

Calysto: skalabilni i precizni prošireni statički proveravač

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Jelena Ivković
jelena.ivkovic5@gmail.com

7. maj 2018.

Sažetak

Otklanjanje grešaka (verifikacija/testiranje/debugovanje), deo je životnog ciklusa razvoja softvera koji troši dosta vremena. Shodno tome, za podršku ovog zadatka, razvijen je veliki broj tehnika i alata. Već dugo se teži ka automatskom otkrivanju grešaka, a svaka od tehnika kojima se to postiže, mora da pravi kompromis između različitih nivoa automatizacije, pokrivenosti koda, preciznosti analize i skalabilnosti.

Automatizacija: Koliko je ručnog napora potrebno?

Pokrivenost: Koliko temeljno su analizom pokrivene sve moguće putanje izvršavanja i vrednosti podataka?

Preciznost: Koliko precizno analiza odgovara stvarnom softveru koji je izvršen?

Skalabilnost: Koliko velika baza kodova može biti analizirana tehnikom ili alatom?

Calysto je alat za Jupyter i Python koji vrši potpuno automatsko, statičko proveravanje uz kombinaciju preciznosti i pokrivenosti kakva nikad pre nije postignuta (Calysto je, takođe, veliki satelit planete Jupiter). Ovaj rad predstavlja arhitekturu, dizajn i optimizaciju koji stoje iza performansi Calysto alata.[\[1\]](#)

1 Uvod

Prošireno statičko proveravanje (eng. *extended static checking*) je termin koji je skovao David L. Detlefs za kombinovano korišćenje modela statičkog proveravanja (lako korišćenje, usmerenost ka specifičnim čestim greškama, nema garancije o preciznosti) i formalne verifikacije (generisanje uslova verifikacije i njihovo proveravanje dokazivačem teorema). Garantuje se pokrivenost koja odgovara formalnoj verifikaciji sa automatizacijom i skalabilnošću koja je bliska onima kod jednostavnih statičkih proveravača.

Calysto je inspirisan i pod uticajem je ESC/Java, CBMC i posebno Saturn proveravača. Calysto prihvata ESC filozofiju kombinovanja lakog korišćenja i statičkog proveravanja sa moćnim analizama formalne verifikacije, ali za razliku od ESC/Java, potpuno je automatizovan, izvodeći inter-proceduralne analize. Takođe, za razliku od ESC/Java, Calysto bitovski tačno izvodi operacije nad podacima. Potpuna automatizacija, inter-proceduralna osetljivost na putanje i bitovska tačnost podsećaju na proveravač modela CBMC. Međutim, CBMC uglavnom može da radi sa kodovima od po nekoliko hiljada linija, dok je Calysto prilagođen realnim aplikacijama sa stotinama hiljada linija koda. Najbliži je Calysto alatu radi statički proveravač Saturn. Calysto kombinuje vrline CBMC-a i Saturn-a, postižući bolju skalabilnost od svega sa sličnom preciznošću i pokrivenošću i bolju preciznost i pokrivenost od svega sa sličnom skalabilnošću.

Ovaj rad opisuje kako Calysto dostiže ovu jedinstvenu kombinaciju pokrivenosti, automatizacije, preciznosti i skalabilnosti.

2 Sistem arhitekture

Calysto sistem se sastoji od tri faze podržane automatskim dokazivačem teorema Spear.

Prva faza je površna analiza pseudonima (eng. *alias*) pokazivača na funkcije, u kojoj se konstruiše graf poziva, uključujući indirektne pozive kroz pokazivače na funkcije. Ovom analizom, osetljivom na tok, ali ne i na kontekst, prateći samo pokazivače na funkcije, žrtvuje se nešto preciznosti. Ova faza je dovoljno precizna u praksi iako zahteva zanemarljivo male resurse.

Sledeća faza je simboličko izvršavanje, u kojoj se izvršava program koristeći simboličke umesto konkretnih vrednosti. Calysto simbolički izvršava funkcije u analiziranom programu, izračunavajući simboličku definiciju za svaku izmenjenu promenljivu i memorijsku lokaciju. Ove simboličke definicije se koriste za stvaranje uslova verifikacije (eng. *verification conditions*) – logičke formule koje su tačne akko neko svojstvo tačnosti važi u programu. Simboličko izvršavanje dozvoljava generisanje uslova verifikacije za bilo koje tvrđenje na bilo kom mestu u programu. Generisani uslovi verifikacije su u suštini formalni uslovi verifikacije: sve moguće putanje programa i vrednosti podataka su razmotrene.

Poslednja faza se sastoji od proveravanja i filtriranja uslova verifikacije. U principu, oni bi mogli biti direktno poslani dokazivaču teorema na proveru. Ovaj pristup se i koristi kod većine drugih alata. Međutim, otkriveno je da i efikasnost i korišćenje mogu biti poboljšani kontrolisanjem i filtriranjem onoga šta se uslovima verifikacije proverava. Da bi se poboljšalo korišćenje, ako dokazivač teorema uspe da pronađe lažno upozorenje (eng. *false positive*), ova faza prijavljuje grešku i odbacuje sve

uslove verifikacije koji odgovaraju istom svojstvu u okviru iste funkcije. Na primer, ako je pokazivač koji može biti *NULL* dereferenciran na više različitih mesta u okviru iste funkcije, a ta funkcija može biti pozvana u više različitih konteksta, Calysto će emitovati samo jednu prijavu greške po kontekstu. Ovim se izbegava pretrpavanje programera prijavama koje odgovaraju istom problemu. Za svaki lažni uslov verifikacije, Calysto izbaci detaljni grafički trag; ako uslov zavisi od bilo koje globalne promenljive, trag je dat sve do korena grafa poziva koji predstavlja *main* funkciju.

Stvarnu proveru validnosti uslova verifikacije obavlja Spear, koji je stabilan i kompletan, potpuno automatizovan dokazivač teorema koji podržava Bulovu logiku, bitovske operacije nad vektorima i bitovski tačnu aritmetiku. Za razliku od drugih statičkih kontrolera koji koriste SAT rešavače ili dokazivače teorema, Spear je posebno dizajniran za uslove verifikacije koje Calysto generiše, optimizujući performanse.

3 Izbor dizajna

Osnovna filozofija dizajna je bila da se počne sa principijenom, potpuno formalnom, preciznom analizom, da se napravi što manje nestabilnih aproksimacija i da se onda fokus prebaci na popravljjanje skalabilnosti. Ovaj pristup pomaže odvajanje brige o tačnosti analize od njene efikasne, praktične implementacije. Donete su tri ključne odluke da bi se Calysto učinio značajno preciznijim od tipičnih statičkih kontrolera:

Prva odluka je bila bitovska tačnost, u smislu da se mašinska aritmetika sprovodi precizno, uključujući sve granične uslove (potkoračenja i prekoračenja) i sve standardne operatore, uključujući množenje, deljenje, ostatak i šift. Ova preciznost rezultuje visokom cenom izračunavanja, ali ona je opravdana iz sledećih razloga: sami granični uslovi su česti izvori grešaka; granični celi brojevi su preduslov za odlučujuća svojstva sa nelinearnim operatorima, a nelinearni operatori se prilično često pojavljuju u realnom kodu.

Inter-proceduralna osetljivost na putanje je bila druga važna odluka u vezi dizajna. Kako broj mogućih putanja obično eksponencijalno raste u odnosu na veličinu koda, ova odluka je takođe skupa u smislu izračunavanja, ali, takođe, opravdana. Na primer, običan kodni idiom koji zahteva inter-proceduralnu osetljivost na putanje je rad sa pogrešnim uslovima. Aplikacije često koriste duge lance funkcijskih poziva za baratanje pogrešnim i posebnim uslovima, npr. proveravanje preduslova ili postuslova, otkrivanje grešaka, štampanje i čuvanje poruka, izlaz sa odgovarajućim kodom greške. Statički kontroleri koji nisu inter-proceduralno osetljivi na putanje mogu da podbace u preciznom izračunavanju uslova pod kojim postoji niz, što dovodi do lažnih upozorenja.

Posledica inter-proceduralne osetljivosti na putanje je treća ključna odluka: Calysto je potpuno, precizno osetljiv na kontekste. Analize osetljive na kontekste razlikuju efekte stanja programa na različitim mestima u kodu sa kojih se funkcija poziva. Za razliku od njih, analize koje nisu osetljive na kontekste mogu biti mnogo efikasnije, zato što funkcija mora biti analizirana samo jednom, bez obzira na to sa koliko različitih mesta se poziva. Međutim, ove analize moraju da spoje stanja sa svih mogućih mesta poziva, što dovodi do gubljenja informacija i lažnih upozorenja. Calysto zadržava definicije inter-proceduralnog konteksta kontrole toka, promenljivih i apstraktnih memorijskih lokacija na koje pokazivači mogu da pokazuju. Postizanje ovakve preciznosti je skupo i vremenski i pro-

storno.

Ove odluke su umnogome povećale kompleksnost izračunavanja, ali su i doprinele malom udelu lažnih upozorenja. Napravljene su i neke odluke vezane za dizajn koje su bile nestabilne (narušavajući pokrivenost, odatle i moguće propuštene greške) i neprecizne (moguće rezultiranje lažnim upozorenjima):

Calysto ne podržava operacije sa brojevima u decimalnom zapisu. Rad sa takvim brojevima je nestabilan pri pretvaranju promenljivih i konstanti iz decimalnog zapisa u celobrojni. Teorijski, direktno se, može dodati bitovski tačan model za decimalni zapis dokazivaču teorema. Praktično efikasno rešenje, međutim, verovatno bi zahtevalo više istraživanja.

Petlje proizvode neodlučivi rezultat klasičnog problema zaustavljanja (eng. *halting problem*). Dakle, Calysto nestabilno aproksimira petlje: od-vija ih jednom završavajući pretpostavkom da je test petlja podbacila, slično kao ESC/Java. Ovo je glavni izvor propuštenih grešaka jer moguće putanje programa nisu analizirane.[1]

```
1      int cnt = 0;
2      bool c2 = false;
3      while ( c1 /* some condition */ ) {
4          if ( c2 ) {
5              cnt++;
6          }
7          c2 = true;
8      }
9      if ( cnt == 0 ) { exit(1); }
10     ...
```

Slika 1: Primer dela realnog koda

Na primer, u kodu sa slike 1 promenljiva *c2* je *false* u prvoj iteraciji, tako da brojač *cnt* može biti povećan tek u drugoj iteraciji. Ako je petlja jednom odvijena, linija 10 postaje nedostižna. Kako Calysto ne proverava nedostižan kod, može doći do propuštanja grešaka. Zahvaljujući ovakvom radu sa petljama u preliminarnim eksperimentima, viđeno je samo nekoliko lažnih upozorenja proveravajući tvrđenja koja su zadali programeri, i nijedno proveravajući automatski generisane uslove verifikacije.[1]

Rekurzija proizvodi isti neodlučivi problem kao i petlje pa se ona obrađuje na sličan način. Poput Saturn-a, Calysto jednostavno razbija cikluse u grafu poziva ignorišući rekurzivne pozive. Ovo dovodi do malog broja lažnih upozorenja u praksi.

Poznato je da je pokazivačka aritmetika (time, i aritmetika nizova) uglavnom neodlučiva. Calysto koristi jednostavniji memorijski model, sličan „modelu logičke memorije“ u kome se pretpostavlja da $*(ptr + i)$ i $*ptr$ referišu na isti objekat, osim što simboličko izvršavanje razlikuje ove dve lokacije ako pojednostavljiivač izraza može da pojednostavi *i* na konstantu. Domen takve konstante se često koristi za pristup poljima strukture, čineći Calysto osetljivim i na polja, tako da je ova dodatna preciznost važna u praksi.

4 Poboljšanje skalabilnosti

Analize koje su bitovski precizne i osjetljive na putanje/kontekste/polja, računski su ekstremno skupe. Prva verzija alata Calysto se nije dobro prilagođavala programima koji su imali više od nekoliko hiljada linija koda. U osnovi poboljšanja analize, nalaze se tri opšta principa: čuvanje i korišćenje strukture problema; korišćenje brzih aproksimativnih analiza za filtriranje i pojednostavljivanje zadataka pre primene “teških” preciznih analiza; keširanje prethodnih rezultata radi ponovnog korišćenja.

5 Zaključak

Ovim radom je ukratko predstavljen Calysto, prošireni statički kontroler koji obezbeđuje jedinstvenu kombinaciju preciznosti i skalabilnosti. Među potpuno automatskim alatima, Calysto je skalabilniji od bilo čega sa sličnom pokrivenošću i preciznošću, a nudi bolju pokrivenost i preciznost od bilo čega sa sličnom skalabilnošću.

Eksperimentalno ocenjivanje je pokazalo da Calysto dobro radi sa stotinama hiljada linija realnih, slobodnih (eng. *open-source*) aplikacija. Takođe, identifikuje prave greške, potpuno automatski. Udeo lažnih upozorenja je bio ispod 23%

Literatura

- [1] Domagoj Babić and Alan J. Hu. Calysto: Scalable and Precise Extended Static Checking. In *ICSE'08: Proceedings of the 30th International Conference on Software Engineering*, pages 211–220, New York, NY, USA, May 2008. ACM.