UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Marina R. Nikolić

PRIKUPLJANJE I PRIKAZ PODATAKA O IZVRŠAVANJU PROGRAMA

master rad

Beograd, 2018.

Mentor:

dr Milena Vujošević Janičić, docent
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Filip Marić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Milan Banković, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Mentoru za predanost i pomoć, firmi za resurse, porodici i prijateljima za podršku

Naslov master rada: Prikupljanje i prikaz podataka o izvršavanju programa

Rezime: tekst apstrakta rada

Ključne reči: profajliranje, pokrivenost koda, GCC, GCOV

Sadržaj

Glava 1

Uvod

1. kratak opis o čemu će biti reči u daljem tekstu
2. iako vidim da je popularno po master radovim ada se piše po poglavljima ovde (tipa, u poglavlju X je opisano to i to), ja bih uvod radije sročila kao pričicu koja prati rad
3. ovde bih dodala na samom početku i na samom kraju značaj teme kao takve i naravno značaj mog doprinosa (na kraju zbog efekta)

Glava 2

Analiza performansi programa

Razvoj softvera je znatno širi pojam od pisanja koda. Obuhvata više, podjednako važnih segmenata, kao što su: planiranje, analiza i usklađivanje sa zahtevima klijenata, testiranje, analiza performansi, optimizacija ili održavanje. Samo investiranjem u svaki ponaosob, može se proizvesti kvalitetan i dugotrajan softver. Njihova kompleksnost proporcijano raste sa značajem i složenošću krajnjeg proizvoda, iz čega proističe i važnost njihovog olakšavanja. Postoje brojne metodologije i tehnike koje su specijalizovane za vođenje procesa karakterističnih za rane faze razvoja, kao što su planiranja ili analize zahteva. Međutim, ovaj rad će se usresrediti na prikaz onih koje olakšavaju procese kasnog razvoja, pre svega testiranja i optimizacije. Za uspešno sprovođenje tih procesa, važan faktor je odabir tehnika koje će se primenjivati i jedinica koda kome su oni najneophodniji, a kvalitetan odabir je uslovljen dobrim poznavanjem samog softvera, njegovih karakteristika i ponašanja. Takvu vrstu informacije obezbeđuje analiza programa.

2.1 Vrste analize programa

U okviru analize programa razmatraju se razni aspekti softvera, pre svega, aspekti ponašanja softvera u različitim sučajevima upotrebe. Analiza softvera može da bude automatizovana i u nastavku teksta biće razmatrani koncepti koji su vezani za automatizovane pristupe. Cilj analize je olakšavanje procesa testiranja korektnosti, naročito eksterno nabavljenih delova softvera kao i procene performansi i optimizacije. Analiza treba da pruži korisne informacije o raspodeli potrošnje resursa, čvorovima ekstremne potrošnje, potencijalnim kritičnim segmentima izvršavanja, korektnosti toka izvršavanja i slično. Poput projekta veštačke inteligencije, njen

krajnji cilj je stvaranje „pametnog prevodioca", koji bi mogao automatski generisati efikasan, a pouzdan kôd. Značaj njenih trenutnih mogućnosti, kao i brzina kojom se unapređuje, ukazuju na veliku verovatnoću ostvarljivosti tog cilja. Analiza programa je veoma širok pojam, koji obuhvata veliki broj vrlo raznovrsnih metoda, ali se može veoma precizno podeliti na dva osnovna tipa. To su statička i dinamička analiza.

Statička analiza programa

Statička analiza programa [16] obuhvata sve metode i tehnike utvrđivanja ponašanja programa, za koje ga nije potrebno izvršiti. Sve procedure se vrše nad izvornim kodom i, prikupljajući podatke o njegovoj strukturi, generišu korisne informacije o mogućim ishodima njegovog budućeg izvršavanja. Primer su mnogobrojne softverske metrike, koje na osnovu podataka o broju linija, klasa ili metoda, izračunavaju takozvani „statistički kvalitet" softvera.

Njena glavna prednost proističe upravo iz toga, što kôd nije potrebno izvršiti. Ovakvim ograničenjem se često odlikuje razvoj velikih i skupih softverskih sistema, gde se zbog materijalnih mogućnosti ne može vršiti testiranje svih manjih jedinica u realnom okruženju. Kao ilustrativan primer se može posmatrati razvoj softera za automatsko navođenje rakete i jedan manji segment tog razvoja koji predstavlja program za izračunavanje potrošnje goriva prilikom jednje vožnje. Testiranje korektnosti sastavne jedinice te veličine se u najvećoj meri vrši na simulatorima. Lansiranje prave rakete za potrebe ovakvog testiranja je ekonomski neopravdano, iako okruženje koje simulator pruža ne obuhvata sve alternativne slučajeve upotrebe.

Sa druge strane, ukoliko uzmemo u obzir činjenicu da vreme izvršavanja proizvoljnog programa može biti proizvoljno dugo, iz neneophodnosti izvršavanja, može se izvesti još jedna velika prednost statičke analize, a to je brzina. Faktor brzine čini osnovu ocene svakog pristupa.

Statička analiza programa se ne bazira na podacima iz konkretnih izvršavanja, već nepromenljivim i sigurnim podacima izvornog koda, zbog čega je odlikuje i nepristrasnost. Nezavisnost od ulaznih podataka i okruženja, omogućava efikasnu detekciju graničnih slučajeva.

Osnovne mane softverskih metrika su uzrokovane uskom vezom njihovih tehnika sa statistikom kao naukom i predstavljaju nepreciznost i smanjenu informativnost o praktičnim slučajevima upotrebe. Rezultati nisu eksperimentalne prirode, već prika-

zuju teorijsko predviđanje ponašanja. Zbog toga se ne trebaju smatrati potvrdama ispravnosti ili performansi, već isključivo tretirati kao smernice pri razvoju.

Postoje i određene statičke metode koje su značajno preciznije od metrika, poput simboličkog izvršavanja [17], proveravanja modela [19] ili apstraktne interpretacije [11]. Ove metode simuliraju ponašanje programa uzimajući u obzir i ulazne vrednosti, čime se povećava preciznost i informativnost. Međutim, uticaj realnih parametara okruženja, čija specifikacija nije u potpunosti poznata, se i dalje zasniva na predviđanju i statističkim informacijama o slučajevima upotrebe. Kao primer nedovoljno potpune specifikacije se mogu posmatrati eksterno nabavljene komponente sa zatvorenim kodom. Nepoznavanje svih alternativnih tokova upotrebe ili greške u dokumentaciji mogu prouzrokovati slabosti u modelima kreiranim ovim metodama.

Dinamička analiza programa

Dinamička analiza programa [13] obuhvata sve metode i tehnike prikupljanja podataka o programu tokom njegovog izvršavanja i utvrđivanja ponašanja programa na osnovu tih podataka. Procedure uglavnom započinju u fazi prevođenja, ali najvažniji deo se obavlja u toku i nakon izvršavanja. Pored strukture koda i statičkih podataka, na njen ishod utiču i ulazne vrednosti, kao i parametri okruženja. Testovi jedinica koda, sistemski testovi i testovi prihvatljivosti koriste isključivo ovaj vid analize programa.

Sve njene glavne prednosti u odnosu na statičku analizu, proističu iz uticaja „realnih parametara". Određene mane softverskih rešenja ispoljavaju se samo u toku rada tog softvera, a mnoge i proističu upravo iz spoljnih faktora ili veze sa njima. Statistički savršen softver koji je nedovoljno primenljiv u praksi, predstavlja jedan od tri osnovna neuspeha prilikom razvoja softvera [6]. Marketinška istraživanja, analize zahteva korisnika i detaljni popisi slučajeva upotrebe se primenjuju u ranim fazama razvoja softvera u cilju zaštite od ove vrste neuspeha. Međutim, pojedini faktori okruženja, poput vrednosti jedne jedinice iz skupa obrade, čiji uticaj se zanemaruje kao dozvoljeno odstupanje, greška zaokruživanja ili usled efekta mase, tzv. „lažne pozitivne ili negativne vrednosti", se ne mogu detektovati metodama koje se baziraju na statistici. Kao ilustrativan primer može se posmatrati testiranje uspešnosti prenosa bitova kroz određeni fizički medijum i sledeći rezultati testiranja: 1 000 000 000 bitova koji su uspešno stigli na destinaciju i 10 izgubljenih bitova. Procenat neuspeha iznosi 0.000001%, što se zaokruživanjem na 5 ili manje decimala

svodi na 0%. Na osnovu ovog podatka, može se zaključiti da je testiranje završeno uspešno, i pritom potpuno zanemariti značaj izgubljenih delova informacije.

Posledice zanemarivanja uticaja pojedinačnih slučajeva obuhvataju brojna prilagođavanja i održavanja u kasnim fazama razvoja, koja se neretko završavaju odustajanjem od razvoja nakon isteka novčanih sredstava ili pronalaska kvalitetnijeg rešenja. Zbog toga su testiranja u realnom okruženju veoma važna, a kako su po svojoj prirodi ograničena resursima, važno je i iz njih ekstrahovati što više informacija za naredne iteracije razvoja. Njih obezbeđuje dinamička analiza programa.

Važna prednost dinamičke analize je i univerzalnost, koja proističe iz činjenice da se sve tehnike primenjuju na izvršnu verziju, bez neophodnog prisustva izvornog koda. Oblast primene je šira, jer obuhvata i programe sa „zatvorenim" kodom. Pisanje celokupnog koda softvera je skupo, kako u ekonomskom, tako i u pogledu utrošenog vremena, zbog čega ne predstavlja dovoljno kompetetivan način proizvodnje. Ovaj princip nije karakteristika samo softverske industrije, već je globalna odlika industrije kao grane privrede. Kao ilustrativan primer se može posmatrati javni prevoz građana i porediti cena jedne autobuske karte u odnosu na cenu goriva i održavanja automobila, na relaciji od nekoliko kilometara. Ukupna cena jednog prevoza se ravnomerno raspoređuje na više putnika, čime je pojedinačna cena po putniku znatno manja. Sa druge strane, prevoznik nema obavezu da proizvod ustupi za tačnu cenu pojedinačnog dela, količnika cene vožnje i broja putnika, iz čega proističe njegova zarada. Postavljanje previsoke cene u cilju maksimalne zarade može prouzrokovati manjak interesovanja za proizvod, usled neisplativosti korisniku, te njen izbor mora biti izbalansiran rezultatima pažljivog proučavanja tržišta. Cena održavanja automobila je dodata u ilustraciju, u cilju naglašavanja troškova održavanja softvera, koje često predstavlja najveći materijalni rashod razvoja. U razvoju softvera, ovaj princip se ogleda u eksternom nabavljanju komponenti, u kom slučaju se često može kupiti samo izvršna verzija. Izvorni kôd predstavlja poslovnu tajnu proizvođača. Procene kvaliteta pre integracije, kao i testiranje kompatibilnosti sa ostatkom softvera, stoga se mogu obaviti jedino dinamičkim pristupom.

Najveća mana ovog pristupa jeste potencijalni osećaj lažne sigurnosti. To je, u određenoj meri, neizostavna stavka svakog testiranja. Neiskusni razvojni timovi se mogu previše osloniti na rezultate analize i time prevideti činjenicu da ona, kao automatizovani proces, ne može garantovati stoprocentnu ispravnost. Alati koji je vrše su takođe softverski proizvodi, i samim tim jednako podložni greškama koliko i kôd koji se njima analizira.

2.2 Profajliranje programa

Posebno mesto u tehnikama dinamičke analize ima profajliranje, i njemu će ovaj rad biti u potpunosti posvećen.

Značaj profajliranja

Profajliranje [14, 20] predstavlja prikupljanje raznih podataka iz izvršavanja programa u realnom ili simuliranom okruženju, koji pružaju uvid u tok i performanse rada programa. Obradom ovih podataka, dobijaju se vredne informacije o vremenskim i memorijskim zahtevima programa, složenosti i iskorišćenosti pojedinih delova koda i slično. Rezultati rada alata za profajliranje predstavljaju korisne smernice za procese testiranja i optimizacije, jer ukazuju na delove koda kojima su oni najneophodniji.

Ulazne vrednosti i parametri okruženja, zajedno sa kodom programa, jedinstveno određuju tok izvršavanja. Uočavanje pozitivnih podataka o izvršavanju van predviđenog toka, ili negativnih u njegovoj unutrašnjosti, za unapred određen slučaj upotrebe, je stoga dobar pokazatelj da se u kodu nalaze greške.

Kao ilustrativan primer mogu se posmatrati program za obradu teksta i slučaj upotrebe koji se sastoji iz tri koraka: učitavanje teksta, podebljavanje jedne reči i memorisanje izmena. Predviđen tok izvršavanja obuhvata prolazak kroz pet funkcija: otvaranje željenog fajla, prikazivanje teksta na ekranu, podebljavanje odabrane reči, memorisanje promena i zatvaranje programa. Na osnovu ovog toka, izvodi se teorijski zaključak da se funkcija koja vrši podebljavanje teksta izvršila, dok funkcija koja iskrivljuje tekst nije. Ukoliko eksperimentalni podaci, poreklom iz konkretnog izvršavanja, nisu u skladu sa teorijskom pretpostavkom, već potvrđuju izvršavanje funkcije za iskrivljivanje ili negiraju izvršavanje funkcije za podebljavanje teksta, može se zaključiti da se program ne izvršava pravilno. Efekat ove dve funkcije se može u određenim slučajevima primetiti i na osnovu prikaza na ekranu, međutim izostanak efekta memorisanja izmena gotovo sigurno neće biti uočen u odgovarajućem trenutku.

Profajliranje pruža i dodatnu olakšicu za budući proces „debagovanja", sužavanjem oblasti pretrage. Detekcija memorijski ili vremenski izrazito zahtevnih segmenata, kao i segmenata koji se veoma često izvršavaju usmerava pažnju razvojnog tima na neophodnost optimizacije, pritom takođe obezbeđujući dodatnu informaciju gde je ona i koliko potrebna. Poređenjem performansi različitih verzija koda, može

se izvršiti dobra procena kvaliteta i odabir odgovarajućeg algoritma u ranim fazama, kada je njegova zamena u velikoj meri jeftinija. Smernice koje profajleri daju mogu znatno „očistiti" kôd od nepotrebnih grananja, logički neiskorišćenih promenljivih, „mrtvog koda" i sličnih propusta. Stoga značajno olakšavaju i proces refaktorisanja koda. Vrši se alatom koji se naziva profajler i sastoji se od tri usko spregnute faze: instrumentalizacija, prikupljanje i obrada podataka.

Faze profajliranja

Instrumentalizacija [7] koda predstavlja ubacivanje dodatnih instrukcija u program sa ciljem merenja karakteristika programa. Instrukcije predstavlju kôd inicijalizacija određenih dodatnih struktura za instrumentalizaciju i pravila za njihovo popunjavanje. Dodatne strukture imaju ulogu skladišta za metapodatke, a za popunjavanje je zadužen sam instrumentalizovani program. Time se stvara opterećenje i smanjuju performanse, ali je, iz više razloga, najpouzdanije i najoptimalnije moguće rešenje. Prvenstveno, iz ugla bezbednosti. Neograničen pristup internim podacima jednog programa ne sme imati niko sem njega samog, jer bi se time otvorile brojne mogućnosti za razvoj novog malicioznog softvera koji bi zloupotrebio ovaj bezbednosni propust, bilo napadajući alat za instrumentalizaciju, bilo poruke koje razmenjuje sa instrumentalizovanim programom. Zaštita u vidu šifrovanja bi zahtevala dodatno trošenje resursa, što nije isplativo. Pored bezbednosnog aspekta, bitan faktor je i sinhronizacija. U sistemu sa eksternim alatom, usklađivanje čitanja i pisanja memorijskih segmenata dodeljenih programu bi iziskivalo dodatno trošenje procesorskog vremena i memorije, a i zaključavanje bi povećalo vremensku složenost.

Faza prikupljanja podataka obuhvata: čitanje dodatnih struktura sa metapodacima, njihovo konvertovanje u pogodniji oblik i eksterno skladištenje. Da bi oblik bio pogodan, neophodno je da predstavlja dobar balans između veličine, koja treba biti što manja, i informativnosti, koja treba biti što veća. Ukoliko neki podaci mogu da se izvedu iz ostalih, oni se eliminišu. Lokacija podataka u eksternom skladištu omogućava dodatnu kompresiju bez gubitka na informativnosti. Dovoljno je upisati vrednost željenog podatka, jer je njegovo značenje precizno određeno redosledom upisa bajtova u eksterno skladiste, odnosno položajem bajtova podatka u odnosu na bajtove specijalnih oznaka za razgraničavanje. Ova faza je takođe poverena samom programu, iz istih razloga kao i instrumentalizacija.

Produkt prve dve faze su sirovi podaci, koji u sebi nose informacije o karakteristikama programa u realnim slučajevima upotrebe, ali kako se podaci prikupljaju

samo ako program ima dodatnu funkciju da u toku rada prikuplja i svoje metapodatke, ne može se obezbediti potpuna preciznost informacija. Uticaj se ne može u potpunosti ukloniti, međutim mora biti sveden na granicu prihvatljivosti. Ispravna instrumentalizacija ne sme uticati na funkcionalnost programa.

Poslednja faza predstavlja obradu sirovih podataka do korisne informacije. Krajnji proizvod predstavlja jedan ili više izveštaja u formatu pogodnom prvenstveno za razvojni tim, ne za računar. Osnovne karakteristike izveštaja treba da budu: uniformnost, preglednost, povišena (vraćanje izvedenih podataka) ili snižena informativnost (filtriranje podataka po kategorijama), unija pojedinačnih i statističkih prikaza i slično. Ovu fazu obično obavljaju eksterni alati, jer je potpuno nezavisna od izvršavanja programa i njegove interne memorije. U zavisnosti od toga koje se karakteristike mere i potreba korisnika, krajnji izveštaji variraju od jednorečeničnih ispisa, preko kolekcija fajlova, do interaktivnih aplikacija. Mogu se meriti razne karakteristike, poput na primer memorijskih zahteva ili tragova izvršavanja, ali po informativnosti i mogućnostima kombinovanja sa drugim informacijama, ističe se pokrivenost koda.

Značaj pokrivenosti koda

Pokrivenost koda [8, 10, 21, 12, 15, 18] predstavlja „stepen izvršenosti koda". Izračunava se kao odnos broja izvršenih i neizvršenih linija, blokova, grana ili funkcija i izražava se u procentima. U strogom smislu, pokrivenost koda je jedan jedini broj, dobijen merenjem nad celim sistemom. Taj broj je sam po sebi veoma informativan. Što je pokrivenost manja, to je verovatnoća da u kodu postoje ozbiljne greške u logici veća.

Međutim, nakon merenja na celom skupu, poželjno je izvršiti i merenja na manjim segmentima: komponentama, klasama ili funkcijama, kako bi se detektovali propusti globalne informacije. Na primer, ukoliko je stil pisanja koda takav da se po fajlovima grupišu slični metodi iz različitih klasa, ovakvim pristupom mogu se bolje detektovati slabo ili nimalo korišćene klase, ili objekti koji se prave i uništavaju bez da utiču na ukupnu funkcionalnost. Podaci o izvršavanju konkretnih linija, mogu doprineti pronalasku petlji koje se izvršavaju veliki broj puta, logički neiskorišćenih delova koda ili bespotrebnih grananja koja se svedu na isti krajnji rezultat. Stoga, pokrivenost koda ne treba shvatati samo u svom najužem smislu, već maksimalno iskoristiti sve njene mogućnosti. Uzroci neočekivane pokrivenosti mogu biti veoma raznovrsni. U daljem tekstu biće predstavljeno nekoliko primera.

Stariji softver koji se duže vreme održava, neretko sadrži visok procenat koda iz prethodnih verzija, koji je vremenom izgubio svoju funkcionalnost. Smenom razvojnih timova, naročito u okruženjima koja ne podržavaju detaljno dokumentovanje učinka, često se gube informacije o funkcionalnosti pojedinih delova koda. Usled nedostatka informacija, novi razvijaoci se često ne odlučuju na eliminisanje ili zamenjivanje delova koda, već se uglavnom vrši dodavanje. Funkcije ili klase, a neretko i čitave komponente, tako postaju „mrtav kôd", koji otežava procese održavanja i „debagovanja". Kôd ovakvog softvera ima naročito malu pokrivenost.

Važan faktor prilikom razvoja softvera predstavlja i balans između preciznosti i brzine. Preopterećivanje programa ispitivanjem malo verovatnih alternativnih slučajeva, dovodi do slabljenja performansi. Pored toga, suvišna grananja mogu proizvesti ogromne količine mrtvog koda, od linija pa do čitavih klasa ili komponenti pisanih isključivo za te specijalne slučajeve. To znatno otežava održavanje koda, debagovanje i refaktorisanje. Mala pokrivenost može biti dobar pokazatelj, a podaci izvršavanja linija odrediti preciznije lokaciju problema.

Gotovo sve današnje sisteme odlikuje konkurentno ili paralelno izvršavanje. Njima se postiže značajan porast efikasnosti, ali i povećava broj potencijalnih problema koji mogu nastati prilikom izvršavanja, poput živih i mrtvih zaključavanja, ili trke za resursima. Ovi problem mogu uzrokovati blokiranje ili prestanak rada celog sistema, a njihovo blagovremeno otkrivanje je veoma teško. Algoritam rada procesora određuje koji će se proces, kada i koliko izvršavati, a programer može jedino implementirati neke vidove zaštite atomičnosti operacija ili nametanja prioriteta procesa. Međutim, i pored zaštitnih mehanizama, dešava se da se nekim procesima ne dodeli vreme na procesoru. Takvi kodovi imaju izuzetno niske pokrivenosti, a najbolji pokazatelj su pokrivenosti pojedinačnih izvršavanja koje iznose nula procenata. Prilikom rada sa nitima, niske pokrivenosti mogu biti simptom i preopterećenosti.

Najozbiljniji problemi koji uzrokuju malu pokrivenost su „greške u logici". One mogu varirati, od pogrešno definisanih uslova u granama ili petljama do potpuno promašenih algoritama. Neočekivana pokrivenost je dobar pokazatelj da u kodu ima ovakvih grešaka. Pokrivenost manja od očekivane može, na primer, biti uzrokovana pozivom pogrešnih funkcija, ulaskom u neproduktivnu granu ili prevremenim izlaskom iz programa. Veća pokrivenost od očekivane može biti simptom nepravilnog rada uslova u naredbi grananja, loše konstruisanih provera u kodu i slično. Kako uzroci mogu biti veoma raznovrsni, dobro je pored pokrivenosti celog softvera, meriti i pokrivenosti na segmentima. Kombinovanjem svih rezultata, sužava se oblast

pretrage i lako locira greška u logici.

Potvrda ispravnosti koda pre nego što ode u produkciju, najčešće su samo dobri rezultati testiranja. Međutim, na ishod testova ne utiču samo karakteristike softvera koji se testira, već i njihova ispravnost. Testovi se često sami ne testiraju dovoljno dobro, što može dovesti do ozbiljnih posledica. Lažan negativan rezultat može uzrokovati bespotrebnu potrošnju vremena i novca na traženje nepostojeće greške u kodu. Lažan pozitivan rezultat može imati još i ozbiljnije posledice, čija težina zavisi od važnosti samog softvera. Stoga je veoma korisno primeniti tehniku određivanja pokrivenosti koda i na testove, a ne samo na primarni softver. Mala pokrivenost je dobar indikator da u kodu postoje segmenti koji nisu testirani, a koji su samim tim potencijalna opasnost.

Računanjem pojedinačnih pokrivenosti možemo doći i do informacija o često korišćenim segmentima koda. One umnogome olakšavaju razvojnom timu prilikom donošenja odluka vezanih za vremensku optimizaciju. Kombinovanjem sa podacima za pojedinačne linije koje alociraju memoriju, mogu se pronaći memorijski zahtevni segmenti koji su dobri kandidati za prostornu optimizaciju.

Najsitniji podaci, poput podataka o izvršavanju pojedinih linija ili blokova se mogu koristiti i za refaktorisanje. Uklanjanjanje mrtvog koda ili razbijanje preopterećenih funkcija, su samo neki od primera refaktorišućih procesa koji su olakšani uz informacije o pokrivenosti koda, a čije sprovođenje umnogome pospešuje održavanje ili dalji razvoj.

Raznovrsnost navedenih primera pokayuje veliki značaj i potencijal pokrivenosti koda. Stoga će na nju biti u poptunosti skoncentrisan ostatak ovog rada.

Glava 3

Podrška informisanju o pokrivenosti koda u okviru *GCC*-a

Poznatiji prevodioci, poput *GCC*-a [3], *ICC*-a [5] i *Clang*-a [1] u određenoj meri poseduju ugrađenu podršku određivanju pokrivenosti koda. Projekat LLVM trenutno prednjači u raznovrsnosti, jer pruža i mogućnost informisanja o pokrivenosti u toku izvršavanja. Sa druge strane, autorima programa koji se odluče za programski prevodilac *GCC*, te informacije su dostupne tek nakon izvršavanja programa, što je kompenzovano znatno boljim performasama, pre svega u pogledu memorijske zahtevnosti. Prikupljanje i obrada podataka o pokrivenosti koda u toku izvršavanja programa korišćenjem tehnika *GCC*-a, kombinuje dobru ideju projekta LLVM i dobre tehnike prevodioca *GCC*, čime prednjači i u oblasti mogućnosti i u oblasti performansi. U okviru projekta na kome je utemeljen ovaj rad, izvršena je detaljna analiza postojećih mogućnosti u okviru prevodioca *GCC* i implementirana je podrška za prikupljanje podataka u toku izvršavanja, kao i novi, unapređeni alat za njihov vizuelni prikaz.

3.1 Trenutna implementacija u okviru *GCC*-a

Programski prevodilac *GCC* sadrži ugrađenu podršku određivanju pokrivenosti koda, integrisanu u statičku biblioteku za prikupljanje podataka po imenu *libgcov* i alat za vizuelni prikaz podataka *gcov* [4, 2].

Metapodaci izvršavanja čuvaju se u deljenoj memoriji programa, u listi posebnih, globalno definisanih struktura tipa `gcov_info`, čija se inicijalizacija ugrađuje u binarni kôd prevođenjem sa posebnim flegovima za instrumentalizaciju: `-fprofile-arcs`

`-ftest-coverage`. Flegovi se navode tokom prevođenja izvornog koda do objektnog fajla, a simboli koji se njima unose razrešavaju se kasnije u fazi linkovanja.

Pored ubacivanja instrukcija u binarni kôd programa, prisustvo flegova za instrumentalizaciju uslovljava obavljanje još jedne važne aktivnosti, a to je kreiranje dodatnog fajla, odmah pored njemu odgovarajućeg fajla izvornog koda, sa ekstenzijom *gcno* (*GCov NOtes file*). To je relativno mali, binarni fajl, koji sadrži sve neophodne statičke informacije o strukturi izvornog koda čijim prevođenjem nastaje. Njegova glavna uloga jeste da predstavlja strukturni kostur budućeg finalnog proizvoda alata, koji će se kasnije nadograditi podacima dobijenim dinamički u toku izvršavanja. Format fajla *gcno* je utvrđen zajedničkim standardom *GCC*-a i alata *gcov*, koji je specijalizovan i za njegovo tumačenje. Redukcija veličine je postignuta maksimalnim stepenom kompresije podataka. Korišćenje specijalnih oznaka, korišćenje pozicije kao interpretacije podatka, kao i pažljivo odabrani minimalni skup potrebnih informacija o strukturi, samo su neke od tehnika kompresije korišćenih u cilju maksimalne štednje memorije. Posebno je važno napomenuti da je čuvanje podataka o strukturi u vidu eksternih binarnih fajlova osnovni uzrok boljih memorijskih performansi *GCC* instrumentalizacije u odnosu na *Clang*-ovo profajliranje, pomenute na kraju prethodnog poglavlja, jer umanjuje rizik od eksplozije veličine samog programa. Uvećanje izvršnog fajla do veće vrednosti od memorijske količine koja je za njega predviđena na sistemu, može dovesti do nepravilnosti u radu, a kako memorijski zahtevniji instrumentlizovani program se ponaša drugačije od regularnog, rezultati testiranja neće odražavati realno stanje. Naročito, na sistemima sa veoma ograničenim memorijskim prostorom, *GCC* instrumentalizacija je jedina moguća.

Svaka struktura tipa `gcov_info` iz liste, odgovara tačno jednom instrumentalizovanom objektnom fajlu koji učestvuje u izgradnji programa. Pored osnovnih podataka poput imena fajla ili verzije alata, svaka struktura tipa `gcov_info` sadrži i pokazivač na niz struktura tipa `gcov_fn_info`, u kojima se skladišti po nekoliko posebnih brojača za svaku funkciju tog fajla. Na osnovu vrednosti u njima, može se konstruisati podatak o količini izvršavanja bilo koje jedinice koda u okviru te funkcije. Tokom rada programa, vrednosti u brojačima se konstantno ažuriraju, i u svakom trenutku odražavaju realno stanje izvršavanja. Ti podaci predstavljaju jezgro informacije o pokrivenosti koda, ali njih eksterni alat poput *gcov*-a ne može direktno koristiti iz više razloga. Prvi razlog je bezbednosne prirode, i velikom merom je obrazložen u prethodnom poglavlju. Eksternim alatima se ni u kom slučaju

ne treba omogućiti čitanje internih podataka programa. Detaljno je obrazloženo i pitanje sinhronizacije pisanja i čitanja, koje važi i u ovom slučaju. Naposletku, ovim pristupom bi podaci imali poreklo samo iz jednog izvršavanja, što bespotrebno ograničava mogućnosti alata.

Rešenje koje je trenutno implementirano u *GCC*-u, upravo iz tih razloga, sadrži jednog „posrednika" između instrumentalizovanog programa i eksternih alata, a to je *libgcov*. Biblioteka, po svojoj prirodi, se ugrađuje u program i time postaje deo njega, što joj daje ekskluzivno pravo pristupa njegovoj deljenoj memoriji. Njen osnovni zadatak je ekstrakcija podataka iz strukture `gcov_info` i njihovo konvertovanje u oblik pogodan za obradu eksternim alatom. Statička funkcija `gcov_at_exit` preuzima vrednosti brojača, računa sumarne i statističke podatke i sve zajedno upisuje u posebni binarni fajl sa ekstenzijom *gcda* (*GCov DAta file*) u unapred utvrđenom formatu i na unapred utvrđenoj lokaciji. Optimalnost veličine se i ovde postiže primenom sličnih tehnika za kompresiju, kao u slučaju fajla *gcno*. Generiše se uvek pored objetktnog fajla kome odgovara, u slučaju da fajl sa istim imenom i ekstenzijom već ne postoji na toj lokaciji. U slučaju višestrukog pokretanja programa, vrednosti iz predhodnih izvršavanja već se nalaze u fajlu *gcda*, te se on samo ažurira, a za sumiranje starih i novih podataka, zadužena je druga funkcija po imenu `__gcov_merge_add_`. Stoga, za zanemarivanje starih podataka, neophodno je premestiti ili ukloniti prethodni fajl *gcda* pre novog pokretanja.

Na kraju izvršavanja instrumentalizovanog programa, svi podaci potrebni za informisanje razvojnog tima o pokrivenosti njihovog koda, nalaze se na fajl sistemu i mogu se pakovati, premeštati i skladištiti. To je veoma korisna činjenica, jer pruža nove mogućnosti kombinovanja rezultata različitih merenja. Ukoliko postoji potreba da se neki test prekine na određeno vreme i započne novi, fajlovi *gcda* prvog testa se mogu spakovati na drugu lokaciju, čime će se za drugi test generisati novi, i ponovo prebaciti pored objektnih pred nastavak prvog testa. Za ekonomično skladištenje mogu se koristiti i kompresovane arhive ili eksterni memorijski mediji. Međutim, njihova osnovna funkcija je da predstavljaju ulazne parametre za alat *gcov*, koji na osnovu njih kreira tekstualni izveštaj, pogodniji za interpretaciju od strane razvojnog tima.

Za generisanje jednog izveštaja, potrebno je alatu *gcov* proslediti u vidu argumentata: jedan izvorni fajl, jedan odgovarajući strukturni fajl: *gcno* i jedan fajl sa vrednostima brojača: *gcda*. Poseban tekstualni fajl sa ekstenzijom *gcov* se kreira za svaki instrumentalizovani fajl izvornog koda. Izveštaj se sastoji od celokupnog

sadržaja izvornog koda, uz dodatak jedne vrednosti ispred svake izvršne linije, koja predstavlja broj puta koliko se ta linija izvršavala. Ukoliko se linija nije izvršila nijednom, ispred nje se stavlja posebna oznaka sastavljana od pet simbola tarabice. Prvih nekoliko linija izveštaja rezervisano je za statističke podatke o imenima fajlova od kojih je kreiran, dok se na standardni izlaz štampa najvažnija vrednost: odnos broja izvršenih linija i ukupnog broja linija, odnosno pokrivenost koda. Na slici 3.1, prikazan je primer osnovnog izveštaja, koji se generiše pozivom alata *gcov* bez dodatnih opcija. Korišćenjem flegova u pozivu alata, izveštaj se može unaprediti i podacima o blokovima, granama, funkcijama i slično.

3.2 Kritika trenutne implementacije u okviru *GCC*-a

Proučavanjem karakteristika implementacije instrumentalizacije u okviru *GCC*-a i mogućnosti određivanja pokrivenosti koda standardnim *GNU*-ovim alatom: *gcov*-om na osnovu te instrumentalizacije, otkrivena su tri velika nedostatka:

1. Podaci su dostupni tek po završetku rada programa
2. Korišćenje neoptimalnog statičkog linkovanja
3. Korisnički interfejs standardnog *GNU*-ovog alata ne pruža jednostavan i intuitivan pregled podataka

U narednom tekstu biće detaljnije objašnjeni ovi nedostaci.

Prikupljanje podataka na kraju izvršavanja

Čitanje podataka iz deljene memorije programa i njihovo skladištenje u fajlove *gcda*, odvija se kao poslednja instrukcija programa pre kraja izvršavanja (`at_exit`). Usled toga, iako analiza *gcov* alatom nije striktno vezana za vremenski tok izvršavanja, ne može se vršiti pre kraja programa. Ovo je veliki nedostatak, koji u nekim specifičnim slučajevima može poptuno onemogućiti proveravanje pokrivenosti koda. Programi kod kojih je vreme rada izuzetno dugo ili su podaci dostupni i/ili korisni samo tokom rada, poput sistema za rad u realnom vremenu, servera ili operativnih sistema, ne mogu koristiti instrumentalizaciju na kraju izvršavanja. Ukoliko

```
    -:    0:Source:1.c
    -:    0:Graph:1.gcno
    -:    0:Data:1.gcda
    -:    0:Runs:1
    -:    0:Programs:1
    -:    1:#include <stdio.h>
    -:    2:
    1:    3:void pozdrav(){
    -:    4:
    1:    5:    printf("Dobar dan! Dobrodosli u test program!\n");
    -:    6:
    1:    7:}
    -:    8:
    1:    9:int oprostaj(){
    -:   10:
    1:   11:    printf("Dovidjenja! Ugodan dan!\n");
    -:   12:
    1:   13:}
    -:   14:
    1:   15:int main(){
    -:   16:
    1:   17:    pozdrav();
    -:   18:
    1:   19:    int a = 1;
    1:   20:    int b = 2;
    -:   21:
    1:   22:    if(a==b){
#####:   23:        printf("Netacno: 1=2\n");
    -:   24:    }
    -:   25:    else{
    1:   26:        printf("Tacno: 1!=2\n");
    -:   27:    }
    -:   28:
    1:   29:    oprostaj();
    -:   30:
    1:   31:    return 0;
    -:   32:
    -:   33:}
```

Slika 3.1: Osnovni izveštaj koji generiše *gcov*

imaju ograničene memorijske mogućnosti, što je često slučaj na ovakvim sistemima, ne mogu koristiti ni instrumentalizaciju programskog prevodioca projekta LLVM. Za obezbeđivanje informacija o pokrivenosti koda ovakvih programa, neophodno je proširiti mogućnosti instrumentalizacije *GCC*-a na prikupljanje podataka u toku izvršavanja.

Statičko linkovanje

Biblioteka *libgcov* je u okviru *GCC*-a implementirana kao statička biblioteka (arhiv). Upotreba statičkih biblioteka nije optimalno rešenje [9], ni prostorno, ni

vremenski. Prilikom linkovanja njeni podaci se kopiraju u program. Ukoliko istu biblioteku koristi više programa, kopiranje će se izvršiti u svaki posebno. Kako se simboli ne razlikuju od programa do programa, ponavljanje je redudantno, što znači da se na ovaj način troši mnogo više memorije nego što je suštinski potrebno. Promene u biblioteci zahtevaju ponovno prevođenje ne samo biblioteke, već i svakog instrumentalizovanog programa ponaosob, što može, zavisno od veličine sistema, predstavljati veliki vremenski utrošak. U sistemu sa neograničenim resursima, prostornim i vremenskim, statički pristup bi predstavljao dovoljno dobro rešenje za instrumentalizaciju bilo kog skupa programa/biblioteka koji taj sistem čini. Međutim, realni sistemi imaju često veoma oštra ograničenja resursa, te je za instrumentalizaciju većeg broja programa/biblioteka potrebno optimizovati sam proces instrumentalizacije, a kao dobra ideja nameće se upotreba dinamičkog linkovanja.

Korisnički interfejs

Prikaz u vidu pojedinačnih izveštaja za svaki fajl izvornog koda, takođe poseduje određene mane. Svaki izveštaj se nalazi na posebnoj lokaciji u okviru direktorijuma projekta, što otežava njihov pregled kao celine. Dodatne informacije, poput onih o pokrivenosti pojedinačnih funkcija, koje se dobijaju dodavanjem opcija u poziv alata, kao i vrednost pokrivenosti fajla, se ne nalaze u okviru izveštaja, već samo ispisa na standardni izlaz, što uzrokuje potencijalni gubitak tih informacija. Vrednost pokrivenosti koda čitavog projekta se ne izračunava, čime je krajnji rezultat oslabljen za još jednu bitnu informaciju. Potreba za prevazilaženjem ovih mana je uticala na formiranje ideje o novom alatu za vizuelni prikaz *gcov* statistike, koji je izgrađen u okviru ovog projekta.

3.3 Unapređenje programskog prevodioca *GCC*

Unapređenje koje je neophodno za korigovanje prethodno navedenih nedostataka može se podeliti u dva odvojena problema:

1. Unapređenje prikupljanja podataka (poboljšanje *backend* podrške)
2. Unapređenje prikaza podataka (poboljšanje *frontend* podrške)

Unapređenje prikupljanja podataka

Cilj ovog unapređenja jeste omogućiti optimalno prikupljanje podataka iz izvršavanja istrumentalizovanog programa u bilo kom trenutku između početka i kraja programa. U toku projektovanja, bilo je potrebno doneti više važnih odluka koje su značajno odredile tok samog razvoja. Kako je odgovornost za celokupni proces prikupljanja, obrade i prikaza podataka o pokrivenosti koda prevedenog *GCC*-om podeljena između alata *gcov* i biblioteke *libgcov*, prva odluka koju je bilo potrebno doneti jeste odabir materijala za prilagođavanje između ove dve komponente.

Odabir komponente za prilagođavanje

Prvobitno je razmatrano rešenje, koje se u ranim fazama projektovanja, linearnom kritičkom razmišljanju nametalo kao očigledno i jednostavno: prilagođavanje alata. Osnovna ideja predstavlja promenu jednog dela ulaznih podataka *gcov*-a, čime bi se vrednosti iz izvršavanja preuzimale direktno iz instrumentalizacionih struktura u deljenoj memoriji programa, umesto iz fajlova *gcda*. Analizom neophodnih izmena za ostvarivanje proširenja mogućnosti na ovaj način, izveden je zaključak da ovo rešenje vodi ka veoma komplikovanoj implementaciji, značajnom padu performansi, kao i ugrožavanju bezbednosti podataka instrumentalizovanog programa. Promena formata ulaznih podataka, iziskivala bi velike algoritamske promene u okviru koda *gcov* programa, čime bi se složenost implementacije gotovo izjednačila sa kreiranjem novog alata. Pristup deljenoj memoriji instrumentalizovanog programa bi predstavljao kritičnu sekciju, usled potencijanog istovremenog upisivanja podataka od strane programa i čitanja istih tih podataka od strane *gcov* alata, zbog čega bi bilo potrebno implementirati određenu vrstu zaštite u vidu zaključavanja ili semafora. Implementacija bi se time dodatno iskompikovala, a performanse, pre svega vremenska složenost, bi značajno opale. Naročito je problematično ugrožavanje performansi instrumentalizovanog programa, jer u cilju pružanja ispravnih informacija, korektna instrumentalizacija mora imati minimalni uticaj na tok i vreme izvršavanja. Bezbednost podataka bi zavisila od kvaliteta implementacije kao i od mogućnosti sistema ukoliko bi bila odabrana naprednija vrsta zaštite. Istrumentalizacione strukture u deljenoj memoriji, po svojoj prirodi su vezane isključivo za trenutno izvršavanje. Stoga bi ovaj pristup takođe ograničio mogućnosti prikupljanja na podatke iz poslednjeg izvršavanja programa.

Detaljna analiza gore navedenog pristupa, kao i postojeće logike instrumenta-

lizacije implementirane u programskom prevodiocu *GCC*, dovela je do formiranja znatno boljeg rešenja. U trenutnoj implementaciji, celokupni posao prikupljanja podataka prepušten je biblioteci *libgcov*. Promenom trenutka kreiranja fajlova *gcda*, postigao bi se željeni rezultat bez uvođenja dodatnih izazova poput bezbednosti ili algoritamskih promena alata. Detaljnijom analizom ustanovljeno je da je vremenska određenost trenutka izbacivanja rezultata posledica potpune kontrole biblioteke nad instrumentalizacijom, odnosno zatvorenošću interfejsa biblioteke prema potencijalnim korisnicima. Celokupna funkcionalnost je definisana tako da se odvija bez posredovanja vlasnika instrumentalizovanog programa. Ukoliko bi kontola poziva funkcije za generisanje fajlova *gcda* bila prepuštena korisniku biblioteke, prikupljanje podataka bi bilo moguće u bilo kom trenutku. Ovo rešenje je jednostavno za implementaciju, optimalno je, bezbedno i pruža mogućnost kombinovanja rezultata iz više izvršavanja bez dodatnih modifikacija alata za generisanje izveštaja. Četiri navedene prednosti su presudile odabir komponente za prilagođavanje u korist biblioteke *libgcov*.

Nova biblioteka, dinamička i nezavisna

Implementiranje podrške za prikupljanje podataka u toku izvršavanja u vidu biblioteke, otvorilo je mogućnost optimizacije performansi i održavanja „u hodu", ukidanjem zavisnosti od programskog prevodioca i prelaskom na dinamičko linkovanje. Osnovna ideja predstavlja zamenu statičke biblioteke *libgcov* njenim dinamičkim, funkcionalnim pandanom, nezavisnim od infrastrukture *GCC*-a.

Dinamičko linkovanje znatno poboljšava vremensku i prostornu složenost [9]. Operativni sistem može smestiti kôd dinamičke biblioteke u segmente *ROM*-a koje deli više procesa, čime se omogućava jedinstvenost koda u okviru memorije. Time se prostorna složenost sa linerane, svodi na konstantnu vrednost količine memorije potrebne za smeštanje jedne biblioteke, ukidanjem zavisnosti složenosti od broja procesa. Instrukcije biblioteke se ne kopiraju u izvršnu verziju, čime se smanjuje i potreban prostor za skladištenje instrumentalizovanih programa. Dodatne tehnike poput tabela indirekcije i lenjog povezivanja simbola omogućavaju i vremensku uštedu. Sa druge strane, upotreba deljene biblioteke olakšava i procese njene implementacije, testiranja i održavanja. Otklanjanje greške u kodu biblioteke ili potencijana kasnija nadogradnja njenih mogućnosti, ne uslovljavaju ponovno prevođenje svih instrumentalizovanih programa. Prevođenje većih sistema iziskuje dosta vremena, pa ova ušteda pravi značajnu razliku.

Ukidanje zavisnosi biblioteke za instrumentalizaju od programskog prevodioca omogućava dodatne olakšice kasnijem održavanju, jer nije potrebno ponovno prevoditi celokupni *GCC* nakon svake izmene u kodu biblioteke, a i promena verzije prevodioca ne iziskuje promene u instrumentalizaciji. Ova izmena nema negativan uticaj na performanse, jer se zamenom čuva ukupni skup simbola i instrukcija.

Novi interfejs biblioteke

Korišćenjem nove biblioteke, odgovornost nad pozivom funkcije za ispisivanje podataka u fajlove *gcda* je prebačena na instrumentalizovani program. To je prirodni preduslov pružanja mogućnosti korisniku da sam odabere trenutak u kojem se ta funkcionalnost vrši.

Osnovni pristup korišćenja podrazumeva definisanje glavne funkcije biblioteke kao eksterne i njen poziv u okviru koda instrumentalizovanog programa.

Za korisnike *Unix*-a i *Unix*-olikih operativnih sistema, implementirana je jedna dodatna pogodnost. Biblioteka omogućava ispisivanje podataka u fajlove *gcda* pomoću signala `SIGUSR1`. Registracija signala se izvršava u okviru biblioteke na početku izvršavanja. Time se postiže da korisnik nema obavezu da svoj kôd prilagođava instrumentalizaciji, već može u željenom trenutku, iz terminala poslati signal komandom „kill -10 PID”, gde je PID broj koji predstavlja jedinstveni identifikator procesa. Ukoliko korisnik ne može precizno odrediti pravi trenutak na taj način, na raspolaganju mu je i dalje osnovni pristup. Korišćenje signala na *Windows* operativnom sistemu nije podržano, zbog čega je u ovom slučaju neophodno koristiti osnovni pristup. Isti princip važi i ukoliko korisnički program predefiniše signal `SIGUSR1`.

Unapređenje prikaza podataka

Cilj ovog unapređenja jeste omogućiti jednostavan, intuitivan, vizuelni prikaz podataka iz izvršavanja istrumentalizovanog programa. `Gcov` izveštaji i statistički podaci se prikazuju odvojeno od kodova, binarnih i izvršnih fajlova, kako bi se olakšalo i ubrzalo pronalaženje i pregledanje. Prikaz u vidu drveta putanja omogućava brz i efikasan pregled, bez narušavanja modularnosti projekta. Pored osnovnih *gcov* izveštaja, u okviru drveta su dostupni i izveštaji koji sadrže statistiku po funkcijama.

U okviru novog alata za vizuelni prikaz podataka o pokrivenosti koda, pored unapređenja pregleda, implementirano je i nekoliko novih funkcionalnosti. Neki od važnih podataka, poput ukupne pokrivenosti projekta, modula ili programa, ne iz-

računavaju se pozivom postojećeg alata, zbog njegove ograničenosti na pojedinačni fajl izvornog koda. *Gcov* se može pozvati i sa više argumenta, ali se svaki obrađuje pojedinačno. Alat kreiran u okviru ovog projekta, omogućava i prikaz ukupne statistike. Generisanje *gcov* izveštaja za sve izvorne fajlove projekta je dosta olakšano. Umesto višestrukih poziva alata i pozicionaranja u okviru direktorijuma projekta, celokupna funkcionalnost se izvršava jednim klikom.

Za korisnike *Unix*-a i *Unix*-olikih operativnih sistema, novi vizuelni alat pruža mogućnost i prikupljanja podataka, odnosno slanja signala `SIGUSR1` željenom instrumentalizovanom programu. Time je ukupna funkcionalnost instrumentalizacije dostupna u okviru jednog grafičkog korisničkog interfejsa. Kao što je već napomenuto, korišćenje signala na *Windows* operativnom sistemu nije podržano, zbog čega je u ovom slučaju za korišćenje alata preduslov imati već kreirane fajlove *gcda*. Isti princip važi i ukoliko korisnički program predefiniše signal `SIGUSR1`.

Glava 4

Implementacija i analiza

4.1 Implementacija

1. Biblioteka
2. GUI (signali za prikupljanje podataka, generisanje izvestaja)

4.2 Demonstracija i uputstvo za upotrebu

1. primer rada biblioteke I GUI-ja sa slikama
2. dobar moment da se naglasi da rad ima primenu na bilo koji kod
3. ne znam jel smem pominjati digitalnu i ko ga sad koristi

4.3 Performanse

1. da li smo postigli cilj
2. da li možemo isto što i pre, pa i više
3. memorija I bezbednost – test sa Valgrindom
4. složenost – vremenska i prostorna
5. jednostavnost upoterebe
6. ne bi bilo loše ovde pomenuti LLVM i njihovu runtime instrumentalizaciju

4.4 Primena

1. Gde bi sve ovo moglo da radi
2. Ne znam koliko smem odavati na čemu je testirano I na čemu radi
3. Ideja: Ako bi se ovakav jedan alat unapredio I ugradio npr u pejsmejker da signalizira da nešto ne radi kako treba, to što je runtime prikupljanje moglo bi nekome spasiti život

Glava 5

Zaključak

1. Šta je urađeno
2. Koji je značaj toga što je urađeno (gde sad radi – onliko koliko smem da kazem)
3. Šta bi još moglo da se uradi:
    a) Ideja: Ako bi se ovakav jedan alat unapredio I ugradio npr u pejsmejker da signalizira da nešto ne radi kako treba, to što je runtime prikupljanje moglo bi nekome spasiti život
    b) Moze mala komparacija sa LLVMom – tipa da se analizira sta je dobro i da se malo unapredi po ugledu na LLVM

Bibliografija

[1] Clang: a c language family frontend for llvm. https://clang.llvm.org/.

[2] Code coverage using gcov. https://web.archive.org/web/20140409083331/http://xview.net/pape

[3] Gcc, the gnu compiler collection. https://gcc.gnu.org.

[4] Gcov official site. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[5] Intel® parallel studio xe 2018: Getting started with the intel® c++ compiler 18.0 for linux*. https://software.intel.com/en-us/get-started-with-cpp-compiler-18.0-for-linux-parallel-studio-xe-2018.

[6] Razvoj softvera - materijali sa predavanja. http://poincare.matf.bg.ac.rs/ smalkov/files/rs.r290.2018/public/Predavanja/Razvoj

[7] Source code instrumentation overview. https://www.ibm.com/support/knowledgecenter/SSSHU

[8] Paul Ammann and Jeff Offutt. Introduction to software testing. *Cambridge University Press*, 2016.

[9] David M Beazley, Brian D Ward, and Ian R Cooke. The inside story on shared libraries and dynamic loading. *Computing in Science & Engineering*, 3(5):90–97, 2001.

[10] R Brader, H Hilliker, and A Wills. Unit Testing: Testing the Inside. *Microsoft Developer Guidance*, 2013.

[11] Ozren Demonja, Stefan Maksimović, and Marko Crnobrnja. Apstraktna interpretacija. http://poincare.matf.bg.ac.rs/ milena/msnr/2017/12/09_

[12] A Glower. In pursuit of code quality: Don't be fooled by the coverage report. *IBM Developer Works blog post*, 2006.

[13] V. Gupta. Measurement of Dynamic Metrics Using Dynamic Analysis of Programs. *APPLIED COMPUTING CONFERENCE (ACC '08), Istanbul, Turkey*, 2008.

[14] A. Homescu. Profile-guided automated software diversity. *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE Computer Society*, 2013.

[15] B Marick. How to misuse code coverage. *Proceedings of the 16th Interational Conference on Testing Computer Software*, 1999.

[16] F Nielson, H. R. Nielson, and C Hankin. Principles of program analysis. *Springer Science & Business Media*, 2015.

[17] Ljubica Peleksić and Milica Kojičić. Simboličko izvršavanje. http://webmail.matf.bg.ac.rs/ milena/msnr/2016/12/11_VerifikacijaSoftvera_PeleksicKojicic.pdf.

[18] A Piziali. "Code coverage," in Functional verification coverage measurement and analysis. *Springer Science & Business Media*, 2007.

[19] Nikola Vlahovic, Mišić Petar, and Muljaić Aleksandar. Proveravanje modela. http://poincare.matf.bg.ac.rs/ milena/msnr/2017/10/03_ProveravanjeModelaVlahovicMisicMuljaic.pdf.

[20] Milena Vujošević Janičić. Verifikacija softvera. http://www.programskijezici.matf.bg.ac.rs/vs/predavanja/03_dinamicka_analiza/03_dinamicl

[21] L William, B Smith, and S Heckman. Test Coverage with EclEmma. *Technical Report Raleigh*, 2008.

Biografija autora

Marina Nikolić (*Sombor, 17. decembar 1992.*) je ...