

# Mutation testing (mutation analysis)

# Mutation testing

- Tehnica de evaluare a unui set de teste pentru un program (avand un set de teste generat, putem evalua cat de eficient este, pe baza rezultatelor obtinute de acest test asupra mutantilor programului)
- Mutation = modificare f mica (din punct de vedere sintactic) a unui program
- Pentru un program P, un mutant M al lui P este un program obtinut modificand f usor P; M trebuie sa fie corect din punct de vedere sintactic.

# Mutanti - esemplu

## Program P

- begin
- int x, y;
- read(x, y);
- if (x > 0)
- write(x+y)
- else
- write(x\*y)
- end

## Mutant M1

- begin
- int x, y;
- read(x, y);
- if (x > **=0**)
- write(x+y)
- else
- write(x\*y)
- end

# Mutanti - esemplu

## Program P

- begin
- int x, y;
- read(x, y);
- if (x > 0)
- write(x+y)
- else
- write(x\*y)
- end

## Mutant M2

- begin
- int x, y;
- read(x, y);
- if (x > 0)
- write(**x-y**)
- else
- write(x\*y)
- end

# Mutanti - esemplu

## Program P

- begin
- int x, y;
- read(x, y);
- if (x > 0)
- write(x+y)
- else
- write(x\*y)
- end

## Mutant M3

- begin
- int x, y;
- read(x, y);
- if (x > 0)
- write(**x+y+1**)
- else
- write(x\*y)
- end

# Tehnica Mutation testing

- Generarea mutantilor pentru programul P (folosind o multime de operatori de mutatie)
- Rularea setului de teste asupra programului P si setului de mutanti; daca un test distinge intre P si un mutant M spunem ca P omoara mutantul M.

# First-order/higher-order mutants

- First-order mutants = mutanti obtinuti facand o singura modificare in program
- n-order mutants = mutanti obtinuti facand n modificari in program
- n-order mutant = first-order mutant of a (n-1)-order mutant,  $n > 1$
- n-order mutant,  $n > 1$ , sunt numiti higher-order mutants

# Mutanti de ordin mai mare - exemplu

## Program P

- begin
- int x, y;
- read(x, y);
- if (x > 0)
- write(x+y)
- else
- write(x\*y)
- end

## Mutant ordin 2

- begin
- int x, y;
- read(x, y);
- if (x > 0)
- write(**x+y+1**)
- else
- write(x\*y)
- end



# Utilizare first-order/higher-order mutants

- In general, in practica sunt folositi doar mutantii de ordin 1.
- Motive
  - Numarul mare de mutanti de ordin 2 sau mai mare
  - Coupling-effect

# Principiile de baza ale mutation testing

- Competent programmer hypothesis (CPH)
- Coupling effect

# Competent programmer hypothesis

- Pentru o problema data, programatorul va scrie un program care se afla in vecinatatea unui program care rezolva in corect problema
- deci, erorile vor fi detectate folosind doar mutanti de ordinul 1

# Coupling effect

- Datele de test care disting orice program care difera cu putin de programul corect sunt suficient de puternice pentru a distinge erori mai complexe.
- Rezultate experimentale arata ca un set de teste care distinge un program de mutantii sai de ordin 1 este f aproape de a distinge programul de mutantii de ordin 2
- Explicatie intuitiva: in general erorile simple sunt mai greu de detectat. Erorile complexe pot fi detectate de aproape orice test

# Strong mutation/ weak mutation

- Un test  $t$  omoroara mutantul  $M$  (distinge  $M$  fata de  $P$ ) daca cele doua se comporta diferit pentru testul  $t$ .
- Intrebare: cand observam comportamentul celor doua programe ?

# Strong mutation/ weak mutation (2)

- Testul t aduce pe P si M in stari diferite - se observa starea programului (valorile variabilelor afectate) dupa executia instructiunii mutate.
- Schimbarea starii se propaga la sfarsitul programului - se observa valorile variabilelor returnate si alte efecte (schimbarea variabilelor globale, fisiere, baza de date), imediat dupa terminarea programului
- **Weak mutation:** prima conditie este satisfacuta
- **Strong mutation:** ambele conditii sunt satisfacute

# Strong mutation/ weak mutation - exemplu

## Program P

- begin
- int x, y;
- read(x, y);
- y := y+1;
- if (x > 0)
- write(x)
- else
- write(y)
- end

## Mutant M

- begin
- int x, y;
- read(x, y);
- y := y-1;
- if (x > 0)
- write(x)
- else
- write(y)
- end

# Strong mutation/ weak mutation – exemplu

- Testul (1, 1) distinge intre P si M d.p.d.v. weak mutation, dar nu distinge intre P si M d.p.d.v. strong mutation
- Testul (0, 1) distinge intre P si M d.p.d.v. strong mutation



# Strong mutation/ weak mutation (3)

- Strong mutation: mai puternica. Se asigura ca testul t detecteaza cu adevarat problema
- Weak mutation: necesita mai putina putere de calcul; strans legata de ideea de acoperire

# Mutanti echivalenti

- Un mutant  $M$  a lui  $P$  se numeste echivalent daca el se conpoirta identic cu programul  $P$  pentru *orice* date de intrare.
- Altfel , se spune ca  $M$  poate fi distins de  $P$ .
- Avem neoie sa decidem daca mutantii sunt sau nu echivalenti pentru a putea evalua eficienta testelor

# Mutanti echivalenti - probleme

- Din punct de vedere teoretic: in general, problema determinarii daca un mutant este echivalent cu programul parinte este nedecidabila (este echipaventa cu halting problem)
- In practica: determinarea echivalentei se face prin analiza codului
- Determinarea mutantilor echivalenti poate fi un proces f complex – principala problema practica a tehnicii mutation testing

# Utilitatea mutation testing

- Evaluarea unui set de date existent (si construirea de noi teste, daca testele existente nu omoara toti mutantii)
- Detectarea unor erori in cod

# Evaluarea unui set de date existent - exemplu

- begin
- int x, y;
- read(x, y);
- if (x > 0)
- write(x+y)
- else
- write(x\*y)
- end

# Evaluarea unui set de date existent - exemplu

- Consideram urmatorii operatori de mutatie:
  - + inlocuit de  $-$
  - $*$  inlocuit de  $/$
  - o variabila sau o constanta  $x$  este inlocuita de  $x+1$

# Mutanti ottenuti

- begin
- int x, y;
- read(x, y);
- if (x > 0)
- M1           if (x+1 > 0)
- M2       if (x> 0+1)
- M3       write(x+1+y)
- M4       write(x+y+1)
- M5       write(x-y)
- else
- M6       write((x+1)\*y)
- M7       write(x\*(y+1))
- M8       write(x/y)
- end

# Set de teste utilizat

- Set de teste  $T = \{t1, t2, t3, t4\}$
- $t1 = (0, 0),$
- $t2 = (0, 1),$
- $t3 = (1, 0),$
- $t4 = (-1, -1)$



# Evaluare - rezultate

P	t1	t2	t3	t4	Mutant distins
P(t)	0	0	1	1	
M1(t)	0	1	NE	NE	Y
M2(t)	0	0	0	NE	Y
M3(t)	0	0	2	NE	Y
M4(t)	0	0	2	NE	Y
M5(t)	0	0	1	1	N
M6(t)	0	1	NE	NE	Y
M7(t)	0	0	1	0	Y
M8(t)	ND	NE	NE	NE	Y

# Evaluare - rezultate

- Mutanti nedistinsi (alive) = {M5}
- Intrebare: Este M5 mutant echivalent ?
- Raspuns: Nu. (1, 1) distinge intre P si M5

# Mutation score (MS)

- $MS(T) = D/(L+D)$ , unde
  - D – numărul de mutanti distinsi
  - L – numărul de mutanti nedistinsi (live mutants) neechivalenti
- Pentru exemplu:  $MS(T) = 7/8$

# Detectarea erorilor folosind mutatia – ex.

## Program P

- begin
- int x, y
- read(x)
- y = 1
- y = 2
- if (x < 0)
- y = 3
- if (x > 2)
- y = 4
- write(y)
- end

← Eroare: instructiune lipsa

## Mutant M

- ...
- if (x < 1)
- y = 3
- ...

# Detectarea erorilor folosind mutatia – ex.

- Aratam ca mutantul M genereaza teste care detecteaza eroarea
- Pentru ca un test t sa distinga intre P si M trebuie ca:
  - Reachability: Instructiunea mutata sa fie executata la aplicarea lui t
  - State infection: Instructiunea mutata sa afecteze statea programului
  - State propagation: Schimbarea de state sa se propage in exterior

# Detectarea erorilor folosind mutatia – ex.

- Pentru exemplul dat, pentru ca un test  $t$  sa distinga intre  $P$  si  $M$ :
  - Reachability: TRUE
  - State infection:  $(x < 1 \wedge \neg(x < 0))$
  - State propagation:  $\neg(x > 2)$
- Conditia rezultata:  $(x = 0) \wedge (x \leq 2) \Leftrightarrow x = 0$
- Pentru  $x = 0$  programul corect intoarce 2 in timp ce programul gresit returneaza 3

# Operatori de mutatie

- Operator de mutatie = Regula care se aplica unui program pentru a crea mutanti
- E.g. inlocuirea/adaugarea/stergerea unor operanzi, stergerea unor instructiuni, etc.
- Programul nou obtinut trebuie sa fie valid din punct de vedere sintactic

# Operatori de mutatie in Java (MuJava)

- Traditional mutation operators (method-level operators) – operatori aplicabili oricarui limbaj procedural
- Class mutation operator – operatori specifici paradigmei orientate pe obiect si sintaxei Java
  - Incapsulare
  - Mostenire
  - Polimorfism si dynamic binding
  - Suprascrierea metodelor
  - Java specific



# Operatori de mutatie in Java (MuJava)

- Operatori tradizionali
- <http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf>
- Operatori de clasa
- <http://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf>