

# Otkrivanje redundantnih 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 redundantnih test primera. Autori su u njemu izneli proces razmišljanja i testiranja sa ciljem dolaska do određenih zaključaka i potencijalnog rešenja problema.

## Sadržaj

<b>1</b>	<b>Uvod</b>	<b>2</b>
<b>2</b>	<b>Ideje</b>	<b>2</b>
2.1	Poređenje testova pomoću pokrivenosti koda . . . . .	2
<b>3</b>	<b>Eksperimenti</b>	<b>4</b>
3.1	Jednostavni primeri . . . . .	4
3.1.1	Maksimum i minimum 4 broja . . . . .	4
<b>4</b>	<b>Alati</b>	<b>5</b>
<b>5</b>	<b>LLVM</b>	<b>6</b>
<b>6</b>	<b>Zaključak</b>	<b>6</b>
	<b>Literatura</b>	<b>6</b>

# 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 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, sa posebnim osvrtom na alat llc u okviru LLVM-a.

## 2 Ideje

U nastavku će biti predstavljene neke ideje za rešavanje 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 instrumentalizacijom kôda. Kao osnovna ideja se javlja izmena izvornog kôda koja će imati u sebi neku vrstu globalnog 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 vrednosti flegova 0 ili 1, gde 0 predstavlja da kôd nije izvršen, a 1 da jeste, onda se proverava 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.

```
1 int abs(int x)
2 {
3     int value;
4     if (x>0)
5     {
6         value = x;
7     }
8     else
9     {
10        value = -x;
11    }
12    return value;
13 }
```

```
1 extern int flags[3]; //koji se inicijalizuje na 0
2
3 int abs(int x)
4 {
5     flags[0] = 1;
6     int value;
7     if (x>0)
8     {
9         flags[1] = 1;
10        value = x;
11    }
12    else
13    {
14        flags[2] = 1;
15        value = -x;
16    }
17    return value;
18 }
```

Za ovakvu izmenu kôda se može koristiti 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 previociem. Ovaj pristup se može pokazati manje efikasnim zbog velikog broja redundantnih 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 redundantan u odnosu na drugi će i dalje biti linearna ali ovoga puta po broju linija koda, a ne po broju osnovnih blokova<sup>1</sup>

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 kompletna. 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

---

<sup>1</sup>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

bolje vremenske efikasnosti.

## 3 Eksperimenti

U ovoj sekciji će biti izneti osnovni eksperimenti koje smo vršili kako bismo došli do određenih zaključaka.

### 3.1 Jednostavni primeri

Najpre, pokušaćemo sa dodavanjem nekoliko jednostavnih primera, koji su, pre svega, jednostavni za analizu, a potom ćemo posmatrati ponašanje primenom različitih test primera, od kojih će neki biti redundantni.

#### 3.1.1 Maksimum i minimum 4 broja

Prvi primer koji posmatramo je otkrivanje maksimuma i minimuma 4 broja. Kod je napisan u programskom jeziku C.

```
1 //This is just a simple example on which we should try to
   think about all the test examples and observe the
   results for them
2 #include <stdio.h>
3
4 int main( void )
5 {
6     int a, b, c, d;
7     int largest, smallest;
8
9     printf( "Enter four integers (separate them with
   spaces): " );
10    scanf( "%d %d %d %d", &a, &b, &c, &d );
11
12    largest = smallest = a;
13
14    if ( largest < b ){
15        largest = b;
16    }
17    else if ( b < smallest ){
18        smallest = b;
19    }
20    if ( largest < c ){
21        largest = c;
22    }
23    else if ( c < smallest ){
24        smallest = c;
25    }
26    if ( largest < d ){
27        largest = d;
28    }
29    else if ( d < smallest ){
30        smallest = d;
31    }
32
33    printf( "\nLargest: %d\n", largest );
34    printf( "Smallest: %d", smallest );
35
```

```

36     return 0;
37 }

```

Sada, kreirajmo neke test primere:

Ulaz: 4 3 2 1

Izlaz: 4 1

## 4 Alati

U ovoj sekciji će biti predstavljeni alati korišćeni za analizu.

Kako bi dobili informacija o izvršenim linijama u izvornom kodu prilikom pokretanja programa 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.

```

1      -:      0:Source:1_max4.c
2      -:      0:Graph:1_max4.gcno
3      -:      0:Data:1_max4.gcda
4      -:      0:Runs:1
5      -:      0:Programs:1
6      -:      1://This is just a simple example on which
we should try to think about all the test examples
and observe the results for them
7      -:      2:#include <stdio.h>
8      -:      3:
9      1:      4:int main( void )
10     -:      5:{
11     -:      6:      int a, b, c, d;
12     -:      7:      int largest, smallest;
13     -:      8:
14     1:      9:      printf( "Enter four integers (
separate them with spaces): " );
15     1:     10:      scanf( "%d %d %d %d", &a, &b, &c, &d
);
16     -:     11:
17     1:     12:      largest = smallest = a;
18     -:     13:
19     1:     14:      if ( largest < b ){
20     1:     15:          largest = b;
21     -:     16:      }
22     #####: 17:      else if ( b < smallest ){
23     #####: 18:          smallest = b;
24     -:     19:      }
25     1:     20:      if ( largest < c ){
26     1:     21:          largest = c;
27     -:     22:      }
28     #####: 23:      else if ( c < smallest ){
29     #####: 24:          smallest = c;
30     -:     25:      }
31     1:     26:      if ( largest < d ){
32     1:     27:          largest = d;
33     -:     28:      }

```

```

34 #####: 29:     else if ( d < smallest ){
35 #####: 30:         smallest = d;
36 -:      31:     }
37 -:      32:
38 1:      33:     printf( "\nLargest: %d\n", largest );
39 1:      34:     printf( "Smallest: %d", smallest );
40 -:      35:
41 1:      36:     return 0;
42 -:      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 LLVM

## 6 Zaključak