Verifikacija softvera

Seminarski rad u okviru kursa
Metodologija stručnog i naučnog rada
Matematički fakultet

Peleksić Ljubica, Milica Kojičić
peleksic.ljubica@gmal.com, milicakojicic@gmail.com

31. mart 2015.

Sažetak

Sadržaj

1 Uvod

2 Simboličko izvršavanje

U ovom odeljku ćemo se baviti simboličkim izvršavanjem programa. Radi se o tome da se umesto prosleđivanja normalnog ulaza programu, prosleđuju proizvoljne simboličke vrednosti. Izvršavanje programa se odvija normalno, osim što vrednosti mogu biti proizvoljne formule sastavljenje od ulaznih simbola. Teški, ali i zanimljivi problemi koji nastaju pri simboličkom izvršavanju, potiču od uslovnih grananja. Proizvodnja velikih razmera pouzdanih programa je jedan od osnovnih uslova za primenu računara u izazovnim problemima današnjice. Neke tehnike se koriste u praksi, a neke su još uvek tema istraživanja. Program treba da ispunjava zahteve, iako nisu date formalne specifikacije. Neki uzorak podataka koji treba da bude obrađen od strane programa je dat programu. Ako je ocenjeno da program proizvodi korektan rezultat za dati uzorak podataka, pretpostavlja se da je on i tačan. Osnovno pitanje koje se postavlja je kako odabrati taj uzorak.

Skorašnji radovi o dokazivanju korektnosti programa formalnom analizom obecavaju mnogo i pokazuje se da je to ultimativna tehnika za pravljenje pouzdanih programa. Ali nije verovatno da će se svođenje teorije na praksu lako rešiti u skoroj budućnosti. Testiranje programa i dokazivanje korektnosti programa mogu se posmatrati kao ekstremne alternative. Dok testira, programer pažljivim proučavanjem rezultata može biti siguran da za neki test primer u uzorku program radi tačno. Tačnost izvršavanja za ulaze koji nisu u uzorku je i dalje podložan sumnji. S druge strane, pri dokazivanju korektnosti programa, programer formalno dokazuje da program pri svakom izvršavanju ispunjava zahteve koji su dati specifikacijom, a da pritom ne mora da izvršava program uopšte. Da bi to mogao, on daje preciznu specifikaciju tačnog ponašanja programa i dalje sledi formalnu proceduru dokazivanja da bi pokazao da su program i specifikacija konzistentni. Poverenje u ovaj metod zavisi od tačnosti pri kreiranju specifikacije i konstrukcije koraka u dokazu, kao i od mašinski zavisnih problema kao što su prekoračenje, zaokruživanje itd... Ovde se radi o praktičnom pristupu između ove dve krajnosti. Iz najprostijeg ugla, ovo je poboljšana vrsta tehnike testiranja. Umesto izvršavanja programa nad skupom ulaza, program se "simbolično" izvršava nad klasama ulaza. To znači da rezultat simboličkog izvršavanja može biti ekvivalentan velikom broju uobičajenih test primera. I ovi rezultati svakako mogu biti provereni od strane programera. Klasa ulaza, okarakterisana svakim simboličkim izvršavanjem, određena je u zavisnosti od kontrole toka programa nad ulazom. Ako je kontrola toka programa u potpunosti nezavisna od ulaznih varijabli, jedno simboličko izvršavanje će biti dovoljno da proveri sva moguća izvršavanja programa. Ako pak, kontrola toka zavisi od ulaza, mora se pribeći analizi slučajeva. Često, skup ulaznih klasa koje pokrivaju sve moguće slučajeve je praktično beskonačan, tako da je ovo i dalje metoda testiranja. Kako god, ulazne klase su određene samo onim ulazima koji utiču na kontrolu toka i simboličko testiranje obećava postizanje boljih rezultata mnogo lakše od običnog testiranja.

U nastavku ćemo se baviti simboličkim izvršavanjem u idealnom smislu, idealnom iz sledećih razloga: Pretpostavka je da programi obrađuju samo cele brojeve proizvoljne veličine. Drveta izvršavanja koji su rezultat simboličkog izvršavanja su za većinu programa beskonačna. Takođe, simboličko izvršavanje *IF* naredbe zahteva dokazivanje teorema, što je

za jednostavnije programske jezike praktično nemoguće ostvariti. Ovaj idealni sistem nam predstavlja standard u odnosu na koji se sistemi za simboličko izvršavanje mogu meriti. Svaki programski jezik ima svoju semantiku izvršavanja, koja opisuje kakve podatke varijable mogu da reprezentuju, kako naredbe manipulišu tim podacima i kojim se tokom odvija izvršavanje naredbi. Paralelno se može definisati alternativna semantika simboličkog izvršavanja, u kojoj se pravi podaci ne koriste, ali mogu biti predstavljeni proivoljnim simbolima. Simboličko izvršavanje predstavlja prirodnu ekstenziju normalnog izvršavanja, u kojoj su operatori programskog jezika prošireni tako da prihvataju simboličke ulaze i proizvode simboličke formule na izlazu. Pritom, semantika izvršavanja je promenjena u smislu da podržava simboličko izvršavanje, ali ni sintaksa jezika, ni pojedinačni programi pisani u tom jeziku nisu promenjeni. Jedini način da se simbolički podaci predstave programu jeste da mu se daju kao ulaz. Svaki put kada program zahteva novu ulaznu vrednost, ona mu se obezbeđuje iz liste simbola $\{\alpha_1, \alpha_2, \alpha_3, ...\}$. Oni će u nekom trenutku biti dodeljeni kao vrednosti programskim varijablama (kao parametri procedura, globalne varijable i slično). Da bismo mogli da baratamo simboličkim ulazima, dozvoljeno je da vrednosti varijabli mogu da budu α_i, kao i celobrojne konstante. I aritmetički izrazi, kao i *IF* naredbe moraju biti proširene tako da mogu da barataju sa simboličkim vrednostima. Time što dozvoljavamo da varijable uzimaju vrednosti polinoma nad α_i, simboličko izvršavanje naredbi dodele teče prirodno. Stanje izvršavanja programa obično uključuje vrednosti varijabli, kao i brojač instrukcija, koji nam daje informaciju o tome koja se naredba trenutno izvršava. Definicija simboličkog izvršavanja *IF* naredbe zahteva da se u stanje izvršavanja uključi i uslov putanje (path condition - *pc*). *Pc* predstavlja logički izraz nad simboličkim ulazima $\{\alpha_i\}$. On nikad ne sadrži programske varijable, i možemo ga predstaviti u obliku konjukcije izraza forme $R \geq 0$ ili $\neg(R \geq 0)$, gde je$R$ polinom nad $\{\alpha_i\}$, na primer: $\{\alpha_1 > 0 \wedge \alpha_2 + 3 > 0\}$. Kao što ćemo videti *pc* predstavlja akumulator osobina koje ulaz mora da zadovolji da bi izvršavanje pratilo određenu pridruženu putanju. Svako simboličko izvršavanje počinje tako što se *pc* inicijalizuje na tačno. Kako se prave pretpostavke o ulazima, u cilju odabira putanje koja je odredjena *IF* naredbom, te pretpostavke se dodaju u *pc*. Simboličko izvršavanje *IF* naredbe teče kao njeno normalno izvršavanje, evaluacijom njoj pridruženog logičkog izraza zamenjivanjem varijabli njihovim vrednostima. Kako su vrednosti varijabli polinomi nad $\{\alpha_i\}$, uslovni izraz je u formi $R \geq 0$, gde je R polinom. Nazovimo taj izraz *q*. Koristeći trenutni *pc*, formiramo sledeće izraze:
(a) $pc \supset q$
(b) $pc \supset \neg q$
Najviše jedan od ova dva tvrđenja može biti tačan (eliminišemo trivijalan slučaj kad je *pc* netačan). Kada je tačan jedan od ova dva izraza tačan, nastavlja se sa izvršavanjem *IF* naredbe tako što se kontrola toka prebacuje na njen *THEN* deo ako je izraz (a) tačan ili na njen *ELSE* deo ako je (b) tačno. Ovakav tip izvršavanja se zove ne račvajući, jer se izvršava tačno jedna od sve grane, za razliku od slučaja kada ni slučaj (a) ni slučaj (b) nisu tačni. U ovakvoj situaciji postoji skup ulaza u program koji zadovoljava *pc* i koji bi pratio *THEN* alternativu, ali postoji i skup ulaza koji bi išao *ELSE* alternativom. Pošto su obe alternative moguće, jedini kompletan pristup bi bio da se ispitaju obe putanje, tako da se ovde simboličko izvršavanje "račva"u dve putanje. Biranjem *THEN* alternative, pretpostavlja se da ulazi zadovoljavaju *q*, pa je ta informacija dodata u *pc* : $pc \leftarrow pc \wedge q$. Isto tako, biranjem *ELSE* alternative dobi-

jamo $pc \leftarrow pc \wedge \neg q$. *Pc* se zove "putanja uslova"jer predstavlja akumulaciju uslova koji određuju jedinstvenu kontrolu toka u programu.

2.1 Drvo izvršavanja

Drvo izvršavanja opisuje putanje kojima je teklo simboličko izvršavanje programa. Svakom čvoru drveta je priključena izvršena naredba (označena brojem), dok svaka veza između čvorova predstavlja direktnu vezu između naredbi. Za svako račvajuće izvršavanje *IF* naredbe od odgovarajućeg čvora polaze dve veze koje su označene sa "T" i "F" za tačan (*THEN*) i netačan (*ELSE*) deo, respektivno. Takođe se svakom čvoru pridružuju vrednosti varijabli, brojač instrukcija, kao i *pc*. Simbolička izvršavanja za funkciju koja izračunava stepen broja su data na *Slici 1.*

```
1  POWER: PROCEDURE(X, Y);
          2 Z ← 1;
          3 J ← 1;
4  LAB:   IF Y ≥ J THEN
5           DO; Z ← Z * X;
6               J ← J + 1;
7               GO TO LAB; END;
8           RETURN (Z) ;
9  END;
```

Nakon naredbe	J	X	Y	Z	pc
	Vrednosti će biti:				
1	?	α_1	α_2	?	tačno
2				1	
3	1	–	–	–	–
4	detalji izvršavanja:				
	(a) evaluacijom $Y \geq J$ dobijamo $\alpha_2 \geq 1$				
	(b) koristeći pc izvršiti proveru:				
	(i) true $\supset \alpha_2 \geq 1$				
	(ii) true $\supset \neg(\alpha_2 \geq 1)$				
	(c) nijedno od ova dva, pa se račva:				
Slučaj kada $\neg(\alpha_2 \geq 1)$:					
4	1	α_1	α_2	1	$\neg(\alpha_2 \geq 1)$
8	ovaj slučaj je završen. (vraća 1 kada je $\alpha_2<1$)				
Slučaj kada $\alpha_2 \geq 1$:					
4	1	α_1	α_2	1	$(\alpha_2 \geq 1)$
5				α_1	
6	2	–	–	–	
7					
4	detalji izvršavanja:				
	(a) evaluacijom $Y \geq J$ dobijamo $\alpha_2 \geq 2$				
	(b) koristeći pc izvršiti proveru:				
	(i) $\alpha_2 \geq 1 \supset \alpha_2 \geq 2$				
	(ii) $\alpha_2 \geq 1 \supset \neg(\alpha_2 \geq 2)$				
	(c) nijedno od ova dva, pa se račva:				
Slučaj kada $\neg(\alpha 2 \geq 2)$:					
4	2	α_1	α_2	1	$\alpha_2 \geq 1 \wedge \neg(\alpha_2 \geq 2)$
8	ovaj slučaj je završen. (vraća α_1 kada je $\alpha_2=1$)				
Slučaj kada $\neg(\alpha 2 \geq 2)$:					
4	2	α_1	α_2	α_1	$\alpha_2 \geq 1 \wedge \alpha_2 \geq 2$
⋮	⋮				
⋮	⋮				
U ovom slušaju simboličko izvršavanje će se nastaviti neograničeno					

Slika 1

Drveta formirana na ovaj način imaju sledeće zanimljive osobine: Za svaki terminalni čvor (koji odgovara kompletno izvršenoj putanji) u drvetu postoji odgovarajući nesimbolički ulaz u program, koji će tokom normalnog izvršavanja programa pratiti istu putanju (listu izvršenih naredbi), što je ekvivalentno tvrđenju da *pc* nikad neće postati netačno. Druga važna osobina jeste da je u svakom terminalnom čvoru vrednost *pc*-ja jedinstvena. Dve putanje koje polaze iz zajedničkog korena drveta odlučivanja imaju jedinstven čvor račvanja, odakle putanje pocinju da divergiraju, u smislu da je u tom čvoru jednom *pc*-ju dodat neki q a drugom $\neg q$. Posto u tom slučaju *pc* neće postati netačno, oni moraju da se razlikuju.

2.2 Komutativnost

Ovako definisano simboličko izvršavanje zadovoljava osobinu komutativnosti. Naime, ako se program izvršava na konvencionalni način, sa određenim skupom celobrojnih vrednosti $\{j_i\}$ na ulazu (dakle prvo se izvrši operacija instanciranja simbola $\{\alpha_i\}$ specifičnim celobrojnim vrednostima), rezultat će biti isti ako prvo izvršimo program simbolički, pa

onda instanciramo simboličke rezultate (dodelimo j_i α_i), što se vidi i na slici iznad. Upravo je ova osobina komutativnosti simboličkog izvršavanja od značaja. Ono proizvodi iste efekte kao i konvencionalno izvršavanje i nije proizvoljna alternativa, već prirodna ekstenzija normalnom izvršavanju programa.

2.3 Simboličko izvršavanje i dokazivanje korektnosti programa

Pri dokazivanju korektnosti programa, programer uz sam program podrazumeva i "ulazni"i "izlazni"predikat koji definišu tačno ponašanje programa. Program je korektan ako za sve ulaze koji zadovoljavaju ulazni predikat, rezultati koji proizilaze iz izvršavanja programa (ako ih uopšte ima) zadovoljavaju izlazni predikat. Jedan od koraka pri dokazivanju korektnosti programa - *generisanje uslova verifikacije*, se poprilično lako može uraditi simboličkim izvršavanjem programa.

Simboličko izvršavanje je takođe korisno i u drugim aspektima programske analize, uključujući i generisanje test primera kao i u optimizaciji programa. Pretpostavimo da imamo dodatne tri naredbe *ASSERT*, *PROVE* i *ASSUME* koje se koriste da povežu predikat sa programom. Sve tri naredbe imaju kao argumente logičke izraze, npr: *ASSERT(X>0)*. Varijable u ovim formulama su programske varijable i ako pretpostavimo da je B argument ovih naredbi, onda se on evaluira koristeći trenutne vrednosti programskih varijabli i rezultujuća vrednost se dodaje *pc*-ju: $pc \leftarrow pc \wedge value(B)$. Naredba *PROVE(B)* se izvršava, formira izraz $pc \supset value(B)$ i pokušava da dokaže da je to teorema. Pored prethodno navedena dva predikata, mogu se definisati i dodatni induktivni predikati (u svakoj petlji najmanje jedan predikat npr.). Predikati su povezani sa programom koristeći *ASSERT* naredbu (inicijalni predikat je *ASSERT* naredba na početku programa). I sada imamo fiksiran skup putanja kroz program gde svaki od njih počinje i završava se sa *ASSERT* i za svaki mora da se dokaže njegova korektnost. Dokaz korektnosti svake putanje dokazuje se njenim simboličkim izvršavanjem i to:

1. Menjanjem *ASSERT* naredbe na početku sa *ASSUME*, kao i menjanjem iste naredbe sa *PROVE* na kraju
2. Inicijalizovanjem *pc*-ja na tačno i programske varijable na α_1, α_2, ...
3. Simboličkim izvršavanjem date putanje
4. Ako *PROVE* na kraju putanje pokaže tačno, putanja je korektna, a inače nije

Pošto simboličko izvršavanje zadovoljava svojstvo komutativnosti, jednostavno je zaključiti da je ovo validan metod dokazivanja. Izvršavanjem *PROVE* na kraju putanje postavlja kandidata za teoremu: dakle, ako pretpostavimo da je predikat na početku bio zadovoljen i mi smo pratili ovu putanju koja je snimljena u *pc*-ju, pitamo se da li trenutne vrednosti programa na kraju ove konkretne putanje zadovoljavaju predikat na kraju. Usvajanje ovakve definicije za *PROVE* naredbu za dokazivanje korektnosti, vrlo je korisno i za tesitiranje korektnosti metodom simboličkog izvršavanja. Naime, u testiranju programa (bilo ono simboličko ili ne) moraju da se ispituju izlazi iz programa i mora da se sudi o njihovoj korektnosti. Ako kriterijum korektnosti može da se formalizuje u smislu ulaznog i izlaznog predikata, onda simbolički izvršilac nakon toga takođe

može da pouzdano proveri test rezultate. Za svaki program čije je simboličko drvo konačno i kod koga je kriterijum korektnosti eksplicitno zadat ulaznim i izlaznim predikatima, iscrpno simboličko izvršavanje i dokazivanje korektnosti su zapravo isti proces.

3 Alat KLEE

KLEE je alat za simboličko izvršavanje, koji može da automatski generiše testove koji postižu veliku pokrivenost na širokom skupu kompleksnih programa. *KLEE* je korišćen za temeljnu proveru 89 samostalnih programa u okviru *GNU COREUTILS*, koji predstavlja jezgro korisničkog okruženja instaliran na milione *UNIX* sistema. Testovi koje je generisao *KLEE* su postigli pokrivenost preko 90% i značajno prevazišli pokrivenost testova koje su programeri pisali ručno. Takođe, *KLEE* se koristi kao alat za nalaženje bagova i primenjen je na 452 aplikacije (na oko 430K linija koda) gde je našao 56 ozbiljnih bagova, uključujući i 3 u *COREUTILS*-u koji su postojali više od 15 godina. Mnoge klase grešaka je teško naći bez izvršavanja koda deo po deo. Neophodnost takvog takvog tesiranja kombinovanog sa teškoćom i slabim performansama random i manualnih pristupa dovelo je do skorih radova u korišćenju simboličkog izvršavanja kako bi se automatizovao proces generisanja test primera. *KLEE* ima dva cilja: (1) da pokrije svaku liniju izvršnog koda u programu i (2) detektuje svaku opasnu operaciju (npr. dereferenciranje) ako postoji ijedna ulazna vrednost koja može da prouzrokuje grešku. *KLEE* to radi simbolički, za razliku od normalnog izvršavanja, gde operacije nad operandima proizvode konkretne vrednosti, ovde se generišu ograničenja koja tačno opisuju skup vrednosti koji je moguć na datoj putanji. Kada *KLEE* detektuje grešku ili kada se stigne do *exit* poziva u putanji, *KLEE* rešava ograničenja trenutne putanje (ovo odgovara *pc*-ju spominjanom ranije) kako bi proizveo test primer koji će pratiti istu putanju kada se program ponovo pokrene normalno (kompajlira sa *gcc*). *KLEE* je dizajniran tako da putanja originalnog izvršavanja programa uvek prati istu putanju kojom je išao *KLEE* (nema lažne pozitivnosti). Ipak, nedeterminizam u proveri koda i bagovi u samom *KLEE*-u ponekad dovedu do toga. *KLEE* koristi različite optimizacije rešavanja ograničenja, reprezentuje stanje programa na kompaktan način i koristi različite heuristike da bi ostvario visoku pokrivenost koda. Takođe, koristi i jednostavan pristup za sinhronizaciju sa okruženjem.

3.1 Korišćenje

Korisnici prvo kompajliraju svoj kod u bajtkod koristeći javno dostupan *LLVM* kompajler za *GNU C*. Pretpostavimo da imamo program 1.c. Kompajliramo ga na sledeći način:
llvm-gcc –emit-llvm -c 1.c -o 1.bc
Dalje se *KLEE* izvršava na ovako generisanom bajtkodu, uz moguće različite opcije koje govore koliko imamo simboličkih ulaza nad kojim testiramo kod, koji je njihov tip i veličina. Za naš program koristimo komandu:
klee –max-time 2 –sym-args 1 10 10 1.bc
Prva opcija *–max-time* govori *KLEE*-u da proverava 1.bc najviše dva minuta. Ostatak opisuje simboličke ulaze - opcija *–sym-args 1 10 10* govori da se koristi 0 do 3 argumenta komandne linije, gde se 1. sastoji od jednog karaktera, a ostala dva od 10.

3.2 Arhitektura

KLEE funkcioniše kao hibrid između operativnog sistema za simboličko procesiranje i interpretera. Svaki simbolički proces ima svoje registre, stek, hip, brojač instrukcija i"uslov putanje" - *pc*. Da bismo izbegli konfuziju sa procesima u *UNIX* sistemima, simboličke procese ćemo da nazivamo *stanjima*. Programi se kompajliraju na *LLVM* asemblerski jezik, koji liči na *RISC* skup instrukcija. *KLEE* direktno interpretira ovaj skup instrukcija i mapira svaku instrukciju u uslov. U svakom trenutku *KLEE* može da izvršava veliki broj stanja. Jezgro *KLEE*-a je interpreterska petlja koja bira stanje koje će da izvrši i dalje simbolički izvršava svaku pojedinačnu instrukciju u kontekstu tog stanja. Ova petlja se izvršava sve dok više nema stanja za izvršavanje ili je korisnički definisano vreme isteklo. Za razliku od normalnih procesa, memorijske lokacije stanja - registri, stek i hip - referišu na drveta umesto na sirove podatke. Listovi ovih drveta su simboličke promenljive ili konstante, a unutrašnji čvorovi su operacije *LLVM* asemblerskog jezika.

Što se instrukcija grananja tiče, tu *KLEE* koristi *STP*, odnosno rešavač ograničenja (*constraint solver*) kako bi utvrdio da li je uslov tačan ili netačan na trenutnoj putanji. Ako jeste, brojač instrukcija se ažurira na odgovarajuću lokaciju, inače su obe grane moguće i tada *KLEE* klonira stanje tako da može da istraži obe putanje. Potencijalno opasne operacije implicitno generišu grane koje proveravaju da li neka od ulaznih vrednosti može da proizvede grešku. Na primer, instrukcija deljenja generiše granu koja proveravala da li je delilac 0. Ako je greška detektovana, *KLEE* generiše test primer koji je hvata i završava to stanje. Problem je što broj stanja u praksi eksponencijalno raste, čak i za najmanje programe *KLEE* generiše na stotine i hiljde stanja tokom nekoliko minuta interpretiranja. Skoro uvek, cena rešavanja ograničenja je dominantna u odnosu na sve ostalo i upravo je ona zaslužna za NP-kompletnu logiku *KLEE*-a i zato se ulaže mnogo truda u pojednostavljivanje izraza pre nego što oni stignu do STP-a.

3.3 Modelovanje okruženja

Veliki deo koda je kontrolisan vrednostima koji su dobijeni iz okruženja. Argumenti komandne linije određuju kako i koje se procedure izvršavaju, ulazne vrednosti u program utiču na *IF* naredbe, a sam program mora biti u stanju da čita iz fajl sistema. Pošto ulazi mogu biti zlonamerni, program mora biti u stanju da rukuje na pravi način tim ulazima, a testirati sve važne vrednosti i granične slučajeve nije trivijalno. Rukovanje okruženjem se izvodi tako što se pozivi koji mu pristupaju redirektuju na modele koji razumeju semantiku željene akcije dovoljno dobro da mogu da generišu odgovarajuća ograničenja. Krucijalno je to što su ti modeli napisani na *C*-u i što korisnici mogu da ih prilagode i prošire, a da ne moraju da razumeju interne detalje vezane za sam alat. Postoji oko 2.500 linija koda koji definišu jednostavne modele za oko 40 sistemskih poziva (*open, read, write, stat, lseek...*). Ovaj se pristup razlikuje od tradicionalnih sistema za simboličko izvršavanje koji su statični i ne interaguju sa okruženjem uopšte.

Na kraju, glavni cilj je da se u budućnosti za neki proizvoljan program rutinski može pokriti vise od 90% koda i da se pritom pokriju svi interesantni ulazi. Iako je još uvek dug put ka tome, rezultati pokazuju da ovaj pristup dobro radi za širok skup realnih programa.

3.4 ... podnaslov

4 Poslednji naslov

5 Zaključak

A Dodatak