UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Irena Blagojević

AUTOMATSKO UKLANJANJE REDUNDANTNIH TESTOVA

master rad

Beograd, 2022.

Mentor:

prof. dr Milena Vujošević Janičić
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

prof. dr Filip Marić
Univerzitet u Beogradu, Matematički fakultet

doc. dr Milan Banković
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: ____________________

Naslov master rada: Automatsko uklanjanje redundantnih testova

Rezime: Tokom razvoja softvera, sa ciljem pronalaženja neispravnosti i delova sistema koji nisu dovoljno razvijeni, neophodno je vršiti testiranje. Kada su u pitanju veliki projekti, vremenom može doći do nagomilavanja testova, pa samim tim i ponovnog proveravanja istih elemenata sistema, čime se otežava održavanje i produžava vreme potrebno za sprovođenje testiranja. Testovi koji izlažu proveri iste elemente sistema nazivamo redundantnim. Ovakve testove moguće je pronaći pažljivom analizom, ali što je softver komplikovaniji, ovaj zadatak postaje zahtevniji i pokazuje se potreba za automatizacijom detektovanja i uklanjanja suvišnih testova. Cilj rada je predstavljanje problema automatskog uklanjanja redundantnih testova pomoću SMT rešavača i implementacija alata u jeziku C++ kojim se demonstrira takav pristup.

Ključne reči: verifikacija softvera, testiranje, redundantni testovi, automatsko rezonovanje, SMT rešavači

Sadržaj

1 Uvod **1**

2 Tehnike testiranja i pokrivenost koda **3**

2.1 Statičko i dinamičko testiranje 3
2.2 Nivoi testiranja 4
2.3 Testiranje crne kutije 5
2.4 Testiranje bele kutije 6
2.5 Pokrivenost koda 11

3 Pristupi rešavanju problema **15**

3.1 Redundantni testovi 15
3.2 Osnova za izgradnju modela 17
3.3 Model zasnovan na pokrivenosti 19

4 Korišćeni alati i tehnologije **23**

4.1 SMT rešavači 23
4.2 Google testovi 27
4.3 Alat *Gcov* 27
4.4 Razvojno okruženje *Qt* 28

5 Opis implementacije i evaluacija **29**

5.1 Arhitektura sistema 29
5.2 Grafički korisnički interfejs 34
5.3 Instalacija i pokretanje 41
5.4 Evaluacija 42

6 Zaključak **47**

Bibliografija **48**

Glava 1

Uvod

Testiranje softvera se može definisati kao postupak ispitivanja rada sistema sa ciljem pronalaženja *grešaka* (eng. *bugs*). Pritom je poželjno pronaći greške što je ranije moguće tokom procesa razvoja softvera. Za testiranje je potrebno poznavati *specifikaciju proizvoda* (eng. *product specification*). Specifikacija proizvoda je dogovor unutar razvojnog tima i definiše proizvod koji tim pravi, opis željenog ponašanja, šta će softver raditi, a šta neće. Do grešaka u softveru dolazi kada je ispunjeno barem jedno od pet pravila definisanih u [23]:

1. Softver se ne ponaša kao što je definisano u specifikaciji proizvoda.
2. Softver se ponaša kao što je u specifikaciji definisano da ne bi trebalo da se ponaša.
3. Softver pokazuje ponašanje koje nije pomenuto u specifikaciji.
4. Softver pokazuje ponašanje koje nije pomenuto u specifikaciji, a treba da bude pomenuto.
5. Softver je teško razumeti, teško koristiti, radi sporo ili će iz ugla krajnjeg korisnika (na osnovu mišljenja testera) ponašanje izgledati neispravno.

U slučaju pravila 3 i 4, to znači da neka nepoželjna ponašanja softvera proizilaze i iz grešaka u samoj specifikaciji.

Kako se softver razvija, tako se skup definisanih testova menja i povećava u cilju pronalaženja grešaka nastalih dodavanjem novog ili menjanjem postojećeg koda. Međutim, tokom vremena, može doći do nagomilavanja test slučajeva koji ne

doprinose sposobnosti celog skupa testova da pronađe greške, jer vrše provere koje postojeći testovi već pokrivaju. Takvi testovi se nazivaju *redundantnim*.

Postojanje redundantnih test slučajeva može značajno da oteža održavanje [18]. Redundantni testovi produžavaju vreme izvršavanja celog skupa testova bez pružanja novih informacija o kodu koji se testira. Takođe, iako je dobra praksa pisati testove koji su otporni na izmene u softveru koji se testira, ukoliko to nije slučaj, testeri moraju pronaći i ažurirati sve relevantne testove nakon izmena napravljenih u jedinici koda koja se testira, kao što se navodi u radu [18]. Pritom, ukoliko se promena testova ne izvrši pažljivo, testovi iz istog skupa mogu davati kontradiktorne rezultate. Pažljivim uklanjanjem redundantnih testova se smanjuje cena održavanja i održava se integritet testova, uz zadržavanje sposobnosti skupa testova da pronađe defekte. Postoje različiti pristupi rešavanju problema automatskog pronalaženja minimalnog skupa testova koji zadovoljava zadate kriterijume, o kojima se može više pročitati u radu [5].

U ovoj tezi bliže je objašnjena motivacija za uklanjanje redundantnih testova, opisan je postupak pravljenja modela na osnovu kog se skup testova može redukovati i implementiran je alat na osnovu tog modela. U glavi 2 su opisane tehnike testiranja koje se mogu koristiti tokom razvoja softvera. Posebna pažnja je posvećena opisivanju pojma pokrivenosti koda i različitim nivoima pokrivenosti pomoću kojih se vrši analiza koda. Problem postojanja redundantnih testova predstavljen je u glavi 3. Opisan je postupak izgradnje modela za uklanjanje ovakvih testova, koji je zasnovan na pokrivenosti koda. Alati i tehnologije koje su korišćene u implementaciji alata za redukovanje skupa testova nazvanog *TestSuiteReduction* ukratko su opisani u glavi 4. Pregled arhitekture alata, detalji njegove implementacije i izgleda grafičkog interfejsa dati su u glavi 5. Prikazani su i rezultati rada alata *TestSuite-Reduction* na primeru dva projekta različitog obima. U poslednjoj glavi 6 iznet je zaključak rada.

Glava 2

Tehnike testiranja i pokrivenost koda

Različite tehnike testiranja mogu pomoći u detektovanju grešaka u sistemu. Osnovna dva pristupa testiranju softvera su testiranje **bele kutije** i testiranje **crne kutije**. Ova podela se vrši u zavisnosti od toga da li je poznato kako softver radi, tj. da li tester ima pristup kodu. Testiranje crne kutije se nekada naziva i funkcionalno testiranje ili testiranje ponašanja. Testiranje bele kutije zasnovano je na internim putanjama, strukturi i implementaciji softvera koji se testira i za razliku od testiranja crne kutije uglavnom zahteva programerske veštine [9]. Na osnovu koda je nekad moguće zaključiti za koje ulaze je manje ili više verovatno da softver izbaci grešku [23]. Postoji i treći pristup, testiranje *sive kutije*, koji podrazumeva posmatranje „kutije" dovoljno za razumevanje implementacije, nakon čega se, na osnovu stečenog znanja, biraju najrelevantniji testovi crne kutije.

2.1 Statičko i dinamičko testiranje

Testiranje softvera se može podeliti i na *dinamičko* i *statičko*, zavisno od toga da li je prilikom testiranja softver pokrenut. **Dinamičko testiranje crne kutije** podrazumeva upotrebu softvera kao što bi ga koristio pravi korisnik, bez poznavanja izvornog koda. U ovom slučaju definišu se *test slučajevi* (eng. *test cases*), u kojima su opisani ulazi i procedura koju je potrebno pratiti prilikom testiranja. Neki pristupi dinamičkom testiranju mogu otkriti dodatne greške, poput korišćenja softvera kao što bi to činio neiskusni korisnik, ili iz ugla zlonamernog korisnika, koji traži slabosti softvera koje se mogu iskoristiti za pristupanje vrednim podacima u sistemu. Primer

statičkog testiranja crne kutije je *testiranje specifikacije*, za koje nije neophodno poznavanje detalja procesa prikupljanja informacija, nego samo da je rezultat tog procesa specifikacija proizvoda, koja predstavlja, na primer, statički dokument, a ne program koji se izvršava.

Statičko testiranje bele kutije ili *strukturna analiza* (eng. *structural analysis*) je proces pažljivog i metodičkog pregledanja dizajna, arhitekture ili koda bez izvršavanja programa. Cilj ovakvog testiranja je rano pronalaženje grešaka i pronalaženje grešaka koje bi se teško mogle izolovati dinamičkim testiranjem crne kutije. *Pregled koda* (eng. *code review*) je jedan vid statičkog testiranja bele kutije i podrazumeva ljudsku kontrolu koda sa ciljem povećanja kvaliteta koda, smanjivanja broja grešaka i poštovanja pravila kodiranja i pisanja dokumentacije.

Dinamičko testiranje bele kutije podrazumeva korišćenje informacija dobijenih poznavanjem šta i kako kôd radi, da bi se donela odluka šta treba testirati i kako pristupiti testiranju. Naziva se i *strukturno testiranje* (eng. *structural testing*), jer se struktura koda može videti i upotrebiti za dizajniranje i pokretanje testova. Dinamičko testiranje bele kutije treba razlikovati od *debagovanja* (eng. *debugging*), kome je cilj ispravljanje greške, dok je cilj dinamičkog testiranja bele kutije pronalaženje grešaka. Preklapanje ove dve tehnike se ogleda u izdvajanju gde i zašto dolazi do defekta. Međutim, dinamičko testiranje bele kutije prethodi debagovanju, izdvajajući najmanji test slučaj kojim se demonstrira greška, a nekada i tačnu liniju koda koja je može proizvoditi [23].

2.2 Nivoi testiranja

Testiranje se može vršiti na četiri različita nivoa. *Jedinica* je „najmanji deo” softvera koje programer može napraviti (npr. funkcija ili klasa, u zavisnosti od programskog jezika), a testiranje na tom nivou se naziva **testiranje jedinica koda** (eng. *unit testing*). Na narednom nivou se testira kako jedinice međusobno interaguju i naziva se **integraciono testiranje** (eng. *integration testing*). Integraciono testiranje na najnižem nivou se naziva *komponentno testiranje* (eng. *component testing*) – integrišu se osnovne jedinice koda i posmatra se komponenta izolovano od drugih komponenti i spoljašnjeg sistema. **Sistemsko testiranje** (eng. *system testing*) podrazumeva testiranje sistema u celosti, pri čemu se ispituje da li je ponašanje sistema u skladu sa definisanom specifikacijom posmatranjem ne samo softvera, već i hardvera, baze podataka i korisničkih uputstava. U sistemsko testiranje spadaju

istraživačko (eng. *exploratory*) i *testiranje prihvatljivosti* (eng. *acceptance testing*). Istraživačko testiranje se vrši kada je aplikacija u finalnom obliku i tada se proveravaju alternativni pravci korišćenja sistema, oni koji nisu bili predviđeni planom testiranja. Testiranje prihvatljivosti vrše sami klijenti i korisnici. Nakon uspešnog testa prihvatljivosti, proizvod se postavlja kod klijenta. **Regresiono testiranje** (eng. *regression testing*) se vrši nakon napravljenih izmena u sistemu da bi se utvrdilo da li promene utiču na očekivano ponašanje sistema. Sprovodi se da bi se testirale funkcionalne osobine sistema kao i njegove performanse. Problem regresionog testiranja je što može zahtevati mnogo vremena i resursa, zbog čega su razvijeni različiti pristupi regresionom testiranju: *redukovanje*, *selekcija* i *prioritetizacija* skupa testova [31]. U prvom pristupu se teži smanjivanju broja testova uz zadržavanje sposobnosti otkrivanja defekata. Kada se radi selekcija skupa testova, privremeno se bira podskup testova imajući u vidu napravljene izmene u kodu. Prioritetizacija podrazumeva uređivanje skupa testova tako da se maksimizuju neka poželjna svojstva (npr. stopa prepoznavanja grešaka), što znači da je moguće da će svi testovi biti izvršeni, ali i da testiranje može biti zaustavljeno u proizvoljnom trenutku tokom izvršavanja.

2.3 Testiranje crne kutije

Testiranje crne kutije predstavlja testiranje isključivo na osnovu zahteva i specifikacije sistema. Mana ove strategije je što se ne može znati tačno koliko sistema je testirano, ali se formalnim pristupom može doći do podskupova testova koji su efikasni i efektivni u pronalaženju defekata. Takođe, tester bi morao napraviti sve moguće kombinacije ulaza programa, validne i nevalidne, da bi pronašao sve greške u sistemu. Ovakvo iscrpno testiranje nikada nije moguće.

Nekoliko ilustrativnih primera praktične nemogućnosti iscrpnog testiranja crne kutije navedeno je u [21]. Jedan takav slučaj je testiranje kompilatora. Morali bi se napisati svi mogući ispravni programi, ali i oni neispravni, da bi se utvrdilo da li kompilator uspešno prijavljuje takve programe kao neispravne. Drugi primer su programi zasnovani na bazama podataka, kao što su sistemi za pravljenje rezervacija. Za testiranje takvih sistema bilo bi neophodno proveriti ne samo sve ispravne i neispravne rezervacije, već i nizove rezervacija, jer trenutni upit bazi zavisi od ishoda prethodnih.

Jedan formalan pristup testiranju crne kutije je *testiranje pomoću klasa ekvivalencije* (eng. *equivalence class testing*). *Klasa ekvivalencije* je skup podataka koje

softverski modul jednako tretira ili koji treba da proizvedu isti rezultat. Za svaku identifikovanu klasu ekvivalencije se pravi po test slučaj. Nakon identifikovanja klasa ekvivalencije, mogu se uočiti i njihove granice i na osnovu njih oblikovati testovi. Ovaj pristup se naziva *testiranje graničnih vrednosti* (eng. *boundary value testing*). Definišu se test slučajevi tako što se izabere jedan ulaz koji se nalazi na granici, jedan koji je „ispod" i jedan „iznad" granice [1].

Test slučajevi se mogu napraviti i uz pomoć *tabela odlučivanja* (eng. *decision table*). Tabela odlučivanja je tehnika za prikaz složenih poslovnih pravila u čitljivom obliku. Redovi u tabeli se dele na dva dela – prvi deo čine uslovi nad ulazom, a drugi moguće akcije. Kolone su pravila koja jedinstvenoj kombinaciji uslova dodeljuju odgovarajuće akcije. Ukoliko su uslovi iz jednog pravila binarni, iz njih se može izvesti tačno jedan test slučaj, inače ih može više proizilaziti iz jednog pravila.

Dijagram stanja (eng. *state-transition diagram*) opisuje kompleksne zahteve sistema i njegovu interakciju sa spoljašnjim svetom. Obilaskom dijagrama stanja, koji predstavlja vrstu usmerenog grafa, mogu se generisati test slučajevi. Dijagrami stanja se mogu koristiti za opisivanje zahteva koji se tiču stanja i njihovih promena, koje se dešavaju kao rezultat akcija.

Ponašanje sistema se može opisati i *tabelama stanja* (eng. *state transition tables*). Dok su dijagrami stanja možda lakši za razumevanje, tabele stanja mogu biti lakše za upotrebu na kompletan i sistematičan način [9]. Prednost ovog pristupa je izlistavanje *svih* kombinacija promena stanja, ne samo validnih. Prilikom testiranja kritičnih, visokorizičnih sistema, može biti neophodno testirati i nevalidne kombinacije. Loša strana tabela stanja je što se njihova veličina brzo povećava sa porastom broja stanja i događaja. Takođe, veliki broj ćelija takvih tabela je prazan.

Pogađanje grešaka (eng. *error guessing*) je tehnika testiranja crne kutije koja se pre svega oslanja na iskustvo i procenu osobe koja testira sistem. Za nju ne postoje specifični alati, već se na osnovu iskustva pretpostavlja koji delovi koda mogu sadržati greške.

2.4 Testiranje bele kutije

Testiranje bele kutije podrazumeva poznavanje unutrašnje strukture softvera. Često se izjednačava sa testiranjem jedinica koda koje izvode programeri, iako je u

[1] Šta tačno predstavlja iznad, a šta ispod granice, zavisiće od jedinice u kojoj su podaci čuvani u sistemu.

pitanju širi pojam [9]. Ovom vrstom testiranja se ispituju različite putanje unutar modula, između modula unutar podsistema, između podsistema unutar sistema ili između čitavih sistema.

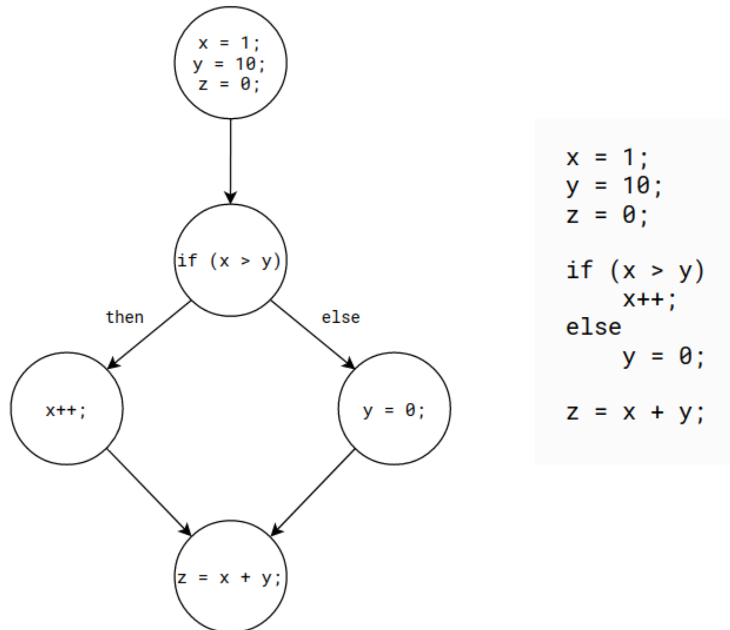
Mana testiranja bele kutije je broj izvršnih putanja sistema, koji može biti toliko veliki da se ne mogu sve putanje testirati, analogno slučaju ispitivanja svih kombinacija ulaza kod modela testiranja crne kutije. Takođe, odabrani test slučajevi možda ne obuhvataju greške koje zavise od podataka, putanje koje ne postoje u *kontroli toka* (eng. *control flow*) neće biti testirane i osoba koja vrši testiranje mora imati programerskog znanja.

Testiranje kontrole toka (eng. *control flow testing*) je jedna od tehnika testiranja bele kutije. Podrazumeva identifikovanje putanja izvršavanja koda na osnovu kojih se prave i izvršavaju test slučajevi. Slično kao kod testiranja crne kutije, pristup kojim bi se testirale sve putanje u programu je nepraktičan, a često i nemoguć, zbog prevelikog broja putanja u jednom programu. Svako grananje udvostručuje broj putanja, a svaka petlja umnožava broj putanja onoliko puta koliko ima iteracija u petlji. Stoga, postoje tehnike testiranja koje omogućavaju kreiranje skupa test slučajeva koje je vremenski moguće testirati, uz obezbeđivanje visokog nivoa *pokrivenosti koda* (eng. *code coverage*). Različiti tipovi pokrivenosti biće opisani u 2.5. *Graf kontrole toka* (eng. *control flow graph*) je osnova testiranja kontrole toka. Za definisanje grafa kontrole toka, potrebno je prvo definisati *osnovni blok* (eng. *basic block*). To je linearna sekvenca instrukcija programa koja ima jednu ulaznu i jednu izlaznu tačku, tj. jednu početnu i jednu završnu instrukciju. Graf kontrole toka je usmereni graf čiji su čvorovi osnovni blokovi, a grane predstavljaju putanje po kojima teče izvršavanje. Primer jednostavnog programa i njegovog odgovarajućeg grafa kontrole toka dat je na slici 2.1.

Testiranje grafa toka podataka (eng. *data flow testing*) je alat za uočavanje neispravne upotrebe podataka nastalih kao rezultat grešaka u kodu. Česta greška u programiranju je referisanje na vrednost promenljive kojoj pre toga nije dodeljena vrednost [9]. Promenljive u kodu imaju svoj životni ciklus – stvaraju se, upotrebljavaju i na kraju uništavaju. U nekim programskim jezicima su stvaranje i uništavanje promenljivih automatski (npr. Java[2]), a u nekima moraju biti eksplicitno navedeni (npr. C[3]). Graf toka podataka prikazuje životni ciklus svake promenljive

[2]Iako je upravljanje memorijom automatsko, Java nudi mogućnost eksplicitnog oslobađanja memorije.

[3]U C-u se memorijom upravlja ručno, ali postoji mogućnost korišćenja *skupljača otpadaka* (eng. *garbage collector*) *Boehm GC*-a.



Slika 2.1: Primer jednostavnog koda u programskom jeziku C (desno) i odgovarajućeg grafa kontrole toka (levo).

u jednom modulu. Testiranje ovakvih dijagrama podrazumeva verifikaciju uočenih ciklusa *stvaranja, upotrebe i uništavanja* (eng. *define-use-kill patterns*) i može mu se pristupiti na dva načina: *statički* – formalno ili neformalno izučavanje dijagrama, i *dinamički* – konstrukcija i izvršavanje test slučajeva. Testiranje grafa toka podataka se može smatrati nadogradnjom testiranja kontrole toka. Ograničenja ovakvog pristupa su vremenska zahtevnost, zbog broja putanja i promenljivih u programu, kao i neophodnost da tester poznaje kôd, njegovu kontrolu toka i promenljive.

Testiranje osnovnih putanja

Tehnika testiranja bliska testiranju kontrole toka je *testiranje osnovnih putanja* (eng. *basis path testing*). Zasniva se na radu Tomasa MekKejba[4] i koristi analizu topologije grafa kontrole toka za pronalaženje test slučajeva. Proces testiranja osnovnih putanja sadržan je u narednim koracima:

1. Izgraditi graf kontrole toka softverskog modula.
2. Izračunati *ciklomatičnu kompleksnost CC* (eng. *cyclomatic complexity*) grafa.

[4] Tomas MekKejb (*Thomas McCabe*), matematičar.

3. Odabrati skup od ukupno CC osnovnih putanja.

4. Napraviti po test slučaj za svaku osnovnu putanju.

5. Izvršiti testove.

Ciklomatična kompleksnost je metrika kojom se opisuje kompleksnost softvera. Definiše se uz pomoć sledeće jednačine:

$$CC = broj_grana - broj_čvorova + 2 \cdot broj_povezanih_komponenti$$

Ciklomatična kompleksnost predstavlja najmanji broj nezavisnih putanja koje ne sadrže petlje (osnovnih putanja), čija linearna kombinacija obuhvata sve moguće putanje kroz modul. Posmatrano u terminima grafa toka, svaka osnovna putanja obilazi barem jednu granu koju nijedna druga putanja ne obilazi. Testiranje osnovnih putanja, što podrazumeva pravljenje i izvršavanje CC test slučajeva, garantuje pokrivenost grana i naredbi, s obzirom da skup osnovnih putanja obuhvata sve grane i čvorove grafa kontrole toka. Ograničenje ovakvog pristupa je što nije uvek moguće napraviti test za svaku osnovnu putanju nekog modula. To je slučaj kada ni za jedan ulaz program ne prolazi kroz tu putanju tokom izvršavanja, tj. putanja je nedostižna.

Na slici 2.2 je prikazan primer grafa kontrole toka i izračunata je njegova ciklomatična kompleksnost. Skup osnovnih putanja ovog grafa su:

I putanja: 1 - 2 - 3 - 4 - 2 - 3 - 5 - 6 - 5

II putanja: 1 - 3 - 4 - 2 - 3 - 5 - 6 - 5

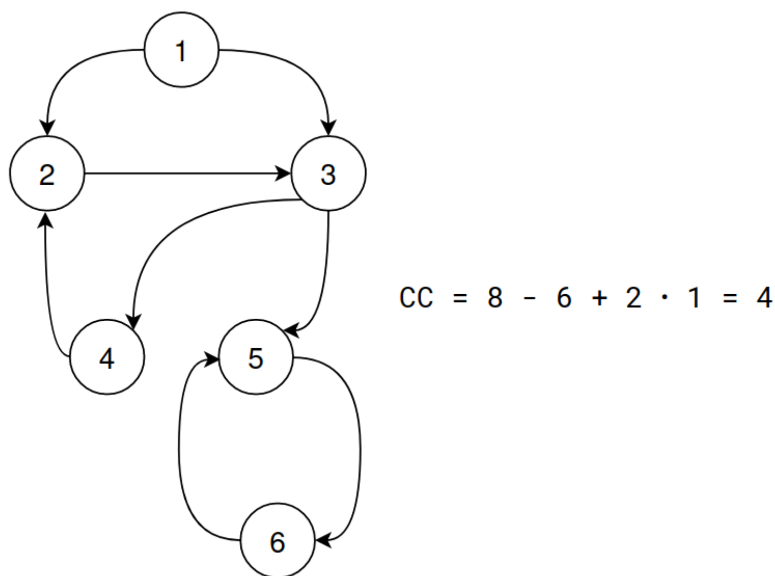
III putanja: 1 - 2 - 3 - 5 - 6 - 5

IV putanja: 1 - 3 - 5 - 6 - 5

Do osnovnih putanja se dolazi tako što se izabere jedna proizvoljna putanja u grafu (putanja 1 - 2 - 3 - 4 - 2 - 3 - 5 - 6 - 5). Zatim, druga putanja se dobija praćenjem toka prve putanje polazeći od ulaznog čvora, menjajući samo prvu napravljenu odluku (u čvoru 1). Odluke u ostatku te putanje se mogu birati proizvoljno sve do mesta gde se druga putanja ne spaja sa prvom, nakon čega se nastavlja redosledom iz prve putanje, tj. prave se iste odluke. U ovom slučaju, druga putanja se spaja sa prvom već u narednom čvoru, tj. u čvoru 3, nakon čega su ove dve putanje iste. Treća putanja se dobija promenom druge napravljene odluke, tj. u čvoru 3 se pravi

odluka da se pređe u čvor 5 umesto u čvor 4, čime se dobija putanja 1 - 2 - 3 - 5 - 6 - 5. Kad više nema putanja koje se mogu dobiti menjanjem odluka u prvoj putanji, pokušava se sa sličnim koracima polazeći od druge putanje – menja se prva nova odluka, odnosno ona koja nije postojala u početnoj putanji. Dakle, polazeći od druge putanje, menja se odluka napravljena u čvoru 3 i ovog puta se bira čvor 5 i time dobija putanja 1 - 3 - 5 - 6 - 5. Ovaj postupak odabira osnovnih putanja bliže je opisan u [29].

Izračunata vrednost ciklomatične kompleksnosti predstavlja gornju granicu broja testova koji se moraju izvesti da bi svaka naredba programa bila izvršena barem jedanput. Ova vrednost se definiše i pomoću $CC = P + 1$, gde je P broj *predikatskih čvorova* (eng. *predicate nodes*), tj. čvorova koji predstavljaju grananje u kodu.



Slika 2.2: Primer grafa kontrole toka i njegova ciklomatična kompleksnost.

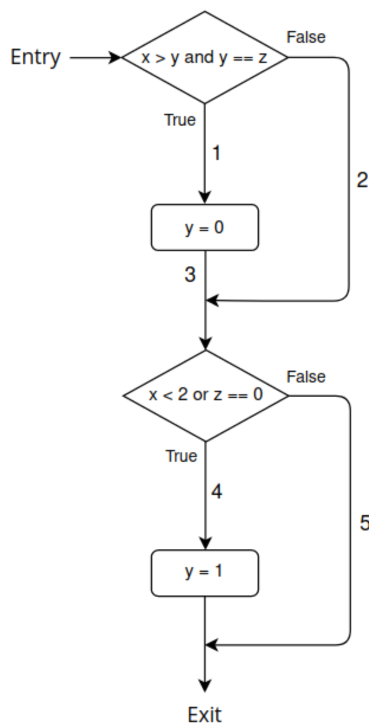
Analiza kompleksnosti može pomoći u povećavanju kvaliteta softvera [29]. Isuviše kompleksni moduli su podložniji greškama, teže ih je razumeti, testirati i menjati, stoga ograničavanje ciklomatične kompleksnosti koda u svakoj fazi razvoja softvera može ublažiti ove poteškoće. Tačna gornja granica se ne može precizno definisati za opšti slučaj, ali se nekad zadaje kao broj 10, uz naglašavanje da postoje programi za koje je veća gornja granica prihvatljiva, kao i da nekada ima smisla ne uračunavati *switch* odnosno *case* naredbe u ciklomatičnu kompleksnost modula.

2.5 Pokrivenost koda

Prilikom testiranja kontrole toka koristi se graf kontrole toka, pri čemu se teži dostizanju što veće pokrivenosti koda. Pokrivenost se može smatrati brojem *elemenata* programa (eng. *items*) koji su ispitani testovima u odnosu na ukupan broj tih elemenata. Pokrivenost se definiše na više različitih nivoa:

- **Pokrivenost naredbi** (eng. *statement coverage*) – mera izvršavanja naredbi u programu. Potpuna pokrivenost naredbi znači da je svaka naredba izvršena barem jedanput.
- **Pokrivenost putanja** (eng. *path coverage*) – mera pokrivenosti kroz sve moguće putanje. Postignuta potpuna pokrivenost znači da je svaka putanja programa izvršena barem jedanput. Ukoliko program ne sadrži petlje i veliki broj grananja, onda je moguće napraviti test slučaj za svaku putanju. Međutim, ako program sadrži petlje, broj svih putanja postaje toliko veliki da testiranje svake predstavlja nesavladiv problem [9, 21].
- **Pokrivenost grana/odluka** (eng. *branch/decision coverage*) – mera prolaska kroz sve grane programa. Da bi pokrivenost grana bila potpuna, neophodno je da svaka odluka u programu bude doneta barem jednom.
- **Pokrivenost uslova** (eng. *condition coverage*) – mera ispitivanja uslova u programu. Za razliku od pokrivenosti grana, da bi pokrivenost uslova bila potpuna, neophodno je da svaki uslov u svakoj odluci barem jednom dobije svaku od mogućih vrednosti.
- **Pokrivenost višestrukih uslova** (eng. *multiple condition coverage*) – mera ispitivanja višestrukih uslova u programu. U ovom slučaju neophodno je barem jednom ispitati svaku moguću kombinaciju uslova u svakoj odluci.
- **Pokrivenost funkcija** (eng. *function coverage*) – mera poziva svih funkcija. Potpuna pokrivenost funkcija jednog programa znači da je svaka funkcija barem jednom pozvana.

Za bolje razumevanje razlika i sličnosti između opisanih kriterijuma pokrivenosti koda u nastavku će biti detaljnije objašnjeni kroz primer kratkog programa, koji je dat na slici 2.3. Da bi bio postignut bilo koji od opisanih nivoa pokrivenosti potrebno je pažljivo odabrati ulazne vrednosti.



Slika 2.3: Primer jednostavnog programa koji treba testirati, sa jednom ulaznom tačkom i označenim delovima putanja kroz koje se može proći prilikom izvršavanja.

- Da bi kriterijum *pokrivenosti naredbi* bio zadovoljen, dovoljno je, na primer, odabrati naredni test:

  `x=1, y=0, z=0` (putanja 1-3-4)

- Potpuna *pokrivenost putanja* se može postići pomoću četiri testa:

  `x=1, y=0, z=0` (putanja 1-3-4)

  `x=2, y=1, z=1` (putanja 1-3-5)

  `x=2, y=2, z=1` (putanja 2-3-5)

  `x=1, y=2, z=3` (putanja 2-3-4)

- *Pokrivenost odluka* se može zadovoljiti odabirom narednih ulaza:

  `x=-1, y=-1, z=-1` (putanja 2-3-4)

  `x=1, y=-1, z=-1` (putanja 1-3-5)

  Ovaj kriterijum može propustiti neke greške. Na primer, ukoliko je u drugom uslovu napravljena greška u kodu i ako je trebalo da bude `z!=0` umesto `z==0`

nijedan od izabrana dva testa neće naići na taj propust. U takvim slučajevima zahtevanje pokrivenosti uslova obezbeđuje proveru oba ishoda (`True` i `False`) svakog uslova u jednoj odluci.

- Za ispunjavanje *pokrivenosti uslova.* potrebno je izabrati testove za koje će u prvom grananju barem jednom važiti `x>y`, `x<=y`, `y==z` i `y!=z`, a u drugom grananju `x<2`, `x>=2`, `z==0` i `z!=0`. Naredni testovi ispunjavaju potpunu pokrivenost uslova, jer svaki od uslova u svakoj odluci uzima obe vrednosti:

  `x=1, y=0, z=1` (putanja 2-3-5)

  `x=0, y=0, z=0` (putanja 2-3-4)

  `x=2, y=1, z=0` (putanja 2-3-4)

- *Pokrivenost višestrukih uslova* zahteva testove koji će obezbediti proveru svih kombinacija ishoda svih uslova u svakoj odluci koja se pravi u programu. U ovom primeru, četiri test slučaja mogu obezbediti pokrivenost višestrukih uslova, koji se mogu videti u tabeli 2.1, kao i vrednosti koje imaju svi uslovi u datom programu u svakom od slučajeva.

- Ukoliko dati primer predstavlja telo neke funkcije koja kao argumente ima promenljive `x`, `y` i `z` i koja je jedina koja je u programu definisana, bilo koji od navedenih ulaza obezbedio bi *pokrivenost funkcija.*

Primer ulaza naveden kao dovoljan za ispunjavanje pokrivenosti uslova pokazuje i da pokrivenost odluka ne sledi uvek iz pokrivenosti uslova. Takođe, ispunjenost oba ova kriterijuma ne znači nužno da su sve naredbe izvršene. Slučajevi u kojima program ima više ulaznih tačaka, programi sa *obradom izuzetaka* (eng. *exception handling*) i programi koji nemaju grananja zahtevaju dodatne test slučajeve da bi se dostigla potpuna pokrivenost naredbi.

Test slučajevi	`x > y`	`y == z`	`x < 2`	`z == 0`	**Putanja**
`x = 2, y = 2, z = 1`	F	F	F	F	2-3-5
`x = 3, y = 2, z = 0`	T	F	F	T	2-3-5
`x = 1, y = -1, z = -1`	F	T	T	F	1-3-4
`x = 1, y = 0, z = 0`	T	T	T	T	1-3-4

Tabela 2.1: Test slučajevi čijim pokretanjem se postiže potpuna pokrivenost višestrukih uslova programa sa slike 2.3, sa pregledom vrednosti svakog uslova i putanje koju obilaze.

Alati za analizu pokrivenosti koda

Alati za analizu pokrivenosti koda (eng. *code coverage analysers*) instrumentalizuju kôd koji nadgledaju da bi tokom izvršavanja programa mogli beležiti koja je linija, grana ili funkcija izvršena. U tabeli 2.2 mogu se videti različiti alati koji služe za analizu C++ koda, kao i nivoi pokrivenosti koje podržavaju [24, 16, 25, 14, 22]. Uz pomoć informacija koje alati za analizu pokrivenosti prikupljaju, moguće je otkriti koji moduli ili delovi modula se nikada ne izvršavaju tokom testiranja. Takođe, ovi alati ukazuju na nedovoljno pokrivene delove koda, kao i na redundantne test slučajeve.

Alat	Putanje	Naredbe	Grane	Uslovi	Višestruki uslovi	Funkcije
Gcov	-	+	+	-	-	+
Testwell CTC++	-	+	+	+	+	+
BullseyeCoverage	-	-	+	+	+	+
FrogLogic Coco	-	+	+	+	+	+
Parasoft C/C++test	+	+	+	+	-	+

Tabela 2.2: Alati za analizu pokrivenosti programa napisanih u C++-u sa poređenjem podržanih nivoa pokrivenosti koda.

Glava 3

Pristupi rešavanju problema

U ovoj glavi se uvode oznake i precizne definicije pojmova, koje se koriste radi opisivanja modela na osnovu kojeg je vršena implementacija alata koji uklanja redundantne testove iz datog skupa testova. Polazi se od definisanja modela u opštim terminima, a zatim se opisuje model koji je zasnovan na pokrivenosti i koji je u okviru rada implementiran. Postoje i modeli koji se oslanjaju na tehnike koje se koriste u istraživanju podataka, kao što je prioritetizacija skupa testova na osnovu sličnosti [20], ali predstavljanje ovakvih modela prevazilazi okvire ovog rada.

3.1 Redundantni testovi

Test je redundantan ukoliko nakon njegovog uklanjanja iz skupa testova uspešnost pronalaženja defekata u programu ostaje ista [18]. U terminima pokrivenosti koda, test se smatra redundantnim ako nakon njegovog uklanjanja pokrivenost koda ostaje nepromenjena. Ovo se naziva *semantička redundantnost* testova (eng. *semantic redundancy*). Postoji i *sintaksička redundantnost* testova (eng. *syntactic redundancy*), koja se odnosi na ponavljanje koda testova. Semantički redundantne testove je, u opštem slučaju, teže otkriti od sintaksički redundantnih [12].

U kontekstu testiranja pomoću klasa ekvivalencije, ukoliko se dodavanjem nekog test slučaja ne poveća pokrivenost koda, taj test slučaj je verovatno u istoj klasi ekvivalencije kao neki postojeći test slučajevi [23]. Zato se često pojednostavljeno smatra da su redundantni oni testovi koji se nalaze u istoj klasi ekvivalencije. U okviru jedne klase ekvivalencije testovi se mogu razlikovati prema pokrivenosti koju dostižu i na osnovu toga se može odlučivati da li su dva testa iz iste klase ekvivalencije redundantna.

Jedan test slučaj može biti i *parcijalno redundantan* (eng. *partially redundant*). Može se definisati kao test slučaj koji ima barem jednu zajedničku ekvivalentnu naredbu sa nekim drugim test slučajem. Naredba testa je naredba koja sačinjava kôd samog testa[1]. Primer takvih test slučajeva su oni koji pokreću isti kôd programa koji se testira, ali vrše različite provere rezultata koji se dobija. U tom slučaju bi se razlikovale naredbe testa kojima se proverava rezultat. Uklanjanje celokupnih testova može pogoršati efikasnost i sposobnost skupa testova da pronađe greške. Jedno moguće rešenje dato je u [28], gde se formira model koji oslikava vezu naredbi testova i stanja kroz koja testovi prolaze, uzimajući u obzir i ulazne vrednosti. Ovaj model uklanja naredbe iz testova, a ne čitave test slučajeve.

Svođenje na problem minimalnog pokrivanja skupa

Kažemo da svaki test t iz skupa testova S izlaže proveri neki skup elemenata, koji će u nastavku biti označen sa $\varphi(t)$. Skup elemenata $\varphi(t)$ može biti, na primer, skup koji sadrži putanju u programu kroz koju se prolazi izvršavanjem testa t ili skup naredbi kroz koje se prolazi izvršavanjem testa t. Uvodimo sledeću definiciju, kojom se opisuje slučaj u kome će se smatrati da su dva testa *duplikati*:

Definicija 1 *Testovi* $t_1, t_2 \in S$ *su duplikati akko* $\varphi(t_1) = \varphi(t_2)$.

Postojanje testova koji su duplikati ukazuje na postojanje redundantnosti u skupu testova. Pronalaženje i eliminisanje svih duplikata predstavlja osnovnu varijantu rešenja problema redundantnih testova. Izbor i broj parova duplikata se menja u zavisnosti od definicije skupa φ, pa treba pažljivo razmotriti različite mogućnosti.

Problem pronalaženja ovako definisanih[2] redundantnih testova je ekvivalentan problemu pronalaženja minimalnog podskupa skupa testova koji u sebi ne sadrži duplikate. Preciznije:

Definicija 2 *Problem pronalaženja redundantnih testova se svodi na odabir minimalnog skupa* P*, gde je* P *podskup skupa svih postojećih testova* S*, takav da za svaka dva testa* $t_1, t_2 \in P$ *važi* $\varphi(t_1) \neq \varphi(t_2)$*, pri čemu je* $\bigcup_{t \in P} \varphi(t) = \bigcup_{t \in S} \varphi(t)$.

[1] Na primer, naredba testa može biti inicijalizacija promenljive ili provera da li je dobijen rezultat očekivan.

[2] Slučaj u kome važi $\varphi(t_1) \subset \varphi(t_2)$ ukazuje na redundantnost testa t_1, ali nije pokriven definicijom 1.

Na taj način se problem automatskog uklanjanja redundantnih testova iz datog skupa S može svesti na problem *minimalnog pokrivanja skupa* (eng. *set cover problem*). Problem minimalnog pokrivanja skupa se nalazi u listi *Karpovih 21 NP-kompletnih problema*[3].

Saglasnost prilikom eliminacije redundantnosti

Model za određivanje redundantnih test slučajeva je *saglasan* (eng. *sound*) ukoliko važi da je test proglašen redundantnim zaista redundantan. Međutim, iako se ka saglasnosti teži, zarad efikasnosti procesa pronalaženja redundantnih testova, neophodno je odreći se saglasnosti. Udaljavanje od saglasnosti se naziva *nagodbom između saglasnosti i performansi* (eng. *trading soundness for performance*) [6]. Dakle, u modelima koji nisu saglasni, test može biti proglašen redundantnim iako to nije [19].

3.2 Osnova za izgradnju modela

Pored mogućnosti postojanja duplikata u skupu testova, može se dogoditi da za tri testa t_1, t_2 i t_3 važi $\varphi(t_1) \cup \varphi(t_2) = \varphi(t_3)$. Da bi takvi slučajevi bili obuhvaćeni, potrebno je da posmatramo ponašanje testova na delovima koda, koje ćemo nadalje nazivati *jedinice koda*. Jedinica koda može biti, na primer, izvorna datoteka, klasa, funkcija ili blok koda. Ukoliko je odabrana jedinica koda dovoljno mala, mogla bi se uočiti jednakost na nivou te jedinice koda upoređivanjem skupova $\varphi(t_1)$ i $\varphi(t_3)$, odnosno $\varphi(t_2)$ i $\varphi(t_3)$. Slično važi kada je $\varphi(t_1) \subset \varphi(t_2)$ – u okviru određene jedinice koda bi se moglo uočiti da testovi t_1 i t_2 izlažu proveri isti skup elemenata.

Za svaki test t i jedinicu koda u uvodi se oznaka $\Phi(t, u)$, koja je restrikcija od $\varphi(t)$ na jedinicu koda u. Da bi se pojam redundantnosti formalno definisao, potrebno je precizirati i šta označava ekvivalentnost dva skupa testova u odnosu na neku jedinicu koda.

Definicija 3 *Skupovi testova* $S_1 = \{t_1, ..., t_n\}$ *i* $S_2 = \{t'_1, ..., t'_m\}$ *su ekvivalentni u odnosu na jedinicu koda u, u oznaci* $S_1 \equiv_u S_2$ *akko važi da je* $\bigcup_{t \in S_1} \Phi(t, u) = \bigcup_{t \in S_2} \Phi(t, u)$.

[3] Ričard Karp (*Richard Manning Karp*), američki naučnik, dobitnik Tjuringove nagrade 1985. godine. U radu iz 1972. godine sastavio je listu od 21 problema za koje je, svođenjem na SAT, dokazao NP-kompletnost.

Neka $\Psi(S)$ označava skup svih jedinica koda kroz koje je tok izvršavanja programa prošao pokretanjem testova iz skupa S. Pod pretpostavkom raspoloživosti svih skupova $\Psi(S)$, može se zadati opštija definicija redundantnosti.

Definicija 4 *Testovi iz skupa* $R = \{t_1, ..., t_n\}$ *su redundantni u skupu testova* $S \supseteq R$ *akko* $(\forall u \in \Psi(S))\ S \equiv_u S \setminus R$.

Matrica pokrivenosti

Neka je matrica $\mathcal{M}_{i,j} = \Phi(t_i, u_j)$ dimenzije $n \times m$, gde je n broj testova, a m ukupan broj jedinica koda, definisana sa:

$$\begin{array}{c} \\ t_1 \\ t_2 \\ \\ t_n \end{array} \begin{array}{c} \begin{array}{cccc} u_1 & u_2 & & u_m \end{array} \\ \begin{pmatrix} \Phi(t_1, u_1) & \Phi(t_1, u_2) & \dots & \Phi(t_1, u_m) \\ \Phi(t_2, u_1) & \Phi(t_2, u_2) & \dots & \Phi(t_2, u_m) \\ \vdots & \vdots & \ddots & \vdots \\ \Phi(t_n, u_1) & \Phi(t_n, u_2) & \dots & \Phi(t_n, u_m) \end{pmatrix} \end{array}$$

Ovu matricu nazivamo *matrica pokrivenosti* [7]. Veličina matrice pokrivenosti zavisi od broja testova i jedinica koda, koji se mogu brojati hiljadama i desetinama hiljada[4].

Na osnovu definicije 1, testovi t_i i t_j su duplikati ukoliko su i-ta i j-ta vrsta matrice $\mathcal{M}$ jednake, tj. sadrže iste elemente na istim indeksima, što zapravo znači da svaku jedinicu koda koju proveravaju testovi t_i i t_j, pokrivaju na isti način.

Na osnovu definicije 3, a u terminima matrice $\mathcal{M}$, važi da je $S_1 \equiv_u S_2$ akko je unija svih elemenata iz redova koji odgovaraju testovima skupa S_1 jednaka uniji svih redova koji odgovaraju testovima skupa S_2, posmatrajući samo vrednosti iz kolone koja odgovara jedinici koda u. Konačno, definicija 4 daje kriterijum kojim se iz početnog skupa testova mogu eliminisati redundantni – potrebno je pronaći one redove čijim se izbacivanjem dobija matrica $\mathcal{M}'$ dimenzije $n' \times m$, gde je $n' < n$, takva da je $\bigcup\limits_i \mathcal{M}'_{i,j} = \bigcup\limits_i \mathcal{M}_{i,j}$ za svaku kolonu u_j, odnosno, za svaku jedinicu koda. Dakle, unija svih elemenata u koloni koja odgovara jedinici koda u u matrici $\mathcal{M}'$ jednaka je uniji svih elemenata iste kolone u polaznoj matrici $\mathcal{M}$.

[4]U odeljku 5.4 je dat primer projekta uključujući i informaciju o broju testova i izvornih datoteka tog projekta.

Uvedeni pojmovi u kontekstu grafa kontrole toka

Podsetimo se, graf kontrole toka je usmereni graf čiji su čvorovi osnovni blokovi, a grane predstavljaju tok izvršavanja. Posmatranje grafa kontrole toka programa može se uzeti kao motivacija za pravljenje modela koji je saglasan. Element iz $\varphi(t)$ može biti putanja u takvom grafu koja se obilazi prilikom pokretanja testa t. Ukoliko dva testa obilaze istu putanju toka, onda u redukovanom skupu nije potrebno zadržati oba testa.

Ograničenje ovako definisanog modela se ogleda u složenosti postupka kreiranja grafa [17], a dodatno se usložnjava računanjem skupa $\Phi(t, u)$ za svako t i sve $u \in \Psi(S)$, odnosno kreiranjem matrice za ovako odabrane skupove. U sledećem delu biće uvedena metrika kojom se model udaljava od saglasnosti, sa ciljem povećavanja efikasnosti.

3.3 Model zasnovan na pokrivenosti

Do sada je opisana ideja za pronalaženje redundantnih testova koja se svodi na traženje onih redova u matrici pokrivenosti čijim se izbacivanjem ne menja unija svih elemenata te matrice. Opštost ovako uvedenog modela daje mogućnost promene pojedinih odluka, a da se pritom zadrži isti postupak. U nastavku je opisan model koji predstavlja unapređenje pristupa iz rada [7].

Neka je skup elemenata koje test t izlaže proveri, odnosno $\varphi(t)$, skup koji za svaku jedinicu koda sadrži informaciju o pokrivenosti koda koju test t dostiže tokom izvršavanja. Pretpostavka koja se pravi je da procenat pokrivenosti koda omogućava razlikovanje elemenata koji su izloženi proveri. Ova pretpostavka dovodi do grube aproksimacije, ali se ona ipak pravi zarad implementacije prototipa alata za uklanjanje redundantnih testova. U skladu sa ovim, $\Phi(t, u)$ se tumači na sledeći način:

Definicija 5 *Neka prilikom pokretanja testa t pokrivenost naredbi jedinice koda u iznosi* $c \in [0, 100]$. *Funkcija* $\Phi(t, u)$ *se preslikava u jednočlan skup* $\{(c, u)\}$.

Ovakvim pristupom se ne uzima u obzir redosled izvršavanja naredbi, stoga se ne može garantovati ista putanja u grafu kontrole toka. Time što se izolovano posmatraju jedinice koda u, stvara se dodatna apstrakcija. Relacija $\equiv_u$ iz definicije 4 sakriva informacije o redosledu izvršavanja pojedinačnih jedinica koda u.

Ukoliko bi za jedinice koda bile odabrane metode, naredna definicija, koja se navodi u [30], u skladu je sa specijalnim slučajem definicije 4 za $n = 1$, tj. za slučaj jednog testa:

Definicija 6 *Test t je redundantan u odnosu na skup testova S akko za svaki metod koji t izvršava, postoji drugi test iz skupa S koji ga izvršava na ekvivalentan način.*

Precizniji oblik definicije 6 može se dobiti ukoliko se iskoristi tumačenje zasnovano na pokrivenosti. To je ujedno i opis pristupa korišćenog u implementaciji i dat je narednom definicijom:

Definicija 7 *Test t je redundantan u odnosu na skup testova S akko za svaki metod koji t izvršava, postoji drugi test iz skupa S koji ga izvršava sa istim procentom pokrivenosti naredbi.*

Redundantnost testova se može definisati na analogan način za drugi nivo pokrivenosti ili drugu jedinicu koda.

Osnovni pristup je, dakle, izdvajanje skupa testova sa minimalnom kardinalnošću odbacivanjem testova koji su redundantni kao što je definisano u 7. Posmatrano kroz matricu pokrivenosti, to znači da ukoliko u svakoj koloni matrice izolovano grupišemo vrste prema vrednosti, onda je iz svake takve grupe neophodno zadržati najmanje jedan test, tj. vrstu. Time svaki procenat pokrivenosti za jednu jedinicu koda ima svog „predstavnika" u vidu zadržanog testa.

Postupak redukovanja skupa testova na osnovu pokrivenosti koda se može opisati sledećim koracima:

1. Izgraditi matricu $\mathcal{M}$
2. Izvršiti grupisanje vrednosti po kolonama matrice $\mathcal{M}$ – izdvojiti konkretne procente pokrivenosti c za svaku jedinicu koda u
3. Prevesti tako grupisane vrednosti u ograničenja u formatu koji prepoznaje *rešavač ograničenja* (eng. *constraint solver*) – obezbediti zadržavanje barem jednog testa koji daje procenat pokrivenosti c (posmatrajući jedinicu koda u izolovano). Ovaj korak je bliže opisan u delu 3.3
4. Korišćenjem rešavača ograničenja pronaći rešenje koje ispunjava zadate uslove, pritom zahtevajući da rešenje sadrži minimalan broj testova

5. Prevesti dobijeno rešenje u polazni domen skupa testova – rezultat rešavača prevesti u skup naziva testova, što predstavlja redukovan skup testova

Poslednji korak je potreban ukoliko nazivi promenljivih koje se koriste prilikom definisanja ograničenja ne odgovaraju nazivima testova, već su na neki način kodirani[5]. Ovako opisani model predstavlja instancu problema *zadovoljavanja ograničenja* (eng. *constraint satisfaction problem - CSP*), koji je NP-težak problem [7]. Za dati skup ograničenja, CSP problem predstavlja problem pronalaženja vrednosti koje treba dodeliti promenljivama koje se pojavljuju u ograničenjima (iz njima odgovarajućih domena) tako da su ograničenja zadovoljena. Predstavljeni postupak formiranja matrice pokrivenosti, na osnovu koje se definišu ograničenja kojima se opisuje redukovani skup, potiče iz rada [7]. Razlika se ogleda u tome što se pokrivenost u tom radu posmatra binarno. Drugim rečima, ne razmatra se procenat pokrivenosti, već samo da li je jedinica koda pokrivena ili ne, pa u tom slučaju ne postoji potreba za drugim korakom, a treći korak je pojednostavljen. Posmatraju se samo oni elementi matrice koji imaju vrednost 1 i za svaku jedinicu koda se definiše uslov da barem jedan od testova koji pokriva tu jedinicu koda bude zadržan u rezultujućem redukovanom skupu.

Prevođenje matrice pokrivenosti u ograničenja

Na osnovu formirane matrice pokrivenosti, potrebno je oblikovati ograničenja koja oslikavaju podatke iz matrice. Zadovoljavanjem tako formiranih uslova, cilj je doći do redukovanog skupa testova. Tokom prevođenja matrice pokrivenosti u ograničenja, za svaki procenat pokrivenosti c jedinice koda u dodaje se uslov kojim se zahteva da se zadrži barem jedan test sa tom pokrivenošću.

Neka testovi $t_1, t_2, ..., t_n$ dostižu procente pokrivenosti jedinice koda u redom $c_1, c_2, ..., c_n$, odnosno neka je data kolona matrice pokrivenosti koja odgovara jedinici koda u:

$$\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}$$

[5] Primer takvog kodiranja je kada se, umesto punog naziva testa, na primer, `TestSkup.NazivTesta`, koristi skraćeni naziv `t1`.

U ovoj koloni mogu postojati elementi koji imaju jednake vrednosti. Neka k od ukupno n elemenata[6] ima istu vrednost v. Označimo te elemente sa $c_{i_1}, c_{i_2}, ..., c_{i_k}$ i neka oni predstavljaju procente pokrivenosti koje postižu redom testovi $t_{i_1}, t_{i_2}, ..., t_{i_k}$. Za svaku vrednost v u jednoj koloni, formira se uslov kojim se zahteva zadržavanje barem jednog od testova $t_{i_1}, t_{i_2}, ..., t_{i_k}$ u rezultujućem skupu testova. Svakom testu se dodeljuje vrednost $\top$ (ako se test zadržava u redukovanom skupu testova) ili $\bot$ (ako se test uklanja iz skupa testova, tj. proglašava se redundantnim). Zatim, za svaku vrednost v se formira uslov koji se može opisati narednom logičkom formulom:

$$t_{i_1} \lor t_{i_2} \lor \ldots \lor t_{i_k}$$

U nastavku se ovaj uslov obeležava skraćeno T_v. Neka se u koloni koja odgovara jedinici koda u pojavljuje m različitih vrednosti $v_1, v_2, ..., v_m$ i neka su uslovi koji se formiraju za svaku od tih vrednosti redom $T_{v_1}, T_{v_2}, ..., T_{v_m}$. Za svaku kolonu se, zatim, pravi naredni uslov:

$$T_{v_1} \land T_{v_2} \land \ldots \land T_{v_k} \tag{3.1}$$

Na ovako opisan način se formiraju ograničenja[7] za svaku kolonu matrice pokrivenosti, koja se prosleđuju rešavaču ograničenja u formatu koji on prepoznaje. Pored opisanih uslova, rešavaču se zadaje i naredni cilj kojim se zahteva da ukupan broj testova u redukovanom skupu budu minimalan:

$$\min \sum_i I(t_i)$$

gde $I(t_i)$ označava:

$$I(t_i) = \begin{cases} 1\ , & t_i \text{ ima vrednost } \top \\ 0\ , & t_i \text{ ima vrednost } \bot \end{cases}$$

Rešavač, zatim, treba da pronađe skup testova koji zadovoljava zadata ograničenja tako što će svakom testu dodeliti vrednost $\top$ ili $\bot$, pri čemu broj testova kojima je dodeljena vrednost $\top$ treba da bude minimalan.

[6] Ovi elementi nisu nužno susedni elementi u koloni.

[7] S obzirom da svaka formula T_v čini disjunkciju i da uslov 3.1 predstavlja konjunkciju formula T_v, formula 3.1 je u *konjunktivnoj normalnoj formi* [13].

Glava 4

Korišćeni alati i tehnologije

Alat implementiran na osnovu modela zasnovanog na pokrivenosti, opisanog u potpoglavlju 3.3 prethodne glave, oslanja se na postojeće tehnologije i alate o kojima će biti više reči u nastavku. Biće ukratko objašnjeni SMT problemi, kao koncept uz pomoć kog se automatizuje odabir redukovanog skupa testova, kao i format *SMT-LIB* koji se koristi za opisivanje ovih problema. Takođe, biće predstavljeni SMT rešavač *Z3*[1], okruženje za testiranje *Google* testovi, alat *Gcov*, kojim se dobijaju informacije o pokrivenosti i okruženje za razvoj višeplatformskog softvera *Qt*, koje je korišćeno za izradu grafičkog dela alata. Treba naglasiti da SMT rešavači nisu jedini alat koji se može koristiti za rešavanje ovakvih problema. Mogli bi se koristiti alati za *celobrojno linearno programiranje* (eng. *integer linear programming - ILP*), koji mogu biti prikladniji za rešavanje ovakvih problema. O njima se može pročitati u [27].

4.1 SMT rešavači

SMT problem (eng. *Satisfiability Modulo Theory*) je problem ispitivanja zadovoljivosti formula logike prvog reda. Za formulu se kaže da je zadovoljiva u nekoj teoriji, ako je tačna u barem jednom modelu te teorije. Najčešće SMT teorije su:

- *Teorija jednakosti sa neinterpretiranim funkcijskim simbolima* (eng. *equality with uninterpreted functions* – EUF) – od predikatskih simbola sadrži samo jednakost, a svi funkcijski simboli i sorte[2] se mogu slobodno interpretirati.

[1]SMT rešavač *Z3* se u ovoj tezi koristi kao rešavač ograničenja.

[2]Sorta može biti, na primer, *Bool*, *Real* ili *Array*.

Neka su a i b sorte $Real$ i neka je f funkcijski simbol arnosti 1 (tj. ima jedan argument). Primer formule u teoriji EUF je:

$$(a = b) \wedge (f(a) = f(b))$$

- *Realna aritmetika* – sastoji se od sorte $Real$ i simbola 0, 1, $+$, $\cdot$, $-$, $/$ i $\leqslant$. Neka su a i b sorte $Real$. Primer formule u realnoj aritmetici je:

$$\neg(a \leqslant 0) \wedge (3 \cdot b \leqslant 1)$$

- Celobrojna aritmetika – sastoji se od sorte Int i simbola 0, 1, $+$, $\cdot$, $-$ i $\leqslant$. Primer formule u celobrojnoj aritmetici, gde je a sorte Int:

$$(10 \cdot a \leqslant 1) \wedge (a \leqslant 5 \cdot a)$$

- *Teorija nizova* – sadrži tri sorte, $Index$, $Value$ i $Array$. Sadrži i dva funkcijska simbola, *select* i *store* za pristup elementima i postavljanje vrednosti elementa. Neka je a sorte $Array$, i i j sorte $Index$ i x sorte $Value$. Primer formule u teoriji nizova je:

$$select(a, i) = select(store(a, j, x), j)$$

- *Teorija bitvektora* (eng. *bit-vector*) – koristi sorte $BitVec_n$, gde je $n \in \mathbb{N}$, koje se interpretiraju skupovima svih vektora bitova fiksirane dužine n. Neki od funkcijski simbola koje prepoznaje su $bvnot_n$, $bvneg_n$, $bvadd_n$, $bvsub_n$, $bvmul_n$, $bvudiv_n$, $bvsdiv_n$, $bvurem_n$, $bvsrem_n$, $bvshl_n$, $bvult_n$ i $bvslt_n$. Interpretiraju se kao odgovarajuće operacije nad bitvektorima, tj. binarnim brojevima[3]. Ova teorija omogućava apstrakciju stvarne hardverske aritmetike. Naredna formula, gde su a i b konstante sorte $BitVec_8$, predstavlja primer formule iz teorije bitvektora:

$$bvult_8(bvshr_8(a, 2), bvneg_8(b)) \wedge bvult_8(a, b)$$

Za neke od ovih teorija SMT problem je *odlučiv*[4] samo za formule određenog oblika, a za druge je odlučiv za proizvoljnu formulu te teorije. Teorija je odlučiva ako postoji efektivan postupak koji u konačnom broju koraka za svaku formulu

[3]Na primer, $bvnot_n$ predstavlja bitovsku negaciju, $bvmul_n$ je množenje po modulu 2^n, $bvshl_n$ je pomeranje bitova ulevo, $bvult_n$ je operacija „manje od” za neoznačene brojeve.

[4]Kod *problema odlučivosti* (eng. *decision problem*) potrebno je dati „da/ne” odgovor za dati ulaz $x \in \{0, 1\}^*$, odnosno potrebno je odgovoriti na pitanje da li dati ulaz zadovoljava određeno svojstvo [26]. Primer odlučivog problema je 3-obojivost grafa.

ispituje da li je ona dokaziva u toj teoriji, bez dodatnih pretpostavki. Softverski alati koji implementiraju procedure odlučivanja zovu se *SMT rešavači* (eng. *solvers*). Detaljnije o SMT problemu može se videti u [13].

Jezikom SMT rešavača može se smatrati jezik *SMT-LIB*, koji se koristi za opisivanje SMT problema. *SMT-LIB* se prvi put pojavio 2003. godine. Jedna važna upotreba SMT rešavača je prilikom rešavanja problema CSP u verifikaciji softvera, što je podstaklo reviziju jezika i 2010. godine dovelo do verzije 2 jezika *SMT-LIB* [10], skraćeno *SMT2*, koji se u ovoj tezi koristi za čuvanje ograničenja koja moraju biti ispunjena. *SMT-LIB* opisuje logičke probleme u *višesortnoj logici prvog reda* (eng. *many-sorted first-order logic*). Detaljno o jeziku *SMT-LIB* se može pročitati u [10], a u nastavku su dati neki osnovni pojmovi.

Jezik *SMT-LIB* sve ulaze i izlaze rešavača (koji podržava ovaj format) posmatra kao niz *S-izraza* (eng. *S-expression*). S-izraz može biti:

- *token*
- niz nula ili više S-izraza uokvirenih zagradama

Tokeni mogu biti niske, brojevi i rezervisane reči kao što su, na primer, komanda `assert`, za zadavanje ograničenja ili `check-sat` za proveravanje zadovoljivosti datih uslova. Takođe, moguće je uvođenje novih funkcijskih simbola pomoću `declare-fun`. Komandom `define-sort` se može zadati skraćeni zapis neke sorte. Omogućeno je i pisanje komentara kojima uvek mora prethoditi dvotačka (`;`).

Primer jedne datoteke napisane u formatu *SMT-LIB* dat je na slici 4.1. U prvoj liniji se inicijalizuje rešavač tako da koristi linearni fragmet logike celobrojne aritmetike (`QF_LIA`) pomoću komande `set-logic`. U liniji 3 se komandom `define-sort` zadaje skraćeni zapis `I` za sortu `Int`, koji se koristi u nastavku. Naredne dve linije uvode promenljive `x` i `y` celobrojnog tipa kao funkcije bez argumenata. U linijama 6 i 7 navode se ograničenja koja treba da budu zadovoljena. To se čini komandom `assert`. U liniji 9 zadata je komanda `check-sat` kojom se rešavač upućuje na proveru zadovoljivosti zadatih pretpostavki. U poslednjoj liniji se navodi komanda `get-model`, koja se može koristiti samo u slučaju da su zadati uslovi zadovoljivi i služi za ispisivanje modela. S obzirom da su u ovom primeru zadata ograničenja zadovoljiva, `get-model` treba da ispiše vrednosti dodeljene promenljivama `x` i `y` tako da zadati uslovi budu ispunjeni[5]. Pojednostavljeno posmatrano, strukturu ove

[5]Na primer, `x` može imati vrednost 2, a `y` može imati vrednost 5.

datoteke čine komanda za definisanje logike koja se koristi, deo u kome se zadaju pretpostavke i provera zadovoljivosti tih pretpostavki.

```
1  (set-logic QF_LIA)        ; Fragment teorije celobrojne aritmetike
2
3  (define-sort I () Int)    ; Skraceni zapis za Int
4  (declare-fun x () I)      ; Deklaracija promenljive x
5  (declare-fun y () I)      ; Deklaracija promenljive y
6  (assert (> (+ x y) 6))          ; Zadavanje uslova x + y > 6
7  (assert (= (+ x (* 2 y)) 12))   ; Zadavanje uslova x + 2*y = 12
8
9  (check-sat)                     ; Provera zadovoljivosti uslova
10 (get-model)                     ; Ispis modela
```

Slika 4.1: Primer jednostavne datoteke u formatu *SMT-LIB*.

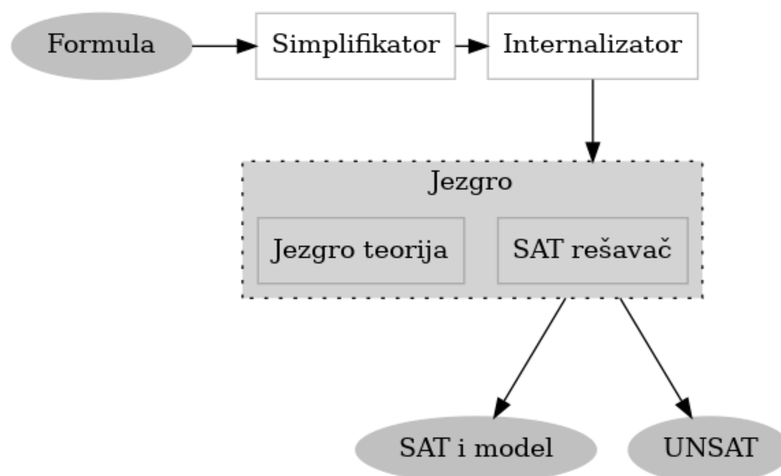
SMT rešavač *Z3*

Z3 je SMT rešavač koji rešava problem odlučivosti za logičke formule koristeći kombinacije teorija, kao što su teorija bitvektora, nizova, EUF i aritmetike. *Z3* koristi termove višesortne logike prvog reda, u kojoj za svaku sortu termova postoji poseban domen iz kojeg se uzimaju vrednosti, za razliku od logike prvog reda, koja podrazumeva jedinstven domen.

Z3 se sastoji iz više faza, prikazanih na slici 4.2, u kojima se vrši ispitivanje zadovoljivosti date formule. Prvo se vrši pojednostavljivanje, tj. simplifikacija, zatim se dobijena formula prevodi u interno korišćeni format s kojim jezgro rešavača može da radi, što se naziva internalizacija. Jezgro rešavača sadrži SAT rešavač[6] i rešavače teorija, od kojih autori Leonardo de Mura i Nikolaj Bjorner[7] ističu važnost teorije jednakosti sa neinterpretiranim funkcijskim simbolima, jer predstavlja sponu između SAT rešavača i drugih teorija. Kao rezultat dobija se poruka o zadovoljivosti ili nezadovoljivosti date formule. U prvom slučaju se uz poruku predstavlja i model koji zadovoljava formulu. Više o *Z3* rešavaču može se videti u [11].

[6]SAT je problem ispitivanja zadovoljivosti formula iskazne logike.

[7]Leonardo de Mura (*Leonardo de Moura*) i Nikolaj Bjorner (*Nikolaj Bjørner*), *Microsoft Research*, autori *Z3* rešavača.



Slika 4.2: Faze kroz koje prolazi *Z3* rešavač tokom ispitivanja zadovoljivosti date formule.

4.2 Google testovi

Google testovi predstavljaju okruženje za testiranje i *oponašanje* (eng. *mocking*) koda u jeziku C++ razvijeno u kompaniji *Google*. Testovi u ovom okruženju se u osnovi sastoje iz tvrđenja, koja mogu rezultovati *uspehom* (eng. *success*), *nefatalnim neuspehom* (eng. *nonfatal failure*) i *fatalnim neuspehom* (eng. *fatal failure*). Razliku između poslednja dva stanja je što fatalni neuspeh zaustavlja dalje izvršavanje funkcije, a nefatalni ne. Testovi se grupišu u *skupove testova* (eng. *test suites*) i savetuje se grupisanje testova po ugledu na strukturu koda koji se testira. Jedan *program za pokretanje testova* (eng. *test program* ili *test executable*) može sadržati više skupova testova. Više o pisanju *Google* testova može se videti u [3].

4.3 Alat *Gcov*

Gcov je jedan od alata koji se isporučuju kao deo kompilatora *GCC* (eng. *GNU Compiler Collection*) i služi za merenje pokrivenosti koda prilikom izvršavanja programa. Koristi se zajedno sa kompilatorom *GCC* da bi se vršila analiza programa sa ciljem otkrivanja nepokrivenih delova programa. Alat *Gcov* se može koristiti kao alat za profajliranje, tj. može ukazati na delove koda koji se često izvršavaju i čija bi optimizacija mogla najviše doprineti efikasnijem izvršavanju programa [17].

Za softver pisan u jeziku C++, neophodno je koristiti dodatne opcije i isključiti optimizacije prilikom kompiliranja, kako bi se omogućilo praćenje izvršavanja linija, grana i funkcija u kodu. Primer pokretanja kompilatora `g++` sa uključenim opcijama za pokrivenost:

```
g++ -g -Wall -fprofile-arcs -ftest-coverage -O0 main.cpp -o main
```

Opcije `-fprofile-arcs` i `-ftest-coverage` mogu se zameniti opcijom `-coverage`. Nakon kompilacije, za svaku datoteku koja sadrži kôd generiše se po jedna datoteka sa ekstenzijom `.gcno`, a nakon izvršavanja programa nastaje datoteka sa ekstenzijom `.gcda`. Zajedno ove dve datoteke pružaju informacije o pokrivenosti, ali se za čitljiviji pregled savetuje korišćenje međureprezentacije, koja se dobija pokretanjem alata *Gcov* sa opcijom `-json-format` [24].

4.4 Razvojno okruženje *Qt*

Qt je okruženje za razvoj *višeplatformskog* (eng. *cross-platform*) softvera. Podržava platforme *Linux*, *Android*, *OS X*, *iOS*, *Windows* i druge. Qt ne predstavlja programski jezik, već okruženje koje je napisano u jeziku C++. Korišćenjem posebnog pretprocesora koji se naziva *MOC* (eng. *Meta-Object Compiler*) i koji se pokreće pre same kompilacije, postiže se proširivanje jezika C++ mehanizmima kao što su signali i slotovi. Kôd dobijen nakon preprocesiranja u skladu je sa standardom jezika C++ [8]. Pomoću okruženja Qt se može praviti grafički korisnički interfejs, korišćenjem modula *Widgets*, koji pruža osnovne elemente potrebne za kreiranje korisničkog okruženja, kao što su *QLineEdit* za učitavanje ulaza zadatog od strane korisnika, *QGroupBox* za grupisanje elemenata ili *QTableWidget* za prikazivanje tabela.

Glava 5

Opis implementacije i evaluacija

U ovoj glavi biće predstavljena implementacija alata *TestSuiteReduction* za automatsko uklanjanje redundantnih testova iz datog skupa testova. Implementacija je vršena na osnovu postupka definisanog u potpoglavlju 3.3 glave 3. Skup testova se zadaje zavisno od tipa testova koji se koriste, npr. *Google* testovi. Nakon uspešno parsiranog spiska testova, vrši se izračunavanje pokrivenosti koda koje se čuva u matrici pokrivenosti. Na osnovu matrice pokrivenosti generišu se klauze koja moraju važiti da bi se pokrivenost koda zadržala. Do rešenja se dolazi pomoću SMT rešavača *Z3*, korišćenjem linearnog fragmenta teorije celobrojne aritmetike. Izvorni kôd aplikacije se može naći na *git* repozitorijumu [4]. Pre opisa samog alata, jednostavnim primerom približena je procedura automatskog uklanjanja redundantnih testova koju implementira alat *TestSuiteReduction*.

5.1 Arhitektura sistema

Razvijeni alat *TestSuiteReduction* se sastoji iz dve celine – biblioteke *TestSuiteReductionAPI* napisane u jeziku *C++* i grafičkog interfejsa napravljenog korišćenjem okruženja *Qt*. Grafički interfejs napravljen je sa ciljem preglednosti procesa i odabranih parametara, kao i zarad veće udobnosti tokom korišćenja.

Biblioteka *TestSuiteReductionAPI* pruže naredne funkcionalnosti:

- rad sa skupom testova – pokretanje pojedinačnih testova i izlistavanje svih dostupnih testova u datom projektu
- rad sa podacima o pokrivenosti koda – prikupljanje podataka o pokrivenosti (pokretanjem alata za analizu pokrivenosti) i njihovo čuvanje u matrici

pokrivenosti u formatu *CSV*

- rad sa rešavačem ograničenja – generisanje ograničenja na osnovu podataka iz matrice pokrivenosti, čuvanje ograničenja u odgovarajućem formatu i pokretanje rešavača sa ciljem pronalaženja rešenja za zadata ograničenja

Biblioteka *TestSuiteReductionAPI* se može posmatrati kroz tri glavne klase pomoću kojih se ostvaruju navedene funkcionalnosti – *TestingFramework*, *CoverageTool* i *Solver*, koje se dalje nasleđuju kako bi se definisalo ponašanje aplikacije za konkretno odabranu biblioteku za testove, alat koji se koristi za izračunavanje pokrivenosti i rešavač ograničenja. Klasa *Report* služi za povezivanje informacija koje pruža klasa *TestingFramework* sa jedne strane i klasa *CoverageTool* sa druge strane. Uz pomoć nje se generišu datoteke u *JSON* formatu koje predstavljaju izveštaje o pokrivenosti koda za pojedinačan test. Pregled svih klasa i njihovih najvažnijih članica i metoda može se videti na dijagramu 5.1.

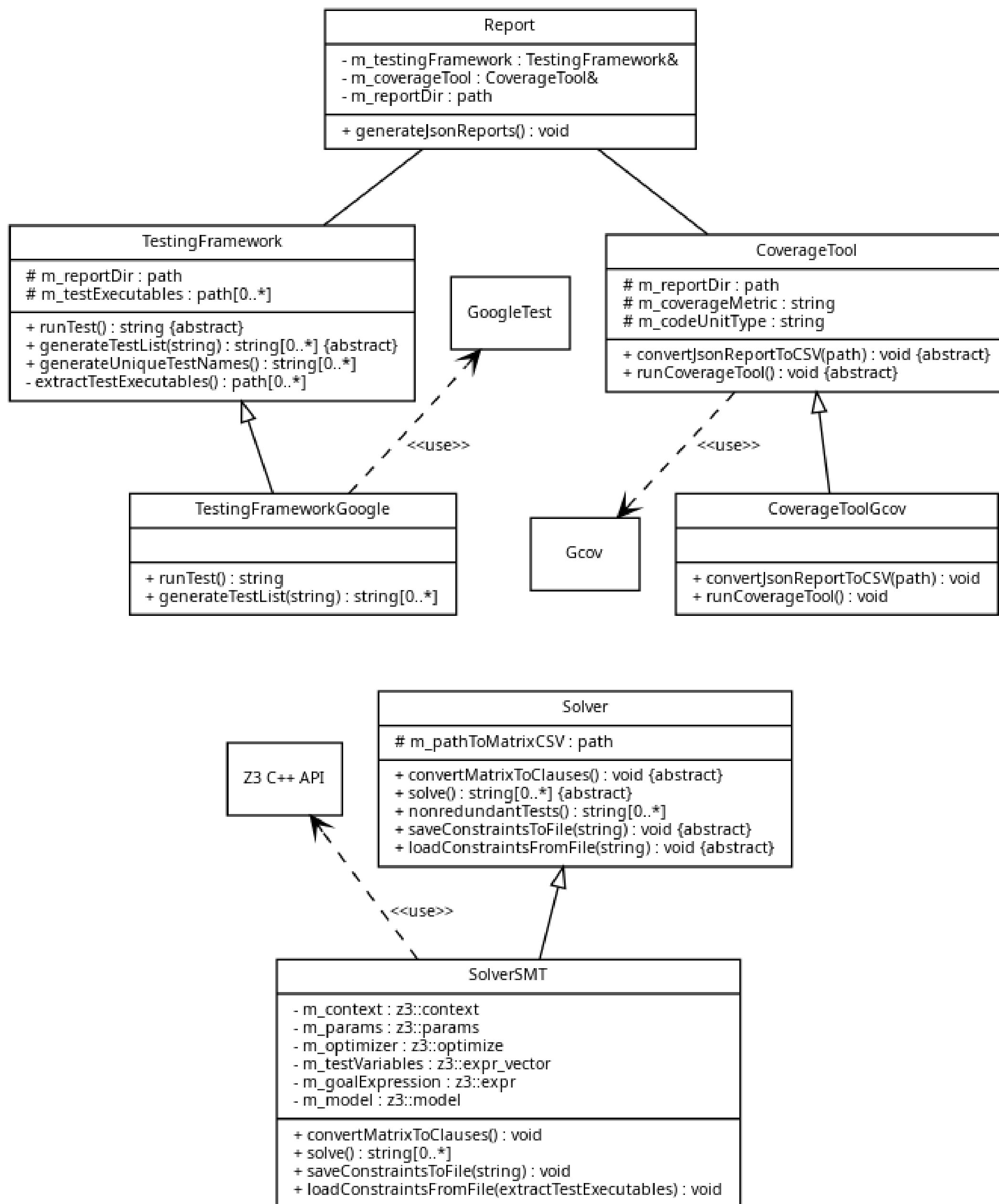
Klasa *TestingFramework*

Klasa *TestingFramework* definiše dve metode koje ne zavise od odabrane biblioteke za testove. U pitanju su metoda za generisanje i dohvatanje naziva svih testova (*generateUniqueTestNames()*) i metoda kojom se dohvataju sve izvršne datoteke pomoću kojih se testovi izvršavaju (*extractTestExecutables()*). U njoj su zadate i virtuelne metode koje moraju biti opisane u klasama koje je nasleđuju.

Klasa *TestingFrameworkGoogle* nasleđuje klasu *TestingFramework* i definiše virtuelne metode *runTest()* i *generateTestList()*. Metoda *runTest()* pokreće test koji joj se prosleđuje i njena implementacija zavisi od odabrane biblioteke za testove. U slučaju *Google* testova, pozivanje pojedinačnih testova se vrši pomoću naredne komande, koja se može izvršiti iz komandne linije:

```
./<testExecutable> --gtest_filter=<testSuite>.<testName>
```

Ova komanda se koristi u metodi *runTest()* za pokretanje testova. Metoda *generateTestList()* izlistava sve testove koji se mogu pokrenuti iz date izvršne datoteke i takođe zavisi od biblioteke za testove koja se koristi. Izvršne datoteke za *Google* testove prihvataju argument `gtest_list_tests`, pomoću kog se dobija spisak svih testova.



Slika 5.1: Dijagram klasa *TestSuiteReductionAPI* biblioteke.

Klasa *CoverageTool*

CoverageToolGcov nasleđuje klasu *CoverageTool* i definiše njene virtuelne metode u skladu sa alatom za analizu pokrivenosti koda *Gcov*. U ovoj klasi se definiše

koji će se tip pokrivenosti koristiti, kao i granularnost u kojoj će se posmatrati izvorni kôd, odnosno koja jedinica koda će se posmatrati, funkcije ili izvorne datoteke. Definisana je i metoda *runCoverageTool()* koja pokreće *Gcov* za sve testove iz date izvršne datoteke. Tačnije, ova metoda pokreće alat *fastcov*[1] koji iskorišćava mogućnost alata *Gcov* za generisanje međureprezentacije koja se može obrađivati paralelno i time brže doći do rezultata. Još jedna metoda koju treba izdvojiti je *convertJsonReportToCSV()*, pomoću koje se parsira izlaz alata *Gcov*, izdvajaju se samo relevantne informacije i čuvaju u formatu *CSV*[2]. Pokretanjem ove metode za svaki test, odnosno za svaki izveštaj o pokrivenosti, i spajanjem tako dobijenih datoteka u jednu, dolazi se do matrice pokrivenosti, koja se čuva u formatu *CSV*.

Klasa *Solver*

Klasa *Solver* daje potpise virtuelnih metoda čije ponašanje moraju definisati sve klase koje je nasleđuju. Klasa *SolverSMT* nasleđuje klasu *Solver* i definiše ponašanje dve ključne virtuelne metode – *solve()* i *convertMatrixToClauses*. Metoda *convertMatrixToClauses* prolazi kroz matricu pokrivenosti, koju učitava iz date *CSV* datoteke, pronalazi grupe istih pokrivenosti u jednoj koloni (odnosno za jednu jedinicu koda) i za svaku kolonu generiše *klauzu* (eng. *clause*) kojom se zahteva da se zadrži barem jedan test iz svake grupe testova koji istim procentom pokrivaju trenutnu jedinicu koda. Ovo je u skladu sa postupkom opisanim u 3.3. Metoda *solve()* pokreće rešavač *Z3* nad generisanim klauzama (ili ograničenjima) korišćenjem C++ API poziva, čime se dolazi do redukovanog skupa testova. U klasi *SolverSMT* definisane su i metode za čuvanje klauza u datoteci i učitavanje iz datoteke u formatu *SMT-LIB*.

Postupak koji implementira metoda *convertMatrixToClauses* može se detaljnije opisati kroz jednostavan primer. Neka skup testova sadrži tri testa – t_1, t_2 i t_3. Neka se program koji se posmatra sastoji iz tri izvorne datoteke, označene sa u_1, u_2 i u_3 i neka su izvorne datoteke jedinice koda. Matrica pokrivenosti $\mathcal{M}$ koja sadrži

[1] Originalni kôd alata *fastcov* se može pronaći u [15].

[2] Format datoteke u kome su polja jednog *sloga* (eng. *record*) razdvojeni zarezima (eng. *comma-separated values*)

informacije o pokrivenosti linija koju dostiže svaki od testova ima sledeći oblik:

$$\begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \end{array} \begin{array}{c} \begin{array}{ccc} u_1 & u_2 & u_3 \end{array} \\ \begin{pmatrix} 90 & 10 & 10 \\ 20 & 0 & 10 \\ 20 & 0 & 90 \end{pmatrix} \end{array}$$

Uslovi koji moraju biti ispunjeni da bi se redundantni testovi uklonili, zadaju se u datoteci u formatu *SMT-LIB*, koja se prosleđuje SMT rešavaču *Z3*. Pre svega, neophodno je uvesti promenljive koje predstavljaju testove. To se postiže korišćenjem komande `declare-fun`. Za svaki test dodaje se linija oblika `(declare-fun t1 () Int)`. Takođe, da bi svakom testu bila dodeljena nula ili jedinica, pri čemu nula predstavlja redundantan test, a jedinica test koji se zadržava u redukovanom skupu, zadaju se naredna ograničenja za svaki test:

```
(assert (>= t1 0))
(assert (<= t1 1))
```

Zatim, u datoteku se dodaju uslovi izvedeni iz matrice pokrivenosti, po jedan za svaku kolonu. Ako posmatramo kolonu koja odgovara jedinici koda u_1, može se primetiti da postoje dva različita procenta pokrivenosti – 90 i 20, što na osnovu koraka 3 iz postupka za redukovanje skupa testova datog u delu 3.3 znači da treba formirati dve grupe testova i zadržati barem po jedan test iz obe grupe. Dakle, metoda *convertMatrixToClauses* bi na osnovu kolone u_1 formirala ograničenja kojima se zahteva da se u redukovanom skupu zadrži barem jedan test koji daje procenat pokrivenosti 90 (samo test t_1 daje tu pokrivenost) i barem jedan koji daje procenat pokrivenosti 20 (testovi t_2 i t_3). Dakle, za svaku jedinicu koda posmatranu izolovano, svaki različiti procenat pokrivenosti mora imati svog „predstavnika"[3] u rezultujućem redukovanom skupu testova. U formatu *SMT-LIB* takav uslov ima sledeći oblik:

```
(assert (and (> t1 0)
             (or (> t2 0)
                 (> t3 0))))
```

Na sličan način se definišu ograničenja za preostale dve jedinice koda.

Rešavaču se zadaje cilj u vidu minimizacije zbira promenljivih koje predstavljaju testove, što se može postići komandom `(minimize (+ t1 t2 t3))`. *Z3* pruža komandu `minimize` kao deo proširenja jezika *SMT-LIB* za zadavanje optimizacionih ci-

[3]Može biti i više od jednog „predstavnika" nekog procenta pokrivenosti.

ljeva[4]. Minimizovanje ovog zbira dovodi do dodele nula i jedinica testovima tako da redukovani skup bude minimalan, pritom zadovoljavajući ograničenjima opisane kriterijume. Celokupna datoteka u formatu *SMT-LIB* ima oblik:

```
(declare-fun t1 () Int)
(declare-fun t2 () Int)
(declare-fun t3 () Int)
(assert (>= t1 0)) (assert (<= t1 1))
(assert (>= t2 0)) (assert (<= t2 1))
(assert (>= t3 0)) (assert (<= t3 1))

(assert (and (> t1 0) (or (> t2 0) (> t3 0))))
(assert (> t1 0))
(assert (and (or (> t1 0) (> t2 0)) (> t3 0)))

(minimize (+ t1 t2 t3))
(check-sat)
(get-model)
```

Komandom `get-model` ispisuju se vrednosti dodeljene promenljivama takve da su zadata ograničenja ispunjena:

```
(model
  (define-fun t3 () Int
    1)
  (define-fun t2 () Int
    0)
  (define-fun t1 () Int
    1)
)
```

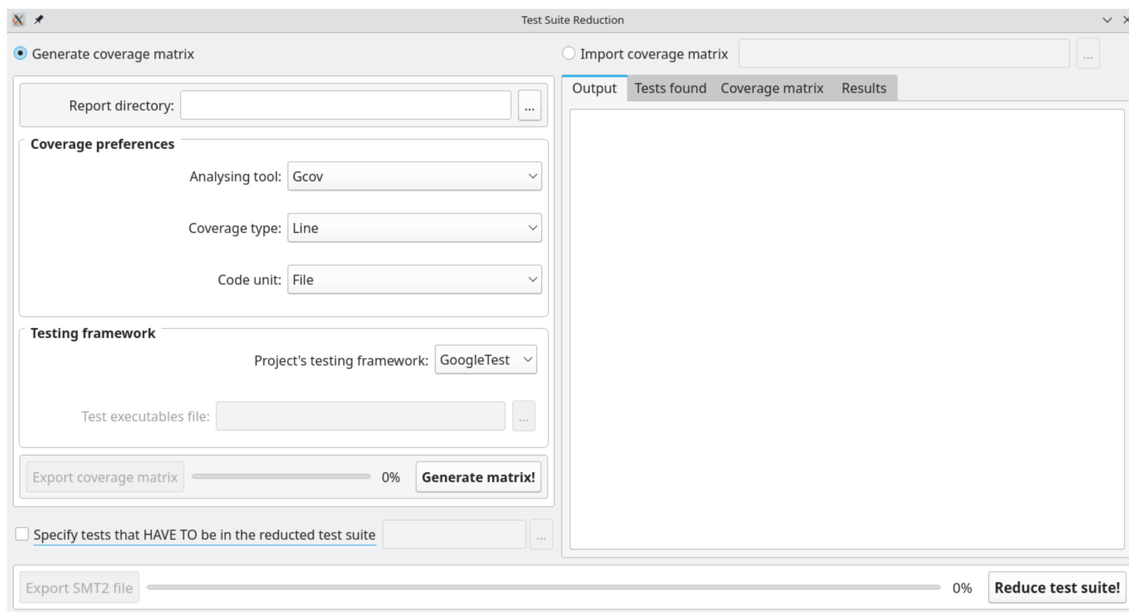
Dakle, rezultat koji se dobija pokretanjem alata *Z3* nad ovom datotekom je redukovan skup testova koji sadrži testove t_1 i t_3, a test t_2 kome je dodeljena vrednost nula, predstavlja redundantan test.

5.2 Grafički korisnički interfejs

Grafički korisnički interfejs se sastoji iz jednog prozora u kome se vrši učitavanje podataka neophodnih za rad alata, kao i dodatna podešavanja. Sastoji se iz dva vizuelno različita dela, koji se mogu videti na slici 5.2. Sa leve strane prozora se

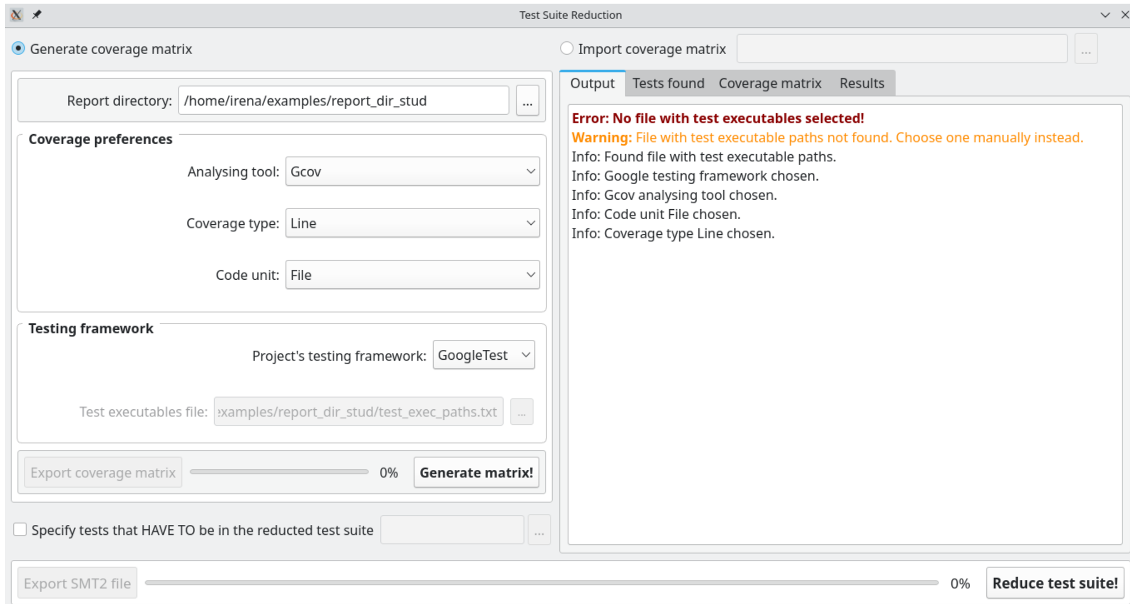
[4]Postoji i komanda `maximize`, kojom se rešavaču zadaje izraz čiju je vrednost potrebno maksimizovati.

nalaze različiti parametri koje je neophodno zadati ili odabrati pre nego što se pozovu bibliotečke funkcije. Sa desne strane prikazuje se trenutno stanje i rezultati. Te informacije su grupisane u četiri grupe, tj. kartice – *Output*, *Tests found*, *Coverage matrix* i *Results*.



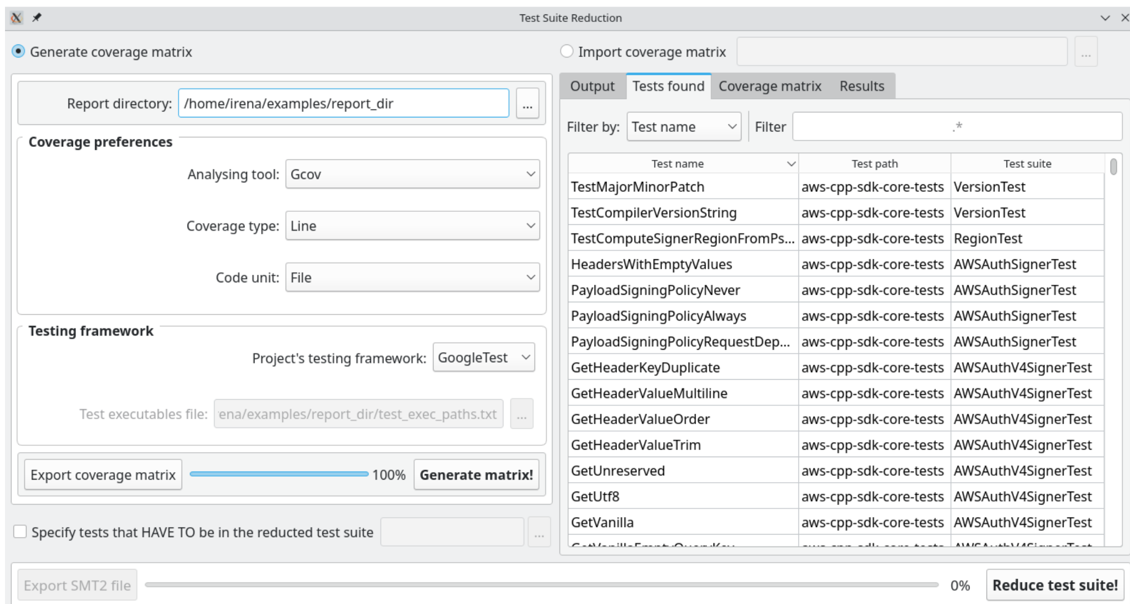
Slika 5.2: Izgled grafičkog interfejsa prilikom prvog pokretanja.

Kartica *Output* prikazuje informacije o promenama koja nastaju tokom upotrebe alata, kao što su učitavanje spiska izvršnih datoteka za pokretanje testova, odabir vrste pokrivenosti, učitavanje matrice pokrivenosti iz datoteke u formatu *CSV*. Pored ovih poruka, ovde se mogu videti i upozorenja i greške do kojih eventualno dolazi tokom korišćenja, jasno naglašenih narandžastom i crvenom bojom. Na slici 5.3 prikazano je kako izgleda ispis nekih poruka dobijenih tokom korišćenja alata. Različito se prikazuju poruke koje predstavljaju greške, upozorenja i informacije o izvršavanju.

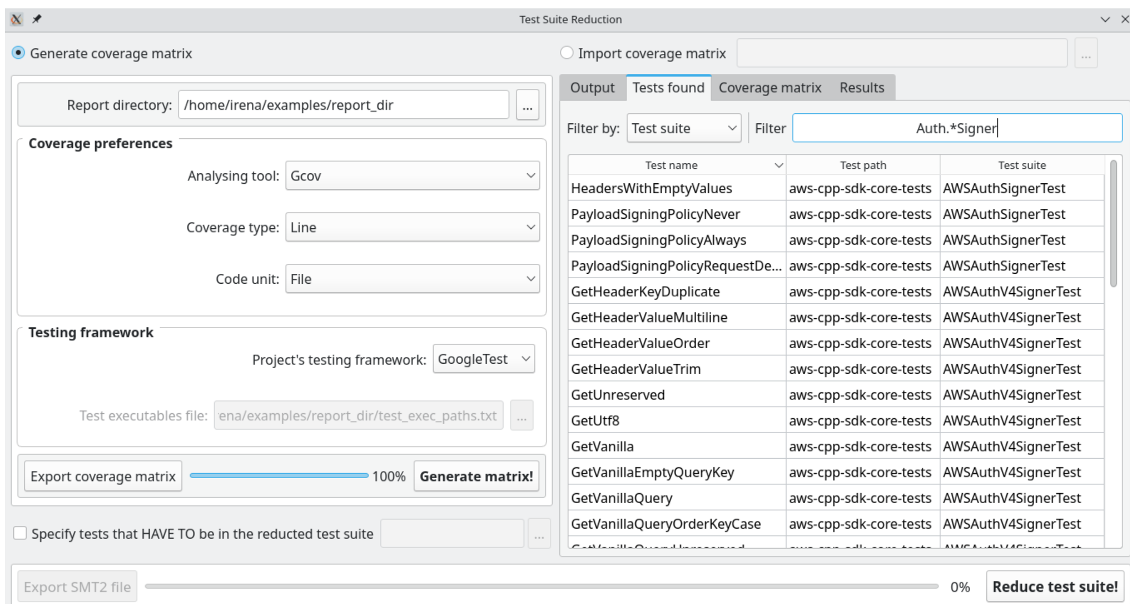


Slika 5.3: Izgled grafičkog interfejsa sa prikazom poruka dobijenih tokom upotrebe.

Kartica *Tests found* prikazuje spisak prepoznatih testova do kojih alat dolazi parsirajući datoteku zadatu pod *Test executables file*, koja treba da sadrži spisak putanja do izvršnih datoteka pomoću kojih se pokreću testovi, nakon čega se korišćenjem metode *generateTestList()* klase *TestingFrameworkGoogle* pronalaze svi testovi koji mogu biti pokrenuti uz pomoć datih izvršnih datoteka. Izgled ove kartice može se videti na slici 5.4. Omogućeno je i filtriranje liste pomoću regularnih izraza. Regularni izraz se može odnositi na naziv testa, naziv skupa testova, putanju do izvršne datoteke ili na bilo koji deo liste. Na slici 5.5 se vidi primer liste testova koji pripadaju skupu testova čiji naziv sadrži reči *Auth* i *Signer*.



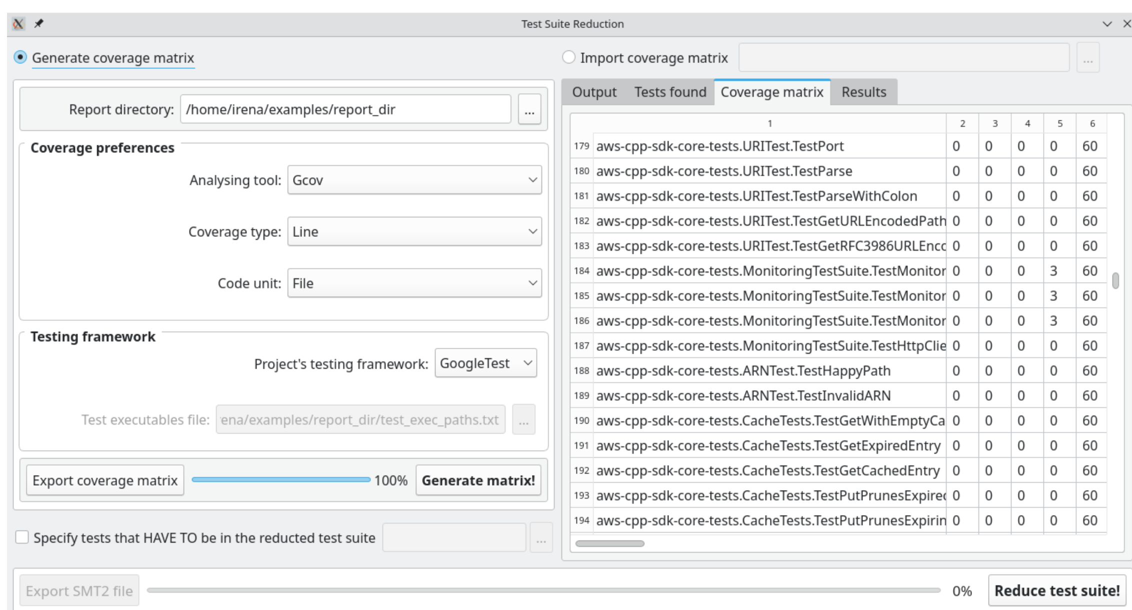
Slika 5.4: Izgled dela grafičkog interfejsa koji služi za pregled svih testova.



Slika 5.5: Primer filtriranja liste testova na osnovu zadatog regularnog izraza.

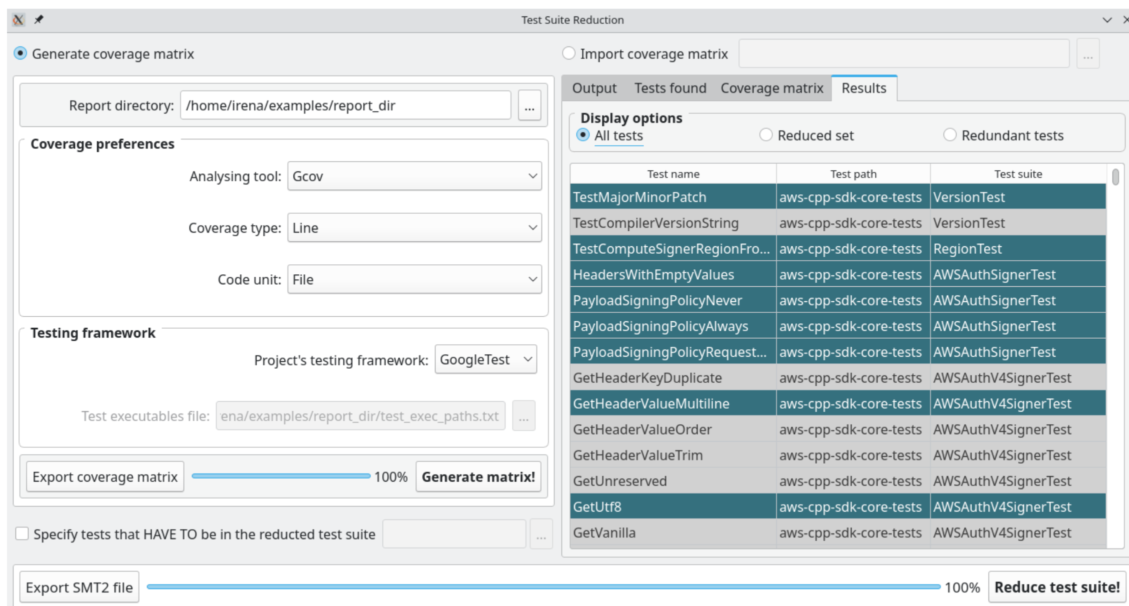
U kartici *Coverage matrix* se može videti matrica pokrivenosti nakon što se ista generiše. Takođe, ukoliko se ne vrši generisanje, već se matrica pokrivenosti učitava iz

datoteke zadate u formatu *CSV*, koja se može zadati nakon odabira *Import coverage matrix* opcije na vrhu prozora, učitani podaci o pokrivenosti će se prikazati u ovoj kartici u vidu tabele, kao što se može videti na slici 5.6. Prikaz matrice pokrivenosti omogućava korisniku analizu podataka o pokrivenosti. Kao nazivi kolona mogli bi se dodati nazivi jedinica koda i u tom slučaju bi korisnik imao više informacija o svakom testu i mogao bi zaključiti koje jedinice koda čine neki test redundantnim u odnosu na ostale testove. Takvom analizom bi se moglo uočiti da li neke testove treba, ipak, zadržati u redukovanom skupu na osnovu delova koda koje proveravaju.



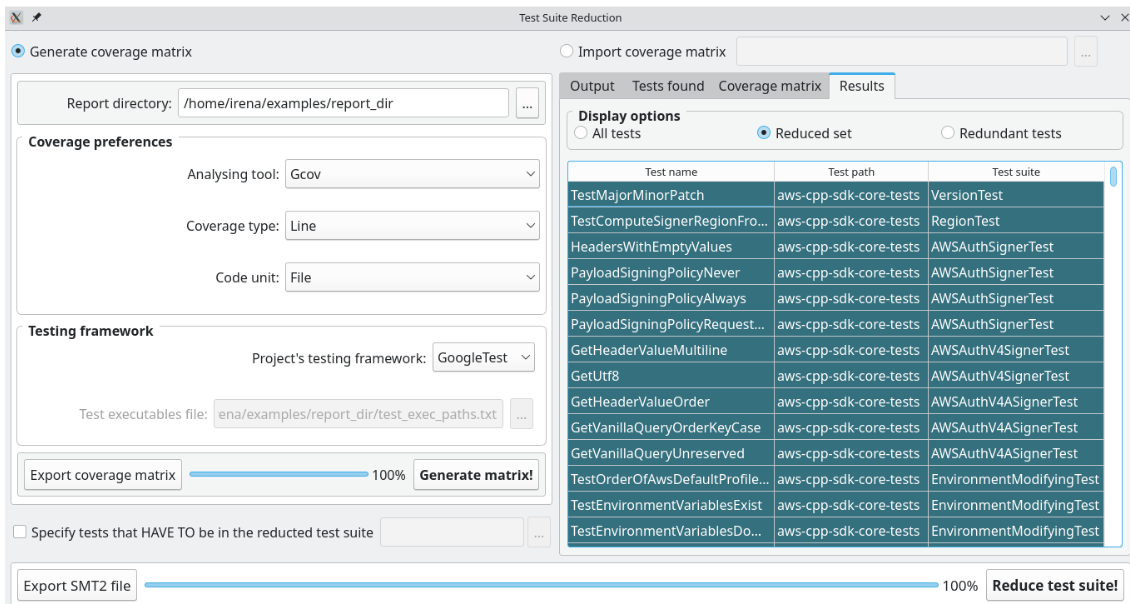
Slika 5.6: Izgled dela grafičkog interfejsa koji služi za pregled matrice pokrivenosti.

Results kartica služi za pregled rezultata u vidu spiska testova koji su proglašeni redundantnim. Omogućen je odabir skupa testova koji će se prikazati – svi testovi, redukovan skup testova ili samo skup testova koji su proglašeni redundantnim. Takođe, vizuelno se drugačije navode testovi iz redukovanog skupa i oni koji su proglašeni redundantnim, kao što se vidi na slici 5.7.

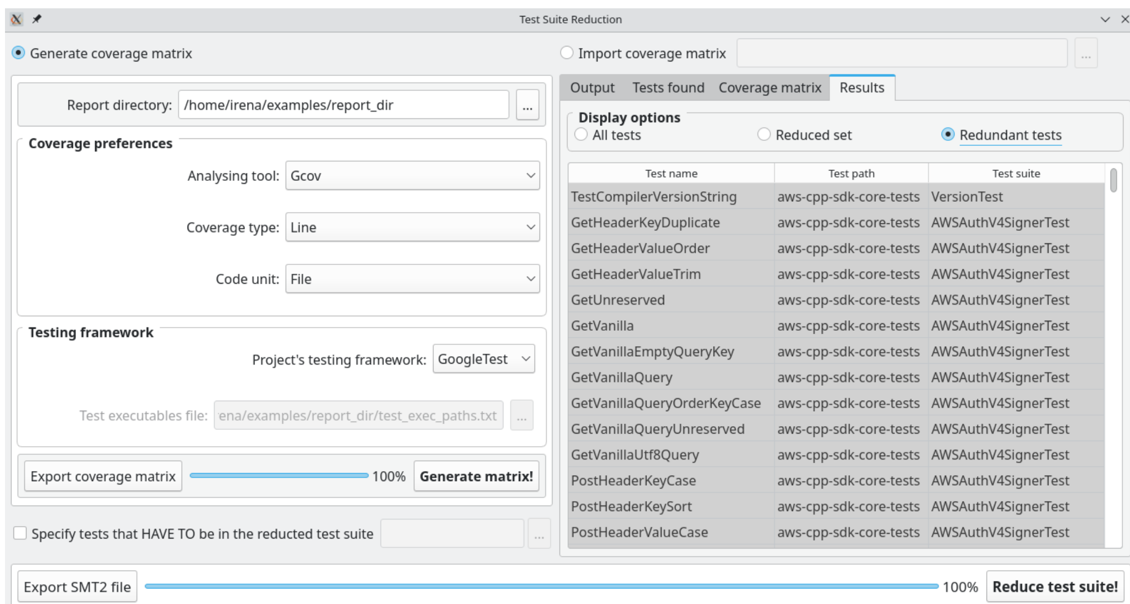


Slika 5.7: Izgled dela grafičkog interfejsa koji služi za pregled rezultata, odnosno prikaz spiska testova sa naznačenim redundantim testovima.

Moguć je prikaz samo testova iz redukovanog skupa, što se vidi na slici 5.8. Može se izdvojiti i samo spisak redundantnih testova odabirom odgovarajuće opcije u kartici *Results*, što se vidi na slici 5.9.



Slika 5.8: Prikaz rezultata rada alata u vidu redukovanog skupa testova.



Slika 5.9: Prikaz rezultata rada alata u vidu spiska testova koji su proglašeni redundantnim.

5.3 Instalacija i pokretanje

Kompilacija biblioteke se vrši pomoću alata *cmake* i podrazumeva dostupnu verziju kompilatora *gcc* koja podržava standard 20. Takođe, neophodno je imati dostupnu biblioteku za rad sa rešavačem *Z3*, koja se može instalirati pomoću naredne komande na operativnim sistemima *Linux* zasnovanim na distribuciji *Debian*:

```
sudo apt install libz3-dev
```

Potrebno je instalirati i biblioteku *TBB*, koja omogućava paralelizaciju koda napisanog u jeziku C++. Može se instalirati pomoću naredne komande:

```
sudo apt install libtbb-dev
```

Da bi se formirala biblioteka, u datoteci *CMakeLists.txt* potrebno je imati linije prikazane na slici 5.10. Ovime će se u zadatom *build* direktorijumu generisati deljena biblioteka *libTestSuiteReductionAPI.so*, kao i propratne datoteke uz pomoć kojih se instalacija dovršava tako što se pokrene naredna komanda:

```
sudo make install
```

Nakon pokretanja ove komande, biblioteka će biti instalirana na sistemu i dostupna za korišćenje od strane drugog softvera.

```
1  add_library(TestSuiteReductionAPI SHARED
2    testingframework.cpp testingframework.h
3    coveragetool.cpp coveragetool.h
4    report.cpp report.h
5    solver.cpp solver.h
6    utils.cpp utils.h)
7
8  install(TARGETS TestSuiteReductionAPI DESTINATION /usr/lib)
9  install(FILES testingframework.h coveragetool.h report.h solver.h
10         DESTINATION /usr/local/include)
```

Slika 5.10: Deo datoteke *CMakeLists.txt* kojim se definiše da rezultat kompilacije treba da bude deljena biblioteka.

Grafički interfejs može upotrebiti funkcije biblioteke korišćenjem zaglavlja koja biblioteka pruža:

```
1 #include <testingframework.h>
2 #include <coveragetool.h>
3 #include <solver.h>
4 #include <report.h>
```

Kompilacijom grafičkog korisničkog interfejsa generiše se izvršna datoteka `frontend` čijim pokretanjem se otvara prozor sa već opisanim izgledom. Za kompilaciju grafičkog interfejsa potrebno je imati dostupan Qt6.

5.4 Evaluacija

Ocena rada alata *TestSuiteReduction* će u nastavku biti izvršena na osnovu pokretanja nad različitim skupovima testova. Korišćena su dva projekta – studentski projekat sa kursa Geometrijski algoritmi [2] na Matematičkom fakultetu[5], kao projekat malog obima i projekat *AWS SDK za C++* [1], kao projekat koji ima upotrebu u produkcijskom okruženju. Oba projekta su pisana u jeziku C++ i koriste Google testove za testiranje.

Analiza rezultata projekta ***GeometrijskiAlgoritmi***

Studentski projekat *GeometrijskiAlgoritmi* sadrži 20 testova i 40 izvornih datoteka sa oko 7000 linija koda čija se pokrivenost prati, izuzimajući sistemske biblioteke. Za kompilaciju se koristi alat *qmake*[6] i potrebno je dodati naredne linije u projektnu datoteku (datoteka sa ekstenzijom `.pro`):

```
1 QMAKE_CXXFLAGS += -g -O0 --coverage \
2         -fprofile-filter-files=algoritmi_studentski_projekti/*
3 QMAKE_LFLAGS += -lgcov --coverage
```

Opcijom `-fprofile-filter-files` se zadaje regularni izraz koji opisuje nazive svih izvornih datoteka čija će se pokrivenost posmatrati nakon kompilacije.

Jedno izvršavanje alata *TestSuiteReduction* nad testovima projekta *GeometrijskiAlgoritmi* utroši oko 800 milisekundi za profajliranje koda i generisanje matrice pokrivenosti, nakon čega je rešavaču *Z3* potrebno oko 4 milisekunde za pronalaženje rešenja zadatog problema. Rezultati rada alata za različite vrste pokrivenosti i koristeći izvornu datoteku kao jedinicu koda prikazani su u tabeli 5.1, u kojoj se može videti i kako se menja vreme izvršavanja skupa koji ne sadrži testove koji su proglašeni redundantnim, kao i ukupan broj generisanih klauza. S obzirom da postoje

[5]Autori projekta su dr Danijela Simić, asistent na kursu, i studenti koji su pohađali kurs školske 2018/19. godine.

[6]Alat *qmake* je deo projekta Qt i služi za upravljanje procesom kompilacije.

izvorne datoteke koje ne sadrže grananja, broj klauza u slučaju korišćenja pokrivenosti grana je manji u odnosu na druge dve vrste pokrivenosti[7]. Broj promenljivih koje se koriste u klauzama jednak je broju testova, tj. ima ih 20. Vreme izvršavanja polaznog skupa testova je oko 250 milisekundi. Matrica pokrivenosti koja se generiše ima dimenzije 20×43.

	Pokrivenost linija	**Pokrivenost grana**	**Pokrivenost funkcija**
`ga02_konveksniomotac.GATesst`	+	+	+
`ga02_unija_pravougaonika.file1Test`	-	-	-
`ga02_unija_pravougaonika.file2Test`	-	-	+
`ga02_unija_pravougaonika.file3Test`	-	-	+
`ga04_kdtree.file1Test`	-	-	-
`ga04_kdtree.file2Test`	-	-	+
`ga04_kdtree.file3Test`	-	-	+
`ga07_ortogonalni_upiti.isto_resenje_naivnog_i_optimalnog_FAJL`	-	-	-
`ga07_rangetree.cvor_podele`	-	-	-
`ga07_rangetree.jednoclano_stablo`	-	-	-
`ga07_rangetree.prazno_stablo_greska`	+	+	+
`ga07_rangetree.testiraj_duplikate`	-	-	-
`ga07_rangetree.testiraj_iste_koordinate`	-	-	+
`ga08_hertelmehlhorn.convexPolygonTest`	-	-	-
`ga08_hertelmehlhorn.emptyPolygonTest`	-	-	-
`ga08_hertelmehlhorn.fileTest`	-	-	+
`ga10_trilateracija.file1TestLat`	-	-	-
`ga10_trilateracija.file1TestLon`	+	+	+
`ga10_trilateracija.file2TestLat`	+	+	+
`ga10_trilateracija.file2TestLon`	+	+	+
Ukupan broj redundantnih testova	**5**	**5**	**11**
Vreme izvršavanja redukovanog skupa	**200** ms	**200** ms	**110** ms
Broj generisanih klauza	**52**	**49**	**52**

Tabela 5.1: Pregled testova iz projekta *GeometrijskiAlgoritmi* koji su proglašeni redundantnim zavisno od korišćene vrste pokrivenosti, sa pregledom ukupnog broja takvih testova, vremenom izvršavanja skupa testova kada se redundantni uklone i brojem generisanih klauza. Simbol + označava test koji je proglašen redundantnim, a - označava test koji je zadržan u redukovanom skupu. Projekat sadrži ukupno 20 testova.

Može se primetiti da se odabirom pokrivenosti funkcija dobija gruba slika redundantnosti u celom skupu testova, ali da su u tom slučaju neki testovi proglašeni suvišnim iako se korišćenjem pokrivenosti nižeg nivoa, odnosno pokrivenosti linija i grana dolazi do zaključka da je manji broj testova redundantan. Na primer, testovi `ga08_hertelmehlhorn.convexPolygonTest` i `ga08_hertelmehlhorn.fileTest` se zadržavaju u redukovanom skupu ako se koristi pokrivenost nižeg nivoa, a jedan od njih biva izbačen u slučaju korišćenja pokrivenosti funkcija, jer oba testa proveravaju

[7] Osim klauza koje se odnose na granice promenljivih koje odgovaraju testovima i koje su oblika `(assert (>= t1 0))` i `(assert (<= t1 1))`, preostale klauze se generišu po jedna za svaku jedinicu koda. Ukoliko je pokrivenost u toj jedinici koda nula, klauza za nju neće biti generisana.

rad iste funkcije, ali u prvom slučaju prosleđeni ulaz je jednostavan i izvršavanje se brzo završava, dok drugi test proverava izvršavanje čitave funkcije. Iz skupa testova `ga10_trilateracija` zadržava se samo jedan od testova za svaki od kriterijuma, jer se proverava izvršavanje algoritma u kome ne postoje grananja i petlje. Za svaki odabrani kriterijum ručnom analizom koda utvrđeno je da su odabrani redundantni testovi zaista redundantni prema tom kriterijumu.

Analiza rezultata projekta *AWS SDK za C++*

Drugi projekat na kome je testiran alat za automatsko uklanjanje redundantnih testova je *AWS SDK za C++*, koji koristi *cmake* za kompilaciju, pa je potrebno dodati naredne linije u datoteku *CMakeLists.txt*:

```
1 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -g -O0 --coverage \
2     -fprofile-filter-files=aws-cpp-sdk.*\.cpp")
3 set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} \
4     -lgcov --coverage")
```

Kao i u prethodnom slučaju, pomoću opcije `-fprofile-filter-files` ograničava se praćenje pokrivenosti na izvorne datoteke iz ovog projekta. Ukupan broj testova u projektu *AWS SDK za C++* je 2522, a izvornih datoteka 41492. Skup testova koji se nadalje posmatra naziva se `aws-cpp-sdk-core-tests` i sadrži 472 testa, pri čemu se koristi 107 izvornih datoteka sa oko 25000 linija koda.

Generisanje matrice pokrivenosti je vremenski najzahtevniji deo izvršavanja, što zbog izvršavanja velikog broja testova, što zbog parsiranja dobijenih izveštaja o pokrivenosti. U slučaju ovog projekta taj proces traje oko 5 minuta, dok rešavaču *Z3* treba oko 30 milisekundi za pronalaženje rešenja. Pre uklanjanja redundantnih testova, izvršavanje neizmenjenog skupa testova traje oko 35 sekundi. Pokretanjem alata za različite vrste pokrivenosti i za izvornu datoteku kao jedinicu koda, dobijeni su redukovani skupovi testova kao što je prikazano u tabeli 5.2, u kojoj se može videti i vreme potrebno za izvršavanje tako redukovanih skupova. Naveden je i broj generisanih klauza koje se prosleđuju rešavaču. Kao što je slučaj bio kod projekta *GeometrijskiAlgoritmi*, broj klauza koje se generišu kada se odabere pokrivenost grana je manji u odnosu na druge dve vrste pokrivenosti. Broj promenljivih korišćenih u klauzama jednak je broju testova – 472. Zbog velikog broja testova, nazivi testova nisu navedeni, već samo broj onih koji su proglašeni redundantnim. Matrica pokrivenosti koja se generiše ima dimenzije 472×106. U slučaju projekta

AWS SDK za C++ se, takođe, može primetiti da pokrivenost visokog nivoa, odnosno pokrivenost funkcija, predstavlja stroži uslov na osnovu kog se veći broj testova proglašava redundantnim. Posmatranje pokrivenost grana predstavlja analizu nižeg nivoa od prethodnog i time se u redukovanom skupu ostavlja veći broj testova, dok se prilikom posmatranja pokrivenosti linija uočavaju najveće razlike između testova, pa je skup redundantnih najmanji. Na ovaj način je skup testova u ovom projektu prepolovljen.

	Pokrivenost linija	Pokrivenost grana	Pokrivenost funkcija
Broj redundantnih testova	**212**	**247**	**352**
Vreme izvršavanja redukovanog skupa	**25** s	**23** s	**18** s
Broj generisanih klauza	**1035**	**1020**	**1035**

Tabela 5.2: Broj testova iz projekta *AWS SDK za C++* koji su proglašeni redundantnim zavisno od korišćene vrste pokrivenosti, vreme izvršavanja skupa testova nakon što se uklone testovi proglašeni redundantnim i broj generisanih klauza prosleđenih rešavaču. Skup testova koji se analizira sadrži 472 testa.

Na ovim primerima se može uočiti i mana ovog pristupa u kome se na osnovu pokrivenosti zaključuje redundantnost. Neki testovi koji su proglašeni redundantnim, to zapravo nisu, kao što je slučaj sa testom `TestThatProviderRefreshes` iz skupa testova `TaskRoleCredentialsProviderTest`. Ovaj alat zadržava test `TestThatProviderDontRefresh`, a `TestThatProviderRefreshes` proglašava redundantnim, jer se pokrivenosti jedinica koda (odnosno izvornih datoteka) za ova dva testa podudaraju. Međutim, nakon analiziranja samih testova, može se videti da iako oba testa vrše pozive istih funkcija, oni proveravaju različite aspekte ponašanja sistema. Test `TestThatProviderRefreshes` proverava da li će sistem obnoviti kredencijale nakon što im rok važenja istekne, a test `TestThatProviderDontRefresh` proverava da li sistem zadržava iste kredencijale ukoliko im rok važenja još uvek nije istekao. Ukoliko bi se upoređivali podaci nad kojim ovi testovi rade, umesto pokrivenosti koju postižu, moglo bi se zaključiti da se radi o testovima koji proveravaju različite funkcionalnosti sistema, odnosno ulazi ovih testova pripadaju različitim klasama ekvivalencije.

Analizom izgrađene matrice pokrivenosti se za svaki test koji je proglašen re-

dundantnim može naći primer drugog testa koji zaista pokriva neku jedinicu koda u istoj meri. Analizom koda takvih testova se možemo uveriti da su u pitanju testovi koji vrše pozive istih funkcija ili da će za date ulaze prolaziti kroz iste grane u kodu, ali se na nekoliko izdvojenih primera može primetiti da to nije uvek dovoljno. Zato možemo zaključiti da je skup testova previše redukovan, iako su rezultati dobijeni pokretanjem alata *TestSuiteReduction* u skladu sa zadatim kriterijumima. Odabrane mere su isuviše grube i za preciznije rezultate neophodno je unaprediti korišćenu strategiju.

Glava 6

Zaključak

U radu su opisane tehnike testiranja i osnovna podela tehnika na testiranje bele i crne kutije, u zavisnosti od dostupnosti koda softvera koji se testira. Objašnjena je pokrivenost koda kao kriterijum koji se može koristiti prilikom analize uspešnosti skupa testova da pronađe greške. Predstavljen je postupak automatskog redukovanja skupa testova uklanjanjem redundantnih testova. Za implementaciju alata kojim se demonstrira taj postupak odabran je jezik C++. Automatizacija dobijanja rešenja se u velikoj meri oslanja na korišćenje SMT rešavača *Z3*. Dobijen je alat koji može pomoći u odabiru testova koji se smatraju suvišnim i time može ubrzati izvršavanje testova datog projekta.

Alat *TestSuiteReduction* je implementiran na takav način da se dodavanje podrške za novo okruženje za testiranje, alat za analizu pokrivenosti ili rešavač svodi na nasleđivanje jedne od osnovnih klasa i definisanje neophodnih funkcija kojima se opisuje željeno ponašanje. Dodavanjem podrške za druge alate mogu se, na primer, obezbediti drugačiji nivoi pokrivenosti. Dobijeni rezultati se mogu iskoristiti za uklanjanje redundantnih testova ili za upoređivanje sa rezultatima koji su dobijeni korišćenjem drugih metoda.

Implementirani alat bi se dalje mogao unaprediti definisanjem kompleksnijih metrika koje se mogu koristiti za poređenje dva testa. Na primer, mogla bi se definisati funkcija koja za jedno izvršavanje prati ne samo procentualnu pokrivenost linija, već i skup izvršenih linija koda. To se može postići implementacijom heš funkcije koja bi sa velikom verovatnoćom vraćala jedinstvenu vrednost za dati skup linija koje su pokrivene od strane jednog testa. Takođe, dodatni kriterijum za minimizovanje skupa testova može biti vreme izvršavanja, odnosno može se davati prednost testovima koji se brže izvršavaju kada postoje dva testa koji daju istu pokrivenost.

Bibliografija

[1] AWS SDK for C++, GitHub repository. https://github.com/aws/aws-sdk-cpp. Version 1.9.257.

[2] Geometrijski algoritmi @ MATF, repozitorijum na BitBucket-u. https://bitbucket.org/geoalg1819/c/src/master/.

[3] GoogleTest. https://google.github.io/googletest/.

[4] Repozitorijum za master rad na Bitbucket-u. https://bitbucket.org/blirena/master.

[5] A survey on Test Suite Reduction frameworks and tools. *International Journal of Information Management*, 36(6, Part A):963–975, 2016.

[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.*, 51(3), 2018.

[7] José Campos and Rui Abreu. Leveraging a Constraint Solver for Minimizing Test Suites. In *2013 13th International Conference on Quality Software*, pages 253–259, 2013.

[8] The Qt Company. Qt documentation. https://doc.qt.io/.

[9] Lee Copeland. *A Practicioner's Guide to Software Test Design*. Artech House Publishers, 2004.

[10] Inc. David R. Cok, GrammaTech. The SMT-LIBv2 Language and Tools: A Tutorial. https://www.lri.fr/~conchon/TER/2013/2/SMTLIB2.pdf.

[11] Leonardo de Moura and Nikolaj Bjørner. Proofs and Refutations, and Z3. volume 418, 01 2008.

[12] Arie Van Deursen, Leon Moonen, Alex Bergh, and Gerard Kok. Refactoring Test Code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001*, pages 92–95, 2001.

[13] Milan Banković Filip Marić. Matematički fakultet, Univerzitet u Beogradu - Automatsko rezonovanje, materijali i slajdovi sa predavanja. `http://poincare.matf.bg.ac.rs/~filip/ar/`.

[14] froglogic GmbH. FrogLogic Coco Code Coverage, code coverage analysis for C, C++, SystemC, C#, Tcl and QML code. `https://www.froglogic.com/coco/features/coverage-levels/`.

[15] Bryan Gillespie. fastcov - A parallelized gcov wrapper for generating intermediate coverage formats fast. `https://github.com/RPGillespie6/fastcov`. Version 1.14.

[16] Verifysoft Technology GmbH. Testwell CTC++, test coverage analyser for C/C++. `https://www.testwell.fi/ctcdesc.html`.

[17] Milena Vujošević Janičić. Matematički fakultet, Univerzitet u Beogradu - Verifikacija softvera, materijali i slajdovi sa predavanja. `http://www.verifikacijasoftvera.matf.bg.ac.rs`.

[18] Negar Koochakzadeh and Vahid Garousi. A Tester-Assisted Methodology for Test Redundancy Detection. *Advances in Software Engineering*, 2010, 2010.

[19] Negar Koochakzadeh, Vahid Garousi, and Frank Maurer. Test Redundancy Measurement Based on Coverage Information: Evaluations and Lessons Learned. In *2009 International Conference on Software Testing Verification and Validation*, pages 220–229, 2009.

[20] Breno Miranda, Emilio Cruciani, Roberto Verdecchia, and Antonia Bertolino. FAST Approaches to Scalable Similarity-Based Test Case Prioritization. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 222–232, 2018.

[21] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 2011.

[22] Parasoft. Parasoft C/C++test, testing solution for C/C++ software development. https://www.parasoft.com/products/parasoft-c-ctest/c-c-coverage-traceability/.

[23] Ron Patton. *Software Testing*. Sams Publishing, 2006.

[24] GNU Project. Gcov, test coverage program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[25] Bullseye Testing Technology. BullseyeCoverage, C++ coverage tool. https://www.bullseye.com/metrics.html.

[26] Luca Trevisan. Stanford University — CS254: Computational Complexity. https://www.cs.stanford.edu/~trevisan/cs254-10/.

[27] Luca Trevisan. Stanford University — CS261: Optimization and Algorithmic Paradigms. https://theory.stanford.edu/~trevisan/cs261/.

[28] Arash Vahabzadeh, Andrea Stocco, and Ali Mesbah. Fine-Grained Test Minimization. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 210–221, 2018.

[29] D. Wallace, A. Watson, and T. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, 1996.

[30] Tao Xie, D. Notkin, and D. Marinov. Rostra: a framework for detecting redundant object-oriented unit tests. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 196–205, 2004.

[31] S. Yoo and M. Harman. Regression Testing Minimization, Selection and Prioritization: A Survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, mar 2012.

Biografija autora

Irena Blagojević (*Smederevo, 9. oktobar 1994.*) je informatičar i trenutno radi na poziciji *SysOps/SRE* u kompaniji *Everseen*. Završila je Gimnaziju u Smederevu, prirodno-matematički smer, 2013. godine. Iste godine upisala je Matematički fakultet Univerziteta u Beogradu i završila osnovne akademske studije 2017. godine, nakon čega upisuje master studije na istom smeru.

Zaposlila se 2018. godine kao *DevOps* inženjer. Odgovornosti na njenom trenutnom radnom mestu podrazumevaju osmišljavanje i implementiranje automatizacije ručnog posla koji se obavlja prilikom pomeranja na narednu verziju softvera, uz nadgledanje njegovog ponašanja u produkcijskom okruženju. Njena interesovanja uključuju prakse za smanjivanje mogućnosti pojave sigurnosnih incidenata i uočavanje procesa koji se mogu sistematizovati u automatizovane procedure zarad povećavanja stabilnosti sistema.