Razvoj okruženja za automatsko testiranje i dinamičku analizu TV aplikacije Elektronskog Programskog Vodiča

Turalija, Matko

SVEUČILIŠTE JOSIPA JURAJA STROSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

Razvoj okruženja za automatsko testiranje i dinamičku analizu TV aplikacije Elektronskog Programskog Vodiča

Diplomski rad

Matko Turalija

Osijek 2016.

SADRŽAJ

1. UVOD 1
2. OSNOVE TESTIRANJA 3
2.1. Osnove testiranja 3
2.2. Hijerarhija pisanja testova 5
2.3. Agile development 7
2.4. Metode testiranja 9
3. OTKRIVANJE CURENJA MEMORIJE I PROGRAMSKO PROFILIRANJE – VALGRIND 13
3.1. Valgrind 15
3.2. Memcheck 19
3.3. Cachegrind 22
3.4. Callgrind 25
3.5. DRD 27
3.6. Massif 30
4. UNIT TESTIRANJE I KONTINUIRANA INTEGRACIJA 34
4.1. CMocka 35
4.2. Kontinuirana integracija 44
4.2.1. Aplikacija za vođenje projekta 45
4.2.2. Jenkins 45
4.2.3. Automatizacija testiranja 45
5. REZULTATI TESTIRANJA I ANALIZA 47
5.1.Rezultati programskog profiliranja 47
5.1.1.Memcheck 47
5.1.2.Callgrind 48
5.1.3.Cachegrind 49
5.1.4.DRD 49
5.1.5.Massif 49
5.2. Rezultati unit testiranja 50
5.3. Zaključak na temelju rezultata testiranja 55
6. ZAKLJUČAK 56
LITERATURA 58
SAŽETAK 59
ABSTRACT 60
ŽIVOTOPIS 61

1. UVOD

Moderni TV prijemnici/STB uređaji, osim prikaza televizijskog sadržaja, omogućuju prikaz Elektronskog Programskog Vodiča (engl. *Electronic Programming Guide)* koji daje detaljne informacije o sadržaju koji se trenutno prikazuje na zaslonu i o ostalim sadržajima koji se nalaze na drugim televizijskim kanalima. Korisnici očekuju ispravnost rada uređaja i njegovih komponenti, uključujući Elektronski Programski Vodič. Prije puštanja proizvoda u slobodnu prodaju potrebno je provjeriti ispravnost istih. Provjera ispravnosti rada programske podrške Elektronskog Programskog Vodiča obrađuje se ovim diplomskim radom.

U okviru diplomskog rada potrebno je izraditi okruženje za automatsko ispitivanje ispravnosti funkcioniranja i dinamičku analizu programskih modula TV aplikacije Elektronskog Programskog Vodiča (EPG), napisanog u C programskom jeziku. Zahtjevi koje okruženje treba ispuniti su:

- Potpuno automatsko testiranje najmanjih jedinica programa koje mogu biti testirane izolacijom jedinice od ostalih dijelova programa i provjerom jesu li se ponašale očekivano za različite ulazne scenarije (engl. *unit testing*),
- Generiranje rezultata *unit* testiranja u formi *XML* (engl. *Extensible Markup Language*) datoteka,
- Mjerenje potrošnje memorije pojedinih funkcija programa te cijelog programa,
- Provjera postoje li pogreške u upravljanju memorijom (otkrivanje curenja memorije, upotreba neinicijalizirane memorije, ...) ,
- Provođenje programskog profiliranja (analiza zauzetosti memorije, broj poziva funkcija, kompleksnost programa, vrijeme izvršavanja pojedinih funkcija) u cilju optimizacije programa,
- Generiranje izvještaja dinamičke analize (potrošnja memorije, pogreške u rukovanju memorijom, rezultati programskog profiliranja),
- Povezivanje implementiranog okruženja sa sustavom za *Continuous Integration* ili sustavom namijenjenog za automatizaciju testiranja (automatsko izvršavanje testova/provedba analize, generiranje rezultata/izvještaja, slanje rezultata/izvještaja korisnicima).

Ciljna platforma je uređaj s Linux operativnim sustavom. Okruženje se razvija korištenjem biblioteka i alata otvorenog programskog koda koji su namijenjeni za *unit* testiranje, otkrivanje curenja memorije i profiliranje programa. Za otkrivanje curenja memorije korišteni su alati instrumentacijskog programskog okvira za dinamičku analizu,

Valgrind, dok je za potrebe *unit* testiranja korištena biblioteka otvorenog programskog koda, CMocka.

Struktura ovog diplomskog rada podijeljena je u pet poglavlja. U prvom poglavlju dan je uvod u temu i opisan je zadatak diplomskog rada. Drugo poglavlje daje uvod u testiranje i opisuje značajke i korištenje alata Valgrind. Trećim poglavljem opisano je *unit* testiranje i sustav automatizacije testiranja. Četvrto poglavlje daje prikaz rezultata testiranja. U petom poglavlju dan je zaključak izrađenog diplomskog rada.

2. OSNOVE TESTIRANJA

2.1. Osnove testiranja

Testiranje se može opisati kao proces vrjednovanja sustava ili njegovih komponenti u namjeri da se sazna zadovoljava li određene zahtjeve. Testiranjem se identificiraju greške, propusti ili neispunjeni zahtjevi. Proces testiranja vrlo je bitan pri izradi konkretnoga projekta jer je potrebno potvrditi i osigurati garanciju da sustav radi kako je zahtijevano. Razvoj programske podrške također zahtijeva testiranje. U životnom ciklusu razvoja programske podrške prikazanoj na slici 2.1. stavka testiranje predstavlja bitnu ulogu, stoga se često dijeli u više faza. Faze testiranja često prate usporedno razvoj programske podrške tj. odvijaju se tijekom razvoja programske podrške.

Sl. 2.1. *Životni ciklus razvoja programske podrške*

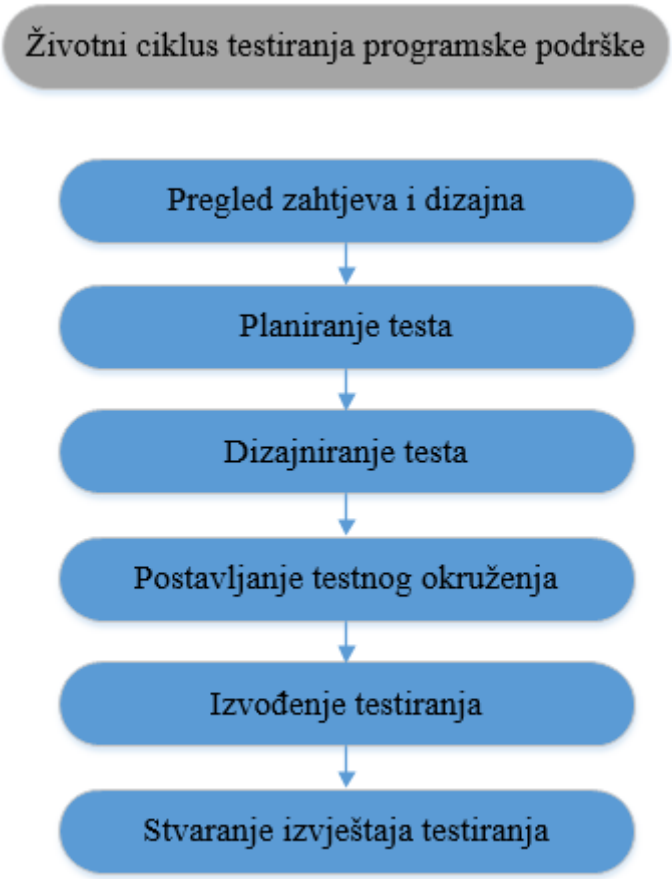
Tablica 2.1. prikazuje osnovne smjernice kojih bi se trebalo pridržavati pri testiranju programske podrške[1].

Tab. 2.1. *Osnovne smjernice pri testiranju*[1]

Broj smjernice	Smjernica
1	Neophodno je definirati očekivani izlazni rezultat za test
2	Programer koji je pisao programski kod bi trebao izbjegavati testiranje istog
3	Tvrtka koja izrađuje program ne bi trebala testirati isti
4	Svaki proces testiranja bi trebao imati opširnu analizu rezultata
5	Testovi bi trebali biti napisani za neispravne i neočekivane ulazne vrijednosti, kao i za ispravne i očekivane ulazne vrijednosti
6	Pregledavanje ne radi li program ono što bi trebao je samo pola posla. Potrebno je pregledati radi li program ono što ne bi trebao
7	Izbjegavati pisanje testova koje nije moguće ažurirati
8	Ne planirati testove na temelju pretpostavke da neće biti grešaka
9	Uzeti u obzir da je vjerojatnost postojanja grešaka u dijelu koda gdje su već pronađene greške je veća nego u dijelovima koda gdje nisu pronađene
10	Prilikom odabira osobe koja će provoditi testiranje potrebno je imati na umu da je testiranje vrlo kreativan i intelektualno zahtjevan zadatak

STLC (engl. *Software Testing Life Cycle*) opisuje tijek ciklusa testiranja programske podrške i prikazan je na slici 2.2. Postoje različite vrste STLC-a, no ovdje će biti opisan STLC korišten pri izradi testova za potrebe ovog diplomskog rada. Testiranje započinje definiranjem zahtjeva u životnom ciklusu razvoja programske podrške. Tijekom faze izrade dizajna, testeri provjeravaju i odlučuju koje aspekte dizajna je moguće testirati i uz koje parametre. Nakon toga slijedi izrada testnog plana u kojoj se opisuje testna strategija, sam testni plan i prijedlog platforme za testiranje. Preciznije, izrada testnog plana započinje popisom stavki koje će biti testirane i koje neće biti testirane, zatim se opisuje pristup koji će biti korišten, utvrđuje se kriterij prolaznosti testa. Slijedi popis zadataka testiranja i opis okruženja za testiranje, te popis zaduženja pojedinog člana testne skupine. Potrebno je napraviti vremenski raspored

izrade testova i njihovih izvođenja. Ovdje se definiraju kontrolne točke za provjeru tijeka testiranja. Na kraju plana potrebno je navesti moguće rizike i načine ublažavanja istih, ako se pojave, te je potrebno definirati način zapisivanja testne dokumentacije i rezultata testiranja.



Sl. 2.2. *Životni ciklus testiranja programske podrške*

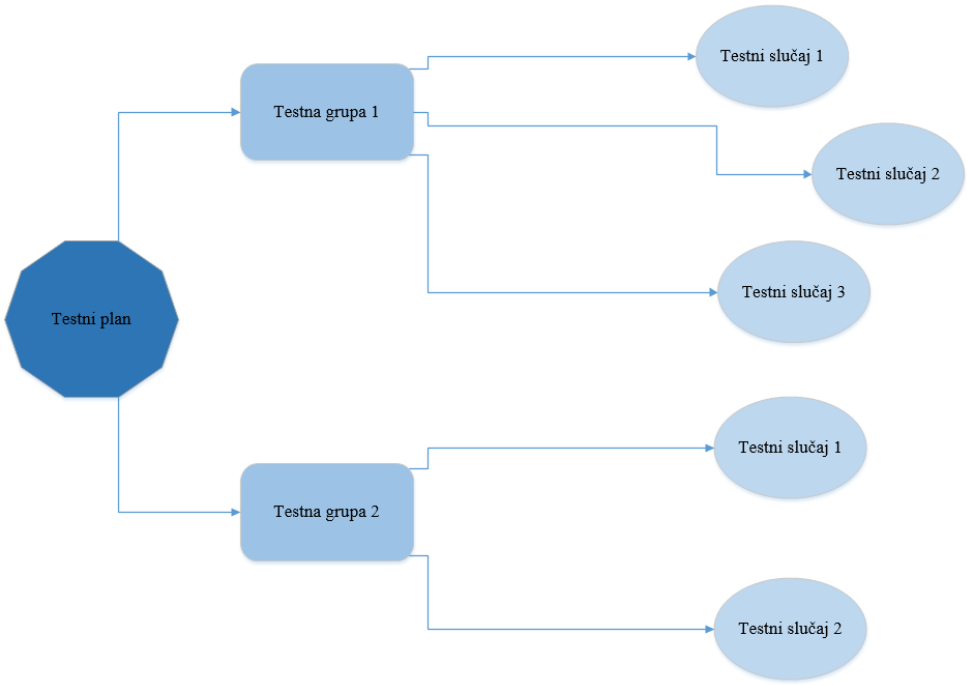
2.2. Hijerarhija pisanja testova

Pri pisanju testova slijedi se hijerarhija izrade gdje se prvo izrađuje testni plan i bira testno okruženje, zatim se izrađuju testne grupe i na kraju testni slučajevi. Testni plan je dokument koji definira opseg i aktivnosti testiranja. On je osnova formalnog testiranja u tijeku izrade projekta. Plan treba biti sažet i vrlo određen. Potrebno je navesti detalje pojedinih testova, testnih okruženja i slično. Pod tim se smatra da je potrebno navesti verzije koje se koriste pri testnom okruženju, načini pokretanja testova itd. Opisi testova u obliku odlomaka

ili članaka bi se trebali izbjegavati, dok bi se trebalo koristiti što više tabličnih prikaza ili popisa. Česte promjene u razvoju programske podrške uzrokuju česte promjene testova, stoga je potrebno pregledavati testni plan i ažurirati ga. Pri pisanju testova potrebno je imati na umu pet bitnih stavki:

1. scenarij testa
2. koraci testa
3. parametri
4. očekivani rezultat
5. stvarni rezultat

Testno okruženje se sastoji od elemenata koji omogućuju izvođenje testova uz konfiguriranu programsku podršku, sklopovlje i mrežu. Ono mora što vjernije oponašati okruženje u kojem će se program izvršavati. Pri dizajniranju testnog okruženja potrebno je provjeriti potrebne licence za programsku podršku u okviru okruženja, provjeriti je li potrebno spremati stanja u slučaju rezerve, identificirati operativni sustav i ostale komponente na kojemu se nalazi testno okruženje, te potvrditi konfiguraciju mreže. Testna grupa je skupina testova kojima se testira programska podrška, često sadrži detaljna objašnjenja za svaku skupinu testova i informacije o konfiguraciji sustava koja se treba koristiti za testiranje. Slika 2.3. prikazuje hijerarhiju pri pravljenju testova.



Sl. 2.3. *Prikaz hijerarhije pri pravljenju testova*

Tester, koji je ujedno i programer, proučava implementaciju programskog koda te određuje sve moguće (ispravne i neispravne) i nemoguće ulazne vrijednosti. Na temelju njih određuje izlazne vrijednosti, koje se također vide iz programskog koda. Pri pripremi testnih slučajeva tester mora voditi računa o zahtjevima, funkcionalnostima, detaljnom dizajnu i sigurnosnim zahtjevima. Tada se kreće u fazu planiranja testnih slučajeva u kojoj se pravi plan i popis svih testnih slučajeva za pojedine dijelove programskog koda. Nakon toga slijedi faza implementacije testnih slučajeva i provođenje istih. Posljednja faza je dobivanje rezultata i kreiranje konačnog izvještaja testiranja.

2.3. Agile development

Postoji nekoliko modela testiranja programske podrške koji su na raspolaganju te je potrebno odabrati odgovarajući model u konkretnom slučaju. Najčešće korišteni pristupi razvoja programske podrške su: *Waterfall model, Prototyping, Incremental development, Iterative and incremental development, Spiral development* i *Agile development*. Pri izradi testova i ovog diplomskog rada korišten je pristup *agile development*, stoga će on detaljnije biti objašnjen.

Agile development je pristup razvoja programske podrške u kojem je vrlo bitna suradnja, prilagođavanje i fleksibilnost programera, bilo u skupini razvoja programske podrške, skupini razvoja testiranja programske podrške ili voditelja projekta. Najbitnije četiri stavke ovog pristupa su[2]:

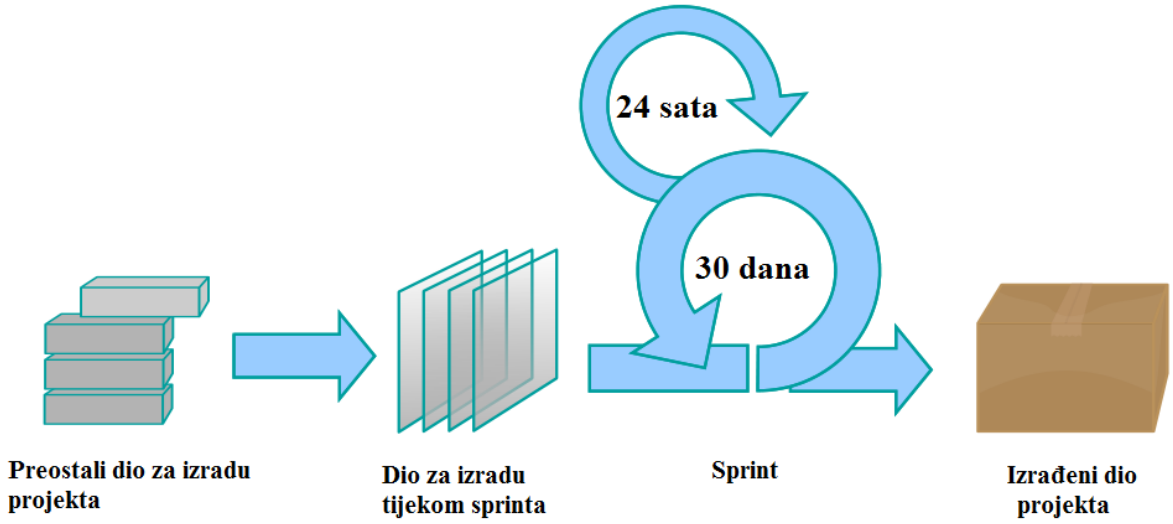
1. *Individuals and interactions:* samoorganizacija i motivacija su bitni, kao i interakcija poput programiranja u paru ili rad u istoj zgradi, sobi,
2. *Working software:* programska podrška koja radi korisnija je nego prezentacije ili dokumentacija pri sastanku s klijentima,
3. *Customer collaboration:* zahtjevi često nisu potpuni na početku razvoja programske podrške, stoga je bitna neprestana komunikacija s klijentima,
4. *Responding to change:* bitno je biti u mogućnosti brzo reagirati i prilagoditi se promjenama u zahtjevima ili razvoju programske podrške, te nastaviti nesmetano s radom.

Promjene u zahtjevima treba smatrati dobrodošlima, čak i u kasnoj fazi razvoja. Bitno je da je klijent zadovoljan, te treba biti u toku sa razvojem programske podrške. Programska podrška koja radi treba biti dostavljana često, najbolje na tjednoj bazi i na temelju nje može se zaključiti u kojoj mjeri je projekt završen. Izrada projekta se temelji na motiviranim

pojedincima, čestom komunikacijom (licem u lice, ako je moguće) između voditelja projekta i programera. Najbolji zahtjevi i dizajn dolaze iz samoorganiziranih skupina jer one neprestano obraćaju pažnju na tehničku izvedbu i dobar dizajn. Postizanje veće učinkovitosti temelji se na dobroj komunikaciji i ocjenjivanju svoga rada i doprinosa.

Agile development temelji se na mogućnosti promjene zahtijeva klijenata bilo kada tijekom razvoja programske podrške. Ovakav način ubrzava razvoj i rad na projektu jer razvojna skupina radi kao jedno kako bi dostigla zajednički cilj. Osnovni princip ovakvog pristupa naziva se sprint. Sprint je vremenski ograničen napor tijekom izrade projekta. Vremensko trajanje sprinta određuje se prije početka istoga, te najčešće traje između jednog tjedna i jednog mjeseca. Svaki sprint završava pregledom i pogledom unatrag. Ovakva analiza omogućuje izvlačenje zaključaka koji mogu biti korisni pri izradi sljedećeg sprinta. Izrada sprinta sastoji se od sljedećih stavki:

- opseg posla koji bi se trebao obaviti,
- zapis zaostalih stavki koje je vjerojatno moguće izraditi,
- priprema stavki i detalja koje se trebaju izraditi tijekom sprinta te
- postavljanje vremenskog ograničenja za sprint.



Sl. 2.4. *Grafički prikaz agilnog razvoja programske podrške*

Grafički prikaz *agile development*-a programske podrške vidljiv je na slici 2.4. Na kraju izrade glasanjem se odlučuje koje stavke projekta će biti izvršene tijekom sprinta. Često se tijekom sprinta rade i dnevni sprintovi[3].

Kroz godine razvoja programske podrške mijenjao se cilj testiranja pa tako postoje:

- testiranja orijentirana otklanjanju neispravnosti,
- testiranja orijentirana demonstriranju rada,
- testiranja orijentirana uništavanju,
- testiranja orijentirana procjeni te
- testiranja orijentirana sprječavanju kvara.

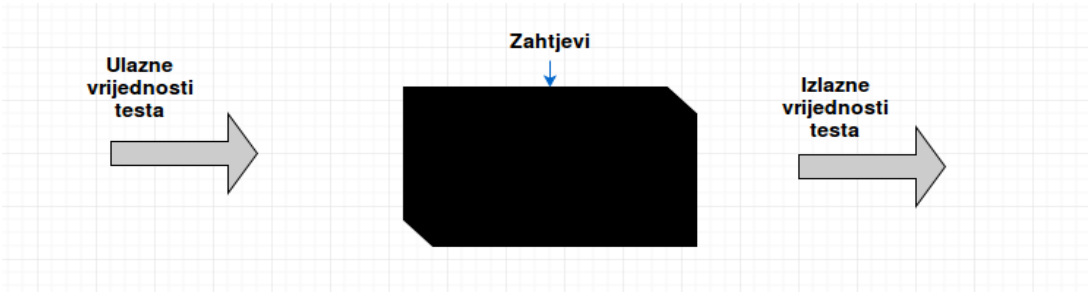
2.4. Metode testiranja

Metode testiranja mogu biti statičke ili dinamičke. Statičke metode testiranja obuhvaćaju osvrte, opise i preglede na napisani programski kod, dok se stvarno pokretanje programskog koda uz dane testne slučajeve smatra dinamičkim testiranjem. Statičko testiranje se često izvodi implicitno putem alata za pisanje i uređivanje programskog koda, te programskog prevoditelja. Dinamičko testiranje se radi tijekom izvođenja programa. Može započeti prije nego što je programski kod u potpunosti gotov. Glavne tehnike dinamičkog testiranja su korištenje *stub* funkcija/*driver*-a ili putem okruženja za otklanjanje kvarova. Statičko testiranje obuhvaća provjeravanje, dok dinamičko potvrđivanje rada programskog koda. Uzmu li se u obzir oba načina testiranja, dobivaju se potpuni rezultati kvalitete programske podrške. Usporedba statičkog i dinamičkog testiranja prikazana je tablicom 2.2.

Tab. 2.2. *Usporedba statičkog i dinamičkog testiranja*

Statičko testiranje	**Dinamičko testiranje**
programska podrška se ne koristi	programska podrška mora se programski prevesti i pokrenuti
nije detaljno, većinom provjerava algoritam i sintaksu	analiza odziva sustava na varijable u programskom kodu, te promjene koje se vrše nad njima
Osvrti, opisi i pregledi programskog koda	*unit* testiranja, integracijska testiranja, testiranja sustava i testiranja prihvatljivosti
Provjeravanje (engl. *Verification*)	Potvrđivanje (engl. *Validation*)

Black-box testing, poznato kao testiranje ponašanja, je tehnika testiranja programske podrške pri kojem dizajn ili implementacija testirane programske podrške nisu poznati testeru. Tehnika je dobila ime jer u očima testera programska podrška koju testira izgleda kao crna kutija (engl. *black box*).



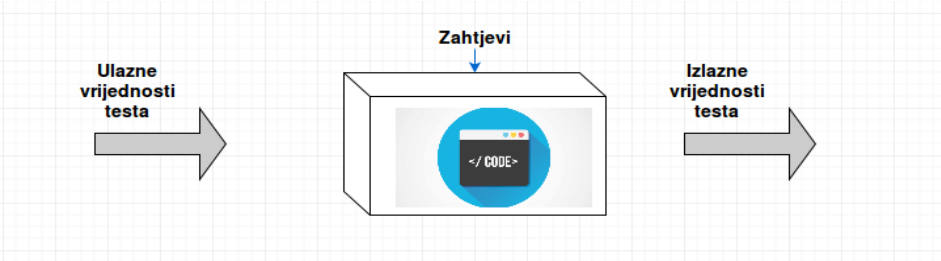
Sl. 2.5. *Black-box testing*

Cilj ovakvog testiranja je pronaći:

- funkcije koje nedostaju ili neispravne funkcije,
- pogreške u sučelju,
- pogreške u strukturi podataka,
- greške u ponašanju ili izvođenju te
- greške inicijalizacije i terminiranja.

Black-box testiranje je pogodno za *Integration Testing, System Testing i Acceptance Testing.* Što je veća razina programske podrške i složenost, to je korisniji ovakav način testiranja. Prednosti ovog pristupa su što nije potrebno poznavanje programiranja ili implementacije i testovi se mogu napraviti čim su gotove specifikacije, dok su nedostatci nemogućnost potpunog testiranja i problematičnost pokrivanja svih mogućnosti rada programa zbog ponekad nejasnih specifikacija. Slika 2.5. prikazuje ideju *black-box* testiranja.

White-box testing je tehnika testiranja programske podrške u kojoj se na temelju napisanog programskog koda ili programske logike stvaraju testovi. Prilikom ove tehnike testiranja potrebna je unutarnja perspektiva sustava, uvid u kod programske podrške. Opis *white-box* testiranja grafički je prikazan slikom 2.6.



Sl. 2.6. *White-box testing*

Postoje tri načina korištenja *white-box testing*-a:

- Pokrivanje jedinica koje omogućuju izvođenje određene funkcije u programskom kodu
- Pokrivanje grananja programskog koda (testiranje if, for, while, do-while, switch-case petlji)
- Pokrivanje svih mogućih prolazaka programskog koda vodeći računa o izvođenim funkcijama i grananjima

```
System Testing = (Broj izvedenih naredbi) / (Ukupni broj naredbi) x 100%

Branch Testing = (Broj testiranih grananja) / (Ukupni broj grananja) x 100%

Path Testing = (Broj izvedenih prolazaka kroz kod) / (Ukupni broj prolazaka kroz kod) x 100%
```

Sl. 2.7. *Prikaz ocjenjivanja pojedinog načina testiranja*

Slika 2.7. prikazuje način ocjenjivanja pri *white-box* testiranju. Prednosti koje pruža *white-box testing* su otkrivanje grešaka koje se mogu sakriti u kodu tijekom njegovog pisanja, otkrivanje beskorisnog koda ili drugih poteškoća koje nisu u skladu s najboljom programerskom praksom i zahtijeva od programera da pažljivo razmotri i odabere implementaciju programske logike. Nedostatci su što vremenski i materijalno zahtijeva puno resursa, postoji mogućnost da se pojedine linije koda slučajno izostave i potrebno je dobro znanje i poznavanje programskog jezika kako bi se *white-box testing* dobro izveo.

Ovakva tehnika testiranja primjenjiva je na tri razine testiranja:

1. *Unit testing*
2. *Integration testing*
3. *System testing*

Pri razini *Unit testing*-a testiranjem se osigurava da programski kod radi kako je zamišljeno, prije nego što se dogodi integracija pojedinih jedinica u cjelokupni sustav. Ovime se postiže otkrivanje greška u ranoj fazi razvoja programske podrške. *Unit testing* koristi se u ovom diplomskom radu i detaljnije je opisan u četvrtom poglavlju. Pri razini *Integration testing*-a testovi se pišu kako bi se testirala suradnja i interakcija pojedinih jedinica unutar sustava. Provjerava se ispravnost ponašanja programskog koda u okruženju u kojem je moguće testirati ponašanja sustava poznata programeru. Pri razini *System testing*-a testira se kompletan sustav i svrha ovog testiranja je procijeniti podudarnost rada sustava i prethodno određenih zahtijeva. Tablica 2.3. prikazuje usporedbu *black-box* i *white-box* testiranja prema razini testiranja, odgovornosti, razini znanja programiranja, poznavanju implementacije i osnova za stvaranje testa.

Tab. 2.3. *Usporedba Black-box i White-box testiranja*

Kriterij	***Black-box testing***	***White-box testing***
Razina testiranja	Više razine testiranja - *Acceptance Testing, System Testing*	Niže razine testiranja - *Unit Testing, Integration Testing*
Odgovornost	Testeri programske podrške	Programeri
Znanje programiranja	Nije potrebno	Potrebno
Poznavanje implementacije	Nije potrebno	Potrebno
Osnova za stvaranje testa	Popis zahtjeva ili specifikacije	Detaljan dizajn

3. OTKRIVANJE CURENJA MEMORIJE I PROGRAMSKO PROFILIRANJE – VALGRIND

Pri izgradnji programskog sustava nastoji se optimizirati njegovu izradu, rad i korištenje. Korisnici od sustava očekuju pouzdanost i raspoloživost pri njegovom korištenju. Pouzdanost se može definirati kao vjerojatnost da će programski sustav uspješno raditi unutar određenog vremenskom razdoblja uz prethodno definirane uvjete rada. Raspoloživost se može definirati kao sposobnost programskog sustava da uspješno obavlja rad u određenom trenutku ili vremenskog razdoblju. U sklopu ovog diplomskog rada izrađena su pojedina testiranja koja omogućuju provjeru pouzdanosti i raspoloživosti sustava TV aplikacije Elektronskog Programskog Vodiča. Testiranjem sustava omogućeno je provjeriti rad sustava i na temelju istoga napraviti analizu rezultata.

Pri testiranju TV aplikacije elektronskog programskog vodiča potrebno je imati na umu da se radi o ugradbenom računalnom sustavu koji radi u stvarnome vremenu. Ove dvije stavke dodatno zahtijevaju da testiranja budu kvalitetno i ispravno provedena. Pod ugradbenim računalnim sustavima se često smatraju mala računala s odgovarajućom programskom podrškom. Najčešći problemi takvih sustava su ograničenje resursa. Takvi sustavi obično imaju malu memoriju, relativno spore procesore, mala veličina programske riječi i sl. Kako bi se izbjegli problemi zbog ograničenja sustava, potrebno je napraviti efikasna rješenja. Programska podrška se najčešće piše u programskim jezicima niske razine (asembler ili C) i koriste specijalizirane računalne arhitekture poput programabilnih DSP-a (engl. *Digital Signal Processing*) ili FPGA-a (engl. *Field-Programmable Gate Array*). Prilikom pisanja programske podrške također nailazi se na velika ograničenja poput nemogućnosti korištenja objektno-orijentiranih programskih jezika, automatiziranog zauzimanja i oslobađanja memorije, sakupljanja otpada nakon završetka programa, korištenja procesora s velikom virtualnom memorijom itd. Razlog tomu je što ugradbeni računalni sustavi imaju postavljene vrlo visoke standarde pouzdanosti za razliku od programske podrške koja se koristi svaki dani. Kvarovi u ugradbenim računalnim sustavima mogu prouzročiti po život opasne situacije, npr. u sustavima aviona ili vojnim sustavima. Iz tog razloga je ispravno testiranje programske podrške za ugradbene računalne sustave od velike važnosti. Na primjer, konkurentnost izvođenja niti u programu često ne postiže dovoljnu pouzdanost, stoga se najčešće koriste semafori i muteksi. Takva tehnika programiranja dovodi ponekad do zastoja, gdje se svi dijelovi programa ne mogu više izvoditi. Krivo korištenje ili nekorištenje muteksa i semafora je izuzetno teško otkriti testiranjem. Program se može izvoditi godinama dok se ne otkrije

pogreška u dizajnu sustava. Pouzdanost rada ovakvih sustava je gotovo najbitnija stavka cijeloga projekta. Boljim pregledavanjem koda, boljim i opširnijim testiranjem, boljim planiranjem razvoja programske podrške greške se mogu smanjiti za veliki postotak. Najveći izazov kod ugradbenih računalnih sustava je izraditi slojeve apstrakcije koje imaju čovjeku razumljiviji programski jezici, a da su ujedno pravovremeni i omogućuju pouzdanu implementaciju višenitnosti[4].

Ugradbeni računalni sustavi imaju malu količinu RAM memorije direktno na pločici i stoga njihova memorija nije proširiva. Ponekada postoji mogućnost proširenja memorije, no ta mogućnost često je skuplja nego cijeli ugradbeni računalni sustav, zato nije isplativa. Programer pri izvedbi programske podrške mora posebnu pozornost posvetiti iskoristivosti memorije. Često se koriste razne tehnike pri rješavanju problema male količine memorije poput:

- ponovno programsko prevođenje sa opcijom -Os (optimizacija za veličinu),
- pronalaženje i odbacivanje beskorisnog koda,
- refaktoriranje dijelova koda koje se ponavljaju u općenitu rutinu (makro funkcija),
- zamjena RAM-a za memoriju za program te
- ugradnja malog interpretera.

Računala koja se svakodnevno koriste imaju MMU (engl. *memory management unit*) koji se brine o virtualnoj memoriji, štiti dijelove memorije koje koristi OS od ostalih program. Ponekad je korisno izraditi MMU koja će rukovati memorijom koja je na raspolaganju programu koji se pokreće na ugradbenom računalnom sustavu.

TV aplikacija Elektronskog Programskog Vodiča mora biti u mogućnosti raditi u stvarnom vremenu. Računalni sustavi stvarnog vremena su sustavi koji na zadani ulaz daju određeni izlaz u prethodno određenom vremenskom razmaku. Ponekad vremenski razmaci mogu biti jako mali pa se čini da sustav vrlo brzo radi, no oni mogu biti puno duži. Pri izradi ovakvih sustava bitno je voditi računa da se zahtjevi, posebno oni na koje korisnik čeka, izvršavaju u stvarnom vremenu, te se radi toga koriste različiti principi kako bi to bilo omogućeno:

- izvodi se mali broj zadataka, kako bi se osiguralo da će uvijek biti izvršeni prije krajnjeg roka,
- neke funkcije se odbacuju ili se smanjuje njihovo vrijeme izvođenja kada sustav ne može ispuniti zadani rok,
- konstantno se prate ulazne vrijednosti sustava i

- prate se resursi sustava i moguće je prekinuti pozadinske procese kako bi izvođenje bilo u zadanom vremenskom roku[5].

U ovom radu ciljna platforma za testiranje TV aplikacije Elektronskog Programskog Vodiča je uređaj s Linux operativnim sustavom, dok je implementacija programskog koda uređaja izvršena putem C programskog jezika. Programski jezik C je viši programski jezik opće namjene. On je proceduralan, što znači da se izvršava liniju po liniju, omogućava pisanje efikasnih programa, omogućava rad „blizu“ sklopovlja i može se izvršavati na nizu različitih računalnih platformi. C je široko prihvaćen programski jezik pri izradi sustava jer se izvodi gotovo jednako brzo kao i asemblerski jezik, dok je čovjeku čitljiviji i jednostavniji. Neki od primjera korištenja C-a su: operacijski sustavi, programski prevoditelji, tekstualni uređivači, pogonski programi za mrežu itd. Pisanje efikasnih programa se omogućuje izradom kvalitetnog programskog koda bez grešaka. Postupak izrade kvalitetnog programskog koda sastoji se od korištenja dobre programske prakse za programiranje i pojedini programski jezik, te testiranja programskog koda kako bi se moguće greške otklonile ili kako bi se programski kod optimizirao. Moguće greške ili mane koje se mogu javiti kod programskog koda su curenje memorije, sporo izvršavanje programa, problemi pri višenitnom programiranju, loša iskoristivnost *heap*-a i *stack*-a i sl. Otkrivanje grešaka i mana programskog koda u ovom diplomskom radu rađeno je putem alata instrumentacijskog programskog okvira za dinamičku analizu naziva Valgrind. Alati ovog programskog okvira koji su korišteni pri izradi ovog diplomskog rada omogućuju programsko profiliranje i otkrivanje curenja memorije te će biti detaljnije opisani u ovom poglavlju, kao i sam Valgrind.

3.1. Valgrind

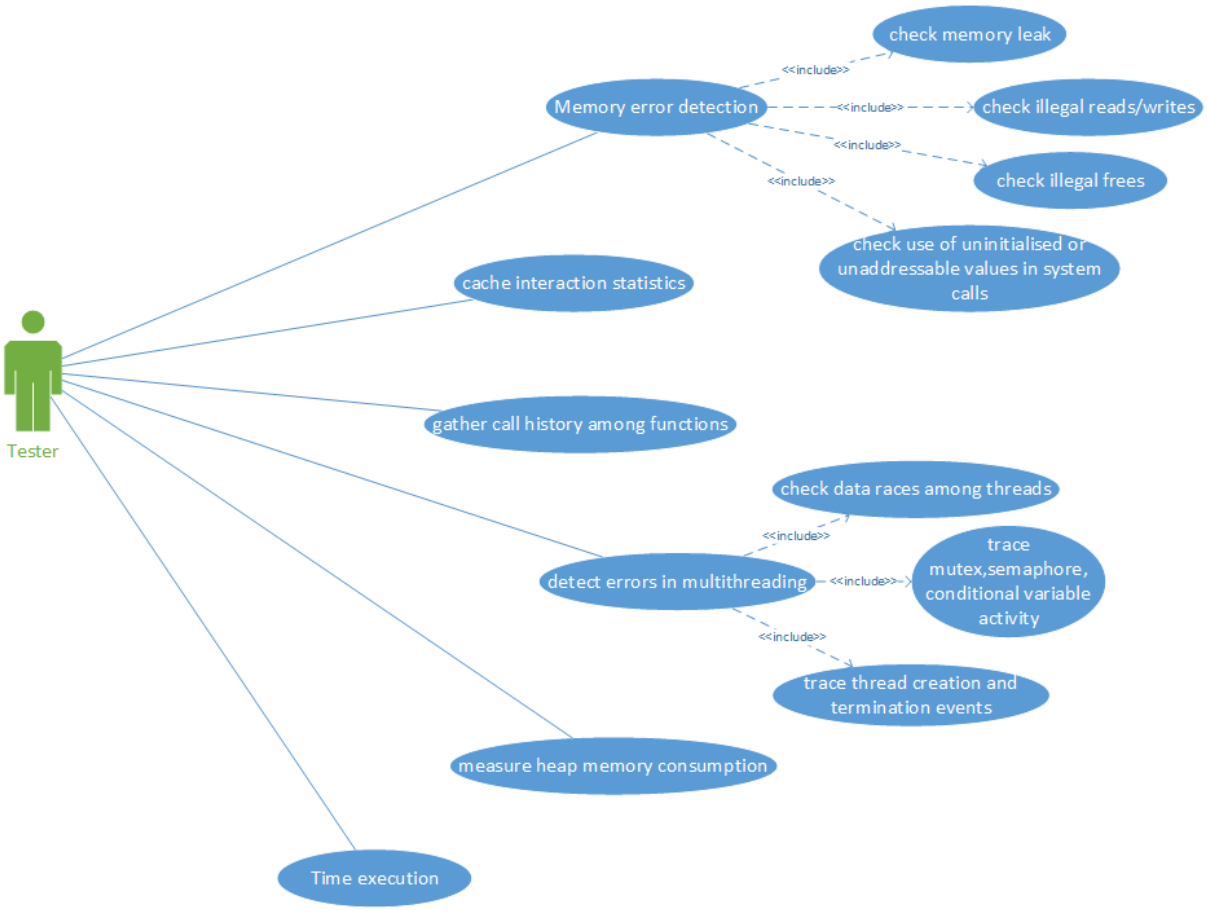
Valgrind je skup alata instrumentacijskog programskog okvira za dinamičku analizu koda. Nije dostupan za sve platforme. Trenutno je dostupan za sljedeće platforme: x86/Linux, AMD64/Linux, PPC32/Linux, PPC64/Linux, PPC64LE/Linux, S390X/Linux, ARM/Linux, ARM64/Linux, MIPS32/Linux, MIPS64/Linux, X86/Solaris, AMD64/Solaris, X86/illumos, AMD64/illumos, X86/Darwin (10.10, 10.11), AMD64/Darwin (10.10, 10.11), ARM/Android, ARM64/Android, MIPS32/Android, X86/Android, x86/FreeBSD, x86/NetBSD, sparcv9/Solaris. Korištenje na Windows platformama nije omogućeno jer se bitno razlikuju od Linux platformi, stoga bi prerađivanje Valgrinda za Windows bio potpuno novi alat. Uporabom Valgrinda omogućeno je programsko profiliranje, otkrivanje curenja memorije, bilježenje vremena izvršavanja programskog koda, interakcija sa predmemorijom (engl. *cache*), analiza višenitnog programiranja, te iskorištenje *heap*-a i *stack*-a. Koristi se u širokoj

uporabi i radi sa svim programskim jezicima, ali ciljana skupina su C i C++. Rad sa svim programskim jezicima je omogućen na način da radi izravno nad binarnim datotekama programskog koda. Uporaba Valgrinda je vrlo jednostavna jer koristi dinamičku binarnu instrumentaciju, te se pokreće putem naredbi iz komandne linije. Upravo to je razlog što se programski kod ne mora ponovno programsko prevoditi (engl. *compile*), povezivati (engl. *link*) i sl. Valgrind je proširiv stoga mu je moguće dodavati samostalno izrađene alate. Niz alata ponuđeni su kao standard pri korištenju Valgrinda:

- Memcheck,
- Cachegrind,
- Callgrind,
- Helgrind,
- DRD,
- Massif,
- SGCheck,
- DHAT i
- BBV

Osnovni alat koji se pokreće pri korištenju Valgrinda je Memcheck, dok je za korištenje drugih alata potrebno navesti njihov naziv kao argument naredbi --tool .

Valgrind je napravljen na način da bude što nenametljiviji i on radi izravno sa izvršnim datotekama programa. Preuzima kontrolu nad programom prije nego što se on počne izvršavati. Informacije o pregledavanju koda se iščitavaju iz izvršnih datoteka i njima dodanih biblioteka kako bi se poruke o pogreškama mogle izraziti u odnosu na lokaciju u izvornom programskom kodu. Program se izvršava na umjetnoj središnjoj jedinici za obradu (engl. *Central Processing Unit*) koju pruža jezgra Valgrinda. Programski kod koji se izvršava nadopunjuje se instrumentacijskim kodom alata Valgrinda koji se koristi. Broj linija instrumentacijskog koda koji se dodaje, ovisi o alatu koji se koristi. Bitno je napomenuti kako se time usporava izvođenje izvornog programskog koda, ponekada i do 100 puta.



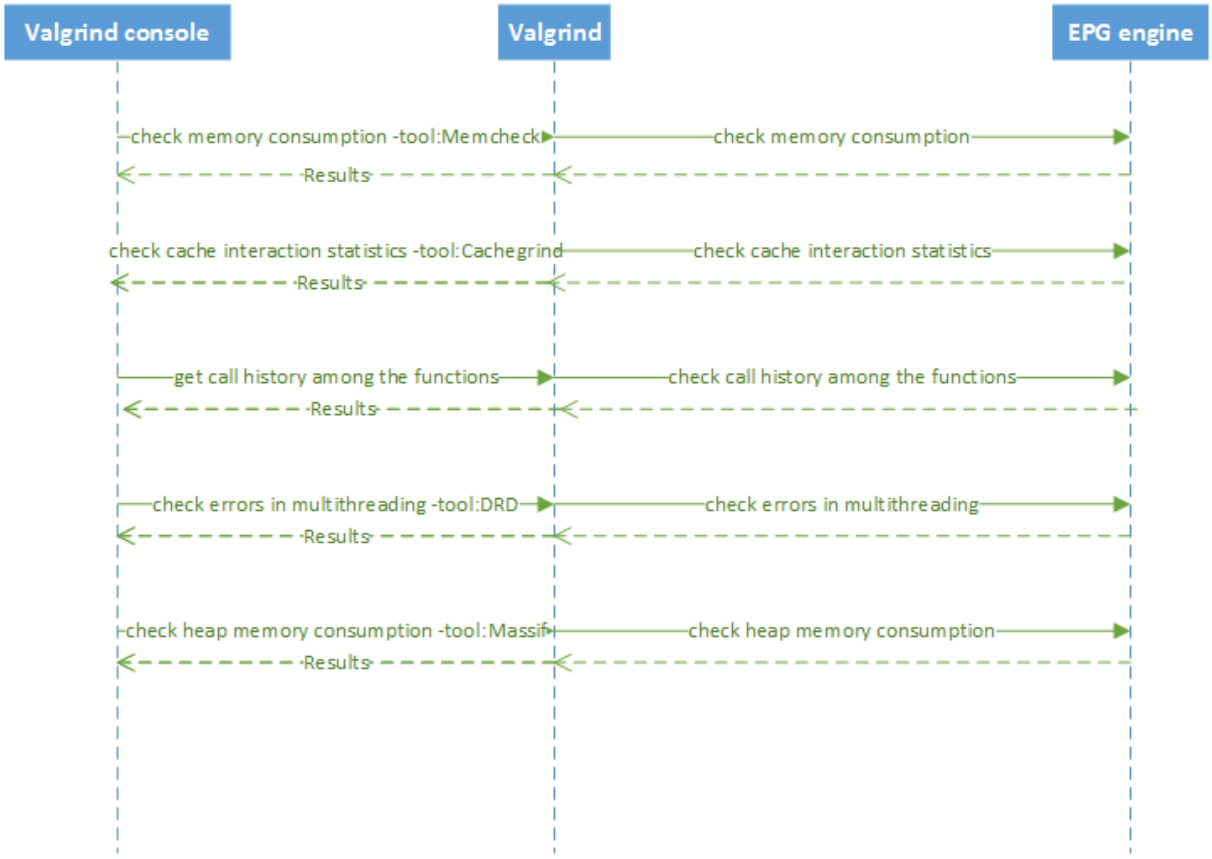
Sl. 3.1. *Prikaz dijagrama korištenja Valgrind-a*

Na slici 3.1. prikazani su slučajevi korištenja Valgrind-a i njegovih alata te izvođenje programskog profiliranja u dijagramu korištenja (engl. *use case diagram*). Korisnik je tester koji stvara slučajeve programskog profiliranja i zatim ih putem Valgrind-a ih izvršava. Glavni slučajevi koji se odrađuju u programskom profiliranju su:

- *Memory leak detection* – alat Memcheck,
- *Cache interaction statistics* – alat Cachegrind,
- *Gathering call history among functions* – alat Callgrind,
- *Detect errors in multithreading* – alat DRD,
- *Measure heap memory consumption* – alat Massif i
- *Time execution* – nije moguće izvesti putem Valgrind-a.

Rezultati se vraćaju u obliku komentara, tekstualnog oblika, gdje se prikazuju pogreške koje su otkrivene. Omogućeno je čitati rezultate iz komandne linije, spremiti ih i čitati iz datoteke

ili poslati putem mrežnog priključka na određenu IP adresu. Bitno je napomenuti da komentari sadrže izmiješane poruke iz Valgrind-ove jezgre i korištenog alata. Kada alat ispisuje pogreške, ispisivat će ih u komentare, ali kada alat obavlja analizu i profiliranje programskog koda tada će rezultate ispisivati u određeni tip datoteke. Razlog tomu je što su rezultati analize i profiliranja najčešće vrlo opširni i veliki, te ih je potrebno dodatno obraditi kako bi se dobile korisne informacije.



Sl. 3.2. *Prikaz sekvencijalnog dijagrama pri korištenju Valgrind-a*

Slika 3.2. prikazuje način odvijanja rada programskog profiliranja. Tester putem Valgrind konzole upisuje naredbe koje se zatim izvršavaju putem Valgrind-a. Valgrind izvršava programsko profiliranje nad programom koji je u ovom radu programska podrška TV aplikacije elektronskog programskog vodiča. Na temelju programskog profiliranja stvaraju se rezultati koji se šalju nazad u Valgrind konzolu. Upisivanjem pojedinih naredbi za Valgrind

omogućuje se korištenje alata i njihovih opcija. Alati koji su korišteni u izradi ovog rada, bit će opisani u sljedećim potpoglavljima.

3.2. Memcheck

Memcheck je alat za provjeru pogrešaka vezanih uz zauzimanje i oslobađanje memorije. Ovo je početni alat Valgrinda. Najbolje radi nad programskim kodom pisanim u C-u i C++-u. Neke od pogrešaka koje Memcheck pronalazi su:

- pristupanje memoriji kojoj se ne bi trebalo pristupati (npr. oslobođenoj memoriji, presipavanje heap blokova, presipavanje *stack*-a),
- korištenje nedefiniranih vrijednosti (npr. varijabli koje nisu inicijalizirane),
- neispravno oslobađanje memorije *heap*-a, nejednak broj *malloc-a* i *free-a*,
- preklapanje pokazivača pri korištenju funkcija poput *memcpy*,
- predavanje loših vrijednosti (npr. negativnih) kao argument pri korištenju *malloc-a* te
- curenje memorije.

Pokreće se naredbom u terminalu *valgrind --tool=memcheck yourProgram*. Memcheck nadzire pristupanje memoriji kojoj se ne bi trebalo pristupiti tijekom izvođenja programa. Međutim, ako se dogodi takvo pristupanje dolazi do segmentacijske pogreške u programu te će Memcheck javiti na kojem mjestu se ona dogodila. Pri korištenju nedefiniranih vrijednosti javlja se pogreška pristupanju varijablama koje nisu postavljene. Bitno je napomenuti kako program može raditi nad memorijom koja se nalazi u susjedstvu tih varijabli. Memcheck prati te varijable i kada ih program pokuša koristiti za daljnji tijek programa, javlja pogreške. Uzroci takvih pogrešaka su lokalne varijable koje nisu prethodno inicijalizirane i korištenje memorijskih adresa *heap*-a gdje su prethodno uskladišteni određeni sadržaji. Prati se broj zauzetih blokova memorije putem funkcija *malloc/new* kako bi se točno znalo je li na određenim mjestima u programskom kodu dozvoljeno pozvati funkcije za oslobađanje memorije, *free/delete*. Nekada je potrebno kopirati sadržaj određene memorijske adrese na drugu memorijsku adresu korištenjem funkcije *memcpy,* te Memcheck provjerava da se te memorijske adrese ne preklapaju. Provjeravaju se poslane vrijednosti funkciji *malloc* kako bi zauzeće memorije bilo ispravno izvedeno. Može se dogoditi da vrijednost dobivena matematičkom operacijom bude netočna, izrazito velika ili čak negativna, a kasnije se koristi kao vrijednost veličine pri dinamičkom zauzimanju memorije.

Najbitnija stavka Memcheck-a je provjeravanje curenja memorije. Curenje memorije može se definirati kao loše korištenje resursa, memorije računala pri izvođenju računalnog

programa. Posljedice curenja memorije mogu biti lošiji rad računala zbog pomanjkanja iskoristive memorije, rušenje aplikacije ili sustava, blokiranje aplikacije ili sustava i sl. Problem curenja memorije je vrlo ozbiljan za ugradbene sustave. Takvi sustavi najčešće imaju vrlo ograničene resurse npr. veličinu memorije, broj jezgri procesora itd. Vrlo je bitno pronaći i zaustaviti curenje memorije pri ugradbenim računalnim sustavima. Tu se često koristi dijeljena memorija, stoga ostavljanje nekakvih zadataka u pozadini koji iskorištavaju memoriju može biti vrlo problematično. Ovaj problem postaje još ozbiljniji, ako ugradbeni računalni sustavi rade kao sustavi u stvarnom vremenu[6].

Memcheck prati blokove na heap memoriji koji su povezani sa pozivima funkcija *malloc/new* i prema tome zna koji blokovi memorije nisu oslobođeni. Putem pokazivača utvrđuje što se dogodilo blokovima memorije. Početni pokazivač se nalazi na početku zauzetog bloka memorije, dok interni pokazivač se nalazi unutar zauzetog bloka memorije.

Curenje memorije opisuje kroz četiri vrste rezultata:

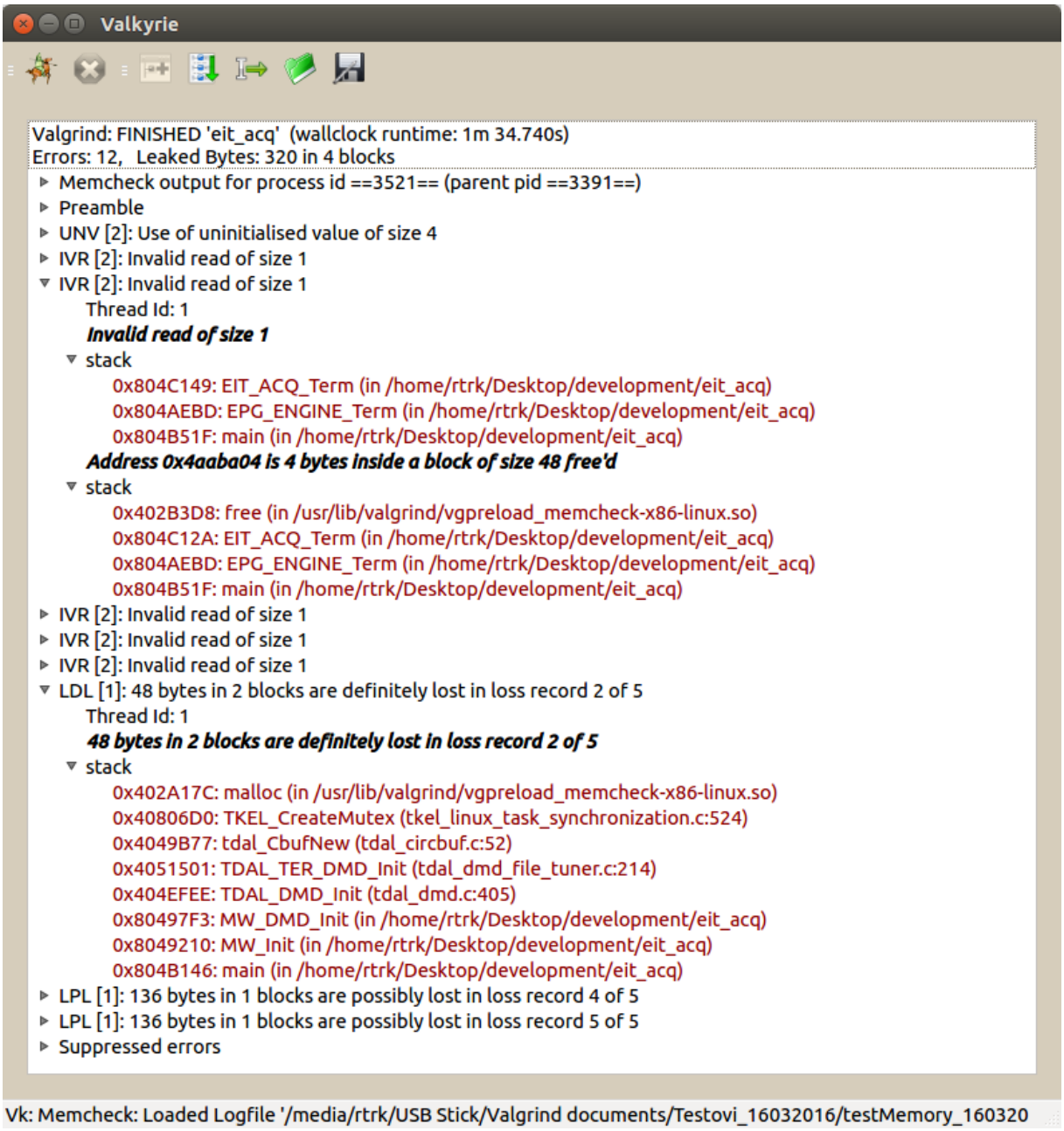
- "Još uvijek dostupna",
- "Sigurno izgubljena",
- "Neizravno izgubljena" i
- "Vjerojatno izgubljena"

"Još uvijek dostupna" memorija se može definirati kada se početni pokazivač može točno odrediti. Kako je to moguće, tada programer može unutar programskog koda osloboditi memoriju funkcijom *free/delete.*

"Sigurno izgubljena" memorija se može definirati kada se ne može uopće pronaći pokazivača na blok memorije koji je prethodno bio zauzet. Uzrok je najčešće loše napisan programski kod.

"Neizravno izgubljena" memorija se može definirati kada su dijelovi memorije koji su bili zauzeti izgubljeni, dok njihovi pokazivači su konzistentni. Ovaj primjer se može pojasniti putem binarnog stabla. Izgubi li se korijenski čvor binarnog stabla, gube se svi ostali čvorovi stabla. Ista stvar se događa s memorijom.

"Vjerojatno izgubljena" memorija se može definirati kada postoje početni pokazivači na blokove memorije, ali postoji barem jedan unutarnji pokazivač. Ovaj rezultat nužno ne označava pogrešku jer se može dogoditi da je unutarnji pokazivač slučajna vrijednost koja pokazuje na taj blok memorije[7].



Sl. 3.3. *Prikaz rezultata rada Memcheck-a putem GUI aplikacije Valkyrie*

Budući da postoje različite vrste curenja memorije, postavlja se pitanje koje od tih vrsta treba smatrati pogreškama. Slikom 3.3. prikazan je rezultat rada Memcheck-a u *GUI* (engl. *Graphical User Interface*) aplikaciji Valkyrie.

Tablica 3.1. daje prikaz nekih od naredbi Memcheck-a i njihovo pojašnjenje.

Tab. 3.1. *Češće korištene naredbe Memcheck-a*

Naredba	Objašnjenje
--leak-check=<no|summary|yes|full>	Početno je postavljeno na *summary* i prikazuje se samo koliko je grešaka curenja memorije nastalo, dok korištenjem *full* ili *yes*, se dobiva detaljniji izvještaj.
--leak-resolution=<low|med|high>	Početno je postavljeno na *high*. Promjenom *leak-resoultion*-a mijenja se izgled izvještaja, no ne utječe na pronalaženje grešaka.
--show-leak-kinds=<set>	Koristi se za odabir koje vrste grešaka curenja memorije će biti prikazane u izvještaju. Početno je postavljeno *definite i possible*, dok je moguće još postaviti *indirect i reachable.*
--track-origins=<yes|no>	Početno je postavljeno na *no* i kontrolira hoće li se pratiti korijeni neinicijaliziranih varijabli.

3.3. Cachegrind

Cachegrind oponaša kako se program rukuje s priručnom memorijom računala i predočava način na koji se program grana u pojedinom trenutku izvođenja. Oponaša računalo koji ima neovisnu priručnu memoriju sa prvom razinom instrukcije (I1 i D1), poduprt s jedinstvenom drugom razinom priručne memorije (L2). Ovakvu arhitekturu ima većina današnjih računala.

Računala mogu imati drugačije arhitekture priručne memorije, npr. mogu imati više razina. Cachegrind ima mogućnost zapažanja ove karakteristike pa kod takvih računala oponaša prvu i zadnju razinu priručne memorije. Razlog tomu je što zadnja razina priručne memorije ima najveći utjecaj na izvođenje programa i ona maskira pristupanje glavnoj memoriji. Prva razina priručne memorije ima slabije izraženo svojstvo asocijativnosti tj. kopija pojedinog dijela glavne memorije može biti upisana u jedno ili dva mjesta u priručnoj memoriji, zato je kod nje relativno lako otkriti gdje programski kod loše surađuje sa priručnom memorijom. Loša suradnja glavne memorije i priručne memorije najčešće očituje kao promašaj čitanja iz priručne memorije. Promašaj čitanja iz priručne memorije je stanje u kojem traženi podaci za rad programa nisu pronađeni u prvoj razini priručne memorije te

uzrokuje kašnjenje rada programa jer je potrebno dohvatiti podatke iz drugih razina priručne memorije. Radi svega navedenoga, Cachegrind uvijek nadzire rad programa sa priručnom memorijom I1, D1 i LL zadnjom razinom priručne memorije (engl. *last-level*). Cachegrind prikuplja sljedeću statistiku rada priručne memorije:

- čitanje I priručne memorije (Ir, jednako broju izvedenih instrukcija), promašena čitanja I priručne memorije (Imr) i promašena čitanja instrukcija LL priručne memorije (ILmr),
- čitanje D priručne memorije (Dr, jednako broju čitanja memorije), promašena čitanja D1 priručne memorije (D1mr) i promašena čitanja podataka LL priručne memorije (DLmr),
- pisanje D priručne memorije (Dw, jednako broju pisanja u memoriju), promašena pisanja D1 priručne memorije(D1mw) i promašena pisanja podataka LL priručne memorije (DLmw),
- Izvršena uvjetna grananja (Bc) i krivo pretpostavljena uvjetna grananja (Bcm) i
- Izvršena indirektna grananja (Bi) i krivo pretpostavljena indirektna grananja (Bim).

Ova statistika se prikuplja za cijeli program i svaku funkciju u programu. Može se vidjeti koja linija programskog koda je uzrokovala koliko interakcije s priručnom memorijom. Današnja računala najčešće imaju sljedeću statistiku:

- L1 promašaj košta oko 10 ciklusa,
- LL promašaj može koštati i do 200 ciklusa te
- krivo pretpostavljeno grananje može koštati između 10 i 30 ciklusa.

Pokretanje alata Cachegrind vrši se upisivanjem naredbe u terminal *valgrind --tool=cachegrind yourProgram*[8].

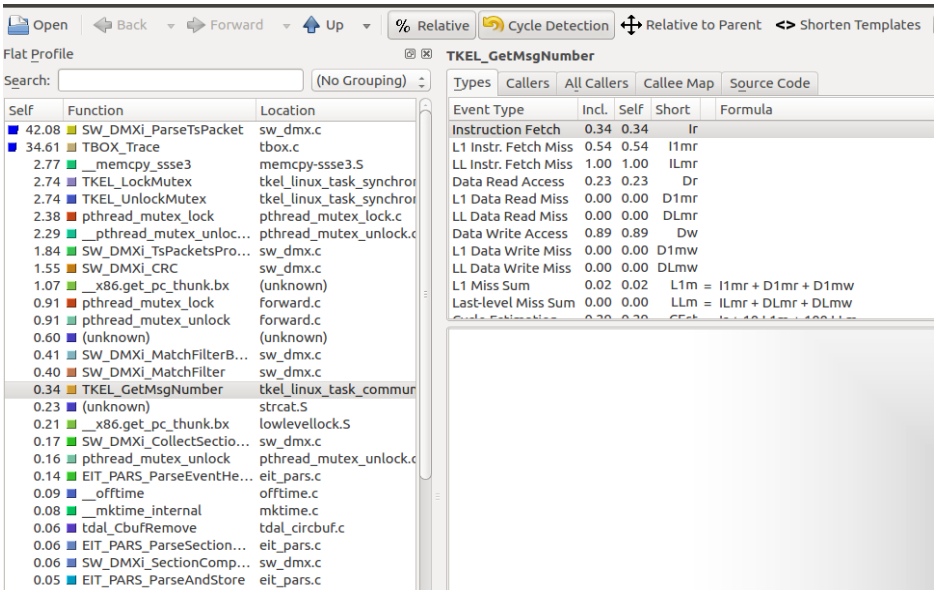
Rezultat analize Cachegrind alata sažeto se prikazuje u konzoli, dok se opširniji prikaz zapisuje u *cachegrind.out* datoteku. Čitanje ove datoteke je omogućeno aplikacijom otvorenog koda KCachegrind. Sažetak analize prikazuje konfiguraciju I1, D1 i LL priručne memorije, naredba koja je upisana za analizu, događaji koji su zabilježeni, događaji koji su prikazani, poredani prikaz događaja, prag i je li uključeno automatsko zapisivanje.

Cachegrind kao rezultat daje veliki broj informacija koje je potrebno analizirati i otkriti koje su informacije bitne za izvođenje programa. Broj pogođenih/promašenih pisanja/čitanja u priručnu memoriju nije izrazito koristan. Mogu se pokazati korisnima samo kada postoji više pokretanja programa. Brojanje poziva funkcija daje korisnije informacije o radu programa. Ovaj prikaz daje informaciju koje od funkcija su koliko puta pozivane. Postoji problem ako su

pojedine funkcije sastavni dio veće funkcije, tada se broj poziva pripisuje funkciji čiji su dio. Bilješke linija po liniju koda su korisne jer daju prikaz koliko instrukcija je izvedeno u pojedinoj liniji programskog koda. Mogu biti vrlo korisne pri otkrivanju zastoja unutar izvođenja programa. Preciznost analize može predstavljati problem. Naime, profiliranje priručne memorije ima nedostatke u sljedećim oblicima:

- ne uračunava aktivnost jezgre (engl. *kernel*),
- ne uračunava aktivnosti drugih procesa koji se mogu istovremeno izvoditi s programom koji se analizira,
- ne uračunava mapiranje memorije iz virtualne u stvarnu, stoga simulacija nije stvarni prikaz onoga što se događa priručnoj memoriji,
- na uračunava promašaje priručne memorije koji nisu vidljivi na razini instrukcija,
- Valgrind drugačije raspoređuje niti programa, nego što one budu raspoređene kada se program izvodi i
- rezultati su vrlo osjetljivi na promjene veličina datoteke i knjižnica koje se koriste u programu

Slika 3.4. daje grafički prikaz rezultata u *GUI* aplikaciji KCachegrind dobivenih Cachegrind-om .



Sl. 3.4. *Prikaz rezultata Cachegrind-a u GUI aplikaciji KCachegrind*

Tablica 3.2. prikazuje neke od naredbi Cachegrind-a i njihovo objašnjenje.

Tab. 3.2. *Češće korištene naredbe Cachegrind-a*

Naredba	Objašnjenje
--I1=<size>,<associativity>,<line size>	Omogućuje specificiranje veličine, asocijativnosti i veličine linije I1 priručne memorije
--D1=<size>,<associativity>,<line size>	Omogućuje specificiranje veličine, asocijativnosti i veličine linije D1 priručne memorije
--LL=<size>,<associativity>,<line size>	Omogućuje specificiranje veličine, asocijativnosti i veličine linije LL priručne memorije
--cache-sim=no|yes	Početno je postavljeno na *yes*, omogućuje ili onemogućuje zbrajanje pristupa priručnoj memoriji i promašaje.

3.4. Callgrind

Callgrind je alat koji prati i bilježi povijest poziva između funkcija tijekom izvođenja programa, te daje prikaz u obliku grafa poziva funkcija. Prikupljeni podaci sadrže broj izvršenih instrukcija, njihovu poveznicu sa kodom, vezu pozivatelja funkcija i pozvanih funkcija i broj takvih poziva. Podaci se zapisuju u datoteku nakon završetka izvođenja programa. Prikaz podataka omogućen je putem dvije naredbe, *callgrind_annotate* i *callgrind_control*. *Callgrind_annotate* učitava podatke, te ispisuje sortiranu listu funkcija s mogućnošću prikaza izvora pozivanja. Grafički prikaz omogućen je putem dodatnog alata, KCachegrind. Omogućen je pregledniji prikaz prikupljenih podataka. *Callgrind_control* je naredba koja omogućuje interaktivno promatranje i kontrolu statusa programa koji se trenutno izvodi pod Callgrind-om, bez zaustavljanja programa[9].

Podaci koji se prikupljaju sadrže broj događaja koji su izravno povezani s funkcijom u kojoj su se dogodili. Callgrind dodatno nadopunjuje tu mogućnost propagirajući broj tih događaja sve do izvora pozivatelja funkcije. Primjerice, ako funkcija *getNumber* poziva funkciju *convertToNumber*, broj događaja iz funkcije *convertToNumber* pribrajaju se broju događaja funkcije *getNumber*. Graf poziva funkcija uz broj događaja u pojedinoj funkciji omogućuje pronalaženje niza poziva funkcija tijekom programa. Sposobnost opažanja poziva

funkcija ovisi o platformi na kojoj se program izvodi. Callgrind najbolje radi na x86 i amd64 arhitekturama, dok nešto lošije rezultate daje za *ARM, Thumb* ili *MIPS*. Razlog tomu je što nema eksplicitnih pozivnih i povratnih instrukcija u tim instrukcijskim setovima.

Osnovno profiliranje Callgrindom vrši se putem naredbe *valgrind --tool=callgrind yourProgram*. Za vrijeme trajanja simulacije, izvođenje se može promatrati pomoću naredbe *callgrind_control -b.* Nakon završetka izvođenja programa, prikupljeni podaci se nalaze u datoteci pod nazivom *callgrind.out.*<*pid*> gdje je *pid* ID procesa programa nad kojim se vrši profiliranje. Datoteka sadrži informacije o pozivima funkcija koje su se izvodile tijekom izvođenja programa, zajedno uz broj izvedenih instrukcija (Ir). Kako bi prikaz grafa poziva funkcija bio razumljiviji, preporučljivo je koristiti KCachegrind. Ako kod sadrži korištenje rekurzivnih funkcija, mora se koristiti KCachegrind za točnost prikaza podataka jer *callgrind_annotate* nema mogućnost prepoznavanja ciklusa. Budući da je veliki dio rada Callgrind-a zasnovan na alatu Cachegrind, omogućeno je mjerenje rada priručne memorije putem dodatne opcije *--cache-sim=yes*. Moguće je promatrati kod na asemblerskoj razini pomoću opcije *--dump-instr=yes*. Treba napomenuti da se takvi rezultati mogu promatrati samo u KCachegrind-u.

Tablicom 3.3. prikazane su neke od naredbi Callgrind-a i njihovo objašnjenje.

Tab. 3.3. *Češće korištene naredbe Callgrind-a*

Naredba	Objašnjenje
--dump-line=<no|yes>	Početno je postavljeno na *yes*. Određuje da se brojanje događaja odvija na razini linije programskog koda.
--dump-before=<function>	Određuje prije koje funkcije će se izbaciti ranije prikupljeni podaci, ista mogućnost postoji i za nakon određene funkcije (*--dump-after*).
--toggle-collect=<function>	Prebacivanje prikupljanja podataka na ulazak ili izlazak iz funkcije.
--dump-instr=<no|yes>	Početno je postavljeno na *no*. Određuje da je brojanje događaja odvija na razini instrukcije i omogućuje podatke na razini asemblerskog koda.
--separate-threads=<no|yes>	Početno je postavljeno na *no.* Opcija koja omogućuje prikupljanje podataka odvojeno za pojedinu nit programa.

3.5. DRD

DRD je Valgrind alat za pronalaženje pogrešaka u višenitnim C i C++ programima. Alat radi za bilo koji program koji koristi POSIX koncepte niti ili koristi koncepte višenitnosti koji se temelje na POSIX konceptima niti.

Postoje dva moguća razloga izrade višenitnih programa:

1. Modeliranje istodobnih aktivnosti - dodjeljivanje jedne aktivnosti za pojedinu nit može biti veliko pojednostavljenje u odnosu na multipleksiranju stanja više aktivnosti u jednoj niti. Stoga je većina programske podrške za servere i ugradbene računalne sustave višenitna.
2. Korištenje više CPU jezgri istovremeno radi ubrzanja - zato su HPC (engl. *High Performance Computing)* aplikacije višenitne.

Višenitni programi mogu koristiti jednu ili više paradigmi, dok odabir paradigme ovisi o aplikaciji. Neki primjeri višenitnih programskih paradigmi su:

- Zaključavanje - podaci su dostupni svim nitima, stoga ih je potrebno zaštititi od istovremenog pristupanja putem zaključavanja,
- Dijeljenje poruka - podaci nisu dostupni svim nitima, već razmjena podataka se vrši putem dijeljenja poruka između pojedinih niti, npr. MPI (engl. *Message Passing Interface*),
- Automatska paralelizacija – programski prevoditelj prebacuje sekvencijalni program u višenitni program. Originalni program može, ali i ne mora sadržavati upute za paralelizaciju, npr. OpenMP (engl. *Open Multi-Processing*) i
- *Software Transactional Memory* - podaci koji se dijele među nitima se ažuriraju putem transakcija. Nakon svake transakcije provjerava se postoje li transakcije koje se sukobljavaju. Primjenjuje se tzv. optimistični princip[10].

Ovisno koja se višenitna programska paradigma koristi, mogu se pojaviti neki od problema:

- *Data races* - jedna ili više niti istovremeno pristupaju istoj memorijskoj lokaciji bez dovoljno dobrog zaključavanja.
- Preduga zaključavanja - jedna nit može uzrokovati zastoj ostalim nitima, ako predugo drži zaključane podatke
- Loša implementacija i korištenje POSIX niti - npr. mutex otključava nit koja ga nije zaključala
- Zastoj - pojavljuje se kada dvije ili više niti čekaju jedna drugu neodređeno dugo
- Lažno dijeljenje - Ako niti koje rade na različitim procesorskim jezgrama pristupitaju često različitim varijablama koje se nalaze u istoj priručnoj memoriji, to će puno usporiti uključene niti zbog čestih razmjena priručne memorije[10].

Tablica 3.4. daje prikaz nekih od naredbi DRD-a i njihovo objašnjenje.

Tab. 3.4. *Češće korištene naredbe DRD-a*

Naredba	Objašnjenje
--trace-alloc=<yes|no>	Omogućuje praćenje aktivnosti zauzimanja i oslobađanja memorije.
--trace-cond=<yes|no>	Omogućuje praćenje aktivnosti kondicionalnih varijabli.
--trace-mutex=<yes|no>	Omogućuje praćenje aktivnosti mutexa.
--trace-semaphore=<yes|no>	Omogućuje praćenje aktivnosti semafora.
--check-stack-var=<yes|no>	Omogućuje praćenje *data races* na *stack-u*.

Najčešće pogreške koje se pojavljuju tijekom profiliranja višenitnih programa su *data races*, preduga zaključavanja i krivo korištenje POSIX biblioteke. DRD nakon završetka izvođenja programa, daje izvještaj profiliranja koji se može pregledati u terminalu ili spremiti u datoteku.

Pri prikazu izvještaja bitno je razumjeti na koji način DRD generira isti:

- svakoj niti je pridružen ID niti koji je broj i jedinstven je za svaku nit pri jednom izvršavanju programa,
- izraz *segment* odnosi se na uzastopni niz operacija dohvaćanja, spremanja i sinkronizacije koji su izvedeni u istoj niti te
- uvijek postoje barem dva pristupanja memoriji koji se nalaze u *data race-u*. Takva dva pristupanja memoriji se nazivaju pristupi memoriji u sukobu.

Pokretanje DRD alata vrši se u terminalu naredbom *valgrind --tool=drd yourProgram*. Slika 3.5. prikazuje izgled izvještaja rada DRD alata.

```
==4080== [1] mutex_unlock    mutex 0xfecd2240 rc 1
==4080== [1] mutex_destroy   mutex 0xfecd2240 rc 0 owner 1
==4080== [1] mutex_trylock   mutex 0xfecd2370 rc 0 owner 10
==4080== [1] post_mutex_lock mutex 0xfecd2370 rc 0 owner 10
==4080== [1] mutex_unlock    mutex 0xfecd2370 rc 1
==4080== [1] mutex_destroy   mutex 0xfecd2370 rc 0 owner 1
==4080== [1] mutex_unlock    mutex 0x56c487c rc 1
==4080== Conflicting store by thread 1 at 0x056a69c4 size 4
==4080==    at 0x565701D: TBOX_SetTraceEnabled (tbox.c:178)
==4080==    by 0x404F9BA: EIT_FETC_Init (eit_fetc.c:184)
==4080==    by 0x5477A5B: EPG_ENGINE_Init (mw.c:617)
==4080==    by 0x547838A: MW_Run (mw.c:781)
==4080==    by 0x804896C: main (epg_client.c:138)
==4080== Allocation context: BSS section of /home/rtrk/Desktop/EPG_ENGINE/lib/chal/libchal.so.1.0.1474
==4080== Other segment start (thread 4)
==4080==    at 0x5585BD8: clone (clone.S:108)
==4080== Other segment end (thread 4)
==4080==    at 0x56DA1D0: sem_wait@GLIBC_2.0 (sem_wait.S:313)
==4080==    by 0x4035FBC: sem_wait@* (in /usr/lib/valgrind/vgpreload_drd-x86-linux.so)
==4080==    by 0x5691031: TKEL_WaitSemaphore (tkel_linux_task_synchronization.c:178)
==4080==    by 0x5693059: TimerTask (tkel_linux_timer_management.c:211)
==4080==    by 0x40303A4: ??? (in /usr/lib/valgrind/vgpreload_drd-x86-linux.so)
==4080==    by 0x40303A4: ??? (in /usr/lib/valgrind/vgpreload_drd-x86-linux.so)
==4080==    by 0x56D3F6F: start_thread (pthread_create.c:312)
==4080==    by 0x5585BED: clone (clone.S:129)
==4080== Other segment start (thread 6)
==4080==    at 0x5585BD8: clone (clone.S:108)
==4080== Other segment end (thread 6)
==4080==    at 0x56DA1D0: sem_wait@GLIBC_2.0 (sem_wait.S:313)
==4080==    by 0x4035FBC: sem_wait@* (in /usr/lib/valgrind/vgpreload_drd-x86-linux.so)
==4080==    by 0x5691031: TKEL_WaitSemaphore (tkel_linux_task_synchronization.c:178)
==4080==    by 0x568D084: TKEL_Dequeue (tkel_linux_task_communication.c:359)
==4080==    by 0x5664725: TDAL_DMDi_TimerTask (tdal_dmd_file_tuner.c:1543)
==4080==    by 0x40303A4: ??? (in /usr/lib/valgrind/vgpreload_drd-x86-linux.so)
==4080==    by 0x40303A4: ??? (in /usr/lib/valgrind/vgpreload_drd-x86-linux.so)
==4080==    by 0x56D3F6F: start_thread (pthread_create.c:312)
==4080==    by 0x5585BED: clone (clone.S:129)
==4080== Other segment start (thread 8)
```

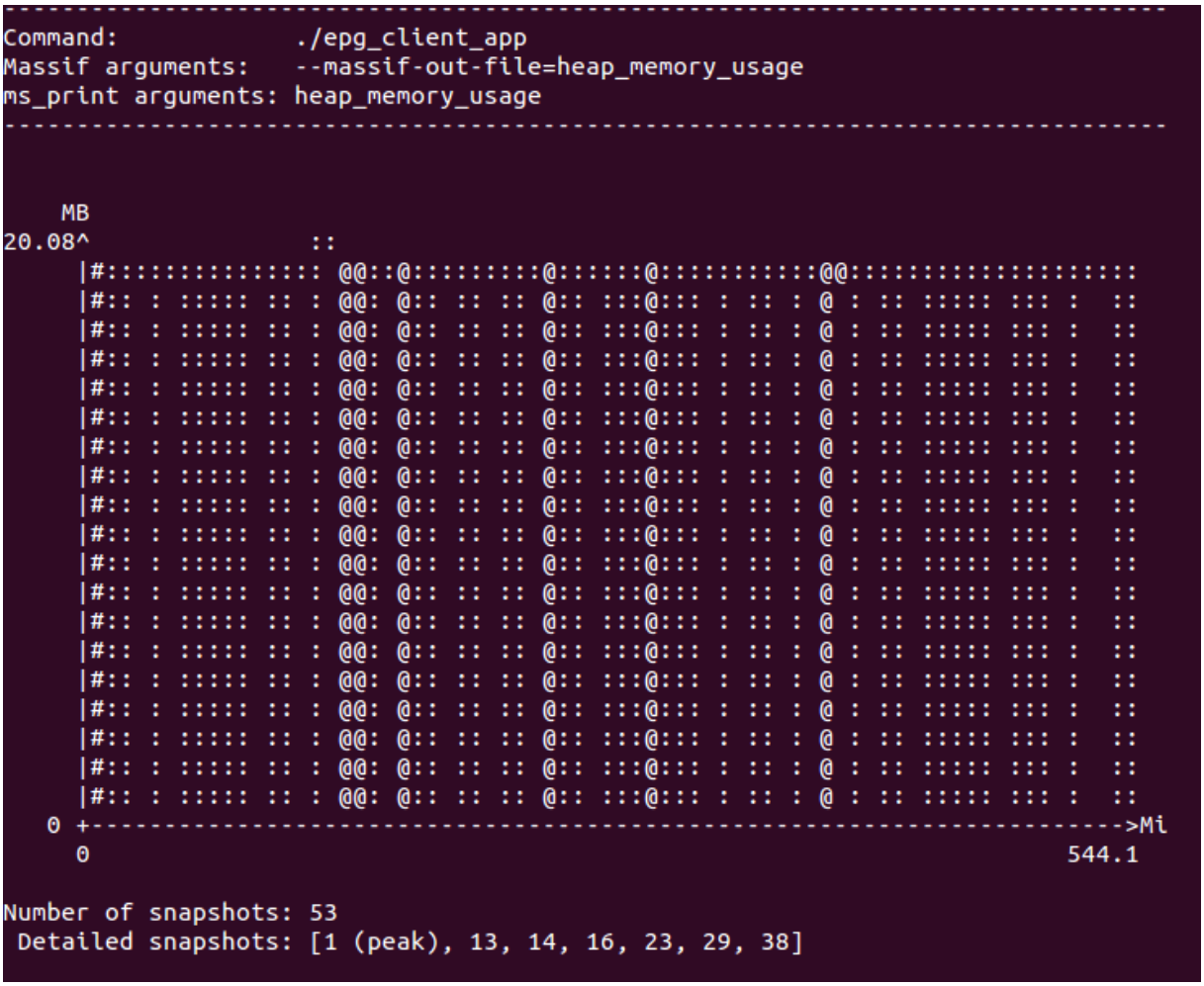
Sl. 3.5. *Prikaz DRD izvještaja*

3.6. Massif

Massif je alat za profiliranje *heap*-a. Mjeri koliko *heap* memorije zauzima program koji se izvodi, uključujući korisni prostor i dodatne dijelove memorije koji su zauzeti za očuvanje poravnanja memorije. Može mjeriti iskoristivost *stack*-a, iako to ne radi prema početnim postavkama. Profiliranje *heap*-a može pomoći smanjiti količinu memorije koju program zauzima, što na modernim računalima s virtualnom memorijom pruža prednosti poput ubrzanja rada programa i smanjenje vjerojatnosti za potrošnjom *swap* prostora. Osim što prikazuje koliko je *heap* memorije zauzeto, Massif prikazuje detaljne informacije o tome koji dijelovi programa su odgovorni za zauzimanje *heap*-a.

Pokretanje Massif-a vrši se sljedećom naredbom u terminalu *valgrind --tool=massif yourProgram*[11].Nakon završetka izvođenja programa, podaci o profiliranju se spremaju u poseban dokument *massif.out.*<*pid*> gdje <pid> je ID procesa, iako je moguće ime dokumenta promijeniti. Prikaz rezultata moguće je vidjeti putem naredbe *ms_print* i naziva dokumenta. Rezultati profiliranja se prikazuju u terminalu, te se prikazuje graf korištenja

memorije tijekom izvođenja programa i detaljne informacije za pojedine alokacije memorije tijekom izvođenja programa. Na grafu je moguće vidjeti maksimum vrijednost zauzete memorije tijekom izvođenja programa.



Sl. 3.6. *Prikaz grafa za Massif profiliranje*

Na slici 3.6. vidljivo je zauzeće *heap*-a. Os ordinata prikazuje količinu korištenog *heap*-a u MB, dok na osi apscisa jedinica prikaza je broj izvedenih instrukcija, Mi. Jedinica za prikaz na osi apscisa može biti, osim broja izvedenih instrukcija (i), vrijeme u milisekundama (ms) i količina bajtova alocirana/dealocirana na *heap-u* (B). Prikaz u bajtovima je koristan za vrlo kratke programe, dok se za ostala korištenja preporučuje broj izvedenih instrukcija kao jedinica. Promjena jedinice vrši se putem dodatne opcije u terminalu *--time-unit=<i|ms|B>*. Graf prikazuje snimke korištenja *heap*-a u obliku okomitih crta koje su iscrtane raznim

znakovima @, : i #. Neki od snimaka imaju detaljne informacije koje se mogu pronaći kasnije u izvještaju. Takvi snimci su iscrtani znakom @. Prema početnim postavkama uzima se deset snimaka uz detaljne informacije koje se kasnije mogu pronaći u izvještaju. Tijekom izvođenja programa postoji maksimum zauzeća memorije *heap*-a. Takav snimak se prikazuje znakom #. Ispod iscrtanog grafa prikazan je broj snimaka tijekom izvođenja programa, te su prikazani snimci čije detaljne informacije su u izvještaju i snimak maksimuma. Izvještaj osim grafa sadrži detaljne informacije snimaka. Svaki od zapisa pojedinog snimka sastoji se od:

- broja,
- vrijeme kada je snimak nastao,
- ukupno iskorištenje memorije u tom trenutku,
- broj alociranih *bajt*ova korisnog *heap*-a u tom trenutku (broj bajtova koje je program zatražio),
- broj dodatno alociranih bajtova *heap*-a u tom trenutku (dodatni broj bajtova alociranih tijekom rada programa) i
- veličina *stack*-a

```
--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 24    254,859,148       21,045,848       21,037,117        8,731            0
 25    267,541,088       21,047,968       21,039,091        8,877            0
 26    277,752,915       21,051,664       21,042,808        8,856            0
 27    288,406,778       21,052,592       21,043,624        8,968            0
 28    296,408,898       21,048,416       21,039,639        8,777            0
 29    306,149,365       21,045,776       21,037,046        8,730            0
99.96% (21,037,046B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->99.92% (21,028,957B) 0x6077F90: TKEL_Malloc (tkel_linux_memory.c:247)
| ->99.65% (20,971,520B) 0x604CDA7: TDAL_TER_DMD_Init (tdal_dmd_file_tuner.c:230)
| | ->99.65% (20,971,520B) 0x6052BD4: TDAL_DMD_Init (tdal_dmd.c:405)
| |   ->99.65% (20,971,520B) 0x5E63680: MW_DMD_Init (mw.c:316)
| |     ->99.65% (20,971,520B) 0x5E63013: MW_Init (mw.c:221)
| |       ->99.65% (20,971,520B) 0x5E652B6: MW_Run (mw.c:768)
| |         ->99.65% (20,971,520B) 0x8048996: main (epg_client.c:160)
| |
| ->00.27% (57,437B) in 1+ places, all below ms_print's threshold (01.00%)
|
->00.04% (8,089B) in 1+ places, all below ms_print's threshold (01.00%)

--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 30    314,032,417       21,049,664       21,040,816        8,848            0
 31    322,156,187       21,046,664       21,037,889        8,775            0
 32    332,792,884       21,046,000       21,037,269        8,731            0
 33    347,532,002       21,045,984       21,037,260        8,724            0
```

Sl. 3.7. *Prikaz detaljnih informacija Massif izvještaja*

Pregled detaljnih informacija prikazuje naziv funkcije, količinu bajtova koju je zauzela, datoteku u kojoj se funkcija nalazi i memorijsku adresu na kojoj je zauzela bajtove. Ovaj prikaz vidljiv je na slici 3.7. Bolji grafički prikaz omogućen je dodatnom aplikacijom *Massif Visualizer*. Informacije koje profiliranje i izvještaj Massif-a pruža relativno je lako razumjeti i djelovati na temelju njih. Početna točka pregledavanja izvještaja je maksimum snimak. Korisno je pregledati graf gdje je moguće vidjeti zauzimanje *heap*-a tijekom izvođenja programa, pa prema tome i pretpostavkama programera vidjeti ponaša li se program kako je željeno. Također je moguće vidjeti koje funkcije zauzimaju najveći dio *heap*-a i prema tome vidjeti kako optimirati program.

Tablica 3.5. daje prikaz i pojašnjenje češće korištene naredbe Massif-a.

Tab. 3.5. *Češće korištene naredbe Massif-a*

Naredba	**Objašnjenje**
--stacks=<yes|no>	Omogućuje profiliranje *stack*-a.
--time-unit=<i|ms|B>	Omogućuje odabir vremenske jedinice. Početno je postavljen broj instrukcija.
--max-snapshots=<n>	Omogućuje postavljanje maksimalnog broja snimaka. Početno je postavljeno 100.
--detailed-freq=<n>	Omogućuje postavljanje frekvencije uzimanja detaljnih snimaka. Početno je postavljeno 10.

4. UNIT TESTIRANJE I KONTINUIRANA INTEGRACIJA

Unit testiranje ili testiranje najmanjih cjelovitih jedinica programskog koda je dio programskog koda, najčešće metoda, koja poziva drugi dio programskog koda i provjerava ispravnost pretpostavljenog rada programskog koda. Kada je pretpostavka neispravna, tada je *unit* test pao. *Unit* se može definirati kao metoda ili funkcija. Proceduralno programiranje, kao C programiranje, *unit* izjednačuje najčešće sa funkcijom. Objektno-orijentirano programiranje, kao C++ ili C# programiranje, *unit* najčešće izjednačuje sa klasom ili metodom. Ovaj način testiranja naziva se *white-box testing*. *White-box testing* je vrsta testiranja koja se odnosi na testiranje unutarnjih struktura koda aplikacije. Poznavanje rada aplikacije potrebno je za pisanje *unit* testova. Tester najčešće odabire ulazne parametre za pojedini test, te na temelju njih očekuje točno određeni izlaz. Ova razina testiranja se radi kako bi se osiguralo da pojedini *unit* radi kako je zamišljeno prije no što se krene u integraciju sustava. Ispravljaju se pogreške u početnoj fazi programiranja, kako bi jednostavnije bilo kasnije ustanoviti moguće pogreške koje se javljaju pri integraciji sustava. Dobro napisan *unit* test trebao bi slijediti ove principe:

- trebao bi biti automatski i imati mogućnost ponavljanja,
- trebao bi biti jednostavan za implementaciju,
- jednom kada se napiše, trebao bi ostati za buduća korištenja,
- bilo tko bi ga trebao moći pokrenuti,
- trebao bi se jednostavno pokretati i
- trebao bi se brzo izvoditi.

Važnost pisanja *unit* testova se nalazi u mogućnosti brzog i jednostavnog testiranja programskog koda koji je napisan. Na ovaj način lako je otkriti koji *unit* trenutno ne radi ispravno kako je zamišljeno. Putem ovog načina testiranja moguće je pronaći greške vrlo rano, pri pisanju koda te na temelju njih zaključiti što bi bilo dobro promijeniti kako se one kasnije ne bi ponavljale. Pri *unit* testiranju moguće je podijeliti načine testiranja na dvije vrste:

1. Testiranje temeljeno na stanju sustava
2. Testiranje temeljeno na interakciji

Testiranje temeljeno na stanju sustava se odnosi na testiranje pri kojem je najbitniji krajnji rezultat testiranja. Ovakav način testiranja provjerava točnost izvedenosti programskog koda provjeravajući stanja u kojima se sustav našao tijekom izvođenja testova. Testiranje temeljeno na interakciji je vrsta testiranja koja se odnosi na interakciju među objektima ili programskim

jedinicama koda. Ovim načinom testiranja najbitnije je provjeriti je li suradnja među jedinicama programskog koda ispravna i odvija li se ona na način koji je programer zamislio. Moguće je da sustav daje točne rezultate, iako nije ostvarena željena suradnja među jedinicama programskog koda što u daljnjem radu sustava može uzrokovati znatno veće probleme. Testiranje temeljeno na stanju sustava je jednostavnije za izraditi i provesti, stoga se najčešće pristupa ovom načinu testiranja na razini *unit* testiranja, dok se interakcija među jedinicama programskog koda radi na višoj razini testiranja, npr. testiranje temeljeno na interakciji.

Glavne gradivne jedinice pri pisanju *unit* testova su *mock* objekti i *stub-ovi. Mock object* je objekt koji oponaša rad pravog objekta, te on najčešće utvrđuje je li test prošao. Nakon što se test izvede utvrđuje se je li objekt radio ispravno, kako je to očekivano tijekom konkretnog izvođenja programskog koda. *Stub* je zamjena pravog koda koju je moguće kontrolirati tijekom izvođenja testa. Vrlo često jedinice programskog koda su međusobno povezane i gotovo je nemoguće izolirati ih na način da svaka jedinica odrađuje svoj posao i prosljeđuje rezultate koji su potrebni drugoj jedinici, stoga se u programskom kodu stvara međuovisnost. Koristeći *stub* moguće je testirati programski kod bez problema međuovisnosti jedinica programskog koda[12]. Osnovni pristup pisanju *unit* testova svodi se na pisanje testova i *stub*-ovanih funkcija. Testovi simuliraju rad programa pri predavanju pojedinih parametara samo funkciji, dok *stub*-ovanje simulira rad pozvanog *unit*-a. Osnovna provjera ispravnosti rada programskog koda izvodi se najčešće putem *assert* funkcija. *Assert* u prijevodu znači tvrditi, te se iz toga može zaključiti kako se ovom funkcijom provjerava tvrdnja. Tvrdnja je povratna vrijednost funkcije. Putem *assert*-a može se vidjeti je li test vratio očekivanu povratnu vrijednost, tvrdnju ili nije. Kasnije se testovi jednog *unit*-a objedinjuju u jedinstvenu testnu grupu kojom se testira pojedini *unit*. *Unit* testiranjem se postiže razbijanje većih funkcija u setove manjih funkcija koje izvode istu operaciju. Ovakav programski kod je lakši za održavanje jer se ovisnost *unit*-a time znatno smanjuje tj.omogućuje se modularnost i izmjenjivost programskog koda[13].

4.1. CMocka

Prije samog pisanja *unit* testova potrebno je odabrati okruženje za izradu istih. Tijekom istraživanja koje okruženje bi omogućilo najkvalitetnije testiranje za problem koji se razmatra u radu, najvažnija stavka je bila mogućnost pisanja *unit* testova u C programskom jeziku. Osim toga, važno je da testovi i samo testiranje zauzimaju što manje računalnih resursa jer se cjelokupni rad odnosi za testiranje ugradbenog računalnog sustava. Kako se testovi i

testiranje, koje je izrađeno i izvedeno u ovom diplomskom radu, odnose na ugradbene sustave vrlo je bitno da ne zauzimaju puno resursa i memorije samog sustava.

Tab. 4.1. *Rezultati istraživanja za odabir okruženja unit testiranja*

Kriterij	**CUnit**	**CuTest**	**MinUnit**	**Ceedling**	**CMocka**
Mock objects	-	-	-	+	+
Asserts	+	+	+	+	+
Rukovanje iznimkama za signale	Rukovanje greškama za funkcije i tipove podataka	-	-	-	+
Testiranje curenja memorije	-	-	-	-	+
Mogućnost zapisa krajnjih rezultata u više formata	samo *XML*	-	-	-	+
Kreiranje rasporeda testiranja	-	-	-	-	+
Dobra dokumentacija i podrška	+	+	-	+	+

Tablica 4.1. prikazuje rezultate istraživanja koje je provedeno u svrhu odabira okruženja koje bi pružilo najpogodnije *unit* testiranje. Iz tablice je vidljivo da CMocka okruženje ima najpogodnije karakteristike te je odabran za pisanje *unit* testova. Najbitnije stavke pri usporedbi okruženja su bile da su otvorenog koda, imaju *mock* i *assert* funkcije, mogućnost zapisa rezultata u više formata, te dobru dokumentaciju i podršku.

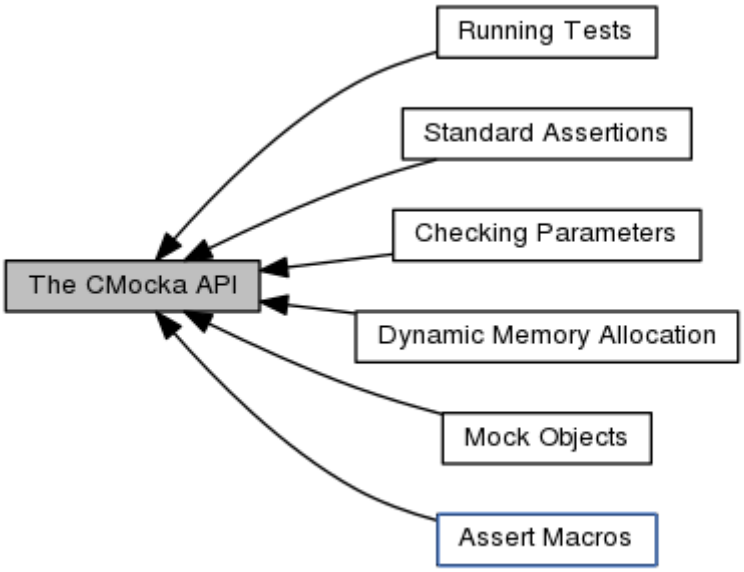
CMocka je jednostavno okruženje za *unit* testiranje u C programskom jeziku koje podržava *mock objects*. Za korištenje i pisanje samih testova potrebno je koristiti samo standradnu C biblioteku, te radi na nizu računalnih platformi uključujući i ugradbene računalne sustave. CMocka daje podršku za *mock object-e* koji su simulacija stvarnih objekata programskog koda te oponašaju implementaciju i njihov rad. Za izradu testova potrebna je samo C biblioteka što omogućuje rad CMocka-e na ugradbenim računalnim sustavima. Nakon zapisanih testova dolazi se do izvođenja testova. Izlazne vrijednosti nakon izvedenih testova

potrebno je dokumentirati kako bi kasnije bila moguća analiza i pregled istih. Putem ovog *framework*-a omogućeni su različiti izlazni formati. Izlazni format koji je korišten za potrebe ovog diplomskog rada je *XML*. Jedan od najbitnijih razloga zbog kojeg je CMocka odabrana za pisanje *unit* testova je vrlo dobra podrška i dokumentacija. Potpuno dokumentiran API je omogućen putem web stranice CMocka-e[14]. Ovdje je moguće pronaći pojašnjenja pojedinih koncepata Cmock-e i njezinih funkcija. Postoji podrška za rukovanje signalima (SIGSEGV, SIGILL...) i CMocka je u stanju oporaviti stanje testa, ako postoje iznimke poput *segfault.* Ujedno je vrlo dobro testirana platforma, temelji se na Google-ovoj platformi *cmockery* te je otvorenog koda.

Testovi napisani putem CMocka-e programski se prevode u samostalne datoteke koje se mogu izvoditi i povezane su sa CMocka bibliotekom, standardnom C bibliotekom i modulom koji se testira. Bilo kakvi vanjski simboli koji se koriste moraju se *mock*-ati, tj. zamijeniti funkcijama koje vraćaju određene povratne vrijednosti kako bi test bio izvediv. Iako je moguće da postoji velika razlika u ciljanom okruženju izvođenja programskog koda i okruženju koje se koristi za testiranje, *unit* testiranje je važeće jer njegov cilj je testiranje logike modula programskog koda na funkcijskoj razini, a ne svih mogućih interakcija s ciljanim okruženjem za izvođenje. CMocka API dijeli se u šest dijelova:

1. Izvođenje testova
2. Osnovni *assert*-i
3. Provjeravanje parametara
4. Dinamičko zauzimanje memorije
5. *Mock* objekti
6. *Assert* makronaredbe

Izvođenje testova daje prikaz načina kako je moguće izvoditi testove putem CMocka-e, dok osnovni *assert*-i prikazuju na koji način provjeravati tvrdnje osnovne C biblioteke. Provjeravanje parametara daje funkcionalnost za skladištenje očekivanih vrijednosti za *mock* parametre funkcija. Dinamičko zauzimanje memorije omogućava provjeru curenja memorije, presipavanje spremnika (engl. *buffer*) i sl. *Mock* objekti oponašaju rad stvarnih objekata programskog koda. *Assert* makronaredbe su niz vrlo korisnih *assert* makronaredbi[14]. Slika 4.1. daje prikaz ranije navedenih dijelova CMocka API-ja.

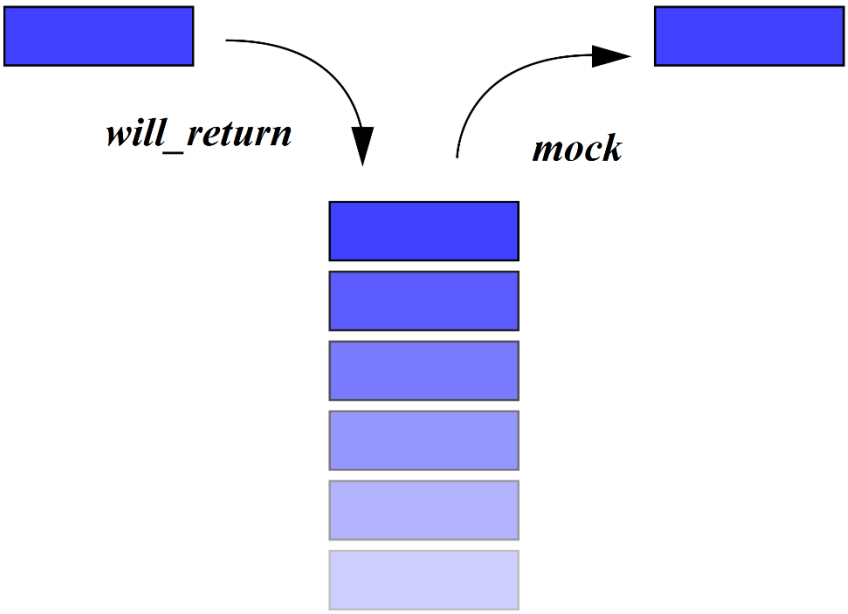


Sl. 4.1. *CMocka API* [14]

Najbitnija dva modula CMocka-e koja su korištena tijekom izrade diplomskog rada su *Mock objects* i *Assert Macros*. *Mock objects* sadrži funkcije kojima se oponaša rad stvarnih objekata programskog koda. Ovakav način rada je omogućen sljedećim funkcijama:

- *mock*,
- *mock_ptr_type*,
- *will_return*,
- *will_return_always* i
- *will_return_count.*

Will_return funkcije imaju zadaću spremanja vrijednosti u niz. Funkcija *will_return* sprema vrijednost u niz koju kasnije iz niza treba pročitati *mock* funkcija. Princip rada može se približiti putem prikaza načina rada *stack*-a te je vidljiv na slici 4.2. *Stack* je apstraktni tip podataka koji služi kao skup elemenata, te ima dvije osnovne operacije *push* i *pop*. *Will_return* funkcije se mogu smatrati kao *push* operacije, dok *mock* funkcije imaju ponašanje *pop* operacije. Prototip funkcija *will_return* zahtijeva funkciju za koju sprema vrijednost i samu vrijednost koju će spremiti. Prototip *mock* funkcije zahtijeva samo tip vrijednosti koji treba pročitati iz niza.

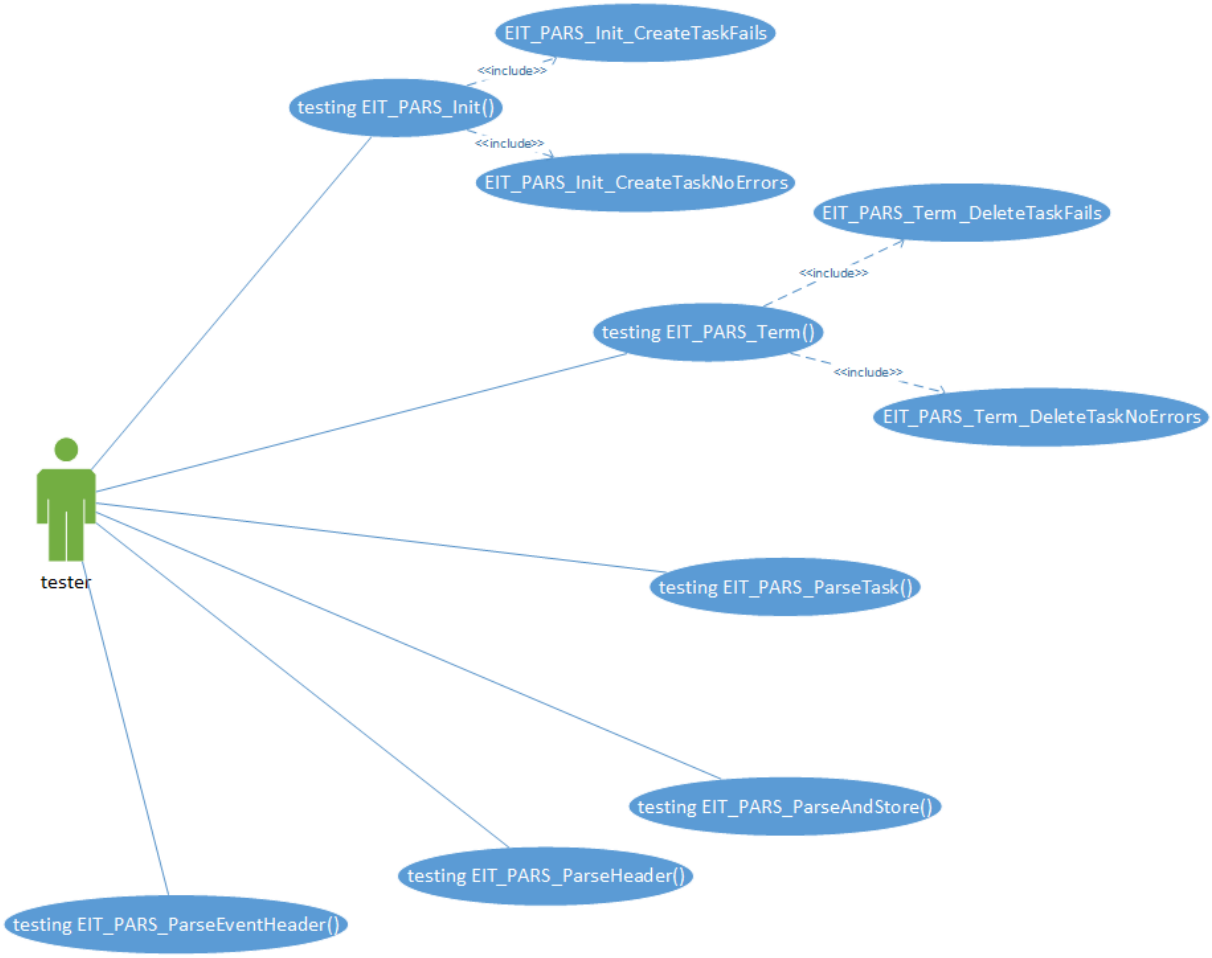


Sl. 4.2. *Prikaz principa rada will_return i mock funckija*

Assert Macros modul sadrži funkcije kojima je omogućeno uspoređivanje pojedinih parametara, tvrdnji i sl. Ovaj modul sadrži niz funkcija, no bit će istaknute samo neke:

- *assert_int_equal*,
- *assert_int_not_equal*,
- *assert_null*,
- *assert_string_equal* te
- *assert_string_not_equal*.

Putem ovih funkcija u testovima je omogućeno provjeravanje jesu li određeni parametri ispravni ili neispravni. Primjerice kod prolaska kroz funkciju generira se povratna vrijednost koju kasnije koristi određeni dio programskoga koda. Kako bi se moglo utvrditi je li ta povratna vrijednost ona koja je poželjna, moguće je koristiti *assert* funkcije. Usporedbom dobivene povratne vrijednosti i željene vrijednosti može se utvrditi radi li funkcija ispravno ili neispravno[14].



Sl. 4.3. *Prikaz dijagrama korištenja za unit testiranje modula za parsiranje paketa podataka*

Slika 4.3. prikazuje dijagram korištenja za izradu *unit* testova modula za obrađivanje paketa podataka. Vidljivo je da se testira svaka funkcija prema pojedinom *if* grananju koje sadrži. Analogno ovom dijagramu korištenja mogu se napraviti dijagrami korištenja za sve preostale module programske podrške TV aplikacije elektroničkog programskog vodiča.

Najčešće postoje dva dijela testa pri pisanju *unit* testova. Prvi dio testa provjerava ulazne parametre kako bi se provjerilo komunicira li dio programskog koda koji se testira ispravno, dok drugi dio testa vraća prethodno programirane izlazne vrijednosti s ciljem kako bi se utvrdilo kako programski kod koji se testira rukuje sa ispravnim i neispravnim slučajevima pri svojem radu. Ovakav način rada je u CMocka-i omogućen putem dvije makronaredbe:

- *will return(function, vaule)* - ova makronaredba dodaje vrijednost u niz *mock* vrijednosti. Cilj je da te vrijednosti koristi *unit* test, tijekom svog izvođenja i rada *mock objecta.*
- *mock()* - ova makronaredba uzima vrijednost iz niza *mock* vrijednosti. Korisnik *mock()* makronaredbe je *mocked* object koji koristi ovu makronaredbu kako bi znao što treba napraviti u testu.

Ova dva tipa makronaredbi moraju se koristiti u paru tijekom pisanja testova, inače će CMocka biblioteka smatrati da je test pao, ako postoji više vrijednosti koje se nalaze u nizu nego što ih je povučeno iz niza za korištenje i rad testa. Isto će se dogoditi, ako je situacija obrnuta.

```
void Test_EIT_MEM_Malloc_NoErrors(void** state)
{
    uint8_t returnValue;
    uint32_t size;
    uint8_t* buffer = NULL;

    size = kEIT_MEM_SIZE;
    MEM_Unit_Init();
    will_return(__wrap_TKEL_LockMutex, TKEL_NO_ERR);
    will_return(__wrap_TKEL_UnlockMutex, TKEL_NO_ERR);

    returnValue = EIT_MEM_Malloc(size, (void**)&buffer);
    assert_int_equal(returnValue, eEIT_MEM_ALLOC_OK);
    MEM_Unit_Free((void**)&buffer);
    MEM_Unit_Term();
}
```

Sl. 4.4. *Prikaz unit testa*

Slika 4.4. prikazuje primjer jednog *unit* testa. Pri samom pisanju testova vrlo je važno jasno definirati ime testa kako bi bilo što jednostavnije prepoznati namjenu testa. Ovaj test vezan je za rukovanje memorijom i funkciju *malloc*, te se ispituje ispravnost funkcije *EIT_MEM_Malloc* kada nema grešaka. Na početku testa potrebno je inicijalizirati pojedine varijable što ovisi o testu i funkciji koja se testira. Pomoću *will_return* funkcija postavljaju se vrijednosti u niz. U ovom testu postavljaju se vrijednosti *TKEL_NO_ERR* za funkcije *TKEL_LockMutex i TKEL_UnlockMutex. Assert_int_equal* funkcija provjerava povratnu

vrijednost funkcije *EIT_MEM_Malloc* nakon završetka testa. Test treba biti ispravan i povratna vrijednost treba biti *eEIT_MEM_ALLOC_OK*.

```
TKEL_err __wrap_TKEL_LockMutex(TKEL_mutex_id MutexID)
{
    TKEL_err errorCode;

    if( MutexID != (TKEL_mutex_id)kEIT_MUTEXID)
    {
        return TKEL_BAD_ARG;
    }

    errorCode = mock_type(TKEL_err);

    return errorCode;
}
```

Sl. 4.5. *Prikaz stub funkcije*

Funkcije koje su predane kao parametri *will_return* funkciji potrebno je implementirati. Ovakve funkcije dobivaju prefiks **__wrap_**. Na temelju ovog prefiksa CMocka biblioteka prepoznaje da treba koristiti *stub* funkciju, a ne stvarnu. Implementacija *stub* funkcija najčešće se svodi na vraćanje vrijednosti koja je prethodno stavljena u niz putem *will_return* funkcije. Vraćanje vrijednosti odrađuje se putem *mock* funkcije. Slika 4.5. prikazuje *stub* funkciju *TKEL_LockMutex* i njezinu implementaciju. U nazivu funkcije vidljivo je korištenje prefiksa **__wrap_**. Na početku *stub* funkcije inicijaliziraju se parametri. Srž implementacije *stub* funkcije nalazi se u korištenju *mock_type* funkcije koja povlači vrijednost koju je prethodno u testu postavila funkcija *will_return*.

```
/**************************************************************************
 * Test Case Name     : Test_EIT_MEM_Malloc_NoErrors
 *
 * Description        : Test Case to check if the EIT_MEM_Malloc() function is working according to specifications,
 *                      when there are no errors.
 *
 * Step description   : 1. Initialize the EIT Memory Management Unit.
 *                      2. Set the 1st parameter of function EIT_MEM_Malloc() to value kEIT_MEM_SIZE.
 *                      3. Stub TKEL_LockMutex() to return TKEL_NO_ERR.
 *                      4. Stub TKEL_UnlockMutex() to return TKEL_NO_ERR.
 *                      5. Call the EIT_MEM_Malloc() function to allocate a block of memory in the memory pool.
 *                      6. Check the return value of the EIT_MEM_Malloc() function.
 *                      7. Free the previously allocated memory in step 5.
 *                      8. Terminate the EIT Memory Management Unit.
 *
 * Expected Results   : 1. The return value of the EIT_MEM_Malloc() function shall be eEIT_MEM_ALLOC_OK.
 **************************************************************************/
```

Sl. 4.6. *Prikaz scenarija unit testa*

Slika 4.6. prikazuje primjer scenarija testa za *Malloc* funkciju EIT Memory Management modula. Naziv testa je *Test_EIT_MEM_Malloc_NoErrors* i provjerava se ispravnost rada funkcije *Malloc* kada se sve izvrši bez pogreške. Opisani su testni koraci koje je potrebno napraviti kako bi se test napisao i izvršio. Na kraju opisa testa napisani su očekivani rezultati čijom provjerom utvrđujemo prolaznost testa.

```
int main(void)
{
    const struct CMUnitTest EIT_ACQ[] =
    {
        cmocka_unit_test(Test_EIT_ACQ_Init_CreateQueueFails),
        cmocka_unit_test(Test_EIT_ACQ_Init_CreateQueueNoErrors),
        cmocka_unit_test(Test_EIT_ACQ_Term_EnqueueFails),
        cmocka_unit_test(Test_EIT_ACQ_Term_GetMsgNumberFails),
        cmocka_unit_test(Test_EIT_ACQ_Term_DeleteQueueFails),
        cmocka_unit_test(Test_EIT_ACQ_Term_NoErrors),
        cmocka_unit_test(Test_EIT_ACQ_Term_MsgNumberNotZero),
        cmocka_unit_test(Test_EIT_ACQ_DmxInit_InvalidDemuxId),
        cmocka_unit_test(Test_EIT_ACQ_DmxInit_AllocateChannelFails),
        cmocka_unit_test(Test_EIT_ACQ_DmxInit_RegisterCallbackFails),
        cmocka_unit_test(Test_EIT_ACQ_DmxInit_SetChannelPidFails),
        cmocka_unit_test(Test_EIT_ACQ_DmxInit_NoErrors),
        cmocka_unit_test(Test_EIT_ACQ_DmxTerm_InvalidChannelId),
        cmocka_unit_test(Test_EIT_ACQ_DmxTerm_ControlChannelFails),
        cmocka_unit_test(Test_EIT_ACQ_DmxTerm_FreeChannelFails),
        cmocka_unit_test(Test_EIT_ACQ_DmxTerm_NoErrors),
        cmocka_unit_test(Test_EIT_ACQ_DmxStart_InvalidChannelId),
        cmocka_unit_test(Test_EIT_ACQ_DmxStart_AllocateFilterFails),
        cmocka_unit_test(Test_EIT_ACQ_DmxStart_SetFilterFails),
        cmocka_unit_test(Test_EIT_ACQ_DmxStart_ControlChannelFails),
        cmocka_unit_test(Test_EIT_ACQ_DmxStart_ControlFilterFails),
        cmocka_unit_test(Test_EIT_ACQ_DmxStart_NoErrors),
        cmocka_unit_test(Test_EIT_ACQ_DmxStop_InvalidFilterId),
        cmocka_unit_test(Test_EIT_ACQ_DmxStop_ControlFilterFails),
        cmocka_unit_test(Test_EIT_ACQ_DmxStop_FreeFilterFails),
        cmocka_unit_test(Test_EIT_ACQ_DmxStop_NoErrors),
        cmocka_unit_test(Test_EIT_ACQ_FreeSectionBuffer_FreeBufferFails),
        cmocka_unit_test(Test_EIT_ACQ_FreeSectionBuffer_NoErrors),
        cmocka_unit_test(Test_EIT_ACQ_FreeSectionBuffer_InvalidBuffer),
        cmocka_unit_test(Test_EIT_ACQ_ChannelCallback_NoErrors),
        cmocka_unit_test(Test_EIT_ACQ_ChannelCallback_BufferLengthZero),
        cmocka_unit_test(Test_EIT_ACQ_ChannelCallback_NullBuffer),
        cmocka_unit_test(Test_EIT_ACQ_ChannelCallback_EnqueueFails),
        cmocka_unit_test(TEST_EIT_ACQ_ChannelCallback_FreeBufferFails),
        cmocka_unit_test(TEST_EIT_ACQ_ChannelCallback_GetMsgNumberFails)
    };

    cmocka_set_message_output(CM_OUTPUT_XML);

    return cmocka_run_group_tests(EIT_ACQ, NULL, NULL);
}
```

Sl. 4.7. *Prikaz testne grupe*

Slika 4.7. prikazuje testnu grupu u kojoj se nalaze svi testovi koji su grupirani prema pojedinim kriterijima. Na početku kreiranja testne skele potrebno je postaviti naziv koji je u ovom slučaju *EIT_ACQ*. Funkcija *cmocka_unit_test* poziva pojedini test koji se tada izvršava, dok se pozivanje i pokretanje cijele testne skele odvija putem funkcije *cmocka_run_group_tests*. Funkcijom *cmocka_set_message_output* omogućeno je postavljanje formata za zapis rezultata izvršenog testiranja. Potrebno je naglasiti kako se testna grupa mora nalaziti unutar *main* funkcije C programa. Pokretanje *unit* testova obavlja se pokretanjem izvršne datoteke koja se prethodno mora programski prevesti i povezati. Programsko prevođenje i povezivanje izvršne datoteke odvija se putem *makefile*-a. On sadrži putanju do potrebnih biblioteka, funkcija te su u njemu navedene *wrap* funkcije koje se koriste pri toj izvršnoj datoteci. Također može navesti verzija programskog prevođenja koja može biti 32-bitna ili 64-bitna.

4.2. Kontinuirana integracija

Kontinuirana integracija (engl. *Continous Integration*) je razvojna praksa koja zahtijeva od programera da se integrira programski kod u zajedničko spremište nekoliko puta dnevno. Svaka prijava se tada potvrđuje automatizirano i programski kod se automatski programski prevodi. Ovim se omogućuje rano otkrivanje pogrešaka. Budući da se programski kod često integrira, zahtijeva znatno manje praćenja pogreške unazad kako bi se otkrio pravi uzrok greške. Time ostaje više vremena za izgradnju bitnijih stavki programske podrške.

Kontinuirana integracija je jeftina. Ako se ne slijedi kontinuirani pristup, bit će potreban dulji period između integracija verzija. Ujedno greške je teže pronaći, a sve to može dovesti do kašnjenja u rasporedu izrade projekta. Uobičajena praksa je da nakon što programer preda svoj kod u spremište, pokreće se programsko prevođenje projekta i stvara se izvršna datoteka.

Dobre prakse pri korištenju kontinuirane integracije su[15]:

- održavati jedno izvorno spremište,
- automatizirati programsko prevođenje,
- učiniti vašu datoteku samo-testirajuću,
- ubrzati programsko prevođenje,
- omogućiti lak pristup zadnjim verzijama izvršne datoteke,
- omogućiti da svatko može vidjeti što se događa i
- automatizirati raspoređivanje.

4.2.1. Aplikacija za vođenje projekta

Aplikacija za vođenje projekta (engl. *Application Management Tool*) služi za rad na cijelom projektu. Omogućava pisanje zahtjeva za programsku podršku, kreiranje programske arhitekture, programiranje, testiranje programske podrške, održavanje programske podrške i sl. Omogućuje lakše pretvaranje ideje u stvarnost. Omogućava nadziranje projekta u svim dijelovima izrade, počevši od stvaranja ideje i zahtjeva do konačnog proizvoda – programske podrške.

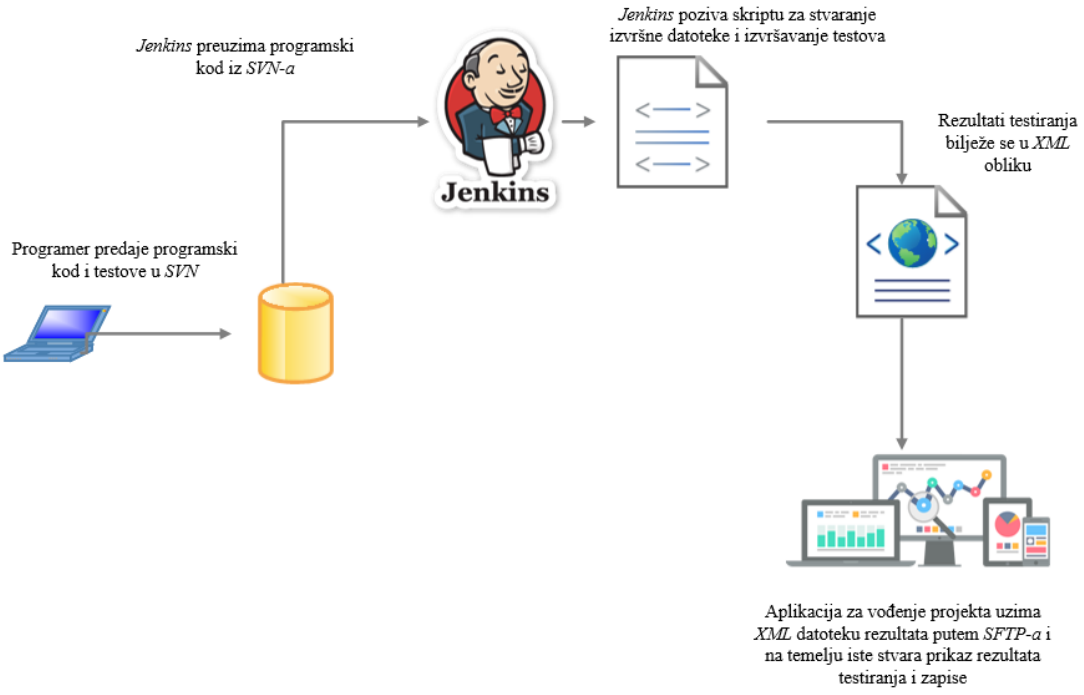
4.2.2. Jenkins

Jenkins je aplikacija otvorenog koda koja omogućuje kontinuiranu integraciju i kontinuiranu predaju projekata bez obzira o kojoj je platformi riječ. Može se uklopiti s nizom tehnologija testiranja. Instalira se na poslužitelj na kojem se odvija središnje programsko prevođenje. Pruža mogućnost *unit* testiranja i automatskog testiranja. Za *unit* testiranje postoje već ugrađene biblioteke, npr. *JUnit*, ali je moguće dodati druge biblioteke. Automatsko testiranje omogućava testiranje više puta kako bi se objektivno moglo zaključiti je li projekt spreman za nastavak u idući dio razvoja. Omogućuje analizu kvalitete koda aplikacije na način da uključuje dodatne alate poput *findbugs*, *checksytle* i *PMD*. Prati upozorenja programskog prevoditelja i *TODO* komentare koje programer zapisuje tijekom rada. Jedna od bitnih stavki Jenkis-a su pametne obavijesti. Obavijesti se šalju samo kada postoje promjene u stvaranju izvršne datoteke i ta promjena se bitno razlikuje od prethodne. Ovakvim načinom rada Jenkis traži pozornost programera kada je to stvarno potrebno. Obavijest se mogu slati pojedinačno, npr. programeru koji je zadnji integrirao programski kod u zajedničko spremište. Sigurnost programskog koda je osigurana od strane *Jenkins*-a putem kontrole ulaza za projekt i *LDAP* protokolom. Omogućeno je praćenje integriranja prema pojedinim programerima i zakazivanje novog izrađivanja izvršne datoteke[16].Tijekom izrade testiranja za potrebe TV aplikacije elektronskog vodiča *Jenkins* je korišten u svrhe automatskog testiranja i zakazivanja izgradnje nove izvršne datoteke.

4.2.3. Automatizacija testiranja

Aplikacija za vođenje projekta i *Jenkins* omogućuju stvaranje okruženja za automatsko testiranje i pregled rezultata testiranja. Programi tijekom izrade programske podrške integriraju svoj programski kod putem alata za kreiranje novih verzija programske podrške i pregleda programskog koda. Pri izradi ovog diplomskog rada korišten je *SVN*. Putem ovoga je

omogućeno programerima da budu detaljno upućeni u zadnje verzije rada ostalih članova projekta i povijest promjena vršenih tijekom izrade programske podrške. Nakon što je kod u *SVN*-u provjerava se verzija koda. *Jenkins* je postavljen na način da svake noći u točno određeno vrijeme preuzme posljednju verziju izvršne datoteke na *SVN*-u i pokrene ju. Unutar izvršne datoteke mogu se nalaziti testovi za programsku podršku. U ovom slučaju radi se o *unit* testovima koji se izvedu i vraćaju *XML* kao rezultat testiranja. Nakon toga rezultati se šalju aplikaciju za vođenje projekta koja obrađuje podatke iz *XML*-a i stvara zapis odgovarajućih izvedenih testova. Slika 4.8. prikazuje proces automatizacije testiranja pri ovom diplomskom radu.



Sl. 4.8. *Prikaz automatizacije testiranja*

5. REZULTATI TESTIRANJA I ANALIZA

Cilj cjelokupnog testiranja je utvrditi ispravnost rada programske podrške TV aplikacije elektronskog programskog vodiča. Na temelju testiranja prikupljeni su rezultati koji će u ovom poglavlju biti pojašnjeni i prikazani.

5.1.Rezultati programskog profiliranja

Testiranja su obavljena pomoću Valgrind-a, čiji je opis dan u drugom poglavlju zajedno s alatima koji su njegov sastavni dio. Daljnja analiza rezultata bit će obrađena prema pojedinom alatu Valgrind-a koji je korišten.

5.1.1.Memcheck

Tab. 5.1. *Sažetak Memcheck rezultata*

Sažetak Memcheck rezultata	
definitely lost	48 bajtova u 2 bloka
indirectly lost	0 bajtova u 0 blokova
possibly lost	272 bajtova u 2 bloka
still reachable	108 bajtova u 2 bloka
suppressed	0 bajtova u 0 blokova

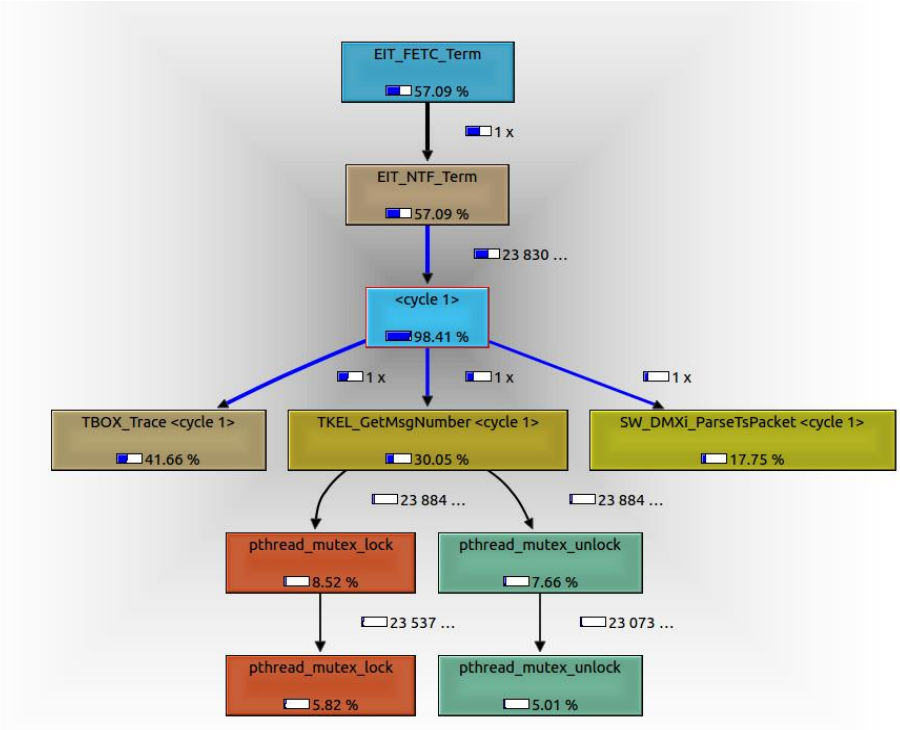
Rezultati testiranja pokazuju kako je sigurno izgubljeno 48 bajtova u 2 bloka memorije, dok je još moguće izgubljenih 272 bajta u 2 bloka. 108 bajtova u 2 bloka je moguće još dohvatiti. Vidljivo je da su određeni bajtovi izgubljeni tijekom rada programa što nije očekivano pri radu programa.

5.1.2.Callgrind

Tab. 5.2. *Sažetak Callgrind rezultata*

Incl.	Self	Called	Funkcija
6.75%	6.42%	27321302	*pthread_mutex_lock*
5.93%	0.50%	27321301	*pthread_mutex_unlock*
98.41%	76.72%	23868866	*<cycle1>*
30.05%	13.58%	23884178	*TKEL_GetMsgNumber*

Rezultati testiranja putem alata Callgrind prikazani su tablicom 5.2. prikazuju kako funkcije *pthread_mutex_lock*, *pthread_mutex_unlock*, *<cycle1>* i *TKEL_GetMsgNumber* uzimaju najviše vremena pri izvođenju programa. Niz funkcija *<cycle>* je dio programskog koda u kojem se odvija najveći dio rada. Očekivano je da niz funkcija *<cycle>* te funkcije *pthread_mutex_lock* i *pthread_mutex_unlock* uzimaju najviše vremena pri izvođenju programa jer se najčešće i koriste. Funkcije *pthread_mutex_lock* i *pthread_mutex_unlock* prethode gotovo svim funkcijama pri izvođenju kako bi omogućile sigurno izvođenje programa. Slika 5.1. grafički prikazuje rezultate iz tablice 5.2.



Sl. 5.1. *Grafički prikaz rezultata Callgrinda*

5.1.3.Cachegrind

Rezultati testiranja alatom Cachegrind prikazuju koje funkcije najviše rade sa priručnom memorijom. Tablicom 5.3. prikazan je sažetak rezultata Cachegrind-a. Vidljivo je da funkcija *SW_DMXi_ParseTsPacket* najviše radi sa priručnom memorijom. Ta funkcija je dio korištenog API-ja za dohvaćanje paketa iz prijenosnoga toka. Funkcija *TBOX_Trace* služi za ispisivanje podataka tijekom rada TV aplikacije Elektronskog Programskog Vodiča.

Tab. 5.3. *Sažetak Cachegrind rezultata*

Self	**Funkcija**
42.08%	*SW_DMXi_ParseTsPacket*
34.61%	*TBOX_Trace*
2.77%	*__memcpy_ssse3*
2.74%	*TKEL_LockMutex*

5.1.4.DRD

Testiranje alatom DRD omogućuje pregledavanje rada programa s nitima. Rezultati ovog testiranja pokazali su kako postoji 226 pogrešaka iz 34 konteksta. Većina pogrešaka vezana je uz sukobe među otključavanju i zaključavanju muteksa.

5.1.5.Massif

Napravljeno je 74 snimaka te je vidljivo da je glavna funkcija koja zauzima memoriju na *heap*-u *TKEL_Malloc*. Najviše je zauzeto 20.1 MB memorije. Ovaj rezultat odgovara očekivanom rezultatu. Modulom za rukovanje memorijom je određeno da 20.1 MB moguće zauzeti tijekom rada programa. Slika 5.2. prikazuje zauzeće *heap*-a i funkcije koje koriste *heap*.

Sl. 5.2. *Grafički prikaz zauzeća heap-a*

5.2. Rezultati unit testiranja

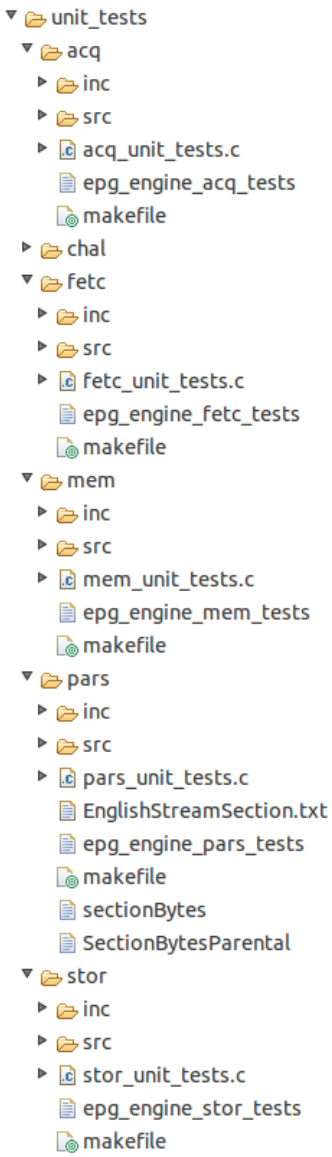
Tijekom izrade diplomskog rada napisano je 277 *unit* testova. Svrha *unit* testiranja je u tome da se u ranoj fazi razvoja programske podrške otkriju pogreške. Testovi su pisani prema pojedinim modulima programskoga koda. CMocka sadrži određena ograničenja zbog kojih su nastajali problemi tijekom pisanja *unit* testova. Naime, CMocka omogućuje stvaranje *stub* funkcija koje se pozivaju tijekom izvođenja testiranja umjesto stvarnih funkcija, ali zbog složenosti programskoga koda to u pojedinim slučajevima nije bilo moguće. Moduli bi načelo trebali biti neovisni, no to ponekad nije u potpunosti moguće izvesti. Pojedini moduli sadrže funkcije koje tijekom svojeg izvođenja pozivaju druge funkcije. Problem nastaje kada se te funkcije nalaze u istom modulu tj. u istom dokumentu. CMocka tada ne prepoznaje **__wrap_** prefiks, te dolazi do pozivanja stvarne funkcije, a ne *stub*-ovane. Problemi su također nastajali pri testiranju funkcija kod kojih osim povratnih vrijednosti je vrlo bitno da izvrše i ostali dio svoje implementacije, npr. da stvarno zauzmu komad memorije. Svi nastali problemi pri *unit* testiranju su uspješno razriješeni, te su svi testovi uspješno izvršavani.

Napisani su testovi za pet modula:

1. Modul za obrađivanje paketa podataka,

2. Modul za dohvaćanje paketa podataka,
3. Modul za rukovanje memorijom,
4. Modul za skladištenje podataka i
5. Modul za komunikaciju s klijentom i dohvaćanje potrebnih podataka klijentu.

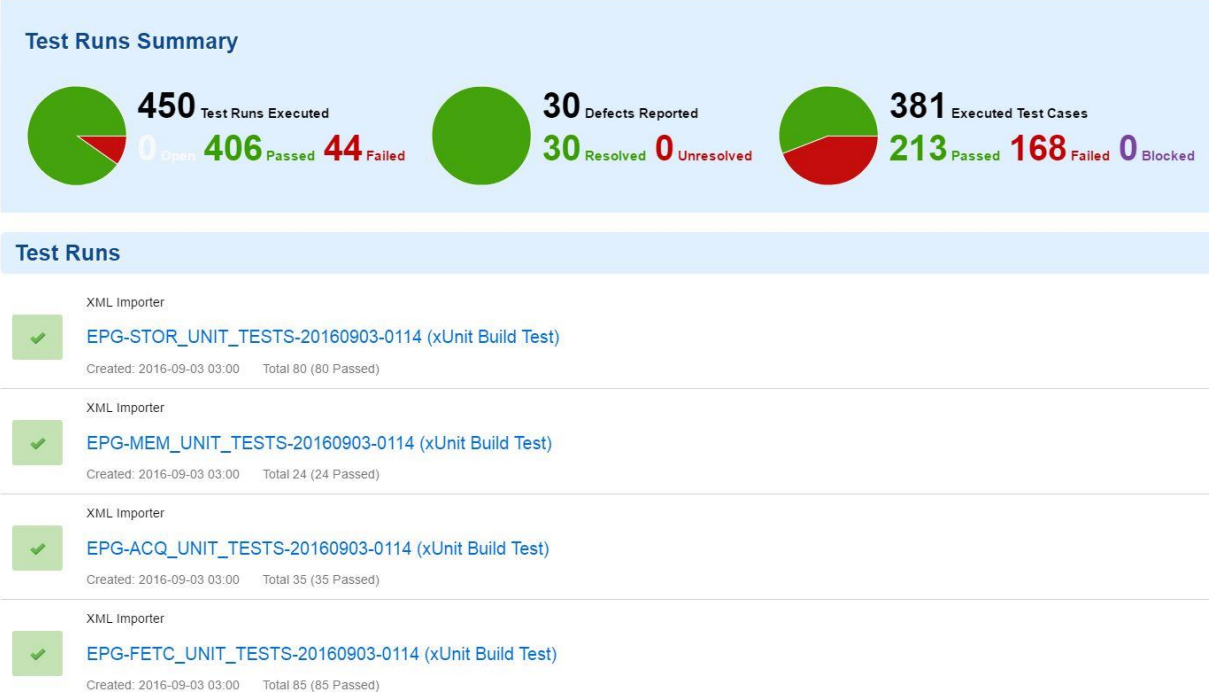
Slika 5.3. prikazuje strukturu popisa testova za sve navedene module.



Sl. 5.3. *Prikaz popisa modula i unit testova*

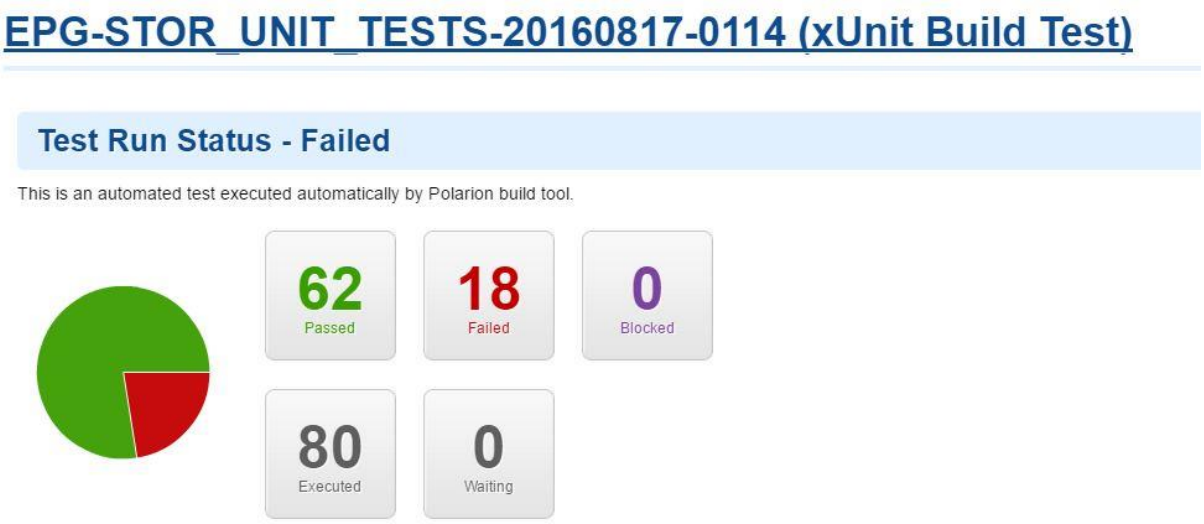
Zapis testova u aplikaciji za vođenje projekta bit će prikazan i pojašnjen na slikama koje slijede u ovom potpoglavlju.

Na slici 5.4. je vidljiv ukupni broj probnih radova, broj probnih radova koji su prošli te broj probnih radova koji nisu prošli. Bilježe se problemi koji su pronađeni te aplikacija za vođenje projekta javlja da bi ih trebalo otkloniti. Vidljiv je ukupni broj testnih slučajeva te broj testnih slučajeva koji su prošli i koji nisu. Ukupni broj probnih radova iznosi 450 od kojih je 406 uspješno, a 44 neuspješno. Tijekom rada dogodilo se 30 problema i svi su razriješeni. Izveden je 381 testni slučaj od kojih je 213 uspješno, a 168 neuspješno.



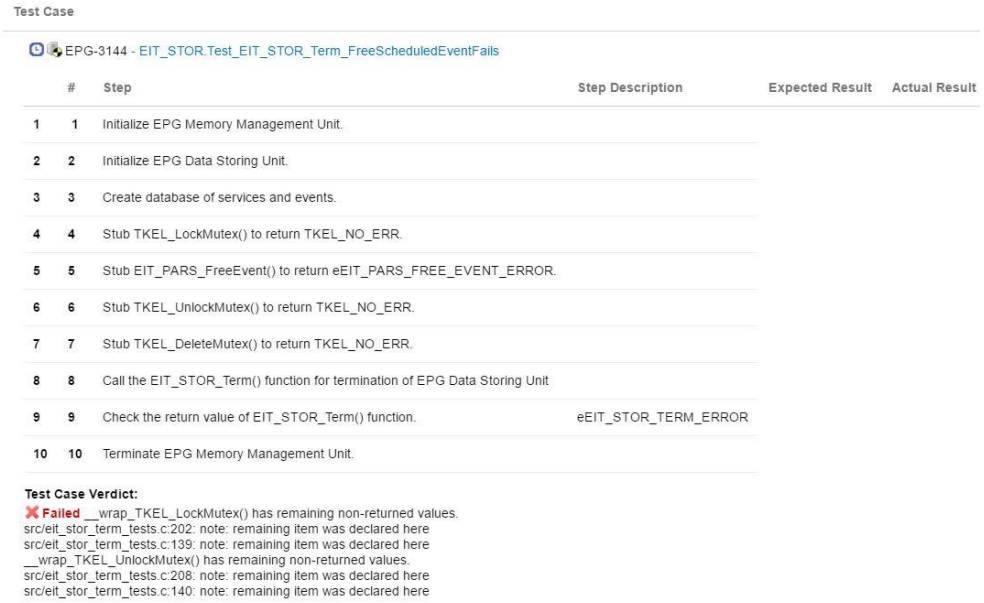
Sl. 5.4. *Prikaz odrađenih unit testova*

Slika 5.5. prikazuje broj odrađenih testova jedinice za spremanje. Vidljivo je da je odrađeno 80 testova gdje je 63 testa prošlo, dok 18 nije prošlo. Na temelju toga aplikacija za vođenje projekta javlja kako taj probni rad nije prošao. Probni rad programske podrške smatra se prolaznim, ako su svi testovi prošli.



Sl. 5.5. *Prikaz odrađenih unit testova za jedinicu za spremanje podataka*

Slika 5.6. prikazuje opis *unit* testa. Vidljivi su koraci testa koji se provode tijekom testiranja te očekivani rezultat na kraju testa. Opis odrađenog testa nalazi se na dnu slike 4.7. gdje je vidljivo da ovaj test nije prošao i dan je opis problema koji je to uzrokovao. Razlog zbog kojeg je test na slici 4.7. neuspješan je zbog nevraćenih vrijednosti funkcija *TKEL_LockMutex* i *TKEL_UnlockMutex*.



Sl. 5.6. *Prikaz unit testa s opisom*

Slika 5.7. daje prikaz testnih slučajeva jedinice za spremanje koji se izvode u *Jenkins*-u. Prikaz daje ocjenu je li test prošao ili nije, naziv testnog slučaja, problem koji uzrokuje, vrijeme izvođenja, alat koji ga je izveo te točan datum i vrijeme izvođenja.

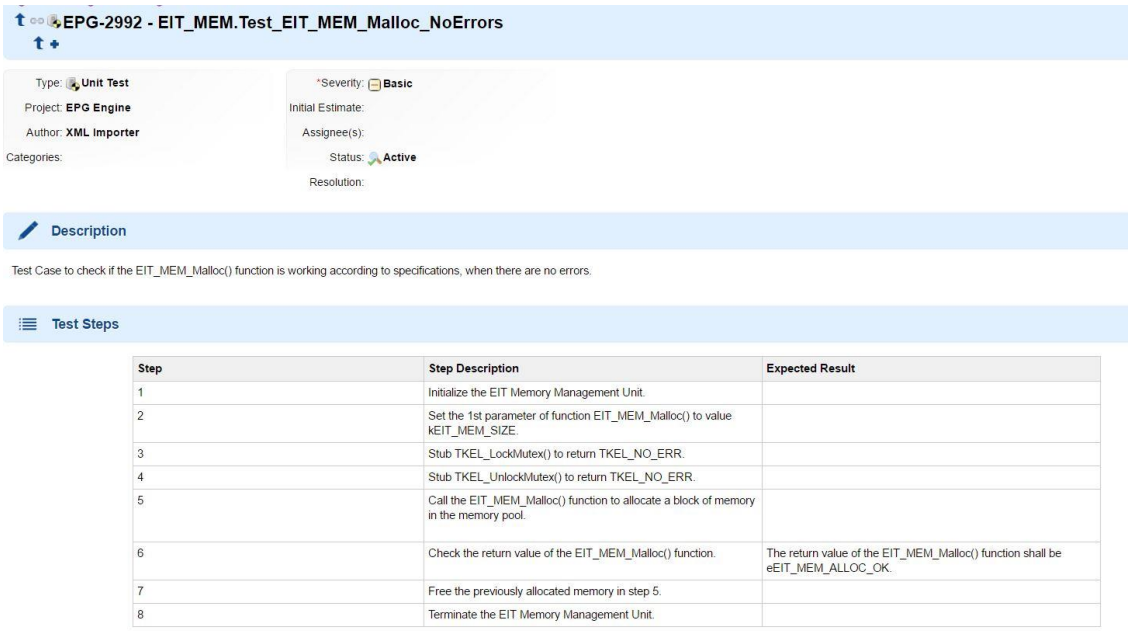
Tests - EPG-STOR_UNIT_TESTS-20160817-0114 (xUnit Build Test)

Test Result	Test Case	Defect	Duration	Executed by	Executed
Passed	EPG-3141 - EIT_STOR.Test_EIT_STOR_Init_CreateMutexFails		0.063 s	XML Importer	2016-08-17 03:00
Passed	EPG-3142 - EIT_STOR.Test_EIT_STOR_Init_NoErrors		0.052 s	XML Importer	2016-08-17 03:00
Passed	EPG-3143 - EIT_STOR.Test_EIT_STOR_Term_LockMutexFails		0.048 s	XML Importer	2016-08-17 03:00
Failed	EPG-3144 - EIT_STOR.Test_EIT_STOR_Term_FreeScheduledEventFails	EPG-4045	0.262 s	XML Importer	2016-08-17 03:00
Failed	EPG-3145 - EIT_STOR.Test_EIT_STOR_Term_FreeOfCurrentDayFails	EPG-4045	0.154 s	XML Importer	2016-08-17 03:00
Failed	EPG-3146 - EIT_STOR.Test_EIT_STOR_Term_FreeOfFollowingEventFails	EPG-4045	0.149 s	XML Importer	2016-08-17 03:00
Failed	EPG-3147 - EIT_STOR.Test_EIT_STOR_Term_FreeOfPresentEventFails	EPG-4045	0.147 s	XML Importer	2016-08-17 03:00
Failed	EPG-3148 - EIT_STOR.Test_EIT_STOR_Term_FreeOfCurrentServiceFails	EPG-4045	0.146 s	XML Importer	2016-08-17 03:00
Failed	EPG-3149 - EIT_STOR.Test_EIT_STOR_Term_UnlockMutexFails	EPG-4045	0.144 s	XML Importer	2016-08-17 03:00
Failed	EPG-3150 - EIT_STOR.Test_EIT_STOR_Term_DeleteMutexFails	EPG-4045	0.157 s	XML Importer	2016-08-17 03:00
Failed	EPG-3151 - EIT_STOR.Test_EIT_STOR_Term_NoErrors	EPG-4045	0.150 s	XML Importer	2016-08-17 03:00
Passed	EPG-3152 - EIT_STOR.Test_EIT_STOR_AddService_NoErrors		0.121 s	XML Importer	2016-08-17 03:00
Passed	EPG-3153 - EIT_STOR.Test_EIT_STOR_AddService_ServiceIsNull		0.046 s	XML Importer	2016-08-17 03:00
Passed	EPG-3154 - EIT_STOR.Test_EIT_STOR_AddService_LockMutexFails		0.049 s	XML Importer	2016-08-17 03:00

Sl. 5.7. *Prikaz unit testova jedinice za spremanje*

Slika 5.8. daje prikaz *unit* testa jedinice za memoriju. Prikazani su koraci testa koji se izvodi i očekivani rezultat na kraju izvođenja, opis testa. Na vrhu slike je vidljivo da je moguće postaviti razinu važnosti testa i dan je prikaz statusa testa. Test se sastoji od 8 koraka i očekivani rezultat na kraju testa je *eEIT_MEM_MALLOC_OK*. Koraci koji se izvode u testu su:

1. Inicijalizacija modula za rukovanje memorijom
2. Postavljanje prvog parametra funkcije na vrijednost *kEIT_MEM_SIZE*
3. *Stub*-ovanje funkcije *TKEL_LockMutex* da vrati vrijednost *TKEL_NO_ERR*
4. *Stub*-ovanje funkcije *TKEL_UnlockMutex* da vrati vrijednost *TKEL_NO_ERR*
5. Pozivanje funkcije *EIT_MEM_Malloc*
6. Provjera povratne vrijednosti funkcije *EIT_MEM_Malloc*
7. Oslobađanje prethodno zauzete memorije u testu
8. Deinicijalizacija modula za rukovanje memorijom



EPG-2992 - EIT_MEM.Test_EIT_MEM_Malloc_NoErrors

Type: Unit Test
Project: EPG Engine
Author: XML Importer
Categories:

*Severity: Basic
Initial Estimate:
Assignee(s):
Status: Active
Resolution:

Description

Test Case to check if the EIT_MEM_Malloc() function is working according to specifications, when there are no errors.

Test Steps

Step	Step Description	Expected Result
1	Initialize the EIT Memory Management Unit.	
2	Set the 1st parameter of function EIT_MEM_Malloc() to value kEIT_MEM_SIZE.	
3	Stub TKEL_LockMutex() to return TKEL_NO_ERR.	
4	Stub TKEL_UnlockMutex() to return TKEL_NO_ERR.	
5	Call the EIT_MEM_Malloc() function to allocate a block of memory in the memory pool.	
6	Check the return value of the EIT_MEM_Malloc() function.	The return value of the EIT_MEM_Malloc() function shall be eEIT_MEM_ALLOC_OK.
7	Free the previously allocated memory in step 5.	
8	Terminate the EIT Memory Management Unit.	

Sl. 5.8. *Prikaz unit testa jedinice za memoriju i njegovog opisa*

5.3. Zaključak na temelju rezultata testiranja

Rezultati programskog profiliranja prikazuju da postoje određeni problemi pri rukovanju memorijom što je prikazano rezultatima Memcheck-a. Ovi rezultati nisu očekivani i prema njima bi trebalo prepraviti kod TV aplikacije Elektronskog Programskog Vodiča kako ne bi bilo curenja memorije. Callgrind i Cachegrind rezultati su očekivani, dok rezultati testiranja putem DRD-a prikazuju određene probleme pri otključavanju i zaključavanju muteksa. Iako su problemi vidljivi u izvještaju, tijekom rada programa nije bilo uočljivih pogrešaka koji su uzrokovane tim tipom problema. Rezultati testiranja putem Massif alata su očekivani. *Unit* testiranje ukazalo je na određene pogreške tijekom izrade TV aplikacije Elektronskog Programskog Vodiča. Uočene pogreške bile su izazvane pogreškama pri programiranju npr. predavanjem krive povratne vrijednosti i slično, dok pogrešaka koje su ugrožavale funkcionalnost i rad programa nije bilo.

6. ZAKLJUČAK

Testiranje je vrlo bitan dio razvoja bilo kakvog sustava pa tako i razvoja programske podrške. Današnje tvrtke zahtijevaju mjere sigurnosti i pouzdanosti na vrlo viskom nivou jer time pridonose svome ugledu, ali istovremeno omogućuju profit. U okviru diplomskog rada analizirana su dva različita načina testiranja, programsko profiliranje i *unit* testiranje. Programsko profiliranje je *black-box* testiranje gdje nije potrebno poznavati implementaciju kako bi se testirala programska podrška, dok je tijekom izrade *unit* testova bilo potrebno poznavati implementaciju i način rada cjelokupnog sustava kako bi testovi bili što kvalitetnije napisani – *white-box* testiranje.

Testiranje koje je obavljeno putem programskog profiliranja izrađeno je putem Valgrinda. Izrađeni su testovi vezani uz memoriju, rad sa priručnom memorijom, broj poziva funkcija, višenitnost i rukovanje *heap*. Najbitnije tijekom testiranja bilo je uočiti moguća curenja memorije. Prema dobivenim rezultatima vidi se kako postoje određeni gubitci memorije koje je potrebno otkloniti. Rad sa priručnom memorijom i broj poziva funkcija u ovom testiranju poslužili su kako bi se stekao uvid u brzinu izvođenja programa, te postoje li određeni zastoji tijekom rada. Rezultati su pokazali kako program radi vrlo efikasno i brzo. Mogućnost apsolutnog mjerenja vremena izvođenja programa nije bila moguća. C programski jezik sadrži biblioteke koje omogućuju mjerenje brzine izvođenja programa, no to bi zahtijevalo stvaranje izmjena u izvornom programskom kodu što na ovom projektu nije bilo moguće. Tijekom izrade diplomskog rada nije pronađeno programsko okruženje u kojem bi bilo moguće testirati stvarno vrijeme izvođenja programa. Testiranje višenitnosti izvedeno je uspješno, te pri dobivenim rezultatima nisu uočeni problemi pri radu same programske podrške. Programsko profiliranje *heap*-a omogućilo je uvid u zauzeće virtualne memorije. Rezultati su pokazali kako je rad sa *heap*-om ispravan te da se zauzima točno onoliko memorije koliko je predviđeno pri pisanju programskog koda.

Unit testiranje izrađeno je putem programskog okruženja CMocka. Ovo programsko okruženje omogućilo je pisanje *unit* testova na vrlo jednostavan način putem C programskog jezika. Korištenje *stub* i *mock* funkcija omogućilo je izoliranje pojedinih modula kako bi se mogli ispravno testirati. Tijekom izrade *unit* testova pojavljivale su se pogreške, no to je cilj testiranja – testirati, pronaći pogrešku i ispraviti ju. Budući da su sve pogreške unutar programskog koda ispravljene, na kraju izrade diplomskog rada svi *unit* testovi su bili ispravni i prošli su. Izrađeno je 277 testova za pet modula unutar programske podrške.

Problemi su nastajali pri testiranju pojedinih funkcija zbog ograničenja CMocka-e, prethodno spomenuto u poglavlju 4.

Kako bi se uvjerili u potpunu ispravnost programske podrške i njezin kvalitetan rad, potrebno je načiniti još testiranja. U poglavlju 1. spomenuti su *integration* testiranje i *system* testiranje, no oni nisu bili tema ovoga diplomskog rada. Budući da je izrada programske podrške TV aplikacije elektroničkog programskog vodiča dinamičan projekt, neke od vrsta testiranja tijekom izrade diplomskog rada nije bilo moguće izvesti zbog čestih promjena i dorada koda.

LITERATURA

[1] G. J. Myers, C. Sandler, T. Badgett, The Art of Software Testing, Wiley, Canada, 2012.

[2] D.Cohen, M.Lindvall, P.Costa, Agile Software Development, Fraunhofer Center Maryland, USA, 2003.

[3] http://scrummethodology.com/, pristup ostvaren 21.07.2016.

[4] E.A.Lee, What are the Key Challenges in Embedded Software?, System Design Frontier, vol.2, br.1, siječanj 2005.

[5] https://en.wikibooks.org/w/index.php?title=Special:Book&bookcmd=download&collection_id=a3e52487426053c4c05d925be350012b7e3180bd&writer=rl&return_to=Wikibooks%3ACollections%2FEmbedded+Systems, pristup ostvaren 25.07.2016.

[6] https://msdn.microsoft.com/en-us/library/ms859408.aspx, pristup ostvaren 05.08.2016.

[7] http://valgrind.org/docs/manual/mc-manual.html, pristup ostvaren 11.06.2016.

[8] http://valgrind.org/docs/manual/cg-manual.html, pristup ostvaren 15.06.2016.

[9] http://valgrind.org/docs/manual/cl-manual.html, pristup ostvaren 20.06.2016.

[10] http://valgrind.org/docs/manual/drd-manual.html, pristup ostvaren 29.06.2016.

[11] http://valgrind.org/docs/manual/ms-manual.html, pristup ostvaren 07.08.2016.

[12] R.Osherove, The art of unit testing with Examples in .NET, Manning, Greenwich – USA, 2009.

[13] http://softwaretestingfundamentals.com/unit-testing/,pristup ostvaren 20.08.2016.

[14] https://api.cmocka.org/, pristup ostvaren 25.08.2016.

[15] https://www.thoughtworks.com/continuous-integration, pristup ostvaren 02.09.2016.

[16] http://www.tutorialspoint.com/jenkins/, pristup ostvaren 07.09.2016.

SAŽETAK

Testiranje je u današnjoj industriji vrlo bitna stavka razvoja projekta. Unutar ovog diplomskog rada izrađeno je programsko profiliranje i *unit* testiranje za potrebe testiranja TV aplikacije elektronskog programskog vodiča. Pri odabiru alata i programskih okruženja za testiranje sustava bilo je bitno da su otvorenog koda i pogodni za testiranje ugradbenih računalnih sustava. Programsko profiliranje se može svrstati u *black-box* testiranje i izrađeno je koristeći Valgrind. Valgrind je skup programskih alata koji omogućava testiranje memorije, rada s priručnom memorijom, brojanje poziva funkcija, višenitnost te rada sa *heap-om*. *Unit* testiranje se može svrstati u *white-box* testiranje te je izrađeno koristeći programsko okruženje CMocka. Ovakav način testiranja radi se putem *stub* i *mock* funkcija. Testiranje je izrađeno za ugradbeni računalni sustav, točnije za programsku podršku TV aplikacije elektroničkog programskog vodiča.

Ključne riječi: Testiranje, Valgrind, Unit testiranje, CMocka, Ugradbeni računalni sustavi

THE DEVELOPMENT OF ENVIRONMENT FOR AUTOMATIC TESTING AND DYNAMIC ANALYSIS OF ELECTRONIC PROGRAM GUIDE TV APPLICATION

ABSTRACT

Testing in the industry today is a very critical component of project. As a part of thesis software profiling and unit testing are made. When choosing tools and programming environments to test systems it was important that it are open source and suitable for testing embedded systems. The program profiling can be classified into black-box testing and is done using Valgrind. Valgrind is a collection of software tools that allows to test memory, working with the cache, counting function calls, multithreading and work with the heap. Unit testing can be classified into white-box testing and is done using a programming environment CMocka. This kind of testing works through the stub and mock function. Testing is designed for embedded systems, specifically for software application TV electronic program guide - EPG.

Keywords: Testing, Valgrind, Unit testing, CMocka, Embbeded Systems

ŽIVOTOPIS

Matko Turalija rođen je 24.02.1993. u Našicama gdje je završio Opću Gimnaziju u Srednjoj školi „Isidor Kršnjavi“. Nakon završetka srednje škole 2011. godine upisuje Elektrotehnički fakultet u Osijeku i odabire smjer računarstvo. Završetkom preddiplomskog studija 2014. godine stječe zvanje prvostupnika računarstva i upisuje diplomski studij, smjer procesno računarstvo. U rujnu 2015. godine postaje stipendist u Institutu RT-RK Osijek.