Implementirano ≠ testirano ≠ ispravno

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

30. april 2019

Sažetak

Softver igra sve veću ulogu u modernom svetu i stoga je sve važnije da on bude ispravan. Navešćemo primere katastrofalnih softverskih grešaka, a zatim ćemo prikazati različite metode za izbegavanje i uklanjanje takvih grešaka u različitim fazama razvoja. Posvetićemo pažnju i mogućnostima unapređenja procesa razvoja softvera koje nam donose nove tehnologije.

Sadržaj

1 Uvod

U vreme „softverske krize“ započet je razvoj na polju softverskog inženjerstva. Programeri su radili na tome da isporuče što moćniji sistem i što efikasnije aplikacije, ali su nailazili na različite probleme, kao što su kašnjenje isporuka, troškovi veći od predviđenog, bagovi koje je teško ispraviti i još mnogo toga. Softver danas ima značajnu ulogu u društvu i sve je rasprostranjeniji, te je njegova pouzdanost veoma relevantna, posebno ako se u obzir uzme prisutnost u zdravstvu, gde ima direktan uticaj na ljudske živote.

Razvoj softvera je disciplina koja se bavi proizvodnjom softvera, razvojnih alata, metodologija i teorema koje podržavaju razvoj. Softverski inženjeri pri izradi prate proces od nekoliko koraka koji su prikazani na slici 3. U nastavku ćemo se fokusirati na deo sa testiranjem, kao i njegovu vezu sa pouzdanošću softvera. Pouzdanost softvera se definiše kao verovatnoća ispravnog rada sistema tokom određenog vremena, pri određenim uslovima [1]. Malo neformalnije, pouzdanost je mera slaganja sistema sa svojom specifikacijom.

Da bismo istakli važnost pouzdanosti, počećemo, u poglavlju 2, sa nekim primerima velikih problema koji su nastali zbog nepouzdanog softvera [2]. Potom ćemo se, u poglavlju 3, baviti ispravnošću programa, kao osnovom pouzdanosti, gde ćemo zagrebati površinu oblasti verifikacije softvera [3]. Nakon toga ćemo, u poglavlju 4, prikazati neke od modela i metrika koji se koriste u procenjivanju pouzdanosti softvera [1]. Na kraju ćemo, u poglavlju 5, razmotriti moguća unapređenja celokupnog procesa razvoja softvera korišćenjem novih dostignuća na polju veštačke inteligencije.

2 Primeri neispravnog softvera

Tokom proteklih decenija, tehnologija je doživela veliki napredak, ali usled grešaka u softveru je prouzrokovala velike materijalne gubitke, kao i ljudske žrtve. Kada započne sa radom i pokaže se ispravnim i odgovarajućim, očekuje se da će softver zadržati to svojstvo. Međutim, serije tragedija i haosa koje su uzrokovane padom softvera pokazuju suprotno.

Primer 2.1 ***Therac-25, 1985.***

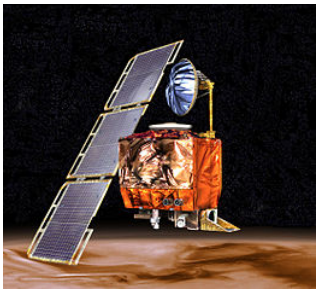
Therac-25 je bila računarski kontrolisana mašina za terapiju zračenjem, proizvedena 1982. Zbog grešaka u konkurentnom programiranju[1], softver nije bio u stanju da detektuje određena stanja, a deo hardvera za interno zaključavanje, koji su prethodni modeli ove mašine posedovali i koji je služio da spreči ovakve greške ukoliko do njih dođe, otklonjen je i sve provere bezbednosti prepuštene su softveru. Ovaj propust je doveo do masovne pojave neodgovarajućih emisija zračenja. Između juna 1985. i januara 1987. dogodilo se šest poznatih nesreća uzrokovanih predoziranjem zračenja koje su dovele do smrti i ozbiljnih povreda pacijenata. Prikaz mašine na slici 1.

[1] Višenitno programiranje, gde se skup sekvencijalnih programa izvršava paralelno.

Slika 1: Therac-25

Primer 2.2 ***Marsov orbiter za proučavanje klime, 1999.***

Marsov orbiter za proučavanje klime (slika 2) je bila robotska svemirska sonda koju je *NASA (eng. National Aeronautics and Space Administration)* lansirala 11. decembra 1998. sa ciljem da istraži klimu i promene na površini Marsa. Kontakt sa sondom izgubljen je 23. septembra usled softverske greške, naime u računaru u kontroli misije na Zemlji koji je jedinicu za vrednost potiska[2] izražavao američkim jedinicama (funtama u sekundi), umesto standardnim jedinicama SI sistema (njutnima u sekundi) kako je bilo ugovoreno između agencije *NASA* i kompanije Lokid Martin. Usled toga je sonda pri ulasku u Marsovu orbitu išla trajektorijom koja je bila preblizu njegove površine i usled zagrevanja, prolaskom kroz gornje slojeve atmosfere se raspala.
Cena ove greške iznosila je oko $125.000.000.



Slika 2: Marsov orbiter za proučavanje klime

Primer 2.3 ***Letovi u Los Anđelesu***

14. septembra 2004. godine, preko 400 aviona je izgubilo vezu sa kontrolom leta u blizini aerodroma u Los Anđelesu, međutim, zahvaljujući rezervnoj opremi unutar aviona, do nesreće nije došlo. Uzrok gubitka veze je bila greška prekoračenja u brojaču milisekundi u okviru sistema za komunikaciju sa avionima. Ova greška je bila otkrivena i pre incidenta, ali tada je sistem već bio isporučen i instaliran na nekoliko aerodroma, pa nije bilo moguće jednostavno ga popraviti i zameniti, već je preporučeno da se sistem resetuje na svakih 30 dana, kako ne bi došlo do prekoračenja. Ova procedura nije bila ispoštovana, i do greške je došlo posle tačno 2^{32} milisekundi, odnosno 49.7 dana od uključivanja sistema.

[2] Sila koja gura telo u suprotnom smeru od kojeg ga neke druge sile vuku, primer sila koja deluje na telo potopljeno u tečnost u suprotnom smeru od gravitacione sile.

Primer 2.4 ***Boing 737 MAX***

Usled greške u softveru koji je „verovao“ da je avion u opasnosti od iznenadnog gašenja motora i forsirao prednji deo aviona da počne da se spušta. Istražitelji su najveću pažnju posvetili proveri računarskog programa MCAS *(eng. Maneuvering Charateristics Augmentation System)*, napravljen da pomogne pilotima da drže kontrolu nad avionima ovog tipa. Program reaguje kada senzori u kljunu aviona pokažu da se letelica penje pod prevelikim uglom, što može da dovede do gubitka kontrole. Piloti su se borili protiv sistema aviona ceo put i ispoštovali sve procedure pre pada aviona. Uprkos njihovim naporima, nisu uspeli da uspostave kontrolu nad letelicom i avion se srušio 10. marta 2019. i u nesreći je poginulo svih 157 osoba koje su bile u avionu, od kojih je jedna žrtva iz Srbije.

3 Ispravnost programa

Greške u razvoju softvera su, kao i svuda, neizbežne. Ne može se očekivati visok nivo pouzdanosti softvera, ako se ne pokuša uklanjanje grešaka, koje se mogu javiti u bilo kom delu sistema. Dakle, kao što se vidi na slici 3, u razvojnom ciklusu softvera testiranje ima izuzetno važnu ulogu.

Slika 3: Ciklus razvoja softvera

3.1 Model problema i verifikacija softvera

> *„Program testing can be used to show the presence of bugs, but never to show their absence!“*
>
> Edsger W. Dijkstra

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 bi se testiranjem dokazala ispravnost programa koji sabira dva 64-bitna broja, bi bi 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 se ne može samo testiranjem dokazati da one ne postoje.

Očigledno 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 [3]:

- 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. Verifikacija i validacija se često, iako to nije tačno, izjednačavaju i poistovećuju sa testiranjem. Razliku između ova dva pojma na interesantan način objasnio je Beri Bem:

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

3.2 Vrste verifikacije i njihov značaj za pouzdanost 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 [3]:

- 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 kôd.

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

- simboličko izvršavanje [4]
  Koriste se simboličke promenljive umesto stvarnih ulaznih vrednosti, a kao izlaz se dobija simbolički izraz.
- apstraktna interpretacija [5]
  Posmatra se apstraktna semantika, nadskup konkretne semantike programa, u kojoj je rezonovanje jednostavnije.

3.3 Formalni dokazi ispravnosti

Formalne metode verifikacije softvera podrazumevaju razvoj softvera direktno iz specifikacije i daju matematički dokaz korektnosti programa. Uz programski kôd 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 [6] i Coq [7]. 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 kôda. Neki od primera formalno verifikovanog softvera su:

- CompCert - kompilator za programski jezik C [8]
- seL4 - jezgro operativnog sistema (koristi se npr. u avio-industriji) [9]
- CakeML - implementacija programskog jezika ML sa formalno verifikovanim kompilatorom [10]
- IronFleet - distribuirani sistemi [11]

Radi ilustracije, u dodatku A se nalazi primer dokaza algoritma quicksort iz dokumentacije Isabelle-a.

3.4 Primer alata za testiranje - Selenium

Selenium je skup softverskih alata koji služi da podrži i obezbedi sprovođenje automatskog i kontinuiranog testiranja softvera. Ovakav način testiranja obezbeđuje postojanu proveru funkcionalnosti i automatsko prijavljivanje nepravilnosti u radu softvera, što predstavlja veliku prednost u odnosu na manuelno testiranje softvera.

U osnovi Selenium je alat koji obezbedjuje automatizaciju internet pretraživača. To znači da pomoću njega možemo upravljati radom pretraživača, kontrolišući njihovo ponašanje i akcije koje će unutar pretraživača biti inicirane. Selenium skripte sadrže tipski grupisane akcije i iteracije nad pretraživačima i obezbeđuju njihovo automatizovano izvršavanje, što se u praksi koristi i za automatizaciju zadataka koji se često ponavljaju ili u svrhu kontinuiranog testiranja softvera. Prema podacima matičnog sajta [12], Selenium predstavlja najzastupljenije rešenje otvorenog kôda u oblasti automatizovanog testiranja.

Selenium je multiplatformsko rešenje u pravom smislu te reči. Podržan je rad sa svim trenutno bitnim veb pretraživačima, kao i pisanje testova u nekom od sledećih programskih jezika: Java, C#, Ruby, Python, Perl i PHP. Ovaj fleksibilni alat može se koristi samostalno, kao i u kombinacija sa drugim test frejmvork platformama[3]. Poseduje mogućnost brze i lake integracije sa mnogim eksternim alatama i API-jima. Inicijalna verzija Selenium-a pojavila se 2004 godine, a trenutno je aktuelna verzija 2.45.0. U pitanju je rešenje koje je u potpunosti otvorenog kôda, razvijen je pod Apache licencom verzije 2.

[3] Frejmvork (eng. framework) obezbeđuje već implementirane funkcionalnosti koje se mogu proizvoljno menjati od strane programera, uz značajnu uštedu vremena.

4 Modeli i metrike pouzdanosti softvera

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 kôda 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 kôd sadrži 0.5 neispravnosti na istom broju linija, dok *NASA* čak ni na 500,000 linija programskog kôda ne sadrži niti jedan softverski problem [13]. 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

Ovi modeli odlikuju preciznim merama i podacima koji se oslanjanju na same karakteristike programskog kôda. Prilikom ocenjivanja, ne uzimaju se u obzir slučajni događaji i greške koje 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[4].

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 kôda, koji će biti drugačiji za svaki programski jezik ili arhitekturu na kojoj se program izvršava. U tabeli 1 se uvodi 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	$n_1/2 \cdot N_2/n_2$
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 [14]

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

Bitno je napomenuti da ne postoji tačno određeno pravilo koje definiše šta spada u operatore, a šta u operande, ali postoje smernice za neke od najpopularnijih jezika. Što se tiče jezika C, pod operatorom se, pored podrazumevanih, smatraju i većina rezervisanih ključnih reči i pozivi funkcija, dok se operandima smatraju identifikatori i konstante.

Primer 4.1 *Primena Holstedove metrike na jednostavnom C kôdu. [15]*

```
1000 main()
     {
1002     int a, b, c, avg;
         scanf("%d %d %d", &a, &b, &c);
1004     avg = (a + b + c) / 3;
         printf("avg = %d", avg);
1006 }
```

Jedinstveni operatori su: (), {}, +, =, /, &, ',', ;, int, scanf, printf, main
Jedinstveni operandi su: a, b, c, avg, 3, "%d %d %d", "avg = %d"
Dakle, prateći uvedenu notaciju:

- $n_1 = 12$, $n_2 = 7$, $n = 19$
- $N_1 = 27$, $N_2 = 15$, $N = 42$
- Obim: $V = 42 \cdot log_2 19 \approx 178.4$
- Težina: $D = \frac{12}{2} \cdot \frac{15}{7} \approx 12.85$
- Vreme za implementaciju: $M = 178.4 \cdot 12.85 \approx 2292.44$
- Vreme za testiranje: $T = \frac{2292.44}{18} \approx 127.357$
- Broj bagova: $E = \frac{178.4}{3000} \approx 0.06$

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.[16]

Ovom metrikom se mogu meriti i manje celine izvornog kôda, 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.

4.2 Probabilistički modeli

Ovi modeli pojavu grešaka i kvarova gledaju 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 dva 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 kôdu 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

Ova grupa modela predviđa da li će doći do poboljšanja pouzdanosti kroz testiranje softvera. Model rasta je zapravo odnos pouzdanosti i stope neuspeha programa predstavljen kao funkcija vremena ili kao broj testova. Postoje dva podmodela: prvi dovodi u korelaciju broj pronađenih i razrešenih grešaka, ukupni akumulirani broj grešaka u vremenskoj jedinici, dok drugi predviđa broj grešaka za vreme testiranja u jedinici vremena. U ovom modelu je najčešća vremenska jedinica jedna nedelja.

5 Budućnost softvera

Sa poboljšanjem hardvera, mogućnosti softvera postaju sve veće. Pomaci u oblastima veštačke inteligencije i mašinskog učenja omogućavaju mašinama da zamene ljude u nekim delatnostima, što donosi mnoge prednosti kao što su ubrzanje rada, smanjenje troškova i uklanjanje grešaka koje ljudi prave.

5.1 Veštački programeri - fantastika ili realnost?

Postoje mnogi programerski alati koji pomažu programerima tako što vrše automatsko generisanje kôda, optimizaciju ili debagovanje. Sa na-

pretkom veštačke inteligencije, mogućnost automatizacije raznih faza razvoja softvera raste.

Naravno, ideja da će računar sam da napiše komplikovan program je naučna fantastika. Da bismo napravili model koji će da napiše neki program, često nam je potrebno više posla nego da sami napišemo taj program. Međutim, programeri danas retko pišu ceo kôd od početka do kraja. Najčešće koristimo razne biblioteke i API-je, koji se konstantno menjaju, unapređuju, a i pišu novi. Učenje novih API-ja postaje iscrpljujuće za programera, te nastaje potreba za sistemima koji bi, za određeni ulaz programera, zaključili šta je zadatak i generisali gotov kôd. Jedan takav sistem je Bayou [17]. Njegove mogućnosti su veoma ograničene, ali sa novim poboljšanjima na polju mašinskog učenja i veštačke inteligencije, možemo očekivati sve više takvih sistema u budućnosti.

Ova oblast tek počinje da se razvija, pa napredni sistemi tog tipa, ako postoje, nisu dostupni široj javnosti. Jasno je da ako je sistem ispravan, generisaće ispravne programe. Međutim, verifikacija sistema koji se trenira mašinskim učenjem nije jednostavna, a ni moguća u svim slučajevima. Programe dobijene na ovakav način će biti neophodno verifikovati i najverovatnije modifikovati, ali će proces razvoja biti znatno ubrzan upotrebom veštačke inteligencije.

6 Zaključak

Na osnovu svega navedenog, zaključujemo da u razvoju softvera nije dovoljno poznavati samo programski jezik i biti vešt u programiranju, već je neophodno imati duboke uvide u domen primene. Najbolje bi bilo da sve što implementiramo i formalno dokažemo. Međutim, kako je to najčešće neizvodljivo, tj. neisplativo, moramo se ozbiljno posvetiti praćenju i održavanju nivoa pouzdanosti softvera na neki drugi, „jednostavniji" način, poput testiranja. Moramo biti svesni da će do grešaka doći. Biće nam lakše da se na to pripremimo i adekvatno odreagujemo kada do greške dođe ako unapred znamo koji delovi su podložni padovima i sa kojom verovatnoćom. Očekujemo da će u budućnosti mnogi koraci razvoja biti automatizovani pre svega upotrebom veštačke inteligencije, ali i drugih metoda.

Literatura

[1] H. Pham, *System Software Reliability (Springer Series in Reliability Engineering)*. Berlin, Heidelberg: Springer-Verlag, 2005.

[2] M. J. Quinn, *Ethics for the Information Age*. Pearson, 7th ed., 2016.

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

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

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

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

[7] 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/.

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

[9] 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.

[10] 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.

[11] 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.

[12] "Selenium." Main website at: https://www.seleniumhq.org.

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

[14] 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.

[15] "Halstead metric example." From Eindhoven University of Technology: https://www.tue.nl/en/university/departments/mathematics-and-computer-science/.

[16] Hoang Pham, "System Software Reliability," 2006. Chapter 5.3: McCabe's Cyclomatic Complexity Metric.

[17] Department of Computer Science, Rice University, "Bayou." On-line at: http://askbayou.com//.

A Dodatak

Algoritam sortiranja quicksort implementiran je na prirodan način za funkcionalni programski jezik. Njegova korektnost dokazana je pomoću matematičke indukcije, koristeći ugrađene automatske dokazivače teorema uz pomoć nekih već dokazanih svojstava sortiranih lista.

```
(*  Author:     Tobias Nipkow
    Copyright   1994 TU Muenchen
*)

section <Quicksort with function package>

theory Quicksort
imports "HOL-Library.Multiset"
begin

context linorder
begin

fun quicksort :: "'a list => 'a list" where
  "quicksort []     = []"
| "quicksort (x#xs) = quicksort [y<-xs. \<not> x\<le>y] @ [x] @
    quicksort [y<-xs. x\<le>y]"

lemma [code]:
  "quicksort []     = []"
  "quicksort (x#xs) = quicksort [y<-xs. \<not> x\<le>y] @ [x] @
     quicksort [y<-xs. x\<le>y]"
  by (simp_all add: not_le)

lemma quicksort_permutes [simp]:
  "mset (quicksort xs) = mset xs"
  by (induct xs rule: quicksort.induct) (simp_all add: ac_simps)

lemma set_quicksort [simp]: "set (quicksort xs) = set xs"
proof -
  have "set_mset (mset (quicksort xs)) = set_mset (mset xs)"
    by simp
  then show ?thesis by (simp only: set_mset_mset)
qed

lemma sorted_quicksort: "sorted (quicksort xs)"
  by (induct xs rule: quicksort.induct) (auto simp add:
     sorted_append not_le less_imp_le)

theorem sort_quicksort:
  "sort = quicksort"
  by (rule ext, rule properties_for_sort) (fact quicksort_permutes
      sorted_quicksort)+

end

end
```

Listing 1: Isabelle dokaz korektnosti algoritma quicksort