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

Redudantni test primeri su oni test primeri koji pokrivaju iste delove koda ili pokrivaju delimično iste delove koda, tj. njihovo pokrivanje se preklapa.

Pronalaženje redudantnih test primera predstavlja veoma bitan deo testiranja softvera kojem do danas nije pridodavan 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 koda u kojima svi učesnici u stvaranju istog (a može ih biti i nekoliko stotina) prilikom dodavanja novih delova koda 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 koda koji su već pokriveni drugim test primerima. Iz tog razloga broj testova za određeni kod može narasti do nerazumnih granica, što još više otežava rad nad softverom otvorenog koda, 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 Ideje

U nastavku će biti predstavljene neke ideje za rešavalje problema pronalaženja redudantnih testova.

2.1 Poređenje testova pomoću pokrivenosti koda

Kao što je već pomenuto u sekciji 1 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 kôoda 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.

```
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 drugog 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 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 koda, 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.

[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

3 Primeri

U ovoj sekciji će biti predstavljena i prodiskutovana dva kompletna slučaja upotrebe alata, na primeru sa custom unit testing frejmvorkom. Primer koji je opisan u nastavku se nalazi na "v2/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 test case-ovi 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
  COMPARE(paran2(16) == 1);
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. 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 "v2/examples/01"koji predstavlja jednostavnu biblioteku sa 4 osnovne operacije:

```c
//library.c

int sabiranje(int a, int b) {
  return a + b;
}

int oduzimanje(int a, int b) {
  return a - b;
}

int mnozenje(int a, int b) {
  return a * b;
}

int deljenje(int a, int b) {
  if(b != 0)
    return a / b;
  return ERR;
}
```

I odgovarajući testovi:

```c
//tst_01.c

TST(test_sabiranje)
BEGIN
  COMPARE(sabiranje(10, 5) == 15);
END

TST(test_oduzimanje)
BEGIN
  COMPARE(oduzimanje(10, -5) == 15);
END

TST(test_deljenje_nulom)
BEGIN
  COMPARE(deljenje(5, 0) == ERR);
END

//tst_02.c

TST(test_mnozenje)
BEGIN
  COMPARE(mnozenje(10, 5) == 50)
END

//tst_03.c

TST(test_deljenje1)
BEGIN
  COMPARE(deljenje(10, 5) == 2);
END

TST(test_deljenje2)
BEGIN
  COMPARE(deljenje(10, 0) == ERR);
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.

4 Alati

U ovoj sekciji će biti predstavljeni alati korišćeni za analizu.

Kako bi dobili informacija o izvršenim linijama u izvornom kodu prilikom pokretanja pograma koristimo alata 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.

```
  -:    2:#include <stdio.h>
  -:    3:
  1:    4:int main( void )
  -:    5:{
  -:    6:    int a, b, c, d;
  -:    7:    int largest, smallest;
  -:    8:
  1:    9:    printf( "Enter four integers (separate them
    with spaces): " );
  1:   10:    scanf( "%d %d %d %d", &a, &b, &c, &d );
  -:   11:
  1:   12:    largest = smallest = a;
  -:   13:
  1:   14:    if ( largest < b ){
  1:   15:        largest = b;
  -:   16:    }
#####:   17:    else if ( b < smallest ){
#####:   18:        smallest = b;
  -:   19:    }
  1:   20:    if ( largest < c ){
  1:   21:        largest = c;
  -:   22:    }
#####:   23:    else if ( c < smallest ){
#####:   24:        smallest = c;
  -:   25:    }
  1:   26:    if ( largest < d ){
  1:   27:        largest = d;
  -:   28:    }
#####:   29:    else if ( d < smallest ){
#####:   30:        smallest = d;
  -:   31:    }
  -:   32:
  1:   33:    printf( "\nLargest: %d\n", largest );
  1:   34:    printf( "Smallest: %d", smallest );
  -:   35:
  1:   36:    return 0;
  -:   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 taraba.

Neophodno je izvršiti neku vrstu parsiranja .gcov fajla kako bi 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, ako u jednoj liniji postoji više od jedne naredbe nismo sigurni koja je tačno naredba je izvršena (sve ili samo jedna). Zbog toga je prvo potrebno izvršiti neku vrstu formatiranja izvornog koda. To možemo postići nekim dodatnim alatom kao što je **clang-formater**.

5 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 koda 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.