Strategije odabira puteva

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Nemanja Mićović, 1085/2016
nmicovic@outlook.com

10. decembar 2018

Sažetak

Verifikacija softvera igra značajnu ulogu u realizaciji projekata u kojima je od velikog značaja da razvijeni softver bude stabilan i u velikoj meri formalno verifikovan. Simboličko izračunavanje se izdvaja kao jedan od delova oblasti koji postiže značajne rezultate pri verifikaciji softvera i daje teorijsku osnovu za funkcionisanje alata koji se intenzivno koriste u praktičnim primenama. Kako složenost pretrage eksponencijalno raste usled pojavljivajna grananja i petlji, odabir puteva pri simboličkom izračunavanju može drastično uticati na brzinu i performanse prilikom pretrage. Postoji nekoliko čestih strategija odabira puteva kao što su pretraga u dubinu i širinu, odabir slučajnog puta, generacijska pretraga i slično, no pokazuje se da (za sada) ne postoji optimalna strategija. Odabir strategije predstavlja jedan od važnijih otvorenih problema u oblasti simboličkog izračunavanja.

Sadržaj

1 Uvod

Softverske greške mogu za sobom doneti drastične posledice i velike gubitke. Raketa *Ariane 5* 1996. godine usled pokušaja konverzije 64-bitnog broja u 16-bitni eksplodirala je 40 sekundi nakon lansiranja. Raketa je razvijana godinama i utrošeno je preko 7 milijardi dolara, dok se procenjuje da je vrednost rakete i opreme oko 500 miliona dolara. Ovo je samo jedan od mnogih primera u kojima temeljna i iscrpna verifikacija softvera igra značajnu ulogu i može dati veliki doprinos za uspešnost projekta. Simboličko izračunavanje kao deo verifikacije softvera predstavlja jedan od glavnih alata kojima se testira i verifikuje softver. Strategije odabira puteva predstavljaju važan deo simboličkog izračunavanja jer mogu drastično da utiču na kvalitet, brzinu i performanse prilikom ispitivanja ispravnosti programa. U delu 2 dat je kratak pregled relevantnih delova oblasti simboličkog izračunavnaja, navedeni su izazovi i problemi koji se javljaju kao i neka od ideja i rešenja za njih. U delu 3 prikazane su česte strategije odabira puteva koje su primenu našle u nekim od popularnijih alata kao što su Exe, Klee, Mayhem, S^2E i SAGE.

2 Simboličko izračunavanje

Simboličko izraćunavanje je podoblast verifikacije softvera u kojoj je cilj simbolički izvršiti program u cilju ispitivanja njegovog ponašanja. Simboličkim izvršavanjem pokušava se pokriti veći broj mogućih putanja i stanja kroz koje može proći program čime se može pokazati da program neće doći u neko stanje greške ili obrnuto, mogu se pronaći konkretne vrednosti parametara koje izazivaju da program dođe u stanje greške. Tehnika je nastala 70ih godina prošlog veka i detaljnije je objašnjena u radu [11] i sličnim radovima iz istog perioda kao što su [2, 7, 14].

2.1 Uvodni primer

U nastavku je dat primer funkcije `foobar` koja nameće uslov $x - y \neq 0$ u liniji 9. Funkcija će biti dovedena u stanje greške ukoliko važi negacija uslova koji se nameće u liniji 9, odnosno $x - y = 0$.

```
1.  void foobar(int a, int b)
2.  {
3.      int x = 1, y = 0;
4.      if (a != 0) {
5.          y = 3 + x;
6.          if (b == 0)
7.          x = 2 * (a + b);
8.      }
9.      assert(x - y != 0);
10. }
```

2.2 Simboličko stablo izvršavanja

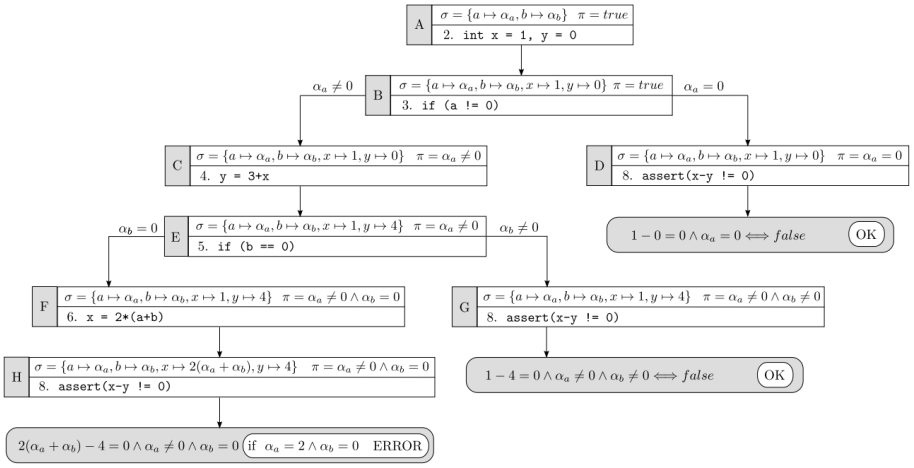
U nastavku teksta biće korišćena sledeća notacija:

- *stmt* - sledeća instrukcija koju je neophodno izvršiti,
- σ - simboličko skladište,
- π - ograničenja na tekućem putu.

U zavisnosti od vrednosti *stmt* izvršavanje napreduje po sledećim pravilima:

- $x = e$: ažurira se simboličko skladište σ tako što se vezuje x sa novom simboličkom vrednošću e_s koja se dobija simboličkim evaluiranjem izraza e
- `if` e `then` s_1 `else` s_2: prave se nove dve grane sa novim stanjima i prave se nova ograničenja puta $\pi \wedge e_s$ i $\pi \wedge \neg e_s$
- `assert(`e`)`: vrši se provera vrednosti e:
  - Ako je $\neg e \wedge \pi$ nezadovoljivo, uslov je uvek tačan.
  - Ako je $\neg e \wedge \pi$ zadovoljivo, traže se vrednosti ulaza.

Na slici 1 prikazano je simboličko izvršavanje funkcije `foobar`.



Slika 1: Simboličko izvršavanje funkcije `foobar`.

Neke od primena simboličkog izračunavanja su:

- otkrivanje grešaka u programima,
- generisanje test primera,
- otkrivanje nedostižnih putanja.

Prve praktične upotrebe simboličkog izračunavanja dešavaju se oko 2005. godine od strane alata DART [9] i EXE [4] koje ilustruju da se koncepti mogu primeniti na praktične probleme. Alat KLEE [3] 2006. godine razvijen od strane univerziteta Stanford (eng. *Stanford University*) je primenjen na 90 programa u paketu `coreutils` gde je pronašao 9 ozbiljnih bagova koji su popravljeni nakon toga. Osim toga, KLEE je uspeo da generiše test primere koji su bili u stanju da pokriju 80-100% izvršivih instrukcija. Od 2008. godine u kompaniji `Microsoft` skoro 30% bagova na razvoju operativnog sistema `Windows 7` je otkriveno koristeći alate zasnovane na simboličkom izvršavanju.

2.3 Izazovi u simboličkom izračunavanju

Postoji više problema koji se javljaju u oblasti simboličkog izvršavanja kao što su problemi sa:

- memorijom,
- okruženjem,
- eksplozijom prostora stanja,
- rešavanjem ograničenja,
- nedostupnošću izvornog koda.

Problemi sa memorijom su problemi u kojima je potrebno odrediti kako okruženje za simboličko izvršavanje razrešava rad sa pokazivačima, nizovima i kompleksnim objektima. *Problemi sa okruženjem* se javljaju u interagovanju sistema za simboličko izvršavanje sa radom sistemskog steka - prenosom parametara, pozivom funkcija i slično. Takođe, poziv spoljašnjih funkcija i biblioteka za koje kod nije dostupan kao i pojavljivanje bočnih efekata (eng. *side effects*) ume da zakomplikuje situaciju. *Eksplozija prostora stanja* je situacia u kojoj se dešava da usled previše grananja u okviru programa okruženje za simboličko izvršavanje dolazi u situaciju da ispituje veliki broj stanja. Grananje (uključujuči i petlje) unutar programa eksponencijalno uvećava broj stanja i postaje vrlo nepraktično ispitati i pratiti sva moguća stanja programa. *Rešavanje ograničenja* ima važnu ulogu u simboličkom izračunavanju jer se njime pokazuje da određeni uslovi važe ili pronalaze primeri za koje uslovi ne važe. Ipak i ovde postoji granica koja se ogleda u broju promenljivih i uslova koje rešavač može da reši, domen sa kojim radi (nelinearna aritmetika ume drastično da oteža rešavanje) i da li je poziv rešavača skupo u odnosu na željene performanse. Postoje situacije u kojima *izvorni kod nije dostupan* te se simboličko izvršavanje vrši nad binarnim kodom (odnosno asemblerskim instrukcijama). Posledice toga su povećavanje vremena izvršavanja i manjak interpretabilnosti jer je kod visokog nivoa nedostupan [1].

2.4 Konkoličko izvršavanje

Postaje evidentno da izazovi koji se javljaju u simboličkom izvršavnaju nisu jednostavno rešivi i zahtevaju dodatan trud kako bi se otklonili. Jedan od poznatih pristupa je *konkoličko izvršavanje* (eng. *concolic execution*) koje predstavlja kombinaciju **konk**retnog (eng. *concrete*) i simb**oličko**g izvršavanja (eng. *symbolic execuction*). Osnovna ideja je preplitanje ova dva vida izvršavanja kako bi se smanjio broj poziva rešavaču ograničenja, smanjio broj stanja koji se istražuje i povećale performanse. U literaturi [1] se izdvajaju dve vrste konkoličkog izvršavanja:

- dinamičko simboličko izvršavanje (eng. *dynamic symbolic execution - DSE*),
- selektivno simboličko izvršavanje (eng. *selective symbolic execution*).

Dinamičko simboličko izvršavanje je bazirano na ideji da konkretno izvršavanje navodi simboličko izvršavanje [1]. Izvršavanje se započinje konkretno sa nekim vrednostima, a rešavač se poziva kako bi se pronašle nove vrednosti i pretraga usmerila dalje. *Selektivno simboličko izvršavanje* se bazira na ideji da je potrebno istražiti samo neke delove softvera detaljno, na primer funkciju B koja se poziva unutar funkcije A. Tada se iz konkretnog izvršavanja funkcije A pri pozivu funkcije B argumenti funkcije B prebacuju u simboličke promenljive i vrši se njeno simboličko izvršavanje. Nakon toga se izvršavanje vraća nazad dalje u konkretno izvršavanje funkcije A. Suprotno ovome, može se desiiti da se funkcija A izvršava simbolički, a da funkcije B nije od interesa za detaljno ispitivanje

pa se prelazi na konkretno izvršavanje funkcije B, a potom se vraća nazad u simboličko izvršavanje ostatka funkcije A [1].

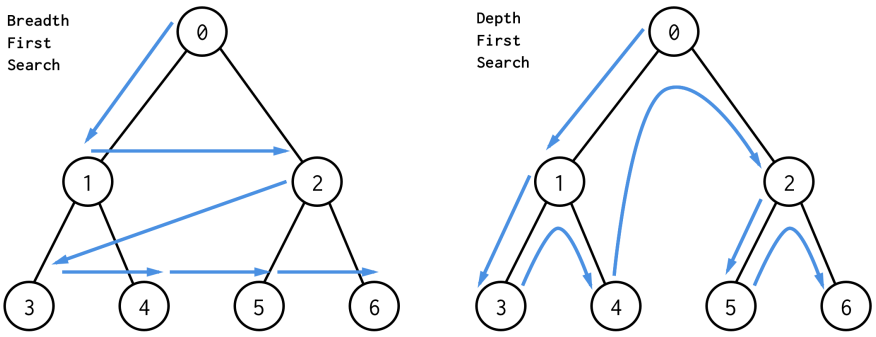
3 Strategije odabira puteva

Kako bi se pretraga ubrzala neophodno je pažljivo vršiti odabir puteva ka kojima se pretraga usmerava i minimizovala potreba za pozivanjem SMT rešavača. Odabir strategije i heurustike (ili heuristika ukoliko se koristi više) predstavlja jedan od važnijih otvorenih problema u oblasti simboličkog izračunavanja. Neke od poznatih strategija i heuristika odabira puteva [1] su:

- Pretraga u dubinu
- Pretraga u širinu
- Odabir slučajnog puta
- Izvršavanje vođeno pokrivenošću koda
- Simboličko izvršavanje najkraćeg rastojanja
- Generacijska pretraga
- Kombinovana strategija

3.1 Pretraga u dubinu

Pretraga u dubinu (eng. *DFS - depth first search*) predstavlja vid pretrage u kojoj se put proširuje dokle god je to moguće, a potom se pretraga vraća nazad do najdublje neistražene putanje [8]. Pretraga u dubinu se koristi u situacijama kada je zauzeće memorije od velike važnosti, ali joj rekurizvni pozivi i petlje stvaraju probleme [1]. Na slici 2 desno prikazan je primer pretrage u dubinu.



Slika 2: Primer pretrage u širinu (levo) i dubinu (desno)

3.2 Pretraga u širinu

Pretraga u širinu (eng. *BFS - breadth first search*) predstavlja algoritam pretrage u kojoj se putevi koji se istražuju tokom pretrage proširuju paralelno [8]. Odnosno, putanje se redno obrađuju ukoliko pretraga nije paralelizovana, ali se pretraga trudi da ravnomerno istražuje sve otvorene

puteve. Iako je memorijski zahtevnija pretraga u odnosu na DFS i zahteva više vremena da se neka putanja istraži, neki alati je koriste jer omogućava da okruženje za simboličko izvršavanje detektuje putanje od interesa ranije i potencijalno ih detaljnije istraži. Na slici 2 levo prikazan je primer pretrage u širinu.

3.3 Odabir slučajnog puta

Odabir slučajnog puta (eng. *random path selection*) je heuristika koja odabir puta vrši otežano verovatnoćama koje se mogu dinamički podešavati u zavisnosti od konfiguracije. Jedna opcija je da verovatnoće budu podešene tako da odabir između dostupnih grana bude jednake verovatnoće ili da se verovatnoće otežaju nekim parametrom poput dužine puta ili arnosti [3]. Ovo je jedna od heuristika koju koristi sistem Klee pomoću koje određuje koji od procesa dobija priliku da se izvršava [3]. Osim toga, odabir slučajnog puta u stvari predstavlja probabilističku verziju pretrage u širinu.

3.4 Izvršavanje vođeno pokrivenošću koda

Izvršavanje vođeno pokrivenošću koda (eng. *coverage optimize search*) predstavlja heuristiku konstruisanu tako da se odabir grana vrši sa ciljem da se maksimizuje pokrivenost koda. Predstavlja jednu od korišćenijih heuristika i koristi se u sistemima kao što su Klee, Exe [4], Mayhem [5] i S^2E [6]. U sistemu Klee heuristika je realizovana tako što se za svako stanje vrši izračunavanje težine koja se kasnije koristi pri odabiru sledećeg stanja. Težina se izračunava tako što se evaluira koliko daleko je najbliža neoktrivena instrukcija, da li stanje relativno skoro pokrilo neki novi deo koda kao i stek pozivanja stanja. U radu [12] se diskutuje slična heuristika sa nazivom *pretraga navođenja podskupom puta* (eng. *subpath-guided search*) koja pokušava da istraži putanje koji nisu toliko intenzivno korišćene u pretrazi tako što bira podskup puta u grafu kontrole toka za koji važi da nije dovoljno često učestvovao u pretrazi.

3.5 Simboličko izvršavanje najkraćeg rastojanja

Ukoliko je unapred određena instrukcija u kodu [1], potrebno je ispitati da li postoji put koji se može realizovati i program dovesti do te instrukcije? Ovakav problem se naziva *problem dosezanja instrukcije* (eng. *line reachability problem*) i pronalazi svoju primenu u navođenju pretrage u okviru simboličkog izvršavanja [13].

Rešavanje problema dosezanja instrukcije predstavlja aktuelnu oblast istraživanja jer simboličko izvršavanje garantuje da je instrukciju moguće dosegnuti i ... DODAT MOZDA

Za razliku od familije heuristika koje maksimuzuju pokrivenost koda (deo 3.4) heuristike čiji opis sledi su bazirane na tome da navode simboličko izvršavanje ka izvršavanju željene instrukcije.

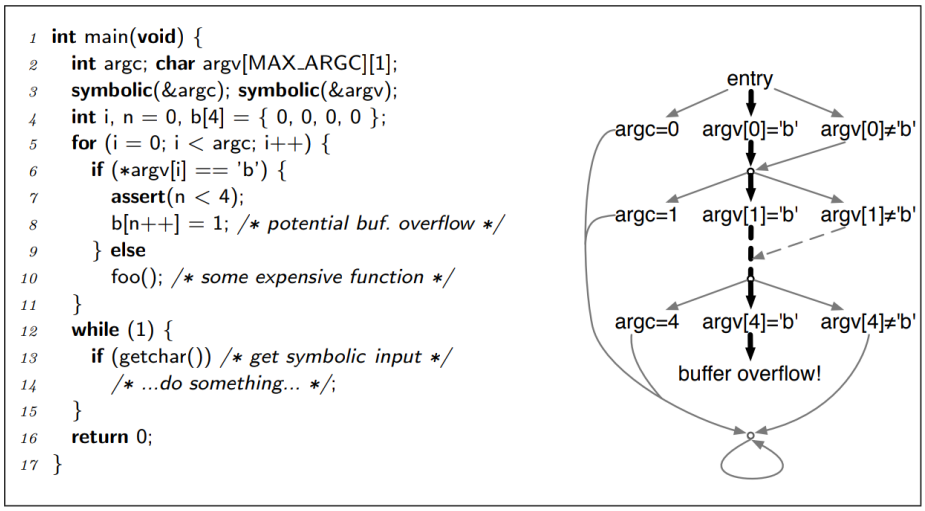
3.5.1 Simboličko izvršavanje najkraćeg rastojanja - SDSE

Simboličko izvršavanje najkraćeg rastojanja (eng. *shortest-distance symbolic execution - SDSE*) je strategija izvršavanja u simboličkom izvršavanju koja koristi meru rastojanja u grafu kontrole toka da navodi simboličko

[1] U literaturi se na nju referiše kao na liniju, razlog za to je referisanje na *liniju koda*.

izvršavanje prema određenom cilju. Originalno opisana heuristika je inspirisana heuristikom koju koristi alat KLEE za navođenje pretrage zasnovano na maksimizaciji pokrivenosti [13].

Osnovna ideja je dati prioritet grananjima u programu koja odgovaraju najkraćem putu do cilja u grafu kontrole toka. Na slici 3 prikazan je primer koda koji obrađuje argumente komandne linije, a u liniji 7 je označen potencijalno opasan slučaj u kojem dolazi do prekoračenja prihvatne memorije (eng. *buffer overflow*). Do greške može doći u situaciji kada je više od četiri prosleđena argumenta jednako `b`, tada će `n` dobiti vrednost `4` i pokušaće se pristup izvan niza `b` deklarisanog i inicijalizovanog na liniji 4. Nakon obrade argumenata, na liniji 12 program započinje svoj dalji rad.



Slika 3: Primer koji ilustruje SDSE.

Ukoliko se koristi odabir slučajnog puta, pretraga neće biti usmerena ka prekoračenju prihvatne memorije već će se probabilistički ravnomerno vršiti pretraga bez dublje intuicije o problemu. Osim toga, do instrukcije u kojoj je detektovana greška se ni ne može doći u delu grafa nakon što se krene obrađivati kod od linije 12, odnosno verovatnoća da ovakva pretraga pronađe potencijalnu grešku nije velika. Ukoliko bi se koristila heuristka koja maksimizuje pokrivenost koda, dosta vremena bi bilo utrošeno u istraživanju putanja nakon linije 12 kako bi se maksimizovala pokrivenost, iako se u tom delu neće ni pronaći greška.

Sa druge strana, pretraga SDSE kojoj se kao cilj postavlja linija 7 se pokazuje kao dobar izbor u ovakom scenariju. Na početku, dolazi se do linije 5 u kojoj se nalazi petlja. Dešava se podela u izvršavanju (eng. *fork*) i razmatraju se različiti scenariji. Kada se vrši odabir o sledećoj instrukciji koju je potrebno izvršiti, bira se prva instrukcija unutar petlje jer je *najbliža* ciljnoj instrukciji, odnosno nalazi se na rastojanju 2 od tekuće instrukcije umesto instrukcije na liniji 12 nakon koje nije ni moguće doći do cilja. Nakon toga, u liniji 6 bira se grana koja zadovoljava uslov u grananju ($*argv[i] ==' b'$) jer se time još približava ciljnoj instrukciji koja je prva sledeća na koju će se naići i tako dalje.

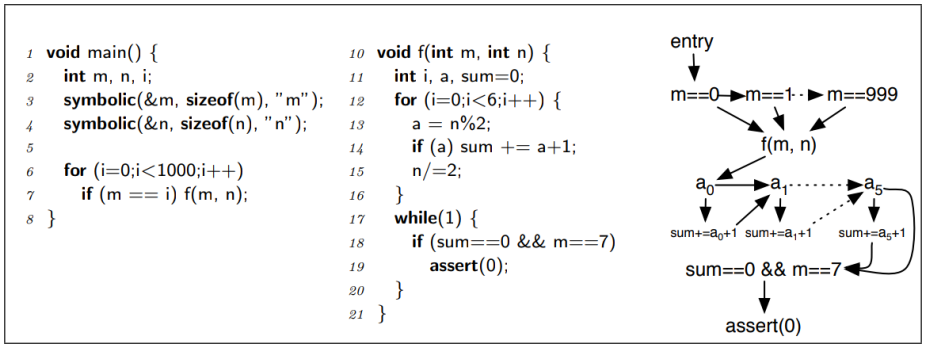
SDSE se pokazuje kao vrlo efikasan način pretrage ali postoje i situacije u kojima nije pogodan za korišćenje kao što je situacija u kojoj postoji

veliki broj različitih putanja do ciljne linije. Pokazuje se da CCBSE u takvoj situaciji može da se pokaže kao bolji izbor [13].

3.5.2 Simboličko izvršavanje za pozivima unazad - CCBSE

Simboličko izvršavanje za pozivima unazad (eng. *call-chain-backward symbolic execution - CCBSE)*) je strategija izvršavanja koja iterativno pokreće simboličko izvršavanje iz tačke prve instrukcije u okviru funkcije, a potom *skače* unazad u steku pozivanja funkcija dok ne pronađe adekvatnu putanju od početka koda.

Na slici 4 prikazan je programski kod u kojem se dolazi u stanje greške ukoliko važi $sum = 0 \wedge m = 7$ (linije 18 i 19). Ukoliko uslov nije ispunjen program će uči u beskonačnu petlju (linja 17). CCBSE radi tako što promenljive m i n zadržava kao simboličke. Potencijalno će morati da istraži svih 2^6 putanja (petlja na liniji 12) kako bi pronašao put p koji doseže instrukciju $assert(0)$ na liniji 19 kojim program dolazi u stanje greške. Potom se skače nazad u funkciju *main* i istražuju se svi putevi koji vode do puta p. Na ovaj način se izbegava ponovna pretraga u funkciji f - odnosno izbegava se ispitivanje potencijalno svih 2^6 koje nameće petlja.



Slika 4: Primer koji ilustruje CCBSE.

3.6 Generacijska pretraga

Alat SAGE (eng. *scalable automated guided execution*) koristi *generacijsku pretragu* koja maksimizuje broj ulaznih testova koji se generišu od strane svakog pokretanja simboličkog izvršavanja [10]. Ukoliko je dato neko ograničenje puta, sva ograničenja na tom putu se sistematski negiraju jedno po jedno, dodaju se u konjunkciju sa određenim prefiksom ograničenja koje je dovelo do tog uslova, a potom se dobijena formula pokušava rešiti nekim od adekvatnih rešavača. Ovim se postiže da jedno simboličko izvršavanje generiše hiljade novih testova u odnosu na pretragu u dubinu ili širinu koje bi u ovom slučaju negirale samo poslednji ili prvi uslov u uslovima putanje i generisali najviše jedan primer u okviru tekućem simboličkog pokretanja.

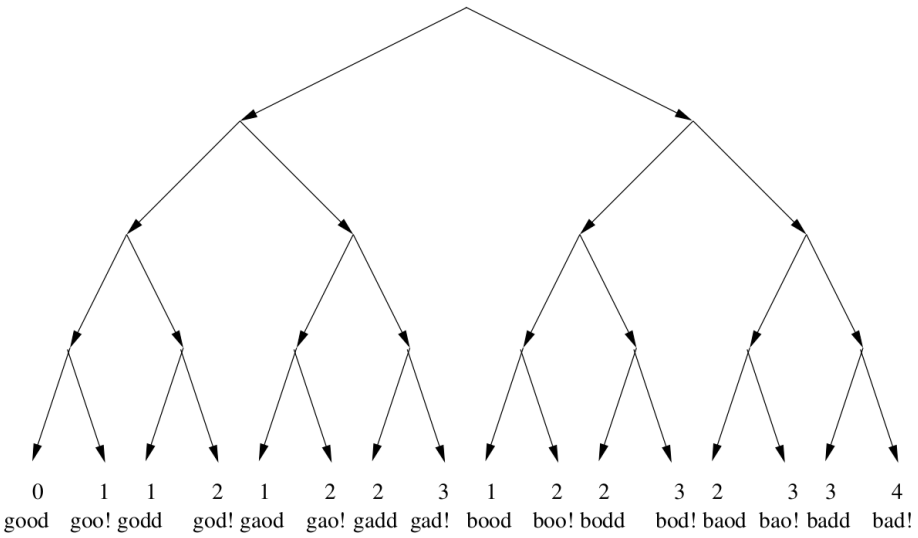
Funkcija *top* prihvata četiri bajta kao ulaz i dolazi u stanje greške ukoliko važi uslov $cnt \geq 4$. SAGE pokreće dinamičko simboličko izvršavanje sa nekim nasumičnim ulazom, na primer `good`. U prvom prolazu se izvršavaju `else` grane u funkciji (odnosno ne izvršava se praktično ništa

jer `else` grane ne postoje za odgovarajući `if`) i generiše se uslov puta:

$$i_0 \neq b \wedge i_1 \neq a \wedge i_2 \neq d \wedge i_3 \neq !$$

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[0] == 'a') cnt++;
    if (input[0] == 'd') cnt++;
    if (input[0] == '!') cnt++;
    if (cnt >= 4) error();
}
```

Dolazi do negacije svakog od ovih uslova koji biva dodat u konjunkciju sa dodatkom uslova koji su važili pre njega. Dobijena formula se potom rešava rešavačem čime se dobijaju novi test primeri. U pomenutom primeru, sva četiri uslova su rešiva i pokretanje rešavača daje nova četiri test primera. Prvo pokretanje i dobijanje ova četiri test primera se nazivaju generacija 0, naredna pokretanja za ova četiri nova test primera se nazivaju generacija 1 i slično. Na slici 5 prikazano je stablo pretrage funkcije *top* pri čemu su u listovima date vrednosti promenljive *cnt* i ulaznih vrednosti *input*.



Slika 5: Prostor pretrage sa vrednošću promenljive *cnt* u listovima.

Alat SAGE se intenzivno koristi u procesu razvoja softvera unutar kompanije *Microsoft* i daje značajan doprinos u fazi evaluacije kvaliteta softverskih proizvoda koje kompanija proizvede.

3.7 Kombinovana strategija

Kombinovana strategija predstavlja strategiju koja pokušava da korišćenjem više različitih algoritama istovremeno ili alterniranjem između njih dođe do boljih rezultata u odnosu na to da se koristi samo jedan algoritam.

4 Zaključak

Simboličko izračunavanje pronalazi važnu primenu u praksi i omogućava da se drastično doprinese kvalitetu softvera koji se razvija. Primeri upotrebe simboličkog izračunavanja se mogu pronaći kroz alate Exe, Klee, Mayhem, S^2E i SAGE. Strategije odabira puteva igraju važnu ulogu u rešavanju problema koji se javljaju usled kompleksnosti softvera koji se koristi u praktičnim primenama. Prikazane strategije imaju svoje prednosti i mane, ali ne funkcionišu u svim situacijama, a performanse umeju da variraju i u zavisnosti od programa čija se ispravnost proverava. Odabir strategije ili heurustike za odabir puteva predstavlja otvoren problem u oblasti verifikacije softver i jedan od važnijih problema u simboličkom izračunavanju.

Literatura

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[2] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.*, 10(6):234–245, April 1975.

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[4] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[5] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, May 2012.

[6] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The s2e platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, February 2012.

[7] Lori A. Clarke. A program testing system. In *Proceedings of the 1976 Annual Conference*, ACM '76, pages 488–491, New York, NY, USA, 1976. ACM.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[10] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012.

[11] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[12] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering symbolic execution to less traveled paths. *SIGPLAN Not.*, 48(10):19–32, October 2013.

[13] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 95–111, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] Leon J. Osterweil. Allegations as aids to static program testing. In *Proceedings of the 1976 Annual Conference*, ACM '76, pages 479–482, New York, NY, USA, 1976. ACM.