

Najslabiji preduslov nestruktuiranih programa

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Čedomir Dimić
dimiccedomir@gmail.com

10. maj 2018.

Sažetak

Sistemi za verifikaciju programa rade tako što transformišu program u logički izraz koji predstavlja ulaz dokazivača teorema. Logički izraz predstavlja najslabiji preduslov programa u odnosu na njegovu specifikaciju. Ako dokazivač teorema dokaže da je logički izraz tačan onda se i program smatra tačnim. Izračunavanje izraza za imperativne, struktuirane programa je jednostavno, iako mogu nastati problemi zbog petlji i efikasnosti izračunavanja. U ovom radu je predstavljen pristup određivanja najslabijeg preduslova za nestruktuirane programe koji je ispravan čak i ako postoje petlje. Izračunavanje je efikasno i rezultujući logički izraz omogućava dokazivaču teorema da efikasno dokaže teoremu [2].

1 Uvod

Tehnika za proveravanje ispravnosti programa se naziva *statička verifikacija softvera*. Statički verifikator softvera uglavnom radi tako što kao ulaz uzima program i njegovu specifikaciju i na osnovu toga pravi logičku formulu prvog reda, čija validnost znači da program zadovoljava specifikaciju. Ona se koristi kao ulaz za dokazivač teorema.

U ovom radu je opisano generisanje uslova za verifikaciju u statičkom verifikatoru programa *Spec#*. Pomoću njega možemo proizvesti uslove za verifikaciju značajno manje nego koristeći *ESC/Java*. Takođe, ovaj verifikator je uopšteniji zato što može da se primenjuje i na grafove kontrole toka, a ne samo na strukturane programe.

Generisanje uslova za verifikaciju se radi u koracima. Početna tačka je graf kontrole toka. To je logičan izbor s obzirom da statički verifikator *Spec#* kao ulaz koristi međujezik .NET virtualne mašine. Koriste se standardne tehnike kompilacije koje dupliraju instrukcije i na taj način se eliminišu višestruke ulazne tačke u petljama, transformišući graf kontrole toka u redcubilni graf. Zatim se eliminišu petlje, proizvodeći aciklični graf kontrole toka koji je ispravan samo ako je ispravan i originalni program. Nakon toga se aciklični program pretvara u pasivni program tako što se naredbe dodele zamene naredbama pretpostavki. Na kraju, primenjuju se najslabiji preduslovi na nestruktuiran aciklični pasivni program da bi se generisali uslovi verifikacije.

U narednim poglavljima će biti opisani koraci ove procedure u obrnutom redosledu. Ali pre toga, biće opisan nestruktuiran jezik i njegov kriterijum izvršavanja i ispravnosti.

2 Programi i ispravnost

Program se sastoji od *osnovnih blokova*. Svaki osnovni blok sadrži *labelu*, telo i skup *naslednika* koji može biti prazan. Pretpostavljamo da je prvi blok programa označen labelom "Start".

Program može proizvesti skup tragova izvršavanja. Jedan trag izvršavanja je sekvenca programskih stanja, gde svako stanje predstavlja valuaciju promenljivih u programu. Trag može biti beskonačan, može da bude u završnom stanju, da završi greškom ili da bude u stanju iz kog je nemoguće nastaviti dalje izvršavanje - neizvodivom stanju. Svaki trag programa se sastoji od izvršavanja blokova počevši od *Start* birajući na kraju tog bloka proizvoljan naredni blok ako postoji. Trag je u završnom stanju ako ne postoji naredni blok koji bi mogao da se izabere, u stanju greške ako **assert** naredba vraća *false*, i u neizvodivom stanju ako **assume** naredba vraća *false*.

Naredba proizvodi skup konačnih tragova izvršavanja. Naredba $x := E$ proizvodi skup tragova σ, τ u završnom stanju, gde je stanje τ isto kao stanje σ osim što postavlja x na $\sigma(E)$. Naredba **havoc** x postavlja x na proizvoljnu vrednost, proizvodeći skup tragova σ, τ koji završavaju u završnom stanju, gde su σ i τ jednaki za sve promenljive osim eventualno x . Naredba **assert** E proizvodi tragove σ koji završavaju u završnom stanju za $\sigma(E)$ i koji završavaju greškom za $\neg\sigma(E)$. Naredba **assume** E proizvodi tragove σ koji završavaju u završnom stanju za $\sigma(E)$ i koji završavaju u neizvodivom stanju za $\neg\sigma(E)$. Naredba **skip** je zapravo skraćena naredba **assert true**.

Koristeći ove naredbe deo koda:

$$\mathbf{if}(E)\{S\}\mathbf{else}\{T\} \quad (1)$$

možemo napisati kao:

$$Start : \mathbf{skip}; \mathbf{goto} Then, Else \quad (2)$$

$$Then : \mathbf{assume} E; S; \mathbf{goto} End \quad (3)$$

$$Else : \mathbf{assume} \neg E; T; \mathbf{goto} End \quad (4)$$

$$End : \dots \quad (5)$$

3 Najslabiji preduslovi

U ovom poglavlju ćemo definisati najslabije preduslove nestruktuiranih programa. To je poslednji korak u proceduri generisanja uslova za verifikaciju. Pretpostavljamo da je program pasivan, odnosno, da nema naredbi dodela. Ova procedura proizvodi uslov za verifikaciju koji je linearan u odnosu na veličinu pasivnog programa.

Za neku naredbu S i predikat Q koji predstavlja postuslov S , najslabiji preduslov S u odnosu na Q se zapisuje kao $wp(S, Q)$ i to je predikat koji opisuje sve preduslove S za koje izvršavanje ne može da pogreši i za koje svako izvršavanje koje završi, završava u stanju koje zadovoljava Q .

Najslabiji preduslovi pasivnih naredbi se definišu kao:

$$wp(\mathbf{assert}, P, Q) = P \wedge Q \quad (6)$$

$$wp(\mathbf{assume}, P, Q) = P \Rightarrow Q \quad (7)$$

$$wp(S; T, Q) = wp(S, wp(T, Q)) \quad (8)$$

Nestruktuirani programi nemaju strukturne naredbe izbora. Umesto toga, oni imaju **goto** naredbe što je naizgled loša odluka, jer ne želimo da naš graf kontrole toka postane stablo, jer na taj način gubimo sve prednosti koje graf kontrole toka donosi.

Rešenje je sledeće. Za svaki blok:

$$A : S; \mathbf{goto} \quad (9)$$

uvodimo pomoćnu promenljivu A_{ok} . A_{ok} je *true* ako je program u stanju iz kojeg su sva izvršavanja koja počinju iz A tačna. Može se postaviti *jednakost blokova*:

$$A_{ok} \equiv wp(S, \bigwedge_{B \in Succ(A)} B_{ok}) \quad (10)$$

gde $Succ(A)$ označava skup naslednika A , tako da drugi argument za wp predstavlja konjunkciju svih blokova B u tom skupu.

Teorema. Za bilo koji pasivni program P ako je uslov verifikacije validna formula onda je P tačno.

4 Pasivni programi

Pretvorićemo program bez petlji u pasivan program tako što ćemo ga prezapisati u SSA formu (eng. single-assignment form) i onda ukloniti sve naredbe dodele.

DSA forma (eng. dynamic single assignment form) je slična standardnoj SSA formi. Razlika je u tome što u DSA formi može da postoji više od jedne definicije promenljive, ali u bilo kom izvršavanju programa najviše jedna od njih će biti izvršena.

Transformisanje programa bez petlji u DSA formu se radi tako što se posle svakog ažuriranja vrednosti promenljive, ta vrednost identifikuje kao novo pojavljivanje promenljive. Npr. naredbu dodele:

$$x := x + 1 \quad (11)$$

menjamo naredbom

$$x_{i+k} := x_i + 1 \quad (12)$$

gde je x_{i+k} novo pojavljivanje promenljive x .

Sve promenljive koje se nalaze u naredbi se zamenjuju svojim trenutnim vrednostima. Algoritam kojim se ovo radi obilazi graf u topološki sortiranom poretku.

Problem sa uvođenjem novih pojavljivanja promenljivih nastaje u tačkama spajanja odnosno u čvorovima grafa kontrole toka koji imaju više od jednog pretka, zato što taj čvor može da nasleđuje konfliktne vrednosti od svojih predaka. Ako bismo posmatrali primer iz drugog poglavlja, u tački *Start* neka je trenutno pojavljivanje za x jednako x_0 . Neka je trenutno pojavljivanje u tački *Then* jednako x_1 , a u tački *Else* x_2 . Postavlja se pitanje da li u tački *End* uzeti vrednost x_1 ili x_2 . Taj problem se rešava tako što se uvodi novo pojavljivanje x_3 i dodaju se nove naredbe dodele na krajevima blokova B i C : $x_3 := x_1$ i $x_3 := x_2$ redom.

Na ovaj način ćemo transformisati program u DSA formu uvodeći nova pojavljivanja za svaku promenljivu u svakoj tački spajanja što može dovesti do toga da imamo više različitih pojavljivanja nego što je potrebno. Možemo da smanjimo ovaj broj tako što čuvamo skup pojavljivanja za svaki blok. Sva pojavljivanja će imati istu vrednost, tako da kada dodemo do tačke spajanja bilo koje od pojavljivanja iz skupa pojavljivanja predaka može biti korišćeno.

Kad se program transformiše u DSA formu, sve naredbe dodele menjamo naredbama **assume**. Naredbu dodele

$$x_i := E \quad (13)$$

zameni ćemo naredbom

$$\mathbf{assume} x_i = E. \quad (14)$$

Možemo da zamenimo naredbu dodele **assume** naredbom, pošto znamo da vrednost x_i nije korišćena pre njene definicije. Na taj način možemo da pretpostavimo da ima željenu vrednost.

5 Petlje

U ovom poglavlju je opisano kako se reducibilni graf kontrole toka transformiše u aciklični graf kontrole toka. Reducibilni graf kontrole toka

je graf kontrole toka gde je moguće identifikovati jedinstvenu glavu petlje za svaku petlju (u ovom poglavlju se koristi standardna terminologija iz kompajlera [1]).

Da bismo pronašli petlje treba prvo da pronađemo sve povratne grane. Postojanje povratne grane jedinstveno određuje petlju. Povratna grana u grafu kontrole toka je ona grana čiji rep dominira u odnosu na njenu glavu. Za neki čvor P kažemo da dominira nad čvorom Q ako sve putanje do čvora Q prolaze kroz P . *Ulazna tačka petlje* (eng. *loop header*) za povratnu granu B je dominantni čvor koji predstavlja drugi kraj povratne grane koja formira petlju. Ulazna tačka petlje L može imati više od jedne petlje povezane s njom: svaka *prirodna petlja* (eng. *natural loop*) se definiše kao par (L, B) .

Na početku uklanjamo sve povratne grane i na taj način dobijamo aciklični graf. Ali takođe, moramo da budemo sigurni da vrednosti svih promenljivih koje su modifikovane u okviru petlje, imaju vrednost koju bi mogle imati u bilo kojoj iteraciji petlje.

Za svaku prirodnu petlju (L, B) , pravimo skup $H(L, B)$ koji sadrži sve promenljive koje su ažurirane bilo kojom naredbom dodele u bilo kojem bloku petlje. Ove promenljive se nazivaju *ciljevi petlje* (eng. *loop targets*). Za svaki cilj petlje v , u $H(L, B)$, uvodimo **havoc** naredbu i umećemo je na početak L , pre bilo koje naredbe u tom bloku.

Svaka petlja ima svoju *invarijantu*: uslov koji mora biti ispunjen pri svakoj iteraciji kroz petlju. Invarijanta petlje može biti napisana od strane korisnika ili izvedena iz neke druge komponente Spec# statičkog verifikatora programa. Invarijante petlji se kodiraju kao prefiksi **assert** naredbi na početku zaglavlja petlje. Ove **assert** naredbe ne mogu biti validirane ako se bilo koja promenljiva koju sadrže nalazi u $H(L, B)$ i zbog toga uvodimo kopiju ovog niza naredbi za svaki čvor koji je predak od L . Pošto su sada sve tvrdnje proverene, pre skoka u petlju menjamo sve naredbe u **assume** naredbe u okviru L . Obradujemo invarijante petlje na ovaj način pre nego što dodamo **havoc** naredbe i uklonimo sve povratne grane. Rezultujuće **havoc** naredbe praćene **assume** naredbama imaju svrhu zadržavanja informacija o ciljevima petlje u okviru invarijanti petlje.

6 Zaključak

U ovom radu je predstavljena procedura za izračunavanje uslova verifikacije za određeni program da bismo bili u mogućnosti da koristimo automatski dokazivač teorema za verifikaciju programa. Ulaz ne mora biti struktuiran program, već je algoritam efikasan i za nestruktuirane grafove kontrole toka. Takođe, ovaj algoritam se može primenjivati i na struktuirane programe tako što će smanjiti ulazne formule.

Prvobitno je ideja bila da se nestruktuiran program transformiše u struktuiran, ali je taj algoritam bio eksponencijalne složenosti. Dobre heuristike bi mogle popraviti taj algoritam, ali s obzirom da novi algoritam može da se primenjuje i na nestruktuirane programe odustalo se od te ideje.

Trenutno se u ovoj proceduri kao dokazivač teorema koristi Simplify. Nije bilo sigurno da li će Simplify imati dobre performanse koristeći nove uslove verifikacije, ali do sada nije bilo značajnijih problema. Ideja je da se pređe na drugi dokazivač teorema koji za razliku od Simplify-a razdvaja slučajeve koristeći SAT rešavač, jer bi koristeći takav dokazivač teorema

rezultati primene ovakvih uslova verifikacije bili još bolji.

Literatura

- [1] Ravi Sethi Alfred V. Aho and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1987.
- [2] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. 2005. on-line at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.480.4483&rep=rep1&type=pdf>.