

UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Marina R. Nikolić

**PRIKUPLJANJE I PRIKAZ PODATAKA O
IZVRŠAVANJU PROGRAMA**

master rad

Beograd, 2019.

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: Za olakšavanje procesa testiranja i optimizacije, mogu se koristiti podaci o pokrivenosti koda, koje generišu alati za profajliranje. Većina poznatijih prevodilaca ima u sebi ugrađenu podršku za prikupljanje podataka iz izvršavanja i njihovu transformaciju do informacije o pokrivenosti različitih jedinica koda. Po svojim performansama ističe se programski prevodilac *GCC*, čiji alat *gcov* generiše izveštaje sa frekvencijama izvršavanja pojedinačnih linija, blokova, funkcija i grana, kao i podatkom o pokrivenosti na nivou fajla izvornog koda.

Osnovne mane trenutne, zvanične implementacije obuhvataju nedostupnost podataka pre završetka rada programa, kao i otežano pregledanje konačnih rezultata na nivou celokupnog projekta. Profajliranje programa sa izuzetno dugim vremenom rada, koji se ne mogu zaustavljati za potrebe analize ili koriste privremena skladišta kojima je pristup ograničen na vremenski okvir trajanja izvršavanja, poput servera, operativnih sistema ili sistema za rad u realnom vremenu, stoga nije moguće ili je izuzetno otežano u okviru dostupne implementacije. Nedostatak informacije o pokrivenosti na nivou modula većeg od pojedinačnog fajla izvornog koda, kao i strukturnog pregledanja svih izveštaja, može uticati na kreiranje pogrešnog zaključka o rezultatima analize i time usmeriti proces testiranja ili optimizacije u lošem smeru.

U okviru ovog rada implementirana je podrška za prikupljanje i prikaz podataka iz izvršavanja programa, prevedenih programskim prevodiocem *GCC*, u toku izvršavanja. Kreirana je nova biblioteka, koja omogućava dostupnost podataka od interesa pre završetka rada programa, kao i novi grafički alat za strukturno i intuitivno pregledanje konačnih rezultata njihove obrade. Testiranje implementiranog rešenja je sprovedeno nad kompleksnim projektom otvorenog koda pod nazivom *QEMU* i prikazano na način koji ujedno predstavlja i uputstvo za upotrebu ovog rešenja za dobijanje informacije o pokrivenosti koda drugih softvera.

Ključne reči: profajliranje, pokrivenost koda, *GCC*, *gcov*

Sadržaj

Sadržaj	v
1 Uvod	1
2 Analiza performansi programa	4
2.1 Vrste analize programa	4
2.2 Profajliranje programa	8
3 Pokrivenost koda u prevodiocu GCC	14
3.1 Opis implementacije ponuđenog rešenja	14
3.2 Nedostaci ponuđenog rešenja	21
3.3 Ideje za unapređenje	22
4 Implementacija	27
4.1 Biblioteka libcoverage	27
4.2 Korisnički interfejs	33
5 Verifikacija i validacija implementiranog rešenja	39
5.1 Analiza karakteristika	39
5.2 Testiranje korektnosti i performansi	44
6 Zaključak	80
Bibliografija	82

Glava 1

Uvod

Razvoj komercijalnog softvera, za koji su standardi kvaliteta u pogledu korektnosti, performansi i upotrebljivosti vrlo visoki, mora sadržati i napredne i iscrpne faze testiranja i optimizacije. Sprovođenje ovih faza, na način koji svojom brzinom i efikasnošću garantuje kompetentnost na današnjem tržištu, uslovljeno je dobrim poznavanjem koda softvera, pre svega ponašanja u različitim slučajevima upotrebe, kritičnih sekcija, memorijski ili vremenski zahtevnih segmenata i slično. Komplexnost funkcionalnih zahteva klijenata, ograničene hardverske mogućnosti kao i odlazak ključnih kadrova, sa visokim stepenom poznavanja projekta, samo su neki od faktora koji utiču na smanjenje nivoa poznavanja interne strukture i ponašanja, čime u velikoj meri otežavaju procese testiranja i optimizacije.

Nadoknađivanje nedostajućeg znanja, može se postići primenom određenih tehnika dinamičke analize programa, kao što je profajliranje. Instrumentalizacijom koda se u njega ugrađuju posebne instrukcije za praćenje izvršavanja, zahvaljujući kojima alati za profajliranje prikupljaju sirove metapodatke koje će dalje preraditi u korisne informacije, poput pokrivenosti, odnosno stepena izvršenosti, celokupnog koda ili neke njegove celine. One obezbeđuju potrebno znanje o tokovima izvršavanja u različitim slučajevima upotrebe, ali i o opterećenosti konkretnih segmenata rada programa. Zbog toga predstavljaju važnu smernicu razvojnom timu u procesima testiranja i optimizacije.

Najpoznatiji prevodioci su: *GCC* [4], *ICC* [7] i *Clang* [2]. *GCC* [31] (*GNU Compiler Collection*) predstavlja skup programskih prevodioca koji transformišu kôd napisan u jeziku: *C*, *C++*, *Objective-C*, *Fortran*, *Ada*, *Go*, ili *D*, u mašinski kod, zajedno sa pratećim bibliotekama i alatima potrebnim za rad u nekom od ovih jezika. Kreirao ga je *Ricard Stalmen*, 1987. godine, za operativni sistem

GNU, u okviru projekta koji je, pod nazivom *GNU (GNU is Not Unix)*, nastao 1983. godine na *MIT*-u. *GCC* se nalazi pod licencom *GNU GPL (GNU General Public License)*, što ga čini slobodnim softverom, otvorenog koda, i kao takav je imao veliku ulogu u razvoju programskih prevodioca. *Clang (C Language)* predstavlja primarni interfejs prema programskim prevodiocima jezika: *C*, *C++*, *Objective C/C++*, *OpenCL*, *CUDA*, i *RenderScript* projekta *LLVM* [8] [25]. Ovaj projekat implementira kolekciju programskih prevodilaca i alata raznih namena (npr. obrada podataka iz izvršavanja programa), napisanih u programskom jeziku *C++*, na način koji odražava modularnost i ponovnu upotrebljivost koda. Započet je 2000. godine na Univerzitetu *Illinois*. *ICC* [6] (*Intel C++ Compiler*) predstavlja skup programskih prevodioca za jezike *C* i *stranoC++*. Vlasništvo je kompanije *Intel*, a celokupan kod je zatvoren i zaštićen komercionalnom licencom. Ovi prevodioci sadrže u sebi sadrže ugrađenu podršku za prikupljanje podataka iz izvršavanja, kao i alate za njihovu obradu i prevođenje u format pogodan za čitanje i analiziranje od strane razvojnog tima.

Zahvaljujući svojim performansama, odnosno boljoj vremenskoj i prostornoj složenosti u odnosu na profajljanje alatima projekta *LLVM*, kao i dostupnosti koda i besplatnim korišćenjem, kojom prednjači u odnosu na *ICC*, ističe se programski prevodilac *GCC* sa svojim alatom *gcov* [5]. Izveštaji kreirani ovim alatom sadrže veoma korisne informacije o izvršenosti pojedinačnih linija, funkcija, blokova ili grana.

Međutim, pored izuzetnih performansi, trenutna implementacija prikupljanja podataka iz izvršavanja u okviru programskog prevodioca *GCC*, poseduje i jedan važan nedostatak u odnosu na *Clang*, a to je nedostupnost tih podataka pre završetka rada analiziranog programa. U specifičnim situacijama, kada je izvršavanje isuviše dugo i/ili se ne sme prekidati zbog profajljanja, potrebno je obezbediti ove informacije u toku rada programa.

Iz tih razloga je, u okviru ovog rada, implementirana podrška za prikupljanje podataka iz izvršavanja u toku rada programa, proširivanjem mogućnosti programskog prevodioca *GCC*. Pored toga, kreiran je i novi grafički interfejs za prikaz tih podataka, koji nadomešćuje nedostatke alata *gcov* u pogledu preglednosti, informativnosti na nivou većem od pojedinačnog fajla izvornog koda, kao i ukupnog kvaliteta korisničkog doživljaja. Testiranje implementiranog rešenja sprovedeno je nad kompleksnim softverom koji služi za emuliranje različitih arhitektura, pod nazivom *QEMU* [16].

Druga glava rada predstavlja uvod u analizu programa, njene vrste i tehnike.

Posebna pažnja posvećena je tehnici dinamičke analize pod nazivom profajljanje, koja obuhvata proces instrumentalizacije koda, prikupljanja i obrade podataka iz izvršavanja programa. Definisani su i pojam i značaj pokrivenosti koda, kao jedne od najvažnijih informacija koja se izvodi iz tih podataka. Počevši od naredne glave vrši se restrikcija na programski prevodilac *GCC*. U okviru treće glave, opisana je detaljno postojeća implementacija prikupljanja podataka i demonstrirane mogućnosti alata *gcov*, dok se u okviru četvrte glave nalaze detalji implementacije unapređenja koje predstavlja osnovu ovog rada: proširenje mogućnosti programskog prevodioca *GCC* u cilju dobijanja podataka iz izvršavanja pre kraja programa i kreiranje novog grafičkog interfejsa za njihov prikaz. Peta glava posvećena je procesima testiranja i analize performansi, najpre nad jednim jednostavnijim primerom, a zatim i nad alatom *QEMU*.

Softversko rešenje za prikupljanje i prikaz podataka iz izvršavanja u toku rada programa je predstavljeno i na konferenciji *ETRAN*, 2017. godine [29].

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 [28] 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, poput zastupljenosti komentara [13] ili ciklomske kompleksnosti [24], 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 jedne 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 potvrđama ispravnosti ili performansi, već isključivo tretirati kao smernice pri razvoju.

Postoje i određene statičke metode koje su značajno preciznije od metrika, poput simboličkog izvršavanja [14], proveravanja modela [19] ili apstraktne interpretacije [20]. 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 nedovoljeno 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 [22] 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 [9]. 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 [23, 32] 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 određenog dokumenta, podebljavanje jedne reči i snimanje izmenjenog dokumenta na disk. 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 npr. funkcija za podvlačenje teksta nije. Ukoliko eksperimentalni podaci, poreklom iz konkretnog izvršavanja, nisu u skladu sa teorijskom pretpostavkom, negiraju izvršavanje funkcije za podebljavanje teksta ili potvrđuju izvršavanje neke nepredviđene, poput funkcije za podvlačenje 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 snimanja izmenjenog dokumenta na disk 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: instrumentalizacije, prikupljanja i obrade podataka.

Faze profajliranja

Instrumentalizacija [10] 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 najefikasnije 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 karakte-

istikama 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.

U zavisnosti od vremenskog trenutka i/ili načina odvijanja procesa instrumentalizacije, možemo razlikovati više vrsta profajliranja. U širem smislu, pod pojmom profajliranje se može podrazumevati i takozvano „manuelno profajliranje”, koje se zasniva na analizi podataka dobijenih izvršavanjem dodatnih instrukcija, unetih u kôd od strane programera, u cilju detekcije grešaka ili analize performansi. Mogu se dodati jednostavne komande za ispis, ali i kreirati složenije strukture za praćenje promenljivih, funkcija, grananja ili petlji, na osnovu kojih se može pretpostaviti tok izvršavanja programa. U užem smislu, restrikcijom na automatizovanu dinamičku analizu, možemo prema vremenskom trenutku vršenja instrumentalizacije, definisati još dva tipa profajliranja:

1. profajliranje zasnovano na instrumentalizaciji u toku izvršavanja
2. profajliranje zasnovano na instrumentalizaciji u toku prevođenja

Kao primer prvog tipa, može se posmatrati platforma za pravljenje alata za dinamičku analizu mašinskog koda pod nazivom *Valgrind* [11] [27]. *Valgrind* alat raščlanjuje originalni mašinski kôd programa u međukod, instrumentalizuje, a zatim ponovo prevodi u mašinski kod, koji se čuva u memoriji i izvršava u svrhu profajliranja. Instrumentalizaciju i prikupljanje, u slučaju drugog tipa, vrši prevodilac, dok je za obradu zadužen dodatni alat, koji se obično nalazi u sklopu projekta samog prevodioca. U nastavku teksta biće razmatran samo ovaj tip profajliranja.

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 [12, 17, 33, 21, 26, 30] 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 „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, debugovanje 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 problemi mogu uzrokovati blokiranje ili prestanak rada celog sistema, a njihovo blagovremeno otkrivanje je veoma teško. Algoritam raspoređivanja procesa operativnog sistema 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. 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 programskog prevodioca *GCC*

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, bi stoga kombinovali dobru ideju projekta *LLVM* i dobre tehnike prevodioca *GCC*, čime bi prednjačili, i kada su u pitanju mogućnosti, i kada su u pitanju performanse. U okviru projekta na kome je utemeljen ovaj rad, je upravo iz tog razloga, izvršena detaljna analiza postojećih mogućnosti u okviru prevodioca *GCC* i implementirana podrška za prikupljanje podataka u toku izvršavanja. U cilju poboljšavanja korisničkog iskustva, kreiran je i novi, unapređeni interfejs za vizuelni prikaz prikupljenih podataka.

3.1 Opis implementacije ponuđenog rešenja

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* [5, 3].

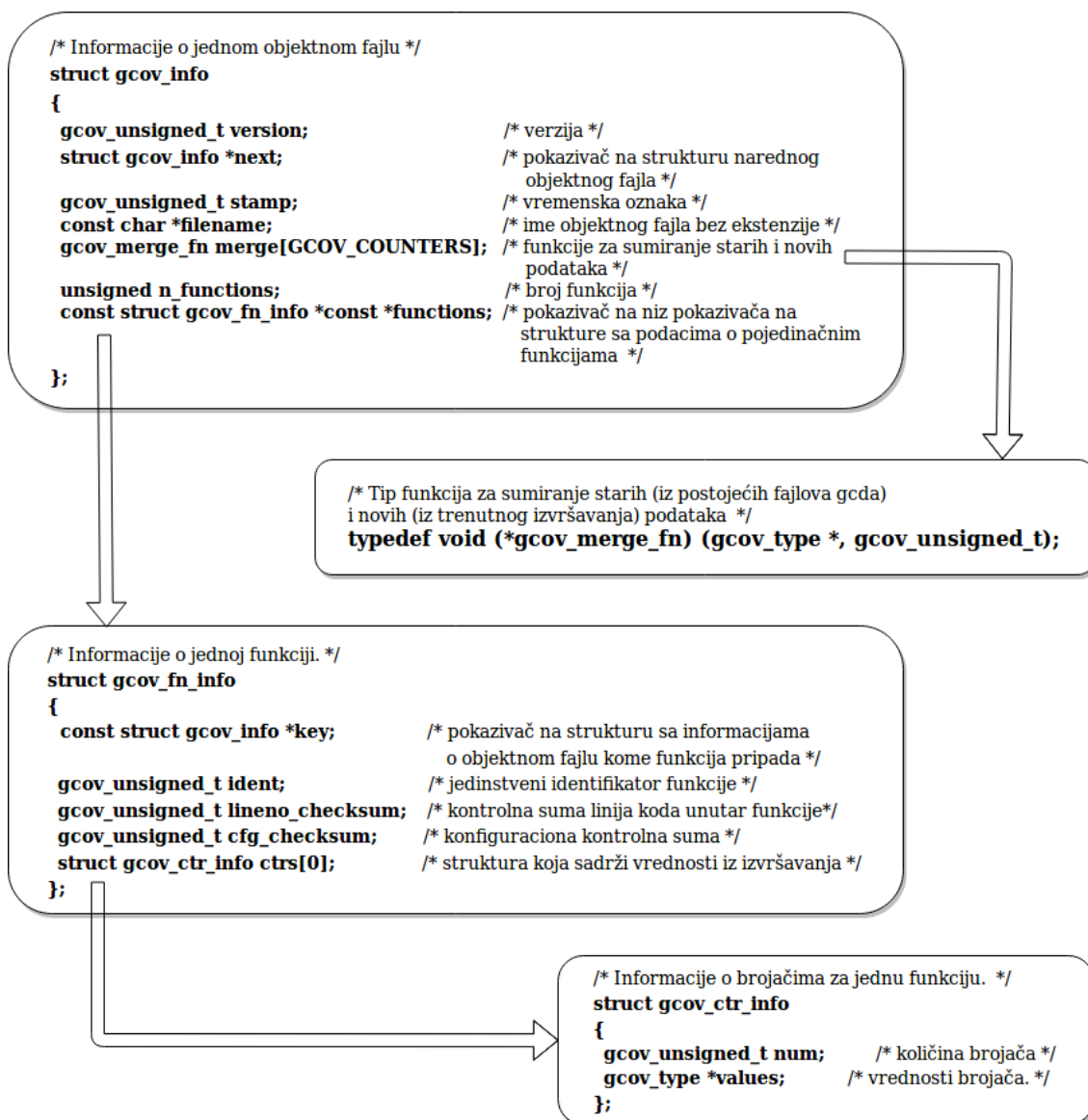
Metapodaci izvršavanja čuvaju se u deljenoj memoriji programa, u listi posebnih, globalno definisanih struktura tipa *gcov_info*, čija se inicijalizacija ugrađuje u

binarni kôd prevođenjem sa posebnim zastavicama za instrumentalizaciju: `-fprofile-arcs` i `-ftest-coverage`. Zastavice 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 zastavica za instrumentalizaciju uslovljava obavljanje još jedne važne aktivnosti, a to je kreiranje dodatnog fajla, odmah pored njemu odgovarajućeg objektnog fajla, sa ekstenzijom *gno* (*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 nadograditi podacima dobijenim dinamički u toku izvršavanja. Format fajla *gno* je utvrđen zajedničkim standardom programskog prevodioca *GCC* 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 statičkih podataka o strukturi izvornog koda u vidu eksternih binarnih fajlova, nasuprot kreiranja novih sekcija u izvršnom fajlu za njihovo skladištenje, osnovni uzrok boljih memorijskih performansi instrumentalizacije programskog prevodioca *GCC* u odnosu na *Clang*. U okviru instrumentalizovanog izvršnog fajla prevedenog programskim prevodiocem *GCC* nalaze se isključivo dodatne instrukcije za kreiranje i popunjavanje struktura za metapodatke, što predstavlja minimum za funkcionalisanje procesa prikupljanja. Uvećanje izvršnog fajla do veće vrednosti od memorijske količine koja je za njega predviđena na sistemu može potpuno onemogućiti njegovo smeštanje u sekundarnu memoriju ili dovesti do određenih nepravilnosti u radu (npr. alternativni tok izvršavanja prouzrokovan neuspehom dinamičkom alokacijom). Najveću opasnost predstavlja upravo ta razlika u ponašanju između memorijski zahtevnijeg instrumentalizovanog i regularnog programa, usled koje rezultati merenja pokrivenosti neće odražavati realno stanje. Uticaj se ne može u potpunosti ukloniti, ali se tehnika kao što je izmeštanje podataka nepotrebnih u fazi prikupljanja, može znatno smanjiti. Stoga, na sistemima sa veoma ograničenim memorijskim prostorom, *GCC* instrumentalizacija je često jedina moguća.

Definicije i međusobne veze najvažnijih instrumentalizacionih struktura su 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.



Slika 3.1: Instrumentalizacione strukture i njihove međusobne veze

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 njihov eksterni alat, kao što je *gcov*, ne može direktno koristiti iz više razloga. Prvi razlog je bezbednosne prirode, i velikom merom je obrazložen u prethodnoj glavi. 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.

Rešenje koje je dostupno u okviru programskog prevodioca *GCC*, 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_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 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 prethodnih 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 premeštati i skladištiti. To je veoma korisno, 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 premestiti 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 sa ekstenzijom *gcno* i jedan fajl sa vrednostima brojača sa ekstenzijom *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. U okviru listinga 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 granama, blokovima i slično.

```
--: 0:Source:paskal.c
--: 0:Graph:paskal.gcno
--: 0:Data:paskal.gcda
--: 0:Runs:1
--: 0:Programs:1
--: 1:#include <stdio.h>
--: 2:
1: 3:int main(int argc, char** argv){
1: 4: int velicina=atoi(argv[1]),vrednost=1,razmak=0,i,j;
1: 5: printf("Paskalov trougao sa %d redova: ",velicina);
6: 6: for(i=0; i<velicina; i++){
20: 7:     for(razmak=1; razmak<=velicina-i; razmak++){
15: 8:         printf(" ");
15: 9:         for(j=0; j<i; j++){
10: 10:             if(j==0 || i==0)
4: 11:                 vrednost=1;
--: 12:             else
6: 13:                 vrednost=vrednost*(i-j+1)/j;
10: 14:             printf("%2d", vrednost);
--: 15:         }
5: 16:     printf("\n");
--: 17: }
1: 18: return 0;
--: 19:}
```

Listing 3.1: Primer osnovnog izveštaja koji generiše *gcov*

Opcijom *-b*, izveštaj se obogaćuje statističkim podacima o funkcijama, osnovnim blokovima, granama i pozivima, kao u okviru listinga 3.2.

```

-:      0:Source:paskal.c
-:      0:Graph:paskal.gcno
-:      0:Data:paskal.gcda
-:      0:Runs:1
-:      0:Programs:1
-:      1:#include <stdio.h>
-:      2:
function main called 1 returned 100% blocks executed 100%
  1:      3:int main(int argc, char** argv){
  1:      4:      int velicina=atoi(argv[1]),vrednost=1,razmak=0,i,j;
call    0 returned 100%
  1:      5:      printf("Paskalov trougao sa %d redova: ",velicina);
call    0 returned 100%
  6:      6:      for(i=0; i<velicina; i++){
branch  0 taken 83%
branch  1 taken 17% (fallthrough)
 20:     7:      for(razmak=1; razmak<=velicina-i; razmak++)
branch  0 taken 75%
branch  1 taken 25% (fallthrough)
 15:     8:          printf(" ");
call    0 returned 100%
 15:     9:      for(j=0; j<i; j++){
branch  0 taken 67%
branch  1 taken 33% (fallthrough)
 10:    10:          if(j==0 || i==0)
branch  0 taken 60% (fallthrough)
branch  1 taken 40%
branch  2 taken 0% (fallthrough)
branch  3 taken 100%
  4:    11:          vrednost=1;
-:    12:          else
  6:    13:          vrednost=vrednost*(i-j+1)/j;
 10:    14:          printf("%2d", vrednost);
call    0 returned 100%
-:    15:      }
  5:    16:      printf("\n");
call    0 returned 100%
-:    17:      }
  1:    18:      return 0;
-:    19:  }

```

Listing 3.2: Primer izveštaja koji generiše alat *gcov* sa opcijom *-b*

Osnovni blok [1] predstavlja jedinicu koda koja se izvršava pravolinijski, odnosno sadrži jedinstvene tačke ulaza i izlaza. Pre definicije svake funkcije, ispisuje se koliko puta je pozvana, koliko puta je regularno završena, kao i procenat izvršenih osnovnih blokova u njoj. Nakon završne linije svakog osnovnog bloka, ispisuju se podaci o pokrivenosti poslednjeg grananja/poziva tog bloka. Za svaku granu, prikazuje se

procenat odabira te grane u odnosu na ukupan broj prolazaka kroz celo grananje. U slučaju poziva, ta pokrivenost je uglavnom 0/100%, odnosno izvršeno/neizvršeno.

Dodavanjem opcije -a, alatu se sugeriše generisanje podataka o pokrivenosti za svaki osnovni blok ponaosob i njihov ispis ispod poslednje naredbe bloka, kao u okviru listinga 3.3.

```

-: 0:Source:paskal.c
-: 0:Graph:paskal.gcno
-: 0:Data:paskal.gcda
-: 0:Runs:1
-: 0:Programs:1
-: 1:#include <stdio.h>
-: 2:
1: 3:int main(int argc, char** argv){
1: 3-block 0
1: 4:     int velicina=atoi(argv[1]),vrednost=1,razmak=0,i,j;
1: 4-block 0
1: 5:     printf("Paskalov trougao sa %d redova: ",velicina);
6: 6:     for(i=0; i<velicina; i++){
6: 6-block 0
20: 7:     for(razmak=1; razmak<=velicina-i; razmak++)
5: 7-block 0
20: 7-block 1
15: 8:         printf(" ");
15: 8-block 0
15: 9:         for(j=0; j<i; j++){
5: 9-block 0
15: 9-block 1
10: 10:             if(j==0 || i==0)
10: 10-block 0
6: 10-block 1
4: 11:             vrednost=1;
4: 11-block 0
-: 12:             else
6: 13:             vrednost=vrednost*(i-j+1)/j;
6: 13-block 0
10: 14:             printf("%2d", vrednost);
10: 14-block 0
-: 15:     }
5: 16:     printf("\n");
5: 16-block 0
-: 17:     }
1: 18:     return 0;
1: 18-block 0
-: 19:}

```

Listing 3.3: Primer izveštaja koji generiše alat *gcov* sa opcijom -a

Ukoliko je linija sadržana u više pojedinačnih blokova kao poslednja, sadržaće više podataka o pokrivenosti, tačno jedan za svaki takav blok.

3.2 Nedostaci ponuđenog rešenja

Proučavanjem karakteristika implementacije instrumentalizacije u okviru programskog prevodioca *GCC* i mogućnosti određivanja pokrivenosti koda na osnovu te instrumentalizacije, standardnim alatom iz kolekcije *GNU*, po imenu *gcov*, otkrivena su tri velika nedostatka:

1. Podaci su dostupni tek po završetku rada programa
2. Korišćenje neoptimalnog statičkog linkovanja
3. Korisnički interfejs standardnog alata iz kolekcije *GNU* 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 (`atexit`). Usled toga, iako analiza alatom *gcov* nije striktno vezana za vremenski tok izvršavanja, ne može se vršiti pre kraja programa. Ovo je veliki nedostatak, koji u nekim specifičnim slučajevima može potpuno 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 *Clang*. Za obezbeđivanje informacija o pokrivenosti koda ovakvih programa, neophodno je proširiti mogućnosti instrumentalizacije implementirane u okviru programskog prevodioca *GCC* na prikupljanje podataka u toku izvršavanja.

Statičko linkovanje

Biblioteka *libgcov* je u okviru programskog prevodioca *GCC* implementirana kao statička biblioteka (arhiva). Upotreba statičkih biblioteka nije optimalno rešenje [15], 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. Neke dodatne informacije koje se dobijaju dodavanjem opcija u poziv alata, poput onih o pokrivenosti pojedinačnih funkcija, 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 interfejsu za vizuelni prikaz *gcov* statistike, koji je izgrađen u okviru ovog projekta.

3.3 Ideje za unapređenje

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 efikasno 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 koje prima *gcov*, 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 u pogledu vremena izvršavanja, 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 zatvorenosti interfejsa biblioteke prema potencijalnim korisnicima. Celokupna funkcionalnost je definisana tako da se odvija bez posredovanja vlasnika instrumentalizovanog programa. Ukoliko bi kontrola 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 programskog prevodioca *GCC*.

Dinamičko linkovanje znatno znatno redukuje vreme potrebno za prevođenje, kao i potreban memorijski prostor [15]. Operativni sistem može smestiti kôd dinamičke biblioteke u segmente *RAM* memorije 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 *gda* 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 operativnih sistema koji podržavaju signale, implementirana je jedna dodatna pogodnost: ispisivanje podataka u fajlove *gda* pomoću signala *SIGUSR1*. Time se postiže da korisnik nema obavezu da svoj izvorni kôd prilagođava instrumentalizaciji. 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 instrumentizovanog 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 grafičkog interfejsa 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. Interfejs 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.

Posebno, za programe koji se izvršavaju na operativnom sistemu *Unix* ili njemu sličnom i koji ne koriste signal `SIGUSR1`, novi grafički interfejs pruža mogućnost i prikupljanja podataka. Time je ukupna funkcionalnost instrumentalizacije dostupna u okviru jedinstvenog grafičkog korisničkog interfejsa. U suprotnom je, za korišćenje novog grafičkog interfejsa, preduslov imati već kreirane fajlove *gceda*.

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 programskog prevodioca *GCC*, čija osnovna funkcionalnost predstavlja prikupljanje podataka iz izvršavanja programa pre završetka njegovog rada.
2. Implementacija novog grafičkog interfejsa za prikaz podataka, u kome je bezbedna i višegodišnjim korisničkim iskustvom potvrđena osnova, alat *gcov*, nadograđena jednostavnim, preglednim, korisnički prilagođenom interfejsom sa povišenom informativnošću.

U narednom tekstu biće detaljno opisane implementacije obe celine, a celokupan kôd potreban za njihovu izgradnju je dostupan u okviru *git* repozitorijuma na adresi: <https://github.com/MarinaNikolic/MASTER/tree/master/SRC>

4.1 Biblioteka *libcoverage*

Implementacija biblioteke *libcoverage* je izvršena u programskom jeziku C. Uzor je predstavljala postojeća biblioteka *libgcov*, podrazumevana biblioteka za prikupljanje podataka iz izvršavanja programa u okviru programskog prevodioca *GCC*. Implementacija je izvršena u četiri faze:

1. konstrukcija baze biblioteke kao potrebnog i dovoljnog skupa funkcija iz koda prevodioca i biblioteke *libgcov* za prikupljanje podataka,

2. modifikacija vremenskog parametra funkcionalnosti prikupljanja podataka iz izvršavanja, u cilju dostupnosti podataka u toku rada programa,
3. ukidanje zavisnosti biblioteke od prevodioca,
4. vremenska i prostorna optimizacija rešenja.

Osnovna funkcionalnost biblioteke *libgcov* se vrši u okviru funkcije: `gcov_exit`. Njen algoritam se sastoji iz iterativnih i dubinskih prolazaka kroz instrumentalizaciju strukture (prikazane na slici 3.1) u cilju prikupljanja podatka i iz procesa upisivanja tih podataka u binarne fajlove sa ekstenzijom *gcda*. 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 zastavica 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 prethodnoj glavi, 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 prevodioca *GCC*, 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 zastavicama, 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`. 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*.

Implementirana je i mogućnost poziva korišćenjem signala *SIGUSR1*. Funkcionalnost je uslovljena navođenjem dodatnog objektnog fajla u fazi linkovanja: *coverage_registration.o*. Kôd, čijim prevođenjem nastaje ovaj objektni fajl, prikazan je u okviru listinga 4.1.

```
#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(COVERAGE_SIGNAL, coverage_handler);
}
```

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

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 ¹. Korišćenje ove metode je opciono i zavisi isključivo od želje korisnika i mogućnosti korišćenja signala datog programa.

Pored osnovnih izmena, izvršene su i dve manje modifikacije funkcije za prikupljanje podataka i kreiranje fajlova *gcda*. Konstanta `GCOV_LOCKED`, definisana preprocesorskom 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 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.

¹Navođenjem jednog od specijalnih atributa: konstruktor funkcija (`__attribute__((constructor))`) / destruktor funkcija (`__attribute__((destructor))`), u okviru deklaracije funkcije, uslovljava se njeno pozivanje pre/nakon funkcije `main()`. To se postiže tako što se funkcije, označene na ovaj način, smeštaju tokom prevođenja u posebnu sekciju objektnog fajla, po imenu `.ctors` / `.dtors`.

Ukidanje zavisnosti biblioteke od prevodioca, odnosno omogućavanje njenog samostalnog prevođenja, prouzrokovalo je da biblioteka izgubi informaciju o verziji prevodioca. Različitost implementacije instrumentalizacije 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 treće faze implementacije, izvrše i 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 instrumentalizacionih struktura. Prilagođavanje biblioteke radu sa strukturama definisanim na stari način, iziskivalo bi izgradnju potpuno nove biblioteke, što predstavlja nepotreban napor. Takođe, aktuelna verzija prevodioca *GCC* 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 prevodioca *GCC*, 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 u okviru listinga 4.2.

```
#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
```

Listing 4.2: Definisavanje 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 glavne kontrolne sume programa se obavljalo, na početku izvršavanja, u okviru funkcije `__gcov_init`. Korišćen je mehanizam pod nazivom: ciklična provera pariteta bloka

[18], nad statičkim podacima programa (imena instrumentalizovanih fajlova izvornog koda). Od verzije 4.7.0, uveden je novi algoritam, po kome se ova kontrolna suma računa u toku čitanja podataka iz instrumentalizacionih struktura, u okviru funkcije `gcov_exit`. Mehanizam ciklične provere pariteta bloka je zadržan, ali je skup podataka potpuno izmenjen. Umesto imena fajlova, koriste se drugi statički podaci, kao što je broj funkcija, a uključeni su i dinamički podaci, odnosno vrednosti brojača. Eventualna oštećenja podataka će sa dovoljno velikom verovatnoćom biti detektovana u kontrolnim sumama pojedinačnih entiteta, pre svega funkcija, pa dinamičko računanje kontrolne sume programa 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-bitnu 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 memorijski 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 optimalna 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 instrumentalizacione 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 gornjem delu listinga 4.3, korišćen je pravolinijski pristup prikazan u donjem delu iste slike.

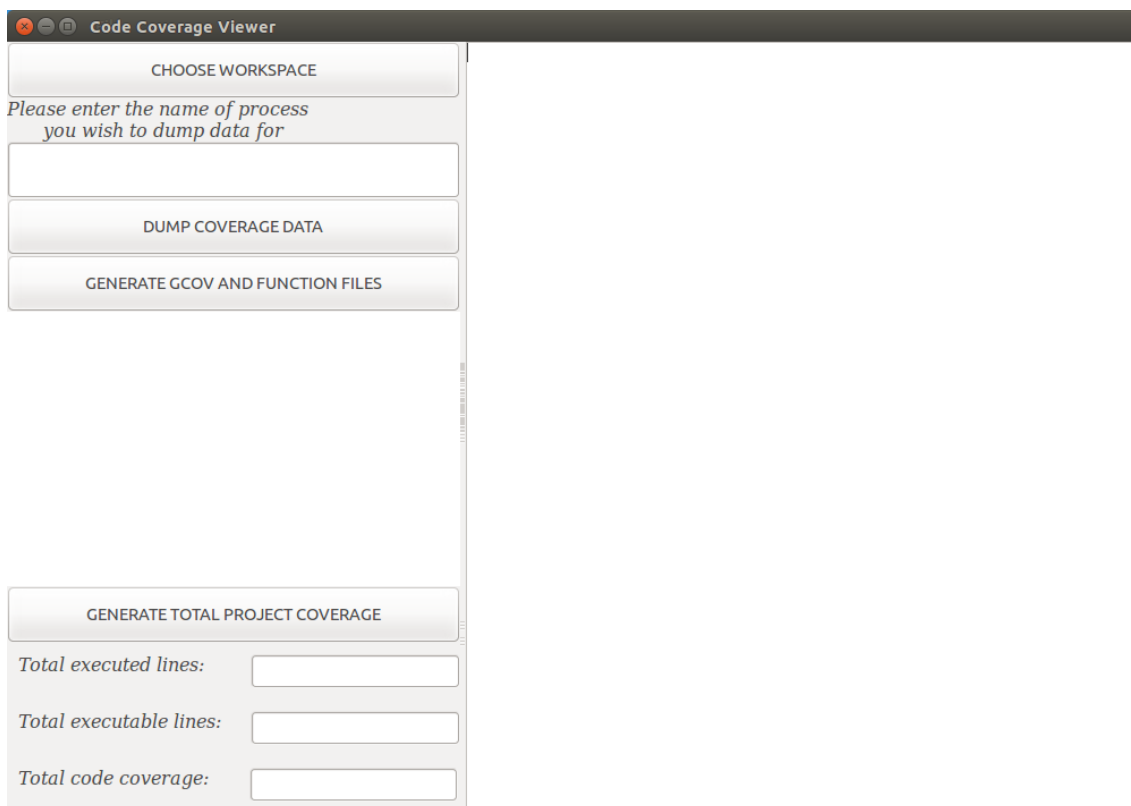
```
definicija: gfi_ptr = gi_ptr->functions[f_ix];
definicija: ci_ptr = gfi_ptr->ctrs;
pristup:    ci_ptr->values;
==>
pristup:    gi_ptr->functions[f_ix]->ci_ptr->values
```

Listing 4.3: Pristup instrumentalizacionim strukturama

4.2 Korisnički interfejs

Implementacija novog grafičkog interfejsa 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 interfejsa jeste da predstavlja jednostavni, pregledni, korisniku prilagođen omotač za celokupan proces prikupljanja i prikaza podataka u toku izvršavanja programa. Usled pružanja dodatnih informacija o statistici pojedinačnih modula i/ili celokupnog projekta, *code_coverage_viewer* je i više od interfejsa.

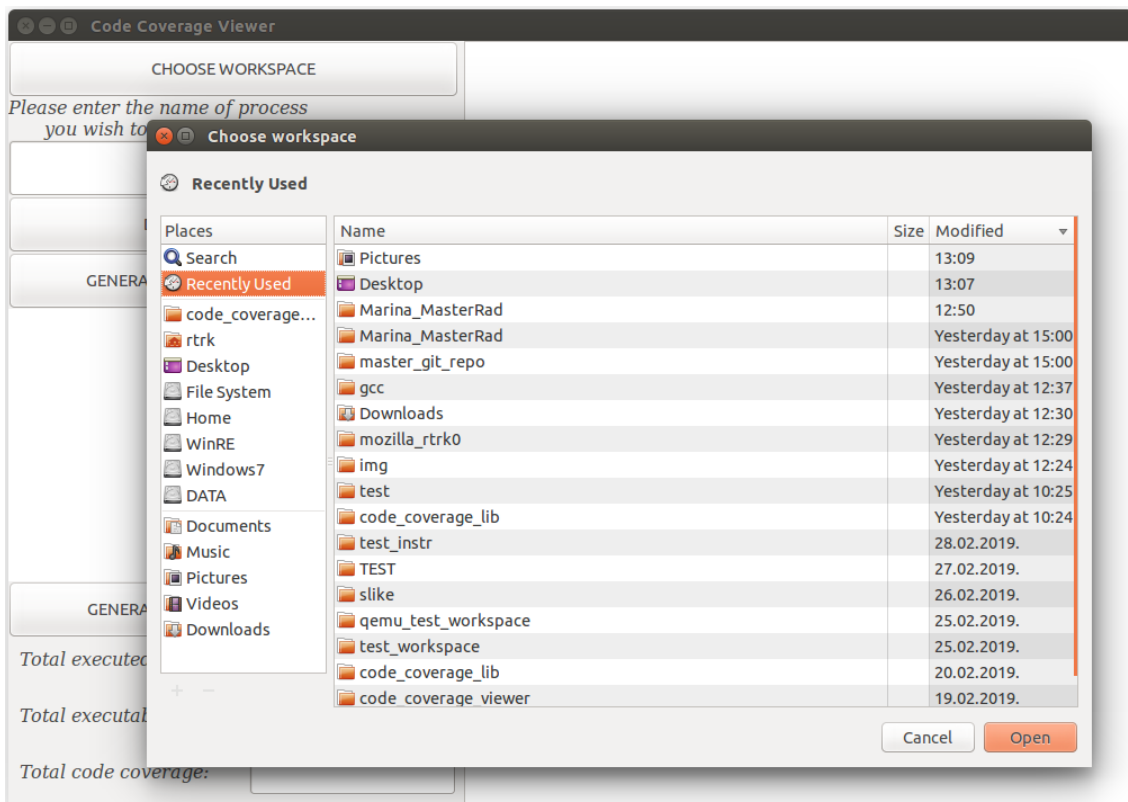
Grafički korisnički interfejs se sastoji iz dve celine: kontrolne i prezentacione, i jedne linije menija. Celine su implemenatciono predstavljene sa dva *skroll*-ujuća panela (`wx.Panel`), dobijena pomoću vertikalnog splitera ekrana (`wx.SplitterWindow`), dok je za kreiranje menija korišćena klasa `wx.MenuBar`. Na slici 4.1 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 4.1: Grafički interfejs po pokretanju

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 interfejsa.



Slika 4.2: Odabir radnog direktorijuma u grafičkom interfejsu

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 nazvom: `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 pokreće već nakon upisa imena, već je potreban klik 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 u slučaju omogućenog prikupljanja podataka putem signala. Ograničenje je napomenuto u specifikaciji grafičkog korisničkog interfejsa, kao i u okviru prozora za pomoć koji se otvara klikom na *Help*.

Izvršavanje sistemske komanade `kill -10 <pid>`, kao i svih ostalih sistemskih komandi je implementirano pomoću funkcije `system`, modula `os`. Ovo je najjednostavniji princip, bez bezbednosnih izazova, koji u potpunosti zadovoljava potrebe novog grafičkog interfejsa. Naredne dve komponente predstavljaju interfejs ka funkcionalnosti generisanja izveštaja alatom *gcov* i kontrole njihovih prikazivanja. Dugme: *Generate gcov and function files* pokreće 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 analizom 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.

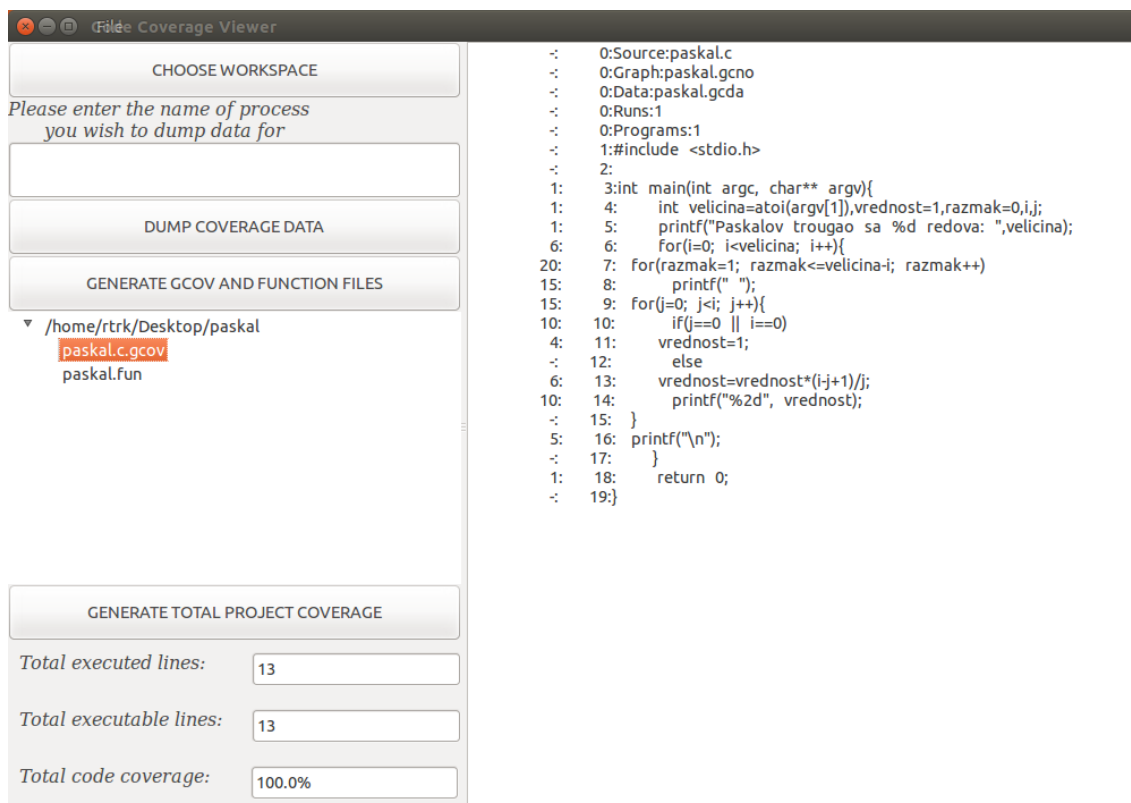
Svi izveštaji se smeštaju u drvoliku strukturu, lociranu ispod dugmeta za njihovo generisanje, koja je implementirana kao instanca klase `wx.TreeCtrl`. Celokupni sadržaj se briše, a drvolika struktura se ponovo izgrađuje, popunjavajući se novim podacima. Ista procedura se vrši, kao što je već napomenuto, 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 pokreće 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* pokreće 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 4.3.

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`, čime se pruža dodatna mogućnost kopiranja sadržaja, menjanja ili brisanja. Prikaz 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 vrednosti funkcionalnosti odabira radnog direktorijuma, ali i omogućilo čitanje fajlova *gcno*.

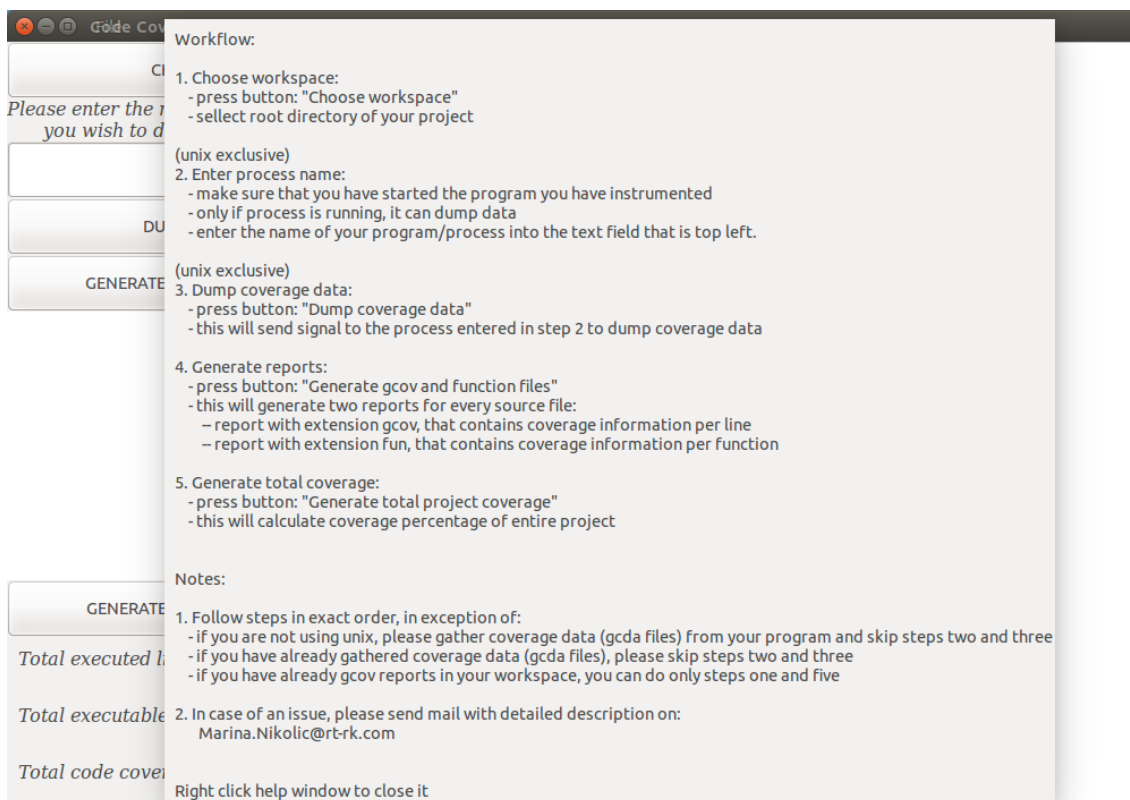
Linija menija je implementirana korišćenjem klase `wx.MenuBar`. Sadrži ime novog interfejsa i jedno dugme, tipa `wx.Menu`, pod nazivom *File*. Jednim klikom na dugme *File*, pojavljuje se lista sa dve opcije: *Help* i *Quit*. Odabiranjem opcije za pomoć, na centru ekrana se pojavljuje novi prozor, klase `wx.PopupWindow`, kao na slici 4.4. Prozor se sastoji iz jedinstvenog panela, čiju celokupnu površinu zauzima



Slika 4.3: Generisana celokupna statistika prikazana u grafičkom interfejsu

tekstualno polje, statičke prirode, sa kratkim uputstvom za upotrebu grafičkog interfejsa. 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 interfejsa.

U okviru specifikacije novog grafičkog interfejsa, kao i u okviru prozora za pružanje pomoći, detaljno je opisana pravilna upotreba. 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 grafičkog interfejsa, čime su pokriveni i slučajevi parcijalne upotrebe, kao na primer, pregledanje već generisanih izveštaja. Takođe, tom prilikom se korisniku otvara i



Slika 4.4: Prozor za pomoć u grafičkom interfejsu

mali prozor za dijalog, instanca klase `wx.MessageDialog`, pomoću koga se informiše o konkretnoj grešci u proceduri i dobija savet o adekvatnom sledećem koraku.

Glava 5

Verifikacija i validacija implementiranog rešenja

U cilju procene kvaliteta implementiranog rešenja, sprovedeni su različiti procesi analize koda radi proučavanja karakteristika od interesa, testiranja korektnosti i procene performansi. U narednom tekstu biće prezentovani njihovi mehanizmi i rezultati.

5.1 Analiza karakteristika

U okviru analize performansi softverskog rešenja za prikupljanje i prikaz podataka o pokrivenosti koda u toku izvršavanja, ispitivanje je sprovedeno prema tri kriterijuma:

1. Bezbednost
2. Algoritamska složenost
3. Jednostavnost upotrebe

Bezbednost

Bezbednost predstavlja jednu od najvažnijih karakteristika softvera kao proizvoda, naročito kada je reč o višenamenskim rešenjima, kao što je slučaj sa bibliotekom *libc*. Od nivoa ispunjenosti ovog kriterijuma, direktno zavisi i dijapazon potencijalne upotrebe softverskog rešenja. U okruženja koja predstavljaju naročiti bezbednosni izazov svrstavamo najpre one sisteme od čije ispravne funkcionalnosti

zavise ljudski životi, ekonomski značajni resursi i poverljivost važnih informacija. Ispitivanje bezbednosti softvera izvršeno je manuelnim tehnikama statičke analize, u kontekstu ispravnosti upravljanja memorijskim resursima i mogućnosti neovlašćenog pristupa. Dodatno, primenjena je tehnika dinamičke analize, pod nazivom profajliranje, u cilju eksperimentalnog potvrđivanja donetih zaključaka o ispravnosti upravljanja memorijskim resursima.

Analizom izvornog koda biblioteke *libcoverage*, utvrđeno je da je ukupna dinamički alocirana memorija dodeljena za smeštanje sadržaja promenljive `buffer` u okviru strukture `gcov_var`, zbog čega je i potpuno oslobođena naredbom: `free(gcov_var.buffer)` pred kraj funkcije `drew_coverage`. Za dodatno potvrđivanje ove tvrdnje, odnosno detektovanje eventualnog nepravilnog upravljanja memorijom, iskorišćen je standardni alat za memorijsko profajliranje iz kolekcije *Valgrind*. Ovaj alat nema mogućnost ispitivanja biblioteke, kao neizvršne jedinice sistema, zbog čega je bilo neophodno analizu sprovesti nad eksternim izvršnim programom. Korektnost rezultata je postignuta korišćenjem jednostavnog programa za računanje srećnog broja, opisanog u okviru pogavlja 5.2, čija je bezbednost unapred poznata usled činjenice da ne koristi dinamičko alociranje memorije. Proizvoljnost odabira je opravdana nezavisnošću izvršavanja dela koda kojim se alocira/oslobađa memorija od vrste programa koji se instrumentalizuje. Dinamičko alociranje memorije je uslovljeno prisustvom jedinstvene izvršne linije koda i isključivo se vrši realokacija jedinstvene promenljive. Oslobođanje se vrši bezuslovno, nakon završetka svih funkcija koje rekurzivno mogu dovesti do novih rezervacija memorije. Porukom: „*ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)*”, pokazano je da biblioteka ne prouzrokuje curenje memorije, pisanje van alociranog prostora, niti slične propuste u ispitanom slučaju.

Neovlašćeni pristup podacima programa je sprečen prirodom rešenja za prikupljanje i prikaz podataka. Implementacija softverskog rešenja za prikupljanje u vidu biblioteke, nasuprot eksternog alata, čini program jedinim vlasnikom podataka sopstvene deljene memorije. Rešenje koje za transport i obradu podataka ne koristi internet, niti bilo koju drugu mrežu, je dodatno otporno na napade eksterne prirode. Time je ispunjen bezbednosni aspekt zaštite podataka od neovlašćenog pristupa.

Algoritamska složenost

Algoritam prikupljanja podataka iz deljene memorije programa nije promenjen u odnosu na postojeću implementaciju. Optimizacija ispisa u fajlove *gcda* je za-

visna od ulaznih podataka i okruženja instrumentalizovanog programa i ne menja kvantitet jedinica za obradu, te ne utiče na procenu algoritamske složenosti. Ekvivalencija vremenske i prostorne zahtevnosti procesa prikupljanja podataka o pokrivenosti koda sa postojećom bibliotekom *libgcov* se može postići pozivom funkcije `drew_coverage` kao argumenta `atexit` funkcije. Zbog toga, složenost kao karakteristika najzahtevnijeg slučaja, ostaje nepromenjena. Ranijim pozivom funkcije za prikupljanje podataka, smanjuje se količina ulaznih podataka algoritama, što u velikim sistemima, sa velikom količinom koda može dovesti do smanjenja potrebnih vremenskih i prostornih resursa.

Kao ilustrativan primer mogu se posmatrati dve komponente: program *dinamičko_ucitavanje*, čiji je izvorni kôd prikazan u okviru listinga 5.1 i jedna proizvoljna deljena biblioteka, koja će biti oslovljavana sa *libprimer.so*. Neka su obe komponente instrumentalizovane pomoću odgovarajućih zastavica i biblioteke *libcoverage.so*. Ukoliko ukolonimo prvu oznaku za komentar, ponovo prevedemo program i pokrenemo, poziv funkcije `drew_coverage` će se obaviti nakon dinamičkog učitavanja biblioteke *libprimer.so* i izvršavanja jedne njene funkcije. U tom slučaju, prikupljanje podataka vršiće se za obe komponente, što podrazumeva prolazak kroz dva skupa instrumentalizacionih struktura i kreiranje bar dva fajla sa ekstenzijom *gcda* (jedan za program i onliko za biblioteku, koliko fajlova izvornog koda je sačinjava). Generisanjem izveštaja alatom *gcov*, može se proveriti prisutnost podataka iz izvršavanja biblioteke. Sa druge strane, analognim uklanjanjem druge oznake za komentar, poziv funkcije `drew_coverage` će se obaviti pre dinamičkog učitavanja biblioteke, usled čega se njene instrumentalizacione strukture neće ni inicijalizovati (ovo tvrđenje je jednostavno proveriti dodavanjem proizvoljnog ispisa u funkciju `__gcov_init` biblioteke *libcoverage.so*). Na ovaj način, ostvarena je ušteda, kako vremena, tako i prostora. Značaj uštede se ne može videti na primeru ove veličine, ali se na osnovu njega može pretpostaviti, da bi ušteda na velikom sistemu bila značajna.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dlfcn.h>
#include "coverage.h"

int main(int argc, char** argv)
{

    printf("Dobrodošli u program za dinamičko učitavanje biblioteka\n");
    printf("Molim vas unesite punu putanju do vaše biblioteke\n");
    //drew_coverage();
    char biblioteka[256];
    scanf("%s", biblioteka);
    printf("Molim vas unesite ime zeljene funkcije iz prethodno unesene
    biblioteke\n");
    printf("Napomena: potpis zeljene funkcije mora biti: void function_name
    ();\n");
    char funkcija[256];
    scanf("%s", funkcija);

    printf("DEBUG\n");

    void *handle;
    void (*func)();

    handle = dlopen(biblioteka, RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "Greska prilikom učitivanja biblioteke %s: %s\n",
        biblioteka, dlerror());
        return 1;
    }

    *(void**>(&func) = dlsym(handle, funkcija);
    if (!func) {
        fprintf(stderr, "Greska prilikom učitivanja funkcije %s: %s\n", funkcija,
        dlerror());
        dlclose(handle);
        return 1;
    }

    func();
    //drew_coverage();
    dlclose(handle);

    return EXIT_SUCCESS;
}
```

Listing 5.1: Izvorni kod: *dinamicko_ucitavanje.c*

Do uštede vremena u kontekstu utroška na izvršavanje samog instrumentalizovanog programa, se može doći u slučaju kodova sa visokim procentom dugih, repetativnih jedinica, pod pretpostavkom da postoji interesovanje samo za procenat pokrivenosti koda. Izvršavanje pojedinačne linije više od jednog puta, nema uticaj na računanje pokrivenosti koda, već samo na pojedinačne izveštaje. Za postizanje stoprocentne pokrivenosti, teorijski je dovoljno izvršiti svaku liniju samo jednom. Stoga prekidanje prikupljanja podataka pre kraja izvršavanja može u ovakvim slučajevima dati jednako korektne rezultate u kraćem vremenskom roku. Kao posledica, nastaje novi oblik testiranja programa, koji možemo nazvati: *testiranje vođeno pokrivenošću*.

Opis testiranja vođenog pokrivenošću biće dat kroz naredni ilustrativan primer. Program koji će biti posmatran predstavlja server za pružanje informacija iz neke baze podataka. U zavisnosti od klijenta, server izvršava različite *SQL* upite nad bazom, obrađuje rezultat i informiše klijenta. Pokretanjem servera izvršava se inicijalni deo koda, nakon čega serverska aplikacija prelazi u stanje čekanja. Ukupan broj izvršnih linija koda servera možemo označiti sa N , a broj izvršnih linija u segmentu koda za prekid rada servera sa M . Kôd zadužen za pokretanje aplikacije i obradu svih različitih zahteva stoga ima $N-M$ izvršnih linija, te ćemo oceniti ciljanu pokrivenost na $(N-M)/100$ procenata. Cilj testiranja predstavlja pokrivanje svih slučajeva upotrebe i analiza ponašanja aplikacije. Ideja testiranja vođenog pokrivenošću jeste izvišvanje minimalnog skupa testova za postizanje tog cilja, gde test u ovom primeru predstavlja obradu jednog klijenta. Nakon određenog broja pokretanja testova, signalizira se serveru da pozove funkciju `drew_coverage`. Pomoću grafičkog korisničkog interfejsa `code_coverage_viewer`, izračunava se ukupna pokrivenost i generišu pojedinačni izveštaji. Sve dok je ukupna pokrivenost manja od $(N-M)/100$, postoji slučaj upotrebe koji nije testiran. Na osnovu izveštaja i trenutnog stanja serverske aplikacije, može se lako locirati kôd koji nije izvršen i u narednom pokušaju inicirati i test koji vrši tu funkcionalnost.

Jednostavnost upotrebe

Biblioteka `libcoverage` se koristi na standardan način, karakterističan za sve dinamičke biblioteke. Osnovni slučaj upotrebe predstavlja uključivanje zaglavlja `coverage.h` ili deklarisanje funkcije `drew_coverage` kao eksterne, kao i poziv funkcije u željenom delu koda programa. Prilikom prevodenja do objektnih fajlova, navode se zastavice za instrumentalizaciju: `-fprofile-arcs` i `-ftest-coverage`, dok se u

fazi linkovanja navodi sama biblioteka: `-L<putanja> -lcoverage`.

Napredni slučaj upotrebe predstavlja korišćenje signala kao okidača za poziv funkcije, i dostupno je samo korisnicima operativnih sistema sa ugrađenom podrškom za signale. Prilikom prevođenja do objektnih fajlova, navode se zastavice za instrumentalizaciju: `-fprofile-arcs` i `-ftest-coverage`, dok se u fazi linkovanja navodi sama biblioteka: `-L<putanja> -lcoverage`, kao i objektni fajl sa kodom za registraciju signala: `<putanja>/coverage_registration.o`. Poziv funkcije `drew_coverage` se inicira izvršavanjem komande: `kill -10 <pidProcesa>` iz terminala. Ukoliko je signal `SIGUSR1` predefinisano u korisničkom programu, a korisnik želi da koristi napredni vid upotrebe, potrebno je da sam implementira ovaj mehanizam u svom programu. Jedan način biće detaljno opisan u kasnijem tekstu, u okviru opisa testiranja.

Intuitivnost i jednostavnost upotrebe novog grafičkog interfejsa za prikaz podataka o pokrivenosti koda tokom izvršavanja, postignuta je kreiranjem kratkog i informativnog uputstva za upotrebu, kao i opisnim nazivima komponenti i njihovim specifičnim redosledom. Na osnovu rezultata neformalne ankete, sprovedene u okviru razvojnog tima projekta u koji je ovo softversko rešenje integrisano, može se pretpostaviti da je za ispravnu upotrebu dovoljno pročitati uputstvo iz opcije za pomoć svega jednom. Time je zadovoljen kriterijum jednostavnosti upotrebe.

5.2 Testiranje korektnosti i performansi

Testiranje validnosti i performansi softverskog rešenja za prikupljanje i prikaz podataka o pokrivenosti koda u toku izvršavanja je originalno sprovedeno nad dva veća softvera izrađena za potrebe digitalne televizije, u okviru kojih se još uvek koristi. Usled zatvorenosti koda ovih softvera, ti rezultati neće biti prikazani u ovom radu. Za potrebe demonstracije rada softverskog rešenja i rezultata testiranja, biće iskorišćena dva pogodnija primera. Najpre će biti demonstrirana korektnost na manjem, preglednijem primeru. Demonstracija mogućnosti realne primene kao i analize performansi u komercijalnim projektima biće izvršene na znatno većem projektu otvorenog koda, pod nazivom QEMU [16].

Jednostavni primer: generator srećnog broja

Demonstracija korišćenja biblioteke *libcoverage* i interfejsa *code_coverage_viewer* biće sprovedena nad jednostavnim primerom koji se sastoji od dva fajla izvornog koda sa šest funkcija, tri naredbe grananja i pet petlji. Poslednja petlja će biti beskonačna, u cilju bolje demonstracije primene prikupljanja koda u toku izvršavanja.

Opis koda primera

Program *lucky* računa srećan broj na osnovu datuma rođenja, rekurzivnim sabiranjem cifara do jednocifrenog broja. Sastoji se iz dva fajla izvornog koda, pisanih u programskom jeziku C: *main.c* (listing 5.2) i *lucky.c* (listing 5.3).

U okviru fajla *main.c* nalazi se glavna funkcija programa koja čita datum rođenja sa standardnog ulaza, proverava validnost, poziva funkciju za računanje srećnog broja i ispisuje rezultat na standardni izlaz. Osnovna funkcionalnost se vrši u okviru beskonačne `while` petlje, čija je jedinica prolaska jedno računanje srećnog broja. Na početku `while` petlje, implementiran je i mehanizam za prekid izvršavanja. Pored glavne funkcije, fajl *main.c*, sadrži još i definicije funkcija za ispisivanje poruka dolaznog i odlaznog pozdrava, koje se pozivaju na početku i na kraju izvršavanja.

Fajl *lucky.c* sadrži definicije tri funkcije: jedne osnovne i dve pomoćne. Osnovna funkcija: `calculate_lucky_number` se poziva iz glavne funkcije programa. Kao ulazne podatke prima tri celobrojne vrednosti za datum rođenja. Zatim računa srećan broj pomoćnom funkcijom za pretvaranje broja u jednocifreni zbir cifara: `to_digit`. Povratna vrednost predstavlja jednu celobrojnu vrednost, odnosno izračunati srećni broj. Druga pomoćna funkcija služi za proveru korektnosti unetog datuma rođenja, poziva se takođe iz glavne funkcije programa i uslovljava izvršavanje funkcije `calculate_lucky_number`.

```
#include <stdio.h>

void greeting(){
    printf("Hello, I am a program that calculates your lucky number\n");
}

void farewell(){
    printf("Goodbye\n");
}

int main(){
    greeting();
    while(1){
        printf("Please, enter y/n to proceed/end program\n");
        char c;
        scanf("\n%c", &c);
        if(c == 'n'){
            break;
        }
        else if(c == 'y'){
            int date, month, year;
            printf("Enter your year of birth: ");
            scanf("%d", &year);
            while(year<1900 || year>3000){
                printf("Please enter year between 1900 and 3000: ");
                scanf("%d", &year);
            }
            printf("Enter your month of birth: ");
            scanf("%d", &month);
            while(month<1 || month>12){
                printf("Please enter month between 1 and 12: ");
                scanf("%d", &month);
            }
            printf("Enter your date of birth: ");
            scanf("%d", &date);
            while(is_date_incorrect(date, month, year)){
                printf("Please enter correct date of %d. month and %d. year: ", \
                    month, year);
                scanf("%d", &date);
            }
            int lucky = calculate_lucky_number(date, month, year);
            printf("Your lucky number is: %d\n", lucky);
        }
    }

    farewell();
    return 0;
}
```

Listing 5.2: Izvorni kod: *main.c*

```
int is_date_incorrect(int date, int month, int year){
    if (month==2 && (date<1 || date >29))
        return 1;
    else if ((month==4 || month==6 || month==9 || month==11) && \
            (date<1 || date >30))
        return 1;
    else if ((month==1 || month==3 || month==5 || month==7 || \
            month==8 || month==10 || month==12) \
            && (date<1 || date >31))
        return 1;
    else if (((year%400)!=0 && ((year%4)!=0 || (year%100)==0)) && \
            month==2 && date==29)
        return 1;
    else
        return 0;
}

int to_digit(int value){
    if (value < 10)
        return value;
    else{
        int sum = 0;
        int value = value;
        int leftover = 0;
        while (quotient != 0){
            leftover = quotient % 10;
            sum = sum + leftover;
            quotient = quotient / 10;
        }
        return to_digit(sum);
    }
}

int calculate_lucky_number(int day, int month, int year){
    int day_code = to_digit(day);
    int month_code = to_digit(month);
    int year_code = to_digit(year);
    int sum_codes = day_code + month_code + year_code;
    int final_code = to_digit(sum_codes);
    return final_code;
}
```

Listing 5.3: Izvorni kod: *lucky.c*

Plan testiranja

Za potrebe testiranja, kreiraju se tri verzije programa lucky:

1. *lucky-vers1* – bez instrumentalizacije

2. *lucky-vers2* – sa instrumentalizacijom i standardnom bibliotekom *libgcov*
3. *lucky-vers3* – sa instrumentalizacijom i novom bibliotekom *libcoverage*.

Plan za testiranje je definisan sledećim koracima:

1. prevođenje programa:

- a) verzija bez instrumentalizacije se prevodi na uobičajeni način, sledećim komandama:

```
gcc -c lucky.c main.c
gcc lucky.o main.o -o lucky-vers1
```

- b) verzija sa instrumentalizacijom i standardnom bibliotekom *libgcov* se prevodi sa dodatnim zastavicama `-fprofile-arcs` i `-ftest-coverage` i u prvom koraku za instrumentalizaciju, a i u drugom za definisanje korišćenja standardne biblioteke *libgcov*:

```
gcc -c lucky.c main.c -fprofile-arcs -ftest-coverage
gcc lucky.o main.o -o lucky-vers2 \
    -fprofile-arcs -ftest-coverage
```

- c) verzija sa instrumentalizacijom i novom bibliotekom *libcoverage* se prevodi sa dodatnim zastavicama `-fprofile-arcs` `-ftest-coverage` isključivo u prvom koraku za instrumentalizaciju, dok se linkeru prosleđuju biblioteka *libcoverage* i objektni fajl: *coverage_registration.o*:

```
gcc -c lucky.c main.c -fprofile-arcs -ftest-coverage
gcc lucky.o main.o -o lucky-vers3 \
    -L<putanja do biblioteke libcoverage> -lcoverage \
    <putanja do registratora/coverage_registration.o
```

2. pokretanje programa:

- a) podešavanje okruženja, neophodno samo za pokretanje treće verzije programa:

```
export LD_LIBRARY_PATH=<putanja do biblioteke libcoverage>
```

- b) pokretanje sva tri programa se zatim vrši na standardan način:

```
./lucky-vers1
./lucky-vers2
./lucky-vers3
```

GLAVA 5. VERIFIKACIJA I VALIDACIJA IMPLEMENTIRANOG REŠENJA

3. sprovođenje inicijalnog dela, za sve tri verzije unapred utvrđenog, slučaja upotrebe:
 - a) odabir opcije: `y`
 - b) unos: `1992 <enter> 12 <enter> 17`
 - c) odabir opcije: `y`
 - d) unos: `3001 <enter> 1992 <enter> 15 <enter>`
 - e) unos: `12 <enter> 38 <enter> 17`

4. Sprovođenje završnog dela, za sve tri verzije unapred utvrđenog, slučaja upotrebe:
 - a) program: *lucky-vers1* se prekida odabirom opcije: `n`
 - b) program: *lucky-vers2* se prekida odabirom opcije: `n` i generišu se izveštaji:

```
gcov lucky.c -f > lucky.fun
gcov main.c -f > main.fun
```
 - c) program *lucky-vers3* se ne prekida u cilju demonstracije prikupljanja i prikaza podataka u toku izvršavanja; sprovode se sledeći koraci:
 - i. Pokreće se grafički korisnički interfejs *code_coverage_viewer*
 - ii. Odabira se radni direktorijum gde se nalazi *lucky-vers3*
 - iii. Unosi se ime programa u odgovarajuće tekstualno polje: *lucky-vers3*
 - iv. Klik na dugme: *Dump coverage data*
 - v. Klik na dugme: *Generate gcov and function files*
 - vi. Klik na dugme: *Generate total coverage*

5. validacija:
 - a) validacija ispisa na standardni izlaz; kriterijumi ispunjavanja se definišu na sledeći način:
 - i. sva tri programa imaju identičan ispis iz inicijalnog dela slučaja upotrebe
 - ii. programi *lucky-vers1* i *lucky-vers2* imaju identičan ispis iz završnog dela slučaja upotrebe
 - iii. program *lucky-vers3* nema ispis iz završnog dela slučaja uporebe

- iv. program *lucky-vers3* ima ispise iz biblioteke i registracionog objektnog fajla.
- b) validacija izveštaja; kriterijumi ispunjavanja se definišu na sledeći način:
 - i. izveštaji generisani za fajlove izvornog koda programa: *lucky-vers2* i *lucky-vers3* se poklapaju na inicijalnom delu slučaja upotrebe
 - ii. izveštaji generisani za fajlove izvornog koda programa *lucky-vers3* nemaju pozitivne podatke iz izvišavanja za zavišni deo slučaja upotrebe
- c) validacija ukupne pokrivenosti

Grafička reprezentacija plana testiranja je prikazana na slici 5.1

Sprovođenje testiranja i analiza rezultata

Pre početka testiranja, neophodno je kreirati radno okruženje:

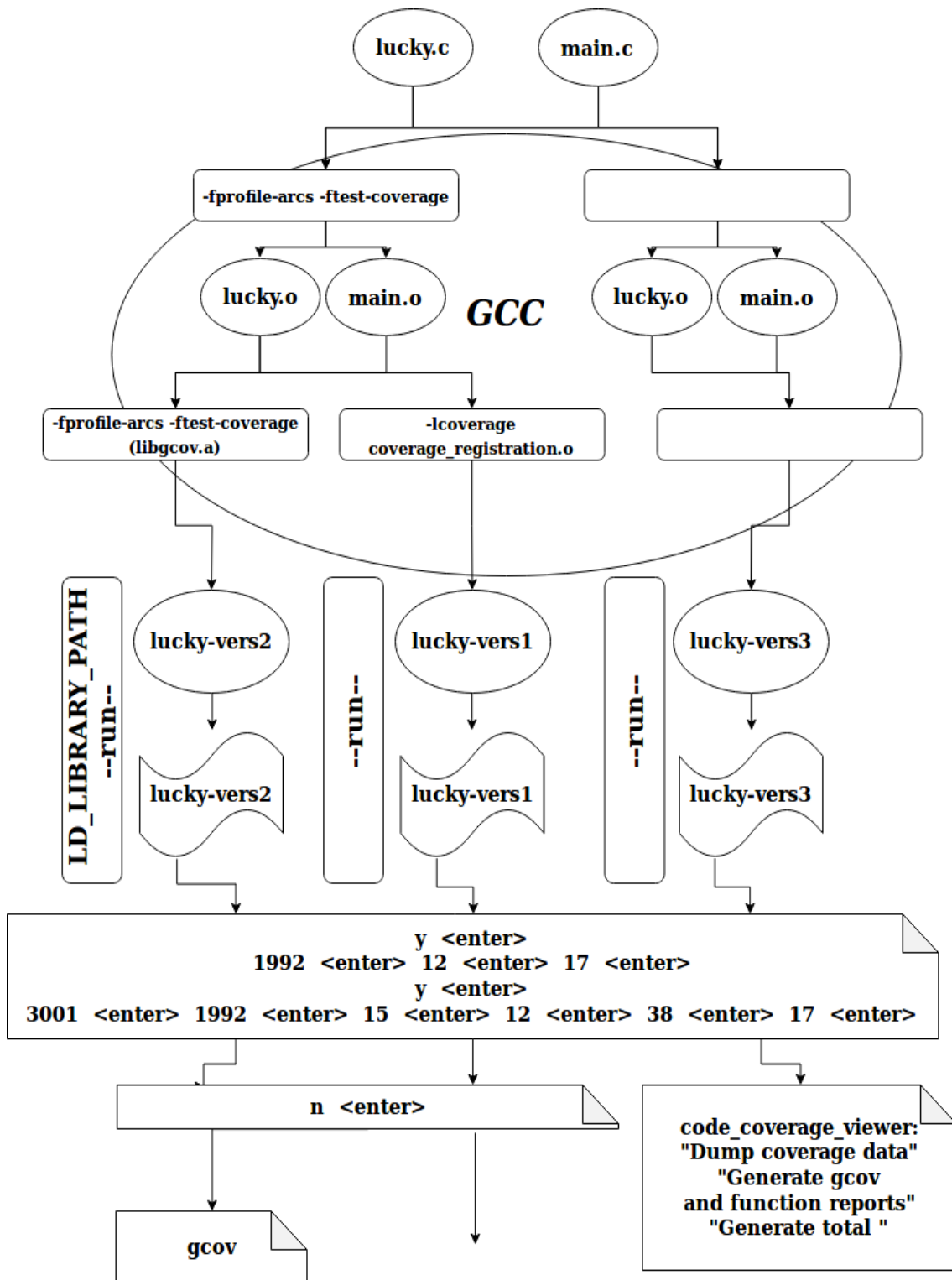
1. kreiranje radnog direktorijuma:

```
mkdir test_workspace
```
2. kreiranje tri bazna direktorijuma sa kodom za različite verzije programa:

```
mkdir test_workspace/version_01
mkdir test_workspace/version_02
mkdir test_workspace/version_03
cp lucky.c main.c test_workspace/version_01
cp lucky.c main.c test_workspace/version_02
cp lucky.c main.c test_workspace/version_03
```
3. kreiranje direktorijuma sa bibliotekom *libcoverage* i registratorom signala:

```
mkdir test_workspace/runtime_coverage
cp libcoverage.so test_workspace/runtime_coverage
cp coverage_registration.o test_workspace/runtime_coverage
cp code_coverage_viewer.py test_workspace/runtime_coverage
```

Prvi korak testiranja obuhvata prevođenje programa na tri načina. Objektni i izvršni fajlove smeštaju se u odgovarajući radni direktorijum konkretne verzije. Za potvrdu uspešnosti ovog koraka, proverava se prisustvo fajlova *gcn0*, komandom *ls* i prisustvo funkcija: *__gcov_init*, *gcov_exit* i *drew_coverage*, komandom *readelf*. Dobijeni rezultati obuhvataju:

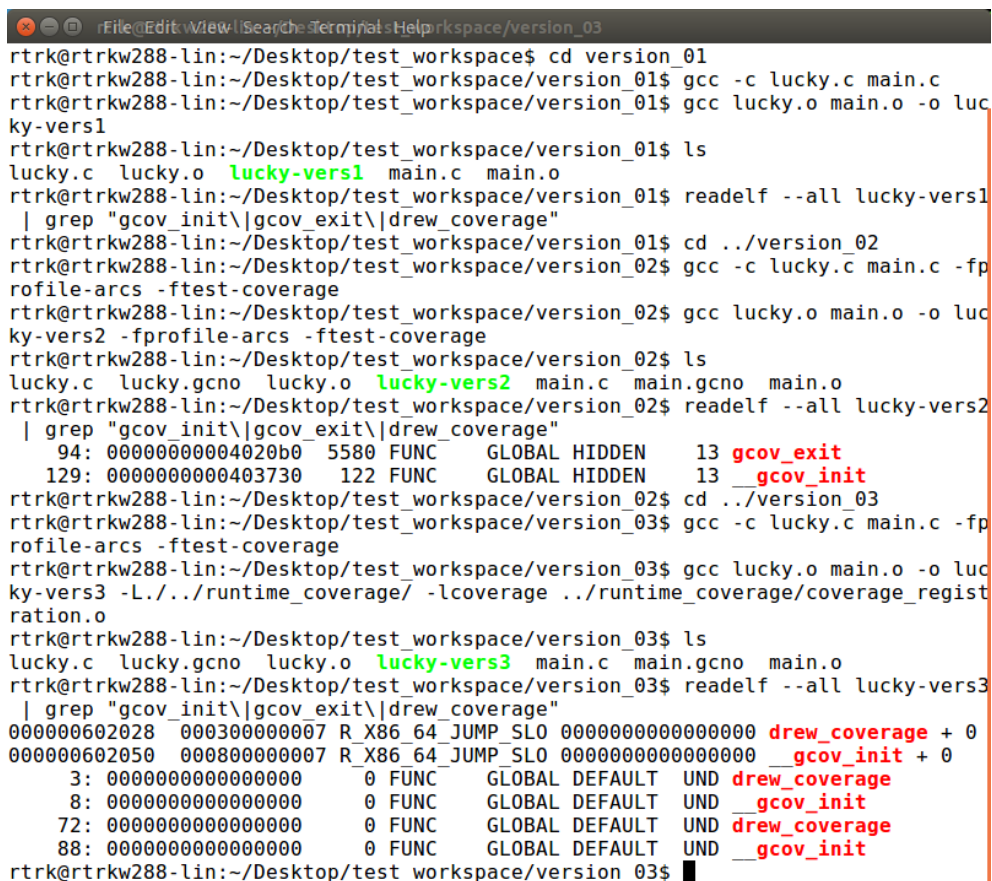


Slika 5.1: Plan testiranja instrumentalizacije programa *lucky*

GLAVA 5. VERIFIKACIJA I VALIDACIJA IMPLEMENTIRANOG REŠENJA

1. prisustvo fajlova: *main.gcno* i *lucky.gcno* isključivo u direktorijumima instrumentalizovanih verzija
2. odsustvo funkcija `__gcov_init`, `drew_coverage` i `gcov_exit` u programu *lucky-vers1*
3. prisustvo funkcija `__gcov_init` i `gcov_exit` i odustvo funkcije `drew_coverage` u programu *lucky-vers2*
4. prisustvo funkcija `__gcov_init` i `drew_coverage` i odustvo funkcije `gcov_exit` u programu *lucky-vers3*

što predstavlja ispravne rezultate. Sprovođenje prvog koraka testiranja i rezultati istog, prikazani su na slici 5.2.



```
rtrk@rtrkw288-lin:~/Desktop/test_workspace$ cd version_01
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_01$ gcc -c lucky.c main.c
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_01$ gcc lucky.o main.o -o lucky-vers1
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_01$ ls
lucky.c lucky.o lucky-vers1 main.c main.o
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_01$ readelf --all lucky-vers1 | grep "gcov_init\|gcov_exit\|drew_coverage"
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_01$ cd ../version_02
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_02$ gcc -c lucky.c main.c -fprofile-arcs -ftest-coverage
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_02$ gcc lucky.o main.o -o lucky-vers2 -fprofile-arcs -ftest-coverage
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_02$ ls
lucky.c lucky.gcno lucky.o lucky-vers2 main.c main.gcno main.o
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_02$ readelf --all lucky-vers2 | grep "gcov_init\|gcov_exit\|drew_coverage"
    94: 000000000004020b0 5580 FUNC GLOBAL HIDDEN 13 gcov_exit
    129: 00000000000403730 122 FUNC GLOBAL HIDDEN 13 gcov_init
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_02$ cd ../version_03
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_03$ gcc -c lucky.c main.c -fprofile-arcs -ftest-coverage
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_03$ gcc lucky.o main.o -o lucky-vers3 -L../runtime_coverage/ -lcovage ../runtime_coverage/coverage_registration.o
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_03$ ls
lucky.c lucky.gcno lucky.o lucky-vers3 main.c main.gcno main.o
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_03$ readelf --all lucky-vers3 | grep "gcov_init\|gcov_exit\|drew_coverage"
000000602028 000300000007 R X86_64_JUMP_SLO 0000000000000000 drew_coverage + 0
000000602050 000800000007 R X86_64_JUMP_SLO 0000000000000000 __gcov_init + 0
    3: 0000000000000000 0 FUNC GLOBAL DEFAULT UND drew_coverage
    8: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __gcov_init
   72: 0000000000000000 0 FUNC GLOBAL DEFAULT UND drew_coverage
   88: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __gcov_init
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_03$
```

Slika 5.2: Testiranje na programu *lucky* - Korak 1

Drugi korak obuhvata pokretanje sva tri programa u tri različita terminala, kako bi se omogućila komunikacija putem standardnog ulaza/izlaza. Za pokretanje trećeg

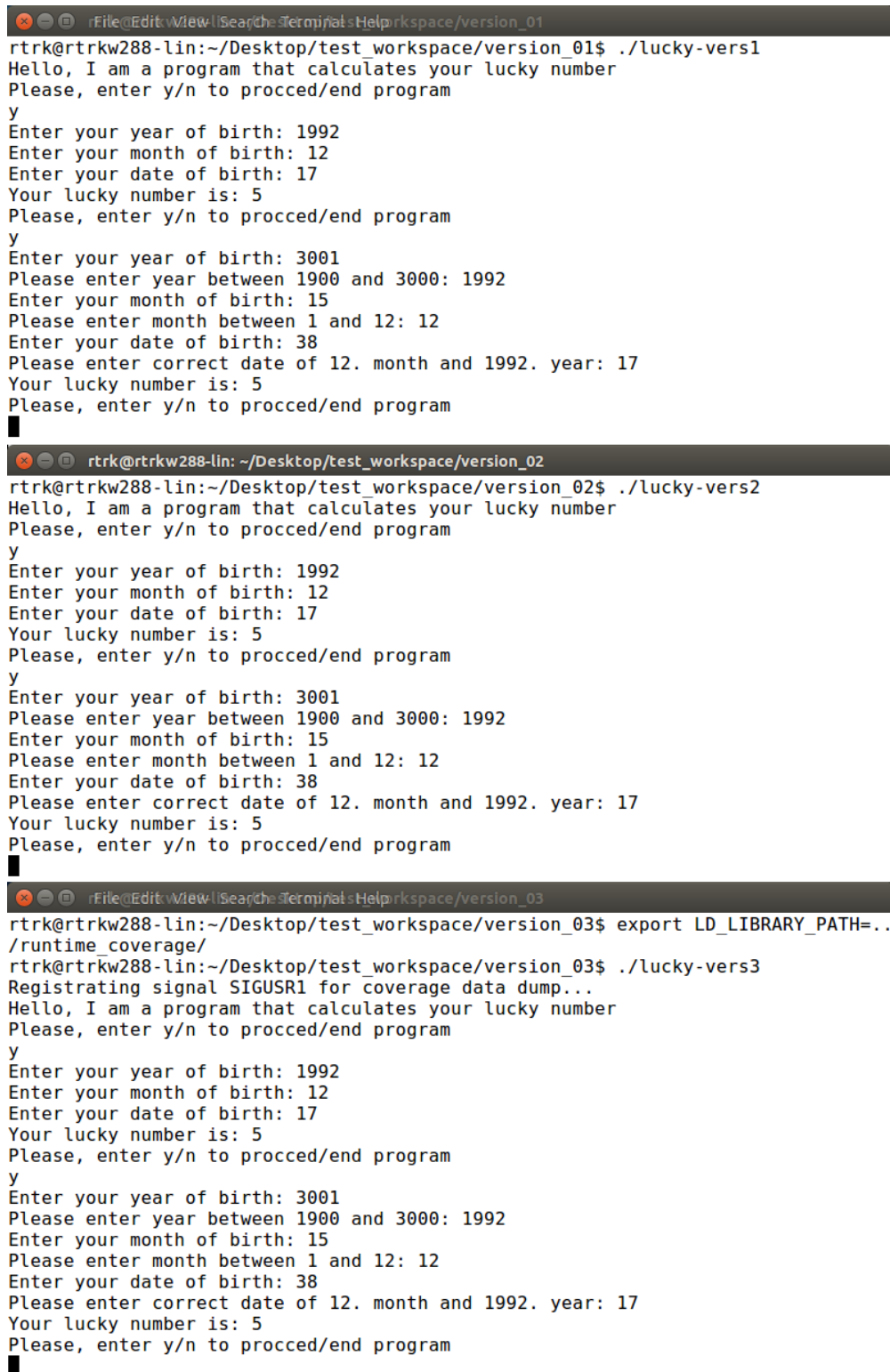
programa, bilo je neophodno uključiti i adresu biblioteke *libcoverage.so* u promenljivu okruženja *LD_LIBRARY_PATH*, kako bi se linker informisao gde je treba potražiti. Prikaz drugog koraka je dat na slici 5.3.

Za izvršavanje trećeg koraka testiranja, bilo je potrebno definisati jedan slučaj upotrebe programa *lucky*. U cilju što bolje demonstracije pojma pokrivenosti i razlika starog i novog pristupa prikupljanju podataka iz izvršavanja programa, odabran je skup komandi koji rezultuju visokoj pokrivenosti. Rezultati ovog koraka prikazani su na slici 5.3. i na osnovu njih se može zaključiti da do ovog trenutka nema razlika u ponašanju između različitih verzija programa, što je jedna od glavih odlika ispravne instrumentalizacije.

Naredni korak predstavlja najbitniji korak testiranja. Materijal generisan u ovom koraku predstavlja krajnji proizvod i njegove karakteristike će učestvovati u procesu finalne validacije. Procedura se razlikuje u zavisnosti od verzije programa. Izvršavanje neinstrumentalizovane verzije, više nije neophodno, te se ista prekida odabirom opcije: *n*. Program, čije podatke iz izvršavanja prikuplja standardna biblioteka *libgcov*, će automatski izvršiti ispis tih podataka na kraju izvršavanja, stoga se i njegovo izvršavanje prekida na isti način. Važno je napomenuti da u slučaju nasilnog prekida programa, poput prekida korišćenjem signala *SIGKILL* ili *SIGARBT*, funkcija *atexit* se neće izvršiti i fajlovi *gcda* neće biti generisani. Stoga se može jasno zaključiti da se podaci o pokrivenosti koda programa koji nemaju ugrađeni mehanizam za takozvani “graciozni izlaz” mogu dobiti samo i isključivo korišćenjem nove biblioteke *libcoverage*. U trenutnom test primeru nije prisutno tako nešto, zbog čega su fajlovi *gcda* uspešno generisani i neprazni. Pozivom alata *gcov*, generišu se uspešno izveštaji, koji će se validirati u narednom koraku. Rezultati koraka 4.a i 4.b prikazani su na slici 5.4.

Program *lucky-vers3* se ne prekida odmah, već se najpre vrši kreiranje fajlova *gcda* i izveštaja. Pokreće se *code_coverage_viewer* i nakon pritiska na dugme: *Choose workspace* odabira se radni direktorijum: *test_workspace/version_03*. U tekstualno polje za unos imena programa, unosi se: *lucky-vers3*, a zatim inicira signal za poziv funkcije *drew_coverage* klikom na dugme *Dump coverage data*, a uspešnost ovog dela procedure može se lako potvrditi postojanjem nepraznih fajlova *gcda* u radnom direktorijumu. Dugme: *Generate gcov and fun files* pokreće kreiranje izveštaja, čija imena smešta u drvoliku komponentu, dok se sadržaj selektovanog izveštaja prikazuje u desnoj polovini ekrana. Slika 5.5 vizuelno demonstrira rezultate ovog koraka.

GLAVA 5. VERIFIKACIJA I VALIDACIJA IMPLEMENTIRANOG REŠENJA



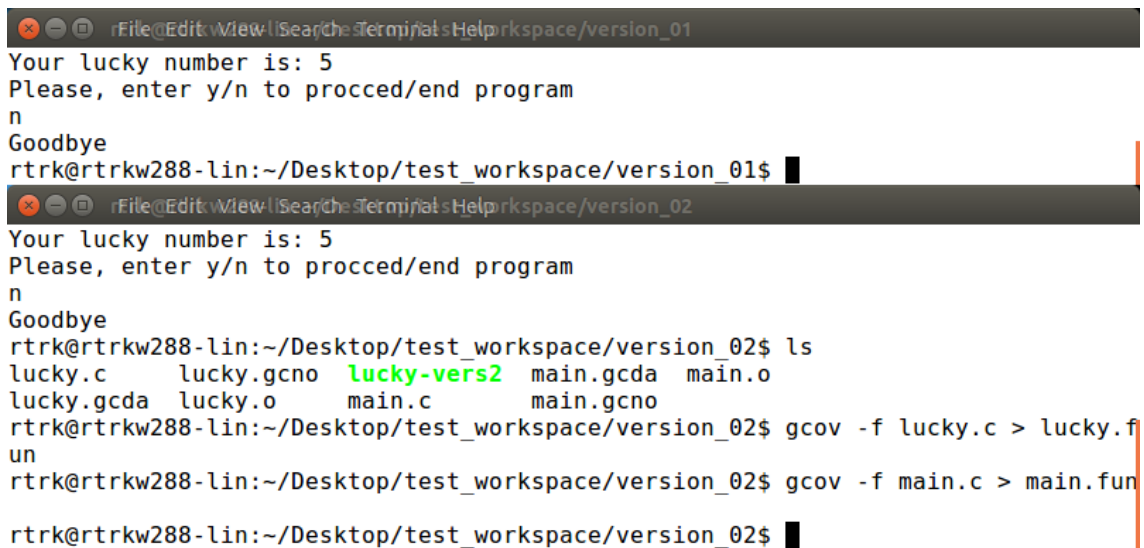
```
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_01$ ./lucky-vers1
Hello, I am a program that calculates your lucky number
Please, enter y/n to procced/end program
y
Enter your year of birth: 1992
Enter your month of birth: 12
Enter your date of birth: 17
Your lucky number is: 5
Please, enter y/n to procced/end program
y
Enter your year of birth: 3001
Please enter year between 1900 and 3000: 1992
Enter your month of birth: 15
Please enter month between 1 and 12: 12
Enter your date of birth: 38
Please enter correct date of 12. month and 1992. year: 17
Your lucky number is: 5
Please, enter y/n to procced/end program
█

rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_02$ ./lucky-vers2
Hello, I am a program that calculates your lucky number
Please, enter y/n to procced/end program
y
Enter your year of birth: 1992
Enter your month of birth: 12
Enter your date of birth: 17
Your lucky number is: 5
Please, enter y/n to procced/end program
y
Enter your year of birth: 3001
Please enter year between 1900 and 3000: 1992
Enter your month of birth: 15
Please enter month between 1 and 12: 12
Enter your date of birth: 38
Please enter correct date of 12. month and 1992. year: 17
Your lucky number is: 5
Please, enter y/n to procced/end program
█

rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_03$ export LD_LIBRARY_PATH=../runtime_coverage/
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_03$ ./lucky-vers3
Registering signal SIGUSR1 for coverage data dump...
Hello, I am a program that calculates your lucky number
Please, enter y/n to procced/end program
y
Enter your year of birth: 1992
Enter your month of birth: 12
Enter your date of birth: 17
Your lucky number is: 5
Please, enter y/n to procced/end program
y
Enter your year of birth: 3001
Please enter year between 1900 and 3000: 1992
Enter your month of birth: 15
Please enter month between 1 and 12: 12
Enter your date of birth: 38
Please enter correct date of 12. month and 1992. year: 17
Your lucky number is: 5
Please, enter y/n to procced/end program
█
```

Slika 5.3: Testiranje na programu *lucky* - Koraci 2 i 3

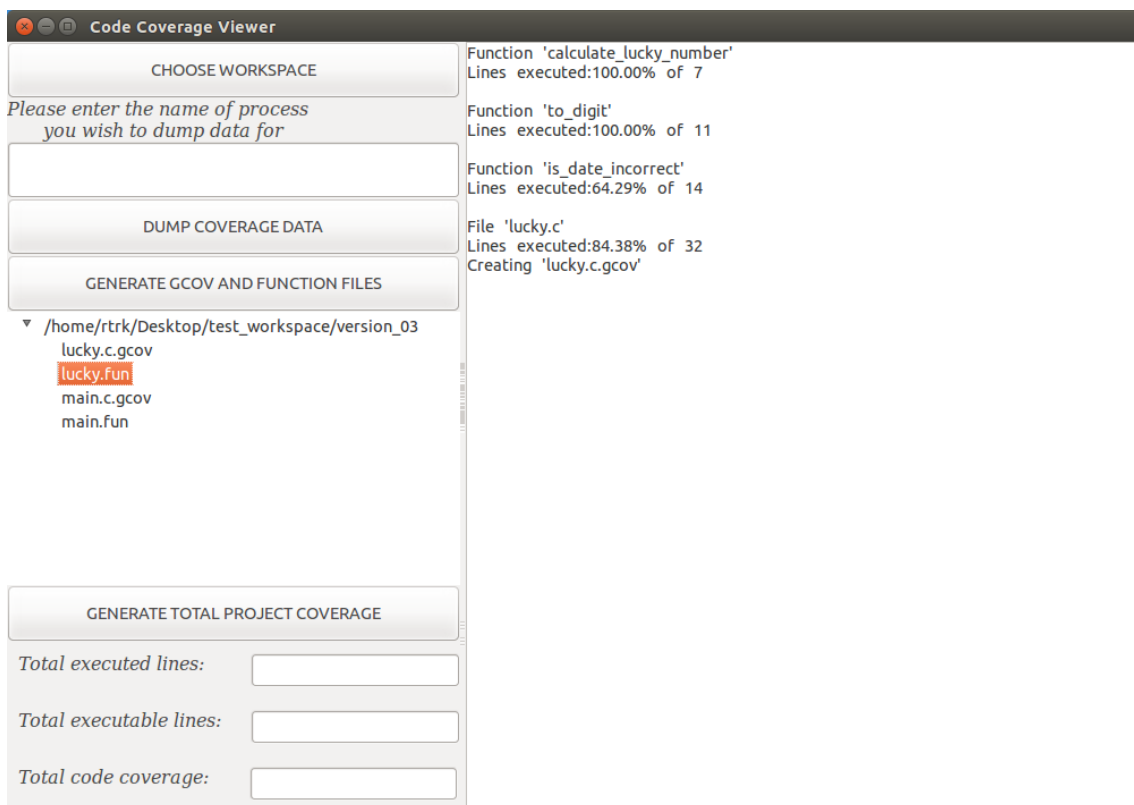
GLAVA 5. VERIFIKACIJA I VALIDACIJA IMPLEMENTIRANOG REŠENJA



```
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_01$
Your lucky number is: 5
Please, enter y/n to procced/end program
n
Goodbye
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_01$

rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_02$
Your lucky number is: 5
Please, enter y/n to procced/end program
n
Goodbye
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_02$ ls
lucky.c      lucky.gcno  lucky-vers2  main.gcda  main.o
lucky.gcda  lucky.o    main.c      main.gcno
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_02$ gcov -f lucky.c > lucky.fun
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_02$ gcov -f main.c > main.fun
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_02$
```

Slika 5.4: Testiranje na programu *lucky* - Koraci 4.a i 4.b



Slika 5.5: Testiranje na programu *lucky* - Korak 4c

Poslednji korak testiranja obuhvata analizu i validaciju rezultata prethodnih koraka. Na slici 5.6 prikazan je ispis programa: *lucky-vers1*, *lucky-vers2* i *lucky-*

GLAVA 5. VERIFIKACIJA I VALIDACIJA IMPLEMENTIRANOG REŠENJA

```
File@Edit:View|Search|Terminal:Help|rkospace/version_01
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_01$ ./lucky-vers1
Hello, I am a program that calculates your lucky number
Please, enter y/n to procced/end program
y
Enter your year of birth: 1992
Enter your month of birth: 12
Enter your date of birth: 17
Your lucky number is: 5
Please, enter y/n to procced/end program
y
Enter your year of birth: 3001
Please enter year between 1900 and 3000: 1992
Enter your month of birth: 15
Please enter month between 1 and 12: 12
Enter your date of birth: 38
Please enter correct date of 12. month and 1992. year: 17
Your lucky number is: 5
Please, enter y/n to procced/end program
n
Goodbye

File@Edit:View|Search|Terminal:Help|rkospace/version_02
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_02$ ./lucky-vers2
Hello, I am a program that calculates your lucky number
Please, enter y/n to procced/end program
y
Enter your year of birth: 1992
Enter your month of birth: 12
Enter your date of birth: 17
Your lucky number is: 5
Please, enter y/n to procced/end program
y
Enter your year of birth: 3001
Please enter year between 1900 and 3000: 1992
Enter your month of birth: 15
Please enter month between 1 and 12: 12
Enter your date of birth: 38
Please enter correct date of 12. month and 1992. year: 17
Your lucky number is: 5
Please, enter y/n to procced/end program
n
Goodbye

File@Edit:View|Search|Terminal:Help|rkospace/version_03
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_03$ export LD_LIBRARY_PATH=../runtime_coverage/
rtrk@rtrkw288-lin:~/Desktop/test_workspace/version_03$ ./lucky-vers3
Registrating signal SIGUSR1 for coverage data dump...
Hello, I am a program that calculates your lucky number
Please, enter y/n to procced/end program
y
Enter your year of birth: 1992
Enter your month of birth: 12
Enter your date of birth: 17
Your lucky number is: 5
Please, enter y/n to procced/end program
y
Enter your year of birth: 3001
Please enter year between 1900 and 3000: 1992
Enter your month of birth: 15
Please enter month between 1 and 12: 12
Enter your date of birth: 38
Please enter correct date of 12. month and 1992. year: 17
Your lucky number is: 5
Please, enter y/n to procced/end program
Dumping coverage data...
Testing...
```

Slika 5.6: Testiranje na programu *lucky* - Korak 5a

GLAVA 5. VERIFIKACIJA I VALIDACIJA IMPLEMENTIRANOG REŠENJA

vers3 na standardni izlaz. U sva tri slučaja ispis iz inicijalnog dela slučaja upotrebe, odnosno zaključno sa poslednjom porukom sadržaja: *Your lucky number is: 5*, je identičan. Time je zadovoljen prvi kriterijum validacije ispisa, odnosno regularnost izvršavanja programa tokom instrumentalizacije. Ispis programa *lucky-vers3* ne sadrži oproštajnu poruku: *Goodbye*, već samo ispise iz registarora i biblioteke, čime su zadovoljena i preostala dva kriterijuma validnosti ispisa.

Zajedničkom analizom koda i ispisa različitih verzija programa *lucky*, mogu se izvesti očekivane vrednosti za izveštaje. Celokupni kod fajla izvornog koda: *lucky.c* se izvršava identičan broj puta u sva tri slučaja. Prikupljanje podataka statičkom bibliotekom *libgcov* se aksiomatski uzima za ispravno, kao sastavni dao programskog prevodioca *GCC*. Usled toga se jednom potvrdom ispravnog prikupljanja podataka novom bibliotekom može smatrati jednakost izveštaja:

test_workspace/version_02/lucky.c.gcov i *test_workspace/version_03/lucky.c.gcov*, odnosno:

test_workspace/version_02/lucky.fun i *test_workspace/version_03/lucky.fun*

Linux komandom *diff* ili jednostavnim posmatranjem može se potvrditi da je kriterijum jednakosti ispunjen, što je vizuelno prikazano na slici 5.7.

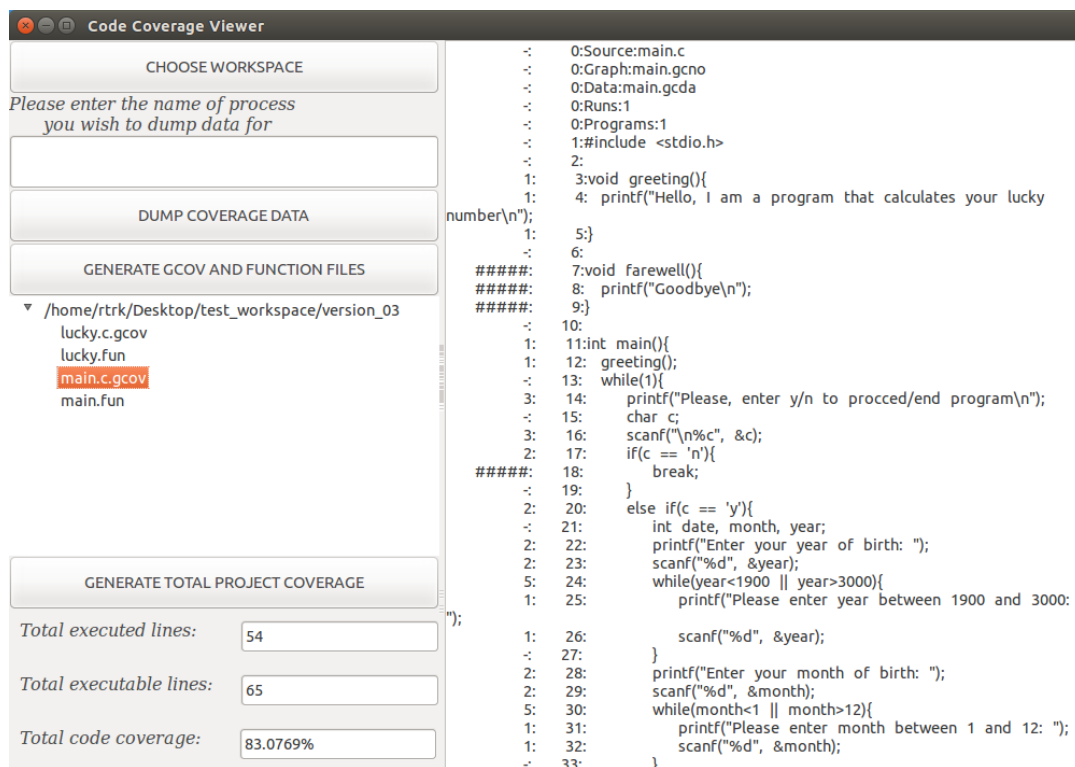


```
rtrk@rtrkw288-lin: ~/Desktop/test_workspace
rtrk@rtrkw288-lin:~/Desktop/test_workspace$ diff version_02/lucky.c.gcov version_03/lucky.c.gcov
rtrk@rtrkw288-lin:~/Desktop/test_workspace$ diff version_02/main.c.gcov version_03/main.c.gcov
12,14c12,14
<      1:      7: void farewell(){
<      1:      8:      printf("Goodbye\n");
<      1:      9: }
---
> #####:      7: void farewell(){
> #####:      8:      printf("Goodbye\n");
> #####:      9: }
22,23c22,23
<      3:     17:      if(c == 'n'){
<      1:     18:          break;
---
>      2:     17:      if(c == 'n'){
> #####:     18:          break;
54,55c54,55
<      1:     49:      farewell();
<      2:     50:      return 0;
---
> #####:     49:      farewell();
> #####:     50:      return 0;
rtrk@rtrkw288-lin:~/Desktop/test_workspace$
```

Slika 5.7: Testiranje na programu *lucky* - Korak 5b

Sa druge strane, izveštaji za fajl izvornog koda: *main.c*, moraju se razlikovati tačno za poslednju iteraciju u okviru najveće petlje i za po jedan poziv funkcija *farewell* and *return*. Na slici 5.7, na kojoj je prikazan izlaz komande *diff* i za ova dva *gcov* izveštaja, jasno se vidi da su ta očekivanja ispunjena.

Pristup koji je trenutno implementiran u okviru prevodioca *GCC* nema ugrađenu podršku za računanje ukupne pokrivenosti. Stoga će se ova funkcionalnost testirati nezavisno od tradicionalnog pristupa, poređenjem rezultata interfejsa *code_coverage_viewer* sa manuelno izračunatim vrednostima. Jednostavnim prebranjem dobijaju se vrednosti u skladu sa onima koje je prikazao *code_coverage_viewer*, koje se mogu videti i na slici 5.8



Slika 5.8: Testiranje na programu *lucky* - Korak 5c

Svi koraci plana za testiranje, definisanog na početku ove sekcije, su dakle ispunjeni, pa se testiranje može proglasiti uspešno obavljenim.

Kompleksni primer: QEMU

Za demonstraciju korektnosti rada biblioteke *libcoverage* i grafičkog korisničkog interfejsa *code_coverage_viewer* u realnim situacijama odabran je projekat *QEMU*,

najpre zbog svoje veličine i značaja, ali i zbog karakteristika na kojima se mogu dobro sagledavati različite prepreke koje se mogu javiti prilikom infiltracije novog softverskog rešenja za prikupljanje podataka iz izvršavanja u realni projekat, kao i načini za njihovo prevazilaženje.

Uvod u QEMU

QEMU [16] je slobodni softver otvorenog koda, čija osnovna funkcija predstavlja emuliranje operativnih sistema ili programa na mašini različite arhitekture od njih samih. Spektar podržanih arhitektura na kojima se *QEMU* može pokrenuti ili koje može emulirati je veoma širok, i obuhvata *MIPS*, *ARM*, *x86*, *PPC* i mnoge druge. *QEMU* ima dva režima rada:

1. sistemski: potpuna emulacija jednog sistema (npr. operativnog) na drugom
2. korisnički (samo za *Linux*): emulacija izvršavanja jedinstvenog procesa, prevedenog za jednu arhitekturu, na mašini druge arhitekture.

Za potrebe ovog testiranja, instrumentalizovaće se kompletni kod projekta *QEMU*, ali će pokretanje biti ograničeno na dva programa korisničkog režima koji simuliraju rad *MIPS* i *ARM* procesora.

Opis koda i sistema za prevođenja projekta *QEMU*

Osnovni radni direktorijum sadrži veliki broj skripti za definisanje fleksibilnih parametara, skripti tipa *makefile* sa instrukcijama za izgradnju sistema i fajlova izvornog koda pisanih u programskom jeziku C. Skripte *makefile*, kao i one za konfiguraciju, su zajedno sa kodovima pojedinih alata smeštene u sam koreni direktorijum projekta. Ostatak je grupisan u poddirektorijume prema funkcionalnosti i/ili programu kome pripadaju.

Glavni deo koda programa koji čine sistemski režim rada se nalazi u poddirektorijumima čiji nazivi sadrže ime arhitekture koja se emulira i sufiks *-softmmu*. U njega se smešta i izvršna verzija, prepoznatljiva po prefiksu: *qemu-system-*.

Poddirektorijumi čiji nazivi sadrže ime arhitekture i sufiks *-linux-user*, sadrže osnovni deo koda odgovarajućeg programa korisničkog režima koji emulira tu arhitekturu. Prevođenjem se kreira izvršna verzija, koja se smešta u isti direktorijum, i čiji naziv sačinjavaju prefiks *qemu-* i naziv arhitekture emuliranog programa.

Proces kreiranja izvršnih programa u okviru *QEMU* projekta se sastoji iz koraka:

1. konfigurisanje parametara za prevođenje
2. prevođenje izvornog koda programa projekta *QEMU* sa tako konfigurisanim parametrima

Za izvođenje prvog koraka zadužena je *shell* skripta, pod nazivom *configure*, koja se nalazi u osnovnom radnom direktorijumu. Podrazumevana konfiguracija se kreira pozivom bez dodatnih opcija. Redefinisanje svakog parametra prevođenja je uslovljeno prisustvom odgovarajuće opcije tokom poziva skripte. Nove vrednosti se upisuju u konfiguracione fajlove koji se takođe nalaze u korenom direktorijumu projekta i koriste kasnije tokom prevođenja. Na ovaj način se može izmeniti programski prevodilac (`--cc=CC`), omogućiti prevođenje sa dodatnim zastavicama (`--extra-cflags=CFLAGS`), odabrati skup programa za prevođenje i slično. Opcija koja je ce biti naročito važna u ovom testiranju je: `--enable-gcov` koja uslovljava instrumentalizaciju uz prikupljanje podataka standardnom bibliotekom *libgcov*.

Pravila za izgradnju svih programa i alata u sklopu *QEMU* projekta su definisana mrežom skripti tipa *makefile*, pri čemu je raspoređivanje izvršeno po kriterijumima pripadnosti i/ili funkcionalnosti. Proces prevođenja se pokreće ključnom rečju *make*. Predefinisano ponašanje podrazumeva izgradnju maksimalnog skupa programa i alata, ali se on po želji može i redukovati korišćenjem određenih opcija prilikom konfigurisanja, u cilju uštede vremena.

Ograničenja i prilagođavanja

Za razliku od jednostavnog primera kao što je program za računanje srećnog broja, sistem pravila za prevođenje, kao i sam kôd projekta *QEMU* su znatno složeniji. Stoga je i implementacija podrške za prikupljanje i prikaz podataka u toku izvršavanja iziskivala veći stepen prilagođavanja i kvantitet izmena.

Prva iteracija je obuhvatala izmene prevođenja, odnosno dodavanje zastavica za instrumentalizaciju, biblioteke *libcoverage* i objektnog fajla *coverage_registration.o*. Očekivana složenost ovog procesa u slučaju prosečnog, većeg projekta, je izrazito velika, iz razloga što zahteva poznavanje sistema za izgradnju softvera do najsitnijih detanja. Za prilagođavanje softvera digitalne televizije, na kojima je biblioteka *libcoverage* originalno testirana, bilo je potrebno više nedelja rada, kao i asistencija stručnjaka za njihove sisteme izgradnje. Međutim, planski i ciljani odabir upravo projekta *QEMU* za ovo testiranje, je dosta redukovao vreme potrebno za prvu iteraciju prilagođavanja. Već ugrađena podrška za statičku instrumentalizaciju na kraju izvršavanja je poslužila kao dobar šablon za oponašanje, koji je nadomestio nedostatak poznavanja sistema.

Po ugledu na opciju: `--enable-gcov`, u skriptu za konfigurisanje je uvedena nova opcija: `--enable-runtime-gcov`, koja takođe uslovljava uključivanje zastavica za instrumentalizaciju u toku prevođenja izvornog fajla do objektnog, ali u

proces linkovanja umesto njih uslovljava prisustvo dinamičke biblioteke *libcoverage* i objektnog fajla za registraciju signala. Ispisivanje odgovarajućih vrednosti u konfiguracioni fajl: *config_host.mak* urađen je identično kao u slučaju postojeće opcije `--enable-gcov`.

Korišćenje istih vrednosti u konfiguracionom fajlu je znatno uticalo na olakšavanje implementacije i validnost procesa instrumentalizacije uz primenu nove biblioteke, ali i proizvelo potrebu za razlikovanjem starog i novog pristupa prikupljanja podataka iz izvršavanja na nivou fajlova izvornog koda. Razlog predstavlja prisustvo koda u okviru fajla *linux-user/exit.c*, koji koristi funkciju standardne biblioteke *libgcov* da proširi mogućnost instrumentalizacije na neke slučajeve neregularnih izlazaka iz programa. Razlikovanje na nivou fajlova izvornog koda ostvareno je uvođenjem nove promenljive `RUNTIME_GCOV`, koja se, u slučaju novog pristupa, prosleđuje zajedno sa instrumentalizacionim zastavicama u toku prevođenja izvornog fajla do objektnog korišćenjem opcije `-D`. Korišćenje funkcije standardne biblioteke *libgcov* je uslovljeno nedefinisanošću ove promenljive. Izmene fajla *linux-user/exit.c* u kojima se uvodi korišćenje promenljive `RUNTIME_GCOV`, prikazane su u okviru listinga 5.4. Kompletne izmene skripte *configure*, prikazane su u okviru listinga 5.5.

```
diff --git a/linux-user/exit.c b/linux-user/exit.c
@@ -19,17 +19,22 @@
 #include "qemu.h"
+ifndef RUNTIME_GCOV
 #ifdef CONFIG_GCOV
 extern void __gcov_dump(void);
 #endif
+endif

 void preexit_cleanup(CPUArchState *env, int code)
 {
 #ifdef TARGET_GPROF
     _mcleanup();
 #endif
+
+ifndef RUNTIME_GCOV
 #ifdef CONFIG_GCOV
     __gcov_dump();
 #endif
+endif
     gdb_exit(env, code);
```

Listing 5.4: Izmene fajla *linux-user/exit.c*

```
diff --git a/configure b/configure

@@ -389,6 +389,7 @@ tcg_interpreter="no"
   gcov="no"
+runtime_gcov="no";
   gcov_tool="gcov"

@@ -1002,6 +1003,8 @@ for opt do
   ;;
+ --enable-runtime-gcov) runtime_gcov="yes"
+ ;;
   --static)

@@ -1676,6 +1679,7 @@ Advanced options
   --enable-gcov          enable test coverage analysis with gcov
+ --enable-runtime-gcov  enable runtime coverage analysis with gcov
   --gcov=GCOV           use specified gcov [$gcov_tool]

@@ -5675,6 +5679,10 @@ write_c_skeleton
   LDFLAGS="-fprofile-arcs -ftest-coverage $LDFLAGS"
+elif test "$runtime_gcov" = "yes" ; then
+ CFLAGS="-fprofile-arcs -ftest-coverage -g -DRUNTIME_GCOV $CFLAGS"
+ LDFLAGS="-L$(BUILD_DIR)/code_coverage_lib -lcoverage $LDFLAGS"
+ LDFLAGS="\$(BUILD_DIR)/code_coverage_lib/coverage_registration.o
   $LDFLAGS"
   elif test "$fortify_source" = "yes" ; then

@@ -6128,6 +6136,7 @@ echo "crypto afalg
   echo "gcov enabled          $gcov"
+echo "runtime gcov enabled $runtime_gcov"
   echo "TPM support           $tpm"

@@ -7023,6 +7032,11 @@ if test "$gcov" =
   fi
+if test "$runtime_gcov" = "yes" ; then
+ echo "CONFIG_GCOV=y" >> $config_host_mak
+ echo "GCOV=$gcov_tool" >> $config_host_mak
+fi
+
+if test "$docker" != "no"; then
```

Listing 5.5: Izmene skripte *configure*

Testiranjem funkcionalnosti implementirane u toku prve iteracije, detektovana je prva smetnja u ispravnom radu podrške za prikupljanje i prikaz podataka u toku izvršavanja. Detaljnom analizom je utvrđeno da se uprkos uspešnoj registraciji funkcije `coverage_handler` za *signal-handler* signala *SIGUSR1*, kôd unutar te funkcije ne izvršava. Uzrok ove pojave je predefinisane svih signala u kodu korisničkih emulatora projekta *QEMU*. Kako bi obezbedio emulaciju maksimalnog kvaliteta koja uključuje i podršku za slanje i prijem *POSIX* signala, sve primljene signale *QEMU* prosleđuje emuliranom korisničkom programu. Poruka karakteristična za signal *SIGUSR1*, koja je ispisana tokom testiranja rezultata prve iteracije, nije dakle vodila poreklo iz emulatora već iz emuliranog korisničkog programa. U cilju prevazilaženja ove prepreke, sprovedena je druga iteracija implementacije podrške za prikupljanje i prikaz podataka u toku izvršavanja, u okviru koje je osmišljen i realizovan novi algoritam za prijem i obradu signala instrumentalizovane verzije *QEMU* emulatora.

Promena konkretnog signala ne bi proizvela drugačije rezultate, usled maksimalne pokrivenosti mehanizma prosleđivanja, dok bi promena tehnike komunikacije sa procesima isuviše negativno uticala na vreme i složenost razvoja. Stoga je potraga za rešenjem okrenuta ka kodu projekta *QEMU*. Sprovedena je nova analiza dokumentacije i izvornog koda, tokom koje je izvršena lokalizacija funkcije za prijem i obradu signala i utvrđen mehanizam njenog rada. Definisane jedinstvenog *signal-handlera* za sve signale, pod nazivom: `host_signal_handler`, obavlja se u fajlu izvornog koda: *linux-user/signal.c*.

Inicijalni plan ugrađivanja podrške za pokretanje prikupljanja podataka iz izvršavanja *POSIX* signalom, podrazumevao je bezuslovnu modifikaciju ponašanja uslovljenog prijemom signala *SIGUSR1* u okviru funkcije `host_signal_handler`. Razmatran je zbog svoje jednostavnosti i lakoće implementacije. Međutim, usled nemogućnosti pravilnog razlikovanja upotebe signala *SIGUSR1* kao okidača prikupljanja podataka o pokrivenosti i slučaja kada ga treba prosleđiti emuliranom programu, nastala je potreba za dodatnim usavršavanjem. Na formiranje nove ideje uticalo je posmatranje argumenata funkcije `host_signal_handler`, tačnije onog argumenta koji predstavlja instancu strukture `siginfo_t`.

U okviru zaglavlja *signal.h*, nalazi se više funkcija koje se mogu koristiti za slanje signala. Jednostavniji od ponuđenih mehanizama, slanje signala bez dodatnih informacija funkcijom `kill`, implementiran je u okviru interfejsa *code_coverage_viewer* kao rešenje koje zadovoljava očekivane funkcionalne zahteve. Sa druge strane, u cilju mogućnosti emuliranja potpuno proizvoljnog programa, koji može komunicirati

i naprednijim mehanizmom, slanjem signala sa dodatnim informacijama funkcijom `sigqueue`, trenutna implementacija projekta *QEMU* sadrži ugrađenu podršku i za prosleđivanje poruke poslate sa signalom. Sadržaj poruke je predstavljen jedinstvenom celobrojnom vrednošću i/ili pokazivačem na strukturu u memoriji proizvoljnog tipa, a čuva se posebnom polju strukture tipa `siginfo_t`: uniji `sigval`. Korisnički emulatori u regularnom slučaju ne vrše čitanje i obradu nijedne informacije po prijemu signala, što pruža dovoljan prostor za nadogradnju *signal-handler* funkcije u fazi između prijema i prosleđivanja. Pozitivna vrednost promenljive `RUNTIME_GCOV`, koja se prosleđuje prilikom prevođenja u slučaju instrumentalizacije novim pristupom, uslovljava čitanje vrednosti broja signala i prateće poruke i dodatnu akciju u slučaju predefinisane kombinacije za prikupljanje podataka iz izvršavanja: signal *SIGUSR1* i tajna poruka: 45949. Specijalna celobrojna vrednost 45949 je dobijena iz poruke: „`call drew_coverage`” kombinacijom *md5* algoritma za heširanje i osnovnih aritmetičkih operacija. Kompletne izmene koda projekta *QEMU* u sklopu druge iteracije implementacije, prikazane su, u okviru listinga 5.6.

Prednost u odnosu na inicijalni plan se ogleda u znatnoj redukciji ograničenja mogućnosti instrumentalizovane verzije emulatora. Gornje ograničenje broja različitih signala je značajno manje od maksimalne celobrojne vrednosti koja se može smestiti u promenljivu tipa `int`. Pored toga, verovatnoća slučaja upotrebe emuliranja korisničkog programa koji koristi tačno ovu kombinaciju je znatno manja nego u slučaju bezuslovne upotrebe signala *SIGUSR1*. Iznimno, podaci iz izvršavanja su ipak namenjeni i potrebni isključivo razvijaoцима *QEMU* projekta. Stoga i ako dođe do pojave ekstremnog slučaja upotrebe na koga utiče ovo minimalno ograničenje, razvojni tim poseduje dovoljno znanja i iskustva da promeni korišćeni signal i/ili poruku.

Izmenom okidača za ispis podataka iz izvršavanja u fajlove *gcda*, došlo je do nekompatibilnosti interfejsa *code_coverage_viewer* i programa instrumentalizovane verzije *QEMU* skupa emulatora.

Prevazilaženje ove prepreke ostvareno je kreiranjem posredničkog programa, sa ulaznim interfejsom kompatibilnim mehanizmu *code_coverage_viewer*-a, odnosno prijemu signala bez poruke, i izlaznim interfejsom kompatibilnim emulatorima projekta *QEMU*, odnosno slanju signala sa porukom. Program nosi naziv: *messenger*, što simbolizuje njegovu ulogu glasnika između novog grafičkog interfejsa i *QEMU*-a.

```

diff --git a/linux-user/signal.c b/linux-user/signal.c

@@ -642,3 +642,7 @@ static inline void rewind_if_in_safe_syscall(void *puc)
 #endif
+#ifdef RUNTIME_GCOV
+extern void coverage_handler(int signo);
+#endif
+
+ static void host_signal_handler(int host_signum, siginfo_t *info,
+                                void *puc)

@@ -666,2 +670,24 @@ static void host_signal_handler(int host_signum,
 siginfo_t *info,
     return;

+
+ int message_int = info->si_value.sival_int;
+ printf("Recieved signal: %d with message: \
+       %d\n", host_signum, message_int);
+ //Checking if signal and message corespond
+ //to predefind coverage dumping comunication
+ if (host_signum == SIGUSR1 && message_int == 45949){
+#ifdef RUNTIME_GCOV
+ //informing about begining of coverage dump procedure
+ //and invoking coverage handler that will do the dumping
+ printf("Program %s is about to dump coverage data\n", \
+       program_invocation_short_name);
+ coverage_handler(host_signum);
+ //Informing about end of coverage dump procedure
+ printf("Program %s has finished dumping coverage data\n", \
+       program_invocation_short_name);
+ // resetting value of message since it is alredy being processed.
+ info->si_value.sival_int = 0;
+ return;
+#endif
+ }

+ trace_user_host_signal(env, host_signum, sig);

```

Listing 5.6: Izmene fajla *linux-user/signal.c*

Program *messenger* se pokreće sa jednim argumentom komandne linije koji predstavlja identifikator instrumentalizovanog pokrenutog *QEMU* emulatora kome treba preneti poruku. Nakon registracije *signal-handler* funkcije za *SIGUSR1*, *messenger* prelazi u stanje čekanja. Po prijemu odgovarajućeg signala, emituje se signal sa porukom 45494, koji uslovljava poziv funkcije *drew_coverage* u emulatoru sa identifikatorom iz argumenta komandne linije. Kôd posredničkog programa, pisan u programskom jeziku C, prikazan je u okviru listinga 5.7.

```

#include <stdio.h> #
    include <signal.h>
        #include <unistd.h>
#include <stdlib.h>
static int pid;
void message_handler(int signo){
    if (signo==SIGUSR1){
        printf("Sending signal to pid %d to dump data\n", pid);
        union sigval coverage_code;
        coverage_code.sival_int = 45949;
        sigqueue(pid, SIGUSR1, coverage_code);
    }
}
int main(int argc, char**argv){
    if(argc == 2) {
        pid = atoi(argv[1]);
        printf("Registrating signal SIGUSR1 for passing message 45949 to pid: %
d...\n",pid);
        signal(SIGUSR1,message_handler);
        while(1){}
    }
    else{
        printf("Incorrect number of command line arguments. Passing %d instead
of 1.\n", argc-1);
        printf("Please pass a single number as argument that presents the PID
of target process.\n");
    }
    return 0;
}

```

Listing 5.7: Kôd u okviru fajla *messenger.c*

Nakon uspešnog okončavanja druge iteracije implementacije podrške za prikupljanje i prikaz podataka u toku izvršavanja u okviru projekta *QEMU*, stvorili su se optimalni uslovi za finalno testiranje.

Plan testiranja

Finalno testiranje biće sprovedeno nad dva *QEMU* emulatora koja rade u korisničkom režimu:

1. *mips-linux-user/qemu-mips*
2. *arm-linux-user/qemu-arm*

Za program koji će emulirati sve verzije, odabrana je neinstrumentalizovana verzija programa *lucky*, detaljnije opisana u prethodnoj sekciji. Za potrebe testiranja, biće kreirane tri verzije *QEMU* skupa emulatora:

1. `<qemu_ vers1>/qemu/<arch>-linux-user/qemu-<arch>` bez instrumentalizacije
2. `<qemu_ vers2>qemu/<arch>-linux-user/qemu-<arch>` sa instrumentalizacijom i standardnom bibliotekom *libgcov*
3. `<qemu_ vers3>qemu/<arch>-linux-user/qemu-<arch>` sa instrumentalizacijom i novom bibliotekom *libcoverage*.

Plan za testiranje je definisan sledećim koracima:

1. konfiguracija i prevođenje:
 - a) verzija bez instrumentalizacije se konfiguriše sa opcijom `--enable-debug` kako bi se postigao isti stepen optimizacije sa instrumentalizovanim verzijama:
`./configure --enable-debug`
dodavanje ove opcije ne utiče na izvršavanje, već je značajno samo za poređenje memoriske zahtevnosti verzija
 - b) verzija sa instrumentalizacijom i standardnom bibliotekom *libgcov* se konfiguriše sa starom opcijom za instrumentalizaciju:
`./configure -enable-gcov`
 - c) verzija sa instrumentalizacijom i novom bibliotekom *libcoverage* se konfiguriše sa novom opcijom za instrumentalizaciju:
`./configure -enable-runtime-gcov`
 - d) prevođenje sve tri verzije se inicira ključnom rečju: *make*
2. analiza memorijske zahtevnosti:
 - a) prostor potreban za skladištenje celokupnog izgrađenog projekta *QEMU*, kao i veličina svakog pojedinačnog emulatora, mora zadovoljavati sledeće nejednakosti:
`size(<qemu_ vers1>) < size(<qemu_ vers3>) < size(<qemu_ vers2>)`
za svaki pojedinačni emulator *QEMU* projekta
3. pokretanje programa:
 - a) podešavanje okruženja, neophodno samo za pokretanje treće verzije:
`export LD_LIBRARY_PATH=<putanja do biblioteke libcoverage>`
 - b) pokretanje svih programa se vrši na standardan način:
`qemu/arm-linux-user/qemu-arm lucky_arm/lucky`
`qemu/mips-linux-user/mips-arm lucky_mips/lucky`

4. sprovođenje inicijalnog dela, za sve tri verzije i oba emulatora unapred utvrđenog, slučaja upotrebe:
 - a) odabir opcije: `y`
 - b) unos: `1992 <enter> 12 <enter> 17`
5. Sprovođenje završnog dela, za sve tri verzije i oba emulatora unapred utvrđenog, slučaja upotrebe:
 - a) programi: `<qemu_ vers1>/qemu/<arch>-linux-user/qemu-<arch>` se prekidaju odabirom opcije: `n`
 - b) programi: `<qemu_ vers2>/qemu/<arch>-linux-user/qemu-<arch>` se prekidaju odabirom opcije: `n` i generišu se izveštaji za svaki fajl izvornog koda ponaosob, standardnim alatom `gcov`:
`gcov <ime_fajla> -f > <ime_fajla>.fun`
 - c) programi: `<qemu_ vers3>/qemu/<arch>-linux-user/qemu-<arch>` se ne prekidaju u cilju demonstracije prikupljanja i prikaza podataka u toku izvršavanja; sprovode se sledeći koraci:
 - i. Pokreće se program *messenger* sa argumentom komandne linije koji predstavlja identifikator procesa *qemu-mips*
 - ii. Pokreće se program *messenger* sa argumentom komandne linije koji predstavlja identifikator procesa *qemu-arm*
 - iii. Pokreće se *code_coverage_viewer*
 - iv. Odabira se radni direktorijum `<qemu_ vers3>/qemu/`
 - v. Unosi se ime programa u odgovarajuće tekstualno polje: *messenger*
 - vi. Klik na dugme: *Dump coverage data* (Napomena: Signal će biti poslat svim procesima sa imenom *messenger* tako da će biti obuhvaćena oba emulatora)
 - vii. Klik na dugme: *Generate gcov and function files*
 - viii. Klik na dugme: *Generate total coverage*
6. validacija:
 - a) validacija ispisa na standardni izlaz; kriterijumi ispunjavanja se definišu na sledeći način:
 - i. sva tri verzije oba emulatora imaju identičan ispis iz inicijalnog dela slučaja upotrebe

- ii. emulatori verzija `<qemu_ vers1>` i `<qemu_ vers2>` imaju identičan ispis iz završnog dela slučaja upotrebe
 - iii. emulatori verzije `<qemu_ vers3>` nemaju ispis iz završnog dela slučaja uporebe
 - iv. emulatori verzije `<qemu_ vers3>` imaju ispise iz biblioteke, registracionog objektnog fajla i funkcije `host_signal_handler`
- b) validacija izveštaja na reprezentativnoj funkciji: `host_signal_handler`; kriterijumi ispunjavanja se definišu na sledeći način
- i. izveštaji verzije `<qemu_ vers2>` ne sadrže pozitivne podatke o izvršavanju funkcije `host_signal_handler`
 - ii. izveštaji verzije `<qemu_ vers3>` sadrže pozitivne podatke o izvršavanju funkcije `host_signal_handler` do poziva funkcije za prikupljanje podataka o izvršavanju iz instrumentalizacionih struktura.
- c) validacija ukupne pokrivenosti

Sprovođenje testiranja i analiza rezultata

Pre početka testiranja, neophodno je kreirati radno okruženje:

1. kreiranje radnog direktorijuma: `mkdir qemu_test_workspace`
2. kreiranje tri bazna direktorijuma za različite verzije programa:
`mkdir qemu_test_workspace/qemu_version_01`
`mkdir qemu_test_workspace/qemu_version_02`
`mkdir qemu_test_workspace/qemu_version_03`
3. instalacija potrebnih biblioteka i alata za rad *QEMU* emulatora i prevođenje progama za *ARM/MIPS* arhitekture:

```
sudo apt-get install build-essential zlib1g-dev pkg-config \  
libglib2.0-dev binutils-dev libboost-all-dev autoconf \  
libtool libssl-dev libpixman-1-dev libpython-dev python-pip \  
libc6-armel-cross libc6-dev-armel-cross libncurses5-dev \  
binutils-arm-linux-gnueabi gcc-arm-linux-gnueabihf \  
g++-arm-linux-gnueabihf
```

i *MTI GNU/Linux Toolchain MIPS32R2-MIPS32R5, MIPS64R2-MIPS64R5 and microMIPS* za *linux x64* preuzeti sa zvaničnog sajta: codescape.mips.com
4. obezbeđivanje koda projekta *QEMU* sa git repozitorijuma:
`cd qemu_test_workspace/qemu_version_<1/2/3>`
`git clone https://github.com/qemu/qemu.git`

- unos izmena neophodnih za prikupljanje i prikaz podataka u toku izvršavanja, ručno ili koristeći

```
git apply
```

- kreiranje direktorijuma sa bibliotekom *libcoverage*, registratorom signala i interfejsom za prikaz podataka o pokrivenosti koda:

```
cp -r SRC/code_coverage_lib/ \
    qemu_test_workspace/qemu_version_03/qemu/
cp -r SRC/code_coverage_viewer/ \
    qemu_test_workspace/qemu_version_03/qemu/
cd qemu_test_workspace/qemu_version_03/qemu/code_coverage_lib
make all
```

- kreiranje direktorijuma sa programom koji će se emulirati:

```
mkdir qemu_test_workspace/lucky_mips
mkdir qemu_test_workspace/lucky_arm
cp lucky.c main.c qemu_test_workspace/lucky_mips
cp lucky.c main.c qemu_test_workspace/lucky_arm
cd qemu_test_workspace/lucky_arm
arm-linux-gnueabi-gcc lucky.c main.c -c
arm-linux-gnueabi-gcc lucky.o main.o -o lucky
cd qemu_test_workspace/lucky_mips
mips-mti-linux-gnu-gcc lucky.c main.c -c
mips-mti-linux-gnu-gcc lucky.o main.o -o lucky -static
```

- kreiranje programa: *messenger*:

```
cp messenger.c qemu_test_workspace/qemu_version_03/
cd qemu_test_workspace/qemu_version_03/
gcc messenger.c -o messenger
```

Prvi korak testiranja obuhvata konfigurisanje i prevođenje tri verzije *QEMU* skupa emulatora. Za potvrdu uspešnosti instrumentalizacije, može se vršiti provera prisustva fajlova sa ekstenzijom *gcno* komandom `find`, kao i prisustvo funkcija `__gcov_init`, `gcov_exit` i `drew_coverage` komandom `readelf`. Očekivani rezultati podrazumevaju:

- prisustvo fajlova sa ekstenzijom *gcno* u direktorijumima:

```
qemu_test_workspace/qemu_version_02
```

i `qemu_test_workspace/qemu_version_03`, odnosno:

```
njihovo odsustvo u qemu_test_workspace/qemu_version_01
```

2. prisustvo funkcija `__gcov_init` i `drew_coverage` i odustvo funkcije `gcov_exit` u programima verzije `qemu_test_workspace/qemu_version_03`
3. prisustvo funkcija `__gcov_init` i `gcov_exit` i odustvo funkcije `drew_coverage` u programima `qemu_test_workspace/qemu_version_02`
4. odsustvo funkcija `__gcov_init`, `drew_coverage` i `gcov_exit` u programima `qemu_test_workspace/qemu_version_01`

Dobijeni rezultati su u skladu sa očekivanjem, što se jasno može videti na slici 5.9, gde su prikazani rezultati sprovođenja prvog koraka testiranja.

```

rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace$ [[ $(find qemu_version_01/ -name "*.gcno") ]] && echo "Yes" || echo "No"
No
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace$ [[ $(find qemu_version_02/ -name "*.gcno") ]] && echo "Yes" || echo "No"
Yes
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace$ [[ $(find qemu_version_03/ -name "*.gcno") ]] && echo "Yes" || echo "No"
Yes
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace$ readelf --all qemu_version_01/qemu/mips-linux-user/qemu-mips | grep "gcov_init\|gcov_exit\|drew_coverage"
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace$ readelf --all qemu_version_02/qemu/mips-linux-user/qemu-mips | grep "gcov_init\|gcov_exit\|drew_coverage"
24448: 0000000000639800 122 FUNC LOCAL DEFAULT 13 __gcov_init
29608: 0000000000638180 5580 FUNC GLOBAL HIDDEN 13 gcov_exit
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace$ readelf --all qemu_version_03/qemu/mips-linux-user/qemu-mips | grep "gcov_init\|gcov_exit\|drew_coverage"
0000009ad4b0 006c00000007 R_X86_64_JUMP_SLO 0000000000000000 drew_coverage + 0
0000009ada00 011a00000007 R_X86_64_JUMP_SLO 0000000000000000 __gcov_init + 0
108: 0000000000000000 0 FUNC GLOBAL DEFAULT UND drew_coverage
282: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __gcov_init
23865: 0000000000000000 0 FUNC GLOBAL DEFAULT UND drew_coverage
25535: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __gcov_init
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace$ readelf --all qemu_version_01/qemu/arm-linux-user/qemu-arm | grep "gcov_init\|gcov_exit\|drew_coverage"
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace$ readelf --all qemu_version_02/qemu/arm-linux-user/qemu-arm | grep "gcov_init\|gcov_exit\|drew_coverage"
26909: 0000000000587e90 122 FUNC LOCAL DEFAULT 13 __gcov_init
31683: 0000000000586810 5580 FUNC GLOBAL HIDDEN 13 gcov_exit
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace$ readelf --all qemu_version_03/qemu/arm-linux-user/qemu-arm | grep "gcov_init\|gcov_exit\|drew_coverage"
000000992460 007000000007 R_X86_64_JUMP_SLO 0000000000000000 drew_coverage + 0
0000009929e0 012600000007 R_X86_64_JUMP_SLO 0000000000000000 __gcov_init + 0
112: 0000000000000000 0 FUNC GLOBAL DEFAULT UND drew_coverage
294: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __gcov_init
24682: 0000000000000000 0 FUNC GLOBAL DEFAULT UND drew_coverage
26373: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __gcov_init
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace$ █

```

Slika 5.9: Testiranje nad projektom *QEMU* - Korak 1

U okviru drugog koraka, sprovedena je ocena memorijske zahtevnosti. U tabeli 5.1 prikazana je veličina celokupnog projekta, kao i nekoliko pojedinačnih programa

Tabela 5.1: Tabele sa kvantitativnim podacima ocene memorije

Veličina izražena u bajtovima	<i>qemu_version_01</i>	<i>qemu_version_02</i>	<i>qemu_version_03</i>
qemu-mips	10556108	15203379	13598393
qemu-arm	10700765	14960036	13917673
qemu-img	6059717	8668398	8655375
qemu-system-mips	34744091	47895829	46295066
qemu-system-arm	40898483	55292999	54251070
qemu/	2457828029	3727564073	3697548246

Veličina izražena u procentima	<i>qemu_version_01</i>	<i>qemu_version_02</i>	<i>qemu_version_03</i>
qemu-mips	100.0000	144.0245	128.8201
qemu-arm	100.0000	139.8034	130.0624
qemu-img	100.0000	143.0496	142.8346
qemu-system-mips	100.0000	137.8532	133.2459
qemu-system-arm	100.0000	135.1957	132.6481
qemu/	100.0000	151.6609	150.4397

unutar projekta, za sve tri verzije. Veličina je prikazana u bajtovima, kao i u obliku procenta regularne veličine. Može se primetiti da se korišćenjem dinamičke biblioteke ostvaruje ušteda od aproksimativno 1% ukupne veličine.

Treći korak testiranja obuhvata pokretanje svih šest programa u šest različitih terminala, sa ciljem lakšeg i boljeg poređenja. Za pokretanje *QEMU* emulatora u koje je linkovana dinamička biblioteka *libcoverage*, preduslov je predstavljalo modifikovanje okruženja. Na vrednost promenljive *LD_LIBRARY_PATH* se nadovezuje dodatno putanja do biblioteke *libcoverage*. Prikaz trećeg koraka, za program *qemu-mips* je dat na slici 5.10.

Za sprovođenje četvrtog koraka testiranja, koristiće se redukovani slučaj upotrebe programa *lucky* iz prethodne sekcije, sa samo jednim odabirom opcije: *y*. Na osnovu rezultata ovog koraka, koji su u slučaju programa *qemu-mips* prikazani na slici na 5.10, može se zaključiti da instrumentalizacija i prikupljanje podataka ne remete regularan rad emulatora. Time je potvrđena ispravnost korišćenja biblioteke *libcoverage* u oglednoj realnoj primeni, definisanoj planom testiranja.

Procedura narednog koraka se razlikuje u zavisnosti od verzije. Izvršavanje neinstrumentalizovanih programa, više nije neophodno, te se prekidaju odabirom opcije: *n*. Programi čije podatke iz izvršavanja prikuplja standardna biblioteka *libgcov* se takođe prekidaju na isti način. Statička biblioteka *libgcov* će automatski obaviti

```

rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace/qemu_version_01/qemu/mips-linux-user
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace/qemu_version_01/qemu/mips-linux-user$ ./qemu-mips ../../lucky_mips/lucky
Hello, I am a program that calculates your lucky number
Please, enter y/n to procced/end program
y
Enter your year of birth: 1992
Enter your month of birth: 12
Enter your date of birth: 17
Your lucky number is: 5
Please, enter y/n to procced/end program

rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace/qemu_version_02/qemu/mips-linux-user
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace/qemu_version_02/qemu/mips-linux-user$ ./qemu-mips ../../lucky_mips/lucky
Hello, I am a program that calculates your lucky number
Please, enter y/n to procced/end program
y
Enter your year of birth: 1992
Enter your month of birth: 12
Enter your date of birth: 17
Your lucky number is: 5
Please, enter y/n to procced/end program

rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace/qemu_version_03/qemu/mips-linux-user
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace/qemu_version_03/qemu/mips-linux-user$ export LD_LIBRARY_PATH=../code_coverage_lib
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace/qemu_version_03/qemu/mips-linux-user$ ./qemu-mips ../../lucky_mips/lucky
Registrating signal SIGUSR1 for coverage data dump...
Hello, I am a program that calculates your lucky number
Please, enter y/n to procced/end program
y
Enter your year of birth: 1992
Enter your month of birth: 12
Enter your date of birth: 17
Your lucky number is: 5
Please, enter y/n to procced/end program

```

Slika 5.10: Testiranje nad projektom *QEMU* - Koraci 3 i 4

funkcionalnost kreiranja fajlova *gda* kao poslednju instrukciju. Pozivom alata *gcov*, za svaki fajl izvornog koda, generišu se dva tipa izveštaja, čije se prisustvo i nepraznost uzimaju za potvrdu uspešnosti ovog koraka testiranja. Validacija je deo procedure narednog koraka.

Programi treće verzije se ne prekidaju do kraja testiranja. Pokreću se dve instance programa *messenger*, zadužene za prijem signala *SIGUSR1* i prosledivanje odgovarajuće poruke preostalim aktivnim emulatorima. Unutar interfejsa *code_coverage_viewer*, odabira se radni direktorijum:

test_workspace/qemu_version_03/qemu/ i ime programa: *messenger*. Klikom na

dugme: *Dump coverage data* kreiraju se fajlovi sa podacima iz dotadašnjeg izvršavanja. Klik na dugme: *Generate gcov and fun files* inicira kreiranje izveštaja, čija se imena mogu videti u drvolikoj komponenti grafičkog korisničkog interfejsa, a sadržaj, nakon selektovanja, u desnom polju. Prikaz terminala i *code_coverage_viewer*-a za oba programa, nakon ovog koraka dat je na slici 5.11

Šesta i poslednja faza testiranja predstavlja analizu dobijenih rezultata u cilju njihove validacije. Sastoji se poređenja ispisa na standardni izlaz sa očekivanim, pri čemu se potvrđuje ispravnost instrumentalizacije kao procesa koji ne utiče na rad programa, i poređenja izveštaja pri čemu se validira ispravnost instrumentalizacije kao procesa koji pruža tačne i precizne informacije.

Na slici 5.12. je dat prikaz ispisa svih šest programa nakon sprovedenog celokupnog slučaja upotrebe. Očekivano poklapanje ispisa koji vodi poreklo iz inicijalnog dela je potvrđeno, stoga je prvi kriterijum validacije ispisa zadovoljen. Prisustvo oprostajne poruke *Goodbye* u ispisima terminiranih programa, kao i poruke o prikupljanju podataka potvrđuju ispunjenost preostalih kriterijuma validacije ispisa.

Validacija samih vrednosti podataka o pokrivenosti je sprovedena nad funkcijom: *host_signal_handler*, za koju se te vrednosti mogu i statički izračunati, analizom koda i plana testiranja. Programima verzije čiji su instrumentalizacioni simboli razrešeni statičkom binliotekom *libgcov*, nije poslat nijedan signal tokom testiranja. Stoga je očekivana vrednost pokrivenosti funkcije koja odgovara na njih jednaka nuli. U okviru listinga 5.8 su prikazani delovi izveštaja: *signal.c.gcov*, koji sadrže podatke o kvantitetu izvršavanja linija koda funkcije: *host_signal_handler*. Može se primetiti da nijedna linija koda nije označena kao izvršena. Niska ##### pored linije sa nazivom i argumentima funkcije potvrđuje da se funkcija nije čak ni delimično izvršavala, odnosno ispunjenje prvog kriterijuma validnosti izveštaja. Programima čije podatke prikuplja biblioteka *libcoverage* je poslat tačno po jedan signal: *SIGUSR1* sa porukom 45949. Na osnovu toga, može se zaključiti da se funkcija *host_signal_handler* izvršavala tačno jednom po programu, tokom koji je uslovljen tom kombinacijom. Pozivom funkcije *coverage_handler*, podaci se prikupljaju iz instrumentalizacionih struktura i beleže. Nakon kreiranja fajlova *gcda*, izvršavanje funkcije *host_signal_handler* se nastavlja, s tim što zbog jedinstvenog preseka stanja, podaci o daljem radu programa neće biti zabeleženi.

GLAVA 5. VERIFIKACIJA I VALIDACIJA IMPLEMENTIRANOG REŠENJA

The image shows a terminal window and a Code Coverage Viewer window. The terminal window displays the following commands and output:

```
rtrk@rtrkw288-lin: ~/Desktop/qemu_test_workspace/qemu_version_03
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace/qemu_version_03$ ps ax | grep
qemu-mips
32065 pts/1 Sl+ 0:00 ./qemu-mips ../.././lucky_mips/lucky
32340 pts/8 S+ 0:00 grep --color=auto qemu-mips
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace/qemu_version_03$ ./messenger
32065&
[2] 32341
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace/qemu_version_03$ Registrating si
gnal SIGUSR1 for passing message 45949 to pid: 32065...

rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace/qemu_version_03$ ps ax | grep
messenger
32341 pts/8 R 0:20 ./messenger 32065
32344 pts/8 S+ 0:00 grep --color=auto messenger
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace/qemu_version_03$ kill -10 32341
Sending signal to pid 32065 to dump data
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace/qemu_version_03$
```

The Code Coverage Viewer window shows the following information:

Please, enter y/n to proceed/end program
 Received signal: 10 with message: 45949
 Program qemu-mips is about to dump coverage data
 Dumping coverage data...
 Data dump initiated...
 Data dump completed...
 Program qemu-mips has finished dumping coverage data

Function	Lines executed
Function 'page_size_init'	Lines executed:85.71% of 7
Function 'target_words_bigendian'	Lines executed:0.00% of 2
Function 'cpu_memory_rw_debug'	Lines executed:0.00% of 26
Function 'cpu_abort'	Lines executed:0.00% of 24
Function 'cpu_single_step'	Lines executed:0.00% of 5
Function 'cpu_breakpoint_remove_all'	Lines executed:60.00% of 5
Function 'cpu_breakpoint_remove_by_ref'	Lines executed:0.00% of 5
Function 'cpu_breakpoint_remove'	Lines executed:0.00% of 6
Function 'cpu_breakpoint_insert'	Lines executed:0.00% of 11
Function 'cpu_watchpoint_insert'	Lines executed:0.00% of 2
Function 'cpu_watchpoint_remove_by_ref'	Lines executed:0.00% of 2
Function 'cpu_watchpoint_remove'	Lines executed:0.00% of 2

The Code Coverage Viewer also shows a tree view of the project structure:

- /home/rtrk/Desktop/qemu_test_workspace/qemu_ve
 - qemu
 - aes.c.gcov
 - aes.fun
 - arm-linux-user
 - accel

At the bottom of the Code Coverage Viewer, there are three input fields for summary statistics:

- Total executed lines:
- Total executable lines:
- Total code coverage:

Slika 5.11: Testiranje nad projektom QEMU - Korak 5c


```

rtrk@rtrkw288-lin: ~/Desktop/qemu_test_workspace/qemu_version_01/qemu/mips-linux-user
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace/qemu_version_01/qemu/mips-linux-
user$ ./qemu-mips ../../../../lucky_mips/lucky
Hello, I am a program that calculates your lucky number
Please, enter y/n to procced/end program
y
Enter your year of birth: 1992
Enter your month of birth: 12
Enter your date of birth: 17
Your lucky number is: 5
Please, enter y/n to procced/end program
n
Goodbye

rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace/qemu_version_02/qemu/mips-linux-user
rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace/qemu_version_02/qemu/mips-linux-
user$ ./qemu-mips ../../../../lucky_mips/lucky
Hello, I am a program that calculates your lucky number
Please, enter y/n to procced/end program
y
Enter your year of birth: 1992
Enter your month of birth: 12
Enter your date of birth: 17
Your lucky number is: 5
Please, enter y/n to procced/end program
n
Goodbye

rtrk@rtrkw288-lin:~/Desktop/qemu_test_workspace/qemu_version_03/qemu/mips-linux-
user$ ./qemu-mips ../../../../lucky_mips/lucky
Registering signal SIGUSR1 for coverage data dump...
Hello, I am a program that calculates your lucky number
Please, enter y/n to procced/end program
y
Enter your year of birth: 1992
Enter your month of birth: 12
Enter your date of birth: 17
Your lucky number is: 5
Please, enter y/n to procced/end program
Recieved signal: 10 with message: 45949
Program qemu-mips is about to dump coverage data
Dumping coverage data...
Data dump initiated...
Data dump completed...
Program qemu-mips has finished dumping coverage data
    
```

Slika 5.12: Testiranje nad projektom *QEMU* - Korak 6a

U okviru listinga 5.8 su prikazani delovi izveštaja: *signal.c.gcov*, koji sadrže podatke o kvantitetu izvršavanja linija koda funkcije: *host_signal_handler*. Može se primetiti da nijedna linija koda nije označena kao izvršena. Niska ##### pored linije sa nazivom i argumentima funkcije potvrđuje da se funkcija nije čak ni delimično izvršavala, odnosno ispunjenje prvog kriterijuma validnosti izveštaja. Programima čije podatke prikuplja biblioteka *libcoverage* je poslat tačno po jedan signal: *SIGUSR1* sa porukom 45949. Na osnovu toga, može se zaključiti da se funkcija *host_signal_handler* izvršavala tačno jednom po programu, tokom koji je uslo-

vljen tom kombinacijom. Pozivom funkcije `coverage_handler`, podaci se prikupljaju iz instrumentalizacionih struktura i beleže. Nakon kreiranja fajlova `gcda`, izvršavanje funkcije `host_signal_handler` se nastavlja, s tim što zbog jedinstvenog preseka stanja, podaci o daljem radu programa neće biti zabeleženi.

```
#####: 645:static void host_signal_handler(int host_signum,
-: 646:                                     siginfo_t *info, void *puc)
-: 647:{
#####: 648:   CPUArchState *env = thread_cpu->env_ptr;
#####: 649:   CPUState *cpu = ENV_GET_CPU(env);
#####: 650:   TaskState *ts = cpu->opaque;
-: 651:
-: 652:   int sig;
-: 653:   target_siginfo_t tinfo;
#####: 654:   ucontext_t *uc = puc;
-: 655:   struct emulated_sigtable *k;
-: 656:
-: 657:   /* the CPU emulator uses some host signals to detect
exceptions,
-: 658:   we forward to it some signals */
#####: 659:   if ((host_signum == SIGSEGV || host_signum == SIGBUS)
#####: 660:       && info->si_code > 0) {
#####: 661:       if (cpu_signal_handler(host_signum, info, puc))
#####: 662:         return;
-: 663:   }
-: 664:
-: 665:   /* get target signal number */
#####: 666:   sig = host_to_target_signal(host_signum);
```

Listing 5.8: Testiranje nad projektom *QEMU* - Korak 6b bez izmena

U okviru listinga 5.9 su prikazani delovi izveštaja: `signal.c.gcov` koji sadrže podatke o kvantitetu izvršavanja linija koda funkcije: `host_signal_handler`.

```

1: 649:static void host_signal_handler(int host_signal,
-: 650:                                siginfo_t *info, void *puc)
-: 651:{
1: 652:    CPUArchState *env = thread_cpu->env_ptr;
1: 653:    CPUState *cpu = ENV_GET_CPU(env);
1: 654:    TaskState *ts = cpu->opaque;
-: 655:
-: 656:    int sig;
-: 657:    target_siginfo_t tinfo;
1: 658:    ucontext_t *uc = puc;
-: 659:    struct emulated_sigtable *k;
-: 660:
-: 661:    /* the CPU emulator uses some host signals to detect
exceptions,
-: 662:       we forward to it some signals */
1: 663:    if ((host_signal == SIGSEGV || host_signal == SIGBUS)
##### 664:        && info->si_code > 0) {
##### 665:        if (cpu_signal_handler(host_signal, info, puc))
##### 666:            return;
-: 667:    }
-: 668:
-: 669:    /* get target signal number */
1: 670:    sig = host_to_target_signal(host_signal);
1: 671:    if (sig < 1 || sig > TARGET_NSIG)
##### 672:        return;
-: 673:
-: 674:
1: 675:    int message_int = info->si_value.sival_int;
1: 676:    printf("Recieved signal: %d with message: %d\n",
host_signal, message_int);
-: 677:    //Checking if signal and message corespond to
predefinid coverage dumping communication
1: 678:    if (host_signal == SIGUSR1 && message_int == 45949){
-: 679:#ifdef RUNTIME_GCOV
-: 680:        // informing about begining of coverage dump
procedure and invoking coverage handler that will do the dumping
1: 681:        printf("Program %s is about to dump coverage data\n",
program_invocation_short_name);
1: 682:        coverage_handler(host_signal);
-: 683:        //Informing about end of coverage dump procedure
##### 684:        printf("Program %s has finished dumping coverage
data\n", program_invocation_short_name);
-: 685:        // reseting value of message since it is alredy
being processed.
##### 686:        info->si_value.sival_int = 0;

```

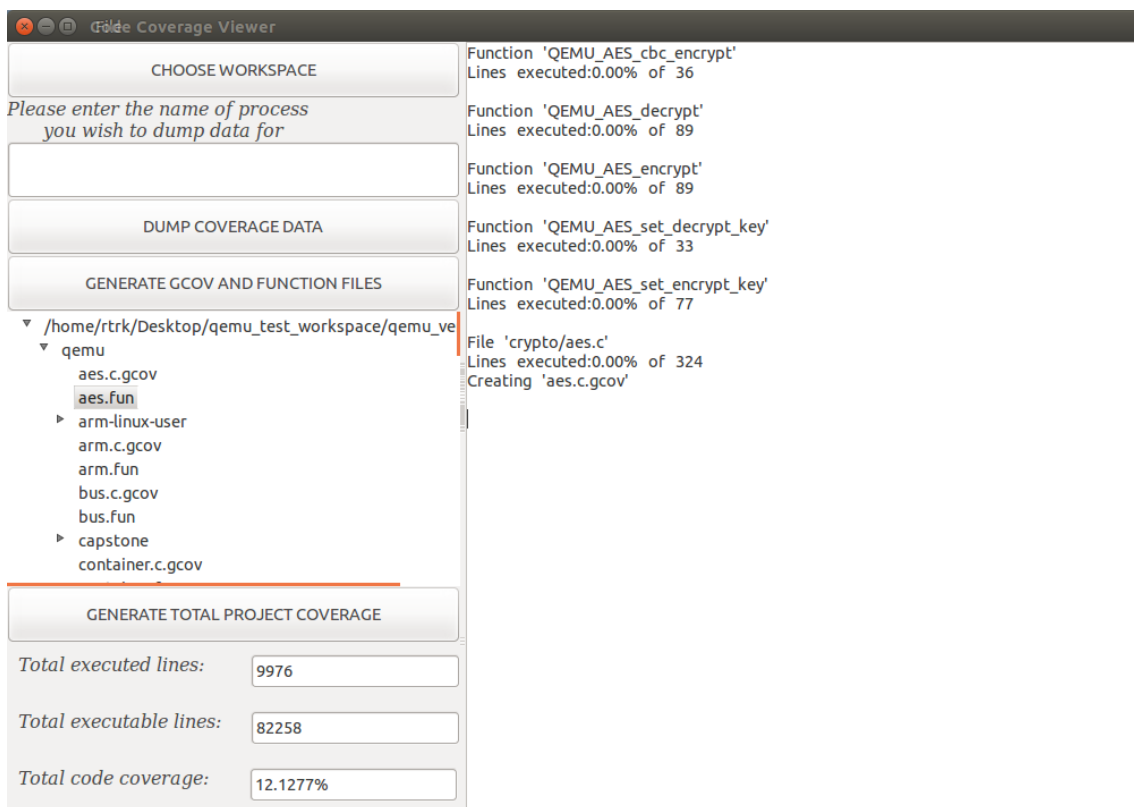
Listing 5.9: Testiranje nad projektom *QEMU* - Korak 6b sa izmenama

Linije koda, zaključno sa onom koja se odnosi na poziv funkcije `coverage_handler`, koje pripadaju toku izvršavanja karakterističnom za prijem signala *SIGUSR1* sa po-

GLAVA 5. VERIFIKACIJA I VALIDACIJA IMPLEMENTIRANOG REŠENJA

rukom 45949, su označene kao jedinstveno izvršene. Sve linije koda koje se izvršavaju nakon poziva funkcije: `coverage_handler`, odnosno kreiranja fajlova `gcda` su označene kao neizvršene. Dobijeni rezultati su u skladu sa očekivanjima i zahtevima drugog kriterijuma validacije izveštaja. Stoga se i ovaj kriterijum može označiti kao ispunjen.

Generisanje ukupne pokrivenosti projekta će biti sprovedeno na isti način kao i u okviru testiranja nad programom `lucky` u prethodnoj sekciji. Jednostavnim prebrajanjem linija koje su u izveštajima označene kao izvršene/izvršne, i računanjem količnika ta dva broja, dobijaju se jednake vrednosti kao one koje su generisane novim interfejsom i prikazane na slici 5.13



Slika 5.13: Testiranje nad projektom *QEMU* - Korak 6c

Na osnovu uspešnih rezultata svih koraka predviđenog plana za testiranje, testiranje se može smatrati uspešnim.

Glava 6

Zaključak

U okviru ovog rada proučavana je analiza programa, njene vrste i tehnike. Posebno je proučavana tehnika dinamičke analize pod nazivom profajliranje, koja obezbeđuje razne informacije o ponašanju programa, među kojima je i pokrivenost (stepen izvršenosti) koda. Ove informacije predstavljaju važne vodilje procesa testiranja i optimizacije softvera. Definisane su i ukratko objašnjene sve tri faze profajliranja: instrumentalizacija, odnosno ubrizgavanje koda za praćenje izvršavanja, prikupljanje metapodataka o izvršavanju programa i njihovo prikazivanje. Detaljno su ispitivane implementacija i mogućnosti poslednje dve faze, kao i prostor za njihovo unapređenje, u okviru programskih prevodioca: *Clang* i *GCC*.

Na osnovu analize implementacija prikupljanja i prikazivanja podataka iz izvršavanja u okviru ova dva programska prevodioca, kao i analize potreba specifičnih softvera sa karakterističnim vremenom izvršavanja, zaključeno je da prevodilac *GCC* ima najveći potencijal za unapređenje do optimalnog rešenja, ugradnjom podrške za prikupljanje podataka iz izvršavanja u toku rada programa. Sa druge strane, posmatranjem karakteristika njegovog ugrađenog alata *gcov*, poput nedostatka ukupne statistike za celokupni projekat, ustanovljen je i dodatni prostor za unapređenje treće faze profajliranja u pravcu bolje preglednosti i povišene informativnosti.

U okviru ovog rada implementirano je softversko rešenje problema prikupljanja i prezentovanja podataka iz izvršavanja programa u toku njegovog rada. Rešenje se sastoji iz *backend* podrške u vidu dinamičke biblioteke *libcoverage*, koja omogućava dostupnost podataka u toku izvršavanja, i *frontend* podrške u vidu grafičkog interfejsa *code_coverage_viewer*, koji unapređuje korisnički doživljaj prezentovanja podataka. Nova biblioteka je izgrađena po uzoru na postojeću biblioteku *libgcov* i u potpunosti pokriva i njene slučajeve upotrebe, odnosno one karakteristične za

period nakon završetka rada programa. Izgrađena je kao dinamička i nezavisna od prevodioca u cilju postizanja maksimalne fleksibilnosti i performansi. Poziv funkcije za kreiranje binarnih fajlova sa podacima iz izvršavanja, prepušten je vlasniku programa. Ponuđena je i opciona implementacija poziva putem signala. Novi grafički interfejs predstavlja omotač celokupnog procesa prikupljanja i prikaza podataka, sa osnovnom ulogom strukturnije i intuitivnije prezentacije podataka iz izvršavanja. Omogućava generisanje i pregledanje linijskih i funkcijskih izveštaja. Bolja preglednost je postignuta smeštanjem izveštaja u drvo koje odražava strukturu direktorijuma. Informativnost je povišena dodatnim funkcijskim izveštajima, kao i prikazom ukupne statistike.

Kvalitet implementiranog rešenja je potvrđen kroz testiranje na više različitih tipova softvera, kao i kroz upotrebu u realnim, komercionalnim projektima. U okviru rada je validacija prikazana kroz proces prilagođavanja unutrašnje infrastrukture jednog kompleksnog softvera otvorenog koda, pod nazivom *QEMU*, kao i upotrebe novog rešenja za dobijanje podataka iz izvršavanja na dva njegova proizvoljno odabrana emulatora. Cilj ovakvog prikaza je da osposobi korisnika za primenu softverskog rešenja implementiranog u okviru ovog rada, za profajliranje potpuno proizvoljnog izvornog koda, pisanog u programskom jeziku *C*.

Dalji razvoj bi mogao ići u pravcu proširivanja funkcionalnosti na druge prevodioce ili programske jezike, unapređivanja biblioteke ili alata. U cilju unapređenja biblioteke bi se mogla implementirati podrška za višestruko pozivanje funkcije za prikupljanje u toku jednog izvršavanja, koje trenutno nije dozvoljeno zbog opasnosti višestrukog sumiranja istih podataka. Sa druge strane, dalji razvoj alata bi mogao obuhvatati dodavanje novih vrsta izveštaja ili novih vrsta sumarnih podataka, poput, na primer, sortirane statistike po svim modulima.

Bibliografija

- [1] Basic block. <https://gcc.gnu.org/onlinedocs/gccint/Basic-Blocks.html>.
- [2] Clang: a C language family frontend for LLVM. <https://clang.llvm.org/>.
- [3] Code Coverage with gcov. https://web.archive.org/web/20140409083331/http://xview.net/papers/gcov/code_coverage_gcov.pdf.
- [4] GCC, the GNU Compiler Collection. <https://gcc.gnu.org>.
- [5] gcov official site. <http://gcc.gnu.org/onlinedocs/gcc/Gcov.html>.
- [6] Intel C++ Compiler User and Reference Guides. <https://www.lri.fr/~lacas/Teaching/SIMD/icc.pdf>.
- [7] 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>.
- [8] LLVM official site. <http://llvm.org/>.
- [9] Razvoj softvera - materijali sa predavanja. <http://poincare.matf.bg.ac.rs/~smalkov/files/rs.r290.2018/public/Predavanja/Razvoj%20softvera.01.2017%20-%20problemi,%20oo,%20uml.p4.pdf>.
- [10] Source code instrumentation overview. https://www.ibm.com/support/knowledgecenter/SSSHUF_8.0.0/com.ibm.rational.testrt.doc/topics/cinstruovw.html.
- [11] Valgrind official site. <http://www.valgrind.org/>.

- [12] Paul Ammann and Jeff Offutt. Introduction to software testing. *Cambridge University Press*, 2016.
- [13] Oliver Arafati and Dirk Riehle. The comment density of open source software code. In *2009 31st International Conference on Software Engineering-Companion Volume*, pages 195–198. IEEE, 2009.
- [14] 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.
- [15] 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.
- [16] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.
- [17] R Brader, H Hilliker, and A Wills. Unit Testing: Testing the Inside. *Microsoft Developer Guidance*, 2013.
- [18] John P Chambers. Cyclic redundancy data check encoding method and apparatus, August 11 1981. US Patent 4,283,787.
- [19] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of model checking*. Springer, 2018.
- [20] 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.
- [21] A Glower. In pursuit of code quality: Don’t be fooled by the coverage report. *IBM Developer Works blog post*, 2006.
- [22] V. Gupta. Measurement of Dynamic Metrics Using Dynamic Analysis of Programs. *APPLIED COMPUTING CONFERENCE (ACC ’08), Istanbul, Turkey*, 2008.

- [23] 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.
- [24] Graylin Jay, Joanne E Hale, Randy K Smith, David P Hale, Nicholas A Kraft, and Charles Ward. Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship. *JSEA*, 2(3):137–143, 2009.
- [25] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.
- [26] B Marick. How to misuse code coverage. *Proceedings of the 16th Interational Conference on Testing Computer Software*, 1999.
- [27] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.
- [28] F Nielson, H. R. Nielson, and C Hankin. Principles of program analysis. *Springer Science & Business Media*, 2015.
- [29] Marina Nikolić, Mladen Nikolić, and Darko Ristivojević. Proširenje alata za profajliranje mogućnošću prikupljanja i prikaza podataka o pokrivenosti koda tokom izvršavanja. *Zbornik 61. Konferencije za elektroniku, telekomunikacije, računarstvo, automatiku i nuklearnu tehniku, ETRAN 2017*, page 6, 2017. https://www.etrans.rs/common/pages/proceedings/ETRAN2017/RT/IcETRAN2017_paper_RT2_1.pdf.
- [30] A Piziali. “Code coverage,” in Functional verification coverage measurement and analysis. *Springer Science & Business Media*, 2007.
- [31] William Von Hagen. *The definitive guide to GCC*. Apress, 2011.
- [32] Milena Vujošević Janičić. Verifikacija softvera. http://www.verifikacijasoftware.matf.bg.ac.rs/vs/predavanja/03_dinamicka_analiza/03_dinamicka_analiza.pdf.
- [33] L William, B Smith, and S Heckman. Test Coverage with EclEmma. *Technical Report Raleigh*, 2008.

Biografija autora

Marina Nikolić (*Sombor, 17. decembar 1992.*) je diplomirani matematičar za računarstvo i informatiku Univerziteta u Beogradu i trenutno radi kao softverski inženjer u kompaniji RT-RK. Završila je Četvrtu beograsku gimnaziju 2012. godine kao dobitnik Vukove diplome na osnovu odličnog uspeha i ostvarenih rezultata na takmičenjima iz matematike i fizike. Iste godine upisala je Matematički fakultet u Beogradu, smer: matematika, modul: računarstvo i informatika, i diplomirala u julu 2017. godine sa prosečnom ocenom 7.75. Poslednju godinu studija provela je kao stipendista novosadske kompanije koja proizvodi i održava softver za uređaje sa ugrađenim računarom, pod nazivom RT-RK. Po završetku osnovnih akademskih studija upisala je master studije, i u toku akademske 2017-2018 godine položila sve ispite predviđene planom i programom master studija, sa prosečnom ocenom 10.0. U julu 2017. se zaposlila u kompaniji RT-RK, u timu koji održava softver za instrumentalizaciju operativnih sistema mrežnih uređaja koje proizvodi američka korporacija pod nazivom CISCO. Trenutno radi u istom timu i usavršava svoje znanje u oblasti računarskih mreža, operativnih sistema i programskih prevodioca.