

Verifikovanje i falsifikovanje programa sa petljama koristeći predikatsku apstrakciju

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Đorđe Todorović mi2090@alas.matf.bg.ac.rs

28. maj 2018.

Sažetak

Seminarski rad obrađuje temu *verifikovanja programa koji koriste petlje*, tačnije upotrebu *predikatske logike* u istoj. Rad se oslanja na naučni rad [1] napisan na pomenutu temu.

Sadržaj

1	Uvod	2
2	Potpورا	2
2.1	Programi i <i>assert</i> -ovi	2
2.2	Semantika programa	3
2.3	Predikatska apstrakcija i uređivanje	3
3	Kontraprimeri sa petljama	4
3.1	Kako predikatska apstrakcija obrađuje petlje	4
3.2	Detekcija petlji u apstraktnim kontraprimerima	5
4	Primer	6
5	Zaključak	6

1 Uvod

Softversko proveravanje modela (*eng. Model Checking*) pruža automatsku podršku za otkrivanje grešaka u programima. Teorijski je zapravo to dobra ideja, ali u praksi softversko proveravanje modela ne funkcioniše idealno, jer postojeće alate ne karakteriše skalabilnost, često se zarad efikasnosti žrtvuje efikasnost itd..

Apstraktne tehnike mapiraju skupove stanja (*eng. state*) u manje skupove, gde se pri tom aproksimira ponašanje i čuvaju glavne karakteristike sistema. Predikatska apstrakcija je jedna od najpopularnijih i najkorišćenijih metoda za sistematsku redukciju programa kako u smislu prostora tako i stanja. Takvu redukciju nazivamo *stanje-prostor* redukcija.

Predikatska apstrakcija se zasniva na ideji da određeni skup predikata prati određena svojstva promenljivih u datom programu. Upravo taj skup predikata određuje i preciznost apstrakcije. Rezultujući apstraktni model predstavlja jednu preaproksimaciju originalnog programa koja čuva sve izvodljive putanje originalnog programa. Nažalost, ako izaberemo neadekvatan, recimo premali skup predikata, koraci apstrakcije će proizvesti neželjene efekte koji će neretko voditi do pojave grešaka, jedna od njih je pojava neželjenih kontraprimera (*eng. spurious counterexamples*). Da bi se prevazišao pomenuti scenario, predikatska apstrakcija se uparuje sa uređivačkom apstrakcijom, vođenom kontraprimerima (*eng. counterexample-guided abstraction refinement*), poznatijom kao CEGAR. U CEGAR apstrakciji, kontraprimeri se automatski analiziraju da bi dobili dodatne predikate koji bi sprečili neželjena ponašanja. Algoritmi koje srećemo u ovoj oblasti su zasnovani na predikatskoj transformaciji (BR02a) ili interpolaciji (HJMM04), eliminišu neželjene kontraprimere jedan po jedan, ne uzimajući u obzir konstrukcije poput kontrole toka u vidu petlje. U slučaju petlje, uređivački algoritam obično dodaje jedan predikat kako bi proširio kontraprimer tako što dodaje dodatnu iteraciju u svakom ciklusu gde je pronađen kontraprimer, rezultujući tako pojavu dužih, ali i dalje neželjenih kontraprimera. Opisani proces se zaustavlja samo u slučaju da postoji broj iteracija petlje koja proizvodi kontraprimer. Ukoliko takav ne postoji, algoritam može generisati predikate do beskonačnosti i verifikacioni proces se ne zaustavlja u tom slučaju.

Informacija o strukturi same petlje je zapravo dostupna u apstraktnom modelu. Proverivači apstraktnog modela ne prijavljuju putanje sa petljama, jer imaju za cilj da prijave kontraprimere koji su što je moguće kraći. U radu je prikazan pristup koji koristi tu dodatnu informaciju kako bi se zaobošli skupi koraci koji se mogu javiti u radu sa petljama. Sreću se dva slučaja: ukoliko postoji putanja koja obilazi petlju i krši specifikaciju, tada se računa broj petlji koji je doveo do proizvodnje kontraprimera, dok je druga situacija ukoliko ne postoji takav kontraprimer, bilo bi dobro da se redukuje skup predikata koji eliminišu čitavu klasu neželjenih kontraprimera koji prolaze kroz petlju.

2 Potpora

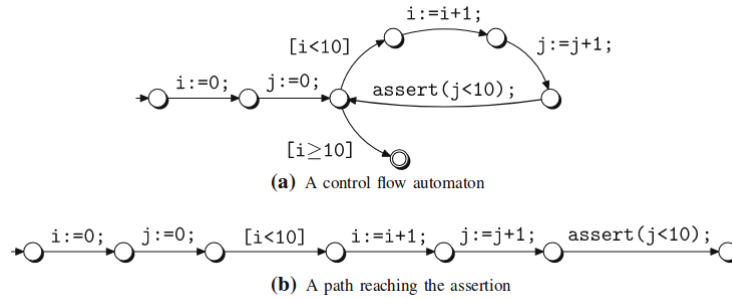
2.1 Programi i *assert*-ovi

Programi su prikazani automatima kontrole toka. Čvorovi su povezani granama, gde je svaka od njih predstavljena nekakvom instrukcijom. Označimo sa C domen valuacije programske promenljive, instrukcija bi

mogla biti preslikavanje domena C u Bulovski domen B . Dalje, pošto radimo sa programima koji bi trebalo da budu bezbedni i korektni, predpostavljamo da neće doći do okidanja *assert*-ova. Zaključak je da ukoliko se *assert* ne okine program je bezbedan.

2.2 Semantika programa

Primer automata na slici 1.



Slika 1: Automat kontrole toka i odgovarajuća putanja

Skup pravila koja povezuju predikate i instrukcije su prikazana na slici 2.

$$\begin{array}{c}
 \frac{}{\{p[x/e]\} \ x:=e \ \{p\}} \text{ assignment} \qquad \frac{}{\{p \Rightarrow q\} \ [p] \ \{q\}} \text{ test} \\
 \\
 \frac{\{p\} \ \pi_1 \ \{q\}, \ \{q\} \ \pi_2 \ \{r\}}{\{p\} \ \pi_1; \pi_2 \ \{r\}} \text{ composition} \qquad \frac{p \Rightarrow q, \ \{q\} \ \pi \ \{r\}, \ r \Rightarrow s}{\{p\} \ \pi \ \{s\}} \text{ consequence}
 \end{array}$$

Slika 2: Logička pravila

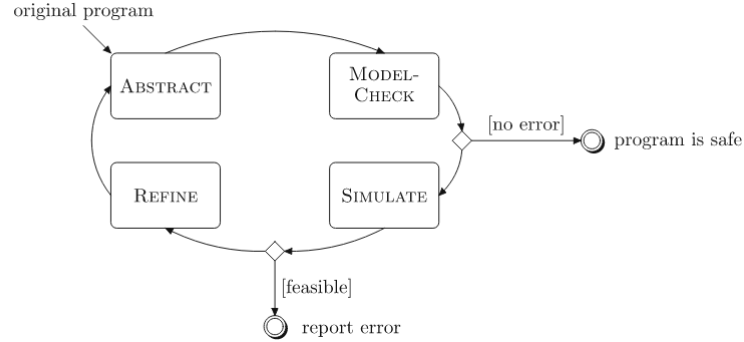
2.3 Predikatska apstrakcija i uređivanje

Predikatska apstrakcija predstavlja tehniku ograničavanja pre- i posle-uslova koji se prate na svakoj lokaciji putanje do Bulovskih kombinacija konačnog skupa predefinisanih predikata P . Postoji konačno mnogo logički neekvivalentnih kombinacija predikata skupa P . Ovo naravno ograničava izražajnost jezika L , gde npr. ne postoji Bulovska kombinacija predikata $(i = 0)$ i $(j = 0)$ kojom bi predstavili predikat $(j < 0)$. Imajući to u vidu, možda je nemoguće dokazati sigurnost neke putanje, iako ista nije okinula *assert*.

Primer :

Pokušavamo da dokažemo sigurnost putanje sa figure 1, koristeći samo Bulovske kombinacije predikata $(i = 0)$ i $(j = 0)$. Pomenuli smo da $(j < 0)$ ne može biti predstavljeno pomoću termova $(i = 0)$ i $(j = 0)$. Ukoliko ne možemo da dokažemo $(j = 0)$ na kraju putanje, ne možemo garantovati sigurnost putanje. Kratak neformalni pregled putanje nam može pokazati da pokušaj mora doživeti neuspeh.

Koristeći Horovu logiku može se pojasniti ograničen skup predikata P.



Slika 3: CEGAR

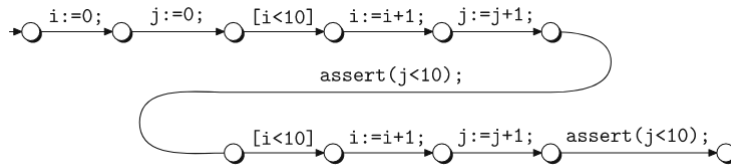
Sumirajući verifikacionu tehniku koja je prikazana na 3, gde se kreće od praznog skupa predikata P, dok sam algoritam koristi apstrakcije predikata da sam izačuna put apstrakcije. Rezultujuće finalno stanje tranzicije sistema analizirano je korišćenjem Alata za proveravanje modela. Ako je apstrakcija pouzdana možemo zaključiti da je originalan program takođe pouzdan. U suprotnom slučaju alati za proveravanje modela pružaju apstraktni kontraprimer. Odgovarajući put je mapiran nazad u originalni program gde je njegova izvodljivost analizirana Horovom logikom. Ukoliko se ispostavi da je kontraprimer neispravan, predikati na koje se naišlo u simulaciji su dodati u skup predikata P (uređeni) i ciklus apstrakcije i uređivanja se ponavlja.

3 Kontraprimeri sa petljama

Tehnika prikazana u prethodnom poglavlju se ne ponaša najbolje ako se *assert* okida prilikom velikog broja petlji. Probaćemo da opišemo heuristike koje ubrazavaju detekciju kontraprimera koji sadrže veliki broj iteracija petlji.

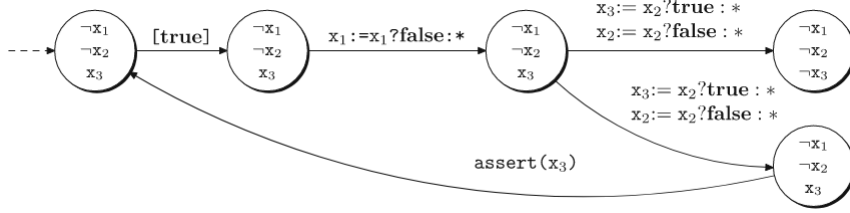
3.1 Kako predikatska apstrakcija obrađuje petlje

Da bi se detektovao kontraprimer koji sadrži nekoliko iteracija u petlji, predikatska apstrakcija mora zahtevati bar jedan predikat za svaku iteraciju u petlji. Na slici 4 se prikazuje takav kontraprimer.



Slika 4: Konkretna putanja sa dve iteracije u petlji

Dodavanjem nekog preuređenog predikata (npr. $j + 1 < 10$) u skup predikata P eliminiše se nepoželjni kontraprimer koji obilazi petlju dva puta. Ali ipak ovaj predikat ne može eliminirati nepoželjni kontraprimer koji obilazi petlju tri puta. Zaključujemo da bi se pronašla nepoželjna putanja koja obilazi petlju deset puta neophodno je dodati sve predikate od $(j + 1 < 10)$ pa sve do $(j + 9 < 10)$. Da bi se ovo postiglo, bar deset koraka preuređivanja je neophodno.

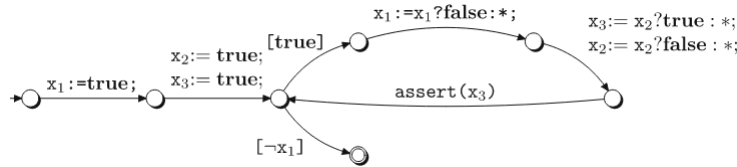


Slika 5: Detekcija potencijalnih petlji u apstraktnoj putanji

3.2 Detekcija petlji u apstraktnim kontraprimerima

Pokažimo na primeru.

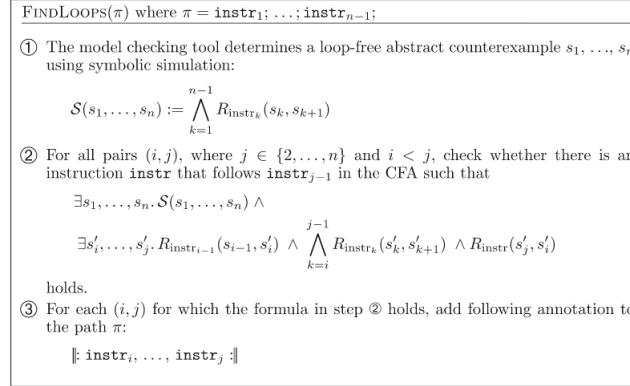
Uzimamo u obzir putanju PI iz primera 4. Istražujući apstraktna stanja odgovarajuće apstratne putanje, koju smo dobili mapiranjem PI u neku putanju apstraktnog sistema iz 6.



Slika 6: Uređena verzija tranzicionog sistema

Počevši od nekog proizvoljnog stanja, dobijamo $x1 = true$, $x2 = true$, i $x3 = true$. Prva iteracija petlje menja ovo stanje u $x1 = false$, $x2 = false$, i $x3 = true$ (prvo stanje u 5). Kada dostignemo tranziciju $x3 := x2 ? true : *; x2 := x2 ? false : *;$ nedeterministička tranziciona funkcija nam nudi izbor: Izmena $x3$ u false, što će uzrokovati okidanje *assert*-a u sledećoj tranziciji, ili da ne menjamo uopšte $x3$ ali da iteriramo kroz petlju još jednom (5). U primeru na slici 6 se može obići petlja bez okidanja *assert*-a.

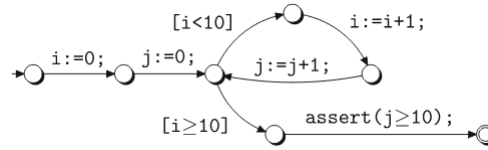
Zaključujemo da nam je jako bitna informacija o detekciji potencijalno nepoželjnih petlji. Alati za proveru modela prikazuju samo putanju PI koja sigurno okida *assert*, ali nam ne prijavljuju potencijalno obidene petlje ovom putanjom. To se može dobiti kreiranjem post-procesiranjem, algoritam prikazan na slici 7:



Slika 7: Algoritam za detekciju petlji u apstraktnom kontraprimeru

4 Primer

Slika 8 predstavlja jedan siguran primer programa. Tradicionalna predikatska apstrakcija, kao što smo napomenuli, zahteva deset koraka preuređivanja da bi se uveli svi neophodni predikati u svrsi pokazivanja sigurnosti programa.



Slika 8: Jedan siguran program

5 Zaključak

Greške su sastavni deo svakog rada, te je njihovo pojavljivanje u računarstvu sasvim uobičajeno. Glavni cilj je iste te greške detektovati i otkloniti na vreme, pre nego što postanu pretnja po čitav sistem. Efikasna i formalna verifikacija sistema je jako važna. Petlje, kao sastavni deo skoro svakog programa predstavljaju jedan od osnovnih konstrukata programskih jezika, te je detekcija grešaka u istim jako važna i formalne tehnike koje obrađuju iste posvećuju posebnu pažnju toj oblasti, ali ujedno predstavljaju jedan od najvećih izazova u proveravanju modela softvera.

Literatura

- [1] Georg Weissenbacher Daniel Kroening. *Verification and falsification of programs with loops using predicate abstraction*. Formal Aspects of Computing, 2009.