

Neograničena provera modela softvera kroz inkrementalno SAT rešavanje

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Ivan Ristović

april 2018.

Sažetak

U ovom radu je opisana nova tehnika neograničene provere modela softvera za pronalazak grešaka programa pisanih u programskom jeziku C, koristeći inkrementalne SAT rešavače. Od polaznog programa se pravi *DimSpec* formula. *DimSpec* formula predstavlja konjunkciju četiri KNF formule koje kodiraju početno, krajnje i svako tranziciono stanje programa, kao i prelaze između susednih stanja programa. Proces dobijanja *DimSpec* formule od izvornog C koda se odvija u više koraka. Prvo se C kod prevodi u *LLVM* module, koji predstavljaju međureprezentaciju između C koda i mašinskog koda. Ti moduli se zatim prevode u *DimSpec* formulu korišćenjem alata *LLUMC* tako da ciljno stanje predstavlja stanje greške. Tako dobijena *DimSpec* formula se može rešiti pomoću inkrementalnih SAT rešavača ili *IC3* algoritmom za proveru invarijanti. Rešenje prestavlja dokaz neispravnosti programa jer je pronađen način da se iz polaznog stanja stigne u stanje greške. Ovakvo kodiranje proširuje funkcionalnost tradicionalne ograničene provera modela softvera jer pokriva velike i beskonačne petlje, uz održavanje pristojnih vremenskih performansi.

Sadržaj

1	Uvod	2
2	Pomoćne tehnike i alati	2
2.1	Inkrementalno SAT rešavanje	2
2.2	LLVM međureprezentacija	3
2.3	Ograničena provera modela	3
2.4	DimSpec formule	3
3	LLUMC kodiranje stanja programa	4
3.1	Kodiranje stanja programa	5
3.2	Kodiranje prelaza između stanja	5
3.3	Prevođenje SMT formule u SAT formulu	6
4	Zaključak	6
	Literatura	6

1 Uvod

Vrlo je važno iscrpno testirati softver od čijeg izvršavanja zavise ljudski životi ili čije potencijalne greške mogu imati velike materijalne gubitke. Tehnike verifikacije softvera pružaju veliko pokriće programskog koda uz formalni dokaz ispravnosti. Postoje razne tehnike verifikacije softvera, npr. *simboličko izvršavanje* [6] i *ograničena provera modela softvera* [3].

U ovom radu je opisana tehnika *neograničene provere modela softvera* [4]. Glavna razlika u odnosu na tradicionalnu ograničenu proveru modela je uklanjanje ograničenja za razvijanje petlji. Naime, ograničena provera modela zahteva razvijanje petlji unapred zadati broj puta zbog neodlučnosti zaustavljanja proizvoljnog programa. Neograničena provera modela ne zahteva razvijanje petlji i stoga predstavlja alat za verifikaciju programa koji u sebi imaju potencijalno beskonačne petlje. Istraživanje je fokusirano na sekvencijalnim C programima i reprezentaciji niskog nivoa koju generiše LLVM [1]. Na osnovu ove reprezentacije se pravi *DimSpec* formula pomoću alata LLUMC [2]. *DimSpec* formula se sastoji od četiri KNF formule koje kodiraju stanja programa, naime: početno stanje, krajnje stanje, međustanja i tranzicije između susednih stanja. *DimSpec* formula se dalje prosleđuje inkrementalnim SAT rešavačima.

Pretpostavlja se da čitalac poseduje znanje iskazne logike, logike prvog reda i problema zadovoljivosti logičkih formula. U poglavlju 2 će biti opisane tehnike i alati na koje se oslanja metoda neograničene provere modela. Definicija stanja programa i proces kodiranja stanja programa u *DimSpec* formulu biti opisan u poglavlju 3.

2 Pomoćne tehnike i alati

Tehnika neograničene provere modela koristi postojeće alate i tehnike verifikacije softvera. Prvo se uz pomoć alata LLVM [1] od polaznog C koda dobijaju programski moduli koji predstavljaju ulaz za alat LLUMC [2]. LLUMC kodira module u *DimSpec* format koji predstavlja validnu logičku formulu koja opisuje stanja modula i prelaza između susednih stanja. *Inkrementalno SAT rešavanje* [5] se koristi za nalaženje modela za takvu *DimSpec* formulu. U ovom poglavlju će biti više reči o navedenim alatima, jer je njihova uloga važna u čitavom procesu neograničene provere modela.

2.1 Inkrementalno SAT rešavanje

Proces inkrementalnog SAT rešavanja [5] predstavlja inkrementalno dodavanje klauza tokom rešavanja problema zadovoljivosti. U tu svrhu se koriste funkcije $add(C)$ i $solve(A)$, gde je C klauza a A skup literala koji se nazivaju pretpostavke (engl. *assumptions*). Klauze se dodaju korišćenjem funkcije add i njihova konjunkcija se rešava pod pretpostavkom da su vrednosti svih literala iz A tačne, koristeći $solve(A)$.

Dodavanje nove klauze se vrši dodavanjem $C \vee a$, gde je a neiskorišćeno iskazno slovo. Klauza postaje relevantna čim se doda literal $\neg a$ u skup A . Literal a se naziva *aktivacioni literal*. Ukoliko se $\neg a$ ne doda u skup A , to je ekvivalentno uklanjanju C iz skupa klauza.

2.2 LLVM međureprezentacija

LLVM [1] je open-source compiler framework projekat. Za potrebe neograničene provere modela se koristi *LLVM* međureprezentacija C koda. Razlog tome je što je jako kompleksno analizirati C kod i skoro nemoguće je podržati sve alate i biblioteke. S druge strane, *LLVM* međureprezentacija direktnije predstavlja instrukcije i vrši razne optimizacije i uprošćavanja, što je čini idealnom za analizu. *LLVM* međureprezentacija koda se sastoji od više modula, gde se svaki *LLVM* modul može grupisati u nekoliko jedinica:

- instrukcija
- osnovni blok
- funkcija

2.3 Ograničena provera modela

Prilikom primene tehnike ograničene provere modela [3], svaka petlja koja se javlja u programu se razvija k puta (otuda naziv *ograničena*). Razlog razvijanja petlji je neodlučivost zaustavljanja proizvoljnog programa. Razvijanje petlji omogućava odlučivost ove tehnike, ali takođe predstavlja ozbiljno ograničenje. Nakon razvijanja petlji i kodiranja stanja programa u logičke formule, konstruiše se formula koja predstavlja negaciju željenih svojstava (specifikacije programa) i to se skupa predaje SAT rešavaču. Ukoliko rešavač nađe model za datu formulu, program ne prolazi verifikaciju i nađeni model se može iskoristiti kao test-primer za koji program ne prati specifikaciju.

Postoje razni alati koji implementiraju ograničenu proveru modela. Jedan od njih je alat *LLBMC*, koji je dizajniran da verifikuje programe od kojih zavisi bezbednost ugrađenih sistema. *LLBMC* predstavlja inspiraciju za alat *LLUMC*, koji je ekvivalent za tehniku neograničene provere modela.

2.4 DimSpec formule

DimSpec formula kodira stanja programa, u oznaci t_0, \dots, t_k gde svako stanje predstavlja istinitosne vrednosti n bulovskih promenljivih x_0, \dots, x_n . *DimSpec* formula se sastoji od četiri formule: \mathcal{I} , \mathcal{U} , \mathcal{G} i \mathcal{T} . \mathcal{I} predstavlja skup inicijalnih klauza, tj. onih klauza koje stanje t_0 zadovoljava. \mathcal{G} predstavlja skup završnih klauza, tj. onih koje stanje t_k zadovoljava. \mathcal{U} predstavlja skup klauza koje su zadovoljene stanjima t_i . \mathcal{T} predstavlja skup klauza koje su zadovoljene parovima susednih stanja t_i, t_{i+1} . Svako stanje u sebi sadrži vrednosti n promenljivih, tako da ukupno imamo kn promenljivih.

Testiranje da li je završno stanje dostižno iz početnog u k koraka, je ekvivalentno ispitivanju zadovoljivosti formule

$$F_k = \mathcal{I}(0) \wedge \left(\bigwedge_{i=0}^{k-1} (\mathcal{U}(i) \wedge \mathcal{T}(i, i+1)) \right) \wedge \mathcal{U}(k) \wedge \mathcal{G}(k),$$

gde $\mathcal{I}(i), \mathcal{U}(i), \mathcal{G}(i), \mathcal{T}(i, i+1)$ predstavljaju formule u kojima je svaka promenljiva x_j zamenjena sa x_{j+in} - njenom vrednošću u i -tom bloku. Jedan od načina da se nađe najmanji broj koraka za koje se završno stanje dostiže iz polaznog je da se rešavaju formule F_1, F_2, \dots sve dok se ne nađe

zadovoljiva formula. Efikasan način da se ovo implementira je korišćenjem inkrementalnog SAT rešavanja:

$$\begin{aligned} \text{step}(0) & : \text{add}(\mathcal{I}(0) \wedge (a_0 \vee \mathcal{G}(0)) \wedge \mathcal{U}(0)) \\ & \quad \text{solve}(\text{assumptions} = \neg a_0) \\ \text{step}(k) & : \text{add}(\mathcal{T}(k-1, k) \wedge (a_k \vee \mathcal{G}(k)) \wedge \mathcal{U}(k)) \\ & \quad \text{solve}(\text{assumptions} = \neg a_k) \end{aligned}$$

Za razliku od ograničene provere modela, tokom primene neograničene provera modela ne proverava se zadovoljivost negirane formule koja kodira stanja programa kako bi se našao kontraprimer. Umesto toga se kao završna stanja smatraju stanja greške (više reči o ovome u poglavljima koji slede) i pokušava se pronaći put od početnog stanja do stanja greške.

3 LLUMC kodiranje stanja programa

Stanja programa posmatramo kao stanja svakog osnovnog bloka *LLVM* modula zasebno. Posmatramo takođe i tranzicije između susednih stanja. Svako stanje se sastoji od promenljivih, a svaka promenljiva se kodira kao bit-vektor dužine n .

Uvodimo dva specijalna stanja:

- *ok* stanje - iz ovog stanja ne mogu nastati greške
- *error* stanje - stanje greške

Pre opisa procesa kodiranja *LLVM* modula u *DimSpec* formulu, potrebno je definisati pojam softverske greške za alat *LLUMC*. Jasno je da je nemoguće pokriti sve moguće greške. Stoga se u nastavku ograničavamo na greške *prekoračenja*. Možemo definisati prekoračenje nezavisno od tipa promenljivih i samim tim nezavisno od reprezentacije promenljive preko njenog bit-vektora. Neka je v promenljiva u potpunom komplementu i neka je l dužina bit-vektora koji reprezentuje v . Neka \max_l vraća maksimalnu vrednost za v reprezentovanu bit-vektorom dužine l . Tada je $\max_l = 2^{l-1} - 1$. Slično, neka \min_l vraća minimalnu vrednost. Tada je $\min_l = -2^{l-1}$. Ukoliko se prilikom sabiranja promenljivih x i y dobija broj veći od \max_l , onda postoji prekoračenje. Prekoračenje se slično definiše za ostale aritmetičke operacije.

Takođe je potrebno uvesti funkcije **assume** i **assert**. Program se ponaša po specifikaciji ukoliko svi pozivi funkcije **assert** vrate **true** pod uslovom da su svi uslovi iz poziva **assume** zadovoljeni. Ukoliko **assume** uslov nije ispunjen, ponašanje programa je nedefinisano i stoga se ne mogu pojaviti greške. Koristeći ove funkcije možemo definisati grešku za program *LLUMC*.

Definicija 3.1. Neka je p program. Greška u programu p postoji ukoliko su svi pozivi funkcije **assume** pre poziva funkcije **assert** ili mogućeg prekoračenja vratili **true** i važi jedno od navedenog:

1. Funkcija **assert** je vratila **false**.
2. Desilo se prekoračenje prilikom izvršavanja aritmetičke operacije.

Bijekcijom $\text{enc}(\text{block}) : \text{BasicBlocks} \rightarrow \mathbb{N}$ svakom osnovnom bloku dodeljujemo jedinstven prirodan broj koji se može predstaviti odgovarajućim bit-vektorom. Ukoliko je N broj osnovnih blokova u programu, potrebno je $\lceil \log_2 |B| + 2 \rceil$ bitova za jedinstveno enkodiranje tih blokova. Promenljivu koja kodira trenutni blok označavamo sa *curr*. S obzirom da vrednosti promenljivih mogu da zavise od vrednosti u prethodnom bloku,

uvodimo promenljivu *prev* koja kodira prethodni blok. Sada možemo da kodiramo skupove $\mathcal{I}, \mathcal{G}, \mathcal{U}, \mathcal{T}$.

3.1 Kodiranje stanja programa

Naš cilj je da pronađemo enkodiranje koje LLVM modul definisan u sekciji 2.2 prevodi u *DimSpec* formulu. Stoga treba da definišemo četiri SMT formule $\{\mathcal{I}, \mathcal{G}, \mathcal{U}, \mathcal{T}\}$ takve da ukoliko postoji tranzicija od \mathcal{I} do \mathcal{G} definisana po \mathcal{T} i ograničena sa \mathcal{U} tada postoji greška u datom programu. Zatim ćemo dobijene SMT formule prevesti u KNF formule, što nam kao rezultat daje *DimSpec* formulu. U nastavku će biti ukratko opisan postupak enkodiranja, dok se detaljni postupak može naći u [4].

Definicija 3.2. Neka *entry* označava prvi osnovni blok programa. Tada se inicijalna formula $\mathcal{I}(k), k \in \mathbb{N}$ definiše kao:

$$\begin{aligned} curr &= enc(entry) \quad \wedge \\ prev &= enc(entry) \end{aligned}$$

Definicija 3.3. Neka *error* označava osnovni blok greške. Tada se ciljna formula $\mathcal{G}(k), k \in \mathbb{N}$ definiše kao:

$$curr = enc(error)$$

Univerzalna formula \mathcal{U} se sastoji od uslova koji moraju biti tačni u svakom stanju. Iz prethodne analize znamo da je broj bitova potrebnih za enkodiranje stanja $\lceil \log_2 |B| + 2 \rceil$. S obzirom da $|B| + 2$ nije nužno stepen dvojke, potencijalno smo enkodirali više brojeva. Ovi brojevi se moraju izostaviti u formuli \mathcal{U} .

Definicija 3.4. Neka je $|B|$ broj osnovnih blokova u *LLVM* modulu. Tada se univerzalna formula $\mathcal{U}(k), k \in \mathbb{N}$ definiše kao:

$$\begin{aligned} curr &\leq |B| + 2 \quad \wedge \\ prev &\leq |B| + 2 \end{aligned}$$

3.2 Kodiranje prelaza između stanja

Enkodiranje tranzicione formule \mathcal{T} ima oblik:

$$stanje(k) \Rightarrow stanje(k + 1)$$

Međutim, grananja predstavljaju problem, kao i pozivi funkcija. Na osnovu toga kojom instrukcijom se blok završava, potrebno ga je drugačije enkodirati. Zarad jednostavnosti, uvešćemo funkciju enkodiranja koja vrši enkodiranje blokova u zavisnosti od toga kojom instrukcijom se završavaju.

$$encode : BasicBlocks \rightarrow SMTFormulas$$

Sada možemo definisati tranzicionu formulu:

Definicija 3.5. Neka je *BB* skup svih osnovnih blokova u *LLVM* modulu i neka je $encode(b), b \in BB$ funkcija enkodiranja opisana iznad. Tada se tranziciona formula $\mathcal{T}(k, k + 1), k \in \mathbb{N}$ definiše kao:

$$\bigwedge_{b \in BB} encode(b)$$

3.3 Prevođenje SMT formule u SAT formulu

Enkodiranje opisano iznad nam kao rezultat daje četiri SMT formule. Ove formule je potrebno prevesti u KNF formule. Najčešće se to radi koristeći tehniku *bit-blasting* [7]. Nakon toga, dobijenu KNF formulu u *DimSpec* formatu možemo proslediti inkrementalnim SAT rešavačima. Alat *LLUMC* automatski vrši ovu transformaciju i kao rezultat vraća *DimSpec* formulu.

4 Zaključak

Neograničena provera modela proširuje skup programa koji se mogu verifikovati tehnikama provere modela tako što uklanja granicu koja je prisutna kod ograničene provere modela što omogućava verifikovanje programa koji imaju beskonačne petlje. Enkodiranjem stanja programa u *DimSpec* format nam omogućava da koristimo već postojeće SAT rešavače za nalaženje modela. Takođe se mogu koristiti algoritmi provere invarijanti i paralelno SAT rešavanje kako bi se dobilo na performansama. Sledeći korak za unapređenje ove tehnike bi verovatno bila primena teorije nizova zarad primene na većem skupu programa.

Literatura

- [1] The LLVM compiler infrastructure. on-line at <http://llvm.org/>.
- [2] LLUMC (Low Level Unbounded Model Checker), 2017. on-line at <https://github.com/MarkoKleineBuening/llumc>.
- [3] Biere A., A. Cimatti, Clarke E.M., Strichman O., and Y. Zhu. *Bounded model checking*. 2003. In: Advances in computers 58.
- [4] Marko Kleine Büning, Tomas Balyo, and Carsten Sinz. *Unbounded Software Model Checking with Incremental SAT-Solving*. 2018. on-line at <https://arxiv.org/pdf/1802.04174.pdf>.
- [5] Eén N. and Sörensson N. *An extensible sat-solver*. 2003. In: Advances in computers 58.
- [6] Khurshid S., Păsăreanu C.S., and Visser W. *Generalized symbolic execution for model checking and testing*. 2003. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems.
- [7] Yakir Vizel, Alexander Nadel, and Sharad Malik. *Solving Constraints over Bit-Vectors with SAT-based Model Checking*. 2017. on-line at http://smt-workshop.cs.uiowa.edu/2017/papers/SMT2017_paper_8.pdf.