Otkrivanje redundantnih test primera pomoću pokrivenosti koda

Seminarski rad u okviru kursa Verifikacija softvera
Matematički fakultet

Rastko Đorđević, Nikola Dimitrijević,
Dimitrije Špadijer, Nadežda Bogdanović
mi14078@alas.matf.bg.ac.rs, nikoladim95@gmail.com,
mm11021@alas.matf.bg.ac.rs, nadezdabogdanovic1@gmail.com

17. april 2019

Sažetak

Veliki problem, naročito kod softvera otvorenog koda i regresionog testiranja, predstavlja gomilanje redundantnih test primera i način njihovog otkrivanja. Naša definicija redundantnog test primera podrazumeva test primer čijim uklanjanjem iz skupa test primera pokrivenost linija koda ostaje neizmenjena. Pokrivenost koda merili smo alatom *gcov* i fokusirali smo se na test primere za QT framework. Cilj rada je da opiše našu implementaciju otkrivanja redundantnih test primera.

Sadržaj

1 Opis problema

Kod velikih projekata, primećuje se nagomilavanje test primera, u čijem se međusobnom odnosu često primećuje visoka stopa redundantnosti. Međusobna redundantnost dva test primera ukazuje na to da se svaka greška, koja se može naći jednim test primerom, može naći i sa drugim i obrnuto. Ovakva pojava ima ogromnog uticaja na efikasnost, vremensku i prostornu, kao i na troškove samog testiranja.

Kako zbog politike vođenja samog projekta postoji velika sloboda dodavanja, a mala, ili gotovo nikakva provera prilikom dodavanja novih test primera, ova pojava se posebno zapaža kod softvera otvorenog koda.

Ovakvome je softveru, zbog svoje prirode, potrebno dosta vremena kako bi stigao do završne faze svog razvojnog ciklusa, tako da je potrebno koliko god je moguće smanjiti trajanje svake od faza, a samim tim i testiranje.

2 Opis rešenja

Oslanjajući se na alat *gcov* za merenje pokrivenosti koda, projekat koji smo odabrali kao prezentujući, pokrećemo najpre sa njegovim test primerima i merimo pokrivenost linija. Zatim uklanjamo jedan po jedan test primer, tako što ga jednostavno zakomentarišemo i merimo pokrivenost bez njega i ako ona ostaje nepromenjena, onda taj test primer označavamo kao redundantan.

Kao kriterijum pokrivenosti koristimo pokrivenost linija koda, zbog svoje intuitivne i prirodne interpretacije. Alat *gcov* koristimo zato što je deo GNU ekosistema i kompatibilan je sa njegovim *gcc* kompajlerom.

Da bismo mogli da uklonimo test primer iz skupa test primera, moramo najpre da ga pronađemo. Za te potrebe smo razvili parser i konteksno slobodnu gramatiku, koja je prilagodljiva različitim načinima generisanja test primera u okviru QT framework-a.

Odabrali smo testove generisane u okviru QT framework-a zbog njegove iskoristivosti na različitim platformama, a i zato što su naši članovi tima već upoznati za rad na takvoj platformi i njeno korišćenje im je udobno.

2.1 Pseudokod osnovne ideje

```
getCoverage()
    compile project
    run tests
    generate coverage
    parse it and return metric

main()
    tests = {}

    for file in directory
        for test in file
            tests.add(test)

    baseline = getCoverage()

    for test in tests
        coverage = getCoverage()
        if coverage < baseline
            test.isredundant = false
        else
            test.isredundant = true
```

```
1022    print tests.filter(isredundant)
```

3 Opis arhitekture sistema

Osnovne komponente našeg rešenja objedinjene su u glavnom C++ fajlu, koji njima upravlja:

- Komponenta za pronalaženje svih test primera i njihovih lokacija:
  Svrha ovog modula je da pronađe sve test primere u projektu i njihove lokacije, kako bi se mogli zakomentarisati, a počiva na integraciji C++ sa *Flex-om* (leksički analizator) i *Yacc-om* (parser). Ovo je moguće zato što QT testovi moraju da se definišu na određeni način, tj. koriste određene ključne reči. Testove se traže rekurzivnim obilaskom direktorijuma u kojem se nalazi projekat koji se testira.
- Komponenta za komentarisanje i dekomentarisanje:
  Implementirana je kao bash skripta, zbog svoje konciznosti
- Komponenta za komunikaciju:
  Pisana je koristeći programski jezik Python i obuhvata sledeće akcije:
  - Kompajliranje projekta koji se testira
  - Izvršavanje kreiranih izvršnih fajlova
  - Izvršavanje alata gcov, kako bi se izračunala pokrivenost
  - Parsiranje izveštaja koji je kreirao gcov alat
  - Vraćanje parsiranog rezultata glavnom programu
- Komponenta koji pronalazi redundantnost:
  Vrši analizu i na osnovu prethodnodno dobijenih izveštaja i utvrđuje za svaki test primer da li je redundantan u odnosu na ceo skup test primera

4 Pokretanje, rezultati i njihova procena

Program se pokreće iz terminala. Program ispisuje lokaciju projekta za čije se testove proverava redundantnost i osnovnu pokrivenost koda. Ako utvrdi da je neki test redundantan, ispisuje se ime testa i pokrivenost koda bez tog testa.

4.1 Pokretanje i rezultati

Prilikom pokretanja programa za projekat Haming, koji računa rastojanje između dve niske, dobijena je osnovna pokrivenost linija 0.847, procenat naiđenih grana 0.775 i procenat grana čiji uslov je zadovoljen 0.4875. Program je pronašao svih sedam testova, od njih je pet proglašeno redundantnim. U tabeli 1 su prikazani rezultati.

U testiranom projektu su dva testa redunantna, ali je njih 71% klasifikovano kao redundantno. Razlog za to je što ostali testovi zajedno pokrivaju udeo grana i linija koji pokriva i jedan test koji se razmatra. Njegovim izbacivanjem i dalje je isti procenat zadovoljenih i naiđenih grana i program zaključuje da je on redundantan, iako svaki od tih testova unosi dodatnu moć otkrivanja grešaka pošto prolaze različitim putanjama kroz kod.

	Stvarno redundantani	Stvarno neredundantani
Predviđeno redundantni	2	3
Predviđeno neredundantni	0	2

Tabela 1: Matrica konfuzije redundantnosti testova

4.2 Prednosti

- Korišćenje je potpuno besplatno
- Pristup i tumačenje rezultata su intuitivni
- Nema lažno negativnih (eng. *false negative*) rezultata: Ako neki test primer nije označen kao negativan, onda to svakako nije

4.3 Mane

- Mnogo lažno pozitivnih (eng. *false positive*) rezultata: Ako je neki test primer označen kao redundantan, ne znači nužno da to i jeste. Ovakav problem se javlja zato što se kao kriterijum pokrivenosti koristi pokrivenost linija.
- Nizak nivo skalabilnosti: Minimalan skup test primera (onaj koji ne sadrži redundantnosti) se računa iznova i iznova za svako pokretanje. Ovo je posebno neefikasno za velike projekte.
- Velika zavisnost od izbora programskog jezika (podržava samo C++ projekte) i načina testiranja (podržava samo QT testove)

5 Alternativna rešenja

Kao što je već pomenuto, rešenje koje smo odabrali, iako intuitivno, ima dosta mana. One se mogu prevazići korišćenjem nekoliko različitih pristupa, od kojih smo mi odabrali dva najinteresantnija i prikazali ih u sekcijama koje slede.

5.1 Otkrivanje redundantnih test primera generisanih rešavačem modela primenom pokrivenosti koda

Ovaj pristup[1] podrazumeva da redundantnost znači da nema grešaka koje se mogu detektovati određenim test primerom, a ne bez njega. Umesto da se redundantni testovi odbace, oni se transformišu tako da se redundantnost izbegne. Ovakav pristup nas oslobađa lažnih rezultata, a da pritom ne gubimo na pokrivenosti koda. Za generisanje, kao i za optimizaciju test primera koristi se proveravač modela (eng. *m*odel-checker).

Da bismo utvrdili šta je test primer, najpre moramo predstaviti pojam putanje.

Definicija 5.1 *Putanja: Putanja* $p := \{s_0, s_1, ...\}$ *Kripke strukture* K *je konačna ili beskonačna sekvenca takva da* $\forall i > 0 : (s_i, s_{i+1}) \in TzaK$, *gde su* s_i *stanja modela.*

Definicija 5.2 *Test primer (eng. test-case): Test primer* t *je konačan prefiks putanje* p *Kripke strukture* K.

Dužina test primera: $length(t) := i$ predstavlja broj svih stanja sa kojima test primer dolazi u dodir.

Definicija 5.3 *Skup testova: Skup testova TS je konačan skup od n test primera. Veličina skupa testova TS je n. Dužina skupa testova je zbir svih dužina njegovih test primera: $length(TS) = \sum_{i=1}^{n} length(t_i)$*

Kao mera kvaliteta skupa test primera koristi se kriterijum pokrivenosti modela. Takav kriterijum se može predstaviti kao skup osobina, gde test primer pokriva određeni aspekt ako je odgovarajuća osobina narušena.

Definicija 5.4 *Pokrivenost testom: Pokrivenost C skupa testova TS, predstavljena skupom svojstava P je definisana kao odnos pokrivenog svojstva i ukupnog broja svojstava:*

$$C = \frac{1}{|P|}|x|x \in P \wedge covered(x, TS)| \tag{1}$$

Predikat $covered(a, TS)$ je tačan ako postoji test primer $t \in TS$ takav da t pokriva a.

Da bismo utvrdili da li su testovi redundantni, moramo najpre znati kako se izvršavaju:

Definicija 5.5 *Izvršavanje test primera: Test primer t za Kripke strukturu K se izvršava uzimanjem ulaznih varijabli svakog stanja, prosleđujući ih SUT-u sa odgovarajućim test-okvirom. Ove vrednosti i dobijene izlazne vrednosti predstavljaju putanju izvršavanja $tr = \{s_i^{'}, s_{i+1}^{'}, ...\}$. Greška je detektovana akko $\exists(s_i^{'}, s_{i+1}^{'}) \in tr : (s_i^{'}, s_{i+1}^{'}) \notin TzaK$, gde se $(s_i^{'}, s_{i+1}^{'})$ smatra za korak.*

5.1.1 Redundantnost u skupu testova

Intuitivno, redundantni test primeri su identični. Za svaka dva test primera t_1, t_2, takvi da je $t_1 = t_2$, svaka greška koja se može otkriti sa t_1, može se istovremeno otkriti i sa t_2 i obrnuto, pretpostavljajući da se oba test primera izvršavaju sa istim preduslovima. Takođe, pokrivenost koju daje pokretanje test primera t_1 jednako je pokrivenosti koju daje pokretanje t_2. Dolazimo do zaključka da skup testova nema potrebe za oba test primera.

Do sličnog zapažanja se dolazi i u slučajevima kada je t_1 prefiks test primera t_1, tako da se svaka greška koja se može detektovati sa t_1, može detektovati i sa t_2, ali ne i obrnuto. U ovom slučaju je t_1 redundantan i nije neophodan u onom skupu testova koji sadrži t_2.

Sve ovo prethodno dovelo nas je do redundantnosti za koju smo mi zainteresovani: Skup testova generisan pomoću rešavača modela je najčešće takav da svi test primeri počinju istim inicijalnim stanjem. Počev od ovog stanja, može se prolaziti različitim putanjama, ali je mnogo putanja jednako nekom određenom stanju. Svaka greška koja se desi prilikom izvršavanja neke od ovih potputanji može biti detektovana bilo kojim test primerom koji počinje ovom potputanjom. U sklopu ovih test primera, potputanja je **redundantna**.

Ovakav pogled na redundantnost nam omogućava da skup test primera predstavimo pomoću drveta, gde je inicijalno stanje, deljeno od strane svih test primera koren drveta. Potputanja je redundantna ako se javlja u sklopu dva ili više test primera. To znači da svaki čvor, osim inicijalnog, koji ima više od jedog deteta, sadrži redundantnost. Ako ima

više različitih inicijalnih stanja, onda za svako od tih inicijalnih stanja postostoji po jedno drvo.

Dubina drveta jednaka je dužini najdužeg test primera. Drvo izvršavanja se može koristiti za merenje redundantnosti.

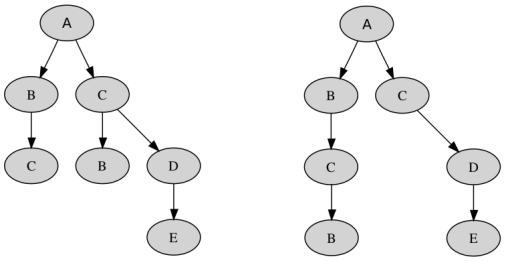
Definicija 5.6 *Drvo izvršavanja skupa test primera: Test primer* $t_i = \{s_0, s_1, ..., s_l\}$ *iz skupa test primera* TS *se može pretstaviti kao drvo, gde je koren jednak inicijalnom stanju zajedničkom svim test primerima:* $root(TS) = s_0$. *Za svaka dva uzastopna i različita stanja* s_i *i* s_j, *gde* s_i *prethodi stanju* s_j, *čvoru* s_i *dodaje se dete* s_j *na sledeći način:* $s_j : (s_i, s_j) \in t_i \rightarrow s_j \in children(s_i)$.

Definicija 5.7 *Redundantnost skupa test primera: redundantnost* R *skupa test primera* TS *je definisana pomoću drveta izvršavanja:*

$$R(TS) = \frac{1}{n-1} \cdot \sum_{x \in children(root(TS))} \Re(x) \tag{2}$$

redundantna vrednost $\Re$ *za decu inicijalnog čvora se računa rekurzivnom formulom:*

$$\begin{cases} (|children(x) - 1|) + \sum_{c \in children(x)} \Re(c), & children(x) \neq \{\} \\ 0, & children(x) = \{\} \end{cases}$$



Slika 1: Drvo sa redundantnošću od 17% i drvo bez redundantnosti

Na slici 1 prikazan je izgled drveta sa i bez redundantnosti. Postupak računanja procenta redundantnosti koristi formulu 2: $R = \frac{1}{7-1} \cdot (0 + (1 + 0)) = \frac{1}{6} = 17\%$.

Transformisanje test primera iskače van domena ovog rada, pa ovde neće biti obrađeno, ali se detaljno objašnjenje može naći u literaturi[1].

5.2 Otkrivanje redundantnosti test primera analizom pokrivenosti koda

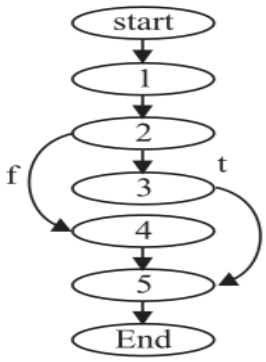
Ovaj pristup[2] podrazumeva sledeće korake:

1. **Nalaženje dijagrama toka:**
   Za program koji je pod testom, pronalazi se dijagram toka (eng. *control flow graph* - CFG). To znači da određeni testovi pobuđuju izvršavanje određenih delova koda, na osnovu kojih se formira CFG.

   Na slici 2a prikazan je program i za njega CFG, prikazan na slici 2b

```
1. main()
1. {
1. Integer a, b, c;
2. Input a, b;
2. if (a>b)
3. then c=a*a;
4. else c=b*b;
5. Output c; }
6. End
```

(a) Program



(b) CFG

Slika 2: Program i njegov dijagram toka

2. **Pronalaženje svih mogućih putanja iz CFG-a:**
Na dijagramu 2b vidimo da su brojevima 1, 2, 3, 4, 5 označena stanja dijagrama toka, kao i početno (Start) i završno stanje (End). Takođe, primećujemo da u iz ovakvog dijagrama možemo ekstrahovati samo dve putanje: prvu koja prati nispunjenost uslova zadatog sa if i označićemo je kao P_1 i drugu koja prati izvršavanje u kojem uslov iz if nije ispunjen, označenu sa P_2.
3. **Kreiranje matrice pokrivenosti:**
Matrica pokrivenosti (putanja) povezuje testove i putanje sa kojima oni dolaze u dodir. Matrica po kolonama sadrži sve putanje, a po redovima sve testove. Informacija u matrici može biti '0' ili '1', gde '1' u i-tom redu i j-toj koloni znači da test primer t_i pobuđuje izvršavanje putanje p_j. Matrica pokrivenosti za naš primer predstavljena je na slici 3.

	P_1	P_2
t_1	1	0
t_2	0	1
t_3	0	1
t_4	0	1

Slika 3: Matrica pokrivenosti

4. **Generisanje podskupova test primera za odgovarajuće putanje** Za svaku putanju se generišu skupovi test primera koji je pobuđuju, a na osnovu tog skupa i matrice pokrivenosti se računa i rezultat pokrivenosti (eng. *coverage score*) za svaki od test primera po formuli: $CS(t_i) = \frac{\sum_{j=1}^{n} C(t_i,p_j)}{n}$, gde je n broj putanja. Za naš primer koda, podaci se nalaze na slici 2b
5. **Razlučivanje redundantnih i neredundantnih test primera** Počevši od podskupa $S1$, primećujemo da on sadrži samo jedan test primer t_1, te ga stavljamo u skup TS_min, koji predstavlja minimalni skup test-primtera koji nam je potreban. Posmatravši skup

Path	Subset S_i	Coverage score
p_1	S1=t1	CS(t1)=0.5
p_2	S2=t2,t3,t4	CS(t2)=0.5, CS(t3)=0.5, CS(t4)=0.5

Slika 4: Podskupovi test primera

S_2, uviđamo da on sadrži testove čija je pokrivenost jednaka, što nas navodi na zaključak da su ovi testovi međusobno redundantni. Postavlja se pitanje: *Koje od ovih testova ubaciti u minimalan skup test primera?*

6. **Razlučivanje korisnih od beskorisnih test primera** Iako nije intuitivno, ovaj korak je ipak neophodan - on daje odgovor na prethodno pitanje. Možemo sada da izaberemo bilo koji test primer koji će ući u minimalan skup test primera koji nam je neophodan, a ostale da eleminišemo. Međutim, može se desiti situacija u kojoj se početni kod prepravi tako da ga test primer koji smo odabrali ne pokriva i u tom slučaju on postaje beskoristan.

   Recimo da je, na slučajan način, u našem primeru izabaran t_2 tako da ulazi u skup TS_{min}. Sada je potrebno za test primere t_3 i t_4 odrediti koji će u budućnosti biti koristan a koji ne. Kako oba test primera prate putanju P_2, iz nje ćemo izvući uslove koji u njoj mogu da važe i oni će nam poslužiti kao kriterijum pri odlučivanju. Jedina dva moguća uslova su $c1 : a < b$ i $c2 : a == b$. U ovom primeru je test t_4 takav da zadovoljava kriterijum c_1, ali ne i c_2, koji još uvek nije uključen u program. Test t_3 je takav da zadovoljava oba ova uslova, pa će on biti označen kao koristan i smešten u sekundarni skup test primera TS_{sec}, koji je rezervi skup test primera, a t_4 će, kako je redundantan i beskoristan biti potpuno odstranjen.

   Zaključno, primenom ovog postupka dobili smo minimalni skup test primera koji nam je za sada potreban $TS_{min} = \{t_1, t_2\}$ i rezervni skup test primera $TS_{sec} = \{t_3\}$ koji po potrebi može da nadopunjuje primarni skup.

6 Zaključak

Potpuno izbegavanje redundantnih test primera je u velikim projektima jako teško i zbog toga se najčešće pribegava različitim tehnikama njihovog detektotavanja, a zatim i otklanjanja, ili transformisanja.

Postoji nekoliko načina koji obezbeđuju rešenje ovog problema. Pokazalo se u praksi da je jedan od najboljih ostvariv uz pomoć analize pokrivenosti koda. Te analize se mogu upotrebljavati na različite načine: korišćenje heuristika, statistika, rešavača modela, tabela pokrivenosti...

Ovakva rešenja su skalabilna i primenljiva i na druge skupove, a ne samo na skupove test primera, te svoju primenu mogu naći i u biotehnologijama, statistici, i ostalim naučnim granama. Zbog toga se očekuje napredak istraživanja u ovoj oblasti paralelno sa napretkom gorepomenutih grana.

Literatura

[1] Gordon Fraser and Franz Wotawa. Redundancy based test-suite reduction. In *FASE*, 2007.

[2] Anurag Panday, Manish Gupta, and Manoj K. Singh. Test case redundancy detection and removal using code coverage analysis. 2013.