

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

Sadržaj	v
1 Uvod	1
2 Analiza performansi programa	2
2.1 Vrste analize programa	2
2.2 Profajliranje programa	6
3 Podrška informisanju o pokrivenosti koda u okviru <i>GCC</i>-a	11
3.1 Trenutna implementacija u okviru <i>GCC</i> -a	11
3.2 Kritika trenutne implementacije u okviru <i>GCC</i> -a	15
3.3 Unapređenje programskog prevodioca <i>GCC</i>	19
4 Implementacija	23
4.1 Implementacija biblioteke	23
4.2 Implementacija grafičkog alata	29
5 Analiza korektnosti i performasi	35
5.1 Primena	35
6 Zaključak	36
Bibliografija	37

Glava 1

Uvod

1. kratak opis o čemu će biti reči u daljem tekstu
2. iako vidim da je popularno po master radovima da se piše po poglavljima ovde (tipa, u poglavlju X je opisano to i to), ja bih uvod radije sročila kao priču 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 slučajevima 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 [18] 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 potvrdom ispravnosti ili performansi, već isključivo tretirati kao smernice pri razvoju.

Postoje i određene statičke metode koje su značajno preciznije od metrika, poput simboličkog izvršavanja [9], proveravanja modela [12] ili apstraktne interpretacije [13]. 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 [15] 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 [16, 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 „debugovanja”, 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. Vršiti 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 predstavljaju 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 skladište, 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, 11, 21, 14, 17, 19] 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 razvijaoči 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 „debugovanja”. 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, debugovanje i refaktorisiranje. 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 bespotrebnju 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 refaktorisiranje. Uklanjanje 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 pokazuje veliki značaj i potencijal pokrivenosti koda. Stoga će na nju biti u potpunosti 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 performansama, 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`

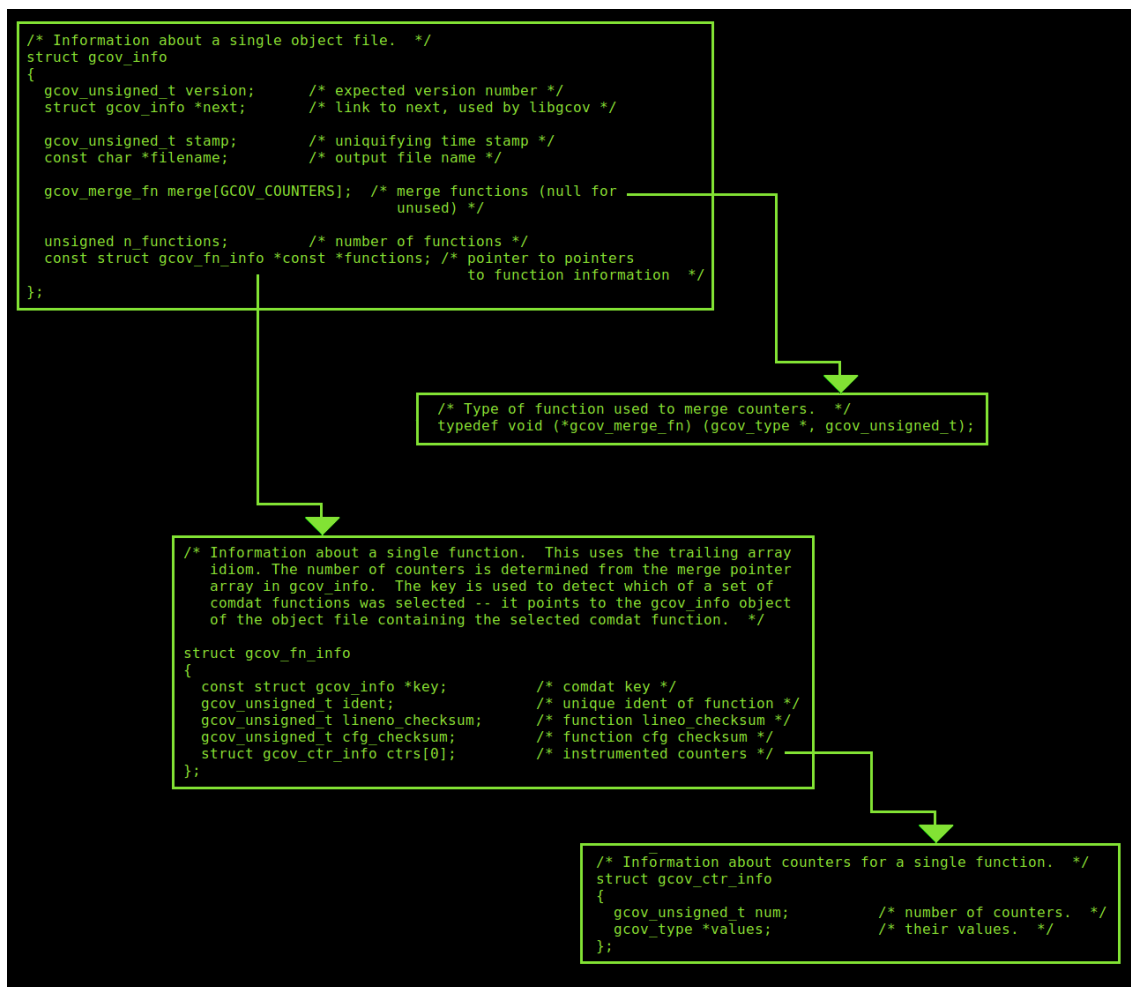
`-ftest-coverage`. Flegovi se navode tokom prevođenja izvornog koda do objektnog fajla, a simboli koji se njima unose razrešavaju se kasnije u fazi linkovanja.

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 objektnog fajla, 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.

Definicije i međusobne veze najvažnijih instrumenatalizacionih struktura prikazani na slici 3.1. 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 pri-

GLAVA 3. PODRŠKA INFORMISANJU O POKRIVENOSTI KODA U OKVIRU GCC-A

rode, 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. Naposljetku, ovim pristupom bi podaci imali poreklo samo iz jednog izvršavanja, što bespotrebno ograničava mogućnosti alata.



Slika 3.1: Instrumentalizacione strukture i njihove međusobne veze

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

GLAVA 3. PODRŠKA INFORMISANJU O POKRIVENOSTI KODA U OKVIRU GCC-A

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 objektnog 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 sastavljena 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 opcija u pozivu alata, izveštaj se može unaprediti i podacima o blokovima, kao na slici 3.1, granama, kao na slici 3.1 i slično.

GLAVA 3. PODRŠKA INFORMISANJU O POKRIVENOSTI KODA U OKVIRU GCC-A

```
-: 0:Source:sample.c
-: 0:Graph:sample.gcno
-: 0:Data:sample.gcda
-: 0:Runs:3
-: 0:Programs:1
-: 1:#include <stdio.h>
-: 2:
3: 3:void greeting(){
3: 4:     printf("Hello, I am a program that prints out Pascal's triangle...\n\n");
3: 5:}
3: 6:void farewell(){
3: 7:     printf("\nGoodbye...\n");
3: 8:}
3: 9:int main(int argc, char** argv){
3: 10:     greeting();
3: 11:     if(argc>1){
2: 12:         int size = atoi(argv[1]);
2: 13:         int value = 1, dist=0, i=0, j=0;
17: 14:         for(i=0; i<size; i++){
85: 15:             for(dist=1; dist <= size-i; dist++){
70: 16:                 printf(" ");
85: 17:                 for(j=0; j<=i; j++){
70: 18:                     if (j==0 || i==0)
15: 19:                         value = 1;
-: 20:                     else
55: 21:                         value = value*(i-j+1)/j;
70: 22:                     printf("%2d", value);
-: 23:                 }
15: 24:                 printf("\n");
-: 25:             }
-: 26:         }
-: 27:     else
1: 28:         printf("Please start the program with desired number of rows\n");
3: 29:     farewell();
3: 30:     return 0;
-: 31:}
```

Slika 3.2: Osnovni izveštaj koji generiše *gcov*

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.

GLAVA 3. PODRŠKA INFORMISANJU O POKRIVENOSTI KODA U OKVIRU GCC-A

```
 -: 0:Source:sample.c
 -: 0:Graph:sample.gcno
 -: 0:Data:sample.gcd
 -: 0:Runs:3
 -: 0:Programs:1
 -: 1:#include <stdio.h>
 -: 2:
function greeting called 3 returned 100% blocks executed 100%
 3: 3:void greeting(){
 3: 4: printf("Hello, I am a program that prints out Pascal's triangle...\n\n");
call 0 returned 100%
 3: 5:}
function farewell called 3 returned 100% blocks executed 100%
 3: 6:void farewell(){
 3: 7: printf("\nGoodbye...\n");
call 0 returned 100%
 3: 8:}
function main called 3 returned 100% blocks executed 100%
 3: 9:int main(int argc, char** argv){
 3: 10: greeting();
call 0 returned 100%
 3: 11: if(argc>1){
branch 0 taken 67% (fallthrough)
branch 1 taken 33%
 2: 12: int size = atoi(argv[1]);
call 0 returned 100%
 2: 13: int value = 1, dist=0, i=0, j=0;
17: 14: for(i=0; i<size; i++){
branch 0 taken 88%
branch 1 taken 12% (fallthrough)
85: 15: for(dist=1; dist <= size-i; dist++)
branch 0 taken 82%
branch 1 taken 18% (fallthrough)
70: 16: printf(" ");
call 0 returned 100%
85: 17: for(j=0; j<=i; j++){
branch 0 taken 82%
branch 1 taken 18% (fallthrough)
70: 18: if (j==0 || i==0)
branch 0 taken 79% (fallthrough)
branch 1 taken 21%
branch 2 taken 0% (fallthrough)
branch 3 taken 100%
15: 19: value = 1;
 -: 20: else
55: 21: value = value*(i-j+1)/j;
70: 22: printf("%2d", value);
call 0 returned 100%
 -: 23: }
15: 24: printf("\n");
call 0 returned 100%
 -: 25: }
 -: 26: }
 -: 27: else
1: 28: printf("Please start the program with desired number of rows\n");
call 0 returned 100%
 3: 29: farewell();
call 0 returned 100%
 3: 30: return 0;
 -: 31:}
```

Slika 3.3: Izveštaj koji generiše alat *gcov* sa opcijom *-b*

Prikupljanje podataka na kraju izvršavanja

Čitanje podataka iz deljene memorije programa i njihovo skladištenje u fajlove *gcd*a, odvija se kao poslednja instrukcija programa pre kraja izvršavanja (*at_exit*).

GLAVA 3. PODRŠKA INFORMISANJU O POKRIVENOSTI KODA U OKVIRU GCC-A

```
-: 0:Source:sample.c
-: 0:Graph:sample.gcno
-: 0:Data:sample.gcda
-: 0:Runs:3
-: 0:Programs:1
-: 1:#include <stdio.h>
-: 2:
3: 3:void greeting(){
3: 3-block 0
3: 4: printf("Hello, I am a program that prints out Pascal's triangle...\n\n");
3: 4-block 0
3: 5:}
3: 6:void farewell(){
3: 6-block 0
3: 7: printf("\nGoodbye...\n");
3: 7-block 0
3: 8:}
3: 9:int main(int argc, char** argv){
3: 9-block 0
3: 10: greeting();
3: 10-block 0
3: 11: if(argc>1){
2: 12:     int size = atoi(argv[1]);
2: 12-block 0
2: 13:     int value = 1, dist=0, i=0, j=0;
17: 14:     for(i=0; i<size; i++){
17: 14-block 0
2: 14-block 1
85: 15:         for(dist=1; dist <= size-i; dist++)
15: 15-block 0
85: 15-block 1
70: 16:             printf(" ");
70: 16-block 0
85: 17:             for(j=0; j<=i; j++){
15: 17-block 0
85: 17-block 1
70: 18:                 if (j==0 || i==0)
70: 18-block 0
55: 18-block 1
15: 19:                     value = 1;
15: 19-block 0
-: 20:                     else
55: 21:                         value = value*(i-j+1)/j;
55: 21-block 0
70: 22:                         printf("%2d", value);
70: 22-block 0
-: 23:                     }
15: 24:                 printf("\n");
15: 24-block 0
-: 25:             }
-: 26:         }
-: 27:     else
1: 28:         printf("Please start the program with desired number of rows\n");
1: 28-block 0
3: 29:     farewell();
3: 29-block 0
3: 30:     return 0;
-: 31:}
```

Slika 3.4: Izveštaj koji generiše alat *gcov* sa opcijom *-a*

GLAVA 3. PODRŠKA INFORMISANJU O POKRIVENOSTI KODA U OKVIRU GCC-A

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 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 [10], 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 redundantno, 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 instrumentalizovanog 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

GLAVA 3. PODRŠKA INFORMISANJU O POKRIVENOSTI KODA U OKVIRU GCC-A

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. Instrumentalizacione 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 instrumentalizacije 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”, uklanjanjem zavisnosti od programskog prevodioca i prelaskom na dinamičko linkovanje.

GLAVA 3. PODRŠKA INFORMISANJU O POKRIVENOSTI KODA U OKVIRU GCC-A

Osnovna ideja predstavlja zamenu statičke biblioteke *libgccov* njenim dinamičkim, funkcionalnim pandanom, nezavisnim od infrastrukture *GCC*-a.

Dinamičko linkovanje znatno poboljšava vremensku i prostornu složenost [10]. 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 zavisnosti biblioteke za instrumentalizaciju 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 instrumentalizovanog 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 izračunavaju 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 pozicioniranja 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

Implementacija unapređenja prikupljanja i prikaza podataka iz izvršavanja programa prevedenog programskim prevodiocem *gcc* je podeljena u dve zasebne, implementaciono nezavisne celine:

1. Implementacija dinamičke biblioteke, nezavisne od *gcc*-a, čija osnovna funkcionalnost predstavlja prikupljanje podataka iz izvršavanja programa pre završetka njegovog rada.
2. Implementacija novog grafičkog alata za prikaz podataka, u kome je bezbedna i iskustvom validirana osnova, alat *gcov*, nadograđena jednostavnim, preglednim, korisnički prilagođenom interfejsom sa povišenom informativnošću.

U narednom tekstu biće detaljno opisane implementacije obe celine.

4.1 Implementacija biblioteke

Implementacija biblioteke *libcoverage* je izvršena u programskom jeziku C, po uzoru na postojeću biblioteku *libgcov*, koja predstavlja podrazumevanu biblioteku za prikupljanje podataka iz izvršavanja programa u okviru programskog prevodioca *gcc*. Osnovna funkcionalnost biblioteke *libgcov* se vrši u okviru funkcije: `gcov_exit`. Njen algoritam se sastoji iz iterativnih i dubinskih prolazaka kroz instrumentalizacije strukture (prikazane na slici 3.1) u cilju prikupljanja podatka i iz procesa upisivanja tih podataka u fajlove sa ekstenzijom *gcda* u posebnom formatu, utvrđenom zajedničkim standardim biblioteke *libgcov* i alata *gcov*. Za prvobitne inicijalizacije, koje je potrebno izvršiti pre poziva funkcije `gcov_exit`, kao i za upravljanje samim pozivom, zadužena je funkcija `__gcov_init`. Jedinstveni poziv funkcije

`__gcov_init` na samom početku izvršavanja instrumentalizovanog programa obavlja se nezavisno od koda biblioteke *libgcov*. Kôd poziva se ugrađuje direktno u binarni kôd programa u toku prevođenja, uz uslov prisustva flegova za instrumentalizaciju. Stoga je kôd ove dve funkcije najpre izdvojen da, uz određene modifikacije, predstavlja bazu nove biblioteke.

Važno je napomenuti da se kôd biblioteke *libgcov* razlikuje u zavisnosti od verzije prevodioca *gcc*. U prethodnom poglavlju, predstavljena je šira slika, pre svega funkcionalnosti, bez implementacionih detalja. Stoga, u tom trenutku, nije bilo potrebno ograničavati se na konkretnu verziju. Princip koji se trenutno opisuje je karakteristika isključivo verzija starijih od 5.1.0. Implementacija biblioteke *libcoverage* je rađena po uzoru na kôd prevodioca verzije: 4.8.0. Pojedine specifične funkcionalnosti su preuzimane iz ranijih ili kasnijih verzija, u cilju optimizacije ili osamostaljivanja.

Ostatak koda koji je preuzet iz *gcc*-a, bilo u celosti, parcijalno ili uz modifikacije, dodat je isključivo u cilju definisanja nerazrešenih simbola prilikom osamostaljivanja biblioteke od prevodioca. Dakle, tokom prve faze implementacije je, na osnovu iscrpne analize strukture i funkcionalnosti onih modula prevodioca *gcc*, koji omogućavaju prikupljanje podataka iz izvršavanja programa, kreirana samoodrživa, ali još uvek ne funkcionalna baza nove biblioteke.

Druga faza implementacije obuhvatala je potrebne modifikacije u cilju postizanja prikupljanja podataka iz izvršavanja programa u toku njegovog rada. Rad alata *gcov*, koji rezultuje finalnim izveštajima, nije zavistan od izvršavanja programa, već samo od dostupnosti ulaznih argumenata: izvornog koda i fajlova *gcno* i *gcda*. Kako su fajlovi *gcno* dostupni već nakon prevođenja izvornog koda do objektnih fajlova, za ostvarenje željene funkcionalnosti je potrebno i dovoljno modifikovati trenutak nastajanja fajlova *gcda*. Uslovljenost završetkom rada programa, je suštinski uzrokovana time što se u okviru funkcije `__gcov_init`, za čije je bezuslovno izvršavanje dovoljno prevesti kôd sa instrumentalizacionim flegovima, poziva funkcija `gcov_exit` kao argument `atexit` funkcije standardne biblioteke *stdlib*. Eliminacijom ovog poziva, postignut je prvi korak ka prikupljanju podataka u proizvoljnom trenutku rada programa. Nakon ovog koraka, naziv `gcov_exit` je postao neprikladan i konfuzan, pa je izvršena njegova korekcija. U novoj biblioteci *libcoverage*, funkcija `gcov_exit` je preimenovana u `drew_coverage`. Na odabir naziva uticale su isključivo estetske i lične preferencije autora. Drugi korak predstavlja pružanje mogućnosti korisniku da sam odabere trenutak u kome će se poziv funkcije izvršiti.

Zaglavlje *coverage.h*, koje takođe ulazi u izgradnju nove biblioteke, sadrži deklaraciju funkcije: `drew_coverage`. Stoga je dovoljno da se u korisničkom kodu, na željenoj lokaciji, uključi ovo zaglavlje ili funkcija navede kao eksterna, a zatim regularno pozove. Razrešavanje simbola nastalih ovim pozivom, biće obavljeno u fazi linkovanja biblioteke *libcoverage.so*.

Za korisnike *Unix*-a i *Unix*-olikih operativnih sistema, implementirana je i mogućnost poziva korišćenjem signala *SIGUSR1*. Za ostvarivanje ove pogodnosti, neophodno je navesti u fazi linkovanja dodatni objektni fajl: *coverage_registration.o*. Kôd čijim prevodenjem nastaje prikazan je na slici 4.1. Pisan je u programskom jeziku C i u celosti se nalazi u okviru fajla: *coverage_registration.c*. Sastoji se iz dve funkcije: `coverage_handler` i `coverage_signal_registry`. Prva funkcija predstavlja alternativni *signal-handler* za *SIGUSR1*, u okviru koga se izvršava poziv funkcije: `drew_coverage`. Druga funkcija je zadužena za registrovanje funkcije `coverage_handler` kao *signal-handler* za *SIGUSR1*, a korišćenjem `__attribute__((constructor))` principa, vezana je za sam početak izvršavanja instrumentalizovanog programa. Korišćenje ove metode je opciono i zavisi isključivo od želje korisnika i mogućnosti korišćenja signala datog programa. Ukoliko operativni sistem ne podržava *POSIX* signale ili korisnički kôd predefiniše signal *SIGUSR1*, ovaj princip se ne može koristiti. Funkcionalnost je implementirana kao opciona i u statičkoj varijanti isključivo iz bezbednosnih razloga. Korišćenje `__attribute__((constructor))` principa u dinamičkoj biblioteci je veoma opasno i stoga u velikoj većini slučajeva nije podržano od strane prevodioca.

Pored osnovnih izmena, izvršene su i dve manje modifikacije funkcije za prikupljanje podataka i kreiranje fajlova *gcda*. Konstanta `GCOV_LOCKED`, definisana pretprocesorskom direktivom, a korišćena u cilju sprečavanja paralelizacije rada funkcije *gcov_exit* iz biblioteke *libgcov* i alata *gcov*, izbačena je iz upotrebe. U novom pristupu, paralelizacija procesa prikupljanja i obrade podataka je, iako bezbednosni izazov, poželjna osobina. Međutim, celokupni kôd koji koristi globalnu promenljivu `gcov_dump_complete`, zadužen za sprečavanje višestrukog izbacivanja podataka o izvršavanju programa, kao i sama promenljiva nisu eliminisani. Bez alternativnog mehanizma odbrane, eliminacija ove funkcionalnosti bi dovela do netačnih vrednosti u izveštaju, odnosno vrednosti multipliciranih brojem poziva funkcije za ispis. Prebacivanje odgovornosti premeštanja starih fajlova *gcda* pre novog poziva funkcije, nije prihvatljivo rešenje sa stanovišta bezbednosti i jednostavnosti upotrebe, dok bi automatsko naknadno prepakivanje fajlova *gcda* narušilo kompatibilnost sa

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

#define COVERAGE_SIGNAL SIGUSR1
extern void drew_coverage();

void coverage_handler(int signo){
    if (signo==COVERAGE_SIGNAL){
        printf("Dumping coverage data... \n");
        drew_coverage();
    }
}

__attribute__((constructor))
void coverage_signal_registry(){
    printf("Registrating signal SIGUSR1 for coverage data dump...\n");
    signal(SIGUSR1,coverage_handler);
}
```

Slika 4.1: Izvorni kôd u okviru fajla: *coverage_registration.c*

postojećim interfejsom trenutne implementacije.

Cilj treće faze implementacije predstavlja uklanjanje zavisnosti biblioteke od programskog prevodioca *gcc*. Prvi korak ka tom cilju, obavljen je još u toku prve faze, dodavanjem određenih funkcija i konstanti iz koda prevodioca, neophodnih za definisanje simbola, koji su u fazi linkovanja označeni kao nerazrešeni. U cilju logičkog osamostaljivanja i sprečavanja višestruke definisanosti simbola, funkcije koje programski prevodilac *gcc* koristi i van procesa instrumentalizacije, su zamenjene sličnim, funkcionalno kompatibilnim, varijantama. Primer je funkcija: *gcc_assert*, koja je zamenjena varijantom preuzetom iz verzije 5.0, pod nazivom: *gcov_nonruntime_assert*. Ukinuta je zavisnost od definisanosti promenljive *IN_LIBGCOV*, koja uslovljava prikupljanje podataka postojećom bibliotekom *libgcov*, na kraju izvršavanja programa.

Ukidanje zavisnosti biblioteke od prevodioca, odnosno omogućavanje njenog samostalnog prevođenja, prouzrokovalo je da biblioteka izgubi informaciju o verziji prevodioca. Različitost implementacije instrmentalizacije i prikupljanja podataka kroz verzije prevodioca *gcc*, morala se prevazići ograničavanjem na konkretnu verziju ili dodatnim modifikacijama. Kako prva opcija suštinski poništava efekat nezavisnosti biblioteke, doneta je odluka da se, u okviru četvrte faze implementacije, izvrše dodatna prilagođavanja koda biblioteke u cilju postizanja što veće multiverzionalnosti. U okviru izmena koje su objavljene pod verzijom 4.7.0. izvršen je i veći

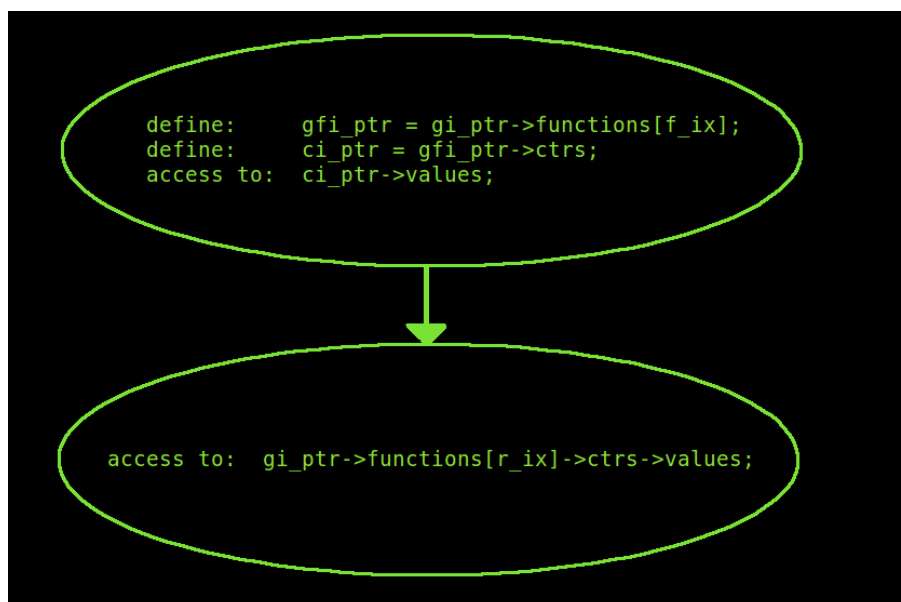
redizajn, ne samo procesa prikupljanja podataka, već i samog formata instrumentacionih struktura. Prilagođavanje biblioteke radu sa strukturama definisanim na stari način, iziskivalo bi izgradnju potpuno nove biblioteke, što je u datom trenutku bio nepotreban napor. Takođe, aktuelna verzija *gcc*-a u vreme ove implementacije je bila 6.2. što predstavlja gornje ograničenje multiverzionalnosti biblioteke *libcoverage.so*. Kasnijom analizom utvrđena je kompatibilnost i sa ostalim podizdanjima u okviru izdanja 6. U cilju izgradnje jedinstvene biblioteke za sve verzije *gcc*-a, počevši od 4.7.0. i zaključno sa 6.5.0., najpre su iz koda eliminisane sve provere verzije prevodioca, alata *gcov* i fajlova *gcno* i *gcda*. Uveden je novi mehanizam određivanja verzije koji osigurava isključivo poklapanje verzije u fajlovima *gcno* i *gcda*, neophodno za rad alata *gcov*. Globane vrednosti unutar same biblioteke koje su zavisne od verzije prevodioca, prilagođavaju se verziji pretprocesorskim direktivama. Prilagođavanje količine brojača koji se koriste za instrumentalizaciju, definisane promenljivom `GCOV_COUNTERS` prikazano je na slici 4.1.

```
/*Number of counters used for each gcc version*/
#if __GNUC__ == 6
#define GCOV_COUNTERS 10
#elif __GNUC__ == 5 && __GNUC_MINOR__ >= 1
#define GCOV_COUNTERS 10
#elif __GNUC__ == 4 && __GNUC_MINOR__ >= 9
#define GCOV_COUNTERS 9
#else
#define GCOV_COUNTERS 8
#endif
```

Slika 4.2: Definisane količine brojača u zavisnosti od verzije prevodioca

Poslednja faza implementacije obuhvata izmene vršene u cilju vremenske i prostorne optimizacije. U starijim verzijama prevodioca *gcc*, računanje kontrolne sume se obavljalo svega jednom, na samom početku izvršavanja instrumentalizovanog programa, u okviru funkcije `__gcov_init`. Od verzije 4.7.0., uveden je novi algoritam, po kome se kontrolna suma računa više puta, u toku čitanja podataka iz instrumentacionih struktura, posebno za svaki objektni fajl programa. Eventualna oštećenja podataka će sa dovoljno velikom verovatnoćom biti detektovana

u globalnog kontrolnoj sumi, pa računanje različite kontrolne sume za svaki element strukture `gcov_info` troši više vremena nego što doprinosi. Stoga su, u novoj biblioteci, kontrolne sume implementirane po uzoru na verziju 4.6.0. Dodatna optimizacija postignuta je redukovanjem broja sistemskih poziva za ispisivanje sadržaja u fajl *gcda*. Umesto velikog broja manjih poziva sistemskog poziva *write* za svaku 32-obitnu vrednost oznake ili brojača, vrši se jedinstveni poziv za celokupan sadržaj jednog fajla *gcda*. Kako se bafer za privremeno čuvanje ne reciklira, već samo proširuje, ovim se nije povećala prostorna složenost. Dodatni memorijски prostor je potreban samo za skladištenje jedne dodatne globalne promenljive koja sadrži broj trenutno neispisanih vrednosti. Sa druge strane, procesoru je prepuštena odluka o veličini pojedinačnog bloka koji će se upisati, što je na sistemu najoptimalnija moguća vrednost. Računanje vrednosti histograma je takođe izbačeno iz trenutne implementacije biblioteke *libcoverage*, pošto se vrednosti nisu koristile ni u jednom dokumentovanom slučaju upotrebe. Prisustvo histograma je zadržano isključivo u definiciji odgovarajuće instrumentalizacije strukture, u cilju kompatibilnosti sa prevodiocem. Manja ušteda prostora je ostvarena i uvođenjem pravolinijskog prolaska kroz strukture. Umesto alociranja privremenog pokazivača na međustrukturu, kao što je prikazano u gorenjem delu slike 4.1, korišćen je pravolinijski pristup prikazan u donjem delu iste slike.

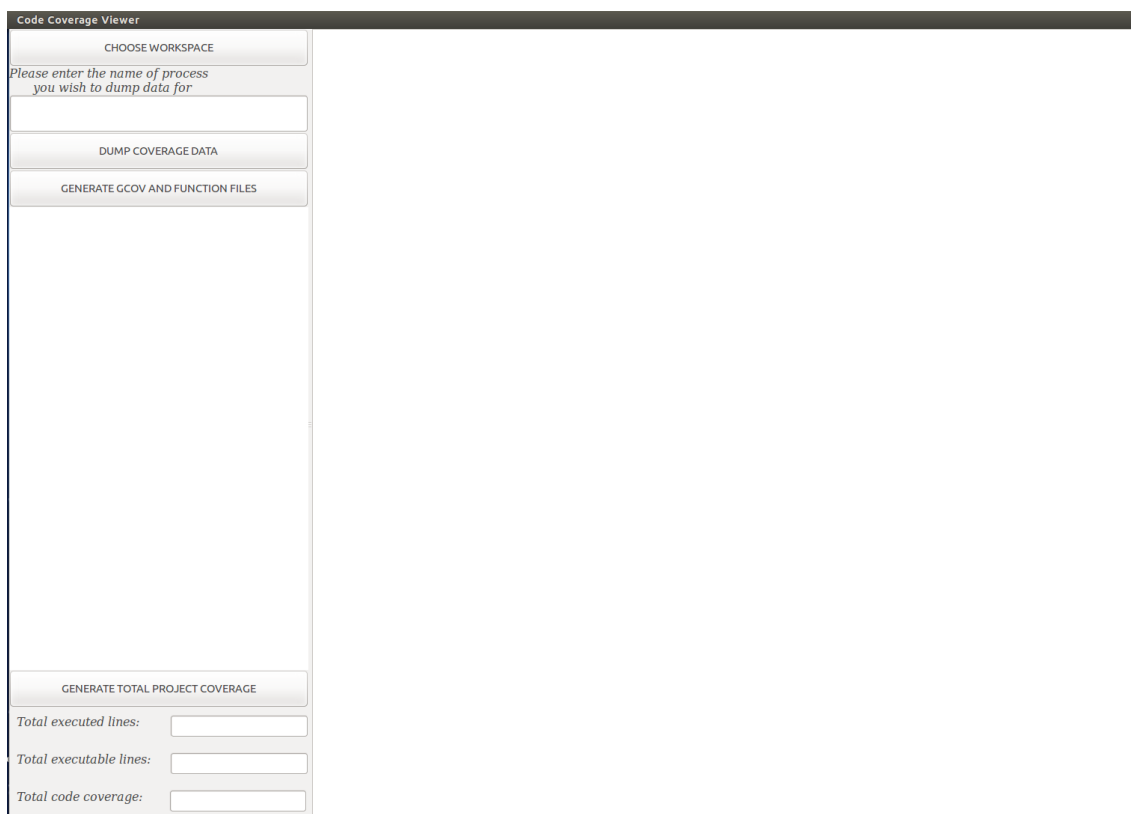


Slika 4.3: Pristup instrumentalizacionim strukturama

4.2 Implementacija grafičkog alata

Implementacija novog grafičkog alata za prikaz podataka iz izvršavanja programa, pod nazivom: *code_coverage_viewer*, izvršena je u programskom jeziku *Python*, korišćenjem posebnog modula za grafičku podršku: *wxPython*. Osnovna uloga novog alata jeste da predstavlja jednostavni, korisniku prilagođen omotač za celokupan proces prikupljanja i prikaza podataka u toku izvršavanja programa.

Grafički korisnički interfejs se sastoji iz dve celine: kontrolne i prezentacione, i jedne linije menija. Celine su implemenatciono predstavljene sa dva *skroll*-ujuća panela (`wx.Panel`), dobijena pomoću vertikalnog splitera ekrana (`wx.SplitterWindow`), dok je za kreiranje menija korišćena klasa `wx.MenuBar`. Na slici 4.2 je dat prikaz grafičkog korisničkog interfejsa nakon pokretanja. Osnovne komponente, čije će osobine i funkcionalnosti biti objašnjene u nastavku, vidljive su i dostupne odmah po pokretanju.



Slika 4.4: Pristup instrumentalitacionim strukturama

Kontrolna celina se proteže levom stranom interfejsa, zauzimajući okvirno jednu trećinu ukupnog prostora. Sadrži deset komponenti, raspoređenih u jedinstvenu

kolonu. Na samom vrhu se nalazi dugme za odabir radnog direktorijuma: *Choose workspace*, kreirano korišćenjem klase `wx.Button`. Klikom na ovo dugme, otvara se novi prozor, instanca klase `wx.DirDialog`, kao na slici 4.2 Pomoću njega se, navigacijom kroz sistem datoteka, odabira koreni direktorijum projekta ili pojedinačne komponente, za koju korisnik želi podatke o pokrivenosti. Nakon odabira, prozor za dijalog se zatvara i vrši se ažuriranje drvolike strukture koja oponaša strukturu odabranog direktorijuma. Više detalja o njoj, biće u nastavku teksta. Ukoliko se u njemu direktno ili rekurzivno nalaze već generisani izveštaji, oni će biti adekvatno prikazani u drvolikoj strukturi i za njih će važiti celokupna dalja funkcionalnost alata. Sledeće tri komponente u koloni čine tekstualno polje za unos imena procesa, labela za pružanje informacije šta u to polje treba uneti i dugme za kreiranje fajlova *gcda*. Tekstualno polje je instanca klase za prikaz dinamičkog teksta, pod nazivom: `wx.TextCtrl`, i služi za unos imena procesa kome treba poslati signal da prikupi podatke o svom dotadašnjem izvršavanju i upiše ih u fajlove sa ekstenzijom *gcda*. Ukoliko se na sistemu izvršava više programa sa tim imenom, signal će biti poslat svakom posebno. Sama funkcionalnost korišćenja komande *kill* za slanje signala *SIGUSR1* u cilju prikupljanja podataka, odnosno pozivanja funkcije `drew_coverage` iz biblioteke *libgcov*, se ne okida upisom imena, već klikom na dugme za kreiranje fajlova *gcda*, pod nazivom *Dump coverage data*. Ovo dugme je takođe instanca klase `wx.Button`, kao i prethodno, dok je za sve labele korišćena klasa za prikaz statičkog teksta, pod nazivom: `wx.StaticText`. Funkcionalnost ove tri komponente je dostupna samo korisnicima *Unix* ili *Unix*-olikih sistema, koji su pored instrumentalizacionih flegova i biblioteke *libcoverage*, svoj program preveli i koristeći objektni fajl *coverage_registration.o*, i nisu predefinisali signal *SIGUSR1*. Ograničenje je napomenuto u specifikaciji alata, kao i u oviru *Help*-a. Izvršavanje systemske komanade `kill -10 <pid>`, kao i svih ostalih systemskih komandi je implementirano pomoću funkcije `system`, modula `os`. Ovo je najjednostavniji princip, bez bezbednosnih izazova, koji u potpunosti zadovoljava potrebe novog grafičkog alata. Naredne dve komponente predstavljaju interfejs ka funkcionalnosti generisanja izveštaja alatom *gcov* i kontrole njihovih prikazivanja. Dugme: *Generate gcov and function files* okida rekurzivnu pretragu radnog direktorijuma i poziv alata *gcov* za sve pronađene fajlove *gcda*. Odgovarajući fajl *gcno* se nalazi na istoj lokaciji gde i *gcda*, dok se putanja do fajla sa izvornim kodom dobija parsiranjem sadržaja fajla *gcno*. Ovim postupkom se kreiraju dva tipa izveštaja:

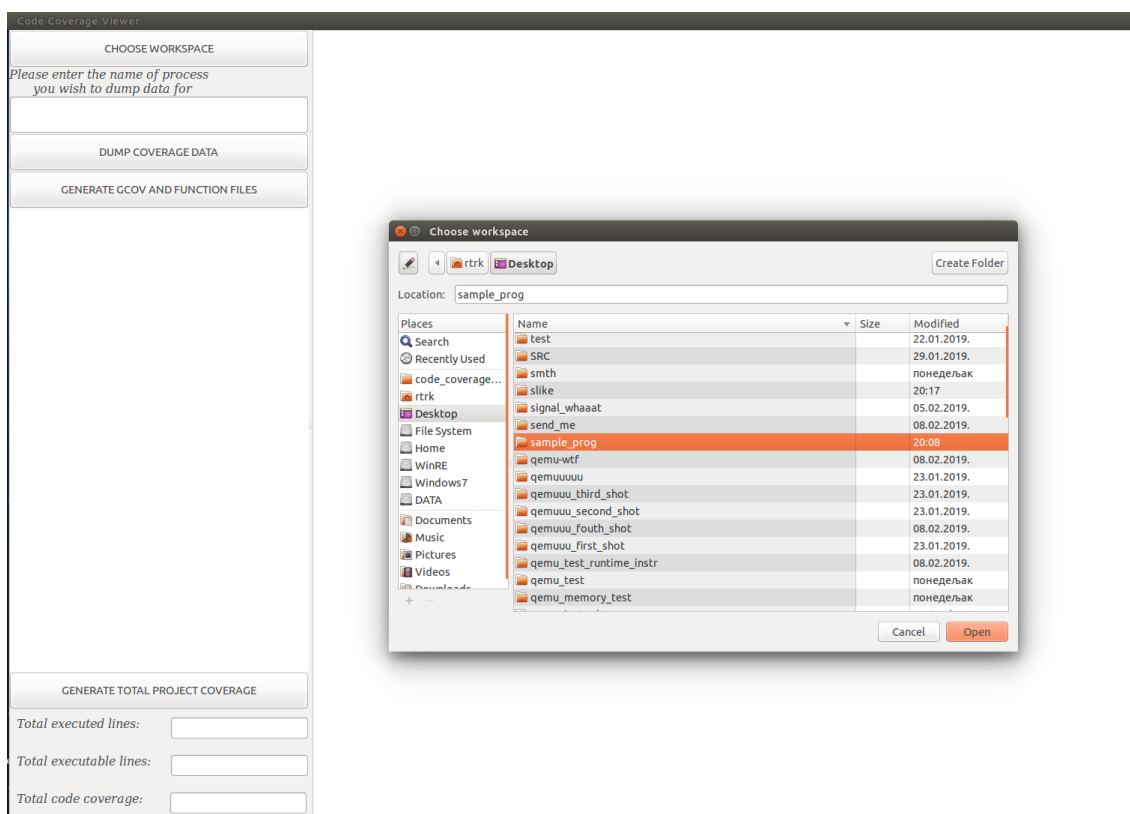
1. Standardni izveštaji sa ekstenzijom *gcov*, koji sadrže kvantitativne podatke za

svaku izvršnu liniju koda

2. Funkcijski izveštaji sa ekstenzijom *fun*, koji sadrže podatke o pokrivenosti pojedinačnih funkcija i ukupnu pokrivenost fajla izvornog koda

Kreiranje oba tipa izveštaja se vrši jedinstvenim pozivom alata, sa jednom dodatnom opcijom: *-f*. Alat *gcov* sam generiše standardni izveštaj, dok sadržaj funkcijskog izveštaja predstavlja preusmereni ispis na standardni izlaz. Svi izveštaji se smeštaju u drvoliku strukturu, lociranu ispod dugmeta za njihovo generisanje, koja je implementirana kao instanca klase `wx.TreeCtrl`. Celokupni sadržaj se briše, a drvolika struktura se ponovo izgrađuje, popunjavajući se novim podacima. Ista procedura se vrši, kao što je već napomenuto, i prilikom promene radnog direktorijuma. Ova struktura ima dvostruku ulogu. Pored intuitivnog prikaza imena i lokacije svakog izveštaja, u vidu drveta koje oponaša strukturu radnog direktorijuma, služi i kao kontrolna jedinica za prikaz sadržaja u desnoj celini grafičkog korisničkog interfejsa. Prikaz sadržaja željenog izveštaja se okida jednim klikom na njegovo ime. Na samom dnu leve celine nalazi se skup od četiri komponente koje služe za određivanje i prikaz ukupne pokrivenosti koda u okviru radnog direktorijuma: jedno dugme i tri kompozitne strukture. Dugme *Generate total project coverage* okida računanje tri podatka: ukupan broj izvršnih linija, ukupan broj izvršenih linija do trenutka kreiranja fajlova *gcda* i procenat koji označava ukupnu pokrivenost koda u okviru odabranog radnog direktorijuma. Svaka od tri kompozitne strukture sadrži po jedno tekstualno polje za prikaz jednog od tako izračunatih podataka i labelu za objašnjenje sadržaja tekstualnog polja. Sam proces računanja zahteva parsiranje standardnih izveštaja u cilju ekstrakcije kvantitativnih podataka izvršenosti pojedinačnih linija i može biti veoma vremenski zahtevan. Stoga nije izvršavan u okviru procesa generisanja izveštaja, već kao zasebna funkcionalnost. Ukupna vremenska složenost je veća pri ovakvom pristupu, ali je vreme potrebno za dobijanje prvih podataka prilično redukovano. U toku izračunavanja ukupne pokrivenosti, korisniku su na raspolaganju pojedinačni izveštaji za pregledanje i analizu. Prikaz grafičkog korisničkog interfejsa nakon generisanja izveštaja i računanja ukupne pokrivenosti dat je na slici 4.2.

Prezentaciona celina se proteže desnom stranom interfejsa, zauzimajući okvirno dve trećine ukupnog prostora. Sadrži svega jednu komponentu, koja služi za prikaz sadržaja konkretnog izveštaja, odabranog pomoću drvolike strukture leve celine. Implementirana je kao instanca klase za prikaz dinamičkog teksta: `wx.TextCtrl`,

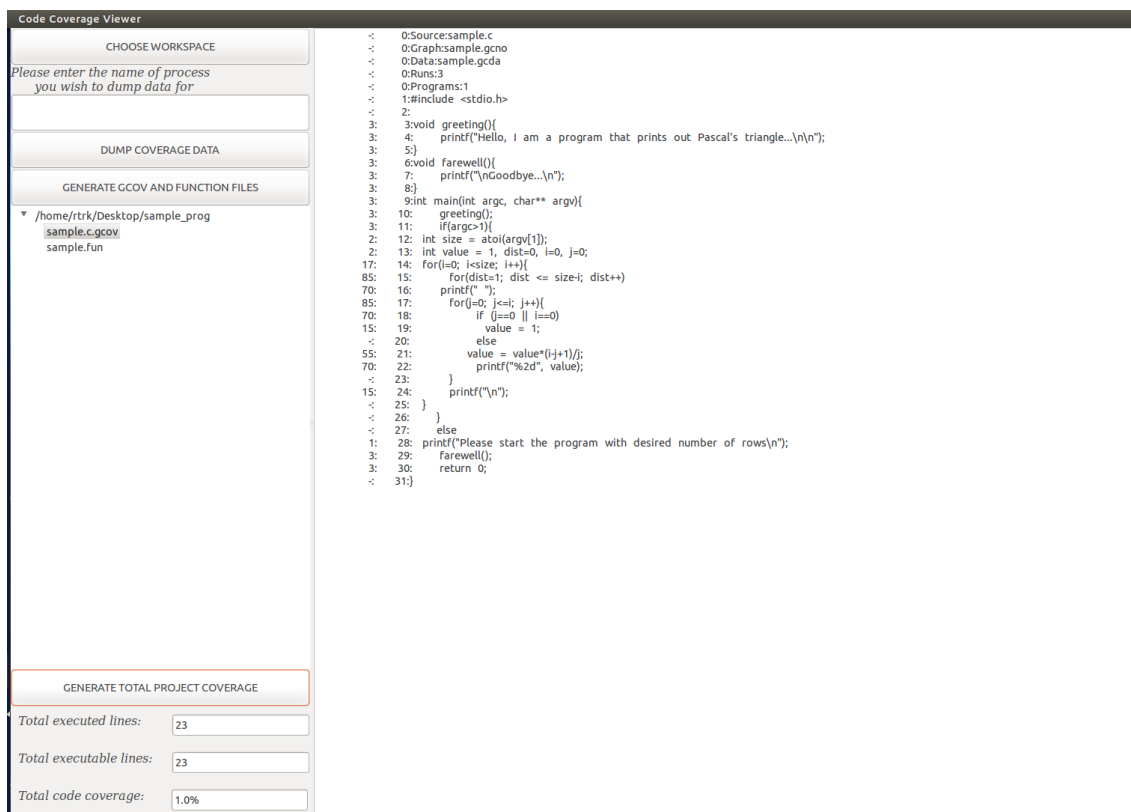


Slika 4.5: Pristup instrumentalitacionim strukturama

čime se pruža dodatna mogućnost kopiranja sadržaja, menjanja ili brisanja. Pri-
kaz izveštaja koji sadrže ne-*ASCII* karaktere, omogućen je korišćenjem unapredene
azbuke koju nudi paket: *ISO-8859-1*. Postavljen je kao podrazumevani paket, kako
bi se unapredila i upotrebnost vrednost funkcionalnosti odabira radnog direktorijuma,
ali i omogućilo čitanje fajlova *gcno*.

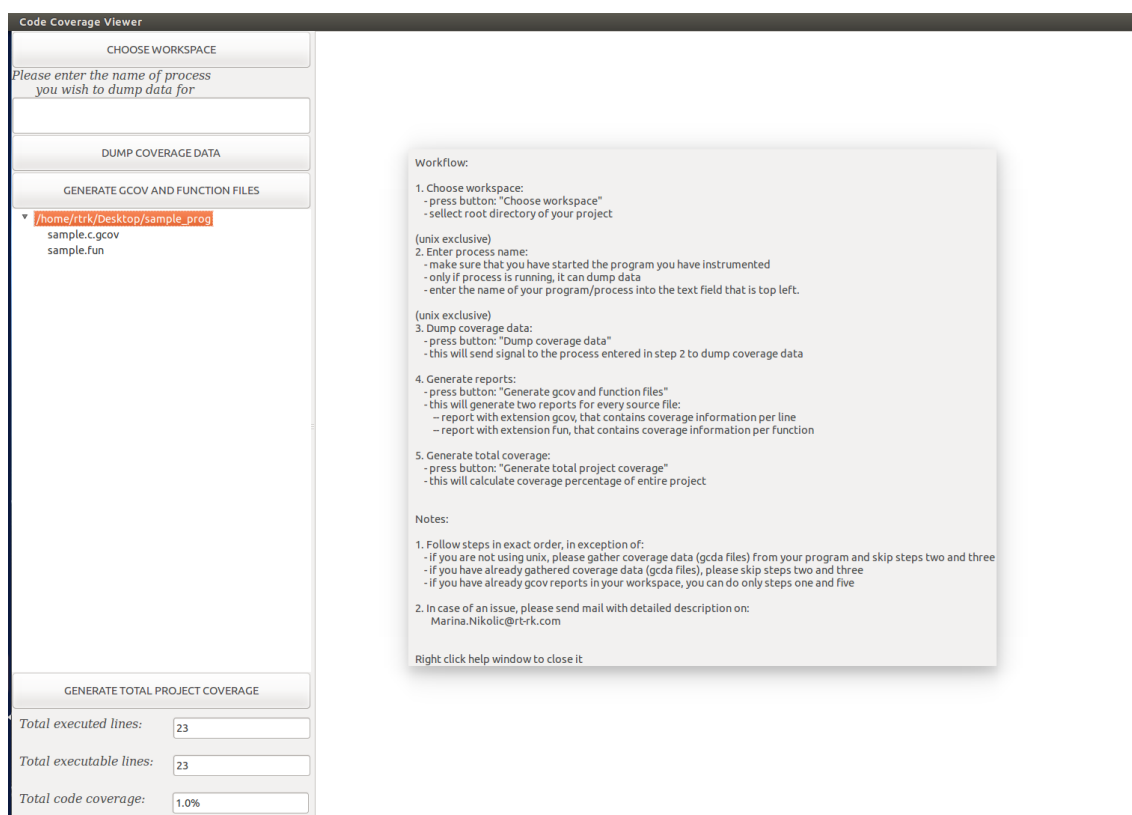
Linija menija je implementirana korišćenjem klase `wx.MenuBar`. Sadrži ime alata
i jedno dugme, tipa `wx.Menu`, pod nazivom *File*. Jednim klikom na dugme *File*,
pojavljuje se lista sa dve opcije: *Help* i *Quit*. Odabiranjem opcije za pomoć, na
centru ekrana se pojavljuje novi prozor, klase `wx.PopupWindow`, kao na slici 4.2.
Prozor se sastoji iz jedinsvenog panela, čiju celokupnu površinu zauzima tekstualno
polje, statičke prirode, sa kratim uputstvom za upotrebu alata. Prozor je mobilan,
što je ostvareno pomoću preračunavanja kordinata, a nakon korišćenja, može se
zatvoriti desnim klikom na bilo koju tačku njegove površine. Opcija *Quit* služi za
zatvaranje glavnog prozora alata.

U okviru specifikacije alata, kao i u okviru *Help*-a, detaljno je opisana pravilna



Slika 4.6: Pristup instrumentalitacionim strukturama

upotreba alata. Za korisnike koji nemaju naviku da čitaju dokumentacije, specifikacije i pomoćne sadržaje, osmišljen je mehanizam podsvesnog navođenja. Naime, redosled komponenti kontrolne celine je osmišljen i implementiran na način koji ima za cilj podsvesno navođenje korisnika na pravilni redosled upotrebe. Akcije donjih skupova komponenti su u velikoj meri uslovljene akcijama onih iznad, što uslovljava korišćenje „odozgo na gore”. Poslednji stepen zaštite od nepravilne upotrebe nalazi se na nivou pojedinačne komponente i onemogućava njen rad, ukoliko potrebni uslovi nisu ispunjeni. Ispunjenost uslova se određuje stanjem radnog direktorijuma i programa, a ne izvršenosti funkcionalnosti alata, čime su pokriveni i slučajevi parcijalne upotrebe alata, kao na primer, pregledanje već generisanih izveštaja. Takođe, tom prilikom se korisniku otvara i mali prozor za dijalog, instanca klase `wx.MessageDialog`, pomoću koga se informiše o konkretnoj grešci u proceduri i dobija savet o adekvatnom sledećem koraku.



Slika 4.7: Pristup instrumentalitacionim strukturama

Glava 5

Analiza korektnosti i performasi

5.1 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 6

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/~smal-kov/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] Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):50, 2018.
- [10] 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.
- [11] R Brader, H Hilliker, and A Wills. Unit Testing: Testing the Inside. *Microsoft Developer Guidance*, 2013.
- [12] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of model checking*. Springer, 2018.

- [13] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.
- [14] A Glower. In pursuit of code quality: Don’t be fooled by the coverage report. *IBM Developer Works blog post*, 2006.
- [15] V. Gupta. Measurement of Dynamic Metrics Using Dynamic Analysis of Programs. *APPLIED COMPUTING CONFERENCE (ACC ’08), Istanbul, Turkey*, 2008.
- [16] 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.
- [17] B Marick. How to misuse code coverage. *Proceedings of the 16th International Conference on Testing Computer Software*, 1999.
- [18] F Nielson, H. R. Nielson, and C Hankin. Principles of program analysis. *Springer Science & Business Media*, 2015.
- [19] A Piziali. “Code coverage,” in Functional verification coverage measurement and analysis. *Springer Science & Business Media*, 2007.
- [20] Milena Vujošević Janičić. Verifikacija softvera. http://www.programskijezici.matf.bg.ac.rs/vs/predavanja/03_dinamicka_analiza/03_dinamicka_analiza.pdf
- [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 ...