') ||(response=='Y'))

Intrari				**Rezultat afișat**	**Decizii acoperite**
N	**x**	**C**	**s**		
25				Cere introducerea unui întreg între 1 și 20	
1	a	A	y	Afișează poziția 1; se cere introducerea unui nou caracter	
		B	n	Caracterul nu apare	

```
// extras cod teste implementate în Java cu librăria JUnit
@Test
public void branchCoverage() {
//…
}
```

Avantaje:

- Este privită ca etapa superioară a testarii la nivel de instrucțiune; testează toate ramurile (inclusiv ramurile nule ale instrucțiunilor *if/else*)

Dezavantaje:

- Nu testeaza condițiile individuale ale fiecărei decizii

(c) Condition coverage (acoperire la nivel de condiție)

- Genereaza date de test astfel încat fiecare condiție individuală dintr-o decizie să ia atât valoarea adevărat cât și valoarea fals (dacă acest lucru este posibil).

- De exemplu, dacă o decizie ia forma c1 || c2 sau c1 && c2, atunci acoperirea la nivel de condiție se obține astfel încât fiecare dintre condițiile individuale c1 și c2 sa ia atat valoarea adevărat cat și valoarea fals

Nota: decizie inseamna orice ramificare in graf, chiar atunci cand ea nu apare explicit in program. De exemplu, pentru constructia for i := 1 to n din Pascal conditia implicita este i <= n

Decizii	Conditii individuale
while (n<1||n>20)	n < 1, n > 20
for (i=0; i<n; i++)	i < n
for(i=0; !found && i<n; i++)	found, i< n
if(a[i]==c)	a[i] = c
if(found)	Found
while ((response=='y') ||(response=='Y'))	(response=='y') (response=='Y')

Intrari				**Rezultat afișat**	**Conditii individuale acoperite**
n	**x**	**C**	**s**		
0				Cere introducerea unui întreg între 1 și 20	
25				Cere introducerea unui întreg între 1 și 20	
1	a	A	y	Afișează pozitia 1; se cere introducerea unui nou caracter	
		B	Y	Caracterul nu apare; se cere introducerea unui nou caracter	

```
// extras cod teste implementate în Java cu librăria JUnit
@Test
public void conditionCoverage () {
//…
}
```

Avantaje

- Se concentreaza asupra condițiilor individuale

Dezavantaj

- Poate să nu realizeze o acoperire la nivel de ramură. De exemplu, datele de mai sus nu realizează ieșirea din bucla *while* ((response=='y') || (response=='Y')) (condiția globală este în ambele cazuri adevărată). Pentru a rezolva această slăbiciune se poate folosi testarea la nivel de decizie / condiție.

(d) Condition/decision coverage (acoperire la nivel de condiție/decizie)

- Generează date de test astfel încât fiecare condiție individuală dintr-o decizie să ia atat valoarea adevărat cât și valoarea fals (dacă acest lucru este posibil) și fiecare decizie să ia atât valoarea adevărat cât și valoarea fals.

Intrări				**Rezultat afișat**	**Condiții/Decizii individuale acoperite**
n	**x**	**C**	**s**		
0				Cere introducerea unui întreg între 1 și 20	
25				Cere introducerea unui întreg între 1 și 20	
1	a	A	y	Afișează poziția 1; se cere introducerea unui nou caracter	
		B	Y	Caracterul nu apare; se cere introducerea unui nou caracter	
		B	n	Caracterul nu apare	

(e) Multiple condition coverage (acoperire la nivel de condiții multiple)

Generează date de test astfel încât să parcurga toate combinațiile posibile de adevărat și fals ale conditiilor individuale.

(f) Modified condition/decision (MC/DC) coverage

- Condition/Decision coverage poate să nu testeze unele condiții individuale (care sunt "mascate" de alte condiții)
- Multiple condition coverage poate genera o explozie combinatorică (pentru n condiții pot fi necesare 2^n teste)

Solutie: O formă modificată a condition/decision coverage.
Un set de teste satisface MC/DC coverage atunci când:

- Fiecare condiție individuală dintr-o decizie ia atât valoare True cât și valoare False
- Fiecare decizie ia atât valoare True cât și valoare False
- Fiecare condiție individuală influențează în mod independent decizia din care face parte

Avantaje:

- Acoperire mai puternică decât acoperirea condiție/decizie simplă, testând și influența condițiilor individuale asupra deciziilor
- Produce teste mai puține – depinde liniar de numărul de condiții

AND

Test	**C1**	**C2**	**C1 ∧ C2**
t1	True	True	True
t2	True	False	False
t3	False	True	False

t1 și t3 acoperă C1

t1 și t2 acoperă C2

OR

Test	**C1**	**C2**	**C1 ∨ C2**
t1	False	True	True
t2	True	False	True
t3	False	False	False

t2 și t3 acoperă C1

t1 și t3 acoperă C2

XOR

Test	**C1**	**C2**	**C1 xor C2**
t1	True	True	False
t2	True	False	True
t3	False	False	False

t2 și t3 acoperă C1

t1 și t2 acoperă C2

Exemplu: C = C1 ∧ C2 ∨ C3

Test	**C1**	**C2**	**C3**	**C**	**Efect demonstrat pentru**
1	True	True	False	True	C1
2	False	True	False	False	
3	True	True	False	True	C2
4	True	False	False	False	
5	True	False	True	True	C3
6	True	False	False	False	

Set de teste minimal

Test	**C1**	**C2**	**C3**	**C**
t1	True	True	False	True
t2	False	True	False	False
t3	True	False	False	False
t4	True	False	True	True

t1 și t2 testeaza C1

t1 și t3 testeaza C2

t3 și t4 testeaza C3

(g) Testarea circuitelor independente

Acesta este o modalitate de a identifica limita superioară pentru numărul de căi necesare pentru obținerea unei acoperiri la nivel de ramură.

Se bazează pe formula lui McCabe pentru Complexitate Ciclomatică:

Dat fiind un graf complet conectat *G* cu *e* arce și *n* noduri, atunci numărul de circuite linear independente este dat de:

$$V(G) = e - n + 1$$

La https://en.wikipedia.org/wiki/Cyclomatic_complexity puteți vedea mai multe formule.

i. V(G) = e - n + 2p, unde

e = numărul de muchii ale graficului.

n = numărul de noduri ale graficului.

p = numărul de componente conectate.

Formulă alternativă, unde fiecare punct de ieșire este conectat înapoi la punctul de intrare (adică graful e complet conectat):

ii. V(G) = e − n + p.

Pentru un singur program (sau subrutină sau metodă), p este întotdeauna egal cu 1.

Deci, o formulă mai simplă pentru o singură subrutină este:

iii. V(G) = e − n + 2.

Terminologie:

- Graf complet conectat: există o cale între oricare 2 noduri
  (există un arc între nodul de stop și cel de start)
- Circuit = cale care începe și se termină în același nod
- Circuite linear independente: nici unul nu poate fi obținut ca o combinație a celorlalte

În exemplul nostru, adăugând un arc de la 25 la 1, avem:

n = 16, e = 22, V(G) = 7

Circuite independente:

a) 1..3, 4, 5, 6, 8, 9..13, 14, 17, 18, 21..23, 24, 25, 1..3
b) 1..3, 4, 5, 6, 8, 9..13, 14, 17, 19..20, 21..23, 24, 25, 1..3
c) 1..3, 4, 1..3
d) 6, 7, 6
e) 14, 15, 14
f) 14, 15, 16, 14
g) 9..13, 14, 17, 18, 21..23, 24, 25, 1

```
// extras cod teste implementate în Java cu librăria JUnit
@Test
public void circuitCoverage() {
//…
}
```

Acesta poartă numele de set de bază.

Orice cale se poate forma ca o combinație din acest set de bază.

De exemplu

1..3, 4, 1..3, 4, 1..3, 4, 5, 6, 7, 6, 8, 9..13, 14, 15, 16, 14 17, 18,

21..23, 24, 25, 1..3

este o combinație din: a, c (de 2 ori), d, f

Avantaje:

Setul de bază poate fi generat automat și poate fi folosit pentru a realiza o acoperire la nivel de ramură.

Dezavantaje:

Setul de bază nu este unic, iar uneori complexitatea acestuia poate fi redusă.

(h) Testare la nivel de cale

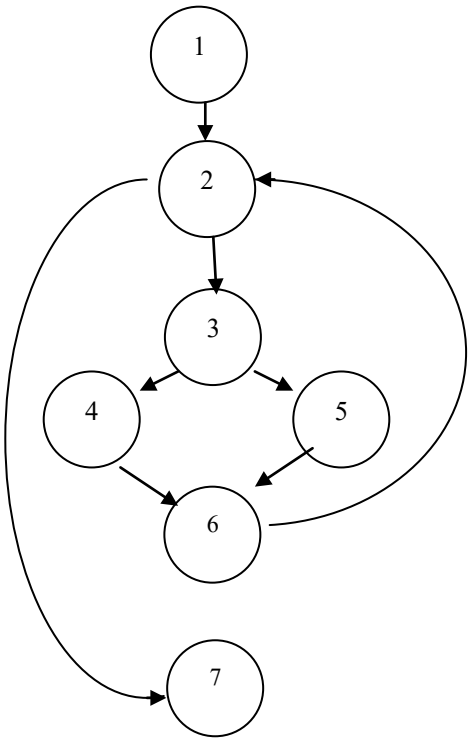
- Generează date pentru executarea fiecarei căi măcar o singura dată
- Problemă: în majoritatea situațiilor există un număr infinit (foarte mare) de căi
- Soluție: Împărțirea căilor în clase de echivalență.

De exemplu: 2 clase pot fi considerate echivalente dacă diferă doar prin numărul de ori de care sunt traversate de același circuit; determină 2 clase de echivalență: traversate de 0 ori și n ori, n > 1.

Aplicație:

Dacă un program este structurat, atunci, folosindu-se o tehnica descrisă de Paige și Holthouse (1977), acesta poate fi caracterizat de o expresie regulata formata din nodurile grafului.

```
1       …
2       while c1 do begin
3       if c2 then
4                       …
5               else …
6       end
7       …
```



Expresia regulată obținută este: 1.2.(3.(4+5).6.2)*.7

Notă: operatorul star (Kleene): R* = R de zero sau mai multe ori la rând (să spunem n)

Pentru n = 0 și n = 1 avem:

1.2.(3.(4+5).6.2 + null).7

de unde rezultă căile:

1.2.7

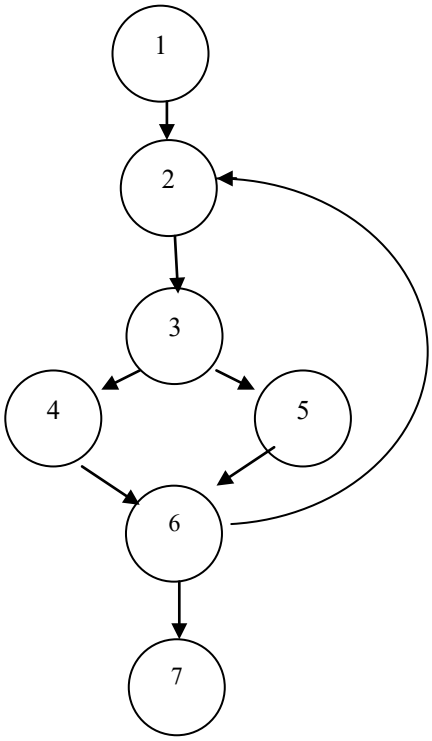
1.2.3.4.6.2.7

1.2.3.5.6.2

Numărul de căi se obține înlocuind în expresia regulată fiecare nod cu 1 (inclusiv pt. null), iar operația de concatenare devine înmulțire:

1.1.(1.(1+1).1.1 + 1).1 = 3 căi

```
1       …
2       repeat s
3               if c2 then
4                       …
5               else …
6       until c1
7       …
```



Expresia regulată: 1.2.3.(4+5).6.(2.3.(4+5).6)*.7

Pentru n = 0 și n = 1 avem:

1.2.3.(4+5).6.(2.3.(4+5).6 + null).7

de unde rezultă căile:

1.2.3.4.6.7

1.2.3.5.6.7

1.2.3.4.6.2.3.4.6.7

1.2.3.4.6.2.3.5.6.7

1.2.3.5.6.2.3.4.6.7

1.2.3.5.6.2.3.5.6.7

Numărul de căi se obține înlocuind în expresia regulată fiecare nod cu 1 (inclusiv pt. null), iar operația de concatenare devine înmulțire:

1.1.1.(1+1).1.(1.1.(1+1).1+1).1 = 2.3 = 6 căi

Pentru exemplul nostru:

1.4.(1.4)*.5.6.(7.6)*.8.9.14.(15.(null+16).14)*17.(18+19).21.24.(
9.14.(15.(null+16).14)*17.(18+19).21.24)*.25

număr de căi:
2.2.3.2.(3.2+1) = 168

Observație: multe căi, din care o mare parte sunt nefezabile (de exemplu, ieșirea de la început din cele două instrucțiuni *for*).

Avantaje:

- Sunt selectate căi pe care alte metode de testare structurală (inclusiv testare la nivel de ramură) nu le ating.

Dezavantaje:

- Multe căi, din care o parte pot fi nefezabile
- Nu exersează condițiile individuale ale deciziilor
- Tehnica descrisă pentru generarea căilor nu este aplicabilă direct programelor nestructurate.