Testiranje mutacijom

Uvod

- Mutaciono testiranje spada u tehnike testiranja **zasnovane na defektima**
- Osnovni koncept testiranja na bazi defekata je da se izaberu slučajevi testiranja koji prave razliku između programa koji testiramo i alternativnih programa koji sadrže hipotetičke greške. Ovo se obično postiže izmenom programa koji se testira da se proizvedu hipotetički pogrešni programi. Ovakvo "sejanje defekata" se može koristiti za procenu temeljitosti (adekvatnosti) serije testova, ili za izbor novih test slučajeva da se dodaju seriji testova, ili da se proceni broj defekata u programu.

Testiranje mutacijom

- Ideja na kojoj se zasniva testiranje mutacijom:
  - U pojednom trenucima čini se male izmene u programu iz skupa unapred definisanih izmena.
- Pravilo po kome se menja tekst programa naziva se **mutacioni operator**, a varijanta programa kada se načini izmena naziva se **mutant**

Mutanti

- Mutanti moraju biti validni (sintaksno ispravni, tj. da prolaze kompajliranje)
- Ponašanje mutanta mora se razlikovati od ponašanja originala samo za mali broj test slučajeva da bi bili korisni. Nije koristan onaj koji pada na svim testovima.

Osnovne pretpostavke mutacionog testiranja

- "**Kompetentan programer**" – Za specificirane zahteve, programer piše program koji se (sintaksno) ne razlikuje mnogo od potpuno korektnog.
- "**Efekat sprege**" – serija testova koji pravi detektuje programe koji se razlikuju od ispravnog samo po jednostavnim greškama (izmena programa na jednom mestu), dovoljno je osetljiva da detektuje i kompleksne greške (izmene na više mesta).

Primer: program koji vrši konverziju krajeva linija između DOS, Unix i Mac konvencija

```
1
2  /** Convert each line from standard input */
3  void transduce() {
4      #define BUFLEN 1000
5      char buf[BUFLEN]; /* Accumulate line into this buffer */
6      int pos=0; /* Index for next character in buffer */
7
8      char inChar; /* Next character from input */
9
10      int atCR = 0; /* 0="within line", 1="optional DOS LF" */
11
12      while ((inChar = getchar()) != EOF ) {
13        switch (inChar) {
14        case LF:
15          if (atCR) { /* Optional DOS LF */
16              atCR = 0;
17          } else {      /* Encountered CR within line */
```

Primer: program koji vrši konverziju krajeva linija između DOS, Unix i Mac konvencija

```
18              emit(buf, pos);
19              pos=0;
20          }
21          break;
22       case CR:
23          emit(buf, pos);
24          pos=0;
25          atCR = 1;
26          break;
27     default:
28          if (pos >= BUFLEN-2) fail("Buffer overflow");
29          buf[pos++] = inChar;
30     }/* switch */
31   }
32   if (pos > 0) {
33       emit(buf, pos);
34   }
35 }
```

Mutacioni operatori

- Za različite programske jezike definiše se skup mutacionih operatora.
- Na sledećem slajdu je primer m.o. za C, dat je opis zajedno sa ograničenjem koje treba da obezbedi da test primeri prepoznaju mutanta.
- Npr. **Svr** operator (zamena jedne skalarne promenljive drugom) može biti primenjeno samo na promenljive kompatibilnih tipova (da ne bi izazvalo sint. grešku) i ograničenje za testiranje je da u trenutku izvršavanja mutiranog iskaza vrednosti originalne i zamenjene promenljive budu različite, da bi mutant imao šanse da bude prepoznat.

Tabela mutacionih operatora za C

(prema Pezzand, Young, Software Testing and Analysis: Process, Principles and Techniques)

ID	Operator	Description	Constraint
Operand Modifications			
crp	constant for constant replacement	replace constant *C*1 with constant *C*2	*C*1 ≠ *C*2
scr	scalar for constant replacement	replace constant *C* with scalar variable *X*	*C* ≠ *X*
acr	array for constant replacement	replace constant *C* with array reference *A*[*I*]	*C* ≠ *A*[*I*]
scr	struct for constant replacement	replace constant *C* with struct field *S*	*C* ≠ *S*
svr	scalar variable replacement	replace scalar variable *X* with a scalar variable *Y*	*X* ≠ *Y*
csr	constant for scalar variable replacem	replace scalar variable *X* with a constant *C*	*X* ≠ *C*
asr	array for scalar variable replacement	replace scalar variable *X* with an array reference *A*[*I*]	*X* ≠ *A*[*I*]
ssr	struct for scalar replacement	replace scalar variable *X* with struct field *S*	*X* ≠ *S*
vie	scalar variable initialization eliminat	remove initialization of a scalar variable	
car	constant for array replacement	replace array reference *A*[*I*] with constant *C*	*A*[*I*]≠*C*
sar	scalar for array replacement	replace array reference *A*[*I*] with scalar variable *X*	*A*[*I*]≠*C*
cnr	comparable array replacement	replace array reference with a comparable array refer	
sar	struct for array reference replacement	replace array reference *A*[*I*] with a struct field *S*	A[*I*]≠*S*

Tabela mutacionih operatora za C

(prema Pezzand, Young, Software Testing and Analysis: Process, Principles and Techniques)

ID	Operator	Description	Constraint
Expression Modifications			
abs	absolute value insertion	replace *e* by abs(*e*)	$e < 0$
aor	arithmetic operator replacement	replace arithmetic operator ψ with arithm. operator φ	$e_1 \psi e_2 \neq e_1 \varphi e_2$
lcr	logical connector replacement	replace logical connector ψ with logical connector φ	$e_1 \psi e_2 \neq e_1 \varphi e_2$
ror	relational operator replacement	replace relational operator ψ with relational operator φ	$e_1 \psi e_2 \neq e_1 \varphi e_2$
uoi	unary operator insertion	insert unary operator	
cpr	constant for predic. replacement	replace predicate with a constant value	
Statement Modifications			
sdl	statement deletion	delete a statement	
sca	switch case replacement	replace the label of one case with another	
ses	end block shift	move } one statement earlier and later	

Testiranje mutacijom

- Za zadati program i seriju testova T mutaciona analiza sastoji se od sledećih koraka:

1. Izaberu se mutacioni operatori. Ako smo zainteresovani za posebne klase grešaka, možemo izabrati skup mutacionih operatora relevantnih za te greške.
2. Generišu se mutanti mehaničkom primenom jednog po jednog mutacionog operatora na prvobitni program.
3. Prepoznavanje mutanata izvršavanjem originalnog programa i svakog generisanog mutanta sa test primerima iz T. Mutanta je **likvidiran** (**killed**) kada se rezultati testiranja razlikuju od originalnog programa.

Primer

- Za dati program transduce razvili smo seriju testova:

Test case	Description
1U	One line, Unix line-end
1D	One line, DOS line-end
2U	Two lines, Unix line-end
2D	Two lines, DOS line-end
2M	Two lines, Mac line-end
End	Last line not terminated with line-end sequence
Long	Very long line (greater than buffer length)

Primer

- Napravili smo sledeće mutante:

ID	Operator	line	Original/Mutant	1U	1D	2U	2D	2M	End	Long
M1	ror	28	(pos >= BUFLEN–2) (pos == BUFLEN–2)	-	-	-	-	-	-	-
M2	ror	32	(pos > 0) (pos >= 0)	-	x	x	x	x	-	-
M3	sdl	16	atCR = 0 *nothing*	-	-	-	-	-	-	-
M4	ssr	16	atCR = 0 pos = 0	-	-	-	-	-	-	-

X u tabeli znači da je mutan likvidiran odgovarajućim testom

Mutanti M1, M3 i M4 nisu likvidirani nijednim od testova. Za njih se kaže da su **živi**.

Testiranje mutacijom

- Mutant može ostati živ iz dva razloga:
  - Mutant se može razlikovati u odnosu na originalni program, ali skup testova ne sadrži odgovarajući test (tj. nije adekvatan u odnosu na mutanta)
    - U ovom slučaju skup testova treba poboljšati dodavanjem novih testova koji će ubiti ove mutante
  - Mutant se ne može razlikovati od originala bilo kojim zamislivim testom – mutant je **ekvivalentan** originalnom programu
    - U opštem slučaju problem određivanja da li je mutant ekvivalentan je neodlučiv

Primer

- Za dati primer, M1 je ekvivalentan originalnom programu
- M2 se može ubiti dodatnim testom
  Mixed: Mix of DOS and Unix line ends in the same file
- M3 za Mixed daje isti rezultat kao i originalni program, ali to je zato što originalni progam ima grešku (otkrila bi se analizom izlaza programa, a ne samo poređenjem programa i mutanta).
  - Kada se otkloni greška u programu, test Mixed ubija i M3

Adekvatnost serije testova T u odnosu na skup mutanata SM:

- Računa se kao procentualni odnos likvidiranih mutanata u odnosu na ukupan broj neekvivalentnih mutanata
- Za naš primer sa Mixed testom i nepopravljenim programom adekvatnost je 66% a kada se program ispravi adekvatnost je 100%

Jaka i slaba mutacija

- Prethodno opisani način, kada se program menja na tačno jednom mestu i kada se moraju nad njim izvršiti svi testovi, zove se **jaka mutacija.**
- Problem jake mutacije je u velikom broju testova, jer se jedan mutacioni operator mora primeniti na svaki iskaz programa, svi mutanti kompletno izvršavati za sve testove itd.

Testiranje slabom mutacijom

- Jedan program se zasejava mnogim defektima. Pravi se “metamutant” koji ima segmente koji odgovaraju i originalnom i izmenjenom kodu, sa mehanizmom biranja segmenata za izvršavanje.
- Dve kopije “metamutanta” se izvršavaju u tandemu, jedan sa izabranim originalnim kodom, a drugi sa mutiranim na osnovu selekcije iz skupa živih mutanata.

Testiranje slabom mutacijom

- Izvršavanje se pauzira posle svakog segmenta da se uporede stanja dva programa.
- Ako je stanje ekvivalentno, izvršavanje se nastavlja sa sledećim segmentom originalnog i mutiranog koda.
- Ako se stanja razlikuju, mutant se označava mrtvim i izvršavanje restartuje od početka sa novom selekcijom živih mutanata.

- U odnosu na jaku mutaciju, ne smanjuje se ukupan broj mutanata za razmatranje, ali se smanjuje broj prevođenja i izvršavanja testova.

Procenjivanje broja preostalih defekata

- Posejati dati broj N defekata u programu. Testirati program sa nekom serijom testova i izbrojati otkrivene greške. Podeliti na broj otkrivenih zasejanih defekata, DS i na broj otkrivenih „prirodnih“ defekata DN. Proceniti ukupan broj preostalih defekata u programu, koristeći formulu:

$$\frac{S}{\text{total natural faults}} = \frac{D_S}{D_N}$$

Pretpostavke analize

- Valjanost ove počiva na mnogim pretpostavkama:
  - Da su izabrane mutacije dobar model za prirodne defekte.
  - Da su defekti su uglavnom međusobno nezavisni
  - Da ne koristimo znanje o mutacionom procesu za projektovanje takvih testova koji su bolji u pronalaženju grešaka usled mutacije nego normalnih defekata).
  - Ako smo zainteresovani za izradu pouzdane procene pri veoma niskim brojevima preostalih grešaka potreban je veoma veliki broj mutanata.