УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Сандра Кукољ

Избор и приоритизација тестних случајева комбиновањем метода тестирања са и без увида у програмски код

МАСТЕР РАД

Нови Сад, 2013.

УНИВЕРЗИТЕТ У НОВОМ САДУ ●**ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:		
Идентификациони број, **ИБР**:		
Тип документације, **ТД**:		Монографска документација
Тип записа, **ТЗ**:		Текстуални штампани материјал
Врста рада, **ВР**:		Дипломски – мастер рад
Аутор, **АУ**:		**Сандра Кукољ**
Ментор, **МН**:		**проф. др Мирослав Поповић**
Наслов рада, **НР**:		**Избор и приоритизација тестних случајева комбиновањем метода тестирања са и без увида у програмски код**
Језик публикације, **ЈП**:		Српски / латиница
Језик извода, **ЈИ**:		Српски
Земља публиковања, **ЗП**:		Република Србија
Уже географско подручје, **УГП**:		Војводина
Година, **ГО**:		**2013.**
Издавач, **ИЗ**:		Ауторски репринт
Место и адреса, **МА**:		Нови Сад; трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		**7/11/6/26/0/0**
Научна област, **НО**:		Електротехника и рачунарство
Научна дисциплина, **НД**:		Рачунарска техника
Предметна одредница/Кључне речи, **ПО**:		**наменски системи, провера квалитета, ББТ; покривеност кода**
УДК		
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		**У овом раду је представљена методологија којом су комбиноване две технике тестирања СТБ уређаја, са и без увида у програмски код, како би се побољшао квалитет тестирања одабране врсте наменских система. Циљ ове методологије је генерисање тестних случајева за нову функционалну тестну кампању засновану на информацијама о покривености кода из претходне тестне кампање, да би се добила највећа могућа покривеност кода.**
Датум прихватања теме, **ДП**:		
Датум одбране, **ДО**:		
Чланови комисије, **КО**:	Председник:	**др Растислав Струхарик**
	Члан:	**др Илија Башићевић**
	Члан, ментор:	**проф. др Мирослав Поповић**

Потпис ментора

UNIVERSITY OF NOVI SAD●**FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:			
Identification number, **INO**:			
Document type, **DT**:		Monographic publication	
Type of record, **TR**:		Textual printed material	
Contents code, **CC**:		Master Thesis	
Author, **AU**:		**Sandra Kukolj**	
Mentor, **MN**:		**prof. Miroslav Popović**	
Title, **TI**:		**Selection and Prioritization of Test Cases by Combining White-Box and Black-Box Testing Methods**	
Language of text, **LT**:		Serbian	
Language of abstract, **LA**:		Serbian	
Country of publication, **CP**:		Republic of Serbia	
Locality of publication, **LP**:		Vojvodina	
Publication year, **PY**:		**2013.**	
Publisher, **PB**:		Author's reprint	
Publication place, **PP**:		Novi Sad, Dositeja Obradovica sq. 6	
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)		**7/11/6/26/0/0**	
Scientific field, **SF**:		Electrical Engineering	
Scientific discipline, **SD**:		Computer Engineering, Engineering of Computer Based Systems	
Subject/Key words, **S/KW**:		**embedded systems; quality assurance; BBT; code coverage**	
UC			
Holding data, **HD**:		The Library of Faculty of Technical Sciences, Novi Sad, Serbia	
Note, **N**:			
Abstract, **AB**:		**In this paper, we present a methodology thatcombines both white-box and black-box testing, in order toimprove testing quality for a given class of embedded systems. The goal of this methodology is generation of test cases for thenew functional testing campaign based on the test coverageinformation from the previous testing campaign, in order tomaximize the test coverage.**	
Accepted by the Scientific Board on, **ASB**:			
Defended on, **DE**:			
Defended Board, **DB**:	President:	**ass. prof. Rastislav Struharik, phD**	
	Member:	**ass. prof. Ilija Bašićević, phD**	Menthor's sign
	Member, Mentor:	**prof. Miroslav Popović**	

Zahvaljujem se mentoru prof. dr Miroslavu Popoviću na stručnoj podršci pri izradi ovog master rada.

Najlepše hvala asistentu Vladimiru Marinkoviću na savetima i pomoći tokom celokupnog rada, kao i svim kolegama koji su doprineli uspešnijoj i bržoj implementaciji projekta.

Na kraju, zahvaljujem se roditeljima na njihovoj stalnoj podršci.

SADRŽAJ

SPISAK SLIKA

SPISAK TABELA

SKRAĆENICE

BBT - Black-box Testing, Testiranje bez uvida u izvorne kodove programa

WBT - White-box Testing, Testiranje sa uvidom u izvorne kodove programa

DTV - Digital Television, Digitalna televizija

OCR - Optical Character Recognition, Optičko prepoznavanje teksta

ATS - Automated Testing System, Automatizovani testni sistem

STB - Set Top Box, Uređaj za prijem digitalnog TV signala

SUT - System Under Test, Sistem koji se testira

DUT -Device Under Test, Uređaj koji se testira

NAViC - Network Attached Video Capture, Uređaj za preuzimanje toka podataka

A/V -Audio/Video, Audio/video

GBT -Gray-box Testing, Testiranje sa delimičnim uvidom u izvorne kodove programa

PBC - Picture Block Compare, Poređenje blokova u slici

DLL - Dynamic Linked Library, Dinamička biblioteka za povezivanje

GUI - Graphical User Interface, Grafička korisnička sprega

SDK - Software Development Kit, Alati za razvoj programske podrške

1. Uvod

U ovom radu predstavljena je metodologija koja kombinuje testiranje bez uvida i sa uvidom u izvorne kodove programa, kako bi se poboljšao kvalitet testiranja za zadatu klasu namenskih sistema. Cilj ove metodologije je generisanje testnih slučajeva za novu testnu kampanju zasnovanu na informacijama o pokrivenosti programskog koda iz prethodne testne kampanje, da bi se dobila što veća pokrivenost. Ove informacije se koriste za selekciju određenih testnih slučajeva radi poboljšanja kvaliteta testiranja i štednje na resursima testiranja. Kao izlaz se dobija skup testnih slučajeva. Dobijene testove obrađuje Executor aplikacija kojom se dobijaju rezultati da li je test ispravan ili neispravan, u zavisnosti od preuzetih slika, izdvajanja teksta (OCR) i poređenja sa očekivanim tekstom. Predstavljena metodologija se na kraju proverava primerom zasnovanom na Android uređaju (platformi). Rezultati primera su pozitivni i ukazuju na to da je predstavljena metodologija primenljiva za testiranje namenskih sistema ove vrste. Ovaj rad je sprovođen u okviru IPA programa za prekograničnu istraživačku mrežu iz domena računarskih informacionih tehnologija (eng. *ICT*). Sproveden je kao međusobna saradnja Univerziteta u Segedinu i Univerziteta u Novom Sadu. Cilj CIRENE projekta je bio poboljšanje kvaliteta testnih slučajeva i redukcija njihovog broja kombinovanjem BBT i WBT metoda.

Rad je sastavljen od sedam poglavlja. Prvo poglavlje je uvod. U drugom poglavlju dat je sažetak vrsta i principa testiranja, a u trećem ideja o njihovoj kombinaciji u cilju prikaza pristupa programskom rešenju, kao i opis metodologije dobijene kombinovanjem BBT i WBT. Četvrto poglavlje daje programsko rešenje. Peto poglavlje prikazuje postupak testiranja, korake pri implementaciji sistema dobijenog kombinovanjem BBT i WBT, okruženje kojim su automatizovani svi neophodni koraci za proces testiranja i rezultate. Šesto poglavlje sadrži zaključak kojim je ukratko opisan dosadašnji rad. Poslednje poglavlje daje spisak literature korišćene pri izradi ovog rada.

2. Vrste i principi testiranja

Ovo poglavlje opisuje osnovne principe funkcionalnog testiranja STB-ova i DTV uređaja i time ukazuje na značaj automatskog BBT testiranja. Nakon pregleda principa testiranja, dat je opis tri vrste testiranja.

Namenski sistemi su obično projektovani tako da obavljaju određene poslove, u određenim okruženjima sastavljenih od fizičke arhitekture (eng. *hardware*) i programske podrške (eng. *software*). Verifikacija namenskih sistema, kao proces dokazivanja da programska podrška zadovoljava specifikaciju, je vrlo važna faza projektovanja. Ukoliko je tehnička specifikacija dovoljno detaljna i informativna, ona može značajno doprineti objektivnosti postupka validacije zahteva korisnika.

Brojne tehnološke inovacije u skorije vreme predstavljaju svakodnevni izazov proizvođačima u industriji potrošnih uređaja. Kao rešenje ovog problema, već decenijama se koriste automatski sistemi za testiranje funkcionalnosti različitih uređaja.

Automatsko testiranje sistema (eng. *ATS*) se primenjuje pri testiranju integrisanih DTV sistema, digitalnih prijemnika (eng. *STB*), *DVD* i *Blu-ray* plejera. ATS se takođe koristi pri testiranju audio i video kvaliteta, automatske navigacije kroz TV meni, preuzimanju i prikazivanju video i audio sadržaja, zapisivanju testnih rezultata u raznim formatima ili u bazi, generisanju izveštaja, itd. Svrha testiranja programske podrške je validacija (da li je ispravna funkcionalnost) i verifikacija (da li i koliko programska podrška ispunjava zahteve potrošača).

Principi testiranja su:

- testiranjem se prikazuje prisutnost defekata u programskoj podršci
- uporno (eng. *exaustive*) testiranje nije moguće, kombinacijom svih mogućnosti
- prvenstveno testiranje prioritetnijih delova programske podrške
- identifikacija gustine grešaka njihovim grupisanjem

- balansirano testiranje resursima koji su na raspolaganju
- neponavljanje istih testnih slučajeva pri svakom izvršavanju (paradoks pesticida)
- testiranje zavisi od sadržaja
- defekti su jasno definisani, a potom rekonstruisani radi njihove ispravke
- neophodno je testirati prilagodljivost na drugačije platforme, određivanjem najkritičnijih kombinacija i preduslova

Testiranje je važan proces u razvoju programske podrške. Cilj testiranja je da se pronađu greške u kodu, kako bi ih naknadno uklonili, ili jednostavno da se proveri pretpostavka da u programu nema grešaka. Ovakvoj pretpostavci se pristupa sa pokušajem da se defekti pronađu. Naravno, u praksi nije moguće ukloniti sve greške, niti u potpunosti dokazati pomenutu pretpostavku, ali je najvažnije obezbediti da su sve specificirane funkcije ispravne pre nego što se odluči da se proizvod postavi na tržište.

Jedna od metoda provere kvaliteta sistema je testiranje programske podrške. Različite metode testiranja programske podrške imaju primene u različitim okruženjima namenskih sistema, koji ponekad zahtevaju specifična rešenja pri testiranju. Metode koje su kombinovane su ukratko opisane u narednim poglavljima.

2.1 Testiranje bez uvida u izvorne kodove programa

Cilj BBT testiranja (eng. *Black Box Testing*) je da se pronađu uslovi pod kojima se program ne ponaša prema zadatim specifikacijama. Testeri nemaju načina da vide unutrašnju logiku sistema, međutim, ovde je bitna specifikacija zahteva. Ukoliko je specifikacija dovoljno temeljna i precizna, BBT može obezbediti pouzdano otkrivanje programskih grešaka. Poblem se javlja zbog toga što nije lako u potpunosti istestirati program zbog ograničenih mogućnosti testera [1].

Metodologija BBT testiranja je efikasan pristup za *funkcionalno testiranje* televizijskih i multimedijalnih uređaja. Ovakvo ponašanje uređaja (eng. *behaviour testing*) zapravo omogućava testeru da, bez poznavanja unutrašnje strukture izvornih kodova (bez ikakvog znanja o implementaciji i ponašanju programa) i činjenica kako programska podrška funkcioniše, testira funkcionalnost određenog programa [2]. Osnovna ideja ovakvog funkcionalnog testiranja je da se uređaj koji se testira zapravo tretira kao crna kutija sa unapred poznatim ulazima i izlazima (u bukvalnom prevodu BBT znači testiranje crne kutije, gde se pod *kutijom* podrazumeva program ili ceo uređaj). Tester ne mora imati nikakvo znanje o kodu programa, odnosno, nema potrebe da

piše, niti čita postojeći kod jer je čitav sistem koncipiran kao da se posmatra iz ugla korisnika. Cilj testera je da prođe kroz kod preko korisničke sprege (eng. *User Interface*), što detaljnije, kao kad bi sam kod bio na raspolaganju. Ova veština se razvija vremenom, kroz praksu i strpljivo promatranje.

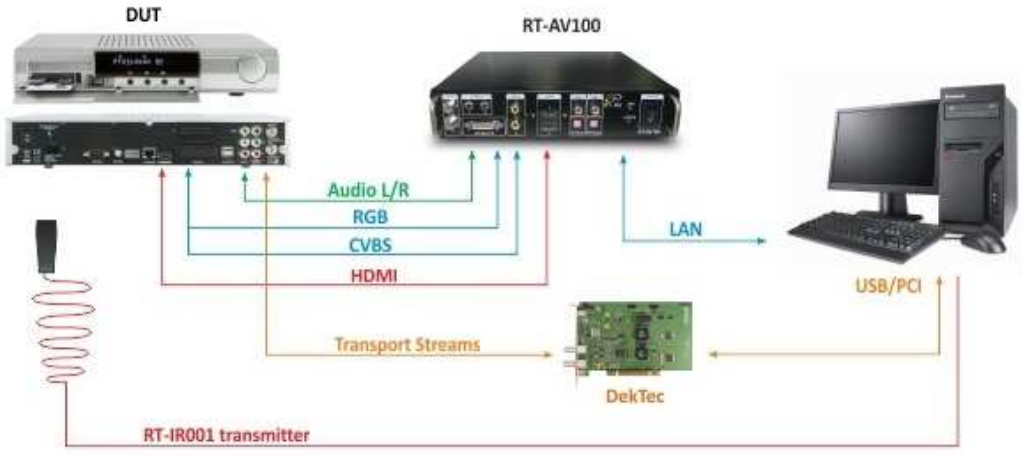
Testni slučajevi mogu da se pišu kada se definišu zahtevi. Međutim, nedostaci su da je testne slučajeve u nekom pogledu teško napisati, nemoguće je sve putanje pokriti i može biti viška testnih slučajeva. Testni slučajevi proveravaju specifikacije i zahteve programske podrške, a cilj im je da lako pronađu defekte u funkcionisanju programske podrške, što objašnjava odakle potiče pojam funkcionalno testiranje. BBT metodi se koriste uglavnom na višim nivoima testiranja (testiranje komponenti, sistemsko testiranje, *User Acceptance Testing*, a može se vršiti i na nivoima testiranja uređaja). Namenjene za funkcionalno testiranje, BBT metode zanemaruju unutrašnje mehanizme sistema ili komponenti i fokusiraju se na izlaze generisane kao odziv sistema na određene ulaze i uslove izvršavanja testnih slučajeva.

Izvršavanje testnih slučajeva može biti ručno, automatsko i polu-automatsko, na osnovu čega se BBT deli na tri istoimena tipa. Ručno testiranje zahteva da se svi koraci testiranja izvršavaju ručno, na osnovu zamišljenog scenarija. Kod polu-automatskog testiranja, tester bira koji će se testni slučajevi izvršavati, kao i kod ručnog testiranja. Međutim, kod polu-automatskog testiranja sistem vrši automatsku kontrolu i upravljanje uređaja u razvoju, a poseban algoritam ugrađen u sisteme za testiranje odlučuje o ispravnosti rezultata testnih slučajeva. U slučaju automatskog testiranja, kriterijumi za donošenje odluka (da li je ishod testnog slučaja uspešan ili neuspešan) se postavljaju na osnovu testnih zahteva. Kriterijumi se prosleđuju testnom mehanizmu za upravljanje, ugrađenom u kontrolnu aplikaciju, kao parametar korišćen za izveštavanje o uspešnosti ili neuspešnosti testnih slučajeva. Automatsko testiranje integrisanih DTV sistema uzima za činjenicu funkcionalno testiranje podržavanih sprega (eng. *interface*). Uređaji koji generišu audio i video sadržaj, namenjeni za testiranje određene sprege, povezani su sa SUT (eng. *System Under Test*), koji izvršava ponovnu obradu sadržaja. Nakon što su ciljevi predefinisanog scenarija ostvareni, dobijeni SUT izlaz se preuzima sa matične ploče TV uređaja i njegov sadržaj se upoređuje sa referentnim. Kontrolni emulatori namenjeni za određene DTV proizvođače omogućava automatsku navigaciju kroz TV meni i podešavanje opcija kao što su boja, osvetljenost, oštrina, jačina zvuka, itd.

Programska podrška BBT testiranja je aplikacija za kontrolu, razvoj i automatsko izvršavanje testnih slučajeva. Programska podrška sistema poseduje algoritme koji vrše poređenje dobijenih izlaza sa očekivanim, tj. referentnim izlazima, a takođe generiše rezultate koji pokazuju da li se uređaj ponaša u skladu sa očekivanjima, ili ne. Osnovna ideja je da se testira šta zapravo programska podrška obavlja, a ne kako je neka funkcija implementirana.

Prednosti ovakvog načina testiranja su da se proverava krajnje ponašanje programske podrške, da je moguće pisati testne slučajeve nezavisno od načina projektovanja programske podrške, a može se koristiti i za testiranje različitih implementacija sa minimalnim brojem izmena. Pokrivenost programskog koda ovakve metode testiranja podrazumeva procenat testiranih zahteva, kao i procenat zabeleženih izuzetaka dobijenih testiranjem. Važno je napomenuti da stopostotna pokrivenost ne znači da je sve u potpunosti istestirano. Potpuno testiranje je uglavnom nemoguće i to predstavlja stalni problem pri testiranju. Okruženje BBT funkcionalnog testiranja STB-a prikazano na slici 2.1 sastoji se iz sledećih komponenti:

- PC testne stanice
- uređaja za preuzimanje slike RT-AV100 (eng. *Grabber device*)
- uređaja za napajanje (eng. *power switch control unit*)
- kontrolera uređaja koji se testira i RC emulatora (eng. *Remote Controller*)



Slika 2.1. BBT okruženje za STB funkcionalno testiranje

2.2 Testiranje sa uvidom u izvorne kodove programa

Testiranje programske podrške ovom metodom ispituje unutrašnju strukturu aplikacije. Obično se koristi na nivou testiranja uređaja. Ova vrsta testiranja se još zove i strukturno testiranje. Dakle, za razliku od BBT metode testiranja, WBT (eng. *White Box Testing*) metodom imamo uvid u izvorne kodove, iskaze i odluke, odnosno, možemo saznati kako program radi. Inženjeru je omogućeno da proučava kod, a njegovo neprestano praćenje i analiza opisuje kriterijum pokrivenosti. Programski kod se izvršava tokom testiranja programa, kako bi se izmerila pokrivenost. Ovaj pojam saopštava u kojoj meri je izvršeno testiranje i obično se

predstavlja u procentima. Kod WBT metoda, pokrivenost znači procenat uslovnih grana gde su obe strane grana testirane.

Izdvajaju se dva tipa WBT kriterijuma pokrivenosti koda:

- pokrivenost instrukcija, kojom se definiše da li bi trebali određeni delovi programa da se izvrše tokom testiranja (npr. blokovi koda, metode, klase, moduli, ili same instrukcije)
- pokrivenost grana, koja definiše kako bi trebale različite putanje programa da se izvrše, ili kako bi trebale da se sprovedu različite odluke tokom testiranja. Sve ovo zavisi od definisanja pozicije na kojoj se program nalazi. Na instrukcijskom nivou se mogu proučavati odluke, pa čak i delovi odluka kao što je pokrivenost uslova. S druge strane, na nivou metoda se mogu proučiti putanje grafa poziva.

Informacije o pokrivenosti je potrebno preuzeti nakon izvršavanja testnih slučajeva. Za to postoji više načina:

- Generisanje traga izvršenja, odnosno delova koda kroz koje je program prošao tokom izvršenja testnog slučaja, je važan deo WBT testiranja. Da bi se izračunala dužina traga izvršavanja (eng. *traceability*) i pokrivenost koda (eng. *coverage*), potrebno je pratiti izvršavanje programa, što se može postići instrumentacijom i debagovanjem.
- Instrumentacija programskog koda podrazumeva umetanje instrukcija koje obezbeđuju informacije o interesantnim delovima programa. Informacioni sadržaj delova od značaja zavise od nivoa pokrivenosti. Ovakva instrumentacija se može ostvariti unutar izvornog koda ili u binarnom kodu.
- Instrumentacija posredničkog sloja (eng. *middleware*) može biti dobro rešenje ako se koristi jedan srednji sloj za više programa, a potrebno je da se dobiju informacije iz svih programa. Posrednički sloj se nalazi između fizičke arhitekture i operativnog sistema, a sastoji se od biblioteka i drajvera. Umetanjem metoda u posrednički sloj, dobijaju se povratne informacije izvršavanja, pa je moguće izvući izvesne informacije koje su potrebne.
- Moguće je izmeniti izvršno okruženje (eng. *execution framework*), tj. virtuelnu mašinu, proširivanjem koda. To je sloj programske podrške između izvršnog binarnog koda i operativnog sistema, odnosno, okruženje u kojem se izvršava specijalni posrednički binarni jezik, tipično prilagođen za prevođenje jednostavnih izvornih kodova. Može se koristiti trag poziva, koji se sastoji od informacija pozivanih metoda.

- Debagovanje je moguće ostvariti na HW nivou, ali je potreban debug port na HW ili debager uređaj, koji može da komunicira sa HW preko uobičajenih portova. Debager može da čita kod u HW i ubacuje tačke prekida (eng. *breakpoints*) u njega, a potom smešta dodatni kod. Na ovaj način se obezbeđuju detaljne informacije u vezi izvršavanja programa.

Ovakve informacije o pokrivenosti se mogu iskoristiti za manipulaciju grupe izvršenih testnih slučajeva. Naime, moguće je odabrati određene testne slučajeve da bi postigli određeni cilj pri testiranju, ili se mogu poređati po željenom redosledu, kako bi postigli traženu pokrivenost koda za kraći vremenski period. Prednosti ovog metoda su dobra pokrivenost koda i mogućnost pokrivanja nekih specijalnih testnih slučajeva. Ispitivaču je omogućeno da u potpunosti pregleda kod programa, dok sastavlja testne slučajeve. Dozvoljava da odredi do koje mere, tj. koliko temeljno treba testirati program.

S druge strane, mane ovog pristupa su da potpuna pokrivenost koda nije uvek dobra pri proceni funkcionalnosti zbog iznenadnih ponašanja i različitih testnih ciljeva. Druga mana je to što testni slučajevi zasnovani na projektovanim karakteristikama mogu da pogrešno protumače probleme sistema, a isto tako postoji potreba da se menjaju testni slučajevi pri izmeni implementacije/algoritma.

Glavne razlike WBT u odnosu na BBT se ogledaju u poznavanju dizajna i detalja o kodu programa. Obično se teži maksimalnom procentu pokrivenosti poznatih svojstava i najviše zadire u algoritamske diskontinuitete i grane. WBT se koristi da bi se obezbedila ispravnost detalja implementacije, dok BBT ima naglasak na susretu sa zahtevima i generalno na ponašanju, a testovi se formiraju tako da obezbede ispravnu funkcionalnost programske podrške. WBT može biti moćnije od BBT u nekim aspektima. Testeri su u mogućnosti da otkriju skrivene greške u programu prolaskom kroz izvorni kod, što podrazumeva da poznaju unutrašnju strukturu ciljnog sistema. Najčešće postoji zahtev da programi budu instrumentalizovani, kako bi se generisao izveštaj.

2.3 Testiranje sa delimičnim uvidom u izvorne kodove programa

Metodom testiranja programske podrške sa delimičnom predstavom o programskim kodovima, ili GBT testiranjem (eng. *Gray Box Testing*), unutrašnja struktura uređaja koji se testira je samo delimično poznata. Inženjeru je obezbeđen pristup unutrašnjim strukturama podataka i algoritmima na osnovu čega se sastavljaju testni slučajevi, ali na nivou korisnika

(BBT nivou). Gray-box metodologija je kombinacija četiri vrste testiranja, BBT, WBT, regresionog i mutacionog testiranja [5]. Cilj ovog testiranja je potraga za defektima uzrokovanih neodgovarajućom strukturom ili pogrešnom upotrebom aplikacija. Drugi naziv ove tehnike testiranja je poluprovidno testiranje.

Neophodno je da testeri poseduju dokumentaciju sa opisom aplikacija, koje sakupljaju da bi napisali testne slučajeve. Zasniva se na generisanju testnih slučajeva na osnovu zahteva, što znači da je potrebno podesiti sve uslove pre nego što se program počne testirati.

Neke od prednosti GBT, budući da predstavlja kombinaciju WBT i BBT testiranja, jesu da se manifestuju dobre strane obe vrste testiranja. Zatim, bazirana je na funkcionalnoj specifikaciji, a testeri rukuju inteligentnim testnim scenarijima. Uprkos svim navedenim prednostima, GBT metoda održava stalnu granicu između testera i inženjera zaduženog za razvoj. Na kraju krajeva, WBT i BBT su komplementni sa svojim individualnim prednostima i ograničenjima.

2.4 Poređenje metoda testiranja programske podrške

U naredne tri tabele date su prednosti i mane metoda BBT, WBT i GBT, redom [6]. Radi jasnijeg pregleda prethodna tri odeljka, tabelom 4 dat je kratak opis osnovnih razlika između tehnika BBT, GBT i WBT.

BBT	
Prednosti	**Mane**
Veoma prikladan i efikasan za velike delove programskog koda	Pokrivenost je ograničena-samo odabran broj testnih scenarija se izvršava
Nije potreban pristup programskom kodu	Neefikasno testiranje zbog činjenice da tester ima ograničeno znanje o testiranoj aplikaciji
Testeri ne moraju imati predstavu o implementaciji, programskom jeziku, niti operativnom sistemu	Tester ne može da se usredsredi na određeni deo programskog koda ili delove podložne greškama

Tabela 1. Prednosti i mane BBT

WBT	
Prednosti	**Mane**
Kada tester ima znanje o izvornom kodu, lako se zaključuje koji tip podataka efikasno može pomoći pri testiranju aplikacije	Mnogo putanja neće biti istestirano jer je nemoguće pronaći sve skrivene greške u kodu
Pomaže pri optimizaciji programskog koda	Budući da testeri moraju biti obučeni, troškovi su veći
Prilikom pisanja testnog scenarija, postiže se maksimalna pokrivenost zahvaljujući znanju testera o prohramskom kodu	Nije lako baviti se ovim vidom testiranja jer je neophodno koristiti specijalne alate, kao npr. analizatore koda i alate za debagovanje

Tabela 2. Prednosti i mane WBT

GBT	
Prednosti	**Mane**
Nudi kombinovane dobrobiti BBT i WBT testiranja	Iz razloga što nije moguć pristup izvornom kodu, mogućnost da se proveri kod i pokrivenost je ograničena
Testeri se ne oslanjaju na izvorni kod, već na definiciju interfejsa i funkcionalne specifikacije	Testni slučajevi mogu biti višak ukoliko je neki već izvršavan
Na osnovu ograničenih informacija, tester može da napravi dobar testni scenario	Testiranje svakog ulaznog toka je nerealno, jer bi potrajalo besmisleno dugo, a iz tog razloga, mnoge putanje programa bi ostale netestirane
Testni slučajevi se prave iz ugla korisnika	

Tabela 3. Prednosti i mane GBT

BBT	GBT	WBT
Nije potrebno znati unutrašnje procese aplikacije	Postoji delimično znanje o unutrašnjim procesima	Tester ima potpun uvid u unutrašnje procese aplikacije
Poznato je još pod nazivima: *closed box testing*, *data driven testing* i *functional testing*	Poznato još kao *translucent testing*, iz razloga štotester ima veoma ograničen uvid u unutrašnjost aplikacije	Poznato pod nazivima *clear box testing*, *structural testing* ili *code based testing*
Izvršavaju ga krajnji korisnici, testeri i inženjeri za razvoj	Izvršavaju ga krajnji korisnici, testeri i inženjeri za razvoj	Obično ovaj metod primenjuju testeri i inženjeri za razvoj
Testiranje je zasnovano na spoljnjim očekivanjima, a unutršnje ponašanje sistema je nepoznato	Testiranje na bazi dijagrama baze podataka visokog nivoa i dijagrama toka podataka	Unutrašnji procesi u potpunosti poznati i tester može praviti testne podatke na osnovu toga
Ovaj metod najkraće traje i veoma jeiscrpan	Vreme trajanja je nešto duže i ovaj metod je istrajniji	Ima najduže vreme trajanja i veliku istrajnost
Nije prikladno za algoritamsko testiranje	Nije prikladno za algoritamsko testiranje	Prikladno za algoritamsko testiranje
Testiranje se vrši metodom pokušaja i grešaka	Poznati opsezi podataka i njihove granice mogu biti testirani	Opsezi podataka i njihove granice mogu se bolje testirati

Tabela 4. Razlike između BBT, GBT i WBT

3. Metodologija dobijena kombinacijom BBT i WBT

3.1 Cilj i zadatak projekta

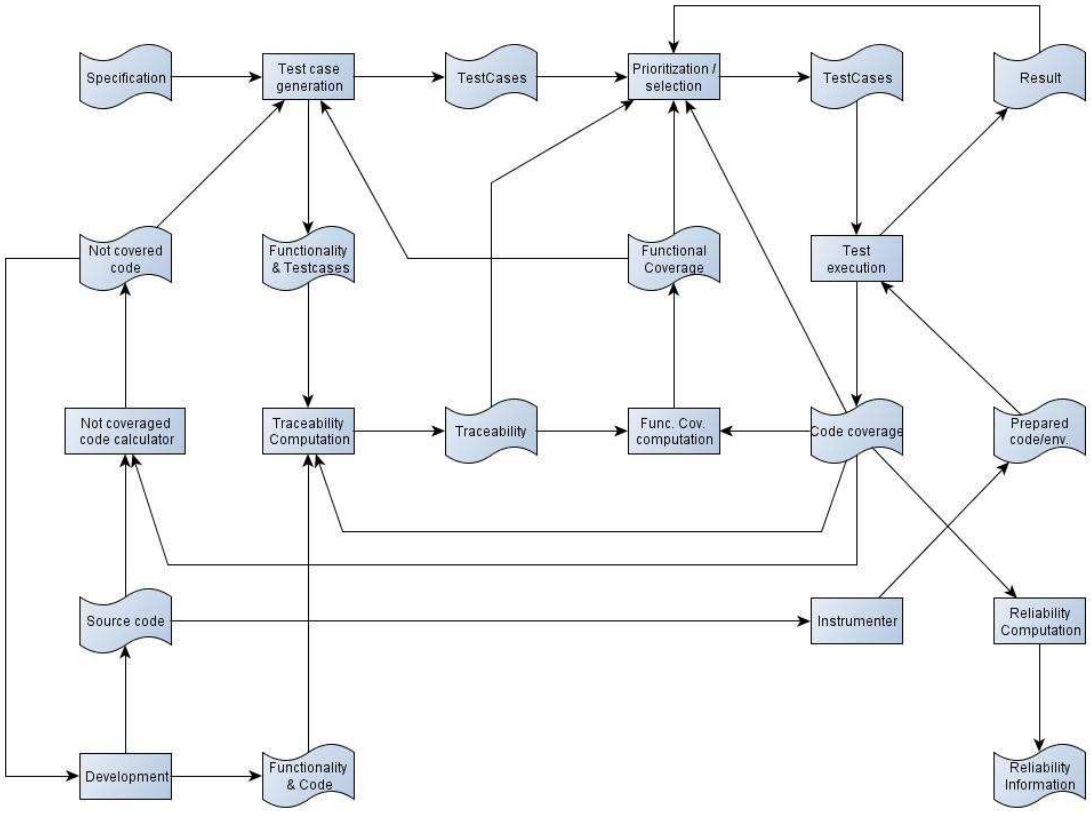
Kao što je već spomenuto u uvodu, ovaj rad je urađen u okviru projekta međugranične saradnje Srbija-Mađarska, ICT istraživačke mreže i saradnje univerziteta u Novom Sadu i Segedinu. Cilj zajedničkog rada je razvoj i testiranje namenskih sistema, konkretno STB i DTV, ali primarno je bilo definisati metodologiju za testiranje namenskih sistema. Naime, definisan je složen proces testiranja dobijen kombinacijom znanja iz BBT i WBT testiranja. Istraživanje je takođe obuhvatalo pretragu radova u kojima je opisano upotrebljavanje tehnika testiranja sa i bez uvida u programski kod, kao i njihova kombinacija u okruženju namenskih sistema. Rad je imao cilj i da se unapredi kvalitet i/ili efikasnost funkcionalnog testiranja, a proces uključuje i korake koji koriste podatke generisane WBT tehnikom testiranja.

Glavni cilj rada je specijalizacija i primena metodologije testiranja na specifičnim namenskim sistemima. Ciljni uređaj je *Marvell* STB zasnovan na Android platformi, a centralni deo BBT sistema je računar na koji su povezani DUT uređaj i uređaji koji se koriste prilikom testiranja. Na računaru je neophodno imati instalisanu programsku podršku za automatsko testiranje kojom se upravljaju svi delovi ATS sistema i verifikuje ispravnost funkcionisanja sistema. Korišćena je RT-Executor kontrolna aplikacija za izvršavanje testnih slučajeva. To je alat koji omogućava kontrolu, razvoj i izvršenje BBT testnih slučajeva, a u novijoj verziji i Python skripti. Uključuje niz biblioteka za analizu slike i zvuka, a koristi i baze podataka da bi se kontrolisala prava pristupa samoj aplikaciji.

Metodologija testiranja sistema je izmenjena kako bi se prilagodila pomenutom sistemu. Proces kombinacije BBT i WBT testiranja je prikazan na slici 3.1. Testni slučajevi se generišu na osnovu specifikacije, automatski ili ručno. Na osnovu njih se prikupljaju informacije na bazi

izvršnog okruženja sa instrumentacijom. Ove informacije inženjeru govore kolika je vrednost pokrivenosti koda za odgovarajući testni slučaj, pored informacija o ishodu izvršenja testnih slučajeva. Informacije o pokrivenosti koda se koriste i na druge načine. Ukoliko na kraju izvršenog testiranja uporedimo ceo izvorni kod sa ukupnom dobijenom vrednošću pokrivenosti koda, lako se mogu definisati nepokriveni delovi koda. Kao razlozi nedostatka informacija o pokrivenosti mogu biti slučajevi:

- kada su nepokriveni delovi koda dosegnuti, ali nema odgovarajućih testnih slučajeva za te delove koda. Iz tog razloga je neophodno napisati nove testne slučajeve.
- kada nepokriveni delovi koda nisu dosegnuti, to znači da su to zapravo nekorišćeni delovi koda (eng. *dead code*) koje je potrebno ukloniti



Slika 3.1. Šema sistema kombinacije testiranja sa i bez uvida u programski kod

Informacije vezane za praćenje traga izvršavanja programskog kodaimaju višestruku namenu. Konkretno, na osnovu ovih informacija se računa pokrivenost koda, ukoliko su date veze između zahteva (funkcionalnosti) i izvornog koda, kao i veze između testnih slučajeva i zahteva. Informacija o tragu izvršavanja se može iskoristiti i za selekciju i prioritizaciju testnih

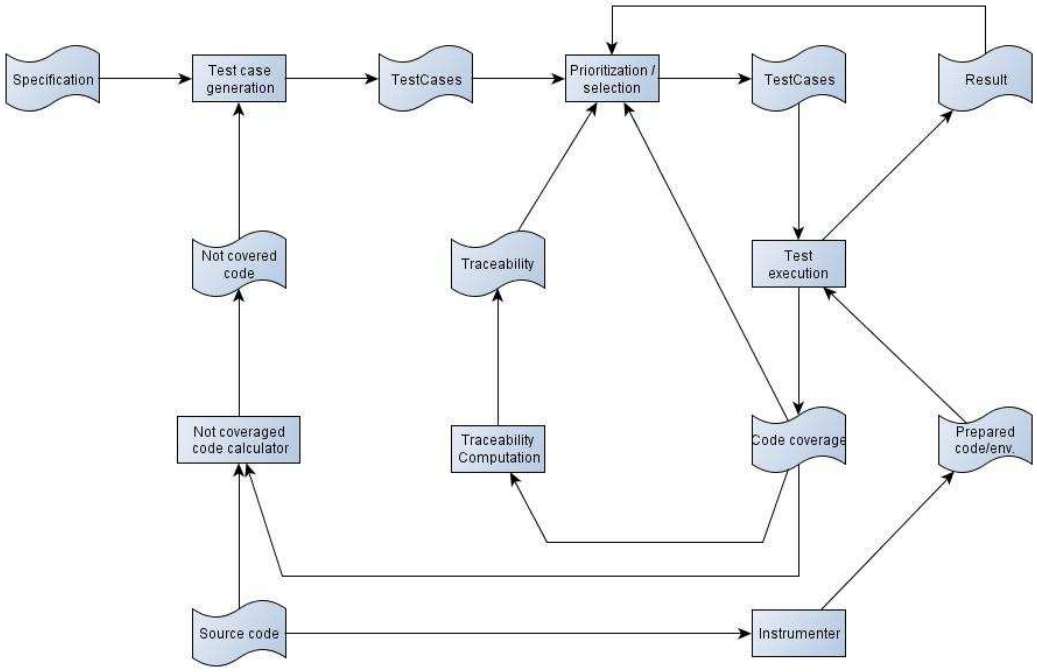
slučajeva. Pokrivenost koda se takođe može iskoristiti kao baza za proračun informacija u vezi dužine traga izvršavanja. Na primer, ako postoje zavisnosti između funkcionalnosti i testnih slučajeva, mogu se dodeliti segmenti koda određenim funkcionalnostima koje oni implementiraju. Informacije o pokrivenosti koda mogu poslužiti i za određivanje pouzdanosti programske podrške i testnih slučajeva.

Opisana metodologija je osmišljena s ciljem uspešnog kombinovanja i poboljšanja dve tehnike testiranja. Ideja je bila da se dobiju informacije o prikupljenim podacima u okruženju namenskog sistema. Ovakve informacije je moguće dobiti na tri načina, na nivou integrisanog kola, na nivou virtuelne mašine i na aplikacionom nivou (instrumentacija koda).

3.2 Specijalizovana metodologija

Na ovom projektu je ciljano da se već poznata metodologija specijalizuje i implementira na određenom namenskom sistemu. Ciljni uređaj je STB zasnovan na Android platformi, odnosno digitalni TV prijemnik. Analizom postojeće metodologije testiranja, napravljene su izmene kako bi se prilagodili ovom sistemu, a načini primene su takođe definisani.

Najvažniji aspekt prilagođavanja metodologije testiranja je prikupljanje informacija o pokrivenosti programskog koda i tragu izvršavanja, koje su od velikog značaja za iscrpno testiranje (eng. *exaustive testing*), ali pre svega i za selekciju testnih slučajeva i njihovo sortiranje na osnovu zadatog kriterijuma. Očekuje se istovremeno smanjenje broja testnih slučajeva i skraćeno vreme testiranja. Opis ovakve modifikovane metodologije prikazan je na slici 3.2.



Slika 3.2. Izmenjena metodologija testiranja

Proces testiranja počinje generisanjem testnih slučajeva na osnovu specifikacije, a koji će se izvršavati automatski pomoću postojeće kontrolne aplikacije. Pre samog izvršavanja testnih slučajeva, bira se neki od kriterijuma selekcije i prioritizacije, na osnovu procene i potreba korisnika. Nakon izvršavanja testa, dobijaju se informacije o pokrivenosti koda i tragovima izvršavanja programa od ranije izvršenih testnih slučajeva (npr. odabirom samo onih testnih slučajeva kojim se poziva bar jedna izmenjena metoda, izvršavanje najpre onih testnih slučajeva kojim se poziva više metoda, ili daju veću pokrivenost, testni slučajevi kojima se poziva najveći broj metoda, ili oni koji su najčešće bili neuspešni u skorijem vremenskom periodu). Za prvo izvršavanje se biraju testni slučajevi na osnovu jednog od dva kriterijuma i tako izabrani se dodaju u testni plan koji se izvršava naknadno. U ovoj iteraciji se testni slučajevi iz odgovarajućeg testnog plana izvršavaju i mere se pokrivenost koda i trag izvršavanja programa. Ove iteracije se ponavljaju dok se ne ispuni cilj željenog kriterijuma, npr. dok se ne dobije određeni nivo pokrivenosti koda. Tokom izvršavanja specijalno pripremljnih testnih slučajeva, koriste se instrumentalizovane verzije aplikacija, koje omogućavaju da se izdvoje pozivane metode, odnosno vrednost mere pokrivenosti koda. Instrumentacijom se modifikuje binarni kod Android aplikacija, ali se ne menja funkcionalnost aplikacije, već se samo umeću dodatne instrukcije pogodne za preuzimanje potrebnih informacija. Pokrivenost koda može da se iskoristi za definisanje veze sa dužinama traga izvršavanja (npr. povezivanjem zahteva sa segmentima koda koji implementiraju određeni zahtev), dok se informacije o dužini traga izvršavanja mogu

iskoristiti za izbor testnih slučajeva. Delovi koda koji nisu pokriveni mogu poslužiti za proširivanje testnog plana, dodavanjem odgovarajućih testnih slučajeva, ili nalaženjem nekorišćenog koda koji se kasnije može ukloniti. Ovi koraci se moraju obaviti ručno, a to najčešće radi osoba koja piše testne slučajeve i programski kod.

4. Programsko rešenje

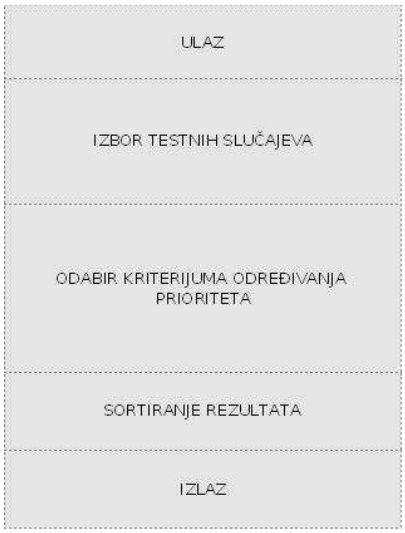
4.1 Opis programskog rešenja

Programsko rešenje za selekciju i prioritizaciju testnih slučajeva je pisano u C++ programskom jeziku, uz pomoć STL biblioteke [7], asastavljeno je iz dve celine. Prvi deo projekta je DLL [8] aplikacija, uz sve definicije funkcija koje se izvoze (*eng. exported functions*) za DLL aplikaciju. U okviru ovog dela programa se uključuje zaglavlje sa deklaracijama klasa. Date klase se izvoze uslovnim uključivanjem (prevođenje je dozvoljeno samo ako je definisan makro SELECTIONPRIORITIZATION_EXPORTS, čime je olakšan izvoz funkcija iz DLL-a).

Drugi deo je program koji sadrži ulaznu tačku aplikacije. Ovde je definisana *main* funkcija sa pozivima uvezenih funkcija (*eng. imported functions*) iz DLL-a. DLL biblioteka se dinamički povezuje u trenutku izvršavanja i pozivaju se funkcije koje su članice klase CSelectionPrioritization. Nakon uspešnog učitavanja biblioteke funkcijom *LoadLibrary(),* dinamički se pravi pokazivač na objekat klase. Operator *new* vraća pokazivač na novi objekat i automatski se pozivaju konstruktor i destruktor.

Ulazni dokumenti smešteni su u okviru kataloga programa za ispitivanje DLL-a. Ovi ulazni podaci se koriste da bi se na osnovu njih odredio raspored testnih slučajeva na osnovu kriterijuma odabira prioriteta, nakon njihovog izbora.Pomenuti ulazni dokumenti će biti detaljnije opisani u sledećem poglavlju, a za sada je dovoljno reći da ih generiše alat za instrumentaciju, da su CSV formata i da sadrže podatke koji se parsiraju da bi se dobile potrebne informacije (kao npr. nazivi testnih slučajeva i dužine tragova izvršavanja) neophodne za rad alata za izbor i prioritizaciju testnih slučajeva. Koraci realizacije rešenja su prikazani na slici 4.1. Najpre se ulazne datoteke učitavaju i proverava se njihova ispravnost. Bira se jedan od načina

izbora testnih slučajeva, a potom i jedan od kriterijuma određivanja prioriteta. Prikupljeni podaci se sortiraju i na izlazu zapisuju u *bbp* datoteku.



Slika 4.1.Sistem za izbor i određivanje prioriteta testnih slučajeva

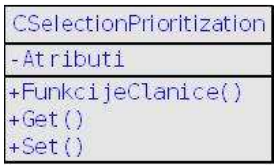
U zavisnosti od potrebe, formiraju se odgovarajuće strukture podataka [8]. Koriste se strukture tipa mape (gde je svaka stavka mape jedan par ključ-vrednosti (K,V)), strukture tipa vektor i strukture tipa liste. Sortiranje rezultujuće mape se vrši na osnovu jedinstvenih ključeva.

4.2 Klasa CselectionPrioritization

Klasa CSelectionPrioritization je uvezena ili izvezena pomoću atributa klase za skladištenje (*dllexport*, *dllimport*). Zahvaljujući tome, nema potrebe za definisanjem potrebnih modula .def datotekom pri kreiranju DLL-a [9]. Koristi se ekstenzija *__declspec*, kojom se specificira da će instanca odgovarajućeg tipa biti skladištena sa *Microsoft* specifičnim atributima klase za skladištenje (eng. *storage-class*). Ukoliko je cela klasa uvezena ili izvezena, kao u ovom slučaju, nije dozvoljeno eksplicitno deklarisati podatke i funkcije članice klase sa dllimport ili dllexport.

Privatne članice-podaci klase CSelectionPrioritization su većina podataka i sve strukture podataka, osim jedne, koja se nalazi u okviru naredne klase. U okviru javne specifikacije unutar deklaracije klase CSelectionPrioritization, nalaze se konstruktor i destruktor, sve realizovane funkcije članice i metode za pristup promenljivama (*get* i *set* metode koje omogućavaju odvajanje detalja o tome kako se podaci čuvaju i koriste), koje su unutar ove klase definisane kao

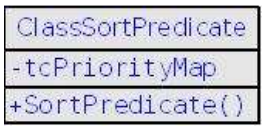
umetnute funkcije (eng. *inline*). Prikaz opisane klase i njenih generalizovanih atributa i funkcija dat je na slici 4.2, gde su sa znakom – označeni javni, a sa + privatni podaci.



Slika 4.2. Dijagram klase CselectionPrioritization

4.3 Funktor klasa SortPredicate

Ova klasa sadrži jedan privatni član-podatak i u okviru nje se preklapa operator (). Pomoću funktor klasa se definiše objekat koji podseća na funkciju po svom izgledu, ali u suštini se u okviru njega koristi operator preklapanja (eng. *operator overloading*) poziva funkcije, koji kao i operator indirekcije i podskriptni operator, mora biti član klase. Ovim je omogućeno da se objekti klase koriste kao imena funkcija. Dakle, ovakvom realizacijom funkcije za sortiranje elemenata strukture mape je omogućeno pisanje imena klase sa parametrom koji predstavlja mapu koja se sortira, umesto poziva slobodne funkcije kao treći parametar funkcije za sortiranje (što nije dozvoljeno, jer je pomenuta mapa privatni podatak član klase CSelectionPrioritization). Prikaz naziva, atributa i operacije klase dat je na slici 4.3.



Slika 4.3. Dijagram klase SortPredicate

4.4 Opis modula SelectionPrioritization.cpp

Naravno, na početku ovog modula je uključeno zaglavlje SelectionPrioritization.h, koje sadrži prethodno opisane deklaracije klasa CSelectionPrioritization i SortPredicate. U ovom modulu se nalazi konstruktor sa svim potrebnim inicijalizacijama promenljivih i strukturapodataka. Pored konstruktora, tu je i destruktor, a potom slede definicije funkcija

članica klase CSelectionPrioritization. U narednom teskstu, neke od funkcija biće ukratko opisane.

4.4.1 Funkcija za izbor svih testnih slučajeva

Prototip funkcije: void CSelectionPrioritization::LoadAll()

Ulazni argumenti: -

Opis: Svi testni slučajevi se preuzimaju iz liste već pri početnom popunjavanju struktura podataka. Ovaj način odabira slučajeva zapravo i nije njihov pravi izbor, već preuzimanje svih raspoloživih testnih slučajeva iz jednog testnog plana. Naime, svi raspoloživi testni slučajevi uzimaju se u obzir, a koji će potom biti poređani prema odgovarajućem redosledu niza oznaka unutar strukture, u ovom slučaju od najniže do najviše oznake testnog slučaja.

4.4.2 Funkcija za prikupljanje svih testnih slučajeva

Prototip funkcije: boolCSelectionPrioritization::PopulateAllTCs(constchar *)

Ulazni argumenti: filesPath

Opis: U ovoj funkciji se zauzima memorija za potrebnu apsolutnu putanju na kojoj se nalaze direktorijumi sa testnim slučajevima, odakle se iz naziva direktorijuma preuzimaju imena testnih slučajeva za njihovo kasnije sortiranje. Zadata putanja se daje kao argument pri pozivu funkcije kojom se pronalaze nazivi testnih slučajeva i ona je ukratko je opisana u narednom odeljku. Argument za ovu funkciju je pokazivač na željenu putanju koja se prosleđuje.

4.4.3 Funkcija za pronalazak kataloga

Prototip funkcije:boolCSelectionPrioritization::ListDirs(const wchar_t *)

Ulazni argumenti: sDir

Opis: Ovom funkcijom se prolazi kroz sve direktorijume na datoj putanji na kojoj se nalaze direktorijumi sa nazivima testnih slučajeva. Nakon što je prvi direktorijum pronađen, zapisuje se u vektor strukturu podataka. Potom se prelazi na naredni direktorijum na datoj putanji i vektor se

ponovo popunjava novim nazivom testnog slučaja. Argument ove funkcije predstavlja pokazivač na apsolutnu putanju do naziva željenih direktorijuma sa testnim slučajevima.

4.4.4 Funkcija za izbor odabranih testnih slučajeva

Prototip funkcije: void CSelectionPrioritization::LoadSelected(bool)

Ulazni argumenti: changesLoaded

Opis: Drugi način izbora slučajeva ispivanja je njihov odabir na osnovu poslednjih izmenjenih metoda u ulaznom dokumentu. Ukoliko se u ulaznom dokumentu nalazi bar jedna metoda, biraju se samo oni testni slučajevi koji pokrivaju tu jedinu metodu. Naravno, ukoliko postoji više metoda, razmatraće se svi oni testni slučajevi koji pozivaju date metode, tako da nema ponavljanja istih testnih slučajeva. Ovo je obezbeđeno posebnom funkcijom koja je u odeljku 4.4.5 opisana. Način na koji su metode predstavljene je isti kao i kod testnih slučajeva, od najniže oznake ka najvišoj. Na osnovu posebne promenljive za odabir vrste izbora testnih slučajeva, bira se jedan od ova dva načina izbora. Argument predstavlja *flag* za označavanje da je funkcija ispravno izvršena.

4.4.5 Funkcija za određivanje prioriteta na osnovu najdužeg traga izvršavanja

Prototip funkcije: void CSelectionPrioritization::LongestTraces(void)

Ulazni argumenti: -

Opis: Trag izvršavanja programa (eng. *trace*) je sistemom zapisan u vidu brojeva koji predstavljaju broj izvršavanja različitih metoda koje su pozivane (eng. *runtime method call*), za svaki testni slučaj ponaosob. Naravno, moguće je da određena funkcija bude pozvana više puta (ili nijednom) u okviru jednog testnog slučaja, ali se ovde o tome ne vodi računa. Više reči o ovome biće u nekom od narednih kriterijuma. Ovaj kriterijum prioritizacije se zasniva na dužini pomenutih tragova izvršavanja, koji su dati u ulaznom dokumentu. Cilj ovog kriterijuma je da se dobije lista testnih slučajeva po opadajućem prioritetu. Testni slučajevi na osnovu kojih je dobijen duži trag izvršavanja imaju veći prioritet. U principu, testni slučajevi na osnovu kojih je dobijen duži trag izvršavanja zapravo su oni kojima se pozivao najveći broj metoda. Računanje dužine traga izvršavanja dato je formulom:

$$trace\_len = cum\_trace\_len[n] - cum\_trace\_len[n-1] \quad (1)$$

Napravljena struktura u ovom slučaju je mapa, čiji su ključevi (naziv testnog slučaja) preslikani na odgovarajuće vrednosti (dužinom traga izvršavanja). U ulaznoj datoteci nije neophodno da postoje informacije za sve testne slučajeve, tako da u tim slučajevima u strukturi mape na mestu dužine traga izvršavanja stoji nula. Testni slučajevi čijim se izvršavanjem dobija najduži trag izvršavanja (pozove se najveći broj različitih metoda), imaju veći prioritet.

Kada postoji učitana struktura sa dužinama traga izvršavanja, moguće je upotrebiti i naredne dve vrste određivanja prioriteta, u zavisnosti od izbora. Ovo su zapravo podaci o pokrivenosti funkcija.

4.4.6 Funkcija za određivanje prioriteta testnih slučajeva kojima se poziva veći broj metoda

Prototip funkcije: bool CSelectionPrioritization::MostCoverage(void)
Ulazni argumenti: -
Opis: Ovaj kriterijum prioritizacije zasnovan je na preuzetim informacijama o pokrivenosti koda iz ulaznog dokumenta. Radi objašnjenja pokrivenosti programskog koda, formula (2) objašnjava dobijene vrednosti u procentima iz *csv* dokumenata. Broj metoda koje su pozivane tokom izvršavanja predstavljen je sa *rm*, a *am* predstavlja broj svih metoda na raspolaganju.

$$coverage = (rm / am) * 100 \quad (2)$$

Na osnovu informacija o metodama koje su pozivane, popunjava se mapa prioriteta na osnovu koje se kasnije sortiraju izlazne strukture. Na osnovu ovog dela se obezbeđuje i nadograđena varijanta ovog kriterijuma, naime, poseban kriterijum koji je opisan u narednom odeljku.

4.4.7 Funkcija za dodatno određivanje prioriteta na osnovu testnih slučajevakojima se poziva najveći broj metoda

Prototip funkcije: void CSelectionPrioritization::MostAdditionalCoverage(void)

Ulazni argumenti: -

Opis: Iz pomenutog razloga, u okviru funkcije za ovaj kriterijum određivanja prioriteta stoji poziv funkcije koja je pomenuta u prethodnom poglavlju, nakon čega sledi poziv funkcije koja formira radnu listu, odnosno listu redukovanu na osnovu samo onih funkcija koje nisu pokrivene prioritetnijim testnim slučajevima. Praktično, uzima se najprioritetniji testni slučaj i prioriteti za ostale testne slučajeve se ponovo računaju na osnovu metoda koje još uvek nisu pozivane.

Na početku ove funkcije se pravi kopija vektora testnog plana, odnosno vektor izabranih testnih slučajeva. Posmatraju se funkcije najprioritetnijeg testnog slučaja, ali najpre se gornji, najprioritetniji testni slučaj briše iz radne liste. Pre bilo kakve izmene strukture, kao u ovom slučaju uklanjanja, vektor se ponovo sortira, kako bi redosled određivanja prioriteta ostao nenarušen.

Iteriranjem kroz mapu testnih slučajeva sa metodama koje se pozivaju i indeksiranjem po testnom slučaju koji se briše, iterira se i kroz izmenjenu mapu nastalu od prethodne, tako da se ona indeksira po iteratoru prethodne. U slučaju da iterator transformisane mape nije jednak obrisanom testnom slučaju, i ako nije kraj pomenute mape, umanjuje se prioritet za dati testni slučaj. U okviru mape testnih slučajeva (preslikanih, odnosno mapiranih odgovarajućim nazivima metoda) se pomoću njenog iteratora, nakon svakog smanjenja prioriteta briše element iz izmenjene mape.

4.4.8 Funkcija za određivanje prioriteta testnih slučajeva koji su najčešće bili neuspešni

Prototip funkcije: void CSelectionPrioritization::MostFailing(long)

Ulazni argumenti: reqInterval

Opis: Računa se vremenski interval u okviru koga se posmatraju testni slučajevi koji su najveći broj puta bili neuspešni (eng. *failed*). Oni koji su bili uspešni (eng. *passed*) ili koji su bili neuspešni manji broj puta imaju niži prioritet od onih koji su češće bili neuspešni.

Najpre se beleži trenutno sistemsko vreme i količina vermena, vremenski razmak (količina vremena izračunata na osnovu određenih parametara) u sekundama. Potreban vremenski period se računa kao razlika trenutnog vremena i pomenutog vremenskog razmaka. Ovako dobijeni vremenski period se prebacuje u format mesec/dan/godina (eng. *mm/dd/yy*). Dakle, period u kome se razmatraju testni slučajevi počinje od trenutka kada počinje izračunati vremenski razmak, sve do trenutnog vremenskog trenutka.

Iterira se kroz vektor testnog plana i sa tim indeksom se prolazi kroz mapu koja sadrži testne slučajeve i njihove rezultate, tokom izvršavanja na određeni datum. Ukoliko je ishod testnog slučaja neuspešan, i ukoliko se datum datog testnog slučaja nalazi unutar proračunatog vremenskog perioda, prioritet mape prioriteta tstnih slučajeva indeksirane sa iteratorom vektora testnog plana se uvećava.Ulazni argument predstavlja vremenski interval izražen u danima.

4.4.9 Funkcija za sortiranje i zapis izlaza

Prototip funkcije: void CSelectionPrioritization::Output(char *, char*, char*,char*, unsigned)

Ulazni argumenti: filePath, projectName, projectDatabase, projectHtml, cutting

Opis: Izlaz iz sistema predstavlja novi dokument u koji se upisuje vektor sa testnim slučajevima, sortiran na osnovu mape prioriteta. Čuvanje izlaznih podataka je korisno za obnavljanje informacija o pokrivenosti (eng. *coverage*), kako za poslednje izvršene testne slučajeve, tako i za čuvanje informacija o pokrivenosti koda testnim slučajevima koji nisu izabrani na početku datog izvršavanja.

Prvi ulazni argument služi za prosleđivanje apsolutne putanje do *bbp* projektnog dokumenta (eng. *Bezier surface files*), dok je drugi argument sam naziv ovog projektnog dokumenta. Treći argument je putanja do baze podataka datog projekta, a naredni je putanja do projektnog HTML-a. Poslednji argument je ograničenje kojim se naznačava da se odatle lista sortiranih testnih slučajeva skraćuje. Pomenuti bbp dokument se kreira iz kontrolne aplikacije, učitavanjem svih željenih testnih slučajeva koji čine plan testiranja.

5. Postupak testiranja i rezultati

Ovo poglavlje daje celokupan opis automatskog procesa testiranja STB-ova. Najpre je dat kratak opis uređaja koji su korišćeni, a potom način na koji su testni slučajevi pisani. BBT sistem daje korisniku vizuelni prikaz toka testiranja, a omogućava i interakciju korinika i programske podrške preko grafičke sprege. Na kraju ovog poglavlja ukratko se opisuje proces testiranja, radi lakšeg razumevanja dobijenih rezultata.

5.1 Pregled korišćenih uređaja za testiranje

Kao što je već pomenuto, DUT čini Marvell STB zasnovan na Android platformi, što sa TV uređajem podrazumeva SUT. STB je uređaj koji se povezuje sa spoljnim izvorom signala, a sadrži dekoder i pretraživač kanala. Njime se signal pretvara u sadržaj koji se može prikazati na TV ekranu. Ovakav STB koji se testira, povezuje se preko programske podrške pomoću zadatih odgovarajućih komandi u aplikaciji korišćenoj za testiranje.

NAViC uređaj se koristi za preuzimanje audio i video signala u realnom vremenu (eng. *grabber device*) u svrhu nadgledanja ili testiranja STB i DTV-a. Njime se snimaju nekompresovane slike ili video sekvence različitih formata i rezolucija u internu memoriju, a prilikom testiranja postoji mogućnost da se prati trenutno stanje izvršavanja (eng. *live preview*) preko LAN sprege. Preuzete slike (ili video) se prebacuju na računar za buduće korišćenje.

Ulogu daljinskog upravljača ima USB kontrolni uređaj pod nazivom RC Emulator, koji simulira komande upravljača koje su zadate pomoću unapred snimljenih makroa (za automatizaciju). Pomoću njega se mogu slati i primati *IR* komande STB-u za kontrolisanje toka

izvršavanja aplikacije. Za njegovo kontrulisanje neophodno je posedovati odgovarajuću programsku podršku za rukovanje, kao i *RC Capture* (*RCC*) aplikaciju kojom se tester „obučava“ na daljinskom upravljaču, da bi se omogućila kontrola preko RT-Executor aplikacije.

5.2 Pisanje testnih slučajeva za Android aplikacije

Unutar definicije svakog od testnih slučajeva, opisuju se svi koraci koji se moraju izvršiti da bi se testirala željena funkcionalnost. U svakom testnom slučaju se sa stanovišta kontrolne logike moraju nalaziti delovi kojim se zadaju akcije generatora signala, komande koje se šalju DUT uređaju, naredbe za uređaj koji snima izlaze, kao i opis algoritma za poređenje i dobijanje rezultata. Generatoru signala zadaju se signali koje je potrebno izgenerisati i poslati uređaju koji se testira. Tip signala koji će biti izgenerisan zavisi od funkcionalnosti koja se testira. To može biti npr. tok podataka pri testiranju slike, ili pak zvučni zapis ukoliko se testira zvuk.

U namenskim sistemima nije jednostavno testiranje bez izvornih kodova programa, ili samo pomoću binarnog koda. Za testiranje je izabrana aplikacija zasnovana na Androidu iz razloga što je za Android platforme izvorni kod javno dostupan (eng. *open-source*), a veoma je korisno da je izvorni kod aplikacije vidljiv zbog testiranja pomoću WBT metoda.

Najpre su pisani testovi u Python programskom jeziku za navigaciju kroz Android aplikaciju preuzetu sa Interneta. Ideja je bila da se koristi što jednostavnija aplikacija koja bi demonstrirala rezultate koji bi se dobili predstavljenom metodologijom nakon testiranja. Preuzeta je aplikacija kalkulator, za koju su testni slučajevi napisani tako da se nasumično generišu operandi, kao i jedna od osnovnih operacija (sabiranje, oduzimanje, množenje, deljenje), s tim da su u konačnim rezultatima prikazani rezultati za jednostavnu kalkulator aplikaciju. Prikaz funkcije za generisanje jednocifrenih, dvocifrenih ili trocifrenih brojeva iz jednog od ovakvih testnih slučajeva dat je na slici 5.1.

Ovakvi testni slučajevi su svakako interesantniji, međutim, ipak je napravljen još jedan skup testnih slučajeva za sistemsku aplikaciju Android meni-ja, a kao jedan od razloga je stara verzija RT-Executor-a koja ne podržava Python skripte. Aplikacija za koju su pisane BBT skripte je *MarvellLegacyUI*, sa različitim podešavanjima u okviru meni-ja. Isečak ovakvog testnog slučaja formata *tst* dat je na slici 5.2. U testnim slučajevima sa ekstenzijom .tst napisani su svi potrebni koraci za testiranje, opisani u daljem tekstu.

Najpre se zadaje korak za postavljanje promenljive *res* za DLL kojim se aktivira alat za generisanje rezultata (postavljanjem varijable opcijom SET), a kojom se na kraju testnog slučaja

preuzimaju rezultati (opcijom GET). Nakon postavljanja *res* promenljive, zadaju se koraci za *preview*, za PCM algoritam (eng. *Picture Block Compare*) i za navigaciju kroz aplikaciju. Nakon što se izvrše ovi koraci i prikupljanje rezultata, slika se preuzima i zapisuje. Ipak, za potrebe demonstracije, napisani su i kompletni BBT testni slučajevi za kalkulator aplikaciju.

Uopšteno, tokom izvršavanja testnih slučajeva može se koristiti više različitih uređaja za snimanje izlaza, u zavisnosti šta se testira. Svakom od tih uređaja se pristupa preko unapred definisane sprege, čime se pojednostavljuje dodavanje novog uređaja u sistem, kao i njegovo kontrolisanje od strane sistemske podrške. Pri završetku testiranja, kontrolni uređaj za snimanje izlaza snima dobijeni izlaz, dok komunikacija sa nekim od uređaja zavisi od raspoloživih sprega.

```
def GenerateRandomNum():
    maxDigits = 3
    digitsBeforeDot = maxDigits

    range_end = (10**maxDigits)-1/999
    number = random.randint(0, range_end) # nasumicni broj do 1000
    print "Random number is ", number
    numStr = str(number)
    numLen = len(numStr)
    print "numLen len: ", numLen
    if numLen == 1: #jednocifreni
        rest = number
        print "One digit ", rest
        if rest == 0:
            device.handler("DUTController", "SET", "[RESET_0]", "")
        elif rest == 1:
            device.handler("DUTController", "SET", "[RESET_1]", "")
        elif rest == 2:
            device.handler("DUTController", "SET", "[RESET_2]", "")
        elif rest == 3:
            device.handler("DUTController", "SET", "[RESET_3]", "")
        elif rest == 4:
            device.handler("DUTController", "SET", "[RESET_4]", "")
        elif rest == 5:
            device.handler("DUTController", "SET", "[RESET_5]", "")
        elif rest == 6:
            device.handler("DUTController", "SET", "[RESET_6]", "")
        elif rest == 7:
            device.handler("DUTController", "SET", "[RESET_7]", "")
        elif rest == 8:
            device.handler("DUTController", "SET", "[RESET_8]", "")
        elif rest == 9:
            device.handler("DUTController", "SET", "[RESET 9]", "")
```

```
elif numLen == 2: #dvocifreni
    print "Two digits "
    rest = number%10
    print "Rest: ", rest
    num = number/10
    f = Fraction(num)
    print "f: ", f
    list2 = [f,rest]
    for listItem in list2:
        if (listItem == 0):
            device.handler("DUTController", "SET", "[RESET_0]", "")
        elif (listItem == 1):
            device.handler("DUTController", "SET", "[RESET_1]", "")
        elif (listItem == 2):
            device.handler("DUTController", "SET", "[RESET_2]", "")
        elif (listItem == 3):
            device.handler("DUTController", "SET", "[RESET_3]", "")
        elif (listItem == 4):
            device.handler("DUTController", "SET", "[RESET_4]", "")
        elif (listItem == 5):
            device.handler("DUTController", "SET", "[RESET_5]", "")
        elif (listItem == 6):
            device.handler("DUTController", "SET", "[RESET_6]", "")
        elif (listItem == 7):
            device.handler("DUTController", "SET", "[RESET_7]", "")
        elif (listItem == 8):
            device.handler("DUTController", "SET", "[RESET_8]", "")
        elif (listItem == 9):
            device.handler("DUTController", "SET", "[RESET_9]", "")
elif numLen == 3: #trocifreni
    print "Three digits "
    rest1 = number%100
    print "Rest_1: ", rest1
    rest2 = rest1%10
    print "Rest_2: ", rest2
    num1 = rest1/10
    f1 = Fraction(num1)
    print "f1: ", f1
    num = number/100
    f = Fraction(num)
    print "f: ", f
    list3 = [f,f1,rest2]
    for listItem in list3:
        if (listItem == 0):
            device.handler("DUTController", "SET", "[RESET_0]", "")
        elif (listItem == 1):
            device.handler("DUTController", "SET", "[RESET_1]", "")
        elif (listItem == 2):
            device.handler("DUTController", "SET", "[RESET_2]", "")
        elif (listItem == 3):
            device.handler("DUTController", "SET", "[RESET_3]", "")
        elif (listItem == 4):
            device.handler("DUTController", "SET", "[RESET_4]", "")
        elif (listItem == 5):
            device.handler("DUTController", "SET", "[RESET_5]", "")
        elif (listItem == 6):
            device.handler("DUTController", "SET", "[RESET_6]", "")
        elif (listItem == 7):
            device.handler("DUTController", "SET", "[RESET_7]", "")
        elif (listItem == 8):
            device.handler("DUTController", "SET", "[RESET_8]", "")
        elif (listItem == 9):
            device.handler("DUTController", "SET", "[RESET_9]", "")
    time.sleep(0.5)
return number
```

Slika 5.1. Funkcija za generisanje nasumičnog broja do tri cifre

```
[step]
description =
device =  DUTController
command = "RIGHT 5"
delay = 400
option = [SET]

[step]
description =
device =  DUTController
command = "OK 1"
delay = 400
option = [SET]

[step]
description =
device =  DUTController
command = "DOWN 1"
delay = 400
option = [SET]

[step]
description =
device =  DUTController
command = "RIGHT 2"
delay = 400
option = [SET]

[step]
description =
device =  DUTController
command = "OK 1"
delay = 400
option = [SET]
```

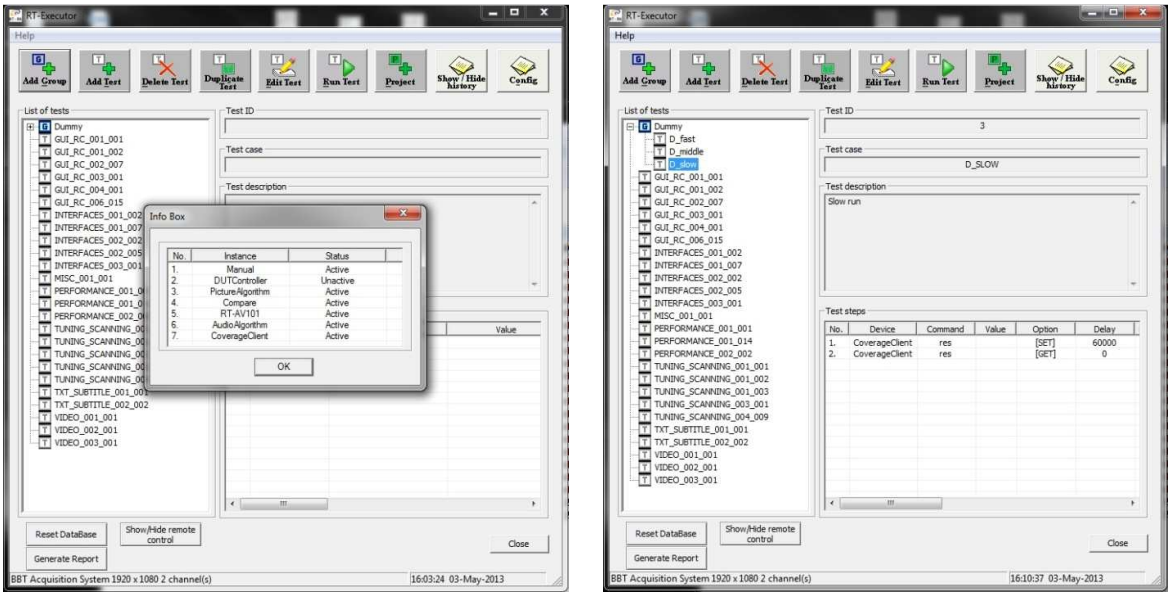
Slika 5.2. Primer dela testnog slučaja za navigaciju kroz aplikaciju

5.3 Proces automatskog testiranja

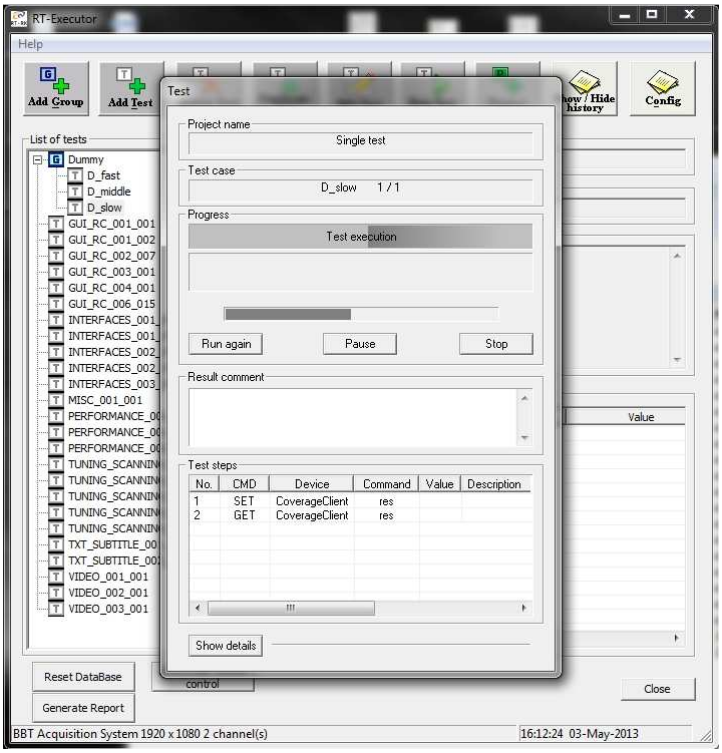
Kao što je u drugom poglavlju već pomenuto, BBT metodom testiranja se samo proverava funkcionalnost sistema. Ne zna se ništa o unutrašnjoj strukturi testiranog uređaja, a tester samo posmatra kako uređaj reaguje na zadate ulaze. WBT metodom se kod testira i pronalaze se greške ili nekorišćeni delovi koda. Prikupljaju se informacije od značaja koje potpomažu funkcionalno testiranje ili pružaju informaciju o tome da li je potrebno da se naprave dodatni testni slučajevi. Time bi se mogao redukovati broj testnih slučajeva kojima je moguće pozvati sve metode, ili bi se mogao napraviti redosled testnih slučajeva, kako bi se najpre odabrali najefikasniji testni slučajevi radi skraćivanja vremena izvršavanja.

Automatizacijom testnog procesa se obezbeđuje konzistencija u generisanju testnih slučajeva i objektivnost, kao i da su svi zahtevi za testiranje sistema obezbeđeni. U svrhe BBT testiranja, koristi se RT-Executor kontrolna aplikacija. Ovim BBT alatom se definiše niz BBT testnih slučajeva u okviru jednog dela ovakvog okruženja, a moguće je i upravljanje daljinskim upravljačem, automatizacija testnih slučajeva i plana testiranja. Testni slučajevi čine testne planove čiji se testovi pokreću jedan za drugim. Na slici 5.3 je dat prikaz početnog stanja RT-Executor aplikacije i opis testnog slučaja koji je izabran, kao i određen broj testnih slučajeva u

okviru datog testnog plana. Kada se testni slučaj pokrene, prikazuje se dodatni prozor kojim se prati napredovanje izvršavanja datog testnog slučaja, što se može videti na slici 5.4. Na kraju testiranja je moguće proveriti ishod pritiskom na dugme *Show/Hide history*.



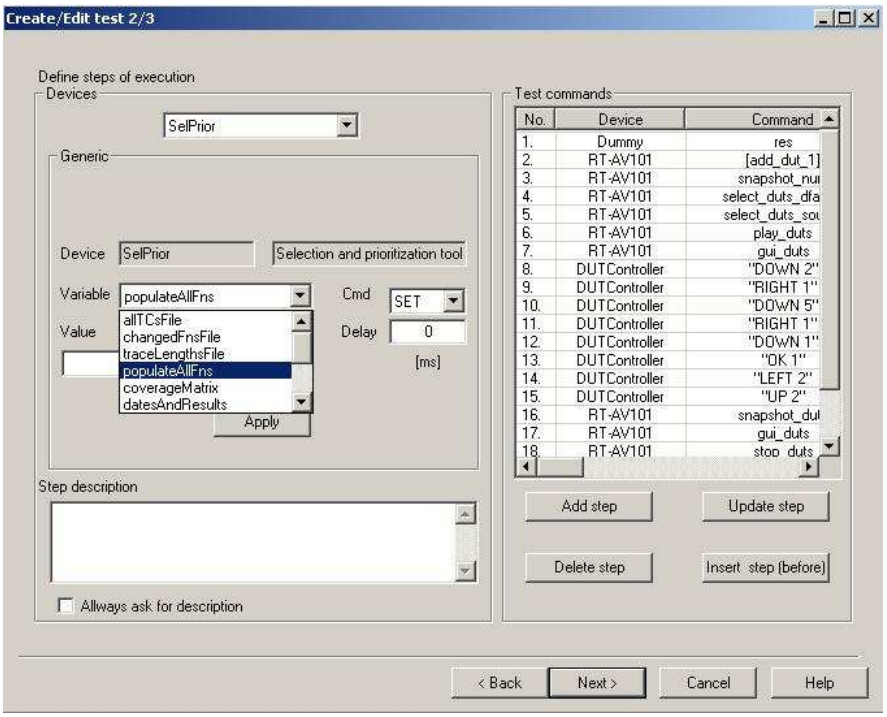
Slika 5.3. RT-Executor aplikacija nakon njenog otvaranja i opis odabranog testnog slučaja



Slika 5.4. Prikaz izvršavanja testnog slučaja

Kao ishod pokretanja testnih slučajeva dobija se drugi skup testnih slučajeva, nakon što je izvršena odabrana prioritizacija testnih slučajeva, a uz to i informacije o pokrivenosti koda pomoću drugog pomenutog dodatka. Ovakvom automatizacijom izvršavanja testnih slučajeva se obezbeđuje veća pouzdanost, što je očigledno jer se odrađuje bez čovekove intervencije. Ovime se povećavaju adaptivne sposobnosti sistema i samo deo testnih slučajeva se zapravo izvršava. Na osnovu rezultata testiranja, dalje se odlučuje koji će se testni slučajevi izvršavati u narednim pokretanjima testnog plana.

Izgled kontrolne aplikacije pri izboru i prioritizaciji testnih slučajeva prikazan je na slici 5.5. Ovaj prozor se dobija odabrom opcije za pravljenje novih testnih slučajeva ili menjanje postojećih, ali, najpre se u učita DLL dodatak za izbor i prioritizaciju, kao i DLL dodatak za proračun pokrivenosti koda. Nakon što su dodaci aktivirani, vrši se izbor testnih slučajeva koji su potom spremni za izvršavanje. Svaki testni slučaj mora imati neke početne korake za njegovu identifikaciju i resetovanje vrednosti uređaja. Takođe mora sadržati i neke završne korake koji proveravaju prikupljene informacije o pokrivenosti koda na osnovu server aplikacije koja je instalisana na uređaju. Na ovaj način RT-Executor prikuplja informacije o aktuelnoj pokrivenosti koda.

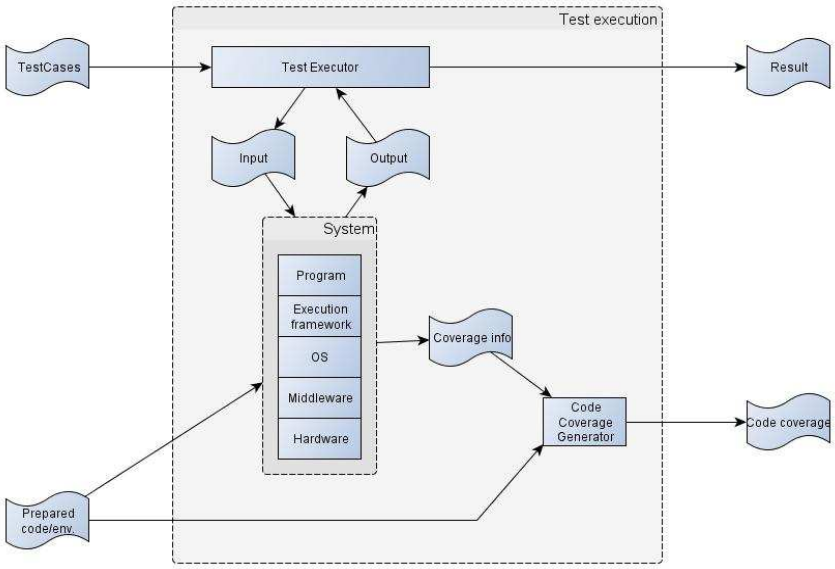


Slika 5.5. Izgled aplikacije pri odabiru kriterijuma za prioritizaciju

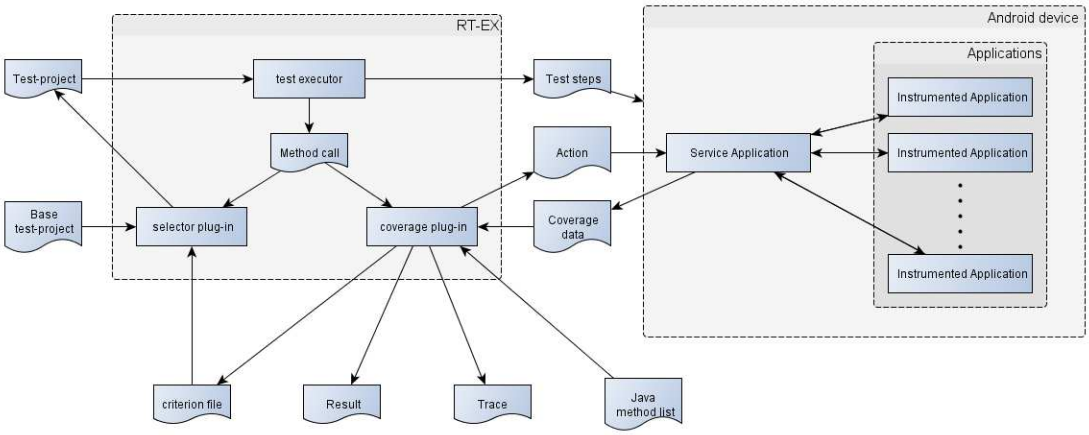
Dobijeni rezultati instrumentalizovanog sistema su tragovi dobijeni izvršavanjem programa. Ovi tragovi se obrađuju u skladu sa pripremljenim okruženjem i generisanim

podacima o pokrivenosti koda. Predložena metodologija automatskog testiranja zahteva odgovarajuće konektore i programe na uređaju (npr. kontroler), i na taj način se obezbeđuje direktna kontrola namenskog sistema.

Prikaz opisanog sistema testiranja dat je na slici 5.6, a ishod testiranja može biti uspešan (eng.*pass*), neuspešan (eng. *fail*) ili kada je ishod nejasan (eng. *inconclusive*), tj, iz nekog razloga se ne mogu odrediti rezultati testnog slučaja. Pod uspešno izvršenim testnim slučajem se podrazumeva ishod testiranja kojim su zadovoljeni svi kriterijumi testnog slučaja. Slika 5.7 prikazuje detaljnije funkcionisanje procesa izvršavanja testnih slučajeva.



Slika 5.6. Prikaz unutrašnje strukture procesa testiranja



Slika 5.7. Šema procesa izvršavanja testnih slučajeva

5.4 Analiza dobijenih rezultata

5.4.1 Instalacija i instrumentacija

Najpre je bilo potrebno instalisati aplikaciju za merenje pokrivenosti koda na uređaj. Instalacija aplikacije za instrumentaciju se vrši u nekoliko osnovnih koraka, kojima prethodi upoznavanje sa kodom programa za instalaciju napisanom na Java programskom jeziku. Prvi korak je postavljanje Android SDK putanje u skripti *config.bat*. Drugi korak je smeštanje datoteke aplikacije (sa ekstenzijom .apk) koja još nije instrumentalizovana u određeni katalog, gde se instrumentacija svaki put obavlja. Treći korak je pokretanje nekoliko izvršnih skripti (eng. *batch files*) sa ekstenzijom .bat, u kojima je, između ostalog, navedena putanja za smeštanje datoteka koje se formiraju. Radi lakšeg budućeg korišćenja dobijenih skripti, napravljeno je nekoliko novih izvršnih skripti, koje pozivaju prethodno pomenute, sa još nekim dodatnim podešavanjima koja se pri svakoj instrumentaciji moraju ponovo izmeniti. Prva izvršna skripta služi za instrumentaciju apk dokumenta i sadrži niz naredbi za postavljanje putanja, kopiranje određenih dokumenata i pozivanje alata za dekompajliranje apk-datoteka (*apktool,* [10]). Nakon ovoga je potrebno u AndroidManifest.xml (eng. *Extensible Markup Language*) ubaciti liniju koda radi dozvole za korišćenje Interneta. U drugoj skripti se poziva skripta za pravljenje instrumentalizovanog apk dokumenata, vrši se konekcija na Marvell ploču i poziva još jedna skripta kojom se instališe instrumentalizovan dokument apk formata.

Nakon uspešne instalacije alata za instrumentaciju koda, u predodređenom *bin* katalogu, na putanji gde je vršena instrumentacija, dobijen je dokument *InstrumentedAPK_methods.csv* (eng. *Comma-Separated Values*). Ovaj dokument je kasnije kopiran u željeni katalog. U datom primeru sa slike 5.8, u petoj liniji je to npr. putanja D:\RT-Executor\RT-Executor\methods. Sve putanje iz konfiguracionog dokumenta su implementirane u okviru prvog dela aplikacije za automatizaciju procesa podešavanja, testiranja i izveštaja o rezultatima. Sadržaj dokumenta koji sadrži opis metoda Android aplikacije kalkulator je dat na slici 5.9, sa zaglavljem (koje opisuje kolone dokumenta) i prototipove metoda ove aplikacije. Na slici 5.10 se mogu videti prototipovi metoda sistemske aplikacije nakon njene instrumentacije.

```
log_level       : debug
log_location    : D:/RT-Executor/RT-Executor/out/log.txt
res_location    : D:/RT-Executor/RT-Executor/out/result.csv
base_location   : D:/RT-Executor/RT-Executor/out/
methods_location : D:/RT-Executor/RT-Executor/methods/
host_name       : 192.168.238.77
```

Slika 5.8. Izgled konfiguracionog dokumenta

```
ProjectName;com.covTest-1_instrumented
0          ;void <init>(com.covTest.CoverageTestActivity)                    ;com.covTest.CoverageTestActivity$1
1          ;public void onClick(android.view.View)                           ;com.covTest.CoverageTestActivity$1
2          ;void <init>(com.covTest.CoverageTestActivity)                    ;com.covTest.CoverageTestActivity$3
3          ;public void onClick(android.view.View)                           ;com.covTest.CoverageTestActivity$3
4          ;public void <init>()                                             ;com.covTest.BuildConfig
5          ;void <init>(com.covTest.CoverageTestActivity)                    ;com.covTest.CoverageTestActivity$2
6          ;public void onClick(android.view.View)                           ;com.covTest.CoverageTestActivity$2
7          ;java.lang.String[] value()                                       ;android.annotation.SuppressLint
8          ;public void <init>()                                             ;com.covTest.R$layout
9          ;void <init>(com.covTest.CoverageTestActivity)                    ;com.covTest.CoverageTestActivity$4
10         ;public void onClick(android.view.View)                           ;com.covTest.CoverageTestActivity$4
11         ;public void <init>()                                             ;com.covTest.R$id
12         ;void <init>(com.covTest.CoverageTestActivity)                    ;com.covTest.CoverageTestActivity$5
13         ;public void onClick(android.view.View)                           ;com.covTest.CoverageTestActivity$5
14         ;public void <init>()                                             ;com.covTest.CoverageTestActivity
15         ;android.widget.EditText access$0(com.covTest.CoverageTestActivity);com.covTest.CoverageTestActivity
17         ;android.widget.TextView access$2(com.covTest.CoverageTestActivity);com.covTest.CoverageTestActivity
16         ;android.widget.EditText access$1(com.covTest.CoverageTestActivity);com.covTest.CoverageTestActivity
19         ;public void onCreate(android.os.Bundle)                          ;com.covTest.CoverageTestActivity
18         ;com.covTest.Calculator access$3(com.covTest.CoverageTestActivity) ;com.covTest.CoverageTestActivity
21         ;public void <init>()                                             ;com.covTest.R$drawable
20         ;public void <init>()                                             ;com.covTest.R
23         ;public void <init>()                                             ;com.covTest.R$string
22         ;public void <init>()                                             ;com.covTest.R$attr
25         ;public float add(float, float)                                   ;com.covTest.Calculator
24         ;public void <init>()                                             ;com.covTest.Calculator
27         ;public float mul(float, float)                                   ;com.covTest.Calculator
26         ;public float div(float, float)                                   ;com.covTest.Calculator
29         ;int value()                                                      ;android.annotation.TargetApi
28         ;public float sub(float, float)                                   ;com.covTest.Calculator
```

5.9. Prikaz dokumenta sa informacijama o metodama aplikacije kalkulatora

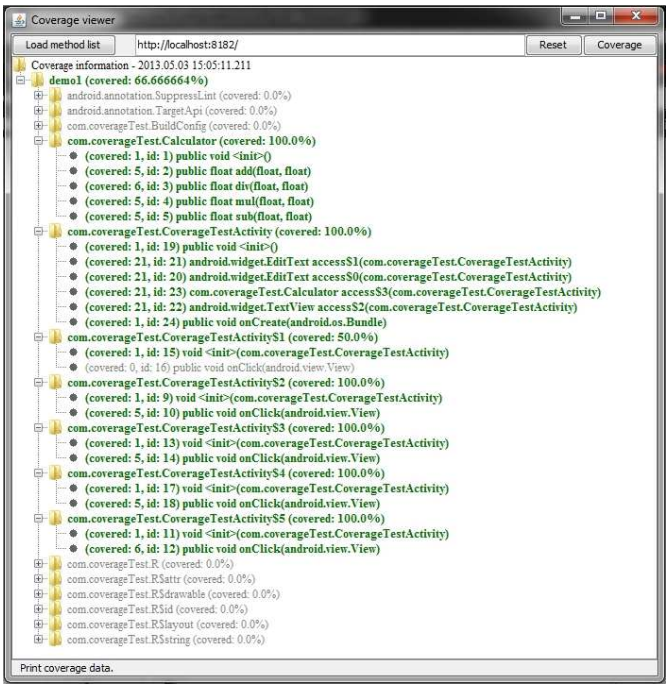
```
InstrumentedAPK
void <init>(com.marvell.videoteam.marvellsetting.MarvellSetting, android.content.Context)
public void onItemClick(android.widget.AdapterView, android.view.View, int, long)
public void <init>(android.content.Context, java.util.List, int, java.lang.String[], int[], int, int)
public android.view.View getView(int, android.view.View, android.view.ViewGroup)
public void <init>()
android.graphics.Bitmap drawableToBitmap(android.graphics.drawable.Drawable)
android.graphics.drawable.Drawable zoomDrawable(android.graphics.drawable.Drawable, int, int)
void <init>(com.marvell.videoteam.marvellsetting.MarvellSetting, android.widget.ListView, android.content.Context)
public void onItemSelected(android.widget.AdapterView, android.view.View, int, long)
public void onNothingSelected(android.widget.AdapterView)
public void <init>()
void <init>(com.marvell.videoteam.marvellsetting.MarvellSetting.ThirdMenuListAdapter, android.widget.SeekBar)
public void onProgressChanged(android.widget.SeekBar, int, boolean)
public void onStartTrackingTouch(android.widget.SeekBar)
public void onStopTrackingTouch(android.widget.SeekBar)
public void <init>()
public void <init>()
void <clinit>()
```

Slika 5.10. Prikazdela dokumenta sa prototipovima metoda sistemske aplikacije

Za prikazane podatke u vezi metoda instrumentalizovane aplikacije, na slici 5.11 se može videti jedan način prikaza pomoću posebno napravljene aplikacije *Coverage Viewer*, uz nešto više detalja, sa podacima predstavljenim na pregledniji način. Program koji je odgovoran za ovaj deo takođe kreira datoteku *result.csv* za dalju obradu. Prikaz ovakvog dokumenta dat je na slici 5.12, sa samo jednom vrstom informacija. Još jedan način prikaza ovih informacija moguć je preko pretraživača, a izgled ovakvog jednog primera dat je na slici 5.13, gde se vidi ukupan broj i broj pozvanih metoda, kao i ovime dobijen procenat pokrivenosti koda. Ovaj procenat se akumulira nakon svakog izvršenog testnog slučaja, što je prikazano formulom (3), za deset testnih slučajeva, čiji su rezultati testiranja prikazani u tabelama 5 i 6. Ovom formulom se zaključuje da se kumulativni procenat pokrivenosti dobija deljenjem broja pokrivenih metoda sa

brojem svih dostupnih metoda. Ukoliko je neka operacija već izvršena tokom ranijih pokretanja, procenat pokrivenosti se neće povećavati.

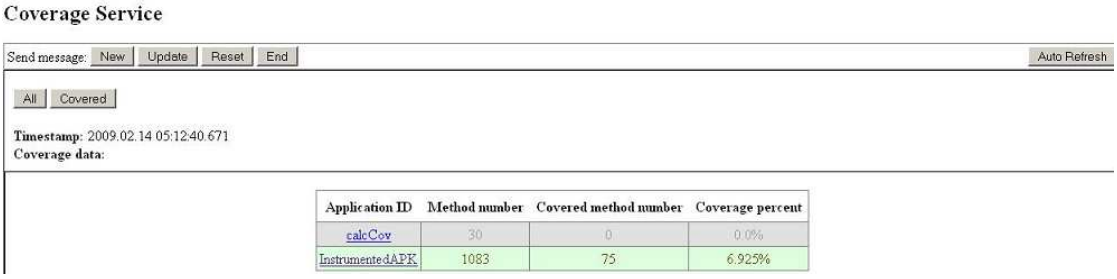
$$\text{cum\_cov} = \frac{\bigcup_{n=1}^{N=10} cov\_met\_num}{\bigcup_{n=1}^{N=10} met\_num} \times 100 \quad (3)$$



Slika 5.11. Prikaz aplikacije sa detaljnim opisom pozivanih metoda

	1	2	3	4	5	6	7
1	Test name	Application name	Trace length	Coverage%	Runtime (ms)	Covered Methods' IDs	Not Covered Methods' IDs
2	D_middle	demo1	170	66.6667	15155	1 2 3 4 5 9 10 11 12 13 14 15 17 18 19 20 21 22 23 24	0 6 7 8 16 25 26 27 28 29
3							

Slika 5.12. Prikaz dela dokumenta *result.csv*



Slika 5.13. Izgled rezultata u pretraživaču

5.4.2 Tabelarni prikaz rezultata

Naredne dve tabele prikazuju rezultate za konkretne testne slučajeve. Tabela 5 daje opis testnih slučajeva, informaciju o tome koje su metode izmenjene, koji je broj pokrivenih metoda za svaki testni slučaj zadat identifikacionom oznakom (eng. *ID*) u prvoj koloni. Ukupan broj metoda za pomenutu aplikaciju kalkulatora je trideset. Još neki od prikazanih podataka su nazivi pokrivenih metoda, vreme izvršavanja, dužine tragova i ishod testiranja (*pass* ili *fail*). Iz ove tabele se mogu videti pomenute informacije za deset testova Android kalkulator aplikacije.

Test Case ID	*Test Case Description*	*Covered Method Number*	*Covered Methods IDs*	*Changed Methods*	*Trace Len.*	*Execution Runtime [ms]*	*Test Outcome*
Sub_1	873-530=343	14/30	M1,…,M13,M14	M4,M5,M8	14	6511	PASS
Sub	957-649=308	15/30	M1,…,M13,M14	M4,M5,M8	14	7654	PASS
Mul_2	616*855=526.68	15/30	M1,…,M13,M15,M18	M5,M18	15	6970	PASS
Mul_1	424*838=355.31	15/30	M1,…,M13,M15,M18	M5,M18	15	5714	FAIL
Mul	43*29=1247	15/30	M1,…,M13,M15,M18	M5,M18	15	7003	FAIL
Div_1	158/278=0.57	14/30	M1,…,M13,M16	M12,M10	14	7367	PASS
Div	428/80=5.35	14/30	M1,…,M13,M16	M12,M10	14	7421	PASS
Add_2	605+915=1.520	14/30	M1,…,M13,M17	M2,M8,M12	14	6007	FAIL
Add_1	739+320=1059	14/30	M1,…,M13,M17	M2,M8,M12	14	6556	PASS
Add	862+74=936	14/30	M1,…,M13,M17	M2,M8,M12	14	6684	PASS

Tabela 5. Opis testnih slučajeva i rezultati

U tabeli 6 je prikazana kumulativna pokrivenost koda, pre primene bilo kakvog kriterijuma prioritizacije, kao i pokrivenost koda u procentima (eng. *Cumulative Coverage*) za svaki od kriterijuma. Svaka druga kolona predstavlja oznaku testnog slučaja. Tamnijim slovima i brojevima su prikazani testni slučajevi koji se mogu uzeti u obzir pri narednom izvršavanju izmenjenog tesnog plana. Ostali testni slučajevi se mogu odbaciti iz razloga što je za njih dobijen isti maksimalni procenat pokrivenosti, tako da predstavljaju višak. Na ovaj način se redukuje broj testnih slučajeva, a time i smanjuje vreme njihovog izvršavanja.

Test Cases Before Prioritization		*Longest Traces*		*Most Coverage*		*Most Additional Coverage*		*Most Failing Tests*	
Test Case	*Cumul. Coverage [%]*	Test Case	Cumul. Coverage [%]	Test Case	*Cumul. Coverage [%]*	Test Case	*Cumul.* Coverage [%]	Test Case	*Cumul.* Coverage [%]
Sub_1	**46.66**	**Mul_2**	**50.00**	**Mul_2**	**50.00**	**Mul_2**	**50.00**	**Mul_1**	**50.00**
Sub	**46.66**	**Mul_1**	**50.00**	**Mul_1**	**50.00**	**Sub_1**	**53.33**	**Mul**	**50.00**
Mul_2	**53.33**	**Mul**	**50.00**	**Mul**	**50.00**	**Div_1**	**56.66**	**Add_2**	**53.33**
Mul_1	**53.33**	**Sub_1**	**53.33**	**Sub_1**	**53.33**	**Add_2**	**60.00**	**Sub_1**	**56.66**
Mul	**53.33**	**Sub**	**53.33**	**Sub**	**53.33**	Mul_1	60.00	**Sub**	**56.66**
Div_1	**56.66**	**Div_1**	**56.66**	**Div_1**	**56.66**	Sub	60.00	**Mul_2**	**56.66**
Div	**56.66**	**Div**	**56.66**	**Div**	**56.66**	Div	60.00	**Div_1**	**60.00**
Add_2	**60.00**	**Add_2**	**60.00**	**Add_2**	**60.00**	Add_1	60.00	Div	60.00
Add_1	60.00	Add_1	60.00	Add_1	60.00	Add	60.00	Add_1	60.00
Add	60.00	Add	60.00	Add	60.00	Mul	60.00	Add	60.00

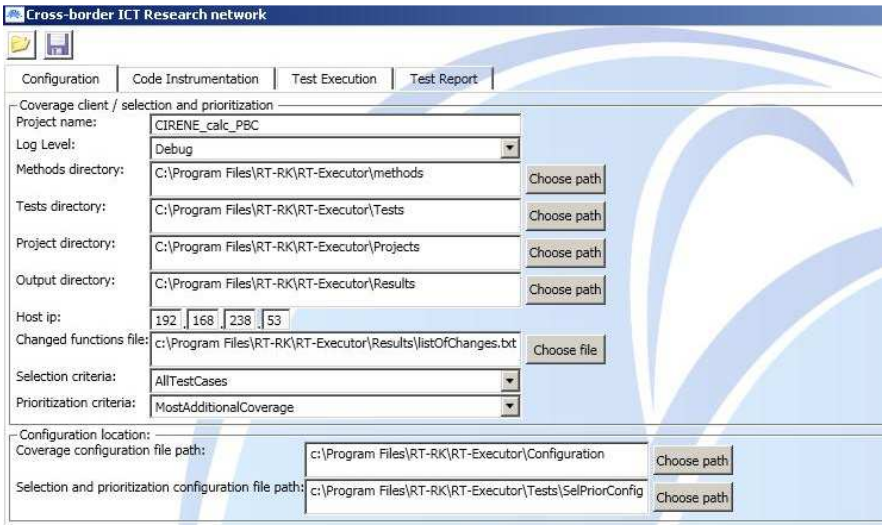
Tabela 6. Testni slučajevi pre i posle prioritizacije

5.4.3 Opis grafičkog okruženja po koracima

Razvijeno je grafičko okruženje (eng. *GUI*) u C# programskom jeziku za automatizaciju celokupnog procesa, odnosno predstavljene metodologije. Dakle, konačni rezultati su predstavljeni na ovaj način, radi prezentacije i eventualne kasnije upotrebe. Koraci pri implementaciji ovakvog okruženja su:

- konfiguracija (podešavanja)
- instrumentacija koda
- testiranje, izbor testnih slučajeva i prioritizacija
- izveštaj o rezultatima

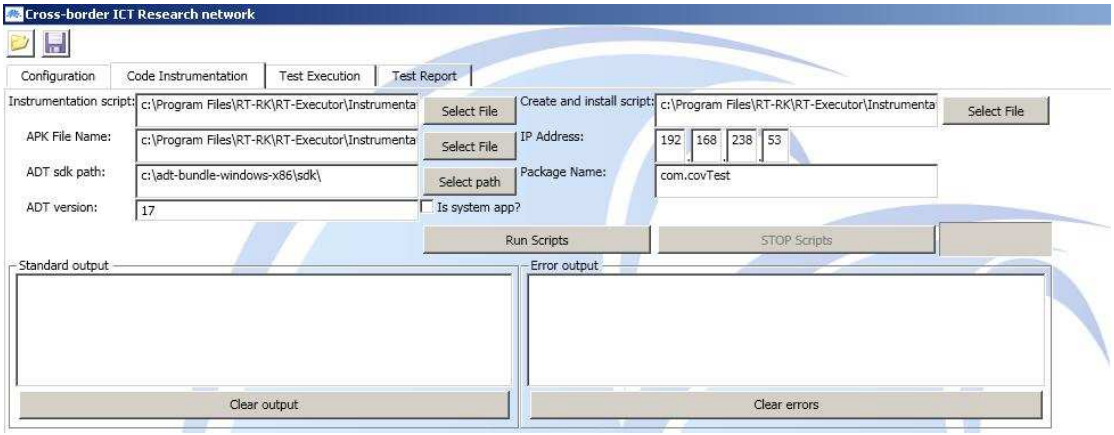
Svaki od navedenih koraka je ukratko opisan u narednim pasusima, a na narednim slikama aplikacije polja su popunjena učitavanjem XML projektnog dokumenta. U prvom koraku se vrše podešavanja u vezi pripreme za naredne korake. To podrazumeva instalaciju Android SDK (eng. *Android software development kit)* i drugih alata kojima se omogućava daljnji tok instalacije aplikacije za instrumentaciju. Na slici 5.14 se može videti koja se sve podešavanja moraju izvršiti pre nego što se pređe na sledeći korak. Zadaju se direktorijumi, IP adresa, kriterijumi izbora i prioritizacije.



Slika 5.14. Podešavanja za automatsko testiranje

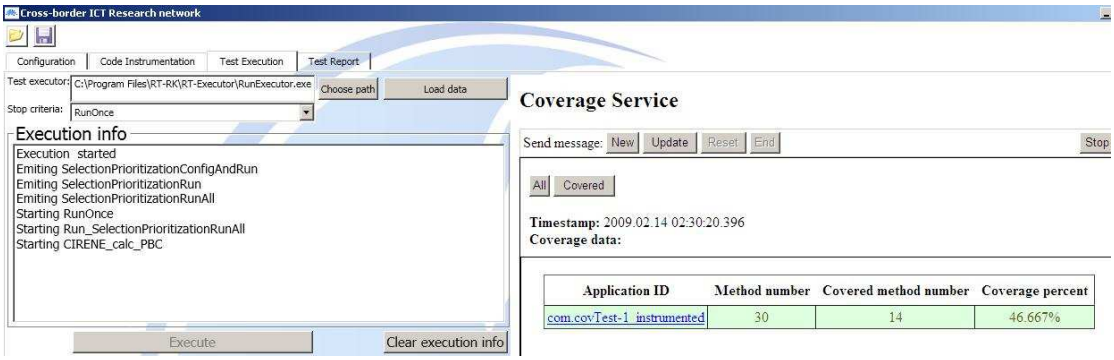
Drugi korak je instalacija odabranih Android aplikacija za testiranje. Potom se vrši njihova instrumentacija pomoću dve osnovne skripte, u okviru kojih se pozivaju druge skripte. Za ove dve skripte se unose parametri koji su specifični za svakog korisnika (kao npr. putanja do SDK alata ili naziv paketa Android aplikacije), a izgled ovog dela aplikacije dat je na slici 5.15. Zbog

izabrane Android platforme moguće je koristiti Java binarnu instrumentaciju. Binarna izvorna datoteka aplikacije (eng. *APK*) se najpre raspakuje. Datoteke koje uključuju programski kod konvertuju se u Java pakete, a sama instrumentacija se izvršava na Java programskom kodu. Izmenjeni paketi se ponovo konvertuju u datoteku formata APK za instalaciju i upotrebu. Instrumentacijom se ubacuju instrukcije na početak svake metode, a time se označava poziv određene metode i ubacuje nova klasa koja beleži i broji ove oznake poziva metoda. Proces instrumentacije nema nikakvog drugog uticaja na izvršavanje aplikacije.



Slika 5.15. Instalacija i instrumentacija Android aplikacija

Nakon uspešne instalacije i instrumentacije Android aplikacije, moguće je preći na njihovo testiranje već pomenutom RT-Executor aplikacijom. U ovom koraku je najpre izvršen odabir testnih slučajeva prema željenom kriterijumu zaustavljanja (eng. *stop criteria*), što se može videti sa slike 5.16. Od tri raspoložive opcije, prikazan kriterijum *RunOnce* pokreće sve testne slučajeve jedanput, vrši izbor i prioritizaciju, a potom se nakon dobijenog ishoda svi testni slučajevi pokreću ponovo i generiše konačni redosled testnih slučajeva i rezultati oba pokretanja. U levom delu prozora je prikazan tok izvršavanja, dok se u desnom delu prozora nalazi ista servisna aplikacija koja je već prikazana u pretraživaču. Na slici 5.17 je dat primer prozora sa prikazom izvršavanja uživo.



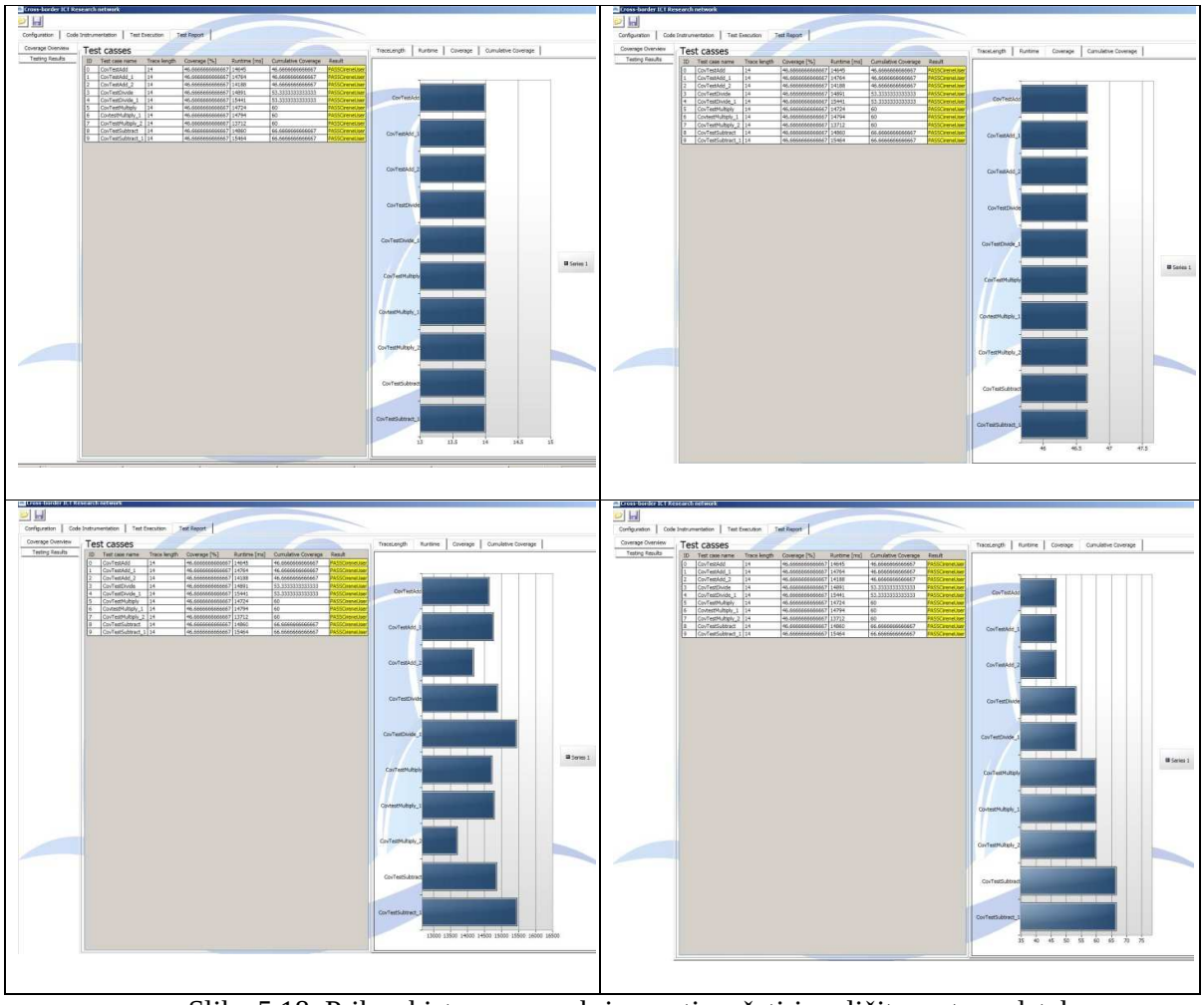
Slika 5.16.Deo korisničke sprege za automatsko izvršavanje testnih slučajeva

Slika 5.17. Prikaz uživo

U poslednjem koraku se prikazuju rezultati testiranja i statistički pregled dobijenih podataka o pokrivenosti koda i tragovima izvršavanja programa. Na slici 5.19 su prikazani histogrami za četiri vrste podataka, sa ishodima testnih slučajeva nakon njihovog prvog pokretanja pri izboru kriterijuma skraćivanja liste testnih slučajeva *RunOnce*. Slika s gornje leve strane se odnosi na prikaz rezultata o tragu izvršavanja, a slika s gornje desne strane odnosi se na statistiku o pokrivenosti metoda. Donja leva slika prikazuje statistiku za vremenski period izvršavanja, a donja desna za kumulativnu pokrivenost metoda. Slika 5.18 daje izgled HTML stranice sa nazivima testnih slučajeva i ishodom izvršavanja.

Nakon drugog pokretanja testnih slučajeva, za predstavljenu kalkulator aplikaciju najbolje rezultate daje kriterijum izbora i prioritizacije sa dodatnim određivanjem prioriteta na osnovu testnih slučajeva koji pozivaju najviše metoda (eng. *MostAdditionalCoverage*). Histogram dobijenih rezultata za ovaj slučaj se vidi na slici 5.20. Odavde se može videti i iz kog razloga su svi ostali testni slučajevi suvišni nakon petog. Naime, za peti, šesti i svaki naredni testni slučaj pokrivenost koda je ista, pa nema potrebe za daljnjim pokretanjem testnih slučajeva (u ovom

slučaju, nakon petog). Tako se lista testnih slučajeva smanjuje, a svako naredno izvršavanje traje kraće.



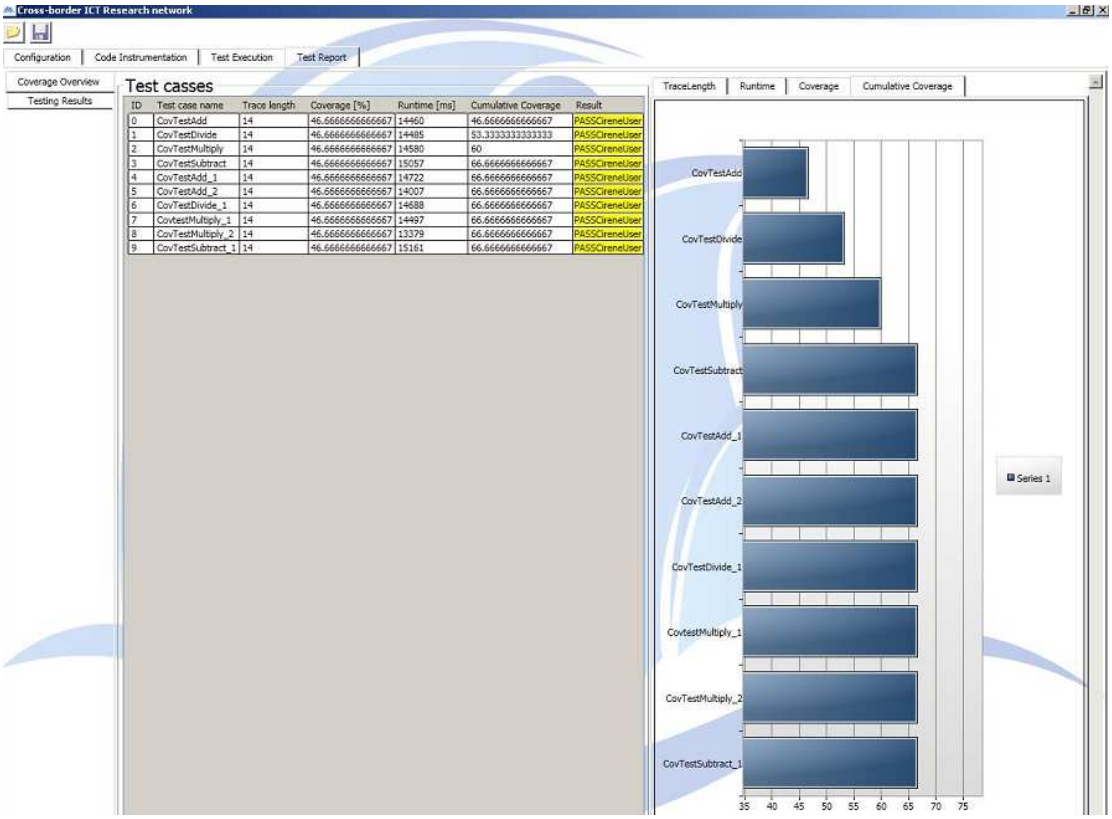
Slika 5.18. Prikaz histograma pokrivenosti za četiri različite vrste podataka

Testing Results

No	Date	Time	Project name	Test case	Build file	Result	Test result comment	Tester	Hardware description
1	2013-06-27	17:17:26	CIRENE_calc_PBC	CovTestAdd	2013	PASS		CireneUser	CireneTestHW
2	2013-06-27	17:17:26	CIRENE_calc_PBC	CovTestDivide	2013	PASS		CireneUser	CireneTestHW
3	2013-06-27	17:17:26	CIRENE_calc_PBC	CovTestMultiply	2013	PASS		CireneUser	CireneTestHW
4	2013-06-27	17:17:26	CIRENE_calc_PBC	CovTestSubtract	2013	PASS		CireneUser	CireneTestHW
5	2013-06-27	17:17:26	CIRENE_calc_PBC	CovTestAdd_1	2013	PASS		CireneUser	CireneTestHW
6	2013-06-27	17:17:26	CIRENE_calc_PBC	CovTestAdd2	2013	PASS		CireneUser	CireneTestHW
7	2013-06-27	17:17:26	CIRENE_calc_PBC	CovTestDivide_1	2013	PASS		CireneUser	CireneTestHW
8	2013-06-27	17:17:26	CIRENE_calc_PBC	CovtestMultiply_1	2013	PASS		CireneUser	CireneTestHW
9	2013-06-27	17:17:26	CIRENE_calc_PBC	CovTestMultiply_2	2013	PASS		CireneUser	CireneTestHW
10	2013-06-27	17:17:26	CIRENE_calc_PBC	CovTestSubtract_1	2013	PASS		CireneUser	CireneTestHW

Tested cases:	10	
Passed:	10	100%
Failed:	0	0%
Inconclusive:	0	0%

Slika 5.19. Ishod izvršavanja testnih slučajeva



Slika 5.20.Prikaz konačnih dobijenih rezultata

6. Zaključak

U ovom radu je dat opis testiranja kombinovanim BBT i WBT tehnika. Ovakva kombinacija se izdvaja po tome što ipak postoji pristup izvornim kodovima programa, za razliku od GBT. Razvijan deo za izbor i prioritizaciju testnih slučajeva je napisan u C++ programskom jeziku, a testni slučajevi su pisani u Python programskom jeziku, a pisane su i BBT skripte za sve instrumentalizovane Android aplikacije. Testni slučajevi se moraju pisati da bi se verifikovale određene osobine razvijenog sistema. Pravljenje potrebnih testnih slučajeva na klasičan način je složen zadatak kojim se troše ljudski resursi. Da bi se povećala efikasnost generisanja i izvršavanja testnih slučajeva, kao i da bi se smanjili troškovi razvoja namenskog sistema, poželjno je automatsko generisanje testnih slučajeva.

BBT grafičko okruženje za automatsko izvršavanje testnih slučajeva obezbeđuje informacije o pokrivenosti programskog koda, pomoću WBT dodatka za kontrolnu aplikaciju. Pomoću servisne aplikacije postoji sprega sa BBT alatom koji preko nje dobija informacije o pokrivenosti koda instrumentalizovane Android aplikacije. Pokrivenost koda se u ovoj metodologiji koristi za računanje dužine traga izvršavanja, izbora i prioritizacije testnih slučajeva i u ostalim delovima metodologije. Dobijene informacije se mogu iskoristiti za poboljšanje kvaliteta testiranja. Na osnovu njih se mogu odabrati i odgovarajući testni slučajevi i sortirati prema potrebnom redosledu, na osnovu date specifikacije. Inženjeri za razvoj sistema i testeri koriste ovakve informacije da provere da li su svi iskazi (ili bar njihova većina) u programu izvršeni bar jedanput. Ovakav pristup dopušta da se izvrše samo neophodni testni slučajevi i povećava se korisnost celokupnog testnog skupa.

Izbor određenih testnih slučajeva se obavlja radi optimizacije raspoloživih resursa, a predstavlja novi pristup u polju automatskog testiranja.Testiranje je vršeno na Marvell STB-u zasnovan na Android platformi. Cilj rada je bio naći vezu između zahteva, testnih slučajeva i

metoda. Ovakve međusobne veze nazivaju se vezama pokrivenosti koda. Prednost metodologije je činjenica da se, nakon odabrane prioritizacije testnih slučajeva, suvišni testni slučajevi izbacuju iz testnog plana za narednu iteraciju testiranja. Nakon ovakvog odbacivanja nepotrebnih testnih slučajeva, sa redukovanim brojem testnih slučajeva se dobija ista pokrivenost programskog koda kao i procenat pokrivenosti dobijen izvršavanjem svih testnih slučajeva. Drugim rečima, ovim pristupom je omogućeno da se sa manjim brojem testnih slučajeva dobije maksimalni procenat pokrivenosti programskog koda i kraće vreme izvršavanja testnih slučajeva. Rezultati ovog rada su u kraćem obliku dati u radu na međunarodnoj konferenciji ECBS-EERC 2013 [11].

7. Literatura

[1] Tingting Yu: *Testing Embedded System Applications*, Universityof Nebraska-Lincoln, 2010.,http://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=1034&context=computerscidiss

[2] Philip Koopman, *Embedded Software Testing*, 18-649 Distributed Embedded Systems 2012., http://www.ece.cmu.edu/~ece649/lectures/09_testing.pdf

[3] Beizer Boris,*Black Box Testing*. New York: John Wiley & Sons, ISBN 0-471-120904-4, 1995.

[4] Kaner, Cem, Jack Falk, and Hung Quoc Nguyen. *Testing Computer Software*, Second Edition. Boston: International Thomson Computer Press. ISBN 1-85032-847-1. John Wiley & Sons, Inc., 1999. ISBN 0-471-35846-0, 1993.

[5] André C. Coulter, *Graybox Software testing Methodology*, Digital Avionics Systems Conference 1999., http://stellar.cleanscape.net/docs_lib/paper_graybox.pdf

[6] Poređenje metoda testiranja, http://www.tutorialspoint.com/software_testing/testing_methods.htm

[7] *STL* tutorijal, http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=standardTemplateLibrary

[8] *STL container* tutorijal, http://www.tenouk.com/download/pdf/Module29.pdf

[9] Opis pravljenja i korišćenja *DLL*-a, http://msdn.microsoft.com/en-us/library/ms235636%28v=vs.80%29.aspx

[10] Opis *apktool* alata, http://code.google.com/p/android-apktool/

[11] Sandra Kukolj, Vladimir Marinković, Bognár Szabolcs, Miroslav Popović, *Selection and Prioritization of Test Cases by Combining White-Box and Black-Box Testing Methods*, ECBS-EERC 2013, 29-30 Aug. 2013.