УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Никола Перић

АЛАТ ЗА ГЕНЕРИСАЊЕ И ПРИКАЗ РАЗЛИКА У ПОКРИВЕНОСТИ КОДА ТЕСТОВИМА

мастер рад

Београд, 2023.

Ментор:

проф. др Милена Вујошевић Јаничић, ванредни професор
Универзитет у Београду, Математички факултет

Чланови комисије:

проф. др Филип Марић, редовни професор
Универзитет у Београду, Математички факултет

доц. др Мирко Спасић, доцент
Универзитет у Београду, Математички факултет

Датум одбране: ______________________

Мајци, оцу и брату

Наслов мастер рада: Алат за генерисање и приказ разлика у покривености кода тестовима

Резиме: Тестирање путања је део сваког развоја софтвера. Једна од основних мера квалитета тестова је покривеност кода. Често коришћени алати за мерење покривености кода попут алата *gcov* и *lcov* дају кумулативне информације о покривености кода при покретању више од једног теста. У оквиру рада је представљена имплементација алата *CovDiff* који служи за генерисање и приказ разлика у информацијама о покривености кода које су настале након независног покретања два теста. Алат *CovDiff* представља надоградњу алата *gcov*. Разлике се приказују на страницама у формату *html* помоћу неколико различитих приказа. Помоћу њих корисник алата може лако упоредити путање које покривају два теста и уочити потенцијалне грешке у изворном коду на местима где се праве разлике а да се то не очекује.

Кључне речи: тестирање, покривеност кода, инструментализација кода, *gcov*, *lcov*, *LLVM*

Садржај

1 Увод **1**

2 Основе тестирања софтвера **3**

2.1 Тестирање . . . 3

2.2 Тестирање путања и покривеност кода . . . 10

3 Компилаторска инфраструктура *LLVM* **16**

4 Постојеће технологије за мерење покривености кода **19**

4.1 Инструментализација кода . . . 19

4.2 Алат *gcov* . . . 21

4.3 Алат *lcov* . . . 24

4.4 Алат *llvm-cov* . . . 26

5 Имплементација алата *CovDiff* **29**

5.1 Структуре података . . . 31

5.2 Покретање тестова, обрада и чување резултата . . . 33

5.3 Приказ информација о покривености кода . . . 39

5.4 Аргументи командне линије . . . 51

5.5 Зависности . . . 52

5.6 Тестирање рада алата . . . 52

6 Закључак **58**

Библиографија **60**

Глава 1

Увод

Тестирање представља врсту динамичке анализе софтвера [25, 30, 7]. Помаже да се у раним фазама развоја софтвера открију грешке и тако побољша његов квалитет. Уколико је позната имплементација и унутрашња структура софтвера онда се прибегава стратегији тестирања беле кутије (енг. *white-box testing*).

При стратегији тестирања беле кутије циљ је писати тест примере тако да тестирају што већи број путања кроз програм. У већини случајева путања кроз програм има велики број и није могуће тестирати их све. Тада се прибегава мерама квалитета тестова каква је покривеност кода (енг. *code coverage*).

Покривеност кода представља степен извршавања изворног кода приликом тестирања. Висока покривеност сугерише да је велики део путања кроз програм покривен тест примерима што смањује могућност постојања неоткривених грешака.

Мерење покривености кода захтева инструментализацију кода. Процесом инструментализације се у изворни кôд убацују инструкције чији је задатак да мере број извршавања одређених делова кода. На тај начин се прикупљају информације и формира извештај о покривености кода.

Програмски преводиоци какви су *GCC* [21] и *Clang* [11], нуде различите механизме за инструментализацију кода. Уз њих постоје и алати који на основу прикупљених информација о покривености кода генеришу извештаје у читљивим форматима. Међу таквим алатима издвајају се алат *gcov* [18] који је део компилаторске колекције *GCC*, алат *lcov* [26] као његова графичка надоградња и алат *llvm-cov* [28] који је део компилаторске инфраструктуре

LLVM [35].

Поменути алати пружају информације о покривености кода ограничене на покретање једног теста. Уколико би се догодило узастопно покретање два теста, генерисане информације би представљале њихову кумулативну покривеност кода. Такве информације су често потребне, међутим постоје и ситуације у којима је потребно да се виде разлике у покривености кода коју та два теста нуде.

У раду је представљена имплементација алата *CovDiff* за генерисање и приказ разлика у покривености кода тестовима. Алат представља надоградњу алата *gcov*. Алат приказује разлике у покривености кода након независног покретања два теста. За случај да се очекује исти излаз за два теста, а то се не догоди, разлике могу сугерисати на места у изворном коду где потенцијално постоји грешка. У оквиру алата *CovDiff* имплементирано је и генерисање приказа разлика у покривености кода помоћу неколико различитих приказа у формату *html*.

У поглављу 2.1 описан је процес тестирања софтвера. Дат је преглед фаза, врста и стратегија тестирања. У поглављу 2.2 описан је значај тестирања путања и покривености кода. Описане су и различите врсте покривености кода које се могу мерити. У поглављу 3 дат је кратак преглед компилаторске инфраструктуре *LLVM*. У поглављу 4.1 описан је механизам инструментализације кода. Поглавља 4.2, 4.3 и 4.4 говоре о постојећим и често коришћеним алатима за мерење покривености кода.

Остатак рада се бави описом имплементације алата *CovDiff*. Поглавља 5.1 и 5.2 говоре о класама које имплементирају обраду информација о покривености кода коју алат *CovDiff* врши. У поглављу 5.3 описан је део алата *CovDiff* одговоран за формирање различитих типова приказа у формату *html*. Поглавља 5.4 и 5.5 говоре о начину употребе алата *CovDiff* и зависностима. У поглављу 5.6 описан је процес тестирања рада алата *CovDiff* над компилаторском инфраструктуром *LLVM*. Поглавље 6 изводи главне закључке рада и истиче могућности за даље унапређење алата *CovDiff*.

Глава 2

Основе тестирања софтвера

Динамичка верификација софтвера обухвата технике испитивања софтвера у току његовог извршавања. Тестирање представља једну од таквих техника. Циљ је да се тестирањем уоче неисправни делови софтвера или да се стекне што веће поверење у његову исправност. Због своје важности, појам тестирања се често поистовећује са појмом верификације софтвера иако је верификација шири појам који обухвата и друге технике [25].

У наставку следи детаљан опис процеса тестирања софтвера. Дат је посебан осврт на тестирање путања програма (енг. *path testing*) и покривеност кода (енг. *code coverage*) као важне мере квалитета тестова.

2.1 Тестирање

Тестирање представља технику динамичке верификације софтвера која је данас неизоставни део процеса развоја софтвера. Оно помаже да се у раним фазама развоја открију грешке и њиховом исправком побољша квалитет софтвера.

Тест пример (енг. *test case*) је документ који дефинише улазе у систем и очекиване излазе за те улазе. Тестирање обухвата процесе планирања, анализе и дизајнирања са циљем да се креирају тест примери који симулирају рад система или компоненте под одређеним условима, као и процесе анализе и бележења резултата добијених након симулације [12].

Тестирањем није могуће доказати исправност програма али јесте могуће показати да програм није исправан [25, 9, 7]. Додатно, вероватноћа постојања нових грешака у неком делу програма је пропорционална броју већ откриве-

них грешака у том делу програма. То значи да делови програма подложни грешкама имају већу вероватноћу постојања неоткривених грешака и добро је уложити додатне ресурсе за тестирање таквих делова [30].

Софтвер се обично развија према захтевима корисника са циљем решавања реалног проблема или са циљем имплементације нових функционалности. Свако одступање од захтева представља грешку у развијеном софтверу. Грешке је немогуће избећи јер су део људске природе програмера али их је неопходно отклонити из разлога што њихово присуство чини софтвер мање квалитетним. Са порастом сложености софтвера расте и значај тестирања из разлога што омогућава да се грешке открију у раним фазама развоја, када је њихово уклањање временски и новчано јефтиније [30, 7].

Због значаја тестирања, многе методологије развоја промовишу имплементацију тестова за сваку од целина софтверског система [34]. Међу таквим методологијама развоја убрајају се и различите агилне методологије развоја какве су екстремно програмирање (енг. *extreme programming*), Скрам (енг. *Scrum*), Канбан (енг. *Kanban*) и друге [30, 16, 13].

Неке методологије у потпуности стављају тестирање у први план. Такав је развој софтвера вођен тестовима (енг. *test-driven development*) који спада у методологије екстремног програмирања, а који подразумева писање тест примера пре писања самог кода [6]. Овај приступ је могућ због чињенице да писање тестова није условљено постојањем кода, већ је спецификација захтева довољна да се зна шта би требало очекивати од софтвера па се сходно захтевима могу писати тестови. Предност методологије развоја вођеног тестовима је већи број написаних тестова што доприноси квалитетнијем софтверу [15].

Фазе тестирања

У општем случају тестирање софтвера се састоји из четири фазе чији редослед прати методологију развоја софтвера која се примењује. То су следеће фазе [19]:

Планирање (енг. *test planning*) представља фазу припреме за процес тестирања и у великој мери зависи од методологије развоја софтвера. У оквиру ове фазе дефинишу се задаци које треба спровести као и начин на који то треба урадити. Дефинишу се врсте тестова које ће се користити узимајући у обзир значај сваке од врста за пројекат. Потом се

дефинише опсег тестирања, приступ, стратегије и методе тестирања али и потребни ресурси и начини комуникације. Задаје се критеријум завршетка у зависности од тога који је ниво квалитета софтвера потребно достићи [30]. Резултат ове фазе представља скуп докумената са описом свих наведених активности и алата који ће у ту сврху бити коришћени.

Анализа, дизајн и имплементација тестова (енг. *test analysis, design and implementation*) је фаза у којој се праве детаљне спецификације начина за извршавање свих планом предвиђених активности. У оквиру ове фазе прецизирају се захтеви корисника, прикупљају подаци о улазима и очекиваним излазима, испитује се могућност тестирања одређених делова кода и креирају се упутства за тестирање. Резултат ове фазе јесте скуп тест примера и тест процедура.

Извршавање тестова (енг. *test execution*) је фаза у којој се извршавају тест примери дефинисани у претходној фази и на тај начин врши провера функционисања система. Тест примери се могу извршавати ручно или аутоматски. Поред тога, ова фаза је одговорна и за дефинисање приоритета по којем ће тест примери бити извршени. Приоритети могу бити дефинисани по брзини извршавања тест примера или по њиховом значају за функционисање система. Уколико неки од тест примера није задовољен потребно је обавестити особу одговорну за део система који тај тест пример тестира. Проблем треба бити забележен у систему за праћење грешака, а након поправке грешке потребно је поново извршити тестове.

Евалуација тестова (енг. *test evaluation*) је фаза у којој се врши процена текућег стања квалитета софтвера на основу резултата извршавања тестова. У оквиру ове фазе се формира извештај са описом онога шта је тестирано и који су разултати тестирања. На основу извештаја се утврђује да ли је систем у складу са захтевима корисника, односно да ли је софтвер спреман за коршћење. Тестирање се затвара када је софтвер испоручен кориснику под условом да испоручилац није одговоран за његово одржавање. Тестирање може бити затворено и у другим ситуацијама, на пример када је пројекат отказан или када је на основу тестирања утврђено да неки циљ не може бити постигнут. Након затва-

рања тестирања документација и тест скрипте настали у току процеса тестирања се архивирају.

Врсте тестирања

Врсте тестирања се могу поделити у две групе. То су **функционално тестирање** (енг. *functional testing*) које испитује функционалну исправност софтвера и **нефункционално тестирање** (енг. *non-functional testing*) које испитује технички квалитет софтвера [30, 12]. Функционално тестирање обухвата тестирање на различитим нивоима какви су појединачни модули програма, групе модула или цео систем. Према нивоу тестирања разликују се [30, 12, 7]:

Јединично тестирање (енг. *unit testing*) проверава исправност малих делова кода који се могу испитати независно. У зависности од парадигме то могу бити класе (*C++* и *Java*), функције (*C*) или то може бити цео програм писан у мање структуираним језицима (*Basic*, *COBOL*). Јединични тестови дефинисани су стандардом за јединично тестирање софтвера *IEEE* [23]. Ову врсту тестова пише програмер и циљ је проверити да ли изоловани делови кода исправно функционишу [33, 14].

Компонентно тестирање (енг. *component testing*) проверава исправност компоненте састављене од више јединица кода. Компонента представља скуп повезаних јединица кода које имају заједнички интерфејс ка осталим јединицама кода. Ова врста тестирања више не третира јединице кода као потпуно изоловане целине већ се оне интегришу у компоненте и тестира се њихово функционисање у таквим целинама. Међутим, на овом нивоу тестирања и даље постоји излованост компоненте од других компоненти и остатка система.

Интеграционо тестирање (енг. *integration testing*) проверава да ли су везе између компоненти добро дефинисане и да ли исправно функционишу. На овом нивоу тестирања компоненте се више не третирају као изловане већ се интегришу у шире целине. Постоје различите стратегије за тестирање интеграције компоненти међу којима се издвајају стратегија Великог праска (енг. *Big-bang testing*) која за циљ има интергацију свих компоненти у једну целину, стратегија одозго на доле (енг. *top-down*

testing) при којој се покушава интеграција компоненти пратећи хијерархију контроле апликације, стратегија одоздо на горе (енг. *bottom-up testing*) при којој се компоненте интегришу према зависностима у употреби [9].

Системско тестирање (енг. *system testing*) обухвата проверу система као целине и испитује да ли се систем понаша у складу са спецификацијама задатим од стране наручиоца. У овај ниво тестирања се убрајају и функционални и нефункционални видови тестирања система. Међу њима су [30]:

Истраживачко тестирање (енг. *exploratory testing*) које проверава различите начине употребе програма са циљем да се препознају, креирају и изврше нови тест примери.

Тестирање прихватљивости (енг. *acceptance testing*) спроводе наручиоци софтвера и крајњи корисници. Циљ је проверити да ли софтвер функционише у складу са потребама корисника. Тестирање прихватљивости може бити референтно тестирање (корисник генерише тест примере који представљају уобичајену употребу система), пилот тестирање (систем се инсталира и тестира се симулирањем свакодневног рада на њему) и паралелно тестирање (након издавања нове верзије система функционишу и стара и нова верзија паралелно).

Тестирање конфигурације (енг. *configuration testing*) је вид нефункционалног тестирања које проверава функционисање система у различитим хардверским и софтверским окружењима. Постоје различите конфигурације хардвера које укључују различит број и типове улазно-излазних уређаја, различите величине меморије, различите комуникационе канале. Поред тога, постоје различити оперативни системи и различита софтверска окружења у којима програми могу да се извршавају (нпр. различите врсте веб претраживача). Овај вид тестирања проверава функционисање система у различитим комбинацијама оваквих и њима сличних конфигурација.

Тестирање капацитета (енг. *volume testing*) је вид нефункционалног тестирања које за циљ има проверу функционисања система

у ситуацијама када је потребно да систем обради велике количине података.

Тестирање компатибилности (енг. *compatibility testing*) је вид нефункционалног тестирања којим се проверава да ли је нова верзија система компатибилна са спољним, већ постојећим системима на које се ослања.

Регресионо тестирање (енг. *regression testing*) се ради након сваке измене у систему и проверава да ли је измена узроковала лош рад других делова система који нису били обухваћени изменом. Обично се изводи покретањем мањег подскупа тест примера [9].

Стратегије тестирања

Стратегије тестирања представљају систематичне методе и технике које се примењују како би се генерисали тест примери. Стратегија се назива ефективном уколико тест примери произведени праћењем стратегије имају велику шансу да открију грешке у програму. За тест примере се још очекује да буду релативно мали, да се брзо извршавају и пружају висок степен поверења у поузданост софтвера. На основу доступности информација о структури и имплементацији софтвера који се тестира, разликују се следеће стратегије тестирања [25, 30, 12]:

Тестирање црне кутије (енг. *black-box testing*) се изводи када није позната структура и имплементација софтвера који се тестира. Назива се још и функционално тестирање (енг. *functional testing*), тестирање понашања (енг. *behavioural testing*) или тестирање вођено подацима (енг. *data-driven testing*). Ова стратегија тестирања ослања се само на спецификацију захтева, односно на очекивано функционисање система из угла корисника. Често није могуће генерисати тест примере који ће да симулирају све могуће улазе у систем. Због тога постоје различити приступи за генерисање прихватљивог броја тест примера који ће са великом вероватноћом открити што већи број грешака [8]. То су:

Тестирање помоћу класа еквиваленције (енг. *equivalence class partition*) је приступ у коме се формирају класе еквиваленције тестова. Класа еквиваленције представља скуп података који се исто

третирају од стране система и који треба да произведу исти резултат. Уколико један тест пример детектује грешку у систему, очекује се да и други тест примери из исте класе еквиваленције детектују ту исту грешку. Уколико тест пример није детектовао грешку, очекује се да је неће детектовати ни други чланови класе. Потребно је детектовати класе еквиваленције тест примера и формирати по један тест пример за сваку. Тиме се смањује број тест примера које је потребно формирати.

Тестирање граничних вредности (енг. *boundary-value analysis*) је приступ који се ослања на детектовање класа еквиваленције а потом и детектовање граница сваке од класа еквиваленције. Када су детектоване границе потребно је формирати по један тест пример за сваку граничну вредност. Додатно, потребно је формирати и по један тест пример за једну вредност изнад и једну вредност испод границе.

Тестирање помоћу табеле одлучивања (енг. *decision table testing*) је приступ који подразумева формирање табеле одлучивања. Табела се састоји из две групе редова. Прву групу редова чине услови дефинисани над улазом, док другу групу редова чине могуће акције. Колоне представљају правила која јединственој кобинацији услова над улазом из прве групе редова додељују акцију у другој групи редова. Тест примери се формирају на основу дефинисаних правила у колонама тако што комбинација редова из прве групе представља улаз у систем, док акција која јој је додељена представља очекивани излаз.

Тестирање помоћу дијаграма стања (енг. *state transition testing*) је приступ у коме се формира дијаграм стања. Дијаграми стања се користе када акције из претходно поменутог приступа зависе и од спољних фактора или од претходних стања система. Стања се приказују као чворови графа док гране представљају прелаз из једног стања у друго узроковано неком акцијом или догађајем из спољашњег света. Тест примере је затим могуће формирати обиласком тако добијеног графа при чему се прави компромис између покривених путева и броја тест примера.

Тестирање погађањем грешака (енг. *error guessing*) је приступ при

којем тестер погађа места на којима би могла да постоји грешка на основу свог искуства и процене.

Тестирање беле кутије кутије (енг. *white-box testing*) подразумева познавање имплементације и унутрашње структуре система који се тестира. Из тог разлога, при овој стратегији, тестове најчешће пише програмер који имплементира софтвер. Како је позната структура кода, потребно је испитати све различите путање кроз програм. Ова стратегија се најчешће користи за писање јединичних тестова и прибегава се мерама квалитета тестова каква је покривеност кода (енг. *code coverage*). Покривеност кода биће детаљно описана у наредном поглављу.

Тестирање сиве кутије кутије (енг. *gray-box testing*) јесте техника тестирања која се користи уколико не постоји приступ комплетном коду већ само делу кода. Тада је могуће комбиновати стратегије тестирања црне и беле кутије. При спровођењу ове стратегије тестирања доступно је шире знање о имплементацији него код тестирања црне кутије. Тада тест примери могу бити прецизније генерисани него при тестирању црне кутије јер је могуће узети у разматрање и путање кроз доступне делове кода а не само улазе и очекиване излазе добијене на основу спецификације захтева [1].

Свака од наведених стратегија за циљ има откривање грешака у систему који се тестира писањем одговарајућих тест примера на систематичан и контролисан начин. Писање тест примера зависи од доступних информација о целокупном систему какве су спецификација захтева, документација кода или сама имплементација решења. Све наведене стратегије имају своје предности и мане једна у односу на другу [31, 24].

2.2 Тестирање путања и покривеност кода

Одабир путања кроз програм које ће бити тестиране представља један од фундаменталних начина за писање тест примера који спроводе програмери при стратегији тестирања беле кутије. За сваку одабрану путању пише се тест пример. Претпоставка је да се грешка десила уколико је извршавање програма спроведено другачијом путањом од очекиване. На пример, уколико се извршавање програма спровело кроз једну грану уместо кроз другу.

За представљање путања кроз програм користи се **граф контроле тока** (енг. *control-flow graph*). Граф контроле тока представља графовску репрезентацију свих путања у програму које могу бити одабране у току његовог извршавања.

Пример графа контроле тока се може видети на слици 2.1. То је усмерени гарф чији чворови представљају основне блокове кода (енг. *basic blocks*). Основи блок кода је низ инструкција програма за који важи следеће:

- постоји само једна тачка уласка у блок односно место на којем почиње извршавање инструкција блока
- постоји само једна тачка изласка из блока односно место на којем се завршава извршавање инструкција блока.

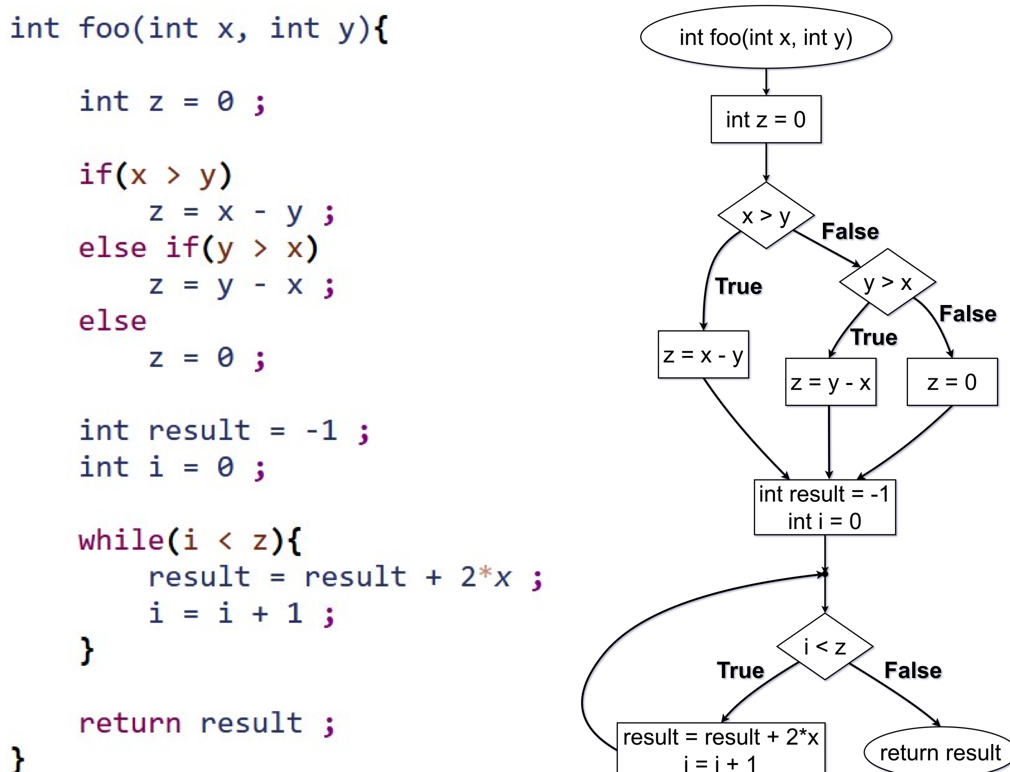
Неформално, основни блок кода се може посматрати као низ инструкција које се извршавају у целини. На десном делу слике 2.1 су правоугаоницима представљени основни блокови кода функције.

Низ инструкција у програму може бити прекинут када се наиђе на инструкцију контроле тока. Гране графа представљају такве тачке у програму и повезују један основни блок са другим основним блоком ка којем контрола извршавања може да оде.

Путање у графу контроле тока могу постојати између произвољних чворова, али су при писању тест примера од интереса путање које почињу оним основним блоком од којег почиње извршавање програма, а завршавају се неким од основних блокова у којем извршавање програма може да се прекине.

Оптимално тестирање путања се своди на писање тест примера који покривају све могуће путање кроз програм. У већини случајева овакав критеријум није могуће задовољити због великог броја путања у програму, па се из тог разлога прибегава другачијим мерама квалитета тестова. Једна од таквих мера јесте **покривеност кода** (енг. *code coverage*). Покривеност кода представља број елемената кода испитаних тестовима у односу на укупан број тих елемената.

Пре почетка тестирања задаје се ниво покривености кода тестовима који је потребно достићи и на тај начин се ограничава број тест примера које треба написати. Могуће је испитивати покривеност различитих елемената кода, па се тако разликују:



Слика 2.1: Функција написана у програмском језику *C* и одговарајући граф контроле тока

Покривеност путања (енг. *path coverage*) — мера проласка кроз могуће путање у програму. Потпуна покривеност путања се постиже обиласком сваке путање барем једном.

Покривеност наредби (енг. *statement coverage*) — мера извршавања наредби. Потпуна покривеност се постиже извршавањем сваке наредбе барем једном.

Покривеност грана (енг. *branch/decision coverage*) — мера проласка кроз гране у програму. Потпуна покривеност се постиже обиласком сваке гране барем једном тј. свака одлука је донета барем једном.

Покривеност услова (енг. *condition coverage*) — мера испитивања услова програма. Потпуна покривеност се постиже тако што је сваки услов у свакој одлуци узео све могуће вредности барем једном.

Покривеност вишеструких услова (енг. *multiple condition coverage*) —

мера испитивања вишеструких услова програма. Потпуна покривеност се постиже тако што је свака могућа комбинација услова у свакој одлуци испитана барем једном.

Покривеност функција (енг. *function coverage*) — мера позива функција програма. Потпуна покривеност се постиже позивом сваке функције барем једном.

Потпуна покривеност грана и наредби су прихваћене као неопходни критеријуми у тестирању софтвера [23]. Уколико није задовољена потпуна покривеност наредби то значи да се при писању тестова на неки начин морала донети одлука о томе који ће делови кода остати нетестирани. Таква одлука може бити оправдана недостатком времена или средстава. Међутим, сваки део кода је подложан грешкама, а у случају да постоје нетестиране наредбе неће постојати ни тест пример који би сугерисао да се грешка јавила у том делу кода. Овај проблем долази још више до изражаја уколико се нетестирани део кода уврсти у већи систем. Све то оправдава захтев за потпуну покривеност наредби.

Наредбе контроле тока су саставни део изворног кода. У зависности од услова који се у таквим наредбама проверава могу се извршити различити блокови кода. Потпуна покривеност грана је неопходан критеријум при писању тест примера како би се провериле наредбе сваког од могућих блокова кода. Из овога се може закључити да потпуна покривеност грана имплицира потпуну покривеност наредби. Импликација важи због тога што ће се потпуним покривањем грана покрити и сваки основни блок графа контроле тока програма који се тестира. Другим речима, уколико су у свакој инструкцији контроле тока тестирани сви исходи тада се у току извршавања програма прошло кроз све блокове кода и самим тим су извршене све наредбе.

Обрнута импликација не важи. Наредбе изворног кода се у већини случајева преводе у више машинских наредби. То је разлог због којег потпуна покривеност наредби изворног кода не гарантује потпуну покривеност машинских наредби већ се остварује само неки проценат покривености машинског кода. Постизањем потпуне покривености грана овај проценат се повећава с обзиром да је тај услов јачи од потпуне покривености наредби.

Претходно тврђење се може илустровати на примеру тернарног оператора у програмском језику *C*. Тернарни оператор оперише над три операнда. Први

операнд се третира као логички израз. Други и трећи операнд представљају израз који ће се извршити у зависности од резултата логичког израза у првом операнду. Овај оператор је у изворном коду записан на једној линији. Међутим, извршавање линије кода на којој се налази тернарни оператор не гарантује покривеност свих машинских наредби у које се овај оператор преводи. Разлог томе је чињеница да се овај оператор преводи у условни скок на машинском нивоу. Условни скок има две гране које воде ка два основна блока. Једном извршена линија изворног кода на којој се овај оператор налази не подразумева да ће се извршити наредбе оба основна блока на машинском нивоу.

Наредбе контроле тока могу у условима које испитују да садрже сложене логичке изразе чији резултат зависи од већег броја параметара (нпр. конјункција више различитих услова). Резултат израчунавања таквих израза једнозначно води у неки од основних блокова кода. Међутим, исти резултат израза је могуће добити помоћу различитих комбинација вредности појединачних параметара који у њему фигуришу. Иако контрола тока води до истог основног блока за сваку тавку комбинацију, не може се говорити о потпуно истим путањама. То је разлог за постојање покривености услова и вишеструких услова.

Покривеност услова и вишеструких услова представљају јачи услов у односу на покривеност грана и захтевају писање тест примера који ће да тестирају различите комбинације параметара сложеног услова. Потпуна покривеност ове две мере имплицира и потпуну покривеност грана [7]. Хијерархија поменутих врста покривености кода се може видети на слици 2.2

Након одабира врсте покривености кода и нивоа покривености који треба постићи, поставља се питање како одабрати путање кроз програм. Тест примере треба писати тако да покривају више мањих путања а не једну компликовану путању. Ако тест примери тестирају мање путање тада је лакше разумети шта може бити узрок грешке уколико до ње дође [7].

Тестирање путања може почети од писања тест примера за очигледне путање кроз програм. Наредни тест примери могу бити писани за путање које су мала варијација претходних путања. Мале варијације у путањи повлаче и лакше писање нових тест примера из разлога што ће се они мало разликовати од претходно написаних тестова. Због таквих малих промена штеди се време у писању тестова и постепено се проверава исправност различитих делова

Слика 2.2: Хијерархија покривености кода. Врсте покривености више у хијерархији имплицирају врсте које су ниже.

кода [7].

Један од начина за одабир путања у графу контроле тока јесте тестирање базних путања (енг. *basis path testing*). Базне путање представљају линеарно независне путање кроз граф контроле тока односно свака базна путања покрива макар једну нову инструкцију у односу на остале базне путање. Укупан број таквих путања се добија израчунавањем цикломатичне комплексности (енг. *cyclomatic complexity*) графа контроле тока по формули

$$C = |E| - |V| + 2 * P$$

где E представља скуп грана графа, V представља скуп чворова графа, а P број повезаних компоненти графа. Од свих базних путања графа контроле тока може се одабрати подскуп путања за које ће бити написани тест примери [7].

Глава 3

Компилаторска инфраструктура *LLVM*

Пројекат *LLVM* [35] представља скуп библиотека и алата који заједно чине велику компилаторску инфраструктуру. То је скуп модуларних и поново искористивих компилаторских технологија које пружају подршку за статичко и динамичко превођење произвољних програмских језика.

Пројекат *LLVM* је отвореног кода (енг. *open source*) под лиценцом *Apache License v2.0* [3] са додатним изузецима. Састоји се из више потпројеката. Примарни потпројекти су:

The LLVM Core libraries је пројекат који пружа модерни оптимизатор независан од изворног кода и циљне архитектуре. Такође пружа и подршку за генерисање кода за различите процесорске архитектуре. Библиотеке су грађене око посебне репрезентације кода која се назива *LLVM* међурепрезентација (енг. *LLVM IR*).

Clang је пројекат који представља предњи део програмског преводиоца и инфраструктуру различитих алата за програмске језике фамилије *C* какви су *C*, *C++*, *Objective C/C++*, *OpenCL*, *CUDA* и *RenderScript* [11]. Део пројекта *Clang* је статички анализатор *Clang* (енг. *Clang Static Analyzer*) и алат *clang-tidy* који аутоматски проналазе грешке у коду.

LLDB је пројекат који имплементира дебагер изграђен над библиотекама које се налазе у пројектима *LLVM Core* и *Clang*.

libc++* и *libc++ ABI су пројекти који пружају имплементацију стандардних библиотека за програмски језик *C++*.

compiler-rt је пројекат који пружа софтверску имплементацију инструкција међукода за које не постоје одговарајуће инструкције у оквиру циљне машине. Обезбеђује и имплементацију библиотека потребних за динамичко тестирање софтвера.

Multi-Level Intermediate Representation (MLIR) је пројекат који има за циљ да помогне у повезивању постојећих програмских преводиоца, као и да унапреди превођење софтвера који се извршава на различитим хардверским компонентама.

Open Multi-Processing (OpenMP) је пројекат који обезбеђује подршку за коришћење конструката *OpenMP* у оквиру пројекта *Clang*. То су конструкти за мулти-платформско вишепроцесорско програмирање са дељеном меморијом у програмским језицима *C*, *C++* и *Fortran*.

polly је пројекат који имплементира оптимизатор петљи и употребе података и представља инфраструктуру за оптимизацију у пројекту *LLVM*.

klee је пројекат који имплементира симболичку виртуелну машину која за циљ има аутоматско генерисање тестова и проналажење грешака у коду.

LLD је пројекат који имплементира нови линкер као замену за системски.

BOLT је пројекат који имплементира оптимизације које се дешавају након линковања. За циљ има оптимизацију кода апликације на основу података прикупљених профајлирањем.

Сви потпројекти заједно чине потпун програмски преводилац који има свој предњи део (енг. *frontend*), средњи део (енг. *middleend*), задњи део (енг. *backend*), оптимизаторе, асемблере, линкере и друге алате који пружају продшку у читавом процесу превођења кода на циљну архитектуру. Предњи део програмског преводиоца обухвата лексичку, синтаксичку и семантичку анализу. Завршава се генерисањем *LLVM* међукода. У средњем делу програмског преводиоца се затим извршавају различите машински независне оптимизације над међукодом. Задњи део програмског преводиоца

обухвата генерисање машинског кода за циљну архитектуру али и машински зависне оптимизације.

Међурепрезентација *LLVM IR* представља хијерархијску структуру која се састоји од неколико ентитета. То су:

- модули — представљају највише ентитете у хијерархији и дефинисани су садржајем датотека у којима се налази међукод
- функције — представљају наредне ентитете у хијерархији и сачињавају један модул
- основни блокови — сачињавају сваку функцију и представљају низ инструкција које се извршавају у целини
- инструкције — сачињавају основне блокове и најнижи су ентитети у хијерархији

Главна особина међукода је та што поштује својство јединственог статичког додељивања (енг. *static single assignment*). То значи да се једној променљивој вредност може доделити само једном. Такође, инструкције међукода су троадресне. То значи да инструкције имају два аргумента док трећи представља локацију за смештање резултата [29].

Предности пројекта *LLVM* лежи у његовој свестраности и флексибилности као и у идеји о поновној употребљивости кода. Све то олакшава грађење нових библиотека и имплементацију подршке за друге програмске језике попут програмских језика *Ruby*, *Python*, *Haskell*, *Rust*, *D*, *PHP* и других [32].

Глава 4

Постојеће технологије за мерење покривености кода

У наредном поглављу биће описани поступак инструментализације кода и постојећи алати за генерисање и приказивање информација о покривености кода. Дат је преглед алата отвореног кода који генеришу извештаје за кôд инструментализован програмским преводиоцима *GCC* и *Clang*.

4.1 Инструментализација кода

Инструментализација кода (енг. *instrumentation*) је поступак којим се убацују нове инструкције у изворни кôд који се преводи или у програм који се извршава. Такве инструкције за циљ имају анализу програма у току његовог извршавања.

Убачене инструкције могу мерити временске и меморијске перформансе програма и у том случају је реч о поступку профајлирања програма (енг. *program profiling*). Профајлирање представља вид динамичке анализе програма чији је резултат скуп података о извршавању програма. Подаци добијени профајлирањем програма представљају његов профил (енг. *profile*). Такви подаци су проценти утрошеног времена у одређеним деловима кода, утрошено време због чекања проточне обраде (енг. *pipline stall*) неке инструкције, информације о путањама кроз кôд које су највише пута извршене, информације о алокацији меморије, информације о броју промашаја у кеш меморији и други [4, 5].

Профајлирање се користи и за мерење покривености кода. Том приликом се убацују различите врсте бројача који броје колико су се пута извршили блокови кода и инструкције гранања или колико су пута позване функције. На основу тих информација формирају се извештаји о покривености кода. У наредном поглављу биће више речи о томе које врсте информација о покривености кода прикупљају програмски преводиоци при овој врсти инструментализације.

Још једна од намена убацивања инструкција јесте детектовање грешака при извршавању програма. Инструкције којима се то постиже убацују алати који се називају санитајзери (енг. *sanitizers*). Санитајзери су део програмских преводилаца [11, 21]. Постоје различите врсте санитајзера, а неке од њих су:

санитајзери адреса — детектују приступ меморији ван граница бафера и приступ меморији која је ослобођена

санитајзери меморије — детектују читање неиницијализоване меморије и откривају случајеве у којима такве вредности утичу на извршавање програма

санитајзери нити — детектују проблеме у раду са нитима какве су грешке приликом трке за ресурсе (енг. *race condition*)

санитајзери цурења меморије — детектују цурење мемоије (енг. *memory leak*) односно случајеве у којима програм не ослобађа меморију која се више не користи

санитајзери недефинисаног понашања — детектују случајеве у којима понашање програма није дефинисано какви су дељење нулом, дереференцирање показивача са вредношћу *NULL*, повратак из функција које би требало да врате вредност али се то не дешава, прекорачење вредности неких типова при извођењу аритметичких операција и други.

Важна особина коју инструментализација кода треба да задовољи је то да се прикупљају само потребни подаци. Превише података успорава програм и саму њихову обраду док, премало података може бити безначајно. Инструментализација кода не сме да утиче на функционалност програма. Уколико она утиче на функционалност тада подаци неће осликавати прави начин рада пограма. Такође, инструментализација кода не би требало да превише успорава рад програма.

Извршавање делова кода који су инструментализовани зависи искључиво од улаза у програм. Може се десити да такви делови не буду извршени при покретању програма па се тада неће прикупити никакви подаци.

Инструментализација кода се може поделити и на основу тога како се убацују нове инструкције у кôд. Инструкције може убацити сам програмер мануелним додавањем у жељене делове кода. Са друге стране, убацивање инструкција се може вршити аутоматски у различитим фазама. Инструкције се аутоматски могу убацити помоћу програмског преводиоца или линкера, могу се додати у већ преведен кôд или се могу убацити за време извршавања самог програма.

4.2 Алат *gcov*

Алат *gcov* [18] је део компилаторске колекције *GNU Compiler Collection* (*GCC*) [21]. *GCC* садржи програмски преводилац за језике *C*, *C++*, *Objective-C*, *Fortran*, *Ada*, *Go* и *D* али и основне библиотеке за те програмске језике. Развијен је у оквиру пројекта *GNU Project* и дистрибуиран под лиценцом *GNU General Public License* (*GNU GPL*) [22] која дозвољава слободно коришћење и модификовање софтвера.

GCC подржава два типа инструментализације програма. Први тип инструментализације има за циљ прикупљање података након профајлирања програма. Други тип инструментализације има за циљ додавање различитих провера у току извршавања програма помоћу санитајзера како би се детектовале грешке.

Инструментализације кода неопходне за анализу покривености кода коју *GCC* нуди, могу се укључити задавањем следећих опција при превођењу програма овим преводиоцем:

- опцијом *-fprofile-arcs* се додају инструкције у кôд којима се у току извршавања програма памте подаци о томе колико су пута све инструкције гранања и позиви функција извршени, колико су пута извршене појединачне гране у инстркцијама гранања и колико се пута програм вратио из позива функција. Када програм заврши са извршавањем ови подаци се памте у бинарним датотекама формата *gcda*, за сваку датотеку са изворним кодом посебно. Ове датотеке се користе за формирање извештаја о покривености кода.

- опцијом *-ftest-coverage* се формирају бинарне датотеке формата *gcno*, за сваку датотеку са изворним кодом посебно. Ове бинарне датотеке садрже информације о томе које линије изворног кода одговарају основним блоковима, као и информације неопходне за реконструкцију графа контроле тока програма.

Алат *gcov* на основу бинарних датотека формата *gcda* и *gcno* формира извештај о покривености кода. Извештај се формира у текстуалном формату унутар датотеке са екстензијом *.gcov*. Свака линија извештаја се сатоји из три колоне које су међусобно одвојене карактером „:”. Прва колона садржи бројач извршавања линије. Друга колона садржи редни број линије у изворном коду. Трећа колона садржи линију изворног кода. Основни формат извештаја се може видети на слици 4.1.

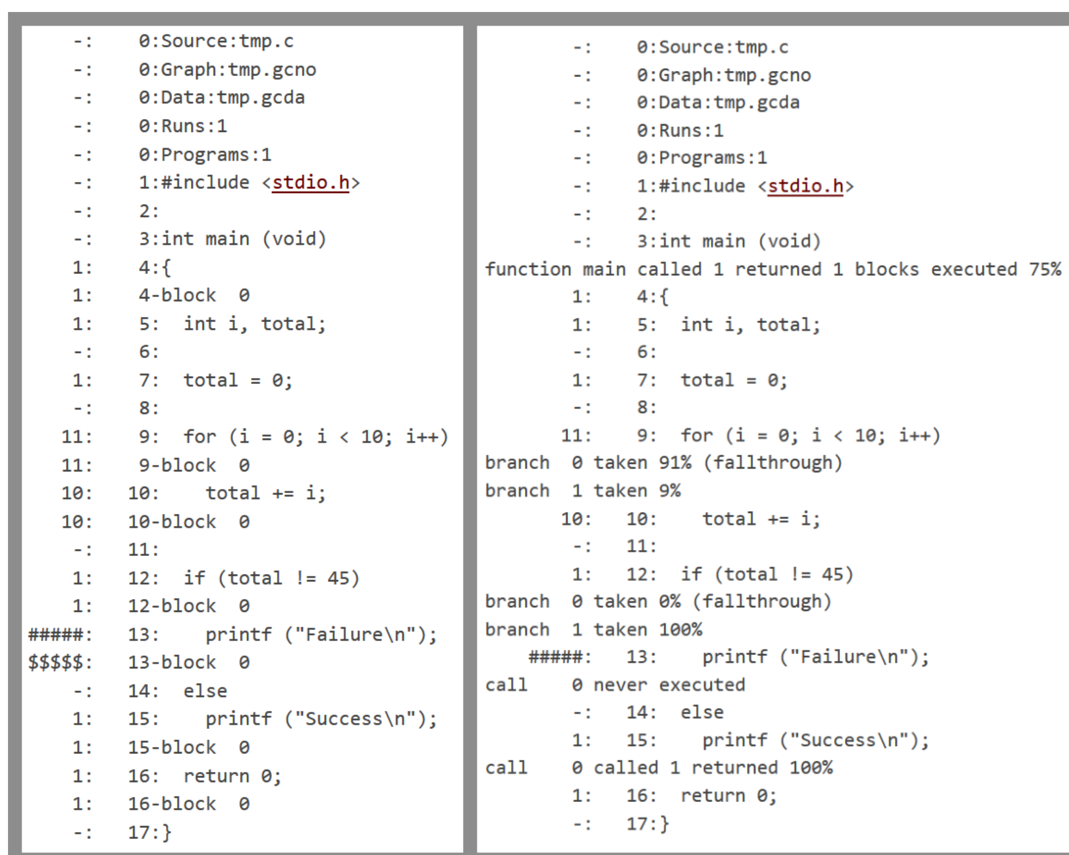
```
        -:    0:Source:tmp.c
        -:    0:Graph:tmp.gcno
        -:    0:Data:tmp.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:
        -:    3:int main (void)
        1:    4:{
        1:    5:  int i, total;
        -:    6:
        1:    7:  total = 0;
        -:    8:
       11:    9:  for (i = 0; i < 10; i++)
       10:   10:    total += i;
        -:   11:
        1:   12:  if (total != 45)
    #####:   13:    printf ("Failure\n");
        -:   14:  else
        1:   15:    printf ("Success\n");
        1:   16:  return 0;
        -:   17:}
```

Слика 4.1: Основни формат извештаја добијен алатом *gcov*

У оквиру прве колоне може се наћи карактер „-” уколико се ради о линијама које се не могу извршити какве су линије са коментарима, претпроцесорским директивама или празне линије. Уколико је у питању линија која се није извршила ни једном, прва колона садржи ознаку „#####”. За све

остале линије прва колона садржи цео број који представља број извршавања линије.

Уколико се алату проследи опција *-a* добијају се бројеви извршавања за сваки основни блок кода. Пример таквог приказа се може видети на слици 4.2. Информације о броју извршавања основног блока кода приказане су на последњој линији блока изворног кода који му одговара, одмах испод извештаја за ту линију. Уколико се више основних блокова завршавају на истој линији, испод ње следе информације о извршавању за сваки од њих. Уколико се основни блок кода није извршио ни једном вредност прве колоне за тај блок има ознаку „$$$$$". У супротном вредност прве колоне садржи цео број који представља број извршавања основног блока.



```
    -:    0:Source:tmp.c
    -:    0:Graph:tmp.gcno
    -:    0:Data:tmp.gcda
    -:    0:Runs:1
    -:    0:Programs:1
    -:    1:#include <stdio.h>
    -:    2:
    -:    3:int main (void)
    1:    4:{
    1:    4-block  0
    1:    5:  int i, total;
    -:    6:
    1:    7:  total = 0;
    -:    8:
   11:    9:  for (i = 0; i < 10; i++)
   11:    9-block  0
   10:   10:    total += i;
   10:   10-block  0
    -:   11:
    1:   12:  if (total != 45)
    1:   12-block  0
#####:   13:    printf ("Failure\n");
$$$$$:   13-block  0
    -:   14:  else
    1:   15:    printf ("Success\n");
    1:   15-block  0
    1:   16:  return 0;
    1:   16-block  0
    -:   17:}
```

```
        -:    0:Source:tmp.c
        -:    0:Graph:tmp.gcno
        -:    0:Data:tmp.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:
        -:    3:int main (void)
function main called 1 returned 1 blocks executed 75%
        1:    4:{
        1:    5:  int i, total;
        -:    6:
        1:    7:  total = 0;
        -:    8:
       11:    9:  for (i = 0; i < 10; i++)
branch  0 taken 91% (fallthrough)
branch  1 taken 9%
       10:   10:    total += i;
        -:   11:
        1:   12:  if (total != 45)
branch  0 taken 0% (fallthrough)
branch  1 taken 100%
    #####:   13:    printf ("Failure\n");
call    0 never executed
        -:   14:  else
        1:   15:    printf ("Success\n");
call    0 called 1 returned 100%
        1:   16:  return 0;
        -:   17:}
```

Слика 4.2: Формати извештаја добијени алатом *gcov* додавањем опције *-a* (лево) и опције *-b* (десно)

Уколико се алату проследи опција *-b* приказују се и информације о извршавању инструкција гранања и позива функција. Овај приказ се може видети на слици 4.2.

За сваку функцију исписује се линија са информацијама о томе колико је пута функција позвана и колико се пута програм вратио из функције. Поред тога исписује се и који је проценат основних блокова кода функције извршен.

За сваки основни блок, испод његове последње линије у изворном коду, исписују се информације о инструкцији гранања или позиву функције којим се тај основни блок завршава. Биће исписано више таквих линија уколико се више основних блокова завршава истом инструкцијом гранања или позивом функције.

За сваку инструкцију гранања која је извршена макар једном биће приказан проценат извршавања њених грана. Тај број се добија као количник броја извршавања гране подељен са укупним бројем извршавања инструкције гранања. За сваки позив функције приказује се количник броја повратака из функције и укупног броја позива функије.

4.3 Алат *lcov*

Алат *lcov* [26] представља графичку надоградњу алата *gcov*. Пружа подршку за формирање извештаја о покривености кода за пројекте који се састоје из великог броја датотека са изворним кодом. Алат формира приказ у формату *html* на основу извештаја о покривености кода који је формиран алатом *gcov*. Приказ прати хијерархију директоријума пројекта и садржи информације о покривености линија, функција и грана. Алат *lcov* је сачињен од скупа алата написаних у програмском језику *Perl* међу којима се издвајају:

- *lcov* — алат за прикупљање информација о покривености кода
- *genhtml* — алат зе креирање приказа у формату *html*
- *gendesc* — алат за креирање описних датотека које користи алат *genhtml*
- *geninfo* — алат за за креирање извештаја о покривености кода у међуформату
- *genpng* — алат за креирање прегледа датотека са изворним кодом у формату *png*

За сваку датотеку са изворним кодом алат генерише по једну страницу у формату *html*. Свака страница садржи изворни кôд одговарајуће датотеке.

Изворном коду су придружене информације о извршавању линија, функција и грана. Поред ових страница, генерисане су и странице за преглед директоријума са списковима датотека које се налазе у њима. Преглед прати хијерархију директоријума пројекта, а свако име датотеке са списка представља везу ка одговарајућој страници са извештајем о покривености кода. Тиме је омогућена лака навигација кроз информације о покривености кода за пројекат.

Све странице у формату *html* смештене су у директоријум чија се путања може задати при покретању алата. Отварањем странице *index.html* у веб претраживачу добија се почетна страница извештаја. У горњем десном углу почетне странице налазе се кумулативни подаци о извршавању линија, грана и функција. У централном делу ове странице приказана је структура директоријума са везама за навигацију ка страницама за појединачне датотеке. Пример почетне странице приказан је на слици 4.3.

LCOV - code coverage report

Current view: top level - lib
Test: coverage.info
Date: 2023-07-25 01:03:01

	Hit	Total	Coverage
Lines:	9	10	90.0 %
Functions:	4	4	100.0 %
Branches:	1	2	50.0 %

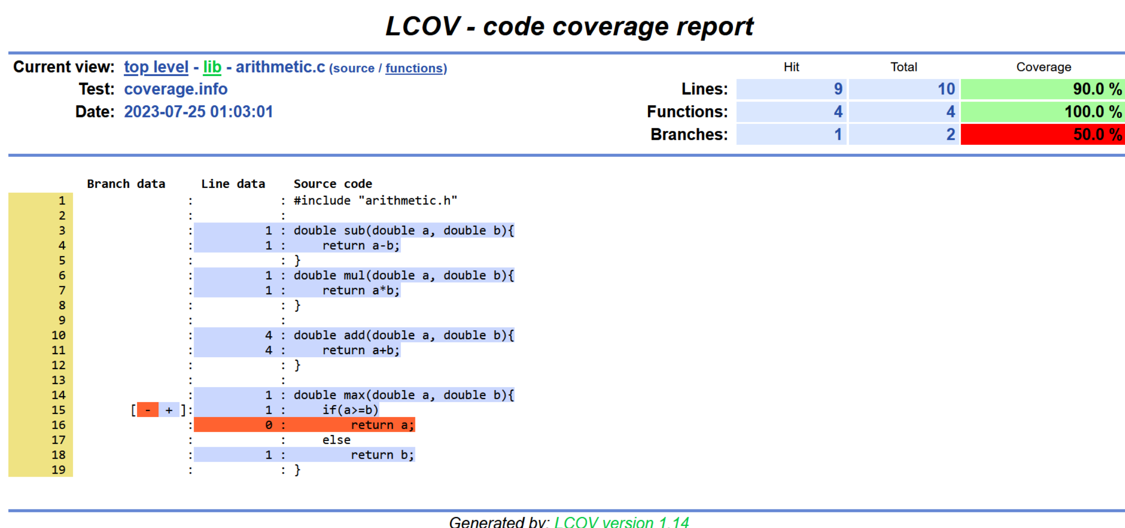
Filename	Line Coverage			Functions		Branches	
arithmetic.c		90.0 %	9 / 10	100.0 %	4 / 4	50.0 %	1 / 2

Generated by: LCOV version 1.14

Слика 4.3: Почетна страница приказа покривености кода добијеног алатом *lcov*

На страницама са приказом извештаја за појединачне датотеке се у горњем десном углу налазе кумулативни подаци о извршавању линија, грана и функција. У централном делу приказане су линије изворног кода датотеке. У колони лево од сваке линије изворног кода налази се бројач извршавања за ту линију. Линије које нису извршене ни једном обојене су црвеном бојом док су остале линије обојене светлоплавом бојом. Ту се може наћи и колона у којој се за сваку инструкцију гранања приказује ознака о томе које су гране извршене. Пример странице са приказом за појединачне датотеке се може видети на слици 4.4.

Информације о покривености грана биће приказане само ако је у конфигурационој датотеци алата *lcov* (*.lcovrc*) опција *lcov_branch_coverage* постављена



Слика 4.4: Приказ покривености кода за датотеку са изворним кодом добијен алатом *lcov*

на 1. Алтернативно, може се задати опција *--rc lcov_branch_coverage=1* при покретању алата.

4.4 Алат *llvm-cov*

Алат *llvm-cov* [28] је део компилаторске инфраструктуре *LLVM*. Помоћу овог алата приказују се информације о покривености кода који је инструментализован алатом *Clang*.

Алат *Clang* подржава три типа инструментализације кода за генерисање информација о покривености:

инструментализација заснована на изворном коду (енг. *source-based*) директно оперише над апстрактним синтаксним стаблом (енг. *abstract syntax tree*)

инструментализација кода помоћу санитајзера (енг. *SanitizerCoverage*) користи се уз различите санитајзере и пружа могућност убацивања корснички дефинисаних функција у кôд

инструментализација компатибилна са *GCC* која може да се користи уз алат *gcov*.

Алат *llvm-cov* се користи да се прочитају информације о покривености кода добијене инструментализацијом заснованом на изворном коду. За извршавање овог типа инструментализације, алату *Clang* је потребно задати одговарајуће опције:

- опцијом *-fprofile-instr-generate* се инструментализује кôд убацивањем посебно алоцираних бројача у меморији који се инкрементирају извршавањем одговарајућег блока кода. Након завршетка извршавања програма вредности бројача се уписују у бинарну датотеку формата *profraw*
- опцијом *-fcoverage-mapping* се генеришу информације које описују мапирање између инструментализованог изворног кода и бројача извршавања

Како би се прочитале информације о покривености кода записане у датотекама формата *profraw* користи се алат *llvm-profdata* који је део пројекта *LLVM*. Овај алат такве датотеке конвертује у датотеке формата *profdata* на основу којих се даље могу формирати читљиви извештаји о покривености кода коришћењем алата *llvm-cov*.

Задавањем опције *show* алату *llvm-cov* генерише се текстуални извештај о покривености кода. Уколико је затада опција *-show-line-counts-or-regions* алат ће генерисати и информације о броју извршавања свих региона односно основних блокова. Информације о извршавању грана и подизраза који се у њима налазе могу се добити задавањем опција *--show-branches=count* и *--show-expansions*.

Текстуални извештај састоји се од три колоне раздвојене карактером „|”. Прва колона садржи број линије у изворном коду. Друга колона садржи број извршавања линије. У трећој колони је приказан садржај линије изворног кода. Линије које не садрже наредбу коју је могуће извршити имају празну другу колону. Такви су коментари, празне линије и претпроцесорске директиве. Линије које се нису извршиле ни једном имају вредност „0” у другој колони. Бројеви извршавања основних блокова приказују се на линији на којој блок почиње. На слици 4.5 је у горњем делу приказан пример текстуалног извештаја, док се у доњем делу слике налази извештај о гранама за једну од функција из примера.

Задавањем опције *report* алат *llvm-cov* исписује кумулативни извештај. Ту су присутне сумарне статистике извршавања основних блокова, функција и грана.

```
    1|   20|#define BAR(x) ((x) || (x))
                            ^20     ^2
    2|    2|template <typename T> void foo(T x) {
    3|   22|  for (unsigned I = 0; I < 10; ++I) { BAR(I); }
                                    ^22     ^20  ^20^20
    4|    2|}
------------------
| void foo<int>(int):
|      2|    1|template <typename T> void foo(T x) {
|      3|   11|  for (unsigned I = 0; I < 10; ++I) { BAR(I); }
|                                      ^11     ^10  ^10^10
|      4|    1|}
------------------
| void foo<float>(int):
|      2|    1|template <typename T> void foo(T x) {
|      3|   11|  for (unsigned I = 0; I < 10; ++I) { BAR(I); }
|                                      ^11     ^10  ^10^10
|      4|    1|}
------------------
```

```
------------------
| void foo<float>(int):
|      2|    1|template <typename T> void foo(T x) {
|      3|   11|  for (unsigned I = 0; I < 10; ++I) { BAR(I); }
|                                      ^11     ^10  ^10^10
|  ------------------
|  |  |    1|     10|#define BAR(x) ((x) || (x))
|  |  |                             ^10     ^1
|  |  |  ------------------
|  |  |  |  Branch (1:17): [True: 9, False: 1]
|  |  |  |  Branch (1:24): [True: 0, False: 1]
|  |  |  ------------------
|  ------------------
|  |  Branch (3:23): [True: 10, False: 1]
|  ------------------
|      4|    1|}
------------------
```

Слика 4.5: Текстуални извештај о покривености кода добијен алатом *llvm-cov*

Алат *llvm-cov* може формирати и извештај у формату *html*. За сваку изворну датотеку која је део извештаја формира се страница у формату *html* са информацијама о извршавању линија. При формирању извештаја у формату *html* прати се структура директоријума у којој се налзе датотеке са изворним кодом. Међутим, за разлику од извештаја у формату *html* добијеног алатом *lcov*, овај извештај нема почетну страницу ни странице за навигацију кроз структуру директоријума.

Глава 5

Имплементација алата *CovDiff*

Претходно описани алати пружају увид у покривеност кода ограничену на покретање једног теста. Уколико би се десило узастопно покретање два теста над инструментализовним пројектом, добиле би се збирне информације о покривености кода. Из таквих информација није могуће закључити каква је мера покривености кода сваког од тестова посебно. Самим тим, није могуће ни упоредити мере покривености кода за два теста. Одатле долази мотивација за имплементацију алата који би превазишао ова ограничења.

Алат *CovDiff* [1] представља надоградњу алата *gcov* која превазилази описане недостатке. Алат генерише и приказује разлике у информацијама о покривености кода које су настале након независног покретања два теста. У току процеса прикупљања информација и рачунања разлика води се рачуна о томе да не дође до њиховог мешања.

Значај разлика у покривености кода лежи у томе што оне представљају једну врсту мере за упоређивање тестова. У случају тестирања путања кроз програм, разлике у покривености кода тестовима могу кориснику алата да сугеришу да ли су тестовима покривене различите путање. Поред тога, разлике у покривености кода су значајне када се за два теста очекује исти излаз, али при извршавању то ипак није случај. На пример, за тестове писане за програмски преводилац, који се разликују само по присуству информација за дебаговање, очекује се да је генерисан исти кôд на излазу. Уколико то није случај, разлике у покривености кода програмског преводиоца за таква два теста могу указати на места у изворном коду преводиоца на којима потенци-

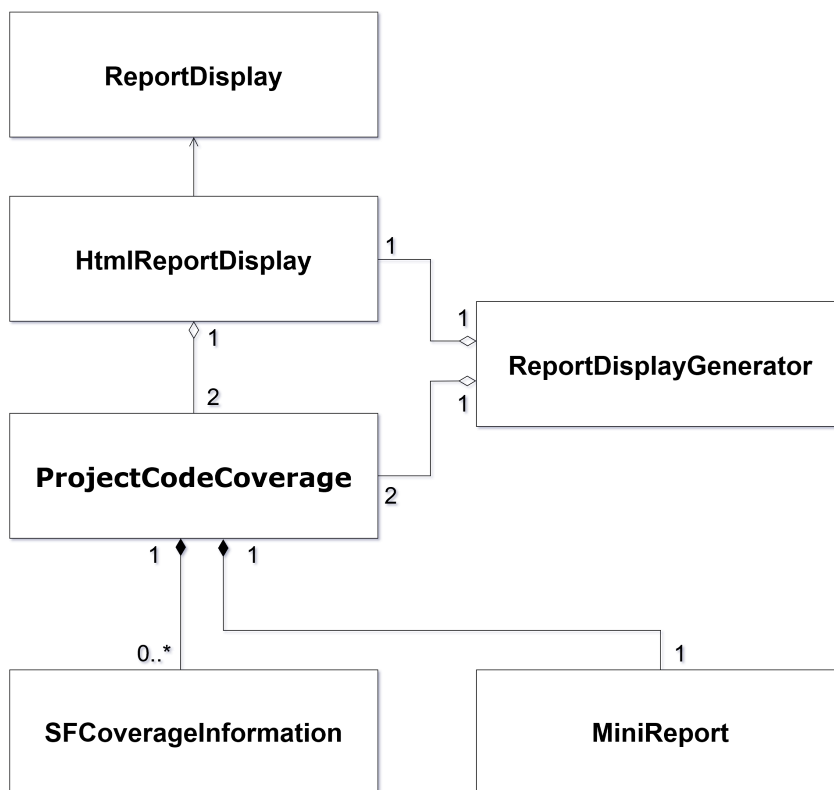
[1] Изворни кôд алата *CovDiff* је јавно доступан на `https://github.com/backspacer303/CovDiff`.

јално постоји грешка.

Алат *CovDiff* приказује резултате у формату *html* по узору на алат *lcov*. Сачувана је идеја о лакој навигацији кроз странице у формату *html* које садрже информације о покривености кода за појединачне изворне датотеке пројекта. Међутим, алат *CovDiff* на тим страницама приказује разлике у покривености кода тестовима и то кроз неколико различитих приказа.

Алат *CovDiff* је имплементиран у програмском језику *Python*. Главни део имплементације представљају класе *ProjectCodeCoverage* и *HtmlReportDisplay*. Класа *ProjectCodeCoverage* је одговорна за покретање тестова, обраду и чување резултата. Она се у позадини ослања на алат *gcov* за генерисање информација о покривености кода. Класа *HtmlReportDisplay* је одговорна за формирање приказа резултата у формату *html*. На слици 5.1 се може видети дијаграм односа између свих класа имплементираних у оквиру алата.

У наставку ће детаљно бити описано функционисање две поменуте класе. Поред тога, биће речи о структурама података за чување резултата али и о начину употребе и захтевима за покретање алата.



Слика 5.1: Дијаграм односа класа алата *CovDiff*

5.1 Структуре података

Структуре података чине класе у оквиру којих се чувају информације прикупљене након покретања тестова. У имплементацији постоје две такве класе. То су класе *SFCoverageInformation* і *MiniReport*.

Класа *SFCoverageInformation* је одговорна за чување информација о покривености кода једне изворне датотеке. У току рада алата, за сваку изворну датотеку за коју постоје информације о покривености кода након извршеног теста, инстанцира се и одржава тачно један објекат ове класе. На слици 5.2 се може видети дијаграм ове класе. Сваки објекат садржи следећа поља:

name садржи назив изворне датотеке.

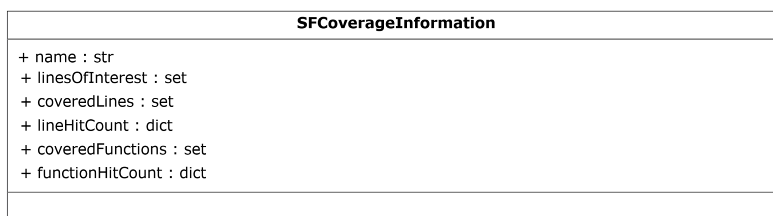
linesOfInterest представља скуп линија од интереса. Линије од интереса су све линије изворног кода које садрже наредбу.

coveredLines представља скуп свих линија изворног кода које су погођене тј. извршене барем једном при покретању теста.

lineHitCount представља мапу која сваки елемент скупа *coveredLines* слика у ненегативан цео број који представља број извршавања одговарајуће линије изворног кода.

coveredFunctions представља скуп имена функција изворне датотеке које су извршене макар једном.

functionHitCount представља мапу аналогну мапи *lineHitCount* која имена функција слика у одговарајући број извршавања.



Слика 5.2: Дијаграм класе *SFCoverageInformation*

Објекти класе *SFCoverageInformation* имају улогу у посредовању информација о покривености кода између дела алата који се бави прикупљањем

информација и дела који се бави генерисањем приказа у формату *html*. На поља ових објеката се реферише и при генерисању разлика у покривености кода тестовима, на нивоу једне изворне датотеке. Из тог разлога су поља *coveredLines* и *coveredFunctions* моделована коришћењем скупова, што омогућава лако извођење операција уније, пресека и разлике.

Класа *MiniReport* је одговорна за чување различитих типова збирних информација након покретања теста. Улога класе *MiniReport* је да пружи глобални увид у утицај једног теста на кôд, односно да се помоћу збирних информација које чувају објекти ове класе стекне први утисак о покривености кода након покретања теста. У складу са тим, у току рада алата се инстанцира један објекат ове класе који одговара покретању једног теста. На слици 5.3 се може видети дијаграм ове класе. Објекти класе *MiniReport* садрже следећа поља:

numOfProcessedReports је укупан број обрађених извештаја. Тај број може бити већи од броја обрађених датотека формата *gcda* из разлога што једна таква датотека може садржати извештаје за више од једне датотеке изворног кода. Најчешћи такав пример су датотеке заглавља чије се информације о покривености налазе и у датотекама формата *gcda* других изворних датотека у које су укључене.

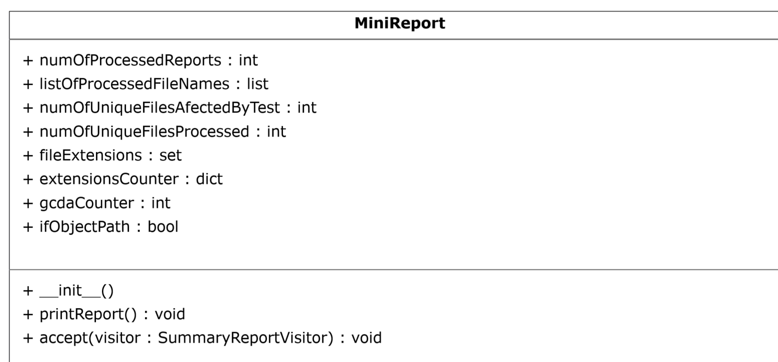
listOfProcessedFileNames представља листу имена свих датотека на које се наишло у току обраде информација о покривености кода. Многа имена у листи се понављају више пута. Поново, најчешћи такав пример су имена датотека заглавља укључених у више других датотека. Вредност поља *numOfProcessedReports* одговара дужини ове листе.

numOfUniqueFilesAfectedByTest је укупан број различитих изворних датотека погођених тестом. Ту се убрајају само изворне датотеке чија је покривеност кода већа од 0%.

numOfUniqueFilesProcessed је укупан број различитих изворних датотека на које се наишло у току обраде информација о покривености кода. Ту се убрајају и изворне датотеке са покривеношћу кода од 0% и овај број одговара броју јединствених елемената листе *listOfProcessedFileNames*.

fileExtensions представља скуп екстензија датотека чија је покривеност кода већа од 0%.

extensionsCounter представља мапу која свакој екстензији додељује број појављивања међу таквим датотекама.



Слика 5.3: Дијаграм класе *MiniReport*

5.2 Покретање тестова, обрада и чување резултата

Централну логику алата имплементира класа *ProjectCodeCoverage*. У току рада алата инстанцира се по један објекат ове класе за сваки тест. Објекат ове класе је задужен за покретање теста а затим и за прикупљање и обраду свих информација о покривености кода насталих након покретања. Након обраде информације о покривености кода финални резултати се чувају на нивоу објекта помоћу структура података описаних у претходном поглављу. На тај начин један објекат класе *ProjectCodeCoverage* пружа информације о покривености кода једним тестом остатку алата. На слици 5.4 се може видети дијаграм ове класе.

Конструктор и поља класе *ProjectCodeCoverage*

Конструктору класе *ProjectCodeCoverage* се прослеђују следећи аргументи при креирању објеката:

projectDirectory представља ниску која означава путању до директоријума у којем је изграђен пројекат чије се информације о покривености кода прикупљају и обрађују. Неопходно је да то буде директоријум у којем

је пројекат изграђен задавањем одговарајућих опција за инструментализацију програма преводиоцу као и опција које сугеришу преводиоцу да генерише информације за дебаговање. Више о овим условима биће речи у поглављу 5.5 које говори о зависностима.

test представља ниску која означава назив теста који ће бити покренут.

command представља ниску која означава назив команде којом ће задати тест бити покренут.

commandArgs представља листу аргумената командне линије који ће бити задати команди за покретање теста.

coverageInfoDest представља ниску која означава путању до директоријума у којем ће привремено бити смештени међурезултати које објекти ове класе генеришу.

targetSourceFile представља ниску која, ако је задата, означава путању до изворне датотеке од интереса.

targetObjectPath представља ниску која, ако је задата, означава путању до објектне датотеке од интереса.

summaryReportsList представља листу објеката кла̂са које чувају збирне информације о покривености кода тестом. У оквиру алата је имплементирана једна таква класа и то је класа *MiniReport*. Посотјање овакве листе омогућава лако прослеђивање објеката других кла̂са овог типа које могу бити имплементиране у будућности.

Објекти класе *ProjectCodeCoverage* садрже јавна поља која одговарајау сваком од наведених аргумената конструктора. Додатно, сваки објекат садржи и јавна поља *reports* и *miniReport*.

Поље *reports* представља мапу која назив сваке изворне датотеке пресликава у одговарајући објекат типа *SFCoverageInformation*. Након покретања теста у мапу се смештају називи изворних датотека погођених тестом и одговарајући *SFCoverageInformation* објекти. На тај начин се чувају информације о покривености кода за сваку изворну датотеку и кроз ово јавно поље су доступне остатку алата.

Слика 5.4: Дијаграм класе *ProjectCodeCoverage*

Поље *miniReport* представља референцу на објекат класе *MiniReport*. Након покретања теста и обраде информација о покривености кода насталих том приликом, објекат ће садржати опште податке о утицају теста на кôд.

Методе класе *ProjectCodeCoverage*

Процес обраде и чувања информација о покривености кода који врше објекти класе *ProjectCodeCoverage* је подељен у три основна корака. То су уклањање информација о покривености насталих претходном употребом кода, покретање теста и претрага директоријума пројекта уз обраду информација о покривености насталих након покретања теста. Улазна метода у овај процес јесте метода *runProjectCodeCoverage()*. Метода се састоји од три подметоде које моделују сваки од три поменута корака. То су следеће методе:

- *clearProjectFromGcda()*
- *runTest()*
- *searchForGcda()*

У оквиру методе *clearProjectFromGcda()* је имплементирано уклањање информација о покривености насталих претходном употребом кода. Имплементација је реализована рекурзивним обиласком директоријума пројекта чија је путања задата кроз поље *projectDirectory* и брисањем свих датотека формата *gcda* на које се при обиласку наиђе. Овај корак је неопходан како се заостале информације о покривености кода не би мешале са новим информацијама насталим након покретања теста. Уколико би дошло до мешања, финални резултати би били погрешни јер би представљали збирне информације о покривености кода тестом и претходном употребом. Будући да се информације о покривености записују унутар датотека формата *gcda* њиховим брисањем у оквиру ове методе је осигурано да неће доћи до мешања информација.

Метода *runTest()* покреће задати тест задатом командом. У ту сврху користи се модул *Python subprocess* и функција *run()*. Функцији *run()* се прослеђује команда за покретање теста задата кроз поље *command* затим назив теста задат кроз поље *test* и листа аргумената командне линије задата кроз поље *commandArgs*. Функција *run()* покреће команду у дете-процесу и чека на завршетак извршавања теста.

Метода *searchForGcda()* је задужена за проналазак свих датотека формата *gcda* насталих након покретања теста и покретање алата *gcov*. Такође, ова метода парсира а потом и чува резултате о покривености кода тестом.

Проналазак датотека формата *gcda* је имплементиран рекурзивним обиласком директоријума задатим кроз поље *projectDirectory*. При обиласку директоријума прескачу се све датотеке чија екстензија није *.gcda*.

Након проналаска датотеке формата *gcda*, потребно је прочитати информације о покривености кода које су у њој записане. У ту сврху користи се алат *gcov*. Помоћна метода *runGcov()* покреће алат у дете-процесу уз додатне аргументе командне линије међу којима су најзначајнији:

- *--json-format* говори алату да ће прочитане информације о покривености кода бити смештене се у датотеку формата *JSON*

- *--branch-probabilities* говори алату да фреквенције извршавања за сваку грану буду присутне у информацијама о покривености

- *--demangled-names* говори алату да имена функција у информацијама о покривености буду записана на читљив начин

Уколико је извршавање алата *gcov* било успешно, формирана је архива у којој се налази датотека формата *JSON* са информацијама о покривености кода. Помоћна метода *readArchiveAndSaveJson()* је задужена за отварање архиве и читање информација о покривености кода. Информације о покривености кода ће бити смештене у мапу која одговара прочитаном садржају из датотеке формата *JSON*.

Следећи корак јесте парсирање информацијама о покривености из мапе и њихово чување у оквиру одговарајућег *SFCoverageInformation* објекта, унутар мапе *reports*. За овај корак одговорне су помоћне методе *parseJsonReport()* і *addReport()*.

Унутар мапе добијене у претходној методи налази се листа изворних датотека чије су информације о покривености кода прочитане из текуће датотеке формата *gcda*. Метода *parseJsonReport()* на почетку свог рада издваја такву листу из мапе. За сваку изворну датотеку присутну у листи се најпре инстанцира празан објекат класе *SFCoverageInformation*, затим се дохватају информације о покривености помоћу одговарајућих кључева и потом смештају у инстанцирани објекат класе *SFCoverageInformation*. Називи значајних кључева мапе су следећи:

file — помоћу овог кључа се дохвата назив изворне датотеке и смешта се у поље *name* објекта класе *SFCoverageInformation*.

lines — помоћу овог кључа издвајају се бројеви линија изворне датотеке за које постоје информације о броју извршавања и одговарајући број извршавања. Сви присутни бројеви линија додају се у скуп *linesOfInterest* објекта класе *SFCoverageInformation*. Такође, бројеви свих линија из листе се додају и у мапу *lineHitCount* тог објекта тако да се пресликавају у одговарајући број извршавања. Са друге стране, само бројеви линија чији је број извршавања већи од нуле, се додају у скуп *coveredLines* објекта класе *SFCoverageInformation*.

functions — помоћу овог кључа издвајају се имена функција изворне датотеке за које постоје информације о броју извршавања. Информације о именима функција се убацују у мапу *functionHitCount* објекта класе *SFCoverageInformation* тако да се пресликавају у одговарајући број извршавања. У скуп *coveredFunctions* објекта класе *SFCoverageInformation* убацују се само имена функција са бројем извршавања већим од нуле.

Овим поступком су прочитане информације за текућу изворну датотеку из листе и смештене у објекат класе *SFCoverageInformation*. Следећи корак је да се тако прочитане информације о покривености кода за изворну датотеку сместе у мапу *reports*.

Убацивање информација у мапу *reports* није тривијално из разлога што информације за једну изворну датотеку могу бити записане у више од једне датотеке формата *gcda*. Зато је при додавању нових информација у мапу *reports* потребно проверити да ли су у њој већ присутне неке информације за ту исту изворну датотеку.

Унутар методе *addReport()* је имплементирана логика за исправно додавање нових информација. Метода прима објекат *SFCoverageInformation* за текућу изворну датотеку генерисан у претходној методи. Затим се дохвата скуп кључева мапе *reports* и проверава се да ли је вредност поља *name* примљеног објекта класе *SFCoverageInformation*, присутна међу њима. Уколико то није случај, значи да се први пут наилази на ту изворну датотеку. Тада се у мапу додаје вредност поља *name* примљеног објекта као нови кључ, док се читав тај објекат додаје као вредност која одговара новом кључу.

Уколико је вредност поља *name* присутна међу кључевима мапе *reports* тада се дохвата стари *SFCoverageInformation* објекат који одговара том кључу, а затим се информације ажурирају на следећи начин:

- Вредности скупова *linesOfInterest*, *coveredLines* и *coveredFunctions* се ажурирају тако да представљају унију одговарајћих старих и нових скупова

- Вредности мâпа *lineHitCount* и *functionHitCount* се ажурирају тако да се кључеви који нису присутни у старим мапама, а јесу у новим, убаце у старе мапе заједно са својим одговарајућим вредностима, а уколико дође до поклапања старих и нових кључева тада се вредности које одговарају старим кључевима увећају за вредности које одговарају новим кључевима

Поштујући ова правила, на крају читавог рада објекта *ProjectCodeCoverage* мапа *reports* ће носити исправне информације о покривености кода тестом за сваку изворну датотеку.

За све изворне датотеке за које је листа покривених линија *coveredLines* празна прескаче се корак додавања у мапу *reports*. Тиме се знатно смањује финални број изворних датотека у мапи будући да су тестови обично писани

да погађају мањи део кода пројекта, па већина изворних датотека неће бити погођена покретањем теста. То за последицу има брже генерисање приказа у формату *html* у каснијим деловима алата, јер је потребно обрадити мању количину података из мапе.

Поред свих описаних корака, на крају рада методе *searchForGcda()* обрађује се и листа *summaryReportsList*. Листа садржи објекте који чувају збирне информације о утицају теста на кôд. Објекти се попуњавају неопходним информацијама које су прикупљене у току рада методе.

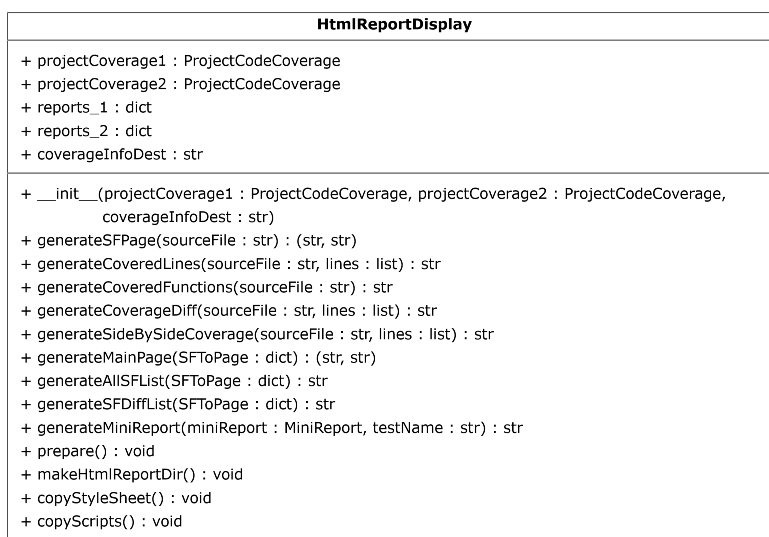
Класа *ProjectCodeCoverage* је имплементирана по обрасцу за пројектовање Посетилац (енг. *Visitor*) [17]. То значи да је одговорност генерисања неопходних збирних информација за објекте из листе *summaryReportsList* препуштена класи *ProjectCodeCoverage*. Таква структура омогућава лаку имплементацију нових класа које садрже другачије типове збирних информација и попуњавање њихових објеката подацима. Довољно је имплементирати нову класу, проследити објекат те класе кроз листу *summaryReportsList* и дефинисати методу за попуњавање објеката тог типа у оквиру класе *ProjectCodeCoverage*.

Унутар методе *visitMiniReport()* је имплементирано попуњавање објеката класе *MiniReport*. У току рекурзивног обиласка директоријума пројекта и обраде свих датотека формата *gcda* прикупљају се информације неопходне за попуњавање објекта класе *MiniReport*. То су информације о укупном броју обрађених датотека формата *gcda*, укупном броју обрађених извештаја и листа свих назива изворних датотека на које се наишло у току обраде. Укупном броју обрађених извештаја одговара укупан број изворних датотека издвојених из свих прочитаних датотека формата *JSON*, док листа свих назива изворних датотека обједињује управо њихове називе. На основу тих информација и финалних информација у мапи *reports* попуњава се објекат класе *MiniReport*.

5.3 Приказ информација о покривености кода

Након покретања тестова и прикупљања информација формира се финални приказ покривености кода тестовима. Приказ се формира помоћу страница у формату *html* и представља излаз алата *CovDiff*. Класа задужена за његово формирање је класа *HtmlReportDisplay*. Она користи информације о покривености кода сачуване у оквиру објеката класе *ProjectCodeCoverage* и

формира садржај страница у формату *html* са неколико различитих типова приказа. На слици 5.5 се може видети дијаграм ове класе.

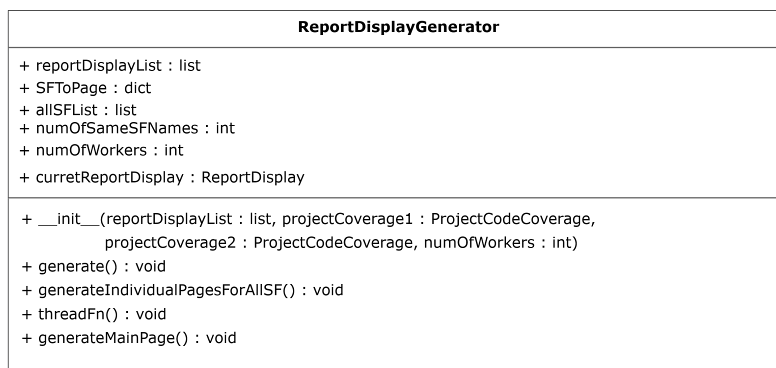


Слика 5.5: Дијаграм класе *HtmlReportDisplay*

За генерисање садржаја страница у формату *html* користи се пакет *Airium* [2]. Функционалности које пружа овај пакет су имплементиране у оквиру истоимене класе. Објекти класе *Airium* садрже методе за формирање елемената странице у формату *html*. Садржај странице је могуће једноставно генерисати због коресподенције између начина на који се користе објекти класе *Airium* и синтаксе програмског језика *Python*. Једноставност генерисања садржаја странице се огледа у томе што је могуће формирати њену структуру само помоћу назубљивања позива метода за формирање елемената, таквог да оно осликава хијерархију елемената у оквиру странице.

Још једна класа која учествује у процесу формирања финалног приказа информација о покривености кода јесте класа *ReportDisplayGenerator*. Она је одговорна за формирање датотека са приказима задатког формата и уписивање њиховог садржаја. Дијаграм ове класе је приказан на слици 5.6.

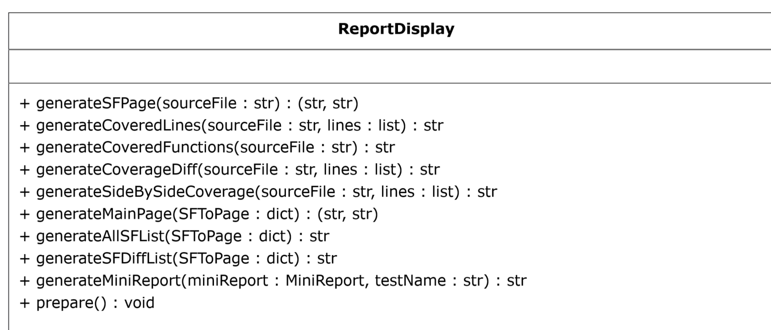
У оквиру алата *CovDiff* је имплементирано генерисање садржаја датотека само у формату *html*. За то је задужена претходно поменута класа *HtmlReportDisplay*. Међутим, класа *ReportDisplayGenerator* је имплементирана тако да подржава генерисање приказа информација о покривености кода у различитим форматима који могу бити имплементирани у будућности. Потребно је објекте кла̂са које имплементирају нове формате проследити кон-



Слика 5.6: Дијаграм класе *ReportDisplayGenerator*

структору класе *ReportDisplayGenerator* кроз листу задатих формата. Класа *ReportDisplayGenerator* затим формира приказе информација о покривености кода у свим задатим форматима.

Класе које имплементирају генерисање информација о покривености кода у новим форматима морају имати имплементиране одређене методе. Списак неопходних метода дат је у оквиру класе *ReportDisplay*. Класе које имплементирају нове формате приказа морају наследити ову класу и имплементирати све методе које су у њој присутне. На тај начин је обезбеђено да у току рада класе *ReportDisplayGenerator* сви формати приказа буду генерисани на исправан начин. Дијаграм класе *ReportDisplay* је приказан на слици 5.7.



Слика 5.7: Дијаграм класе *ReportDisplay*

У оквиру методе *generateIndividualPagesForAllSF()* је имплементиран процес генерисања страница посвећених свакој изворној датотеци појединачно који обавља класа *ReportDisplayGenerator*. Генерисање страница се врши у паралели. У ту сврху користи се инстанца класе *Pool* из модула *Python*

multiprocessing.

Пре свега, формира се скуп свих изворних датотека као унија скупа кључева мапе *reports* првог и скупа кључева мапе *reports* другог објекта класе *ProjectCodeCoverage*. Помоћу инстанце класе *Pool* генерише се скуп радника (енг. *workers*). Радници су деца-процеси главног процеса алата. Сваком раднику додељује се подскуп скупа свих изворних датотека. За додељени подскуп изворних датотека радник генерише странице у задатом формату.

Радници се извршавају конкурентно, а сваки радник свој подскуп изворних датотека обрађује секвенцијално. Када неки од радника заврши генерисање страница свог подскупа изворних датотека, додељује му се нови подскуп. Тај поступак се понавља докле год не буду генерисане странице у задатом формату за сваку изворну датотеку из полазног скупа. При генерисању страница радници попуњавају и мапу *SFToPage* која имена изворних датотека слика у путању до странице која јој одговара.

У наставку следи опис приказа информација о покривености кода у формату *html* који генерише класа *HtmlReportDisplay*. Дат је опис конструктора класе и основних метода. Након тога биће описани различити типови приказа које ова класа генерише.

Конструктор класе *HtmlReportDisplay*

Конструктор класе *HtmlReportDisplay* прима две референце на објекте класе *ProjectCodeCoverage* — за сваки тест по један објекат. У објектима класе *ProjectCodeCoverage* чувају се информације о покривености кода тестом у оквиру мапе *reports*. Референце служе да се приступи мапама, прочитају подаци о покривености и помоћу њих формирају различити прикази. Поред ових референци, констурктор класе *HtmlReportDisplay* прима и путању до директоријума у којем ће бити смештене све генерисане странице у формату *html*.

Основне методе класе *HtmlReportDisplay*

Класа *HtmlReportDisplay* наслеђује класу *ReportDisplay* и имплементира све методе дефинисане у тој наткласи. Процес генерисања садржаја страница у формату *html* који врши класа *HtmlReportDisplay*, састоји се из две

главне наслеђене методе. Оне имплементирају генерисање два основна типа страница. То су подметоде:

- *generateMainPage()*
- *generateSFPage()*

Метода *generateMainPage()* имплементира генерисање садржаја почетне странице док метода *generateSFPage()* имплементира генерисање садржаја страница за појединачне изворне датотеке. У те сврхе обе методе користе пакет *Airium*. Обе методе имају уређен пар две ниске као своју повратну вредност. Прва ниска представља путању на којој страница треба бити формирана док друга ниска представља саджај странице у формату *html*.

На почетку сваке странице за појединачне изворне датотеке у формату *html* налази се назив датотеке са изворним кодом. Остатак садржаја странице чине:

- Збирни приказ покривених линија
- Извештај о покривености функција
- Приказ разлика у покривеним линијама
- Упоредни приказ покривених линија

У наставку следе описи сваког од наведених приказа као и опис садржаја почетне странице.

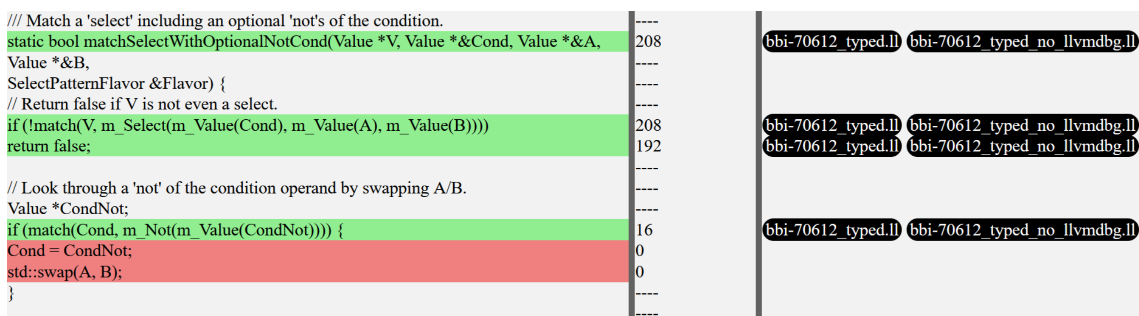
Збирни приказ покривених линија

Збирни приказ покривених линија (енг. *Summary Report*) представља унију линија покривених са оба теста. На основу овог приказа могу се уочити линије покривене само једним од тестова. Управо је то један од начина да се уоче разлике у покривености кода тестовима и то на нивоу једне изворне датотеке. Поред тога, јасно се виде и линије које нису покривене ни са једним од тестова, што може бити корисна сугестија за писање нових тестова. Приказ је представљен табелом са следећим колонама:

- *Line number* — представља редни број линије у изворној датотеци
- *Line* — представља линију изворног кода

- *Number of hits* — представља број извршавања линије
- *Tests that cover line* — представља листу имена тестова који покривају линију

На слици 5.8 се може видети како изгледа овај приказ. Сваки ред табеле се односи на једну линију изворног кода датотеке. Конкретан текстуални садржај линије је приказан у колони *Line*.



Слика 5.8: Збирни приказ покривених линија

Текстуални садржај може бити обојен зеленом бојом у случају да је линија покривена барем једним од два теста, црвеном бојом уколико линија није покривена ни са једним тестом или може бити необојен уколико линија није од интереса. Линије обојене неком од боја представљају линије од интереса тј. линије изворне датотеке на којима се налази наредба.

Колона *Number of hits* може садржати вредност „0” уколико линија није погођена ни једним тестом, вредност већу од нуле уколико је линија погођена макар једним тестом или вредност „----” уколико је у питању линија која није од интереса.

Помоћна метода одговорна за формирање збирног приказа покривених линија је метода *generateCoveredLines()*. На почетку се у методи из обе мапе *reports* објеката *ProjectCodeCoverage* дохватају објекти *SFCoverageInformation*, на основу имена изворне датотеке. Уколико је редни број линије члан неког од скупова *linesOfInterest* објеката *SFCoverageInformation*, тада је у питању линија од интереса. Број извршавања рачуна се као збир вредности у које се слика редни број линије унутар мâпа *lineHitCount* објеката *SFCoverageInformation*. Листа тестова који покривају линију се одређује на основу тога да ли редни број линије припада неком од скупова *coveredLines* објеката *SFCoverageInformation*.

Извештај о покривености функција

Извештај о покривености функција (енг. *Functions Report*) представља списак функција које су погођене неким од тестова. Извештај је представљен табелом са следећим колонама:

- *Function name* — представља име функције
- *Number of hits* — представља збирни број извршавања функције

У табели се налазе само функције погођене неким од тестова. Колона *Number of hits* представља збирни број погодака функције у оба теста.

Помоћна метода која имплементира формирање извештаја о покривености функција је метода *generateCoveredFunctions().* На почетку рада методе дохватају се одговарајући *SFCoverageInformation* објекти из *reports* мâпа на исти начин као при формирању претходног приказа. Од интереса су мапе *functionHitCount* унутар *SFCoverageInformation* објеката. У табели се налазе имена функција која представљају кључеве унутар обе мапе *functionHitCount.* Бројеви извршавања функција су вредности из *functionHitCount* мâпа које одговарају тим кључевима. Уколико се у мапама налазе исти кључеви вредности се сабирају па се добија збирни број извршавања функција.

Приказ разлика у покривеним линијама

Приказ разлика у покривеним линијама (енг. *Coverage Diff*) садржи приказ линија које су покривене првим, а нису покривене другим тестом и обрнуто, приказ линија које су покривене другим, а нису покривене првим тестом. Овим приказом јасно су означене оне линије изворног кода које представљају разлику у покривености кода тестовима на нивоу једне изворне датотеке. Приказ је представљен табелом са следећим колонама:

- *Line number* — представља редни број линије у изворној датотеци
- *test1_name BUT NOT test2_name* — представља линију изворног кода
- *Line number* — представља редни број линије у изворној датотеци
- *test2_name BUT NOT test1_name* — представља линију изворног кода

// If we're not strictly identical, we still might be a commutable instruction	342	// If we're not strictly identical, we still might be a commutable instruction
if (BinaryOperator *LHSBinOp = dyn_cast(LHSI)) {	343	if (BinaryOperator *LHSBinOp = dyn_cast(LHSI)) {
if (!LHSBinOp->isCommutative())	344	if (!LHSBinOp->isCommutative())
return false;	345	return false;
	346	
assert(isa(RHSI) &&	347	assert(isa(RHSI) &&
"same opcode, but different instruction type?");	348	"same opcode, but different instruction type?");
BinaryOperator *RHSBinOp = cast(RHSI);	349	BinaryOperator *RHSBinOp = cast(RHSI);
	350	
// Commuted equality	351	// Commuted equality
return LHSBinOp->getOperand(0) == RHSBinOp->getOperand(1) &&	352	return LHSBinOp->getOperand(0) == RHSBinOp->getOperand(1) &&
LHSBinOp->getOperand(1) == RHSBinOp->getOperand(0);	353	LHSBinOp->getOperand(1) == RHSBinOp->getOperand(0);
}	354	}
if (CmpInst *LHSCmp = dyn_cast(LHSI)) {	355	if (CmpInst *LHSCmp = dyn_cast(LHSI)) {
assert(isa(RHSI) &&	356	assert(isa(RHSI) &&
"same opcode, but different instruction type?");	357	"same opcode, but different instruction type?");
CmpInst *RHSCmp = cast(RHSI);	358	CmpInst *RHSCmp = cast(RHSI);
// Commuted equality	359	// Commuted equality
return LHSCmp->getOperand(0) == RHSCmp->getOperand(1) &&	360	return LHSCmp->getOperand(0) == RHSCmp->getOperand(1) &&
LHSCmp->getOperand(1) == RHSCmp->getOperand(0) &&	361	LHSCmp->getOperand(1) == RHSCmp->getOperand(0) &&
LHSCmp->getSwappedPredicate() == RHSCmp->getPredicate();	362	LHSCmp->getSwappedPredicate() == RHSCmp->getPredicate();
}	363	}

Слика 5.9: Приказ разлика у покривеним линијама

На слици 5.9 се може видети како приказ изгледа. Сваки ред табеле се односи на једну линију изворног кода датотеке. Конкретан текстуални садржај линије је приказан у другој и четвртој колони.

Имена друге и четврте колоне сугеришу да ће текстуални садржај који се у њима налази бити обојен одговарајућуом бојом уколико је линија покривена само једним од тестова. Па су тако линије из друге колоне обојене жутом бојом уколико су покривене првим, а нису покривене другим тестом, док су линије из четврте колоне обојене наранџастом бојом уколико су покривене другим, али не и првим тестом. Све остале линије неће бити обојене ни једном бојом.

Помоћна метода која је одговорна за генерисање приказа разлика у покривеним линијама је метода *generateCoverageDiff()*. Као и у претходним приказима, на почетку методе издвајају се *SFCoverageInformation* објекти из *reports* мâпа који одговарају имену изворне датотеке. Након тога, формира се разлика скупа *coveredLines* првог *SFCoverageInformation* објекта у односу на скуп *coveredLines* другог *SFCoverageInformation* објекта а затим се формира и разлика скупа *coveredLines* другог у односу на скуп *coveredLines* првог *SFCoverageInformation* објекта. На тај начин се формирају скупови линија које су покривене само првим односно само другим тестом.

Упоредни приказ покривених линија

Упоредни приказ покривених линија (енг. *Side By Side Comparison*) представља приказ линија покривених првим тестом и до њих постављен приказ

линија покривених другим тестом. Помоћу овог приказа могуће је визуално поредити покривеност кода једним и другим тестом на нивоу једне изворне датотеке и уз то поредити и бројеве извршавања сваке од линија. Извештај је представљен табелом са следећим колонама:

- *Line number* — представља редни број линије у изворној датотеци
- *test1_name* — представља линију изворног кода
- *Hit count* — представља број извршавања линије при покретању првог теста
- *Line number* — представља редни број линије у изворној датотеци
- *test2_name* — представља линију изворног кода
- *Hit count* — представља број извршавања линије при покретању другог теста

На слици 5.10 се може видети како изгледа овај приказ. Сваки ред табеле се односи на једну линију изворног кода датотеке. Текстуални садржај линије изворног кода приказан је у другој и петој колони. Те две колоне носе, редом, називе првог и другог теста што сугерише да ће у њима бити означене све линије покривене првим односно другим тестом.

Линија из друге колоне се боји зеленом бојом уколико је покривена првим тестом, док се линија из пете колоне боји зеленом бојом уколико је покривена другим тестом. Уколико је линија покривена тестом, одговарајућа *Hit count* колона ће садржати број извршавања линије при покретању тог теста. За линије које нису покривене тестом одговарајућа *Hit count* колона ће садржати вредност „0” уколико су у питању линије од интереса или вредност „---” ако линије нису од интереса.

Помоћна метода која имплементира генерисање упоредног приказа покривених линија је метода *generateSideBySideCoverage()*. Као и у претходним случајевима, на почетку рада методе дохватају се *SFCoverageInformation* објекти. Користе се скупови *coveredLines* издвојених објеката да се одреди да ли је линија покривена и којим тестом. Помоћу одговарајућих мâпа *lineHitCount* издвајају се бројеви извршавања линија за сваки од тестова.

20	DILocation *DebugLoc::get() const {	84	20	DILocation *DebugLoc::get() const {	64
21	return cast_or_null(Loc.get());	84	21	return cast_or_null(Loc.get());	64
22	}	---	22	}	---
23		---	23		---
24	unsigned DebugLoc::getLine() const {	2	24	unsigned DebugLoc::getLine() const {	1
25	assert(get() && "Expected valid DebugLoc");	2	25	assert(get() && "Expected valid DebugLoc");	1
26	return get()->getLine();	2	26	return get()->getLine();	1
27	}	---	27	}	---
28		---	28		---
29	unsigned DebugLoc::getCol() const {	2	29	unsigned DebugLoc::getCol() const {	1
30	assert(get() && "Expected valid DebugLoc");	2	30	assert(get() && "Expected valid DebugLoc");	1
31	return get()->getColumn();	2	31	return get()->getColumn();	1
32	}	---	32	}	---
33		---	33		---
34	MDNode *DebugLoc::getScope() const {	3	34	MDNode *DebugLoc::getScope() const {	1
35	assert(get() && "Expected valid DebugLoc");	3	35	assert(get() && "Expected valid DebugLoc");	1
36	return get()->getScope();	3	36	return get()->getScope();	1
37	}	---	37	}	---
38		---	38		---
39	DILocation *DebugLoc::getInlinedAt() const {	1	39	DILocation *DebugLoc::getInlinedAt() const {	0
40	assert(get() && "Expected valid DebugLoc");	1	40	assert(get() && "Expected valid DebugLoc");	0
41	return get()->getInlinedAt();	1	41	return get()->getInlinedAt();	0
42	}	---	42	}	---

Слика 5.10: Упоредни приказ покривених линија

Почетна страница

Након генерисања страница за појединачне изворне датотеке, генерише се и страница у формату *html* која представља почетну страну. Приказ почетне стране се може видети на слици 5.11. Метода одговорна за њено генерисање је метода *generateMainPage()*.

Почетна страна садржи везе ка свим страницама за појединачне изворне датотеке и преглед резултата које алат генерише треба да почне од ње. У оквиру почетне странице налази се неколико приказа. То су:

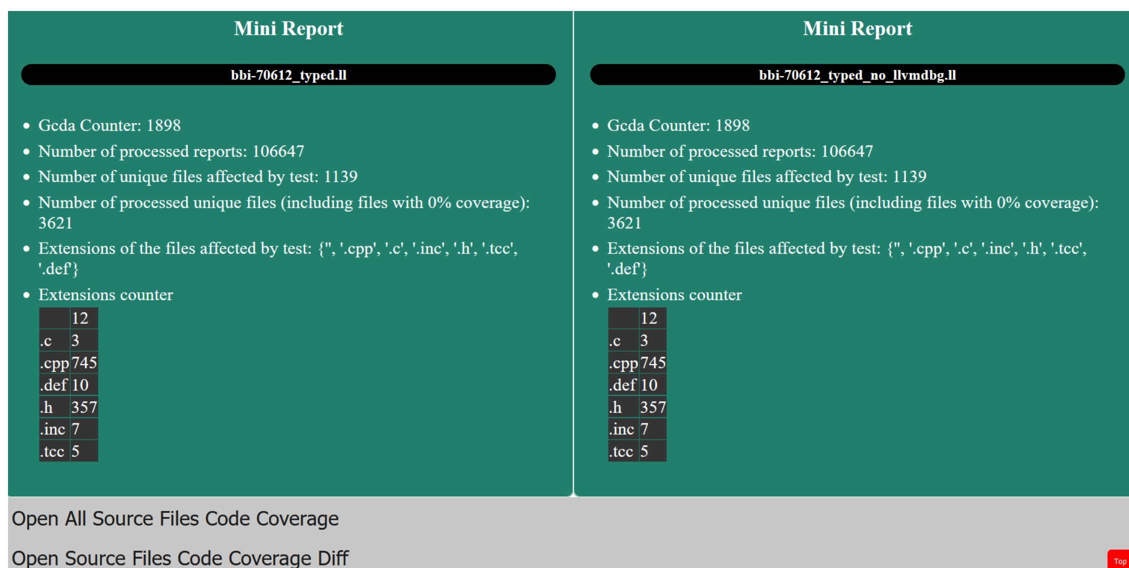
- прикази садржаја објеката *MiniReport* за оба теста
- листа свих изворних датотека погођених неким од тестова са везама ка одговарајућим страницма
- листа свих изворних датотека у којима постоје разлике у покривености кода тестовима

Прикази садржаја објеката *MiniReport* за оба теста налазе се на врху странице. Приказана су сва поља која објекти класе *MiniReport* садрже. Помоћу овог приказа се за оба теста може стећи први утисак о њиховом утицају на кôд.

Приказ листе свих изворних датотека погођених неким од тестова је формиран помоћу табеле са следећим колонама:

- *Source file absolute path* — колона садржи називе изворних датотека задате као апсолутне путање

- *test1_name* — колона садржи информације о утицају првог теста на сваку изворну датотеку
- *test2_name* — аналогно претходној али за други тест



Слика 5.11: Почетна страна

Редови табеле су све изворне датотеке погођене неким од тестова. Скуп свих изворних датотека погођених неким од тестова прави се као унија кључева *reports* мâпа *ProjectCodeCoverage* објеката.

Сваки ред табеле се односи на једну изворну датотеку. Називи изворних датотека представљају везе ка одговарајућим страницама у формату *html* на које је могуће кликнути. Кликом на њих, у новом језичку претраживача, отвара се страница изворне датотеке.

Друга и трећа колона носе називе, редом, првог и другог теста. За сваку изворну датотеку у табели, у оквиру ових колона, приказан је проценат покривености кода оговарајућим тестом.

Проценат покривености кода тестом се рачуна као количник броја елемената скупа *coveredLines* и броја елемената скупа *linesOfInterest* помножен са 100. Скупови *coveredLines* и *linesOfInterest* су прочитани из објеката класе *SFCoverageInformation* који одговарају изворној датотеци за први и други тест. Како су у скуповима *linesOfInterest* изостављене све линије које не садрже наредбу, проценат покривености кода је израчунат на исправан начин.

Погрешно би било рачунати проценат покривености кода у односу на апсолутни број линија у изворној датотеци, јер су тим поступком обухваћене и линије које не садрже наредбу какве су нпр. линије са коментарима или празне линије.

Поред процента покривености и у другој и у трећој колони се могу наћи ознаке *diff*. Уколико се ознака *diff* нађе у другој колони то значи да постоје линије изворног кода датотеке које покрива први тест, а не покрива други, а уколико се ознака *diff* нађе у трећој колони то значи да постоје линије које покрива други тест, а не покрива први. Могуће је да се ознака *diff* нађе у обе колоне.

Уз које изворне датотеке и у којим колонама ће се наћи ознака *diff* одлучује се помоћу истих разлика скупова покривених линија које су рачунате и при генерисању приказа разлика у покривеним линијама. Уколико постоје елементи у некој од две израчунатате разлике скупова покривених линија то је знак да треба ставити *diff* ознаку у колони одговарајућег теста.

Листа свих изворних датотека у којима постоје разлике у покривености кода тестовима је генерисана издвајањем оних изворних датотека из претходне листе које у некој од колона садрже ознаку *diff*. На тај начин је олакшан преглед оних изворних датотека у којима постоје разлике у покривености кода тестовима. На слици 5.12 је у горњем делу приказана листа свих изворних датотека док је у доњем делу приказана листа само оних са ознаком *diff*.

Source file absolute path	bbi-70612_typed.ll	bbi-70612_typed_no_llvmdbg.ll
/home/nikola/Desktop/llvm-project/llvm/lib/Target/WebAssembly/WebAssemblyDebugFixup.cpp	2.273 %	2.273 %
/home/nikola/Desktop/llvm-project/llvm/include/llvm/DebugInfo/CodeView/TypeHashing.h	80.000 %	80.000 %
/home/nikola/Desktop/llvm-project/llvm/lib/Analysis/ModuleDebugInfoPrinter.cpp	1.351 %	1.351 %
/home/nikola/Desktop/llvm-project/llvm/lib/Support/Debug.cpp	46.296 %	46.296 %
/home/nikola/Desktop/llvm-project/llvm/lib/IR/DebugLoc.cpp	40.000 % diff	35.000 %
/home/nikola/Desktop/llvm-project/llvm/lib/Support/DebugCounter.cpp	19.737 %	19.737 %

Source file absolute path	bbi-70612_typed.ll	bbi-70612_typed_no_llvmdbg.ll
/home/nikola/Desktop/llvm-project/llvm/include/llvm/Analysis/TargetTransformInfoImpl.h	29.167 % diff	26.268 %
/home/nikola/Desktop/llvm-project/llvm/include/llvm/IR/InstrTypes.h	48.322 % diff	47.651 %
/home/nikola/Desktop/llvm-project/llvm/lib/Analysis/AliasAnalysis.cpp	22.476 % diff	21.333 %
/home/nikola/Desktop/llvm-project/llvm/lib/Transforms/Scalar/MemCpyOptimizer.cpp	9.909 % diff	9.648 %
/home/nikola/Desktop/llvm-project/llvm/lib/Transforms/Scalar/CorrelatedValuePropagation.cpp	25.776 % diff	22.671 %
/home/nikola/Desktop/llvm-project/llvm/lib/Transforms/IPO/CalledValuePropagation.cpp	60.317 % diff	59.259 %

Слика 5.12: Листе изворних датотека на почетној страни

За сваку листу је имплементиран и механизам претраге изворних датотека

које се у њима налазе. Претрага се врши по називу изворне датотеке. На тај начин је олакшан преглед резултата за случај да тестови погађају велики број изворних датотека.

5.4 Аргументи командне линије

При покретању алата из командне линије потребно је задати одређене аргументе. Обавезни аргументи командне линије су следећи:

directory_path представља путању до директоријума у којем је изграђен пројекат.

test1 представља назив првог теста.

test2 представља назив другог теста.

coverage_dest представља путању до директоријума у којем ће бити смештени резултати.

command [command_arg1 [command_arg2 ...]] представља команду за покретање тестова. Команда за покретање теста може бити било која наредба која се може извршити у дете-процесу помоћу интерфејса који нуди модул *Python subprocess*. Након команде за покретање тестова се може наћи нула или више аргумената који ће бити прослеђени команди. Аргументи се задају без минуса испред. Команда се помоћу *subprocess* модула покреће у формату

command test1 -command_arg1 -command_arg2 ...

односно

command test2 -command_arg1 -command_arg2 ...

Уколико су тестови сами по себи извршне датотеке, као команду је могуће задати „./”. То ће сугерисати алату да директно покрене тестове.

Опциони аргументи командне линије су следећи:

--h, --help је опција којом се штампа упутство за коришћење алата са описима сваког од аргумената.

--report-formats је опција којом се задаје листа формата извештаја о покривености кода које је потребно генерисати. Приликом навођења, формати се раздвајају зарезом. Уколико се не наведе ова опција, подразумевано се генерише приказ у формату *html*. Алат подржава генерисање приказа информација о покривености кода само у формату *html* па се ова опција може користити након имплементације нових формата.

--summary-reports је опција којом се задаје листа сумарних извештаја које је потребно генерисати. Приликом навођења, сумарни извештаји се раздвајају зарезом. Уколико се не наведе ова опција, подразумевано се генерише сумарни извештај *MiniReport*. Алат подржава генерисање само сумарног извештаја *MiniReport* па се ова опција може користити након имплементације нових сумарних извештаја.

5.5 Зависности

Претпоставка за коришћење алата је да је пројекат изграђен задавањем одговарајућих опција које преводиоцу сугеришу да инструментализује кôд за анализу покривености. Уколико се пројекат преводи помоћу преводиоца *GCC* или *Clang* онда су то опције *-fprofile-arcs* и *-ftest-coverage*. Потребно је задати и опције преводиоцу за генерисање информација за дебаговање. За случај два поменута преводиоца то је опција *-g*. Потребно је још инсталирати и пакет *Airium* будући да се тај пакет користи за генерисање садржаја страница у формату *html*. Пакет је могуће инсталирати командом:

$ pip install airium

5.6 Тестирање рада алата

Рад алата *CovDiff* је током развоја тестиран на различитим програмима. У наставку ће бити приказано тестирање алата над једним једноставним пројектом који садржи основне аритметичке функције и тестирање алата над пројектом *LLVM*. Детаљни опис корака тестирања рада алата *CovDiff* покретањем над два поменута пројекта је јавно доступан. Јавно су доступни и

изворни кодови тестова, као и комплетни резулати добијени након оба тестирања. [2]

Тестирање над једноставним пројектом

Рад алата *CovDiff* је тестиран покретањем над једноставним пројектом. Пројекат се састоји од само једне изворне датотеке. Унутар ње написано је неколико једноставних функција у програмском језику *C*. То су функције које као своју повратну вредност имају збир, разлику, производ и максимум два прослеђена реална броја.

Написана су два тест примера који ће да тестирају рад тих функција. Први тест пример два пута позива функцију за сабирање, једном позива функцију за одузимање и једном позива функцију за проналажење максимума два реална броја. Други тест пример два пута позива функцију за сабирање и једном позива функцију за множење два реална броја.

Разлике у информацијама о покривености кода пројекта за таква два теста које је алат *CovDiff* генерисао састоје се из:

- линија функција за сабирање и проналажење максимума, као разлике које иду у корист првог теста
- линија функције за множење као разлике које иду у корист другог теста.

Такав резултат је и очекиван будући да се у првом тесту не позива функција за множење док се у другом тесту не позивају функције за одузимање и проналажење максимума. На слици 5.13 се може видети овај приказ разлика у покривености кода, док се на слици 5.14 може видети упоредни приказ покривености кода помоћу та два теста.

Тестирање над компилаторском инфраструктуром *LLVM*

Рад алата *CovDiff* је тестиран покретањем над компилаторском инфраструктуром *LLVM*. Циљ таквог покретања алата био је да се провери стабилност рада при обради велике количине информација о покривености кода

[2] Изворни кодови тестова, опис корака и резултати тестирања рада алата *CovDiff* су јавно доступни на `https://github.com/backspacer303/CovDiff/tree/main/Examples`.

Line Number	../CodeCoverage/Test/lib_tests/test1 BUT NOT ../CodeCoverage/Test/lib_tests/test2	Line Number	../CodeCoverage/Test/lib_tests/test2 BUT NOT ../CodeCoverage/Test/lib_tests/test1
1	#include "arithmetic.h"	1	#include "arithmetic.h"
2		2	
3	double sub(double a, double b){	3	double sub(double a, double b){
4	return a-b;	4	return a-b;
5	}	5	}
6	double mul(double a, double b){	6	double mul(double a, double b){
7	return a*b;	7	return a*b;
8	}	8	}
9		9	
10	double add(double a, double b){	10	double add(double a, double b){
11	return a+b;	11	return a+b;
12	}	12	}
13		13	
14	double max(double a, double b){	14	double max(double a, double b){
15	if(a>=b)	15	if(a>=b)
16	return a;	16	return a;
17	else	17	else
18	return b;	18	return b;
19	}	19	}
20		20	

Слика 5.13: Приказ разлика у покривености кода датотеке са аритметичким операцијама

Line Number	../CodeCoverage/Test/lib_tests/test1	Hit count	Line Number	../CodeCoverage/Test/lib_tests/test2	Hit count
1	#include "arithmetic.h"	---	1	#include "arithmetic.h"	---
2		---	2		---
3	double sub(double a, double b){	1	3	double sub(double a, double b){	0
4	return a-b;	1	4	return a-b;	0
5	}	---	5	}	---
6	double mul(double a, double b){	0	6	double mul(double a, double b){	1
7	return a*b;	0	7	return a*b;	1
8	}	---	8	}	---
9		---	9		---
10	double add(double a, double b){	2	10	double add(double a, double b){	2
11	return a+b;	2	11	return a+b;	2
12	}	---	12	}	---
13		---	13		---
14	double max(double a, double b){	1	14	double max(double a, double b){	0
15	if(a>=b)	1	15	if(a>=b)	0
16	return a;	0	16	return a;	0
17	else	---	17	else	---
18	return b;	1	18	return b;	0
19	}	---	19	}	---
20		---	20		---

Слика 5.14: Упоредни приказ покривености кода датотеке са аритметичким операцијама

пројекта *LLVM* и при генерисању велике количине страница са приказима у формату *html* за изворне датотеке пројекта *LLVM*. Пред тога, циљ је био и симулирати ситуацију у којој алат може да помогне при откривању грешака у изворном коду пројекта.

Пре покретања потребно је превести пројекат *LLVM* уз задавање опција

за инструментализацију његовог кода. Како је при превођењу пројекта *LLVM* коршћен програмски преводилац *GCC* потребно је задати опције *-fprofile-arcs* и *-ftest-coverage*. Додатно, пројекат *LLVM* је преведен уз задавање опције за генерисање информација за дебаговање. Детаљно упутство за превођење пројекта *LLVM* могуће је наћи на [10]. Након превођења пројекта *LLVM* на овај начин, формиране су датотеке формата *gcno* за сваку преведену изворну датотеку пројекта.

Тест примери су формирани превођењем једноставне функције написане у програмском језику *C* до међурепрезентације *LLVM IR* помоћу алата *Clang*. Превођење до међурепрезентације се постиже задавањем опције *-emit-llvm* алату *Clang*. Први тест пример формиран је тако што је додатно задата и опција *-g* која сугерише алату *Clang* да генерише информације за дебаговање. При формирању другог тест примера ова опција је изостављена. На овај начин формирана су два тест примера која се разликују само по присуству информација за дебаговање.

Такви тест примери формирани су са циљем да се у разликама у покривености кода пројекта *LLVM* нађу делови пројекта задужени за обраду информација за дебаговање. Додатно, унутар пролаза *RemoveRedundantDebugValues* [36] који је део пројекта *LLVM*, убачен је део кода који ће уклонити једну инструкцију из првог основног блока функције коју обрађује. Овај пролаз је задужен за елиминацију редундантних информација за дебаговање унутар функције. Како пролаз обрађује само информације за дебаговање не сме се десити да промени било који други део кода функције над којом оперише. Из тог разлога убачени део кода представља грешку у изворном коду пројекта *LLVM* јер нарушава ово правило. На слици 5.15 се може видети убачен део кода у пролаз *RemoveRedundantDebugValues* обојен зеленом бојом.

Алат *CovDiff* покренут је над пројектом *LLVM* који садржи описану грешку у обради информација за дебаговање. Алату су прослеђена два формирана теста која се разликују само по присуству информација за дебаговање. Као команда за покретање тестова задат је алат *llc* [27] који ће међурепрезентације тестова превести до асемблерског кода.

При овом покретању алата *CovDiff* обрађено је 1293 датотека формата *gcda* са информацијама о покривености кода пројекта *LLVM* првим тестом и 1247 датотека формата *gcda* са таквим информацијама за други тест. Формирано је 1293 страница са приказима у формату *html* за изворне датотеке пројекта

```
diff --git a/llvm/lib/CodeGen/RemoveRedundantDebugValues.cpp b/llvm/lib/CodeGen/RemoveRedundantDebugValues.cpp
index feb31e59f5fd..9c03d3035009 100644
--- a/llvm/lib/CodeGen/RemoveRedundantDebugValues.cpp
+++ b/llvm/lib/CodeGen/RemoveRedundantDebugValues.cpp
@@ -204,6 +204,11 @@ bool RemoveRedundantDebugValues::reduceDbgValues(MachineFunction &MF) {

   bool Changed = false;

+  // Error: Processing debug information should not affect the generated code!
+  MachineBasicBlock &MBB = *MF.begin();
+  MachineInstr &MI = *MBB.begin();
+  MI.eraseFromParent();
+
   for (auto &MBB : MF) {
     Changed |= reduceDbgValsBackwardScan(MBB);
     Changed |= reduceDbgValsForwardScan(MBB);
(END)
```

Слика 5.15: Кôд убачен у пролаз *RemoveRedundantDebugValues*

LLVM. Овим покретањем се показује да алат *CovDiff* ради стабилно при обради и генерисању велике количине података.

На почетној страници у формату *html*, у оквиру листе изворних датотека у којима постоје разлике у покривености кода пројекта *LLVM*, могу се уочити оне датотеке које садрже кôд за обраду информација за дебаговање. То је последица тога што је један тест садржао информације за дебаговање док други није. У оквиру те листе налази се и назив датотеке са изворним кодом пролаза *RemoveRedundantDebugValues* у којем постоји грешка. Кликом на назив отвара се одговарајућа страница са приказима у формату *html* за изворну датотеку пролаза. Корисник алата *CovDiff* на тој страници може уочити део изворног кода пројекта *LLVM* у којем постоји грешка из разлога што ће управо тај део кода бити означен као разлика у покривености кода пројекта првим тестом у односу на други тест. Тај приказ је могуће видети на слици 5.16.

```
bool RemoveRedundantDebugValues::reduceDbgValues(MachineFunction &MF) {
 LLVM_DEBUG(dbgs() << "\nDebug Value Reduction\n");

 bool Changed = false;

/ Error: Processing debug information should not affect the generated code!
 MachineBasicBlock &MBB = *MF.begin();
 MachineInstr &MI = *MBB.begin();
 MI.eraseFromParent();

 for (auto &MBB : MF) {
  Changed |= reduceDbgValsBackwardScan(MBB);
  Changed |= reduceDbgValsForwardScan(MBB);
}

 return Changed;
}

bool RemoveRedundantDebugValues::runOnMachineFunction(MachineFunction &MF) {
// Skip functions without debugging information.
if (!MF.getFunction().getSubprogram())
return false;

// Skip functions from NoDebug compilation units.
 if (MF.getFunction().getSubprogram()->getUnit()->getEmissionKind() ==
DICompileUnit::NoDebug)
return false;

 bool Changed = reduceDbgValues(MF);
 return Changed;
}
```

```
202 bool RemoveRedundantDebugValues::reduceDbgValues(MachineFunction &MF) {
203 LLVM_DEBUG(dbgs() << "\nDebug Value Reduction\n");
204
205 bool Changed = false;
206
207 // Error: Processing debug information should not affect the generated code!
208 MachineBasicBlock &MBB = *MF.begin();
209 MachineInstr &MI = *MBB.begin();
210 MI.eraseFromParent();
211
212 for (auto &MBB : MF) {
213 Changed |= reduceDbgValsBackwardScan(MBB);
214 Changed |= reduceDbgValsForwardScan(MBB);
215 }
216
217 return Changed;
218 }
219
220 bool RemoveRedundantDebugValues::runOnMachineFunction(MachineFunction &MF) {
221 // Skip functions without debugging information.
222 if (!MF.getFunction().getSubprogram())
223    return false;
224
225 // Skip functions from NoDebug compilation units.
226 if (MF.getFunction().getSubprogram()->getUnit()->getEmissionKind() ==
227 DICompileUnit::NoDebug)
228 return false;
229
230 bool Changed = reduceDbgValues(MF);
231 return Changed;
232 }
```

Слика 5.16: Грешка у пролазу *RemoveRedundantDebugValues* присутна у приказу разлика у покривености кода тог пролаза

Глава 6

Закључак

Овим радом су обухваћене основне идеје и мотиви за тестирања софтвера. Представљене су различите врсте и стратегије тестирања. Посебно је описано тестирање путања као важан облик тестирања које спроводе програмери. Даље је представљена покривеност кода као често коришћена мера квалитета тестова. У склопу тога обухваћени су и механизми и алати за мерење покривености кода.

У раду је представљен алат *CovDiff* и приказана је његова имплементација. Алат генерише и приказује разлике у информацијама о покривености кода након независног покретања два теста. Тиме се кориснику пружа нови поглед на информације о покривености кода.

Алат *CovDiff* представља унапређење алата *gcov* и превазилази његова ограничења у генерисању информација о покривености кода при узастопном покретању више тестова. Побољшање се састоји и у томе што алат *CovDiff* даје упоредне информације о покривености за два теста наглашавајући разлике.

Алат *CovDiff* преузима и добре особине алата *lcov*. Генерише приказе информација о покривености кода у формату *html*. То омогућава лак пренос резултата имајући у виду да је формат *html* лако читљив на свим системима. Задржана је и идеја о лакој навигацији кроз странице са приказима информација о покривености путем веза на почетној страници. Поред веза, на почетној страници је дат и сумарни извештај о обради информација о покривености који кориснику даје први увид у утицај теста на кôд.

На страницама које садрже информације о покривености кода датотека налази се неколико врста приказа. Сваки од њих кориснику алата може

дати другачији поглед на разлике у покривености и помоћи у откривању потенцијаних грешака у програму. Такође, прикази кориснику дају увид у то колико различитих елемената кода покривају тестови један у односу на други.

Алат *CovDiff* се може даље проширити тако да генерише и приказује разлике у покривености кода за више од два теста. Тиме би се добиле богатије упоредне информације.

У току рада алата *CovDiff* бришу се све претходне информације о покривености кода. То значи да је потребно поново прикупити информације о покривености кода пројекта, чак и када се изворни кôд теста и пројекта који се тестира нису променили. Ово може бити правац за даљи развој алата који га може учинити ефикаснијим. Једном генерисане информације о покривености кода је могуће чувати. У те сврхе могуће је искористити неку врсту нерелационе базе података. Информације о покривености кода би се поново прикупљале само онда када је промењен изворни кôд теста или пројекта. За праћење промена изворног кôд теста или пројекта могуће је искористити неки од система за верзионисање, попут система *Git* [20].

Разлике у покривености кода тестовима представљају једну врсту мере сличности између два теста. Ова чињеница се може искористи за будући развој алата. На основу прикупљених информације о покривености кода могуће је на различите начине дефинисати мере сличности за тестове. На основу тога колико је неки тест сличан другом могуће је један избацити и тако вршити елиминацију редундантних тестова.

Библиографија

[1] Shivani Acharya and Vidhi Pandya. Bridge between black box and white box–gray box testing technique. *International Journal of Electronics and Computer Science Engineering*, 2(1):175–185, 2012.

[2] Airium, 2023. on-line at: `https://pypi.org/project/airium/`.

[3] Apache License, Version 2.0, 2023. on-line at: `https://www.apache.org/licenses/LICENSE-2.0`.

[4] Thomas Ball and James R Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, 1994.

[5] Thomas Ball, Peter Mataga, and Mooly Sagiv. Edge profiling versus path profiling: The showdown. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 134–148, 1998.

[6] Kent Beck. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[7] Boris Beizer. *Software Testing Techniques, 2nd Edition.* Itp - Media, 1990.

[8] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems.* John Wiley & Sons, Inc., 1995.

[9] Robert Binder. *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley Professional, 2000.

[10] Building LLVM with CMake, 2023. on-line at: `https://llvm.org/docs/CMake.html`.

[11] Clang: a C language family frontend for LLVM, 2023. on-line at: `https://clang.llvm.org/`.

[12] Lee Copeland. *A practitioner's guide to software test design*. Artech House, 2004.

[13] Lisa Crispin and Janet Gregory. *Agile testing: A practical guide for testers and agile teams*. Addison-Wesley Professional, 2008.

[14] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211. IEEE, 2014.

[15] Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on software Engineering*, 31(3):226–237, 2005.

[16] John Erickson, Kalle Lyytinen, and Keng Siau. Agile modeling, agile software development, and extreme programming: the state of research. *Journal of Database Management (JDM)*, 16(4):88–100, 2005.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[18] Gcov - a test coverage program, 2023. on-line at: `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`.

[19] David Gelperin and Bill Hetzel. The growth of software testing. *Communications of the ACM*, 31(6):687–695, 1988.

[20] Git, 2023. on-line at: `https://git-scm.com/`.

[21] GNU Compiler Collection, 2023. on-line at: `https://gcc.gnu.org/`.

[22] GNU General Public License, 2023. on-line at: `https://www.gnu.org/licenses/gpl-3.0.en.html`.

[23] Institute of Electrical & Electronics Engineers (IEEE). IEEE Standard for Software Unit Testing. *ANSI/IEEE Std 1008-1987*, pages 1–28, 1986.

[24] Mohd Ehmer Khan and Farmeena Khan. A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Science and Applications*, 3(6), 2012.

[25] Janusz Laski and William Stanley. *Software verification and analysis: An integrated, hands-on approach.* Springer Science & Business Media, 2009.

[26] Lcov - LTP GCOV extension, 2023. on-line at: `https://github.com/linux-test-project/lcov`.

[27] llc - LLVM static compiler, 2023. on-line at: `https://llvm.org/docs/CommandGuide/llc.html`.

[28] llvm-cov - emit coverage information, 2023. on-line at: `https://llvm.org/docs/CommandGuide/llvm-cov.html`.

[29] LLVM Language Reference Manual, 2023. on-line at: `https://llvm.org/docs/LangRef.html`.

[30] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing.* John Wiley & Sons, 2011.

[31] Srinivas Nidhra and Jagruthi Dondeti. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012.

[32] Projects built with LLVM, 2023. on-line at: `https://clang.llvm.org/`.

[33] Per Runeson. A survey of unit testing practices. *IEEE software*, 23(4):22–29, 2006.

[34] Nayan B Ruparelia. Software development lifecycle models. *ACM SIGSOFT Software Engineering Notes*, 35(3):8–13, 2010.

[35] The LLVM Compiler Infrastructure, 2023. on-line at: `https://llvm.org/`.

[36] The RemoveRedundantDebugValues pass, 2023. on-line at: `https://llvm.org/doxygen/RemoveRedundantDebugValues_8cpp.html`.

Биографија аутора

Никола Перић рођен је 14.12.1997. године у Београду. Завршио је основну школу „Вук Караџић” у Рипњу, а потом и Дванаесту београдску гимназију као носилац Вукове дипломе. Смер информатика на Математичком факултету Универзитета у Београду уписао је 2016. године а дипломирао 2021. године. Након тога уписује мастер студије на истом факултету. Од 2022. године ради као софтверски инжењер у фирми *Syrmia* у оквиру тима за системски софтвер. Сарађује са запосленима из фирме *MediaTek Inc.* на подршци и имплементацији архитектуре *nanoMIPS* у оквиру компилаторске инфраструктуре *LLVM*.