Konkoličko izvršavanje

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Aleksandar Anžel, 1025/2018
aleksandar.anzel@gmail.com

8. decembar 2018

Sažetak

U ovom radu je predstavljena sažeta priča o konkoličkom izvršavanju kao i njegova tesna veza sa simboličkim izvršavanjem. Primerima se pokazuju prednosti i nedostaci obe metode kao i primena istih nad realnim, svakodnevnim softverom. Rad se velikim delom oslanja na naučni rad [1] napisan na pomenutu temu.

Sadržaj

1 Uvod

Simboličko izvršavanje je popularna tehnika analize programa, uvedena sredinom sedamdesetih godina, da bi se testiralo da li se određena svojstva programa mogu ugroziti pomoću specijalnog softvera. [3, 8, 9, 10] Pod simboličkim izvršavanjem smatramo istovremeno izvršavanje svih mogućih puteva jednog programa. Ovakav način testiranja programa omogućava da identifikujemo sve moguće nebezbedne ulaze tog programa. Specijalan softver koji se koristi za testiranje programa pomoću simboličkog izvršavanja nazivamo simboličkim izvršivačima (eng. *Symbolic Execution Engines*). Simboličko izvršavanje nam garantuje **korektnost** (eng. *Soundness*) - ako izvršivač tvrdi da je nešto greška, to će i biti greška, kao i **kompletnost** (eng. *Completeness*) - sve greške će biti pronađene.

Sa druge strane, primetimo da je ovakav način testiranja programa jako zahtevan i u praksi se pokazuje da ga je jako teško skalirati nad većim programima. Neki od problema sa kojima se suočava simboličko izvršavanje pri testiranju realnih, svakodnevnih programa su:

- Memorija: kako bi simbolički izvršivač trebao da obrađuje pokazivače, nizove i druge složenije objekte?
- Okruženje: kako bi simbolički izvršivač trebao da obrađuje komunikacije preko memorijskog steka?
- Eksplozija prostora: kako bi simbolički izvršivač trebao da obrađuje eksploziju putanja (pojava petlji)?
- Razrešavanje uslovnosti: obrađivanje nelinearnih uslova unutar naredbi grananja je jako neefikasan proces.

Pored gore navedenih problema, još jedan značajan problem je fragmentisanost realnih programa tj. izdvojenost procedura i funkcija programa u zasebne datoteke. Implementacija simboličkih izvršivača koji bi mogli statički da isprate sve pozive jednog programa bi bio jedan jako zahtevan posao jer je jako teško precizno predvideti moguće sporedne efekte poziva funkcija prilikom izvršavanja glavnog programa. Zato često u praksi, nad realnim programima, žrtvujemo **korektnost** simboličkog izvršavanja zarad boljih performansi.

Fundamentalna ideja koja nam omogućava prevazilaženje ovih problema i koja omogućava praktičnu primenu simboličkog izvršavanja jeste mešanje konkretnog i simboličkog izvršavanja tj. primena takozvanog **konkoličkog** izvršavanja, pri čemu je ovaj termin nastao spajanjem reči *konkretan* i *simbolički*. U narednim sekcijama ćemo predstaviti par načina primene konkoličkog izvršavanja.

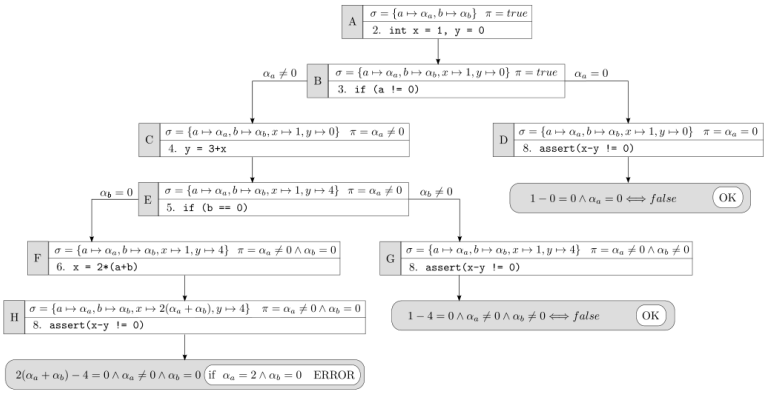
```
1.  void foobar(int a, int b) {
2.      int x = 1, y = 0;
3.      if (a != 0) {
4.          y = 3+x;
5.          if (b == 0)
6.              x = 2*(a+b);
7.      }
8.      assert(x-y != 0);
9.  }
```

Slika 1: Zadatak: za koje vrednosti a i b *assert* ne prolazi?

2 Izvršavanja kroz primere

2.1 Simboličko izvršavanje

Posmatrajmo C kôd na slici 1 i pokušajmo da vidimo za koje vrednosti ulaza funkcija *assert* (u liniji 8) pada. Ako označimo sa σ simboličku memoriju (čuva konkretne ili simboličke α_i vrednosti promenljivih) i sa π uslove putanja, tada je simboličko izvršavanje programa *foobar* prikazano pomoću drveta:



Slika 2: Drvo dobijeno simboličkim izvršavanjem funkcije *foobar* na slici 1. Svako stanje izvršavanja, označeno velikim slovom, sadrži trenutnu naredbu, simboličku memoriju σ i uslove putanji π. Listovi predstavljaju evaluaciju uslova u *assert* liniji.

Sa slike 2, analizirajući listove tj. stanja *{D, G, H}* zaključujemo da samo stanje H može proizvesti tačnost uslova $x - y = 0$. To znači da svaki ulaz programa 1 koji zadovoljava:

$$2(\alpha_a + \alpha_b) - 4 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_b = 0 \tag{1}$$

čini da *assert* pada. Konkretni parametri se mogu dobiti primenom SMT (eng. *Satisfiability Modulo Theories*) rešavača [2] nad uslovima stanja, koji bi proizveli rešenje $a = 2$ i $b = 0$.

Jasno uočavamo i jednu od glavnih mana ove metode - skalabilnost. Teorijski gledano, simboličko izvršavanje će nam uvek obezbediti **korektnost** i **kompletnost** ali je često sam proces izvršavanja neisplativ nad realnim, svakodnevnim programima. Prethodni program 1, iako je relativno kratak, simboličkim izvršavanjem proizvodi relativno veliko drvo izvršavanja 2.

2.2 Konkoličko izvršavanje

2.2.1 Dinamičko simboličko izvršavanje

Jedan od popularnih pristupa konkoličkom izvršavanju je **dinamičko simboličko izvršavanje** (eng. *DSE - Dynamic Symbolic Execution*) tj. *dinamičko generisanje testova*. [5] Ova tehnika predstavlja efikasan način prevazilaženja prethodno opisanih problema žrtvujući **korektnost** zarad boljih performansi.

Tehnika dinamičkog simboličkog izvršavanja se zasniva na sledećem: nakon biranja proizvoljnog, konkretnog ulaza za program on se izvršava istovremeno i konkretno i simbolički. Kad god se konkretnim izvršavanjem uđe u neku granu izvršavanja, tu granu preuzima simbolički izvršivač i među uslovima putanja (π) se dodaje uslov tekuće grane izvršavanja. Ukratko, simboličko izvršavanje upravlja određeno konkretno izvršavanje. Samim tim, simbolički izvršivač ne mora da poziva SMT rešavač koji će pokazati da li je uslov grane izvršavanja (ne)zadovoljiv jer to možemo dobiti direktnom zamenom konkretnih vrednosti u sam uslov grane izvršavanja i direktnom proverom rezultata. Da bi se istražile i druge putanje programa, uslovi putanja, dobijeni iz jednog ili više uslova grana izvršavanja, se mogu negirati pa se tek onda može pozvati SMT rešavač koji bi našao rešenja novih uslova putanje i samim tim izgenerisao nove konkretne ulaze. Ovaj postupak se zatim može primenjivati koliko god je potrebno da bi se dobila tražena pokrivenost programa.

Posmatrajmo C kôd na slici 1 i pored simbola σ, π i α_i, opisanih u 2.1, uvedimo i pomoćni simbol σ_c koji će nam označavati memoriju konkretnih vrednosti. Pretpostavimo da smo za ulazne vrednosti uzeli $a = 1$ i $b = 1$. Dakle, konkretno izvršavanje se kreće putanjom $A \hookrightarrow B \hookrightarrow C \hookrightarrow E \hookrightarrow G$ kroz drvo pokazano na slici 2. Pored simboličkih memorija σ, pokazanih na slici 2, memorije konkretnih vrednosti će kroz posećena stanja izgledati na sledeći način:

- $\sigma_c = a \mapsto 1, b \mapsto 1$ u stanju A,
- $\sigma_c = a \mapsto 1, b \mapsto 1, x \mapsto 1, y \mapsto 0$ u stanjima B i C,
- $\sigma_c = a \mapsto 1, b \mapsto 1, x \mapsto 1, y \mapsto 4$ u stanjima E i G.

Nakon provere uslova unutar *assert* funkcije, u liniji 8, možemo da generišemo novi niz putanja negirajući uslov poslednje putanje, tj. $\alpha_b \neq 0$. SMT rešavač tada generiše nove vrednosti koje zadovoljavaju uslove grane izvršavanja $\alpha_a \neq 0 \wedge \alpha_b = 0$ (na primer $a = 1$ i $b = 0$) i izvršavanje se nastavlja na isti način putanjom $A \hookrightarrow B \hookrightarrow C \hookrightarrow E \hookrightarrow F$.

Pored toga što dinamičko simboličko izvršavanje koristi konkretne ulaze da upravlja simboličkim izvršavanjem, ova tehnika mora i da bira novu granu koju će negirati svaki put kada je potrebno istražiti novu putanju. Kako je skup neposećenih putanja često jako veliki naspram skupa posećenih, odabir efikasnog algoritma izbora može igrati važnu ulogu. Na primer, DART [6] bira narednu granu za negiranje koristeći **pretragu u dubinu** (eng. *DFS - Depth-first search*) [11], dok sistem SAGE [7] koristi *generišuću pretragu*.

```
void foo(int x, int y) {
   int a = bar(x);
   if (y < 0) ERROR;
}
```
(a)

```
void qux(int x) {
   int a = bar(x);
   if (a > 0) ERROR;
}
```
(b)

```
void baz(int x) {
   abs(&x);
   if (x < 0) ERROR;
}
```
(c)

Slika 3: Konkoličko izvršavanje: (a) testiranje funkcije *foo* iako izvršivač ne može simbolički ispratiti funkciju *bar*, (b) primer propuštene putanje, (c) primer divergencije putanje

Naredni primeri pokazuju nedostatke tehnike dinamičkog simboličkog izvršavanja opisane u 1. Posmatrajmo funkciju *foo* na slici 3(a) i pretpostavimo da konkolički izvršivač simbolički ne prati funkciju *bar*. Ako su $x = 1$ i $y = 2$ nasumično odabrani ulazni parametri, konkolički izvršivač izvršava *bar* (koja vraća $a = 0$) i preskače granu izvršavanja koja bi aktivirala naredbu greške. Istovremeno će se simbolički izvršavati putanja funkcije *foo* sa uslovom putanje $\alpha_y \geq 0$. Primetimo da uslovi grananja u funkciji *bar* nisu poznati konkoličkom izvršivaču. Da bi zatim istražio alternativne putanje, izvršivač negira uslov putanje unutar uslova grananja funkcije *foo*, generišući ulaze $x = 1$ i $y = -4$, što samim tim vodi konkolički izvršivač drugom putanjom. Na ovaj način izvršivač može da obiđe obe putanje funkcije *foo* bez simboličkog praćenja funkcije *bar*.

Posmatrajmo sada funkciju *qux*, na slici 3(b), koja proverava povratnu vrednost funkcije *bar* u uslovu grananja. Kako konkolički izvršivač nema nikakav uvid u povezanost promenljivih a i x, jer *bar* nije simbolički praćena funkcija, ne postoji garancija da će se ikad generisati alternativni put pomoću konkretnih ulaza i samim tim proizvesti greška.

Sličan problem se javlja i u funkciji *baz* na slici 3(c). U ovom primeru, funkcija *baz* poziva funkciju *abs* koja svoj argument postavlja na njegovu apsolutnu vrednost. Neka je $x = 1$ inicijalna konkretna vrednost promenljive x. Konkretno izvršavanje neće aktivirati naredbu greške, te će konkolički izvršivač doći do uslova putanje $\alpha_x \geq 0$ i pokušati da izgeneriše novu ulaznu vrednost negirajući ovaj uslov. Ali i ovako dobijena nova konkretna vrednost, npr. $x = -1$, ne aktivira naredbu greške usled (simbolički nepraćenih) bočnih efekata funkcije *abs*.

Ovi nedostaci su jako značajni i često se javljaju u svakodnevnim programima usled poziva funkcija, izuzetaka, pokazivača i sl. te ih konkolički izvršivači moraju jako pažljivo obrađivati.

2.2.2 Selektivno simboličko izvršavanje

Selektivno simboličko izvršavanje (eng. S^2E - *Selective Symbolic Execution* [4]) se zasniva na drugačijem pristupu mešanja simboličkog i konkretnog izvršavanja, baziranog na osnovu opažanja da često želimo da istražimo samo određene komponente programa, pri čemu nas ne zanima njegov ostatak.

Posmatrajmo sledeću situaciju. Pretpostavimo da imamo funkciju A koja poziva funkciju B pri čemu se način izvršavanja menja pri samom pozivu. Imamo dve mogućnosti:

1. *Iz konkretnog u simboličko i nazad:* argumenti funkcije B su simbolički i B se simbolički izvršava. Zatim se B i konkretno izvrši i (konkretna) povratna vrednost se vrati funkciji A koja zatim nastavlja konkretno izvršavanje.
2. *Iz simboličkog u konkretno i nazad:* argumenti funkcije B su konkretni i ona se konkretno izvršava a po njenom završetku se nastavlja simboličko izvršavanje funkcije A.

Ovakav način izvršavanja može znatno da utiče kako na **korektnost** tako i na **kompletnost**:

- **Korektnost**: da bi simboličko izvršavanje preskočilo sve putanje koje ne bi mogle da se izvrše usled primenjenih konkretizacija, ova metoda pamti uslove putanja koje čuvaju informacije o tome kako se argumenti konkretizuju, koji su bočni efekti izvršavanja funkcije B i koje su njene povratne vrednosti.

- **Kompletnost**: konkretizacija može da dovede do promašenih grana izvršavanja nakon što A nastavi sa izvršavanjem.

Da bi se ovo prevazišlo svi zapamćeni uslovi se označavaju kao *meki* tj. kad god se pri povratku u A, usled mekog uslova, ne može otići alternativnom putanjom iste grane, onda se izvršavanje vraća unazad i biraju se drugačiji argumenti funkcije B. Da bi omogućila ponovnu konkretizaciju argumenata za B, ova metoda pamti sve uslove grananja tokom konkretnog izvršavanja funkcije B i na osnovu njih bira nove konkretne vrednosti tako da one omoguće drugačiju putanju konkretnog izvršavanja u funkciji B.

3 Zaključak

Verifikacija softvera predstavlja jako bitan segment razvoja softvera. Simboličko izvršavanje, teorijski, je jedna od najboljih metoda automatske statičke analize softvera ali se u praksi ona susreće sa različitim teškoćama. Veliki deo tih problema se može prevazići konkoličkim izvršavanjem softvera koje predstavlja prirodnu nadogradnju simboličkog izvršavanja adaptiranog za rad na realnom softveru. Ove tehnike su danas neizostavan deo automatske analize softvera i njihova primena u poslednjoj deceniji nije samo poboljšala postojeća rešenja, već je dovela i do novih, pa čak u nekim slučajevima i do ogromnih napredaka u ovoj oblasti.

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):50:1–50:39, May 2018.

[2] Clark Barrett, Daniel Kroening, and Thomas Melham. *Problem solving for the 21st century: Efficient solver for satisfiability modulo theories*. Knowledge Transfer Report, Technical Report 3. London Mathematical Society and Smith Institute for Industrial Mathematics and System Engineering, 6 2014.

[3] 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.

[4] 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.

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

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

[7] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[8] W. E. Howden. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Transactions on Software Engineering*, SE-3(4):266–278, July 1977.

[9] James C. King. A new approach to program testing. *SIGPLAN Not.*, 10(6):228–233, April 1975.

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

[11] Википедија. Pretraga u dubinu — Википедија,, 2018. [Online; accessed 11-новембар-2018].