Apstrakcija predikata

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Nadežda Bogdanović, 1093/2018
nadezdabogdanovic1@gmail.com

10. decembar 2018

Sažetak

U ovom radu biće prikazana apstrakcija predikata kao tehnika provere modela softvera, koja se oslanja na kombinovanje dva osnovna pristupa u formalnoj verifikaciji, a to su dedukcija i istraživanje. Takođe, biće na konkretnim primerima, radije nego putem formula, teorema i njhovih dokaza, prikazano kako se sam rezultat apstrakcije (apstraktni domen predikata) koristi za proveravanje modela softvera. Ideja vodilja je pristup od vrha ka dnu: apstraktni domen se u slučaju prijavljivanja greške nad modelom profinjuje raščlanjavanjem apstrahovanih predikata, ili dodavanjem novih.

Sadržaj

1 Uvod **2**

2 Primer programa koji se proverava **4**

3 Apstrakcija **6**
3.1 Egzistencijalna apstrakcija 6

4 Proveravanje modela **8**
4.1 Šta znači ako je rešavač vratio grešku? 9

5 Simulacija **10**

6 Profinjavanje **11**
6.1 Alternativa 12

7 Alati **13**

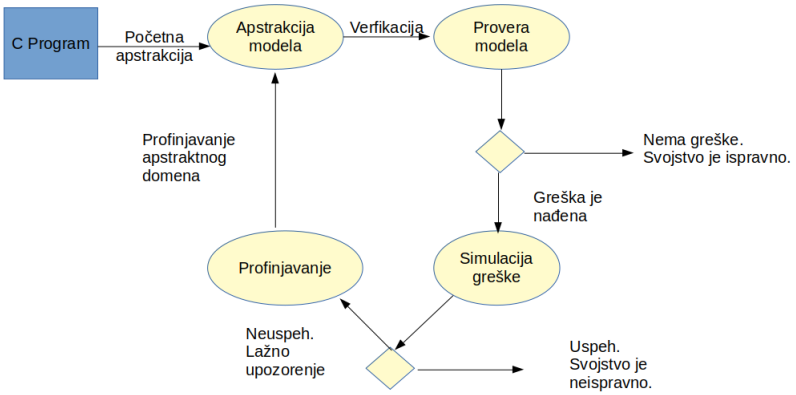
8 Zaključak **14**

Literatura **15**

1 Uvod

Jedan od pristupa provere ispravnosti softvera jeste proverom svih stanja u koje taj softver može da dospe. Vrlo često tih stanja ima jako mnogo ili čak neograničeno i u tom slučaju je potrebno taj broj smanjiti na konačno mnogo tako da taj novi broj stanja bude moguće ispitati u razumnom vremenu. Smanjenje se vrši objedinjavanjem dva ili više istih stanja u jedno - **apstrahovanjem**. Postavlja se pitanje: Kako ćemo znati koja su stanja ista? To su ona stanja koja zadovoljavaju iste **predikate**[1].

Apstrakcija sama po sebi ne bi bila bitna kada se ne bi negde iskoristila i zato se prilikom njenog pominjanja neizostavno opisuje i ciklus u kojem ona učestvuje.



Slika 1: CEGAR metoda

Na slici 1 prikazana je **CEGAR** (eng. *Counterexample-guided Abstraction Refinement*) [2] metoda koja koristi apstrakciju predikata u kombinaciji sa ostalim metodama za proveru modela.

Nakon apstrakcije vrši se **provera modela** (eng. *model checking*) [4] i ako greške nema, svojstvo koje se testira je ispravno, to jest, sistem je bezbedan i ne dospeva u stanje greške.

Ako je greška nađena, putanja koja vodi do greške zapisana u obliku logičkog iskaza se propusti kroz rešavač. Ako je iskazna formula zadovoljiva, postoji greška u programu. Inače greška koja je prijavljena je lažna i naziva se **lažni kontraprimer** (eng. *spurious counterexample*) [4], [2], [6], [5].

Da bi se ova lažna greška otklonila, radi se **profinjavanje** (eng. *refinement*) [4], [2], [6], [5], [1], [3] apstraktnog domena dodavanjem novih predikata i samim tim raščlanjavanje nekih stanja koja su bila spojena u jedno apstraktno stanje.

Zatim se postupak ponavlja. To što je greška bila lažna, ne znači da ona u sebi ne krije pravu grešku, i zato se dodavanjem novog predikata mora opet proći kroz proveru.

[1] Predikati su logički iskazi koji se zadaju nad modelom i moraju važiti tokom celog izvršavanja programa. Ako se prilikom provere modela desi da se iskazu promeni vrednost, ili da jedan ili više iskaza imaju međusobno protivrečne vrednosti, onda kažemo da je sistem dospeo u nebezbedno stanje

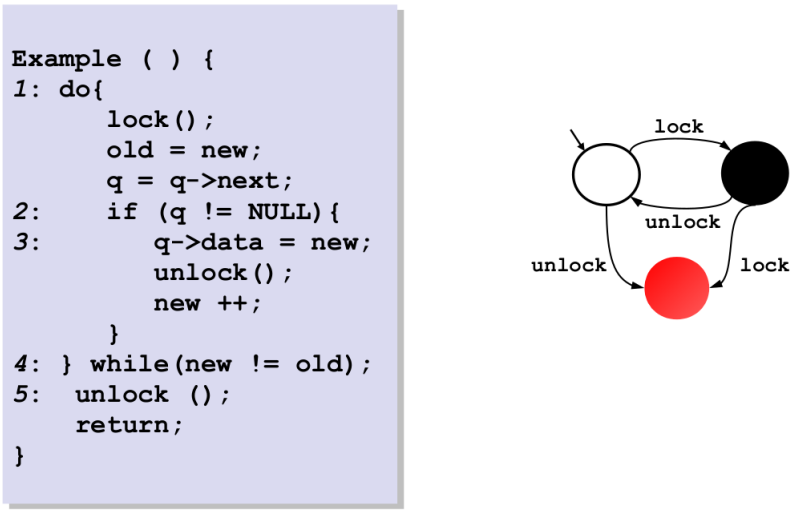
Ova metoda [2]:

- Nikada ne prijavljuje lažnu grešku
- Nikada ne prijavljuje lažnu potvrdu ispravnosti
- Sigurno je konačna za konačne modele
- Ne garantuje zaustavljanje u slučaju beskonačnog modela

U nastavku će svaki od ovih koraka biti prikazan detaljnije kroz primer [4].

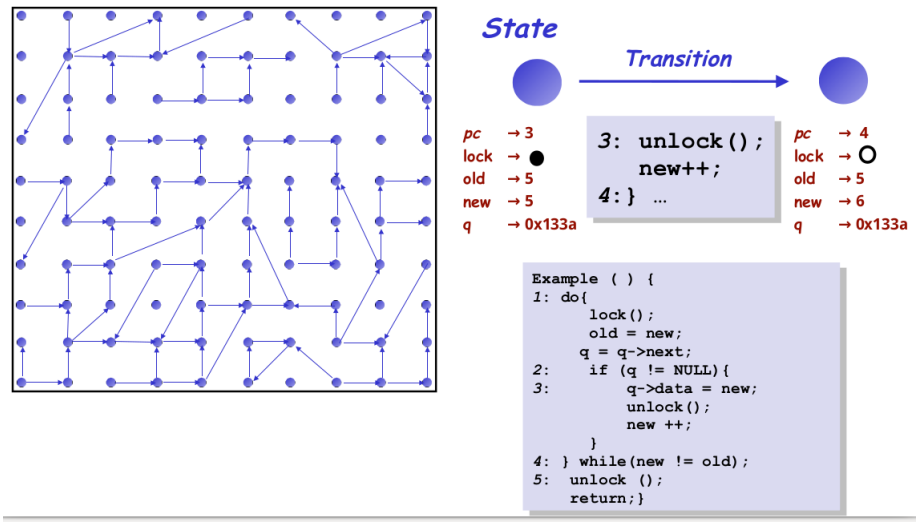
2 Primer programa koji se proverava

Na slici 2 s leva je prikazan program koji želimo da proverimo. Čitanjem koda možemo da primetimo, kako je i pokazano na desnom grafu da program može da dospe u stanje greške kada pokušavamo da otključamo već otključanu sekciju, ili da zaključamo već zaključanu. Sada je potrebno

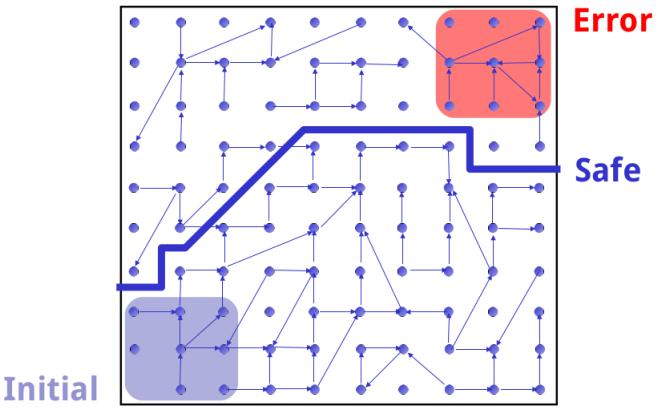


Slika 2: Primer programa koji se proverava

ispitati iz kog je sve stanja moguće doći do stanja greške. To je najlakše videti ako se program predstavi grafom kao na slici 3:



Slika 3: Predstavljanje programa usmerenim grafom: čvorovi su stanja, a grane prelazi



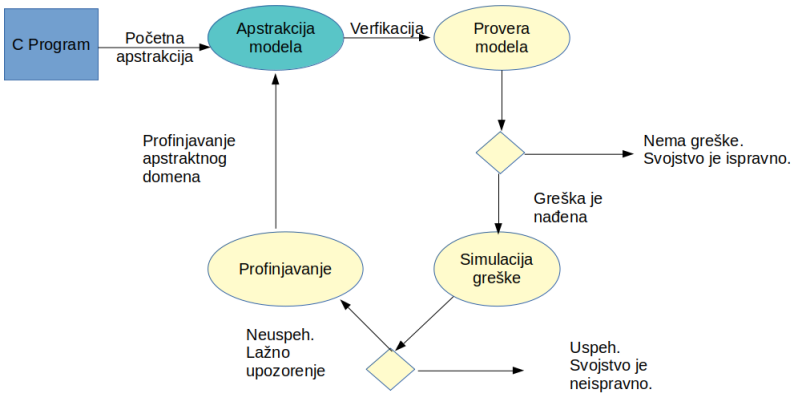
Slika 4: Prikaz početnog stanja, stanja greške i bezbedne putanje kroz program

Ako bolje pogledamo graf, videćemo da zapravo liči na kvadratnu matricu 10x10. To je zato što predloženi program ima 10 naredbi i za svaku je potrebno ispitati na koju drugu se naredbu može preći. Ako u kodu postoji mogućnost za prelaz sa naredbe na naredbu, to se na grafu obeležava strelicom od jednog stanja ka drugom.

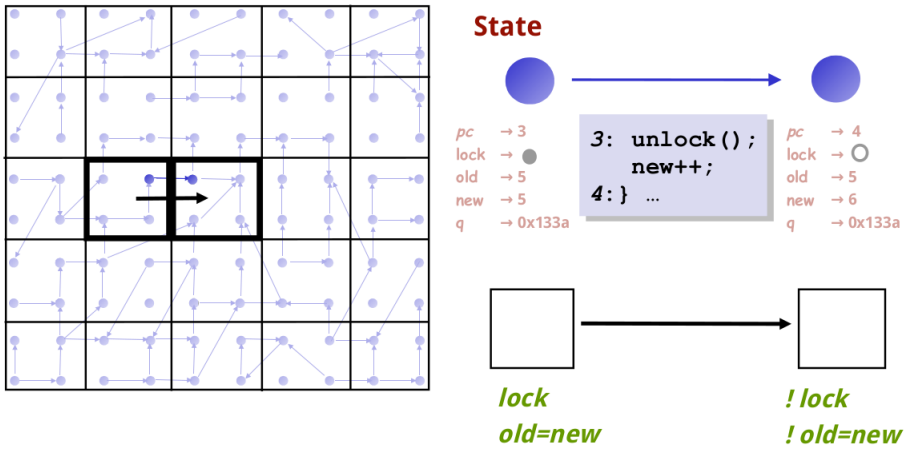
Na slici 4 možemo da vidimo odakle se počinje izvršavanje programa (plavi kvadrat), koja je bezbedna putanja izvršavanja programa (plava linija), kao i kako izgleda zona greške (crveni kvadrat).

3 Apstrakcija

Nakon što smo prikazali sva moguća stanja i prelaze, potrebno je smanjiti taj broj stanja apstrahovanjem, kao što je prikazano na slikama 5 i 6. Stanja koja zadovoljavaju isti predikat ćemo objediniti u jedno. Nad ovim modelom postavićemo predikat **lock**.



Slika 5: Apstrakcija modela



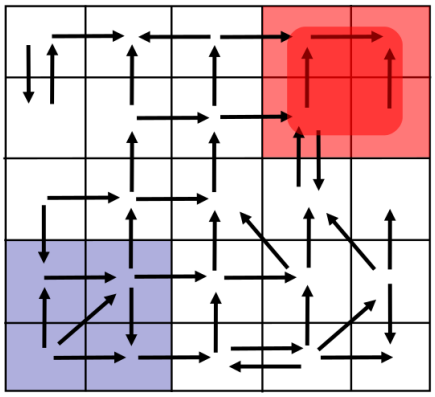
Slika 6: Prikaz apstrakcije stanja

Bitno je da primetimo da su apstraktna stanja konačna i da je njihov broj konačan [4].

3.1 Egzistencijalna apstrakcija

Prilikom apstrakcije se promenljive konkretnog koda zamenjuju bulovskim promenljivama. Svaka Bulovska varjabla odgovara predikatu po-

stavljenom nad varijablom u originalnom kodu. Formalno posmatrano, predikat je *funkcija* [3] koja mapira konkretno stanje r u bulovsku promenljivu. Kada sve predikate primenimo na određeno stanje, dobićemo niz bulovskih promenljivih b, koji predstavlja apstraktno stanje. Ovo preslikavanje u apstraktno stanje je funkcija $b = \alpha(r)$ i zovemo je *apstraktna funkcija*. Primenom skupa predikata na sva stanja dobijamo *apstraktni model*.



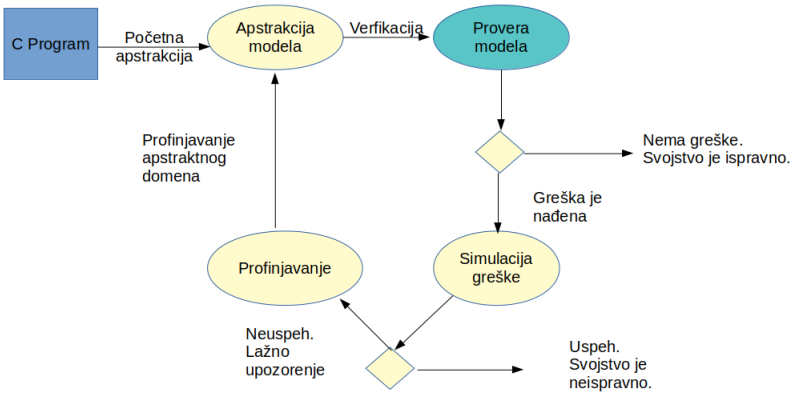
Slika 7: Prikaz apstraktnog modela

Teorema 1 *Stanje apstraktnog modela može da napravi prelaz iz stanja b u drugo stanje b' akko u konkretnom modelu postoji prelaz iz stanja r u stanje r' i b se izvodi apstrahovanjem iz r, a b' iz r'.*

Ovaj dobijeni tranzicioni sistem se naziva **egzistencijalna apstrakcija** (eng. *existential abstraction*) [3]. Egzistencijalna apstrakcija obezbeđuje da ako je apstraktno stanje ispravno, onda su ispravna i sva stanja koja su učestvovala u stvaranju tog apstraktnog stanja. **Tačnije, ona obezbeđuje zadovoljivost - dovoljno je ispitati apstraktni model da bismo utvrdili da li je konkretan model ispravan.**

Apstraktni model za naš primer prikazan je na slici 7.

4 Proveravanje modela

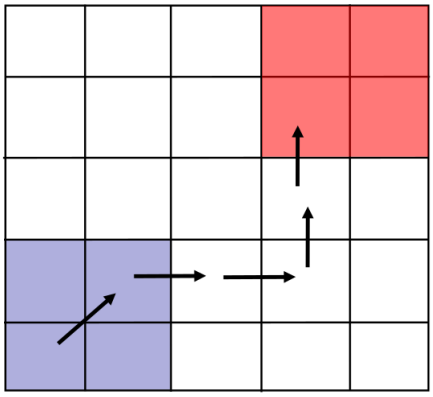


Slika 8: Provera modela

Nakon što je apstraktni model postavljen, potrebno ga je proveriti propuštanjem kroz rešavač, kao što je prikazano na slici 8.

U zavisnosti od toga da li se ispituje ceo kod, deo koda, putanja kroz kod, stanja ili prelazi[2], isptivači mogu koristiti različite strategije.

Jedan od često korišćenih i za implementaciju najlakših pristupa jeste provera dostupnosti određenog stanja [6], [4]. Kako to izgleda u našem primeru, može se videti na slici 9.



Slika 9: Dostizanje stanja greške

Za svako stanje se u listi pamti putanja kako se do tog stanja došlo i onda se proverava da li je stanje greške ikako dostupno [6]. Ovo možda deluje kao neefikasan pristup, ali u mnogome olakšava rad u kasnijim koracima. Postoje optimizacije, koje ne prate sva stanja već samo ona za koja se pregledom koda utvrdilo da mogu biti kritična.

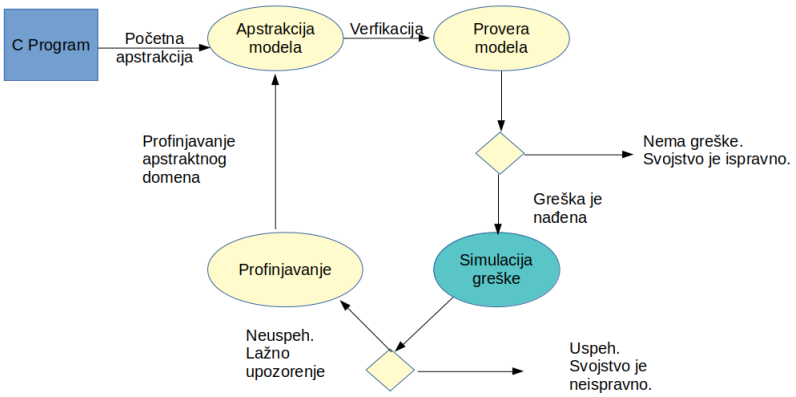
[2] U radu je korišćen kao zajednički termin svojstvo

Ako ispitivač nije vratio grešku, svojstvo koje ispitujemo je ispravno.

4.1 Šta znači ako je rešavač vratio grešku?

Ako je ispitivač vratio grešku u apstraktnom modelu, ne mora da znači i da konkretan model sadrži grešku. Apstraktni model koji smo dobili je zapravo **preapstrahovan** (eng. *over aproximate, over abstract*) [4], [6]. Moglo je da se desi da smo apstrahovali dva stanja za koja nam se, u odsustvu određenog predikata, učinilo da su ista, ali da ona to zapravo nisu. Zato je potrebno da prijavljenu grešku dodatno proverimo, kako bismo utvrdili da li je lažna ili ne.

5 Simulacija



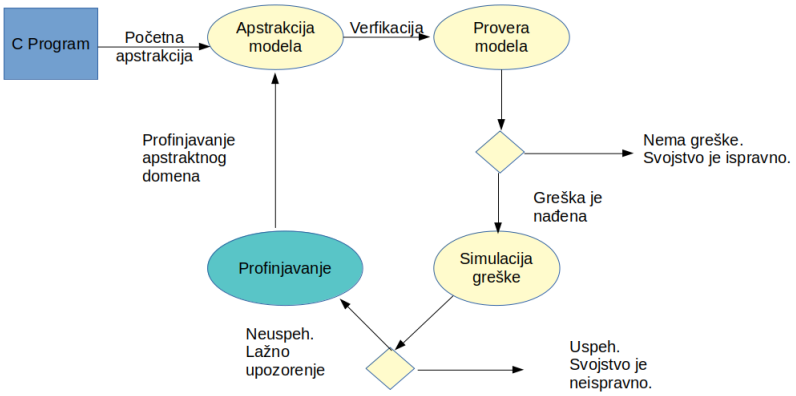
Slika 10: Simulacija greške

U ovom koraku se proverava da li je greška koja je prijavljena zaista greška ili ne, kao što je prikazano na slici 10.

Pošto smo utvrdili u prethodnom koraku da je stanje greške dostižno, sada za to stanje *razvijamo* putanju kako smo do njega došli [6]. To je lako, zato što smo u prethodnom koraku pamtili putanje za stanja i sada je potrebno da tu putanju napišemo u obliku iskazne formule. To možemo da uradimo jer stanja apstraktnog modela koja proveravamo, imaju bulovske vrednosti. Kada se razvije, putanja će biti oblika $\pi = b_1 \wedge b_2 \wedge .... \wedge b_n$. Takvu putanju propustimo kroz SMT ili neki njemu sličan rešavač i ako je kao rezultat dobijeno da je putanja koju smo ispitivali *zadovoljiva*, onda u originalnom kodu zaista postoji greška [3]. U suprotnom, greška koja nam je prijavljena proverom modela je lažna.

6 Profinjavanje

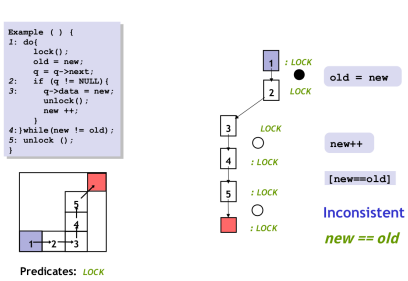
Kako se ispostavilo da je greška lažna, moramo da utvrdimo koja stanja smo apstrahovali, a da to nismo smeli.



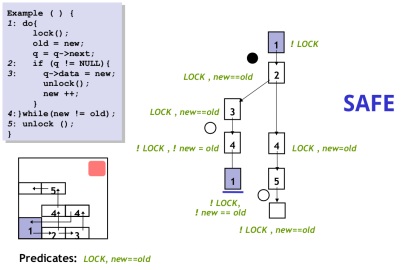
Slika 11: Profinjavanje

Profinjavanje, prikazano na slici 11 podrazumeva dodavanje novog predikata, koji se naziva **odvajajući predikat** (eng. *separating predicate*) [3]. Do njega se dolazi analizom konflikta prijavljenog kontraprimera koja je prikazana na slici 12(a).

Kao što se vidi sa slike, samo jedan predikat **lock** nije dovoljan. Analiza pokazuje da problem nastaje kod zadovoljavanja uslova **new == old** i zato je *kandidat* za odvajajući predikat **new == old**.



(a) Pre dodavanja kandidata prediakta

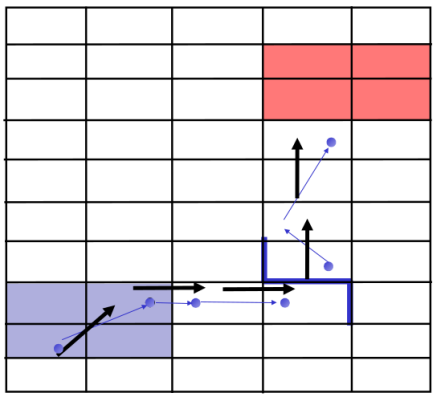


(b) Nakon dodavanja kandidata predikata

Slika 12: Analiza konflikta

Zatim ponovo radimo proveru kontraprimera, ali ovog puta sa kandidatom predikatom. Kao što pokazuje slika 12(b), stanje sistema je sa ovim predikatom bezbedno i ovaj *kandidat predikat* postaje *odvajajući predikat* i dodaje se u skup predikata.

Dodavanjem novog predikata menja se i stanje apstraktnog modela, što se vidi na slici 13. Mesto umetanja novog predikata označeno je plavom linijom, odakle se vidi kako je novi predikat koji se umeće dobio naziv *odvajajući*.



Slika 13: Promena apstraktnog modela

Međutim, ovo nije kraj. Pošto se sam model promenio, potrebno je još jednom proći kroz CEGAR ciklus, kako bismo se uverili da je model zaista ispravan.

6.1 Alternativa

Ova analiza kontraprimera koju smo videli naziva se **analiza po najjačem preduslovu** (eng. *strongest precondition*) [2], [4], [3].

Drugi način podrazumeva obilazak kontraprimera u suprotnom smeru: od kraja koda ka početku i naziva se **analiza po najslabijem preduslovu** (eng. *weakest precondition*) [2], [4], [3]. Prednost ovog načina analiziranja je u tome što nam pruđa u vid u to koji je osnovni preduslov koji moramo da ispunimo da bi stanje sistema bilo ispravno, a često je taj podatak jedini koji sebi možemo da priuštimo. Međutim, za razliku od prethodnog načina, ne pruža uvid u celokupno stanje sistema i teže je za implementaciju.

7 Alati

Neki od alata [2], [4] u industriji koji podražavaju rad apstrakcije predikata ili se koriste u pojedinačnim koracima CEGAR modela:

- SMV: alat za proveru stanja konačnog modela i generisanje kontra-primera
- SLAM (Microsoft): proverava da li se softver ponaša u skladu sa željenim zahtevima
- SLIC: zasnovan na radu konačnih automata. Koristi za praćenje ponašanja C koda, proveru redosleda izvršavanja sekvenci koda i promene vrednosti promenljivih
- MAGIC (CMU): koristi se za verifikaciju konkurentnih C programa
- BLAST model checker (Berkely): koristi se za proveru modela C programa
- SATABS (CMU): koristi se za proveru pokazivačku i bit-vektorsku aritmetiku

8 Zaključak

Apstrakcija predikata je moćna tehnika za proveru modela. Koristi se kako za verifikaciju softvera, tako i za verifikaciju hardvera. Osim toga, kako je zasnovana na pretrazi i razvijanju putanja, u teoriji se može koristiti kao pomoć heuristikama za mašinsko učenje. Pošto je njen cilj smanjivanje već postojećih stanja na razuman nivo, teoretski se može koristiti za izgradnju i proveru bioloških, hemijskih i fizičkih modela, kao i za modeliranje i praćenje poslovnih procesa i organizacije.

Međutim, ova tehnika, iako značajna i primenljiva u velikom broju ne samo računarskih oblasti, naišla je na slabu primenu u industriji u odnosu na svoje mogućnosti (veliki broj alata ima mogućnost provere samo C koda). Razlog leži u tome što ova tehnika, za samo konstruisanje valjanog apstraktnog modela zahteva dobro poznavanje teorije automata, a za proveru tog modela i poznavanje iskazne logike i načine ispitivanja zadovoljivosti. Takođe, ova tehnika nije uvek potpuno ista za sve modele, nego se mora prilagođavati samom modelu nad kojim se sprovodi. Povrh svega, da bi se garantovala ispravnost, svaki od koraka se mora i formalno dokazati.

Konačno, u ovoj oblasti se očekuje u skorijoj budućnosti napredak, kako u razvoju optimizacija tehnike, tako i u razvoju samih alata, paralelno sa napretkom u samoj oblasti teorije računarstva za koju je usko vezana.

Literatura

[1] Patrick Cousot and Radhia Cousot. On abstraction in software verification. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, pages 37–56, 2002.

[2] Daniel Kroening. Predicate abstraction: A tutorial, 2012.

[3] Daniel Kroening and Sanjit A. Seshia. Formal verification at higher levels of abstraction. In *Proceeding ICCAD '07 Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 572–578, 2007.

[4] Mathias Payer. Jpf2: Predicate abstraction.

[5] Edgar Pek and Nikola Bogunović. Predicate abstraction based verification framework: First results. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, Zagreb, 2002.

[6] Andreas Podelski Ranjit Jhala and Andrey Rybalchenko. *Predicate Abstraction for Program Verification*, chapter 15. Springer, Cham, 2018.