Eksplozija putanja i regulisanje broja putanji(odsecanje nedostižnih putanji i sumiranje funkcija i petlji)

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Filip Lazić, 1101/2017
filipl41@yahoo.com

11. decembar 2018

Sažetak

U ovom tekstu su ukratko opisane glavne tehnike koje su doprinele velikoj primeni simboličkog izvršavanja u industriji. Čitalac posle čitanja ovog rada biće upoznat sa glavnim tehnikama koje su rešile veći deo glavnog problema u izvršavanju i sa budućim istraživanjima u ovoj oblasti.

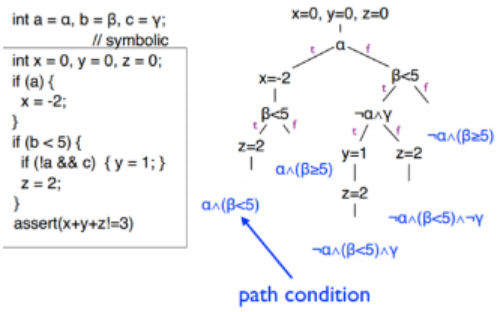
Sadržaj

1 Uvod

Simboličko izvršavanje je popularna tehnika analize programa nastala 70-ih godina prošlog veka. Cilj simboličkog izvršavanja je da utvrdi da li određena svojstva programa može da pokvari neki deo koda. Iako je relativno dugo postojala ova ideja, proboj je nastao tek od 2005.godine kada su izbačeni alati:

- DART Godefroid and Sen, PLDI 2005 (koji uvodi dinamičko izvršavanje u simboličko izvršavanje)
- EXE Cadar, Ganesh, Pawlowski, Dill, and Engler, CCS 2006 (STP: podška za teoriju nizova)

Osnovna ideja simboličkog izvršavanja je izvršavanje programa nad simboličkim vrednostima, to jest preslikavanje konkretnih vrednosti promenljivih u simboličke vrednosti. Sve moguće putanje u programu, to jest svi mogući ulazi, predstavljaju stablo izvršavanja.



Slika 1: Simboličko drvo

Na slici 1 je prikazan primer programskog koda i odgovarajućeg stabla izvršavanja.

U teoriji simboličko izvršavanje deluje lako, dovoljno je proveriti svaku moguću putanju i videti gde eventualno postoji problem. U praksi je ovo nemoguće, jer svaki iole ozbiljniji program ima previse mogućih putanja koje se ne mogu sagledati u realnom vremenu. Takođe postoje problemi modelovanja hipa i rezonovanja o pokazivačima kao i modelovanja okoline. U ovom radu će biti opisano kako se razrešava problem eksplozije putanja

2 Motivacija

Kao što je u uvodu naglašeno najveći problem simboličkog izvršavanja je eksplozija putanja. Glavni uzroci ekspolzije putanja su petlje i pozivi funkcija. Svaka iteracija petlje vodi ka novoj uslovnoj grani u simboličkom drvetu i lako dolazi do kombinatorne eksplozije. Ukoliko uslov pelje sadrži jednu ili više simboličkih vrednosti, broj generisanih grana može potencijalno biti beskonačan, što sledeći primer pokazuje :

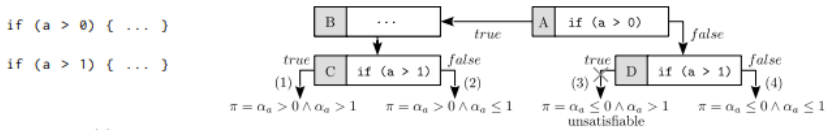
Glavne strategije za rešavanje problema eksplozije putanja na koje ćemo najviše obratiti pažnje u ovom radu su:

- Odsecanje nedostižnih putanja(eng. *pruning unrealizable paths*)
- Sumiranje funkcija i petlji (eng. *function and loop summarization*)

3 Odsecanje nedostižnih putanja

Prvo što nam prirodno pada na pamet kada razmislimo o problemu eliminacije putanja, je da eliminišemo putanje i stanja do kojih sigurno nećemo doći. To radimo pomoću SMT rešavača, ukoliko rešavač može da dokaže da postoji putanja koja je nezadovoljiva, to jest nijedna kombinacija ulaznih vrednosti nije bitna, pa ovu granu ne moramo ni razmatrati, odnostno možemo je bezbedno odbaciti.

Na slici 2 prikazan je primer ove strategije.



Slika 2: fragment koda i primer odgovarajućeg simboličkog izvršavanja

Postoje 2 različita pristupa ovoj strategiji : pohlepna evaluacija(eng. *eager evaluation*) i lenja evaluacija(eng. *lazy evalution*).
Kod pohlepne evaluacije SMT rešavač se poziva za svaku granu, dok lenja evaluacija koristi posebne heuristike i tako redukuje broj putanja koje proverava.
Kako SMT rešavač može da istraži veliki prostor pretrage ali samo jedne putanje odjednom, često će završiti proveravajučći iste delove koji se nalaze u različitim putanjama, zbog čestih preklapanja. Strategija koju koristi lenja evaluacija je da se iz svake putanje izdvoji minimalno nezadovoljivo jezgro(eng. *unsat core*) i tako eliminišemo što je više iskaza moguće održavajući nezadovoljivost. Pomoću ove tehnike mogu se izbaciti putanje koje imaju isto minimalno jezgro, i tako se njihov broj redukuje.

4 Sumiranje funkcija i pelji

Često se pojedini delovi koda izvršavaju više puta, najčešće je to slučaj kod petlji ali i kod funkcija. Da bi se izbegao isti posao simbolički egzekutor sumira ova izvršavanja i koristi već izračunate analize za ponovnu upotrebu.

4.1 Sumiranje funkcija

Ukoliko imamo neku funkciju f ona može biti pozvana više puta tokom izvršavanja, pa radi poboljšanja perfomansi koristimo sumiranje. Ova tehnika hvata efekte poziva funkcije u novoj formuli ϕ_w, koja sjedinjava ograničenja koja su pronadjena tokom analize ulaza funkcije u putanji w, opisuje klase ekvivalencije konkretnih izvršavanja sa ograničenjima posmatranim na izlazu. Sumiranje funkcija je logička formula definisana kao disjunkcija ϕ_w formula iz različitih klasa.[1] produžava simboličko izvršavanje tako što generiše sume kao formule logike prvog reda sa neintepretiranim funkcijama, dozvoljavajući nekompletne sume(sadrži samo neki podskup puta u funkciji)koje mogu biti proširene na zahtev tokom analize.
[2] istražuje novi način sumiranja, koji je baziran na sledećoj intuiciji: ako se 2 stanja programa razlikuju samo za neke vrednosti, koje se ne pojavljuju kasnije, izvršavanje generisano sa ta 2 stanja proizvešće iste

posledice (eng. *side effects*). I tako posledice nekog fragmenta koda mogu biti uhvaćene i korišćene kasnije.

4.2 Sumiranje petlji

Sumiranje petlji koristi preduslove i postuslove koji su dinamički izračunati tokom simboličkog izvršavanja rezonovanjem izmedju uslova petlje i simboličkih promenljivih. Čuvanje suma petlji ne samo da dozvoljava simboličkom motoru(eng. *engine*) da izbegne redutantna izvršavanja iste petlje u istom stanju programa, već takođe omogućava generalizovanje sumiranja tako da ono pokrije razlilčita izvršavanja iste petlje pri različitim uslovima.
Rani radovi na ovom polju su generisali sume za pelje koje ažuriraju simboličke promenljive tokom iteracija dodavajući im fiksiranu vrednost. Takođe nisu podržavali ugnježdene petlje ili viseputne pelje (eng. *multipath-loops*), to jest petlje sa više grana unutar tela. Proteus [3] je okvir(eng. *framework*) predložen za sumiranja višeputnih petlji. On klasifikuje petlje po šablonima promene vrednosti promenljivih u uslovu petlje(npr. da li je vrednost promenlijve ažurirana) i po putanjama unutar petlje. Klasifikacija pojačava produđenu formu grafa kontrole protoka(eng. *control flow graph*), koji se kasnije koristi za konstruklciju automata koji modeluje putanje. Automat se obilazi DFS algoritmom i različita sumiranja se konstruišu na osnovnu svih izvodljivih izvršavanja u petlji. Klasifikacija određuje da li petlja može biti sačuvana precizno ili se mora aproksimirati. Precizno sumiranje višeputnih petlji sa nepravilnim šablonima i sumiranje ugnježdenih petlji su i dalje nerešeni problemi.

5 Zaključak

Tehnike simboličkog izvršavanja su izuzetno napredovale u poslednjoj deceniji, i sada imaju veliku primenu u različitim oblastima kao što su testirane i verifikacija softvera, bezbednost i analiza koda. Ovaj trend porasta uticaja simboličkog izvršavanja nije samo povećao kvalitet postojećih rešenja, već je uneo i neke nove ideje koje su vodile ka ogromnim praktičkim probojima. Jedan od najvećih problema ranije je bila kombinatorna eksplozija što je umnogome uticalo na perfomanse i samim tim izbegavalo se simboličko izvršavanje. Tehnike opisane u radu, su umnogome rešile ovaj problm, i one su jedan od glavnih razloga uspona simboličkog izvršavanja, ali one još nisu savršene i imaju svoje nedostatke, pa su istraživanja i unapređenja na ovu temu svakodnevna.

Literatura

[1] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–381. Springer, 2008.

[2] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. Rwset: Attacking path explosion in constraint-based test generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366. Springer, 2008.

[3] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. Proteus: Computing disjunctive loop summary via path dependency analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 61–72. ACM, 2016.