Pregled osnovnih tehnika testiranja

Seminarski rad u okviru kursa
Metodologija stručnog i naučnog rada
Matematički fakultet

Lazar Mrkela, Ivan Milić
lazar.mrkela@gmail.com, milic92@gmail.com

30. april 2015.

Sažetak

U današnjem informacionom dobu, gde informacione tehnologije imaju uticaj na sve aspekte društva, kvalitet softvera je veoma značajan. Upravo testiranjem možemo poboljšati kvalitet. Kako sve kompleksniji softver zahteva da se sve više vremena razvojnog procesa posveti testiranju, značajno je da svaki učesnik razvoja bude upoznat sa osnovnim konceptima testiranja. Nepostojanje načina testiranja koji bi pronašao sve greške dovodi do pojave različitih tehnika testiranja. Cilj ovog rada je da pruži kratak pregled nekih značajnih tehnika testiranja, kao i da uputi čitaoca na literaturu sa detaljnijim opisom tehnika. Predstavljene su dve grupe tehnika: tehnike zasnovane na modelu crne kutije i tehnike zasnovane na modelu bele kutije.

Sadržaj

1 Uvod

Testiranje predstavlja značajan deo razvojnog ciklusa softvera. Prosečno testiranje čini 40-50% napora za izgradnju softverskog sistema, naravno ovaj procenat zavisi od zahtevane pouzdanosti sistema. Nove metode razvoja softvera, kao i sve kompleksniji softver, povećavaju značaj testiranja [8].

Za razumevanje rada potrebno je objasniti neke osnovne pojmove. Test slučaj (eng. *test case*) je dokument koji definiše ulaze u sistem i očekivane izlaze za te ulaze. Proces testiranja obuhvata procese planiranja, analize i dizajniranja sa ciljem kreiranja test slučajeva, a zatim i njihovo izvršavanje i analizu dobijenih rezultata [3]. Značenje pojma pokrivenosti (eng. *coverage*), koji se često koristi u daljem tekstu, zavisi od konteksta u kojem se javlja. Uopšteno pokrivenost bi predstavljala broj nekih elemenata programa (eng. *items*) ispitanih testovima u odnosu na ukupan broj tih elemenata [13]. Konkretne definicije pokrivenosti su navedene u okviru opisa konkretnih tehnika.

Kako nije moguće ispitati sve test slučajeve, razvijene su razne tehnike za pametan izbor test slučajeva. Tvrđenje ilustrujemo primerom 1.1. U radu je prikazano ukupno 6 tehnika, podeljenih u dve osnovne grupe. Teorijski koncepti tehnika su ilustrovani jednostavnim primerima.

Primer 1.1. Neka je tester dobio zadatak da testira jednostavan kalkulator. Može da krene sa sabiranjem. Prvo unese 1 + 0 i očekuje da je rezultat 1. To je jedan test slučaj. Nastavlja dalje sa 1 + 1 i očekuje 2. Jasno je da takvih slučajeva ima mnogo, a to je samo jedan deo mogućih ulaza. Može zatim da pređe na 2 + 0 i da nastavi taj niz. Posle toga treba testirati sabiranje više brojeva, a zatim i ostale operacije. Međutim, ni tu nije kraj. Treba isprobati i neispravne ulaze, npr. kombinacije svih slova sa tastature. Ispostavlja se da tester ne može na ovaj način da ispuni zadatak, pa čak i da mu računar generiše ulaze. Tester mora neke od mogućih test slučajeva da zanemari i time preuzme rizik propuštanja nekih grešaka [12]. Pri tom izboru se služi gore pomenutim tehnikama.

2 Klasifikacija tehnika

Klasifikacija koja je upotrebljena za prikaz tehnika testiranja u našem radu jeste podela na tehnike zasnovane na modelu:

- **crne kutije** (eng. *black box testing*) - Ne koristi znanje o unutrašnjoj strukturi. Drugi nazivi su funkcionalno testiranje (eng. *functional testing*), testiranje ponašanja (eng. *behavioural testing*), testiranje vođeno podacima (eng. *data driven testing*).
- **bele kutije** (eng. *white box testing*) - Koristi znanje o unutrašnjoj strukturi. Drugi nazivi su strukturalno testiranje (eng. *structural testing*), testiranje vođeno logikom (eng. *logic driven testing*).

Takva podela i detaljan prikaz glavnih tehnika je dat u knjizi [3] koja je poslužila kao osnovni izvor za pisanje ovog rada. Ovoj klasičnoj podeli dodata je i treća vrsta, tehnike zasnovane na modelu sive kutije (eng. *grey box testing*). Ona predstavlja sredinu između modela crne i bele kutije. Kod tehnika tog modela postoji neki uvid u unutrašnju strukturu sistema, ali ne u toj meri kao kod modela bele kutije [6].

Klasifikacija tehnika predstavlja jedan od predmeta istraživanja raznih autora. Pored ovde prikazane, postoje i druge klasifikacije tehnika testiranja [14], od kojih su neke u potpunosti drugačije [7].

3 Model crne kutije

Softver se posmatra kao crna kutija. Ne koristi se znanje o njegovoj unutrašnjoj strukturi, već samo specifikacija zahteva. Ovakav način testiranja se fokusira na ponašanje sistema, posmatrano iz korisničkog ugla. Zadatak testera je da sistemu pruži ulaze, a zatim da proveri izlaze u odnosu na datu specifikaciju [11]. Pokazuje se da za netrivijalne programe nije moguće koristiti tehniku isprobavanja svih mogućih ulaza (eng. *exhaustive input testing*). Zbog toga je cilj tehnika ovog modela da pronađu prihvatljiv broj test slučajeva (tj. kombinacija ulaza) koji će otkriti što više grešaka. Takav cilj se ostvaruje postavljanjem nekih pretpostavki o ispitivanom softveru [10]. Prednost ovakvog načina jeste potpuna razdvojenost programera i testera. Mogućnost testiranja bez znanja o unutrašnjoj strukturi je pogodna za testiranje programa sa kompleksnim kodom, ali nije pogodna za testiranje algoritama [9, 1]. Naredna četiri poglavlja ukratko prikazuju neke važne tehnike zasnovane na prethodno opisanom modelu.

3.1 Klase ekvivalencije

Testiranje pomoću klasa ekvivalencije (eng. *equivalence class testing*) je tehnika koja se koristi da smanji broj test slučajeva na prihvatljiv nivo, pritom održavajući razumnu pokrivenost testovima. Ovde se pokrivenost odnosi na procenat svih mogućih ulaza koji će biti ispitan testovima. Ovu jednostavnu tehniku koriste intuitivno skoro svi testeri, iako oni možda nisu svesni da je to formalna metoda oblikovanja testova.

Klasa ekvivalencije predstavlja skup podataka koji se tretiraju jednako od strane modula ili koji treba da proizvedu isti rezultat. Iz tačke gledišta testiranja, sve vrednosti podataka u okviru jedne klase su *ekvivalentne* svim ostalim vrednostima u okviru te klase.
Konkretno, očekujemo da:

- Ako jedan test slučaj u jednoj klasi ekvivalencije detektuje grešku, *svi* ostali test slučajevi u okviru iste klase ekvivalencije će verovatno detektovati istu grešku.
- Ako jedan test slučaj u jednoj klasi ekvivalencije ne detektuje grešku, *nijedan* drugi test slučaj u okviru iste klase ekvivalencije verovatno neće detektovati grešku.

Koraci za korišćenje testiranja pomoću klasa ekvivalencije su jednostavni. Prvo, identifikujemo klase ekvivalencije. Zatim, pravimo test slučaj za svaku klasu ekvivalencije. Možemo praviti dodatne test slučajeve za svaku klasu ekvivalencije, ali oni retko otkrivaju greške koje prvi slučajevi nisu pronašli.

Primer 3.1. Modul za ljudske resurse određuje kako se obrađuju prijave za zaposlenje bazirane na starosti osobe. Pravila naše organizacije su prikazana u tabeli 1 koja predstavlja specifikaciju zahteva ovog modula. Iz prvog pravila vidimo da za ovaj modul ne moramo da testiramo 0,1,2 ... 14, 15 i 16. Samo jedna vrednost treba biti testirana. Ovde smo pretpostavili da je pravilo implementirano jednom *if* naredbom. Bilo koja vrednost iz ovog opsega je podjednako dobra kao bilo koja druga. Isto važi i za ostale opsege. Opsezi, kao prethodno opisani, se nazivaju klase ekvivalencije. Koristeći pristup klasa ekvivalencije, smanjili smo broj test slučajeva sa 100 (testiranje za svaku godinu starosti) na 4 (testiranje jedne godine starosti za svaku klasu ekvivalencije).

Tabela 1: Pravila organizacije pri zapošljavanju iz primera 3.1

Godine	Pravilo
0-16	Ne zaposliti
16-18	Može se zaposliti samo sa polu-radnim vremenom
18-55	Može se zaposliti sa punim radnim vremenom
55-99	Ne zaposliti

3.2 Granične vrednosti

Testiranje pomoću klasa ekvivalencije je najosnovnija tehnika oblikovanja testova i ona nas vodi do ideje o testiranju graničnih vrednosti (eng. *boundary value testing*), druge ključne tehnike oblikovanja testova koja će biti predstavljena. Ova tehnika se takođe koristi da smanji broj test slučajeva na prihvatljiv nivo, pritom održavajući razumnu pokrivenost testovima.

Testiranje graničnih vrednosti se fokusira na granice zato što se tu krije mnogo grešaka. Iskusni testeri su se sreli sa ovom situacijom mnogo puta. Neiskusni testeri imaju intuitivan osećaj da se greške dešavaju često na granicama. Greška koju programeri često čine je pogrešno kodiranje testova nejednakosti. Primer toga je pisanje $>$ znaka umesto $\geq$ znaka. Najefikasniji način pronalaženja ovih grešaka, bilo u zahtevima, bilo u kodu, je kroz pregledanje. Međutim, koliko god bilo efikasno naše pregledanje, treba da testiramo kod da se uverimo u njegovu ispravnost.

Koraci za korišćenje testiranja graničnih vrednosti su jednostavni. Prvo, identifikujemo klase ekvivalencije. Zatim, identifikujemo granice svake klase ekvivalencije. I na kraju, pravimo test slučaj za svaku graničnu vrednost birajući jednu tačku na granici, jednu tačku ispod granice i jednu tačku iznad granice. *Ispod* i *iznad* su relativni termini i zavise od jedinica vrednosti podataka. Pritom, tačke ispod i iznad granice mogu biti u drugim klasama ekvivalencije i nema razloga da dupliramo test.

Primer 3.2. U prethodnom poglavlju, zadata su pravila koja određuju kako obrađivati prijave za zaposlenje na osnovu starosti osobe koja se prijavljuje. Primećuje se problem na granicama svake klase. Starost *16* je uključena u dve različite klase ekvivalencije (kao što su i 18 i 55). Prvo pravilo kaže da ne zapošljavamo osobe sa 16 godina. Drugo pravilo kaže da se osobe sa 16 godina mogu zaposliti sa polu-radnim vremenom. Kako su pravila trebala da izgledaju prikazano je u tabeli 2. Test slučajevi u ovom primeru su naredne vrednosti na granici ili blizu granice: {-1, 0, 1}; {15, 16, 17}; {17, 18, 19}; {54, 55, 56}; {98, 99, 100}.

Tabela 2: Ispravljena pravila organizacije pri zapošljavanju iz primera 3.2

Godine	Pravilo
0-15	Ne zaposliti
16-17	Može se zaposliti samo sa polu-radnim vremenom
18-54	Može se zaposliti sa punim radnim vremenom
55-99	Ne zaposliti

3.3 Tabele odlučivanja

Izrada tabela odlučivanja (eng. *decision table*) je tehnika za prikaz složenih poslovnih pravila u lako čitljivom obliku [5] pomoću koje se mogu napraviti i test slučajevi. Prvu grupu redova tabele čine uslovi nad ulazom, a drugu moguće akcije. Kolone tabele predstavljaju pravila koja jedinstvenoj kombinaciji uslova dodeljuju odgovarajuće akcije. Uslovi pravila mogu biti binarni ili sa više od dve vrednosti. Iz prvih se može direktno izvesti tačno jedan test slučaj, dok iz drugih može više njih. Izbor različitih test slučajeva iz jednog pravila može se vršiti u kombinaciji sa drugim tehnikama, kao što su klase ekvivalencije ili granične vrednosti [2]. Kada imamo više pravila kod kojih akcija ne zavisi od vrednosti nekog uslova možemo ih spojiti u jedno pravilo (eng. *table collapsing*). Takav uslov u novom pravilu označavamo sa '-' i nazivamo nebitnim (eng. *don't care*). Struktura tabele se najbolje vidi na konkretnom primeru.

Primer 3.3. Klijent zahteva isplatu gotovine na bankomatu neke banke. Sistem treba da odluči da li će da odobri isplatu. Odlučivanje vrši pomoću podataka o sredstvima na računu i o dozvoljenom minusu. Način odlučivanja je prikazan tabelom odlučivanja 3. Iz drugog pravila može se izvesti test slučaj tako što se za ulaz uzme da korisnik nema dovoljno sredstava na računu i da mu je dozvoljen minus. Zatim se izlaz iz programa poredi sa očekivanom akcijom, a to je da je isplata odobrena. Prvo pravilo je dobijeno spajanjem dva pravila. Kada ima dovoljno sredstava na računu, nezavisno od toga da li je minus dozvoljen ili ne, isplata se odobrava [4].

Tabela 3: Odlučivanje pri zahtevu za isplatu gotovine predstavljeno tabelom odlučivanja

	Pravilo 1	**Pravilo 2**	**Pravilo 3**
Uslovi			
Dovoljno sredstava na računu	Da	Ne	Ne
Dozvoljen minus	-	Da	Ne
Akcije			
Isplata odobrena	Da	Da	Ne

3.4 Dijagrami stanja

Dijagram stanja (eng. *state-transition diagram*) kompaktno opisuje kompleksne zahteve sistema i njegov način interakcije sa spoljašnjim svetom. Primenjuje se kod sistema čije akcije zavise od akcija izvršenih u prošlosti i koji reaguju na spoljašnje događaje. Osnovna struktura dijagrama je prikazana na slici 1 i čine je:

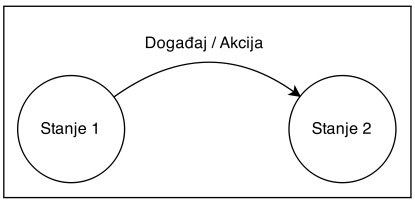
Stanje Čuva znanje o prošlim događajima i definiše reakciju na buduće.

Prelaz Promena iz jednog stanja u drugo.

Događaj Nešto izvan sistema što preko interfejsa izaziva prelaz.

Akcija Operacija sistema izazvana prelazom.

U svakom trenutku sistem se nalazi u nekom od konačno mnogo stanja i čeka na neki događaj. Kombinacija stanja i događaja određuje stanje u koje sistem prelazi. Pri prelasku sistem može da izvrši još neku akciju, obično pravljenje nekih izlaza. Ovakav sistem se može modelovati

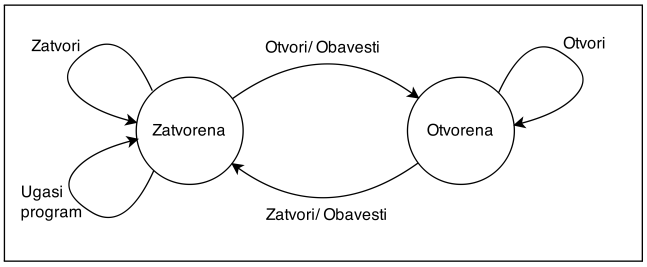


Slika 1: Osnovna struktura dijagrama stanja.

konačnim automatom (eng. *finite state machine*) [15]. Dijagram stanja je jedan od načina prikaza takvog modela. Tabele stanja (eng. *state transition tables*) su drugi način. Osnovna prednost tabela stanja jeste njihov sistematični pristup, prikazuju sve moguće kombinacije stanja i događaja. Takvim pristupom mogu da se uoče situacije u kojima ponašanje sistema nije definisano, što može da spreči pojavu grešaka [2].

Test slučajeve možemo da pravimo obilaskom, jer dijagram stanja predstavlja vrstu usmerenog grafa [15]. Pri pravljenju skupova test slučajeva možemo zahtevati različite nivoe pokrivenosti, pri čemu se pravi kompromis između pokrivenosti i količine testova. Primer dobrog komprimisa je skup testova koji omogućava da se svaki prelaz ispita bar jednom. Pored toga, možemo zahtevati da se svako stanje ili svaka putanja kroz dijagram obiđu bar jednom . Kod tabela stanja, iz svakog reda se može direktno izvesti jedan test slučaj.

Primer 3.4. Predstavljene koncepte ilustrujemo konkretnim primerom. Posmatramo softverski sistem za maloprodaju koji ima ugrađenu opciju za otvaranje i zatvaranje (fioke) kase. Dijagram stanja kase u takvom sistemu je prikazan na slici 2 , a odgovarajuća tebala stanja je data tabelom 4. Problematičan događaj nastaje kada korisnik sistema želi da ugasi program. Opasnost od ostavljanja otvorene kase nije navedena u dijagramu stanja, ali jeste u tabeli stanja. Moguća rešenja su da sistem upozori korisnika i spreči zatvaranje programa ili da automatski zatvori kasu. Jedan primer test slučaja koji aktivira svaki prelaz datog dijagrama bar jednom dat je sledećim nizom naredbi: otvori, otvori, zatvori, zatvori, ugasi program. Pri izvršavanju navedenih naredbi proverava se da li sistem reaguje u skladu sa zadatim zahtevima.



Slika 2: Dijagram stanja kase softverskog sistema za maloprodaju opisanog u primeru 3.4.

Tabela 4: Tabela stanja koja odgovara dijagramu stanja prikazanom na slici 2 i opisanom u primeru 3.4

Trenutno stanje	**Događaj**	**Akcija**	**Naredno stanje**
Zatvorena	otvori	obavesti	Otvorena
Zatvorena	zatvori	-	Zatvorena
Zatvorena	ugasi program	-	Zatvorena
Otvorena	otvori	-	Otvorena
Otvorena	zatvori	obavesti	Zatvorena
Otvorena	ugasi program	-	Nedefinisano

4 Model bele kutije

Ovakav način testiranja podrazumeva znanje o unutrašnjoj strukturi softvera. Tester kreira test slučajeve na osnovu izučavanja implementacije [11]. Analogno ispitivanju svih kombinacija ulaza kod tehnika zasnovanih na modelu crne kutije, može se zahtevati ispitivanje svih putanja kroz program. Međutim, pokazuje se da je takav pristup nepraktičan (nekad i nemoguć) [10]. Zbog toga tehnike nastoje da omoguće kreiranje praktično prihvatljivog broja test slučajeva, ali i da obezbede visok nivo pokrivenosti. Pri detaljnom izučavanju izvornog koda mogu se uočiti razne skrivene greške, ali ovakav pristup podiže cenu razvoja jer zahteva visoko stručnog testera [9]. U naredna dva poglavlja prikazane su tehnike ovog modela grupisane prema tome da li se posmatra kontrola toka ili tok podataka.

4.1 Kontrola toka

Grafovi kontrole toka (eng. *control flow graphs*) predstavljaju osnovu testiranja zasnovanog na kontroli toka (eng. *control flow testing*). Ovi grafovi dokumentuju kontrolnu strukturu modula. Moduli koda se konvertuju u grafove, putanje kroz grafove se analiziraju i test slučajevi se kreiraju na osnovu tih analiza. Na primer, naredni kod je predstavljen grafom kontrole toka na slici 3.

```
1     if (a > 0) { x = x + 1; }
2     if (b == 3) { y = 0; }
```

Jednu putanju kroz graf možemo da ispitamo zadavanjem test slučaja sa ulazima a = 6 i b = 3. Takav test slučaj bi pokrio i sve naredbe, ali to ne znači da je on dovoljan. Primetimo da postoji još mogućih putanja (npr. za a = -1 i b = 4) i da ih navedeni test slučaj ne pokriva.

U testiranju zasnovanom na kontroli toka, definišemo pokrivenost na više različitih nivoa u zavisnosti od elemenata unutrašnje strukture programa čija se pokrivenost ispituje:

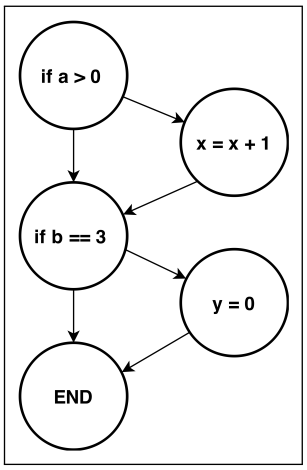
Pokrivenost funkcija Mera poziva svih funkcija

Pokrivenost naredbi Mera izvršavanja svih naredbi

Pokrivenost grana Mera prolaska kroz sve grane

Pokrivenost uslova Mera ispitivanja svih izraza na tačno/netačno

Pokrivenost putanja Mera prolaska kroz svaku moguću putanju



Slika 3: Graf kontrole toka prethodno razmatranog koda

Pre početka testiranja, odgovarajući nivo pokrivenosti treba biti izabran.

Svaki softverski modul sadrži mali broj unikatnih, nezavisnih putanja kroz sebe. Te putanje se nazivaju bazne putanje (eng. *basis path*). Struktura koda može biti testirana pokretanjem testova kroz ovaj mali broj putanja. Beskonačan broj različitih putanja se u stvari svodi na upotrebu i ponovnu upotrebljivost baznih putanja [2]. Testiranje baznih putanja (eng. *basis path testing*), odnosno struktuirano testiranje (eng. *structured testing*) je sačinjeno od sledećih koraka:

1. Izvođenje grafa kontrole toka iz softverskog modula
2. Izračunavanje ciklomatične kompleksnosti grafa (**C**)
3. Odabir skupa **C** baznih putanja
4. Pravljenje test slučaja za svaku baznu putanju
5. Izvršavanje ovih testova

Ciklomatična kompleksnost (eng. *cyclomatic complexity*) je najmanji broj baznih putanja koje generišu sve moguće putanje kroz modul. Ciklomatična kompleksnost grafa se izračunava pomoću jednačine 1.

$$C = grane - čvorovi + 2 \tag{1}$$

Kreiranjem i izvršavanjem **C** test slučajeva, automatski se garantuje i pokrivenost grana i pokrivenost naredbi zato što skup baznih putanja pokriva sve grane i čvorove grafa kontrole toka.

4.2 Tok podataka

Promenljive koje sadrže vrednosti podataka imaju definisani životni ciklus. One se kreiraju, koriste i uništavaju. U nekim programskim jezicima, kreiranje i uništavanje se vrši automatski. Promenljiva se kreira prvi put kad joj se dodeli vrednost i uništava kada se završi izvršavanje programa. U ostalim jezicima, kreiranje je formalno. Promenljive se eksplicitno deklarišu naredbama kao što su:

```
1     int x; // x se kreira kao integer
2     string y; // y se kreira kao string
```

Ove deklaracije se obično nalaze u okviru bloka koda koji počinje sa otvorenom zagradom '{' i završava sa zatvorenom zagradom '}'. Promenljive definisane u okviru bloka se kreiraju kada se njihove definicije izvrše i automatski se uništavaju na kraju bloka. Ovo se zove opseg važenja (eng. *scope*) neke promenljive. Na primer:

```
1    {                    // počinje spoljašnji blok
2     int x;              // x je definisano u spoljašnjem bloku
3     ...;                // moguć pristup promenljivoj x
4         {               // počinje unutrašnji blok
5         int y;          // y je definisano u unutrašnjem bloku
6         ...;            // moguć pristup promenljivoj x i y
7         }               // y je automatski uništeno
8     ...;                // moguć pristup samo promenljivoj x
9    }                    // x je automatski uništeno
```

Promenljive mogu da se koriste u računanju i u proverama uslova, ali je podjednako bitno u oba slučaja da je dodeljena vrednost promenljivoj pre nego što je korišćena.

Graf toka podataka (eng. *data flow graph*) je sličan grafu kontrole toka po tome što prikazuje tok obrade kroz modul. Dodatno, graf toka podataka prikazuje definicije, korišćenje i uništavanje svake promenljive modula. Ovi dijagrami se konstruišu i utvrđuje se da su definiši-iskoristi-uništi (eng. *define-use-kill*) obrasci odgovarajući. Prvo, izvršava se statični test dijagrama (eng. *static test*). Pod *statični*, misli se na ispitivanje dijagrama kroz pregledanje. Drugo, izvršava se dinamični test na modulu (eng. *dynamic test*). Pod *dinamični*, misli se na konstruisanje i izvršavanje test slučajeva.

U okviru statičnog testiranja toka podataka, na dijagramu kontrole toka označavaju se definiši-iskoristi-uništi informacije za svaku promenljivu koja se koristi u modulu. Za svaku promenljivu se posebno ispituju obrasci duž putanja kontrole toka. U okviru dinamičnog testiranja toka podataka, pretpostavlja se da je graf kontrole toka ispravan s obzirom da je testiranje toka podataka bazirano na kontroli toka modula. Za svako *definisanje* pronađeno je svako njegovo *korišćenje* i za svako *korišćenje* pronađeno je odgovarajuće *definisanje*. Da bi se ovo uradilo, potrebno je numerisati putanje kroz modul. Nakon što se izlistaju sve putanje, kao kod testiranja zasnovanog na kontroli toka, za svaku promenljivu pravi se bar jedan test slučaj da se pokrije svaki definiši-iskoristi par.

5 Zaključak

Rad omogućava početnicima da razumeju osnovne koncepte tehnika testiranja. Sažeto su predstavljene dve osnovne grupe tehnika testiranja. Kroz primere je prikazana suština pojedinih tehnika. Prikazana klasifikacija tehnika nije kompletna, ali je čitalac upućen na dodatne izvore. Kombinovanjem prikazanih tehnika moguće je poboljšati testiranje. Budući rad bi mogao da uključi sažet prikaz još nekih važnih tehnika modela crne kutije (npr. testiranje parova (eng. *pairwise testing*) i pogađanje grešaka (eng. *error guessing*)) i detaljniji opis tehnika modela bele kutije. Postoje razni alati koji mogu testeru da olakšaju primenu prikazanih tehnika. KLEE i LDRA su dva primera za model bele kutije. KLEE generiše izveštaj koji prikazuje broj putanja kroz zadatu funkciju, broj instrukcija kao i test slučajeve za svaku putanju. LDRA je softverski paket koji pruža statičku analizu koda, analizu pokrivenosti koda kao i analizu kontrole toka / toka podataka, kroz detaljne izveštaje i grafike i time dopunjuje mnoga razvojna okruženja.

Literatura

[1] A. Anitha. Overview of software testing techniques and metrics. *International Journal of Advanced Research in Computer and Communication Engineering*, 2(12), 2013.

[2] Rex Black and Jamie L. Mitchell. *Advanced Software Testing - Vol. 3: Guide to the ISTQB Advanced Certification As an Advanced Technical Test Analyst*. Rocky Nook, 1st edition, 2011.

[3] Lee Copeland. *A Practitioner's Guide to Software Test Design*. Artech House, Inc., Norwood, MA, USA, 2003.

[4] Mark Debono. A guide to Using Decision Tables, 2012. Work available online at: http://reqtest.com/requirements-blog/a-guide-to-using-decision-tables/.

[5] J.B. Dixit. *Structured System Analysis and Design*. Laxmi Publications Pvt Limited, 2007.

[6] Irena Jovanović. Software testing methods and techniques. *The IPSI BgD Transactions on Internet Research*, 30, 2008.

[7] Cem Kaner, James Bach, and Bret Pettichord. *Lessons Learned in Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[8] Lu Luo. Software testing techniques. *Institute for software research international Carnegie mellon university Pittsburgh, PA*, 15232(1-19):19, 2001.

[9] Farmeena Khan Mohd Ehmer Khan. A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. *International Journal of Advanced Computer Science and Applications(IJACSA)*, 3(6), 2012.

[10] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[11] 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.

[12] R. Patton. *Software Testing*. Sams, 2001.

[13] S.Kumar. ISTQB Certification Exam Study Material. Available online at: http://istqbexamcertification.com/.

[14] Sira Vegas, Natalia Juristo, and Victor R Basili. Maturing software engineering knowledge through classifications: A case study on unit testing techniques. *Software Engineering, IEEE Transactions on*, 35(4):551–565, 2009.

[15] F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme. *Modeling Software with Finite State Machines: A Practical Approach*. CRC Press, 2006.