Otkrivanje redudantnih test primera

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Una Stanković, Mirko Brkušanin, Miloš Samardžija
una_stankovic@yahoo.com, mirkobrkusanin94@gmail.com, miloss208@gmail.com

maj 2018.

Sažetak

U ovom tekstu je ukratko prikazan proces kreiranja metodologije za otkrivanje redudantnih test primera. Autori su u njemu izneli proces razmišljanja i testiranja sa ciljem dolaska do određenih zaključaka i potencijalnog rešenja problema.

Sadržaj

1 Uvod

Pronalaženje redudantnih test primera predstavlja veoma bitan deo testiranja softvera kojem do danas nije pridavan veliki značaj. Razlog koji stoji iza toga je što još uvek nije razvijena adekvatna metodologija koja bi programerima i testerima omogućila da bez dodatnog troška (posebno vremenskog) otkriju test primere koji su redudantni, a potom ih i uklone.

Značaj otkrivanja redudantnih test primera se posebno ističe u projektima otvorenog kôda u kojima svi učesnici u stvaranju istog (a može ih biti i nekoliko stotina) prilikom dodavanja novih delova kôda dodaju i nove test primere, bez ikakve provere da li takvi test primeri već postoje i da li se tim test primerima pokrivaju neki delovi koji su već pokriveni drugim test primerima. Iz tog razloga broj testova za određeni kôd može narasti do nerazumnih granica, što još više otežava rad nad softverom otvorenog kôda, kao i otkrivanje grešaka, propusta i slično.

U radu će biti iznet detaljan proces razmišljanja, testiranja i kreiranja metodologije za otkrivanje ovakvih test primera.

2 Definicija redudantnog testa

Da bismo mogli vršiti pronalaženje redudantnih testova u projektu prvo moramo odlučiti šta tačno čini neki test redudantnim. Nije lako odrediti kriterijume sličnosti neka dva testa pogotovo što oni nekada mogu biti i subjektivni. Neka opšta definicija bi mogla da glasi: test je redudantan u odnosu na drugi ako ispituje potpuno istu funkcionalnost programa i daje isti rezultat. Sada je potrebno precizirati šta podrazumevamo pod istom funkcinalnošću i istim rezultatom.

Ukoliko posmatramo funkciju koja na osnovu datuma određuje naredni dan ona će po potrebi vrišiti uvećanja meseca ili godine ukolko je ulaz poslednji dan u nekom mesecu ili godini. Postoje instrukcije koje se neće izvršiti prilikom svakog poziva funkcije i zbog toga će postojati testovi koji pokrivaju sve kritične delove kôda. Prema tome kada govorimo o istoj funkcionalnosti možemo zapravo govoriti o tome da li su izvršene iste naredbe u kôdu. Ovo nas navodi na ideju da proveru redudantnih testova možemo vršiti preko provere koje tačno naredbe kôda neki test izvršava. Ovakvo ispitivanje pokrivenosti kôda predstavlja poptuno objektivu meru. Međutim ovaj pristup provere redudantnosti sa sobom nosi određena ograničenja i poteškoće.

2.1 Ista pokrivenost kod različitih rezultata

Posmatrajmo funkciju koja ispituje da li je broj paran. Jedna vrlo kratka implementacija bi mogla biti sledeća:

```
1 int paran1(int n) {
2   return n % 2 == 0;
3 }
```

Za bilo koju ulaznu vrednost ove funkcije izvršiće se putpuno iste naredbe tj. izvršiće se ista i jedina linija kôda. Prema tome ukoliko već postoji test koji proverava navedenu funkciju za neku proizvoljnu vrednost onda će svaki drugi test biti redudantan u odnosu na već postojeći (ukoliko se ograničimo samo na pokrivenost ove funkcije). Tada će takav test samo ispitati da li se izvršila naredba računanja ostatka, a ne i

tumačenje samog rezultata pa se postavlja pitanje da li je takav test dovoljan. Ukoliko želimo da vidimo da li je rezultat smislen vršićemo proveru ne u samoj funkciji već nakon njenog poziva. Takva provera podrazumeva grananje koje daje odgovor da li je rezultat ispravan što izaziva pokrivenost razlčitih linija. Međutim ukoliko se naš način provere pokrivenosti ograničava samo na tu funkciju onda možemo dobiti i lažno pozitivan rezultat. Dobijanje iste pokrivenosti za različite rezultate predstavlja jedan od nedostataka ovog pristupa. Tipičan primer u kojem sa javlja ova situacija je prilkom provere za granične vrednosti parametara (0, INT_MAX, INT_MIN i sl.).

Ukoliko neki test pada onda postoji greška u kôdu koju je potrebno ispraviti pre same provere redudantnosti testova. U nastavku se podrazumeva pretpostavka da svi testovi prolaze i ne dovodi se u pitanje da li je pokrivenost testova potpuna već se samo vrši provera suvišnih testova među već postojećim.

2.2 Ista pokrivenost prilikom različitog broja iteracija

Još jedan od načina gde se može javiti ista pokrivenost kôda za različite slučajeve upotrebe je prilikom postojanja petlji u programu. Pitanje koje je ovde ključno je da li različit broj iteracija predstavlja ispitivanje različite funkcionalnosti jer ukoliko postoji greška u kôdu ona se neće nužno javiti prilikom prve ili nekoliko prvih iteracija. Jedan od načina rešavanja ovog problema bi mogao biti da različit broj iteracija posmatramo kao različitu pokrivenost. Međutim nekada i to ne mora biti dovoljno. Posmatrajmo sledeći primer:

Imamo funkciju koja kao parametar prima broj koji predstavlja broj iteracija petlje:

```
int f(int n) {
  for (int i=0; i<n; i++) {
    ...
  }
}
```

I program koji je koristi:

```
int main() {
  int a,b,c;
  scanf("%d %d %d", &a, &b, &c);
  f(a);
  f(b);
  f(c);
  return 0;
}
```

Za ulaze programa: **1 2 9** i **3 4 5** imamo potpuno istu pokrivenost čak i kada posmatramo broj izvršavanja. Sve linije u **main** funkciji će biti izvršene jednom a sve linije unutar petlje u funkciji **f** će biti izvršene po 12 puta. Naravno može se postaviti pitanje smislenosti ovakvih testova i da li ipak treba testirati funkciju f pojedinačno a ne po 3 puta ali u opštem slučaju problem pokrivenosti petlji nije lako rešiti. U ovom radu se petlje pojednostavljuju i samo se posmatra da li je neka linija kôda izvršena, a ne i koliko puta.

2.3 Poverenje u rezultat

Zbog navedenih nedostataka postavlja se pitanje da li ovakvim ispitivanjem redudantnosti smemo doneti zaključak da je neki test suvišan. Ukoliko ne poznajemo testove koje proveravamo onda ne možemo doneti preciznu odliku samo na osnovu rezultata predstavljenog postupka. Dodatak koji bi mogao pomoći je da imamo podatak o tome u odnosu na koje testove je neki test redudantan pa dalje možemo izvršiti ručnu proveru ili čak i proveru nekom dubljom analizom koji bi bilo potrebno tada razviti. U svakom slučaju možemo posmatrati rezultat na drugi način a to je da testove koji nisu proglašeni redudantnim sigurno treba zadržati.

3 Implementacija

U nastavku će biti predstavljeni detalji implementacije predstavljene ideje za rešavanje problema pronalaženja redudantnih testova.

Kao što je već pomenuto u sekciji 2, redudantni testovi su oni koji pokrivaju iste delove kôda. Prema tome problem se može svesti na prepoznavanje delova kôda koji neki test primer pokriva, a zatim pronaći drugi test koji pokriva isti kôd ili neki njegov nadskup. U tom slučaju će prvi test biti redudantan.

Ovo nas navodi na to da nam je potreban neki način praćenja naredbi u kôdu (ili blokova kôda) koji su izvršeni što možemo postići nekom instumentalizacijom kôda. Kao osnovna ideja se javlja izmena izvornog kôda koja će imati u sebi neku vrstu globanog niza flegova (eng. flags) koji se aktiviraju kada je neki kôd dostignut ili neki poseban uslov ispunjen. U svom najjednostavnijem obliku ti flegovi mogu postojati na početku svakog bloka kôda i na početku svake grane. Time je jednoznačno određena svaka putanja kroz kôd ne uzimajući u obzir višestruka izvršavanja tela petlji koje će biti posebno spomenute kasnije. Ako se ograničimo na to da su vrednosi flegova 0 ili 1, gde 0 predstavlja da kôd nije izvršen, a 1 da jeste, onda se provera da li su izvršene iste putanje svodi na poređenje niza flegova. Ukoliko su isti i testovi su isti. Provera da li je jedan test sadržan u drugom se takođe jednostavno proverava u linearnom vremenu po broju flegova. Dodatna pogodnost ograničavanja vrednosti na samo 0 ili 1 je da se može izvršiti optimizacija korišćenjem bitovskih operatora. Alternativno svaki fleg može biti poseban brojač koji se uvećava svaki put za jedan što umanjuje efikasnost, ali doprinosi tome da se razlikuju testovi koji imaju različit broj iteracija kroz istu petlju.

U nastavku sledi primer kôda koji ilustruje ideju dodavanja fleg-ova.

```
int abs(int x)
{
  int value;
  if (x>0)
  {
    value = x;
  }
  else
  {
    value = -x;
  }
  return value;
}
```

```
extern int flags[3]; //koji se inicijalizuje na 0

int abs(int x)
{
  flags[0] = 1;
  int value;
  if (x>0)
  {
    flags[1] = 1;
    value = x;
  }
  else
  {
    flags[2] = 1;
    value = -x;
  }
  return value;
}
```

Za ovakvu izmenu kôda se može koristi Clang tačnije njegova biblioteka LibTooling uz pomoću koje je moguće obići AST stablo i izvršiti „source to source" izmene. Drugi način bi bio korišćenje nekog alata koji prati izvršene naredbe kao što je gcov, koji dolazi sa GCC prevodiocem. Ovaj pristup se može pokazati manje efikasnim zbog velikog broja redudantnih informacija koje nudi. Ukoliko se radi pod pretpostavkom da su svi testovi ispravni, gcov će nuditi po jedan broj za svaku liniju kôda, gde može biti dovoljan samo jedan po bloku. Provera da li je jedan test redudantan u odnosu na drugi će i dalje biti linearna ali ovoga puta po broju linija kôda, a ne po broju osnovnih blokova[1]

Dok je provera između dva testa linearna to nije slučaj ukoliko se vrši kompletna provera svih testova. Ukoliko želimo da nađemo najmanji skup testova koji vrši isto pokrivanje kao i ceo skup onda nailazimo na problem minimalnog pokrivanja skupa. Ovo je poznati optimizacioni problem za koji je dokazano da je NP kompletan. Naš problem se jednostavno svodi na problem pokrivanja skupa. Skup koji pokrivamo je celokupni niz flegova. Svaki fleg je jedan element, a svaki test je jedan podskup koji sadrži samo one elemente koji odgovaraju aktiviranim flegovima za taj test. Problem pokrivanja skupa ima i približna rešenja koja se mogu iskoristiti zarad bolje vremenske efikasnosti.

U okviru projekta su implementirana tri algoritma za pokrivanje skupa: optimalni, pohlepni i algoritam provere redudantnosti koji proverava samo parove testova. Alat koristi samo optimalni, a ostali algoritmi pred-

[1] Osnovni blok je niz instrukcija u kome će se sve instrukcije izvršiti od početka do kraja, nema labela (može samo prva), nema skoka i slično.

stavljaju ostatak procesa razvijanja rešenja, i upotrebljavani su privremeno, dok nije implementirana optimalnija verzija. Takođe je razvijen alat "with_test_info" koji je služio za proveru ispravnosti algoritama. Alat se može koristiti za proveru pokrivenosti kôda ali se program mora pokrenuti u potpunosti. U direktorijumu mora postojati tekstualna datoteka sa nazivom "test_info.run" koja opisuje način pokretanja programa. U datoteci se unose tri linije za svaki test koje redom predstavljaju: naziv testa, komanda kojom se pokreće program (najčešće "./test") i ulaz programa (koji može biti prazna linija).

4 Primeri

U ovoj sekciji će biti predstavljena i prodiskutovana dva kompletna slučaja upotrebe alata, na primeru sa našim jednostavnim frejmvorkom (eng. framework) za jediničnо testiranje[2]. Primer koji je opisan u nastavku se nalazi na "with_unit_test_framework/examples/03_false_positives", i sadrži dve jednostavne funkcije za testiranje parnosti broja. Prva funkcija sadrži samo jedan izraz koji testira parnost, i on ujedno predstavlja i povratnu vrednost funkcije. Druga funkcija ima istu funkcionalnost, ali koristi grananja.

```
//library.c

int paran1(int n) {
    return n % 2 == 0;
}

int paran2(int n) {
    if(n % 2 == 0)
        return 1;
    return 0;
}
```

Odgovarajući skupovi testova (eng. test cases) koji sadrže jedinične testove za testiranje funkcionalnosti ovih funkcija:

```
//paran1_test_case.c

TST(test_neparan)
BEGIN
  COMPARE(paran1(11) == 0);
END

TST(test_paran)
BEGIN
  COMPARE(paran1(16) == 1);
END

//paran2_test_case.c

TST(test_neparan)
BEGIN
  COMPARE(paran2(11) == 0);
END

TST(test_paran)
BEGIN
```

[2]Implementacija je jednostavna, ali neće biti opisana, s obzirom da fokus projekta nije na tome.

```
22   COMPARE(paran2(16) == 1);
23 END
```

Pokretanjem alata se dobija izlaz koji obaveštava da test paran1_test_case.test_paran predstavlja redudantan test. Ovaj primer je odabran jer predstavlja slučaj u kojem je rezultat lažno-pozitivan[3]. Razlog za to je što oba testa izvršavaju isti skup linija (u ovom slučaju je to jedna linija), i alat jedan od njih klasifikuje kao redudantan. Međutim, sasvim je opravdano da se oba testa nađu među jediničnim testovima, jer je logično da se testiraju oba ishoda (broj je paran/broj nije paran).

U nastavku je prikazan primer "with_unit_test_framework/examples/01"koji predstavlja jednostavnu biblioteku sa 4 osnovne operacije:

```
1  //library.c
2
3  int sabiranje(int a, int b) {
4    return a + b;
5  }
6
7  int oduzimanje(int a, int b) {
8    return a - b;
9  }
10
11 int mnozenje(int a, int b) {
12   return a * b;
13 }
14
15 int deljenje(int a, int b) {
16   if(b != 0)
17     return a / b;
18   return ERR;
19 }
```

I odgovarajući testovi:

```
1  //tst_01.c
2
3  TST(test_sabiranje)
4  BEGIN
5    COMPARE(sabiranje(10, 5) == 15);
6  END
7
8  TST(test_oduzimanje)
9  BEGIN
10   COMPARE(oduzimanje(10, -5) == 15);
11 END
12
13 TST(test_deljenje_nulom)
14 BEGIN
15   COMPARE(deljenje(5, 0) == ERR);
16 END
17
18 //tst_02.c
19
20 TST(test_mnozenje)
21 BEGIN
22   COMPARE(mnozenje(10, 5) == 50)
23 END
24
25 //tst_03.c
```

[3]Ovo je zapravo diskutabilno, i zavisi od toga da li se prilikom pisanja testova prati samo procenat pokrivenih grana ili ne.

```
26
27 TST(test_deljenje1)
28 BEGIN
29   COMPARE(deljenje(10, 5) == 2);
30 END
31
32 TST(test_deljenje2)
33 BEGIN
34   COMPARE(deljenje(10, 0) == ERR);
35 END
```

Pokretanjem alata, dobijamo da je test tst_03.test_deljenje2 redudantan. U ovom slučaju je rezultat sasvim opravdan. Vidimo da ovaj, i test tst_01.test_deljenje_nulom testiraju u potpunosti istu funkcionalnost.

5 Alati

U ovoj sekciji će biti predstavljeni alati korišćeni za analizu.

Kako bismo dobili informaciju o izvršenim linijama u izvornom kôdu prilikom pokretanja programa, koristimo alat gcov. Prvo je potrebno prevesti odgovarajući .c fajl sa narednim opcijama:

```
gcc -g -Wall -fprofile-arcs -ftest-coverage -O0 test.c -o test
```

Zatim se pokreće program **test** koji generiše .gcda fajl sa informacijama o izvršenim linijama. Iz tog fajla pomoću alata gcov dobijamo .gcov tekstualni fajl.

```
gcov test.gcda
```

Primer dobijenog .gcov fajla.

```
1     -:    2:#include <stdio.h>
2     -:    3:
3     1:    4:int main( void )
4     -:    5:{
5     -:    6:    int a, b, c, d;
6     -:    7:    int largest, smallest;
7     -:    8:
8     1:    9:    printf( "Enter four integers (separate them
        with spaces): " );
9     1:   10:    scanf( "%d %d %d %d", &a, &b, &c, &d );
10    -:   11:
11    1:   12:    largest = smallest = a;
12    -:   13:
13    1:   14:    if ( largest < b ){
14    1:   15:        largest = b;
15    -:   16:    }
16####:   17:    else if ( b < smallest ){
17####:   18:        smallest = b;
18    -:   19:    }
19    1:   20:    if ( largest < c ){
20    1:   21:        largest = c;
21    -:   22:    }
22####:   23:    else if ( c < smallest ){
23####:   24:        smallest = c;
24    -:   25:    }
25    1:   26:    if ( largest < d ){
26    1:   27:        largest = d;
27    -:   28:    }
28####:   29:    else if ( d < smallest ){
29####:   30:        smallest = d;
```

```
30  -:   31:    }
31  -:   32:
32  1:   33:    printf( "\nLargest: %d\n", largest );
33  1:   34:    printf( "Smallest: %d", smallest );
34  -:   35:
35  1:   36:    return 0;
36  -:   37:}
```

Lako se uočava da imamo '1' za svaku granu programa koja je izvršena, ako je program dobro struktuiran. Ako se u neku granu nije ušlo to je označeno nizom znakova '#'.

Neophodno je izvršiti neku vrstu parsiranja .gcov fajla kako bismo izvukli informacije o izvršenim linijama. Ono što želimo da dobijemo je niz celobrojnih vrednosti koji dalje možemo da poredimo sa drugim izvršavanjima istog fajla. Ako bismo za dva test primera imali iste nizove karaktera u fajlu, odnosno isti skup izvršenih grana za njih, onda smatramo da pokrivaju iste grane.

Postoji par detalja koje smo izostavili. Naime, gcov samo daje informaciju da li je linija izvršena, a ako u jednoj liniji postoji više od jedne naredbe, nismo sigurni koja je tačno naredba izvršena (sve ili samo jedna). Zbog toga je prvo potrebno izvršiti neku vrstu formatiranja izvornog kôda. To možemo postići nekim dodatnim alatom kao što je **clang-formater**.

6 Zaključak

Na osnovu postignutih rezultata, može se primetiti da je ovaj problem izuzetno težak, i da ne postoji jedinstveno i najbolje (u smislu uspešnosti klasifikacije testova kao redudantnih/neredudantnih) automatizovano rešenje. Problem je moguće rešiti delimično, uz moguće greške poput lažno-pozitivnih klasifikovanja testova. Procenat lažno-pozitivnih klasifikovanja se može umanjiti detaljnijom analizom petlji/rekurzije, ali se ne može u potpunosti odstraniti, jer je sam pojam redudantnih testova nedovoljno jasno definisan, i nešto što za nekoga predstavlja redudantnost, za drugu osobu može predstavljati sasvim regularan i opravdan test. Dakle, u samu definiciju je uključena i subjektivnost, zbog čega se zaključuje da se ovakav tip alata ne može usavršiti do te mere da se greške uopšte ne javljaju.

Rezultat dobijen našim alatom (metodologijom) treba uzeti sa dozom opreza, i potrebno je vršiti dodatnu analizu testova kako bi se utvrdilo da li neki od njih predstavlja lažno-pozitivno upozorenje. Lažno-pozitivni rezultati mogu se javiti u slučaju petlji i rekurzivnih poziva funkcija (i ne isključivo samo u tim situacijama), pošto metodologija uzima u obzir da li je neka linija kôda izvršena, ali ne i to koliko je puta izvršena. To može predstavljati problem kod testova koji testiraju granične vrednosti. Mogu se izvesti dodatna poboljšanja preciznosti detaljnijom analizom petlji.

7 Moguća unapređenja

Neki od mogućih načina za poboljšanje preciznosti i kvaliteta samog alata (i metodologije) su detaljnija analiza petlji i poziva funkcija, kao i detektovanje koji to testovi pokrivaju testove koji su proglašeni redudantnim (videti sekciju 2). Takođe, istraživanjem novih tehnika za rešavanje problema redudantnosti testova se potencijalno može dobiti mnogo pre-

cizniji i efikasniji alat (npr. tehnike koje ne podrazumevaju pokretanje testova).