# Sinteza, merenje i predikcija

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Ozren Demonja, Filip Novović, Marko Crnobrnja
mi12319@alas.matf.bg.ac.rs, filnovovic@gmail.com, mi12024@alas.matf.bg.ac.rs

12. septembar 2019.

**Sažetak**

U ovom tekstu predstavljen je projekat čija je zamisao odrediti uticaj različitih ali ekvivalentnih struktura na resurse potrebne za automatsko dokazivanje funkcionalne ekvivalentnosti C programa. U tu svrhu upotrebljene su skripte koje prave manje transformacije nad postojećim korpusom C kodova, a zatim iterativno i nad tim novonastalim korpusom kako bi se testni skup proširio. Nakon toga se meri vreme koje alat CBMC utroši pri proveri modela i dokazivanja ekvivalentnosti struktura. Konkretna implementacija pokazala se nedovoljnom za zadovoljavajuće predviđanje ovog vremena

## Sadržaj

**1 Uvod** **2**

**2 Problem** **2**

**3 Arhitektura sistema** **2**
3.1 Parser . . . . . 3
3.2 Transformacije . . . . . 3
3.3 Generisanje koda . . . . . 3
3.4 Merenje vremena . . . . . 3

**4 Rezultat** **3**

**5 Moguća unapređenja** **4**

# 1 Uvod

Dokazivanje funkcionalne ekvivalencije izmedu programa je korisna metoda verifikacije softvera. Ipak, ona zahteva znatnu količinu računarskih resursa. Kako bi se njihova upotreba mogla bolje organizovati, bilo bi poželjno odrediti neke zavisnosti izmedu različitosti koje postoje između semantički ekvivalentnih programa i resursa potrebnih za dokazivanje ekvivalentnosti. Ovaj rad će ispitati uticaj nekih jednostavnih transfomacija na vreme potrebno za dokaz ekvivalentnosti u alatu CBMC, pri različitim parametrima.

# 2 Problem

Kao jednostavan oblik različitosti ekvivalentnih kodova, razmatramo nekoliko transformacija programskog koda:

1. Dodavanje koda čiji rezultat izvršavanja ne utiče na ostatak programa.
2. Prenošenje dela koda u funkciju.
3. Pretvaranje for i while petlji jednih u druge, pretvaranje u ekvivalentnu konstrukciju sa goto naredbom.
4. Pretvaranje množenja u iterativno sabiranje, deljenje u oduzimanje.
5. Zamena operatora poređenja sa negacijom suprotnog operatora, zamena logičkih veznika po De Morganovim zakonima.
6. Zamena inkrementiranih izraza sa ++ ili += ekvivalentnim naredbama dodele.
7. Transformacija uslovnih naredbi u izraze sa ternarnim operatorom i obrnuto, zamena switch naredbe uslovnom naredbom.
8. Transformacija break i continue naredbi sa ekvivalentnim goto naredbama.
9. Transformacija shift left bitovskog operatora sa ekvivalentim kodom koji se zasniva na množenju levog operanda brojem dva u petlji određeni broj puta

Iako je daleko od toga da se razlika izmedu svih ili čak većine ekvivalentnih programa može svesti na ovakve transformacije, one ipak predstavljaju ne potpuno trivijalne izmene čija svojstva prilikom dokazivanja ekvivalentnosti vredi ispitati. Ove transformacije primenjene su programski nad korpusom rudimentarnih C programa i ekvivalentnost tako izmenjenih programa dokazana je u alatu CBMC.

# 3 Arhitektura sistema

Projekat je implementiran pomoću skriptova pisanih u programskom jeziku Pajton. Glavni moduli sistema implementiraju:

- Omotač za kod (klasa koja učitava kod i nudi osnovne operacije nad njim)
- Parser koji raščlanjuje kod
- Transformacije koda grupisane po sličnosti
- Generisanje transformisanih programa
- Merenje vremena potrebnih da bi se ekvivalentnost dokazala

## 3.1 Parser

Klasa Parser implementirana je u datoteci code parser.py. Njena funkcija je izdvajanje nezavisnih celina koda iz Code objekta koji joj je prosleđen. Implementiran je na osnovu regularnih izraza koristeći paket re. Pored izvlačenja traženih blokova naredbi, parser takode pronalazi naredbe koje se mogu transformisati, a nalaze se u programu, i vraća njihovu listu.

## 3.2 Transformacije

Transformacije su implementirane u datotekama add function transform.py, arithmetic transform.py, branch transform.py, loop transform.py, relational logical transform.py i pomoćnom modulu transform.py. Ovi moduli su uglavnom zasnovani na funkcijama oblika getXData, formY i XToYTransform, gde prva uzima potrebne vrednosti iz naredbi, druga formira naredbe na osnovu naredbi, a treća koristi prve dve za željenu transformaciju. Razmotrimo, na primer, funkcije koje se koriste za transformaciju for petlje u while petlju u loop transform.py na slici 1.

## 3.3 Generisanje koda

Upotreba transformacija nad datim korpusom jednostavnih programa koji se nalaze u direktorijumu test code implementirana je u test.py i predstavlja prvu fazu stvaranja ekvivalentnih kodova. Ona prolazi kroz listu mogućih transformacija za dati program koju dobija od parsera i zatim ih redom primenjuje i tako dobijene programe u direktorijum function equivalent code.

Druga faza generisanja koda predstavlja dalje proširivanje korpusa primenjivanjem transformacija nad samim sobom. Kako bi ovo bilo moguće vrši se neophodno pozivanje pomoćnog programa *indent* kako taj kod ne bi imao neslaganja sa regex šablonima na kojima se zasniva prepoznavanje i izvlačenje transformacija. Transformacije koje umeću u inicijalni kod rekurzivnu ili nerekurzivnu funkciju mogu se izvršiti samo jednom u toku iterativnog transformisanja jednog fajla kako ne bi došlo do dupliranja istoimenih funkcija. Dodatno, neophodno je navesti broj iteracija transformisanja korpusa.

## 3.4 Merenje vremena

U modulu measure.py nalazi se funkcija getCbmcOutputAndExecutionTime koja pokrece CBMC kao potproces, meri vreme koje zahteva i prosleđuje rezultat koji on vraća. Pored toga, ovde se nalaze i funkcije za stvaranje i pisanje direktorijuma sa dobijenim podacima. U measure_test.py je kod koji koristi modul measure.py kako bi generisao potrebne podatke za sve primere u function equivalent code.

# 4 Rezultat

Generisane podatke istražili smo koristeći Python skripte koje su sačuvane u obliku Jupyter sveske kao Analysis.ipnyb. Kako je postojao određen broj primera gde verifikacija nije uspela, oni su izbaceni iz daljeg razmatranja. Od dve mere trajanja verifikacije, execution time i cbmc time od kojih prva meri trajanje procesa a druga vreme koje CBMC prijavljuje,

```
def getForData(code):
    pattern = re.compile(r'(\w+)\s+.*;\s*\n*\s*for
        (.*);(.*);(.*){')
    for match in re.finditer(pattern, code):
        variable = match.group(1)
        init = match.group(2).strip()
        condition = match.group(3).strip()
        increment = match.group(4)[:-1].strip()
        statements = getStatementsInsideCurlyBraces(
            code)

        return (variable + "␣" + init[1:] + ";\n",
            condition, increment, statements)

#...

def formWhileLoop(variable, condition, increment,
    statements):
    loop = variable
    loop += "while(" + condition + ")"
    statements = addStatementInsideBlock(statements,
        "\n" + increment + ";\n", on\_begin=False)

    return loop + statements

#...

def forToWhileTransform(code):
    variable, condition, increment, statements =
        getForData(code)
    return formWhileLoop(variable, condition,
        increment, statements)
}
```

Slika 1: Primer toka transformacije

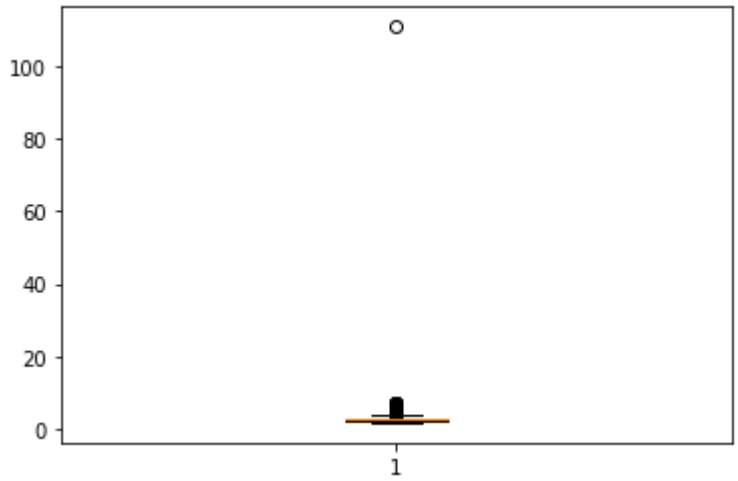
posmatrana je prva jer druga iz nepoznatih razloga nije prijavljena u svim slučajevima. Na slici 2 prikazan je boksplot execution time.

Kako bi se otkrila zavisnost vremena izvršavanja i primenjenih tranformacija i parametara verifikacije, primenjena je grebena regresija sa unakrsnom validacijom sa prethodnom standardizacijom ciljane promenljive iz paketa scikit-learn. Na slici 3 prikazani su dobijeni koeficijenti regresije.

Kao $R^2$ greška regresije dobijena je izrazito niska vrednost jednaka 0.024 što ukazuje da naš model objašnjava samo zanemarljivo mali deo varijanse. Stoga, dobijeni model ne može dati dobar algoritam za predikciju trajanja provere ekvivalencije.

# 5 Moguća unapređenja

Pouzdan algoritam predikcije verovatno bi zahtevao dodatne podatke o obimu i strukturi razmatranog programa pored informacije o konkretnoj




Slika 2: Boxplot vremena izvršavanja

transformaciji, jer se čini da ovi aspekti doprinose većini varijanse izmedu različitih trajanja. Druga mana koja bi se mogla ispraviti je upotreba execution time merenja kao zamene za cbmc time. Prva verovatno varira više u zavisnosti od incidentalnih pojava tokom izvršavanja verifikacije što doprinosi većem šumu u podacima. U daljem ispitivanju problema, verovatno bi bilo preporučljivo koristiti i složeniji parser za generisanje transformacija koji bi mogao obraditi i strukture koje se ne mogu opisati regularnim izrazima.

```
func-2 bitleft operator     -0.108663
--depth                     -0.133722
--unwind                    -0.040374
--partial-loops              0.007922
--no-unwinding-assertions    0.039416
multiplication operator      0.000000
divide operator              0.000000
less operator                0.006492
lessEq operator              0.030497
greater operator            -0.034598
greaterEq operator          -0.043158
eq operator                  0.007532
neq operator                 0.038464
and operator                -0.051455
or operator                 -0.088847
decrement operator           0.000000
increment operator          -0.085191
if statement                -0.023986
? statement                 -0.063204
switch statement            -0.076210
continue statement           0.000000
break statement             -0.076210
recursion block              0.000000
garbage block                0.000000
bitleft operator             0.000000
for-while                    0.048197
for-goto                     0.000000
while-goto                   0.000000
dtype: float64
```

Slika 3: Koeficijenti dobijeni kao rezultat primene grebene regresije