Pouzdanost softvera

Seminarski rad u okviru kursa
Metodologija stručnog i naučnog rada
Matematički fakultet

Nenad Ajvaz, Stefan Kapunac, Filip Jovanović, Aleksandra Radosavljević
nenadajvaz@hotmail.com, stefankapunac@gmail.com,
jovanovic16942@gmail.com, aleksandraradosavljevic.@live.com

2. april 2019

Sažetak

ABSTRAKT ABSTRAKT

Sadržaj

1 Uvod

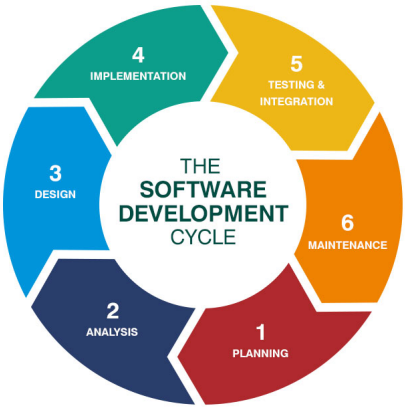
- Velika uloga softvera u danasnje vreme
- Softver je sve rasprostranjeniji, uskoro ce postati osnovna radna snaga
- Sa povecanjem uloge softvera u drustvu, njegova pouzdanost ima veci znacaj, jer od softvera vec uveliko zavise i ljudski zivoti
- Za razliku od ljudi, softver ne pravi slucajne greske, ali do gresaka ipak dolazi iz raznih razloga (mali promaci i male greske se vremenom ispoljavaju)
- U nastavku cemo predstaviti mere i modele pouzdanosti softvera, metode za poboljsanje pouzdanosti, kao i primere i u kojima su greske u sistemu dovele do ozbiljnih problema

2 Primeri padova softvera

- Izmedju ostalih, dodati primer za Boing (sad ovaj sto je pao, 10. marta)

3 "Program testing can be used to show the presence of bugs, but never to show their absence!" - Edsger W. Dijkstra

Greške u razvoju softvera su, kao i svuda, neizbežne. Ne možemo da se nadamo visokom nivou pouzdanosti softvera ako te greške ne pokušamo na neki način da uklonimo. Dakle, kao što se vidi na slici 1, u razvojnom ciklusu softvera testiranje ima izuzetno važnu ulogu.



Slika 1: Ciklus razvoja softvera

3.1 Ali...

Za skoro svaki, čak i najjednostavniji program postoji ogroman broj mogućih ulaznih vrednosti, tako da je iscrpno testiranje svih mogućnosti

praktično neizvodljivo. Na primer, da bismo testiranjem dokazali ispravnost programa koji sabira dva 64-bitna broja, bilo bi nam potrebno $2^{64} \cdot 2^{64}$ različitih testova. Uz pretpostavku da jedan test traje jednu nanosekundu, iscrpno testiranje bi trajalo 10^{22} godina. Testiranje je važno i podiže stepen pouzdanosti softvera, pomaže nam da uvidimo da postoje greške u programu, ali gotovo nikada ne možemo samo testiranjem da dokažemo da one ne postoje.

Očigledno nam je potreban bolji pristup za dokazivanje ispravnosti programa. Grana računarstva koja se time bavi je verifikacija softvera. Tačnije, proces verifikacije podrazumeva proveru da li program zadovoljava zadatu specifikaciju. Specifikacija predstavlja model problema i obično sadrži sledeće informacije [1]:

- ulazni parametri
- izlazni parametri
- globalne promenljive kojima se može pristupiti
- preduslovi - ograničenja na ulazne vrednosti
- postuslovi - uslovi koje izlazne vrednosti moraju da zadovolje (za ulaze koji zadovoljavaju preduslove)

Verifikacija se sprovodi pod pretpostavkom da je specifikacija validna. Sa druge strane, validacija proverava da li specifikacija zadovoljava korisničke potrebe. Razliku između verifikacije i validacije na interesantan način objasnio je Beri Bem:

> verification = Are we building the product right?
> validation = Are we building the right product?

U nastavku ćemo detaljnije objasniti različite vrste verifikacije sofvera, kao i njihov značaj za pouzdanost softvera.

3.2 Verifikacija softvera

Verifikacija softvera se u osnovi deli na dinamičku i statičku.
Dinamička verifikacija podrazumeva ispitivanje korektnosti programa u toku njegovog izvršavanja. Najčešći oblik dinamičke verifikacije softvera jeste testiranje.
Najšira podela testiranja [1]:

- testiranje crne kutije
  Naziva se i funkcionalno testiranje.
  Testovi se biraju na osnovu specifikacije.
- testiranje bele kutije
  Naziva se i strukturno testiranje.
  Testovi se biraju na osnovu interne strukture koda.

Sa druge strane, statička verifikacija podrazumeva analizu izvornog koda programa, na osnovu koje se dolazi do zaključaka o njegovoj korektnosti. Neke od metoda koje se koriste u statičkoj verifikaciji softvera su:

- simboličko izvršavanje [2]
  Koriste se simboličke promenljive umesto stvarnih ulaznih vrednosti, a kao izlaz se dobija simbolički izraz.
- apstraktna interpretacija [3]
  Posmatra se apstraktna semantika, nadskup konkretne semantike programa, u kojoj je rezonovanje jednostavnije.

3.3 Zašto samo odmah ne uradimo sve tačno?

Formalne metode verifikacije softvera podrazumevaju razvoj softvera direktno iz specifikacije i daju matematički dokaz korektnosti programa. Uz programski kod uporedo se formira i formalan matematički dokaz kojim se obezbeđuje ispravnost, bez potrebe za naknadnim testiranjem. U te svrhe se koriste alati za interaktivno dokazivanje teorema, kao što su Isabelle/HOL [4] i Coq [5]. Ovakvo rešenje je najbolje što se tiče pouzdanosti softvera, jer je matematički dokazana ispravnost. Međutim, zbog potrebe za stručnjacima koji će to sprovoditi, povećavaju se troškovi, kao i trajanje razvojnog procesa, tako da ne čudi što se u industriji ove metode koriste samo u retkim slučajevima, i to samo za najkritičnije delove koda. Neki od primera formalno verifikovanog softvera su:

- CompCert - kompilator za programski jezik C [6]
- seL4 - jezgro operativnog sistema (koristi se u avio-industriji) [7]
- CakeML - implementacija programskog jezika ML sa formalno verifikovanim kompilatorom [8]
- IronFleet - distribuirani sistemi [9]

primer nekog dokaza u izabelu u dodatku...

4 Modeli i metrike pouzdanosti softvera

Kao što je prethodno napomenuto, prilikom izgradnje softvera, veoma je važno imati na umu da se greška može naći u bilo kojoj fazi njegovog razvoja. Zato je bitno da se pronađe način procene broja potencijalnih grešaka u sistemu.
Statistika pokazuje da se krajem 20. veka na 1000 linija koda pojavi oko 8 grešaka, ne računajući prethodno testiranje sistema. Različiti izvori napominju da je danas taj broj veći, nabraja se 15-50 grešaka na 1000 linija. Majkrosoft tvrdi da njihov kod sadrži 0.5 neispravnosti na istom broj linija, dok NASA čak ni na 500,000 linija programskog koda ne sadrži niti jedan softverski problem [10]. Ovi brojevi su rezulat mnogih spoljašnjih faktora i nisu odgovarajuće merilo propusta i grešaka. Zbog toga su napravljeni razni modeli i metrike koji preciznije daju informaciju o stabilnosti softvera. U nastavku će biti opisana dva tipa takvih modela.

4.1 Deterministički modeli

Ovaj model odlikuje preciznim merama i podacima koji se oslanjanju na same karakteristike programskog koda. Prilikom ocenjivanja, ne uzimaju se u obzir slučajni događaji i greške koji oni prouzrokuju, već su performanse računate prema egzaktnim podacima. Oni uključuju broj različitih operatora, operanada, kao i broj mašinskih instrukcija i grešaka. Fokus je na mehanizmu i načinu rada samog softvera, gde se uvek mogu predvideti očekivana stanja.
Dva najpoznatija deterministička modela su Holstedova metrika i Mek-Kejbova ciklomatična složenost.[1]

[1] Moris Hauard Holsted (1977), Tomas Mek-Kejb (1976)

4.1.1 Holstedova metrika

Ova metrika se smatra jednom od najboljih za procenu kompleksnosti softvera, ali i težinu prilikom njegovog testiranja i debagovanja. U obzir ocene implementacije ulazi broj instrukcija i operacija izvornog koda, koji će biti drugačiji za svaki programski jezik ili arhitekturu na kojoj se program izvršava. Uvodi se sledeća notacija:

Obeležje	Opis	Formula
n_1	broj jedinstvenih operatora	
n_2	broj jedinstvenih operanada	
n	vokabular programa	$n_1 + n_2$
N_1	ukupan broj operatora	$n_1 \cdot log_2(n_1)$
N_2	ukupan broj operanada	$n_2 \cdot log_2(n_2)$
N	dužina programa	$N_1 + N_2$
I	broj mašinskih instrukcija	
V	obim programa	$N \cdot log_2(n_1 + n_2)$
D	težina razumevanja i debagovanja	$n_1/2 \cdot N_2/2$
M	vreme utrošeno na implementaciju	$V \cdot D$
T	vreme utrošeno na testiranje	$M/18$
E	broj grešaka i bagova	$V/3000$

Tabela 1: Obeležja u Holstedovoj metrici. Brojevi su dobijeni empirijski. [11]

- TODO: ubaciti primer iz knjige?

4.1.2 Mek-Kejbova ciklomatična složenost

Ova metrika računa složenost dijagrama koji se pravi na osnovu grafa kontrole toka podataka programa. Čvorovi grafa predstavljaju različite komande, a oni su povezani usmerenom granom ako je sledeća naredba prvog čvora ona naredba u drugom čvoru.
U slučaju da je graf povezan, ciklomatični broj jednak je broju linearno nezavisnih putanja kroz graf. Jednostavnije rečeno, ako program ne sadrži uslove i grananja, taj broj je 1. Zatim bi se taj broj povećao za 1 svaki put kada se naiđe na neku od ključnih reči: if, while, for, repeat, and, or itd. U naredbi case bi se svaki slučaj posebno računao. Mek-Kejb je za Fortran okruženje odredio da je gornja granica za ciklomatični broj 10, i program se u tom slučaju može smatrati visokog kvaliteta.
Ovom metrikom se mogu meriti i manje celine izvornog koda, kao što su pojedinačne funkcije, klase, moduli i potprogrami.
Ciklomatični broj C(G) grafa G se računa po formuli:

$$C(G) = E - V + 2P,$$

gde je:
E = broj grana grafa
V = broj čvorova grafa
P = broj povezanih komponenti grafa

Ciklomatični broj grafa sa više od jedne povezane komponente se računa kao suma ciklomatičnih brojeva svakih od povezanih komponenti. Kada se merenje vrši nad funkcijom, P će biti 1, jer funkcija predstavlja

jedinstvenu povezanu komponentu.
- TODO: ubaciti primer iz knjige?

4.2 Probabilistički modeli

Ovaj model pojavu grešaka i kvarova gleda kao na verovatnosne događaje. Softver se posmatra u fazi izvršavanja i pravi se mreža modela koja predstavlja ponašanje programa. Rezultat ovog modeliranja je primenjiv na razne metode iz oblasti statistike, što omogućava dubinsku matematičku analizu, detekciju anomalija, generisanju testova i simulaciju raznih slučajeva. Postoji nekoliko probabilističkih modela, od kojih će u nastavku biti opisana tri proizvoljna.

4.2.1 Modeli stope neuspeha

Ova grupa modela se koristi radi prikazivanja kolika je stopa otkazivanja programa po grešci. Kako se broj preostalih grešaka menja, tako se i stopa neuspeha prilagođava promeni.
Postoji nekoliko varijacija ovih modela, ali se svi oslanjaju na princip da u kodu postoji N međusobno nezavisnih grešaka sa jednakom verovatnoćom ispoljavanja. Kada ona nastane, greška se otklanja i nijedna nova greška se ne pojavljuje prilikom uklanjanja.
Razlika je u tome što neki podmodeli podrazumevaju povećanje broja grešaka kroz vreme, drugi podrazumevaju smanjenje, a neki ga smatraju konstantnim. Takođe postoje modeli koji sa sigurnošću tvrde da je greška uklonjena, a postoje i oni koji uspešnost otklanjanja procenjuju verovatnoćom p.

4.2.2 Modeli rasta pouzdanosti

- TODO: reliability growth models

4.2.3 Markovljev model strukture

- TODO: Markov structure models

5 Budućnost softvera

- TODO

6 Zaključak

- TODO

Literatura

[1] J. Laski and W. Stanley, *Software Verification and Analysis*. London: Springer-Verlag, 2009.

[2] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[3] Patrick Cousot, "Abstract Interpretation," 2008. on-line at: https://www.di.ens.fr/~cousot/AI/.

[4] University of Cambridge and Technische Universität München, "Isabelle/HOL," 2018. on-line at: https://isabelle.in.tum.de/.

[5] Thierry Coquand, Gérard Pierre Huet, Christine Paulin-Mohring, Bruno Barras, Jean-Christophe Filliâtre, Hugo Herbelin, Chetan Murthy, Yves Bertot and Pierre Castéran, "Coq," 2019. on-line at: https://coq.inria.fr/.

[6] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, pp. 107–115, July 2009.

[7] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, (New York, NY, USA), pp. 207–220, ACM, 2009.

[8] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "Cakeml: A verified implementation of ml," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, (New York, NY, USA), pp. 179–191, ACM, 2014.

[9] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, "Ironfleet: Proving practical distributed systems correct," in *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, (New York, NY, USA), pp. 1–17, ACM, 2015.

[10] Hacker News, "Number of errors on 1000 lines of code," 2016. On-line at: https://news.ycombinator.com/.

[11] IBM Knowledge Center, "Halstead Metrics." On-line at: https://www.ibm.com/support/knowledgecenter/en/SSSHUF_8.0.2/com.ibm.rational.testrt.studio.doc/topics/csmhalstead.htm.

A Dodatak

Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe. Ovde pišem dodatne stvari, ukoliko za time ima potrebe.