TESTARE FUNCTIONALA – Datele de test sunt generate pe baza cerintelor. structura nu are niciun rol.
- Tipul de specificație ideal pentru testarea funcțională este alcătuit din pre-condiții și postcondiții.
- Majoritatea metodelor funcționale se bazează pe o partiționare a datelor de intrare astfel încât datele aparținând unei aceeași partiții vor avea proprietăți similare (identice) în raport cu comportamentul specificat.

a) Partitionare de echivalenta – partitionam domeniul problemei (input) in partitii/clase de echivalenta -> Datele dintr-o clasa sunt tratate identic. Toate valorile dintr-o clasa au specificat acelasi comportament, vor fi procesate la fel. Domeniul de iesire va fi tratat la fel. Clasele nu trb sa se suprapuna. Daca se suprapun trb descompuse. Pot fi alese date invalide (in afara claselor, nu sunt procesate de nici o clasa). Alegerea e arbitrara. **Avantaje:** Reduce drastic nr de date de test doar pe baza specificatiei. | Potrivita pt aplicatii de tipul procesarii datelor, in care intrarile si iesirile sunt usor de identificat si iau valori distincte. **Dezavantaje:** Modul de definire al claselor nu este evident. | In unele cazuri, desi specificatia ar putea sugera ca un grup de valori sunt procesate identic, acest lucru nu este adevarat. Metodele functionale si cele structurale sunt aplicate impreuna. | Mai putin aplicabile pt situatii cand intrarile si iesirile sunt simple, dar procesarea e complexa.

EXEMPLU: 1) Domeniul de intrari: un intreg pozitiv n, cu valori intre 1 si 20 -> N_1=1..20, N_2={n|n<1}, N_3={n|n>20}
2) Domeniul de iesire: C_111 = { (n, x, c, s) | n \in N_1, |x| = n, c \in C_1(x), s \in S_1} **C_112** = { (n, x, c, s) | n \in N_1, |x| = n, c \in C_1(x), s \in S_2} **C_121** = { (n, x, c, s) | n \in N_1, |x| = n, c \in C_2(x), s \in S_1} **C_122** = { (n, x, c, s) | n \in N_1, |x| = n, c \in C_2(x), s \in S_2} **C_2** = { (n, x, c, s) | n \in N_2} **C_3** = { (n, x, c, s) | n \in N_3}

Intrări				Rezultat afișat (expected)
n	x	c	s	
0				Cere introducerea unui întreg între 1 și 20
25				Cere introducerea unui întreg între 1 și 20
3	abc	a	y	Afișează poziția 1; se cere introducerea unui nou caracter
		a	n	Afișează poziția 1
3	abc	d	y	Caracterul nu apare; se cere introducerea unui nou caracter
		d	n	Caracterul nu apare

b) Analiza valori frontiera(Boundary) – folosita impreuna cu echivalenta, se concentreaza pe examinarea valorilor de frontiera ale claselor, care de obicei sunt o sursa importanta de erori.
Avantaje/Dezavantaje: Aceleași ca la metoda anterioară. | În plus, această metodă adaugă informații suplimentare pentru generarea setului de date de test și se concentrează asupra unei arii (frontierele) unde de regulă apar multe erori.

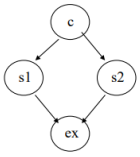
EXEMPLU: Deci se vor testa următoarele valori: • N_1 : 1, 20 • N_2 : 0 • N_3 : 21 • C_1: = c_11 se află pe prima poziție în x, c_12 se află pe ultima poziție în x • Pentru restul claselor se ia câte o valoare (arbitrară)

c) Partiționarea în categorii (category-partitioniong) – Această metodă se bazează pe cele două anterioare. Ea caută să genereze date de test care "acoperă" funcționalitatea sistemului și maximizeaza posibilitatea de găsire a erorilor.
Cuprinde următorii pasi: 1. Descompune specificația funcțională în unitati (programe, functii, etc.) care pot fi testate separat. **2.** Pentru fiecare unitate, identifică parametrii și condițiile de mediu (ex. starea sistemului la momentul execuției) de care depinde comportamentul acesteia. **3.** Găseste categoriile (proprietăți sau caracteristici importante) fiecărui parametru sau condiții de mediu. **4.** Partiționează fiecare categorie în alternative. O alternativă reprezintă o mulțime de valori similare pentru o categorie. **5.** Scrie specificația de testare. Aceasta constă în lista categoriilor și lista alternativelor pentru fiecare categorie. **6.** Creează cazuri de testare prin alegerea unei combinații de alternative din specificația de testare (fiecare categorie contribuie cu zero sau o alternativă). **7.** Creează date de test alegând o singură valoare pentru fiecare alternativă
Avantaje și dezavantaje: Pașii de inceput (identificarea parametrilor și a condițiilor de mediu precum și a categoriilor) nu sunt bine definiți și se bazează pe experienta celui care face testarea. Pe de altă parte, odata ce acesti pași au fost trecuti, aplicarea metodei este foarte clară. | Este mai clar definită decat metodele funcționale anterioare și poate produce date de testare mai cuprinzatoare, care testează funcționalitati suplimentare; pe de altă parte, datorită exploziei combinatorice, pot rezulta date de test de foarte mare dimensiune.
• **n: 1)** {n | n < 0}; **2)** 0; **3)** 1 [ok, lungime1]; **4)** 2..19 [ok, lungime_medie]; **5)** 20 [ok, lungime20]; **6)** 21; **7)** {n | n > 21};
• **x 1)** {x | |x| = 1}. [if ok and lungime1]; **2)** {x | 1 < |x| < 20}. [if ok and lungime_medie]; **3)** {x | |x| = 20} [if ok and lungime20];
• **c 1)** { c | c se afla pe prima poziție în x} [if ok]; **2)** { c | c se afla în interiorul lui x} [if ok and not lungime1]; **3)** { c | c se afla pe ultima poziție în x} [if ok and not lungime1]; **4)** { c | c nu se afla în x} [if ok];
• **s 1)** y [if ok]; **2)** n [if ok]

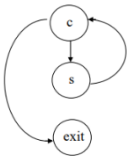
TESTARE STRUCTURALA - datele de test sunt generate pe baza implementării (programului), fără a lua în considerare specificația (cerințele) programului
- pentru a utiliza metode structurale de testare programul poate fi reprezentat sub forma unui graf orientat
- datele de test sunt alese astfel încât 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 INTR-UN GRAF ORIENTAT
- Pentru o secventa de instructiuni se introduce un nod

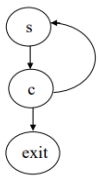
• if c then s1 else s2



• while c do s



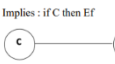
repeat s until c



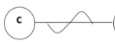
TESTARE FUNCTIONALA - Metoda Grafului Cauză-Efect (Cause-Effect Graphing)
• Partiționarea în categorii poate produce un număr mare de combinații de intrări, dintre care o mare parte poate fi nefezabilă. • Metoda grafului cauză-efect (cunoscuta și ca modelarea dependențelor) se concentrează pe modelarea relațiilor de dependență între condițiile de intrare ale programului (cauze) și condițiile de ieșire (efecte). • Relația dintre acestea este exprimată sub forma unui graf cauză-efect. • Graful cauză-efect = reprezentare vizuală a relației logice dintre cauze și efecte, exprimabilă ca o expresie Booleană. **Cauză =** orice condiție în specificație (cerințe) care poate afecta răspunsul programului. **Efect =** răspunsul programului la o combinație de condiții de intrare. Efectul nu este în mod necesar o ieșire (poate fi un mesaj de eroare, un display, o modificare a unei baze de date sau chiar un punct de testare intern).

Relații Cauză-Efect

Implies : if C then Ef

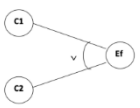


Not – Implies : if (¬C) then Ef



And – Implies: if (C1 && C2) then Ef

Or – Implies: if (C1 || C2) then Ef

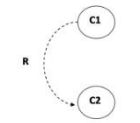


Constrângeri între cauze

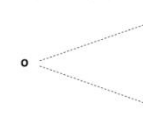
E (exclusive): fie C1 sau C2 (cel mult una dintre ele)

I (Inclusive): cel puțin C1 sau C2

R (Requires) C1 cere C2 (dacă C1 atunci C2)

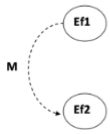


O (one and only one): unul și numai una dintre C1 și C2



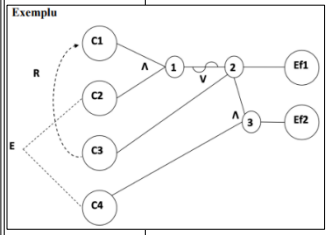
Constrângeri între efecte

M (Masks): Ef1 maschează Ef2 (dacă Ef1 atunci ¬Ef2)



Crearea tabelului de decizie din graful cauză-efect
Input: Un graf cauză-efect avand cauze C1, ..., Cp și efecte Ef1, ..., Efq.
Output: Un tabel de decizie avand N = p + q randuri și M coloane, unde M depinde de relatia dintre cauze și effect
Procedura de creare a tabelului de decizie:
1. Initializeaza nr_coloane=0(tabel de decizie gol)
2. For i=1 to q
2.1. e = Efi (selectează următorul efect pentru procesare)
2.2. Găsește combinațiile de condiții care produc apariția efectului e. Fie V1, ..., Vmi aceste combinații, mi > 0. Setează Vk(j), p < j ≤ p+q, la 1 dacă efectul Efj apare ca urmare a combinației respective și la 0 în caz contrar
2.3. Actualizează tabelul de decizie. Adaugă V1, ..., Vmi la tabel ca și coloane succesive începand cu poziția nr_coloane + 1.
2.4. nr_coloane = nr_coloane + mi

Exemplu



Pas 1: nr_coloane = 0
Pas 2: i = 1
Pas 2.1: e = Ef1
Pas 2.2: Se caută valorile lui C1, C2, C3 astfel încât ¬ (C1 ∧ C2) ∨ C3 = 1

1	0	1
0	1	1
0	0	1
1	1	1
1	0	0
0	1	0
0	0	0

---> In plus, se aplica constrangerea C3 implica C1

1	0	1
1	1	1
1	0	0
0	1	0
0	0	0

---> Se adauga C4=0 si valorile corespunzatoare pt Ef1 si Ef2

V1	1	0	1	0	1	0
V2	1	1	1	0	1	0
V3	1	0	0	0	1	0
V4	0	1	0	0	1	0
V5	0	0	0	0	1	0

	1	2	3	4	5
C1	1	1	1	0	0
C2	0	1	0	1	0
C3	1	1	0	0	0
C4	0	0	0	0	0
Ef1	1	1	1	1	1
Ef2	0	0	0	0	0

2.3. Matricea obținută este transpusă și adaugată la tabelul de decizie începând cu poziția nr_coloane + 1 = 1 (rezultat = tabelul din stanga)
2.4. Se actualizeaza nr_coloane = 0 + 5 = 5

2.5. i=2
2.6. e = Ef2
2.7. Se cauta valorile lui C1, C2, C3, C4 astfel incat (¬ (C1 ∧C2) ∨ C3) ∧ C4 = 1

Folosind combinațiile C1, C2, C3 anterioare pentru Ef1, obținem:

1	0	1	1
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	1

În plus, se aplică constrângerea ca C2 și C4 să nu existe simultan:

1	0	1	1
1	0	0	1
0	0	0	1

Se adauga valorile corespunzatoare pentru Ef1 si Ef2:

V1	1	0	1	1	1	1
V2	1	0	0	1	1	1
V3	0	0	0	1	1	1

Pas 2.8. Matricea obtinuta este transpusa si adaugata la tabelul de decizie incepand cu pozitia nr_coloane + 1 = 6

	1	2	3	4	5	6	7	8
C1	1	1	1	0	0	1	1	0
C2	0	1	0	1	0	0	0	0
C3	1	1	0	0	0	1	0	0
C4	0	0	0	0	0	1	1	1
Ef1	1	1	1	1	1	1	1	1
Ef2	0	0	0	0	0	1	1	1

Pas 2.9. nr_coloane = 5+3=8

STOP. nr_coloane este M = 8.

Generarea cazurilor de testare

Fiecare coloană din tabelul de decizie generează cel puțin un caz de testare, corespunzător combinației C1, ..., Cp respective.

Problemă: Explozie a stărilor datorită combinației de cauze.
Soluție: Limitarea numărului de cazuri de testare folosind euristici.

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 data.

a) Statement coverage(acoperire la nivel de instructiune) - 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
N	x	C	s	
1	a	a	y	
		b	n	

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.

Dezavantaje: 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 (branch) – la nivel de decizie/ramura - • 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
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

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 conditiile individuale ale fiecarei decizii.

c) Condition coverage: • Genereaza date de test astfel incat 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
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

Avantaj: Se concentreaza asupra conditiilor 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 - 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.
e) Multiple condition coverage - Genereaza date de test astfel încât să parcurga toate combinațiile posibile de adevărat și fals ale conditiilor individuale.

f) Modified condition/decision 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 2n 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.

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	False	False	True	C2
4	True	False	False	False	
5	True	False	True	True	C3
6	True	False	False	False	

Set de teste minimal

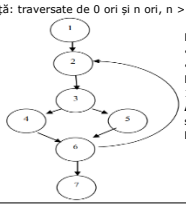
Test	C1	C2	C3	C
t1	True	True	False	True
t2	False	True	False	False
t3	True	False	False	False
t4	True	False	True	True

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 -> numărul de circuite liniar 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.
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 (Paige, Holthouse) - • Generează date pentru executarea fiecarei căi măcar o singura data. • 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.

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



Regex: 1.2.3.(4+5).6.(2.3.(4+5).6)*.7
Pt n=0, n=1: 1.2.3.(4+5).6.(2.3.(4+5).6+null).7
•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
Nr cai se obtine inlocuind 1 pt fiecare nod (si pt null), concat=inmult 1.1.1.(1+1).1.(1.1.(1+1).1+1).1=2.3=6 cai
Avantaje: Sunt selectate cai pe care alte metode de testare structurala nu le ating (inclusiv branch).
Dezavantaje: •Multe cai, o parte nu fezabile.
•Nu exerseaza conditiile individuale ale deciziilor.
•Tehnica descrisa pt generarea cailor nu este aplicabila direct prog Nestructurate.

STRUCTURAL TESTING – LINEAR CODE SEQUENCE AND JUMP COVERAGE - Un LCSAJ este o cale (execuție) a unui program formată dintr-o secvență de cod (Linear Code Sequence) urmată de un salt (Jump) al controlului programului.
Un LCSAJ este definit ca un triplet (X, Y, Z), unde • X este startul secvenței lineare, • Y este sfarsitul secvenței liniare, • Z este linia de cod unde este transferat controlul dupa sfârșitul secvenței liniare.
Exemplu: calculul x la puterea y, x > 0, y >= 0.

```
1  begin
2      int x, y, z;
3      read(x, y);
4      z = 1;
5      while (y > 0) {
6          z = z*x;
7          y = y -1;
8      }
9      write(z);
10 end
```

LCSAJ	Start	End	Jump to
1	1	8	5
2	5	8	5
3	5	5	9
4	1	5	9
5	9	9	Exit

Consideram T = {t1, t2}, unde t1 = (x = 3, y = 0), t2 = (x = 3, y = 2)
t1: (1, 5, 9) → (9, 9, exit)
t2: (1, 8, 5) → (5, 8, 5) → (5, 5, 9) → (9, 9, exit)

T acoperă toate cele 5 LCSAJ.
Un set de teste care realizează o acoperire la nivel de decizie nu realizează în mod necesar o acoperire la nivel de LCSAJ.

MUTATION TESTING - Tehnica de evaluare a unui set de teste pentru un program (având un set de teste generat, putem evalua cât de eficient este, pe baza rezultatelor obținute de acest test asupra mutanților programului).
Mutation = modificare foarte mică (din punct de vedere sintactic) a unui program.
Pentru un program P, un mutant M al lui P este un program obținut modificând foarte ușor P; M trebuie să fie corect din punct de vedere sintactic.
Tehnica Mutation testing: • Generarea mutanților pentru programul P (folosind o mulțime de operatori de mutație). • Rularea setului de teste asupra programului P și asupra setului de mutanți; dacă un test distinge între P și un mutant M spunem că P omoară mutantul M.
Mutanți de primul ordin / mutanți de ordin mai mare (first-order/higher-order mutants)
• First-order mutants = mutanți obținuți făcând o singură modificare în program • n-order mutants = mutanți obținuți făcând n modificari în program • n-order mutant = first-order mutant of a (n-1)-order mutant, n > 1
• n-order mutant, n > 1, sunt numiți higher-order mutants.
In general, în practică sunt folosiți doar mutanții de ordin 1. Motive: • Numarul mare de mutanți de ordin 2 sau mai mare • Coupling-effect.
Principiile de bază ale mutation testing: • Competent programmer hypothesis (CPH) - Pentru o problemă dată, programatorul va scrie un program care se află în vecinatatea unui program care rezolvă în mod corect problema (și deci, erorile vor fi detectate folosind mutanți de ordinul 1). **• Coupling effect -** Datele de test care disting orice program care diferă cu puțin de programul corect sunt suficient de puternice pentru a distinge erori mai complexe. Rezultate experimentale arată că un set de teste care distinge un program de mutanții săi de ordin 1 este foarte aproape de a distinge programul de mutanții de ordin 2. Explicație intuitivă: în general erorile simple sunt mai greu de detectat. Erorile complexe pot fi detectate de aproape orice test.
Strong mutation/ weak mutation - Un test t omoară mutantul M (distinge M față de P) dacă cele două se comportă diferit pentru testul t. Întrebare: când observam comportamentul celor două programe ?
• Testul t aduce pe P și M în stari diferite - se observă starea programului (valorile variabilelor afectate) dupa execuția instrucțiunii mutate.
• Schimbarea stării se propagă la sfarșitul programului - se observă valorile variabilelor returnate și alte efecte (schimbarea variabilelor globale, fișiere, baza de date), imediat dupa terminarea programului.
• Weak mutation: prima condiție este satisfăcută. **• Strong mutation:** ambele condiții sunt satisfăcute.
• Strong mutation: mai puternică. Se asigură că testul t detectează cu adevarat problema
• Weak mutation: necesită mai puțina putere de calcul; strâns legată de ideea de acoperire
Mutanți echivalenți - Un mutant M a lui P se numește echivalent dacă el se comportă identic cu programul P pentru orice date de intrare. Altfel, se spune că M poate fi distins de P. Din punct de vedere teoretic: în general, problema determinării dacă un mutant este echivalent cu programul părinte este nedecidabilă (este echivalentă cu halting problem). În practică: determinarea echivalenței se face prin analiza codului. Determinarea mutanților echivalenți poate fi un proces foarte complex – principala problemă practică a tehnicii mutation testing (avem nevoie să decidem dacă mutanții sunt sau nu echivalenți pentru a putea evalua eficiența testelor).
Utilitatea mutation testing: • Evaluarea unui set de date existent (și construirea de noi teste, dacă testele existente nu omoară toți mutanții) • Detectarea unor erori în cod.
Mutation score: MS(T) = D/(L+D), unde **D** – numărul de mutanți distinși **L** – numărul de mutanți nedistinși (live mutants) neechivalenți.
Pentru ca un test t să distingă între P și M trebuie ca: **• Reachability:** Instrucțiunea mutată să fie executată la aplicarea lui t **• State infection:** Instrucțiunea mutată să afecteze starea programului **• State propagation:** Schimbarea de stare să se propage în exterior.
EXEMPLU: Pentru exemplul dat, pentru ca un test t să distingă între P și M: **• Reachability:** TRUE **• State infection:** (x < 1 ∧¬ (x < 0)) **• State propagation:** ¬ (x > 2) Condiția rezultată: (x = 0) ∧ (x <= 2) ⇔ x = 0
Pentru x = 0 programul corect întoarce 2 în timp ce programul greșit returnează 3.

WHAT IS TESTING? - • It's always the case that we have some software under test **(SUT).** • On the other hand, selecting a good set of test inputs, and designing good acceptability test end up being actually really hard, and basically, these are what we are going to be spending this course talking about.
• The goal of testing isn't so much as finding bugs, but rather it's finding bugs as early as possible. **More testing is not always better**. In fact, the quality of testing is all about the cost/benefit tradeoff. And fundamentally, testing is an economic activity. We're spending money or we're spending effort on testing in order to save ourselves money and effort later. Going along with this, testing methods should be evaluated about the cost per defect found. We're trying to make arguments **that a single test case is a representative of a whole class of actual executions of the system that we're testing**.
Creating testable software - • SUT: • clean code • refactor • should always be able to describe what a module does & how it interacts with other code • no extra threads • no swamp of global variables • no pointer soup • Modules should have unit tests • when applicable, support fault injection • assertions, assertions, assertions.
Assertions: Executable check for a property that must be true for your code. •Assertions are not for error handling, •NO SIDE EFFECTS, •No silly assertions.
Why assertions? - • Make code self-checking, leading to more effective testing • Make code fail early, closer to the bug • Assign blame • Document assumptions, preconditions, postconditions, & invariants.
Disabling assertions – Advantages: • Code runs faster • Code keeps going What is it that we're trying to do with our system? Is it better to keep going or is it better to stop? Keeping going after some condition is true that will lead to an assertion violation may lead to a completely erroneous execution. On the other hand, possibly, that's better than actually stoping. **Disadvantages. •** What if our code relies on a side-effecting assertion? • Even in production code may be better to fail early.
Specifications - • The SUT is providing some set of APIs, i.e. a set of function calls that can be called by another software.
Domains and ranges - • If we think of a piece of software as a mathematical object, we'll find the software has a domain of values. Correspondingly, every piece of software also has a range. • Sometimes as a software tester, you'll test code with an input that looks like it should be part of the domain and the code will malfunction, will crash with some sort of a bad error, perhaps maybe not throw an exception but rather actually exit abnormally. • Restrictions on the domain of functions are actually a very valuable tool in practice because otherwise, every function or piece of software that we implement, has to contain maximal defensive code against illegal inputs. And in practice, this kind of defensive coding is not generally possible.
Crashme - • This is one of the ways that we actually test operating systems: using a tool called crashme that allocates a block of memory, writes totally random garbage into it, then it masks off all signal handlers, i.e. system level exception handlers and it jumps into the block of garbage, i.e. it starts executing completely garbage bytes.
• This brings us to is the idea of **defensive coding** i.e, error checking for its own sake to detect internal inconsistencies.
• If we just hope that the software does the right thing, then one of the golden rules of testing is we shouldn't ever just hope that it does something; we need to actually check this.
Fault injection - • First, you should always try to use low level programming or interfaces that are predictable and that return friendly error codes. Given a choice between using the UNIX system call and using the Python libraries, you'd almost always choose the Python libraries. • We don't always have the option of doing this so we're forced to use these bad style APIs sometimes. From a testing point of view, we can often use the technique called fault injection to deal with these kind of problems. • So we have a stub function and we can sometimes cause the open system call to fail.
Faults injected into a SUT should be: • not all possible faults, • not none, • yes – faults that we want our code to be robust to.
Therac – radiation machine – avea un fault la nivel hardware, daca lumea introducea repde datele si ii dadea drumul avea sansa sa ii dea o cantitate prea mare de radiatii – ideea ca sunt conditii de cursa.
Nonfunctional inputs - • These are inputs that affect the operation of a SUT that have nothing to do with the APIs provided or that are used by the software that we're testing.
Unit testing - Unit testing means looking at some small software module at a time and testing it in an isolated fashion. • The main thing that distinguishes unit testing from other kinds of testing is that we're testing a smaller amount of software. •The goal of unit testing is to find defects in the internal logic of the SUT as early as possible, in order to create more robust software modules that we can compose later with other modules and end up with a system that actually works.
Integration testing - • Integration testing refers to taking multiple software modules that have already been unit tested and testing them in combination with each other. • What we're really testing are the interfaces between modules, and the question is did we define them tightly enough.
System testing - • Here we're asking the question does the system as a whole meet its goals? • And often at this point we're doing black box testing, and that's for a couple of reasons: • the system is probably large enough, • we're not so much concerned with what's going on inside the system, • at this level we are often concerned with how the system will be used, • we may not care about asking the system work for all possible use cases. Rather, we would simply like to make sure that it performs acceptably for the important use cases.
Differential testing - we are taking the same test input delivering it to 2 different implementations of the SUT and comparing them for equality.
Stress testing - is a kind of testing where a system is tested at or beyond its normal usage limits. • Stress testing is typically done to assess the robustness and reliability of the SUT.
Random testing - we use the results of a pseudo-random number generator to randomly create test inputs, and we deliver those to the SUT.
Regression testing - always involves taking inputs that previously made the system fail and replaying them against the system.
Testing car software = stress testing = • We have a piece of SUT that is some sort of a critical embedded system, for example, it's controlling airbags on a vehicle. And this vehicle is going to be out in the field for many years, and so cars get subjected to fairly extreme conditions.
Testing a web service = system/validation testing - SUT is some sort of a web service, exposed to some small fraction of users who have been selected, based on their willingness and desire to use new features.
Testing a new library = Unit testing - Let's say this library is implementing numerical functions, and these numerical functions have been sometimes throwing floating point exceptions and crashing our system.
Being great at testing – Developer wants his code to succeed, the Tester wants it to fail.

TRADEOFFS IN RANDOM TESTING – Advantages: •less tester bias, weaker hypotheses about where bugs are, • once testing is automated, human cost is 0, •every fuzzer finds different bugs, • surprises
Disadvantages - • input validity can be hard, • oracles are hard too, • no stopping criterion, • may find unimportant test bugs, • can be hard to debug when test case is large and/or makes no sense, • every fuzzer finds different bugs, • may find the same bugs many times.

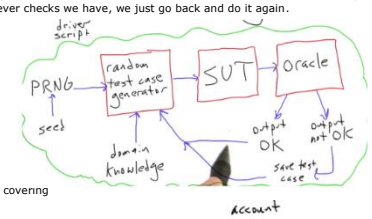
```
def correct_test():
    # Queue1 silently holds only 2 byte unsigned integers,
than wraps around

    q = Queue1(2)
    succeeded = q.enqueue(100000)  # value greater than 2^16
    assert succeeded
    value = q.dequeue()
    assert value == 100000  # test1 failed
    # Queue2 silently fails to hold more than 15 elements

    q = Queue2(30)
    # try to enqueue more than 15 elements
    for i in range(20):
    succeeded = q.enqueue(i)
    assert succeeded  # test2 failed
    # Queue3 implements empty() by checking if dequeue() succeeds.
    # This changes the state of the queue unintentionally.

    q = Queue3(2)
    succeeded = q.enqueue(10)
    assert succeeded
    assert not q.empty()  # the function checks by trying to dequeue
    value = q.dequeue()
    assert value == 10  # test3 failed
    # Queue4 dequeue() of an empty queue returns False instead of None

    q = Queue4(2)
    value = q.dequeue()
    assert value is None  # test4 failed
    # Queue5 holds one less item than intended

    q = Queue5(2)
    for i in range(2):
    succeeded = q.enqueue(i)
    assert succeeded  # test4 failed
```

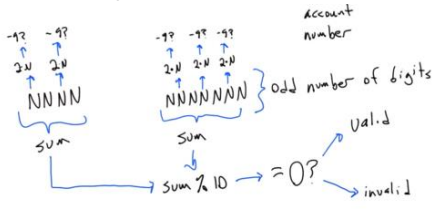
COVERAGE TESTING - • Usually problems in released software are coming from things that people forgot to test, i.e. testing was inadequate and the developers were unaware of that fact. • In the following we present a collection of techniques called **code coverage** where automated tools can tell us places where our testing strategy is not doing a good job.
• Let's consider a subset of the input domain. For purposes of finding defects in the SUT, we pick an arbitrary point and execute the system on it. We look at the output, and if it is acceptable, then we're done testing that class of inputs.
Coverage - • Test coverage is trying to accomplish exact the same thing that partitioning was accomplishing, but it goes about in a different way. • Test coverage is an **automatic way of partitioning the input domain with some observed features of the source code**. • One particular kind of test coverage is called **function coverage** and is achieved when every function in our source code is executed during testing.
• In practice, we start with a set of test of cases, and we run them all through the SUT. We see which functions are called and then we end up with some sort of a score called a test coverage metric. • The score assigned to a collection of test cases is very useful. • For each of the functions that wasn't covered, we can go and look at it and we can try to come up with the test input that causes that function to execute. • If there is some function baz() for which we can't seem to devise an input that causes it to execute, then there are a couple of possibilities. One possibility is that it can't be called at all. It's dead code. Another possibility is that we simply don't understand our system well enough to be able to trigger it.
Test coverage - Test coverage is a measure of the proportion of a program exercised during testing. **Advantages:** • Gives us an objective score, • When coverage is <100% we are given meaningful tasks, **Disadvantages:** • Not very helpful in finding errors of omission, • Difficult to interpret scores <100%, • 100% coverage does not mean all bugs were found.

• Why did test coverage fail to identify the bug? Statement coverage is a rather crude metric that only checks whether each statement executes once. Each statement executes at least once that lets a lot of bugs slip through. • The lesson here is we should not let complete coverage plus a number of successful test cases fool as into thinking that a piece of code of right. It's often the case that deeper analysis is necessary.

Coverage metrics: • Line coverage - is very similar to statement coverage but the metric is tied to actual physical lines in the source code. In this case, there is only one statement for each line so statement coverage and line coverage would be exactly identical. • But on the other hand, if we decided to write some code that had multiple statements per line, line coverage would conflate them whereas statement coverage would consider them individual statements. For most practical purposes, these are very similar. So, statement coverage has a slightly finer granularity.
• Loop coverage specifies that we execute each loop 0 times, once, and more than once.
Concurrent software - • The transfer function does not synchronise, i.e. it hasn't taken any sort of a lock while it manipulates the accounts. If these threads are operating on the same accounts concurrently, then it's going to be a problem.
Synchronization coverage - We can delete all of the locks out of a code, run it through some tests, and often it passes. This is in spite of the test coverage metric called synchronization coverage, which ensures that during testing this lock actually does something.
Interleaving coverage means if we recall functions which accessed shared data are actually called and in a truly concurrent fashion, i.e. by multiple threads at the same time.
• The obvious problem is, if we partition the input domain this way and we go ahead and test, it is easier to reach the region in the input domain, and get good coverage, but **we won't be able to find errors of omission.**
Infeasible code – What happens when we have some statements that we haven't been able to cover? 3 possibilities - **•Infeasible code, •Code not worth covering, •Inadequate test suite.**
Automated white box testing – Klee - • Is not a form of code coverage but what rather a way to get software tools to automatically generate test for our code. • The goal for this tool is to generate good path coverage for the code. • Therefore, a tool will iterate the process of generating inputs that take different branches multiple times and then using what it learned about the code build up a set of constraints to explore different paths, pass it to the solver and the solver is either going to succeed in coming up with a new value or possibly it will fail.
How to use coverage - • We strongly believe that if we have a good test suite, and we measure it's coverage, the coverage will be good. We do not believe, on the other hand, that if we have a test suite which gets good coverage, it must be a good test suite. • Used in the right way, coverage can be a relatively low cost way To improve the testing that we do for a piece of software. • Used incorrectly, it can waste our time, and perhaps worst, lead to a false sense of security

RANDOM TESTING - • Test cases are created using input from a random number generator. PRNG here stands for pseudorandom number generator. A seed completely determines the sequence of random numbers it's going to generate. • Random testing diagram: • SUT executes and produces some output. The output is inspected by a test oracle. The oracle makes a determination whether the output is either good or bad. • If the output is good, i.e., if it passes whatever checks we have, we just go back and do it again.
• Random testing can significantly increase our confidence that the SUT is working as intended.
• First of all, it can be tricky to come up with a good random test case generator, and second, they can be tricky to come up with good oracle. We've already said that these are the hard things about testing in general, making test cases, and determining if outputs are correct.
Input validity - • But realistically, it's usually a little bit more involved than the one we just saw. The key problem is generating inputs that are valid, i.e. inputs that are part of the input domain for the SUT.
• So, in most cases when we do random testing what we're looking for is something like the blue "flat" line, which indicates that we're covering all parts of the SUT roughly equally.
Random testing vs fuzzing
Around 2000 the connotation of the term fuzzing was penetration testing, i.e. finding security vulnerabilities in applications.
• One of them is where we find a particular test case that makes our system fail and we'd like to save off that test case for later use in regression testing.






```
        q.checkRep()
        removeEmpty += 1
    else:
        expected_value = l.pop(0)
        assert dequeued == expected_value
        remove += 1
  while True:
    res = q.dequeue()
    q.checkRep()
    if res is None:
        break
    z = l.pop(0)
    assert z == res
  assert len(l) == 0
  print("adds: " + str(add))
  print("adds to a full queue: " + str(addFull))
  print("removes: " + str(remove))
  print("removes from an empty queue: " + str(removeEmpty))

def time_test():
  q = Queue(50000)
  for i in range(50000):
      q.enqueue(0)
      q.checkRep()
  for i in range(50000):
      q.dequeue()
      q.checkRep()

random_test()
time_test()
```

```
# at a random position of the buffer #
change the byte to a random one (5
lines of python code)
numwrites                          =
random.randrange(math.ceil((float(le
n(buf)) / FuzzFactor))) + 1

for j in range(numwrites):

  rbyte = random.randrange(256)
  rn = random.randrange(len(buf))
  buf[rn] = "%c"%(rbyte)
```

```
# TASK:
# Write a random tester for the Queue class.
# The random tester should repeatedly call the Queue methods
# on random input in a semi-random fashion.
# For instance, if you wanted to randomly decide between
# calling enqueue and dequeue, you would write something like
this:
#
# q = Queue(500)
# if (random.random() < 0.5):
#   q.enqueue(some_random_input)
# else:
#   q.dequeue()
#
# You should call the enqueue, dequeue, and checkRep methods
# several thousand times each.

def random_test():
 N = 4
 add = 0
 remove = 0
 addFull = 0
 removeEmpty = 0
 q = Queue(N)
 q.checkRep()
 l = []
 for i in range(100000):
     if random.random() < 0.5:
         z = random.randint(0, 1000000)
         res = q.enqueue(z)
         q.checkRep()
         if res:
             l.append(z)
             add += 1
         else:
             assert len(l) == N
             assert q.full()
             q.checkRep()
             addFull += 1
     else:
         dequeued = q.dequeue()
         q.checkRep()
         if dequeued is None:
         assert len(l) == 0
         assert q.empty()
```

Generating random inputs - • All of these ways of generating input are variations on a single theme which we call **generative random testing**, i.e. inputs are created from scratch. • There's an entirely different approach called mutation-based random testing, i.e. inputs are created by randomly modifying existing non-randomly created inputs by the SUT.
Mutation based random testing - • A generative random tester will test inputs from a cluster found in some part of the input domain. • A mutation-based random tester will start with some known input, will randomly modify it ending with test cases that are in the same neighborhood as the original input. So we're exploring interesting parts of the input domain, that we could have never reached this part of the input domain using any kind of a generative random test case generator.

Oracles - • Oracles are extremely important for random testing because if you don't have an automated oracle, i.e. if you don't have an automated way to tell if a test case did something interesting then you've got nothing.
Weak oracles - are some of the ones that are most useful in practice. They can only enforce fairly generic properties about SUT.
• If the oracle is not automated, you don't have an oracle. **• Medium oracles** – Assertions, • One example of a medium power oracle is assertion checks that the programmer has put into the software. • A medium power oracle doesn't guarantee anything even remotely close to actual correct operation. **Strong oracles –** alternate implement, differential testing of compilers. Use one if you can find one.
Functional inverse pairs – strong oracle – we have available some function and also its inverse – use as pair
Null space transformation – strong oracle - we take a random test case and we make some changes to it that it shouldn't affect how it's treated by the SUT.