Simboličko izvršavanje unazad

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Ana Jakovljević
ana.jakovljev98@gmail.com

22. januar 2021.

Sažetak

Verifikacija softvera je deo svakog razvoja softvera i važno je da ona bude kvalitetna kako bi se dobio pouzdaniji softver i kako bi se manje vremena trošilo na otkrivanje i ispravljanje grešaka. Tema ovog rada je simboličko izvršavanje unazad, značajna tehnika verifikacije softvera. U okviru rada obrađeno je nekoliko tehnika koje se zasnivaju na simboličkom izvršavanju unazad i koje pokušavaju da prevaziđu izazove koje ono postavlja.

Ključne reči: simboličko, izvršavanje, unazad, CCBSE, Mix-CCBSE, simkretno

Sadržaj

1 Uvod

Neispravnosti softvera mogu imati razne posledice. To mogu biti veliki materijalni troškovi, ali mogu biti i materijalno nemerljive posledice. Zbog toga je verifikacija programa, iako težak, veoma važan proces utvrđivanja ispravnosti softvera.

Obezbeđivanje kvaliteta softvera zahteva značajan deo vremena razvoja softvera. Brooks je procenio da se 50% vremena razvoja softvera utroši na verifikaciju, a kasnije su Hailpern i Santhanam procenili da cena debagovanja, testiranja i ostalih aktivnosti verifikacije softvera prelazi 50% razvojnog vremena [7].

Ručni pristup verifikaciji je intenzivan i nesiguran jer su ljudi skloni greškama, pa je zbog toga potrebna automatizacija. King je u tu svrhu 1976. godine predložio korišćenje simboličkog izvršavanja [6].

U ovom radu je predstavljena jedna strategija simboličkog izvršavanja: simboličko izvršavanje unazad. U nastavku su navedene opšte informacije o simboličkom izvršavanju (2). Zatim je opisana strategija simboličkog izvršavanja unazad, razlozi za korišćenje, primene i izazovi (3). Na kraju su predstavljene tehnike koje koriste ovu strategiju kombinovanu sa drugim pristupima (4).

2 Simboličko izvršavanje

Simboličko izvršavanje predstavlja automatsku formalnu metodu verifikacije softvera. Za razliku od konkretnog izvršavanja sa konkretnim vrednostima ulaza, simboličko izvršavanje je izvršavanje programa sa simboličkim reprezentacijama ulaznih vrednosti, gde jedna simbolička vrednost predstavlja klasu konkretih vrednosti.

Konkretni izvršavači mogu da razviju izraze koji se sastoje samo od konkretnih vrednosti, dok simbolički dodatno omogućavaju i postojanje simboličkih vrednosti u izrazima.

Simbolički izvršavač konstruiše vezu između ulaza i izlaza, duž bilo koje putanje izvršavanja, kreiranjem ograničenja putanje. Ograničenja predstavljaju uslove putanje, formulu nad simboličkim ulazima koja sadrži sve odluke koje su do tada donete. Da bi izvršavanje išlo određenom putanjom potrebno je da ulazne vrednosti zadovoljavaju uslov te putanje [1, 7].

3 Simboličko izvršavanje unazad

Simboličko izvršavanje unazad (eng. symbolic backward execution) je varijanta simboličkog izvršavanja koja se, krećući od specifičnog ciljnog izraza, izvršava dok ne dođe do ulaza u program, pri čemu uzima u obzir samo puteve koji mogu doći do cilja.

Rešavanjem uslova putanje dobijaju se konkretne vrednosti za ulaze koji vode program niz putanju koja pokriva definisani cilj. Nekoliko putanja može biti istraženo u isto vreme i periodično se provera da li su dostižne. Ukoliko se dogodi da je uslov putanje u nekom trenutku nezadovoljiv, tada se putanja odbacuje i vrši se korak unazad [2, 5].

Simboličko izvršavanje unazad je nad-aproksimativni pristup jer ne može da garantuje da će u svim koracima svako simboličko stanje koje razmatra biti dostižno, s obzirom na redosled prikupljanja informacija [7].

3.1 Razlike u odnosu na tradicionalan pristup

Simboličko izvršavanje unazad i simboličko izrvšavanje unapred razvijaju graf kontrole toka na različite načine što uzrokuje različitost ova dva pristupa. Razlike određuju kada je bolje primeniti jednu, a kada drugu strategiju.

U opštem slučaju, izvršavanje unazad ne registruje dovoljno brzo nedostižne putanje, u odnosu na izvršavanje unapred. Simboličko izvršavanje unapred donosi odluke o odbacivanje grane posmatranjem uslova na početku grane, dok simboličko izvršavanje unazad može da konstatuje neizvodljivost putanje nakon što obradi celu granu i tek onda dođe do uslova koji je nezadovoljiv. Dodatno, neke korisne informacije, poput inicijalizovanih vrednosti, nisu dostupne u pristupu unazad.

Sa druge strane, kada se vrši direktno simboličko izvršavanje prema specifičnom definisanom cilju koji treba dostignuti, uslovi blizu cilja ne uspevaju uvek da budu dostupni za vreme izvršavanja unapred. U takvim slučajevima simboličko izvršavanje unazad ima prednost, jer što bliže ciljnom izrazu započne simboličko izvršavanje to je veća šansa da putevi do tog izraza budu pronađeni. Ako se traži izraz koji je dostupan na samo nekoliko putanja u kojima su moguća mnoga grananja, kombinatorno postoji mala šansa da standardno simboličko izvršavanje unapred napravi dobre izbore i pronađe odgovarajuću putanju.

Oba pristupa imaju svoje mane i prednosti, pa postoje i tehnike koje kombinuju ova dva pristupa [7, 8].

3.2 Izazovi simboličkog izvršavanja unazad

Praktična cena i doprinos simboličkog izvršavanja zavise od kompleksnosti putanja programa i ušestalosti pojavljivanja instrukcija teških za razmatranje. Navedeni su neki od izazova koje simboličko izvršavanje unazad postavlja:

- Od suštinske važnosti je *dostupnost interproceduralnog grafa kontrole toka celog programa* koji omogućava otkrivanje mesta poziva funkcija. Konstrukcija takvog grafa može biti zahtevna u praksi, a funkcija može imati različita mesta poziva i na taj način činiti pretragu veoma skupom.
- Uslovi putanja mogu sadržati proizvoljan ceo broj ograničenja i ta *ograničenja mogu biti različite kompleksnosti* (npr. nelinearna ograničenja). U tom slučaju njihovo rešavanje može biti računski teško, a nekada čak i neizvodljivo.
- Simbolička procedura odlučivanja ne može da razlučuje o *eksternim metodama* u koje nema direktan uvid, poput bibliotečkih funkcija i sistemskih poziva.
- *Petlje koje zavise od vrednosti podataka* mogu zahtevati dodatan broj iteracija za pronalazak zadovoljivog uslova putanje, što vodi neograničenom prostoru pretrage.

Rešavanje jednog izazova postavlja druge izazove. Na primer, da bi program bio precizno verifikovan potrebno je simbolički izvršiti biblioteke koje program koristi, čime se povećava cena simboličkog izvršavanja. Sa druge strane, ignorisanjem biblioteka se poboljšava skalabilnost, ali se uvodi nepreciznost [2, 5, 7].

3.3 Primena simboličkog izvršavanja unazad

I pored navedenih izazova, postoje mnogi razlozi za korišćenje simboličkog izvršavanja unazad.

Tehnike za direktno simboličko izvršavanje rešavaju problem dostizanja određenog zadatog cilja. Za to postoje različite strategije, a strategije koje su se pokazale najuspešnijim zasnivaju se na simboličkom izvršavanju unazad [8].

Raspodela grešaka u programima može često da bude neujednačena, pa grupa testova koja pokriva većinu programa može da ne uspe da pokrije delove koji sadrže mnoge greške. Cilj alata za generisanje testova najčešće je maksimizacija ukupne pokrivenosti, pa dobijeni testovi mogu propustiti male delove koda koji su problematični. Alternativni pristup ima za cilj automatsko pronalaženje ulaza za testove koji pokrivaju određene ciljne izraze u kodu, i u tom slučaju se koristi tehnika simboličkog izvršavanja unazad [5].

Još neke primene su date u [3, 7, 8]:

1. *Otkrivanje specifikacija programa:* Moguće je istraživati kod i posmatrati kada se neki njegovi delovi izvršavaju, što je korisno kada programer pokušava da razume nepoznat kod tako što istražuje pod kojim okolnostima se određeni izrazi koda izvršavaju.
2. *Validacija grešaka:* Ovakva analiza može smanjiti uticaj lažno pozitivnih rezultata dobijenih od alata za pronalaženje grešaka. Na primer, moguće je utvrđivanje da greška na određenoj liniji prijavljena alatom statičke analize, zaista postoji. Kada alat pronađe dovoljne preduslove za prijavljenu grešku ona se smatra validiranom i ima veći prioritet.
3. *Generisanje test slučajeva:* Ovo je veoma korisno kod debagovanja i regresionog testiranja. Programer može primiti izveštaj o grešci u određenoj liniji koja nije pokrivena test slučajem, pa kako bi reprodukovao grešku, mora pronaći ostvarivu putanju do te linije. U slučaju regresionog testiranja, ukoliko dođe do izmene programa treba proveriti samo delove programa na koje je promena potencijalno uticala.

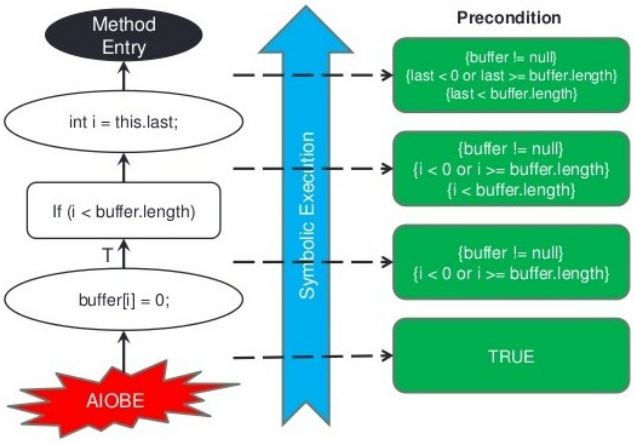
4 Tehnike koje koriste simboličko izvršavanje unazad

Osnovna tehnika simboličkog izvršavanja unazad podrazumeva polaženje od ciljnog izraza, pretragu unazad na osnovu grafa kontrole toka i kreiranje uslova putanje sve dok se ne stigne do ulazne tačke programa. Ilustracija osnovne tehnike data je na primeru reprodukcije pada softvera.

Pri reprodukciji pada softvera, prvi korak neophodan za debagovanje je otkrivanje preduslova pada što se vrši simboličkim izvršavanjem unazad [4]. Proces kreiranja uslova putanje prikazan je na slici 1.

Klasično simboličko izvršavanje unazad nailazi na mnoge prepreke, pa su osmišljene drugačije tehnike koje pokušavaju da prevaziđu te probleme. Neke od njih će biti navedene i opisane u nastavku.

[1] Slika preuzeta sa slajdova za rad [4]



Slika 1: Kreiranje uslova putanje pristupom unazad [1]

4.1 Simboličko izvršavanje unazad kroz lanac poziva

Tehnika *simboličkog izvršavanja unazad kroz lanac poziva* (eng. call-chain backward symbolic execution), u nastavku CCBSE, započinje utvrđivanjem validne putanje u okviru funkcije gde se nalazi ciljni izraz. Pretpostavimo da je ciljni izraz l unutar funkcije f.

CCBSE počinje simboličko izvršavanje unapred od početka funkcije f, proizvodeći skup delimičnih putanja P_f koje započinju u f i vode do l. Zatim CCBSE poziva simboličko izvršavanje unapred od početka svake funkcije g koja poziva f, tražeći putanje koje završavaju pozivima f. Za svaku takvu putanju p, pokušava se nastavljanje kroz putanje p' iz Pf, dok se ne dostigne linija l, a zatim se sve dostižne putanje $p + p'$ dodaju u P_g. Proces tako nastavlja unazad kroz lanac poziva dok CCBSE ne pronađe putanju od početka programa do l.

Glavna razlika ovog pristupa u odnosu na običan pristup je što CCBSE unutar svake funkcije vrši simboličko izvršavanje unapred, dok se u običnom pristupu sve vrši unazad.

U nastavku je dat primer programa gde je cilj pronaći ulazne vrednosti koje uzrokuju da se izvrši komanda $assert(0)$.

Na ovom primeru se tehnike zasnovane na izvršavanju unapred pokazuju veoma loše. Dešava se ili da zaglave na početku ili da zaglave u beskonačnoj petlji. Za sve $m \neq 37$ se troši vreme, jer nijedna od tih vrednosti ne vodi do rešenja.

CCBSE počinje izvršavanje f sa obe promenljive, m i n, podešenim da budu simboličke. Pošto CCBSE ne zna koje vrednosti mogu biti prosleđene do f, CCBSE će potencijalno istražiti 2^6 putanja dobijenih for petljom, i jedna od njih, na primer p, ce dospeti do assert-a. Kada je p pronađeno, CCBSE prelazi na $main$ i istražuje razne putanje koje dosežu do poziva funkcije f. Na mestu poziva f, CCBSE će pratiti p i tako brzo pronaći ostvarivu putanju do greške [8].

```
void main(){
    int m, n;
    symbolic(&m, sizeof(m), "m");
    symbolic(&n, sizeof(n), "n");
    foo();
    g(m,n);
}

void g(int m, int n){
    int i;
    for(i=0;i<1000;i++){
        if(m==i) f(m,n);
    }
}

void f(int m, int n){
    int i, a, sum=0;
    for(i=0;i<6;i++){
        a = n%2;
        if (a) sum += a+1;
        n/=2;
    }

    while(1) {
        if (sum == 0 && m == 37)
            assert(0);
    }
}
```

4.2 Pomešano simboličko izvršavanje unazad kroz lanac poziva

CCBSE može pronaći putanju dosta brzo, međutim to ima cenu jer su upiti često kompleksniji nego u pretrazi unapred, a može biti utrošeno značajno vreme na isprobavanje putanja koje počinju u sredini programa, a koje su konačno neizvodljive.

Možemo posmatrati izmenjenu funkciju *main*() koja poziva funkciju g samo ukoliko je $m \geq 30$. U tom slučaju CCBSE ne uspeva dobro da se pokaže jer ne shvata da je m najmanje 30, pa razmatra konačno neizvodljive uslove $0 \leq m \leq 36$.

```
void main(){
    int m, n;
    symbolic(&m, sizeof(m), "m");
    symbolic(&n, sizeof(n), "n");
    foo();
    if (m >=30) g(m,n);
}
```

Zbog toga je osmišljena nova tehnika: *pomešano simboličko izvršavanje unazad kroz lanac poziva* (eng. Mix-call-chain backward symbolic execution), u nastavku Mix-CCBSE.

Mix-CCBSE pokreće simboličko izvršavanje unapred (S) od *main*-a u isto vreme kada pokreće CCBSE od ciljne linije. Kao i pre, pretraga unazad će zaglaviti u pronalasku putanje od ulaza u g do *assert*-a. Međutim, u pretrazi unapred je g pozvan sa $m \geq 30$, pa se f uvek poziva sa $m \geq 30$, zbog čega ubrzo biva pogođen pravi uslov $m == 37$. Primetimo da pretraga unazad mora pronaći putanju od ulaza u f do *assert*-a pre nego što je f pozvan sa $m = 37$ u pretrazi unapred, kako bi se ove dve pretrage uklopile. Ako se to ne dogodi, Mix-CCBSE biva degenerisana tako da se njeni delovi izvršavaju nezavisno paralelno, što je najgori slučaj.

Ako S naiđe na putanju p koja je konstruisana pomoću CCBSE, pokušava se praćenje p, da bi se, ukoliko je moguće, stiglo do ciljne linije

(uporedo se S normalno nastavlja). Ovako, Mix-CCBSE radi bolje od CCBSE i S kada se pokreću odvojeno, jer u poređenju sa CCBSE može da preskače mnoge pozive funkcija od početka programa kako bi stigla do putanja koje se grade, a u poređenju sa S može da izvrši značajno skraćivanje pretrage kada naiđe na putanju izgrađenu od CCBSE [8].

4.3 Simkretno izvršavanje

S obzirom da se klasičan pristup bori sa različitim izazovima, u [5] je za njihovo rešavanje predloženo *simkretno izvršavanje*. Simkretno izvršavanje je pristup koji kombinuje simboličko izvršavanje unazad sa konkretnim izvršavanjem radi efikasnog pronalaženja ciljnih ulaza. Ovaj pristup deluje u dve faze:

1. Simboličko izvršavanje unazad se koristi za pronalazak izvodljive putanje izvršavanja od datog cilja do bilo koje ulazne tačke programa. Počevši od ciljnog izraza, istražuje se graf kontrole toka unazad. Ukoliko procedura odlučivosti pri proveri zadovoljivosti uslova ne može da odgovori na upit, tada algoritam uklanja poslednje dodato ograničenje iz uslova putanje i tretira ga kao potencijalno zadovoljivo, odlažući rešavanje za drugu fazu. U ovoj fazi se konstruiše *trag* (eng. trace) programa na praćenoj putanji, koji se u svakom koraku pretrage dopunjava trenutnim izrazima bez obzira na nihovu zadovoljivost.
2. Konkretno izvršavanje unapred u ovoj fazi počinje kada simboličko izvšravanje unazad dođe do ulazne tačke. Izvršavanjem traga programa duž otkrivene putanje, ova faza koristi heurističku pretragu da pronađe ulazne vrednosti koje zadovoljavaju ograničenja koja su bila problematična u prvoj fazi. Algoritam iznova izvršava trag programa nad ulaznim vrednostima, određujući koliko su uslovi grana u tragu blizu da budu zadovoljivi, a zatim modifikuje neki od ulaza kako bi se približio potpunom rešenju.

U nastavku je dat primer funkcije i traga koji se za nju kreira.

```
0     void function(){
1         int res = x+23;
2         if (res==8192){
3             if(sin(u) > 0)
4                 assert(0);
5         }
6     }
```

Za izraze čije je rešavanje odloženo za drugu fazu, algoritam dodaje pozive specijalnoj metodi *change*(). Pretraga prati jednu putanju izvršavanja, pa se if-izrazi i ostali uslovi ne dodaju direktno u trag. Umesto toga, algoritam dodaje poziv *fit()* metodi koja signalizira kroz koju od grana uslova je prošla pretraga.

Dobija se uslov putanje $sin(u) > 0 \wedge res = 8192 \wedge res = x + 23$. Međutim, procedura odlučivanja ne može da reši ovaj uslov zbog postojanja poziva eksternog metoda (sin), pa se problematično ograničenje preskače i dobija se putanja $res = 8192 \wedge res = x + 23$ sa rešenjem $x = 8169$.

```
0     void trace(int x, double u){ //instrukcije druge faze:
1         int res = x+23;
2         fit(res, '==', 8192); //pronaci ulaze sa res==8192
3         double v = sin(u);
4         change(v);            //menjati ulaze koji uticu na v
5         fit(v,'>',0);         //pronaci ulaze sa v>0
6     }
```

Ovako izgleda trag kreiran nakon prve faze. Poziv *change*() metode u tragu označava da vrednost v mora biti pronađena heurističkom pretragom. Druga faza počinje izvršavanjem traga sa ulazima $x = 8169$ i $u = 0.0$, rešenjima dobijenim u prvoj fazi. Izvršavanjem poziva $fit()$ metode druga faza određuje da uslov $v > 0$ nije još uvek zadovoljen. Zato se prilagođava jedan od ulaza koji utiče na v i ponovo se izvršava trag. Ovaj proces se nastavlja dok se ne pronađe rešenje ili se ne utroši previše vremena.

Integracija konkretnog izvršavanja omogućava simkretnom izvršavanju rešavanje aritmetičkih ograničenja koja su previše teška za procedure odlučivanja i omogućava efikasno upravljanje eksternim metodama. Ako petlja duž putanje zahteva previše simboličkih prelaza simkretno izvršavanje petlju tretira kao poziv eksterne metode, tako da se problem pronalaženja pravog broja iteracija rešava u jeftinijoj, konkretnoj fazi [5].

4.3.1 Poređenje sa konkoličkim izvršavanjem

Cilj konkoličkog izvršavanja nije pokrivenost specifičnog cilja, već što veća ukupna pokrivenost. Poput konkoličkog, simkretno izvršavanje je jače od čistog simboličkog zbog sposobnosti da ublaži ograničenja rešavača kroz konkretno izvršavanje. Za razliku od konkoličkog, simkretno izvršavanje može izbeći istraživanje nerelevatnih putanja. Eksperimentalna izračunavanja pokazuju da simkretno izvršavanje pronalazi ulaze u više bitnih slučajeva nego konkoličko i da istražuje manje segmenata putanja [5].

5 Zaključak

Ovim radom je obuhvaćena osnova i motiv upotrebe simboličkog izvršavanja unazad. Uvedeno je i opisano nekoliko tehnika koje se zasnivaju na njemu. Dati su primeri iz radova koji ilustruju konkretne mane i prednosti predstavljenih strategija. Eskperimentalni rezultati primene prikazanih pristupa, u poređenju sa nekim drugim pristupima, mogu biti pronađeni u radovima [5, 8]. Implementacije CCBSE i Mix-CCBSE u Otter-u mogu biti pronađene u [8], a sam simbolički izvršavaš Otter u [9]. Grub opis alata koji vrši simkretno simboličko izvršavanje, Cilocnoc-a, dat je u [5].

Literatura

[1] Materijali za kurs Verifikacija softvera na Matematičkom fakultetu Univerziteta u Beogradu. on-line at: http://www.verifikacijasoftvera.matf.bg.ac.rs.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), May 2018.

[3] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: A powerful approach to weakest preconditions. PLDI '09, page 363–374, New York, NY, USA, 2009. Association for Computing Machinery.

[4] N. Chen and S. Kim. Star: Stack trace based automatic crash reproduction via symbolic execution. *IEEE Transactions on Software Engineering*, 41(2):198–220, 2015.

[5] Peter Dinges and Gul Agha. Targeted test input generation using symbolic-concrete backward execution. In *Proceedings of the 29th*

ACM/IEEE International Conference on Automated Software Engineering, ASE '14, page 31–36, New York, NY, USA, 2014. Association for Computing Machinery.

[6] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[7] Yude Lin. Symbolic execution with over-approximation. 2017.

[8] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In Eran Yahav, editor, *Static Analysis*, pages 95–111, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[9] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. *Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems*, page 445–454. Association for Computing Machinery, New York, NY, USA, 2010.