Okruženje za testiranje ADAS programskih rješenja

Barić, Davor

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Diplomski sveučilišni studij Automobilsko računarstvo i komunikacije

OKRUŽENJE ZA TESTIRANJE ADAS PROGRAMSKIH RJEŠENJA

Diplomski rad

Davor Barić

Osijek, 2019.

Sadržaj

1. UVOD 1
2. TESTIRANJE ADAS SUSTAVA I PROGRAMSKI ALATI 3
2.1. ADAS sustavi 3
2.2. Testiranje programske podrške 4
2.3. *Python* programski jezik 5
2.4. *SQLite* baza podataka u *Pythonu* 6
2.5. Paralelna obrada, višeprocesni i višenitni način rada 7
3. POSTOJEĆE OKRUŽENJE ZA TESTIRANJE ADAS PROGRAMSKIH RJEŠENJA 8
4. POBOLJŠANJE OKRUŽENJA ZA TESTIRANJE ADAS PROGRAMSKIH RJEŠENJA 10
4.1. Ideja i model rješenja 10
4.2. Programski dio rješenja 13
4.2.1. Detekcija promjena i dodavanje testova u bazu podataka 14
4.2.2. Dohvaćanje testova iz baze podataka 19
4.2.3. Rad s procesima 20
5. VERIFIKACIJA POBOLJŠANOG OKRUŽENJA ZA TESTIRANJE ADAS PROGRAMSKIH RJEŠENJA 23
5.1. Testiranje postojećeg okruženja na računalu „Institut RT-RK“ 23
5.2. Testiranje rada *SQLite* baze podataka na osobnom računalu 23
5.3. Testiranje rada *SQLite* baze podataka na računalu „Institut RT-RK“ 26
5.4. Testiranje rada s procesima 27
6. ZAKLJUČAK 29
LITERATURA 30
SAŽETAK 32
ABSTRACT 33
ŽIVOTOPIS 34

1. UVOD

Posljednje desetljeće obilježeno je iznimnim napretkom i razvojem autonomne vožnje [1, 2]. Takav tehnološki napredak rezultirao je pojavom nove prometne paradigme na svim postojećim razinama, od iskustava korisnika do promjena na razini državne regulacije prometnog sustava [3]. Većina promjena koje autonomna vozila donose postojećem prometnom sustavu su povoljne i napredne. Neke od pozitivnih promjena su smanjenje ljudskih pogrešaka i gužvi u prometu, veća ugodnost i produktivnost vožnje, veća mobilnost korisnika i proizvoda, bolja mogućnost dijeljenog prijevoza te očuvanje okoliša [4, 5]. Smanjenje broja prometnih nesreća, ubrzanje vremena putovanja te povećanje efikasnosti trošenja goriva i parkiranja dovest će do značajnog povećanja broja autonomnih vozila, kvalitete vožnje te uštede financijskih sredstava [4]. Procjenjuje se da bi godišnja ušteda po jednom autonomnom vozilu iznosila oko 2000 dolara godišnje, no, može dosegnuti i 4000 dolara, ako se uračunaju troškovi mogućih nesreća u prometu [2]. Razvijanje autonomnih vozila otvara i značajne prilike za otvaranje novih radnih mjesta te eksponencijalni rast i razvoj automobilske industrije [4].

Unatoč navedenim prednostima autonomnih vozila, i dalje postoje prepreke za njihovu implementaciju i probijanje na masovno tržište [6]. Početni troškovi razvoja i uvođenja autonomnih vozila, kao i prilagodbe postojeće prometne infrastrukture vjerojatno će biti veliki [5]. Također, potrebno je uspostaviti i ujednačiti standarde licenciranja i testiranja [2]. Podaci o pouzdanosti autonomnih vozila nisu točno definirani, a prisutna su i pitanja o sigurnosti, kao i o nedostatku privatnosti tijekom osobnih putovanja, što otvara pitanje prihvaćanja autonomnih vozila od strane korisnika. Prema podacima Europske komisije (2018), početni podaci pokazuju da je 58% anketiranih Europljana spremno prihvatiti autonomna vozila, ali samo ako su zadovoljeni najviši standardi sigurnosti i zaštite [4]. U obzir se moraju uzeti i etička pitanja povezana s preuzimanjem odgovornosti za vožnju i moguće prometne nesreće [6]. Potrebno je također razmotriti utjecaj i interakcije razvoja autonomnih vozila s ostalim komponentama prometa, infrastrukturnim planiranjem te drugim industrijskim, ekonomskim i tehnološkim sektorima [4,5].

S obzirom na to da se u budućnosti očekuje daljnji te ubrzani napredak tehnologije autonomnih vozila, potrebno je uložiti dodatni trud u pružanje odgovora na navedena pitanja te razriješiti postojeće dileme [2, 3, 5]. Kako bi ti ciljevi bili postignuti, potrebno je provesti detaljno planiranje, proširiti istraživanja u tom području te osnovati okvir za licenciranje autonomnih vozila na državnoj razini, što uključuje određivanje prikladnih standarda za pouzdanost, sigurnost i privatnost podataka korisnika [2, 5, 6].

Sve većem razvoju sigurnosti i ugodnosti u autonomnim vozilima doprinose sofisticirani sustavi, kao što su ADAS sustavi (engl. *Advanced driver – assistance systems*), koji se u današnje vrijeme u sve većoj mjeri implementiraju u vozila. Zadatak ADAS sustava je osloboditi vozača od rutinskih zadataka te ga upozoriti na potencijalne probleme i opasnosti u vožnji [7]. Takvi sustavi izrazito su složeni, tako da je prije njihove upotrebe potrebno provesti detaljno testiranje njihove programske podrške. Testiranje se sastoji od kreiranja velikog broja testova koji imaju mogućnost provjere određenog ADAS sustava. Kako bi se smanjili ljudski resursi koji su potrebni u testiranju, potrebno je imati na raspolaganju okruženje koje će omogućiti lakšu manipulaciju testovima namijenjenih za testiranje ADAS sustava te samo izvođenje takvih testova.

U okviru ovog diplomskog rada potrebno je osmisliti i implementirati testno okruženje u *Python* programskom jeziku koje će omogućiti lakše pronalaženje i pokretanje pojedinih testova za testiranje ADAS sustava. Okruženje treba uključiti sve testove pisane u *Python* programskom jeziku koji su pohranjeni na određenoj putanji na računalu. Okruženje također treba optimizirati redoslijed i raspodjelu izvršavanja testova kako bi se postigla maksimalna vremenska efikasnost.

Uvidom u postojeće rješenje koje je implementirano u *Python* programskom jeziku, zabilježene su mane kao što su spremanje datoteka (testova) u jedan veliki rječnik te korištenje istog tog rječnika kroz cijeli sustav, što se može opisati kao nepraktično i nepregledno. Također postojeće rješenje ima mogućnost zapisivanja podataka u RAM memoriju, što dovodi do problema gubitka podataka u slučaju nepravilnog gašenja sustava. Ideja je novim rješenjem povećati robusnost te sigurnost cijelog sustava korištenjem baze podataka. Osim spremanja testova u jedan veliki rječnik, mana postojećeg rješenja je i pokretanje testova na nitima (engl. *thread*). Pokretanje velikog broja testova na nitima dovodi do zagušenja u programu. Kako bi se izbjeglo zagušenje ideja je prebaciti rad sa nitima na rad sa procesima kako bi se više testova odvijalo na više različitih procesorskih jezgri.

Ovaj diplomski rad sastoji se od šest poglavlja. Drugo poglavlje bavi se opisom korištenih alata i tehnologija u izradi testnog okruženja te daje teorijsku podlogu diplomskog rada. Treće poglavlje opisuje postojeće stanje te probleme koji su uočeni na postojećem testnom okruženju. Četvrto poglavlje obuhvaća izradu i opis vlastitog rješenja, odnosno, poboljšanja koja su napravljena u odnosu na postojeće rješenje. U petom poglavlju opisano je testiranje izrađenog vlastitog rješenja te su postignuti rezultati uspoređeni s postojećim okruženjem tablično i grafički. U zadnjem poglavlju dan je zaključak na cjelokupni diplomski rad.

2. TESTIRANJE ADAS SUSTAVA I PROGRAMSKI ALATI

U ovom poglavlju objašnjene su korištene tehnologije kao što su *Python* programski jezik, *SQLite* baza podataka, višeprocesni i višenitni način rada te što predstavljaju ADAS sustavi i testiranje njihove programske podrške. Ove tehnologije će biti objašnjene zbog korištenja u postojećem okruženju te zato što predstavljaju osnovu za izradu novog rješenja zasnovanog na postojećem okruženju.

2.1. ADAS sustavi

ADAS (engl. *Advanced Driver Assistance Systems*) je termin koji se koristi za opisivanje rastućeg broja aktivnih i pasivnih sigurnosnih funkcija u vozilima. Cilj takvih sustava je poboljšanje sigurnosti vozača, putnika i pješaka smanjivanjem ukupnog broja prometnih nesreća, kao i ozbiljnosti posljedica nesreća motornih vozila. ADAS predstavlja napredne sustave za nadzor i pomoć vozaču pri svakodnevnoj vožnji. ADAS sustavi upozoravaju vozača na potencijalne opasnosti, reagiraju kako bi se vozaču omogućilo zadržavanje kontrole nad vozilom u funkciji izbjegavanja nesreće te, ako je potrebno, odnosno, ako se nesreća ne može izbjeći, smanjuju njezine razmjere i posljedice [8]. Na taj način ADAS sustavi umanjuju utjecaj ljudskih pogrešaka u vožnji. Prema američkom nacionalnom istraživanju uzroka nesreća motornih vozila, oko 94% nesreća uzrokovali su vozači [9] čineći pogreške prepoznavanja, donošenja odluka, izvedbe te pogreške koje nisu povezane s izvedbom (primjerice, spavanje tijekom vožnje). S obzirom na sve navedeno, može se reći da je ADAS osnova nove generacije automobilskih elektroničkih sigurnosnih sustava, najvažnija tehnologija autonomnih vozila te jedna od najbrže rastućih i nužnih tehnologija u području sigurnosti prometa [10,11].

ADAS je kombinacija elektroničkih sustava čija je funkcija automatiziranje i poboljšanje svih ostalih sustava unutar vozila kako bi se postigla što ugodnija, ali i što sigurnija vožnja. Neke od funkcija koje ADAS sustavi uključuju su: prepoznavanje pješaka i znakova, pomoć pri kontroli brzine te pomoć pri zadržavanju unutar prometne trake.

Prema Udruženju inženjera automobilske industrije SAE [12], postoji 5 razina autonomije vožnje:

- razina 0 – sustav samo upozorava vozača, dok vozač u potpunosti kontrolira sve sustave (brzinu, razmak i smjer vozila),
- razina 1 – vozač kontrolira brzinu i razmak ili smjer vozila uz pomoć sustava za pomoć vozaču u vožnji,
- razina 2 – vozač mora stalno nadzirati sustav, koji u potpunosti preuzima kontrolu nad vozilom u određenim situacijama,

- razina 3 – vozač ne treba pratiti vožnju (autonomna vožnja u odgovarajućim uvjetima npr. autocesta s označenim trakama), ali još uvijek mora biti spreman preuzeti kontrolu u predefiniranom vremenskom intervalu, ako je to potrebno,
- razina 4 – vozač ne mora nadgledati sustav, sustav preuzima cijelu vožnju, ali nije funkcionalan u svim mogućim situacijama, i
- razina 5 – sustav ima potpunu autonomiju te se vožnja odvija bez potrebe vozača, vozilo ne treba imati volan i pedale.

U ovome trenutku svi proizvođači na europskom tržištu u serijskoj proizvodnji su na razini 1 ili 2 autonomije, dok je tek ove godine tvrtka Audi predstavila prvu certificiranu razinu 3. U razinu 4 spadaju neki testni primjerci vozila, kao što su Google i Audi. Razina 5 trenutno je još u razvoju [4, 12].

2.2. Testiranje programske podrške

Testiranje programske podrške predstavlja proces traženja grešaka. U testiranje spadaju sve aktivnosti koje su potrebne pri određivanju sposobnosti određenog programa da izvrši određeni zadatak. Testiranje programske podrške znatno je složenije od testiranja sklopovske podrške gdje se na temelju ulaznih vrijednosti generiraju pripadajuće izlazne vrijednosti. Testirati se mogu pojedini dijelovi ili gotova programska rješenja. Cilj testiranja nije samo provjeriti radi li program ispravno već provjeriti sigurnost, pouzdanost ili procijeniti performanse gotovog rješenja. Procjenjuje se da se oko 50% vremena prilikom razvoja softvera potroši na testiranje. Najčešći razlozi testiranja su [13]:

- poboljšanje kvalitete,
- provjeravanje i potvrda ispravnosti,
- procjena pouzdanosti.

Sve većim razvojem ADAS funkcija potrebno je razmotriti i nove metode testiranja. Buduće ADAS funkcije moći će u potpunosti kontrolirati kretanje vozila te se mora razmotriti interakcija između postojećih i budućih ADAS funkcija. Čak i ako pojedine ADAS funkcije rade kako je predviđeno, pri opremi vozila s više funkcija može doći do neočekivanog ponašanja. Jedinstvena strategija ispitivanja ne može se primijeniti na sve ADAS funkcije pa se ovisno o aspektima sigurnosti i složenosti funkcije mogu koristiti različite metode testiranja. Često prilikom testiranja ADAS funkcija postoji mogućnost korištenja simulacijskog softvera, gdje se podaci prikupljeni sa senzora mogu simulirati u stvarnom vremenu.

Kod ADAS sustava moguće je testirati na potpunoj razini vozila i na funkcionalnoj razini. Razlika je u tome što se na funkcionalnoj razini testira i provjerava samo pojedina ADAS funkcija, dok se na potpunoj razini vozila, provjerava interakcija između ADAS funkcije i ostatka vozila.

Glavna svrha na funkcionalnoj razini je ispitivanje određenih sustava npr. ADAS funkcije i njezine unutarnje komponente sustava. To se postiže opremom vozila potrebnim senzorima i algoritmima. Prvi korak je provjeriti odgovaraju li podaci sa senzora onima iz stvarnosti. To se izvodi snimanjem ceste tijekom izvođenja testova. Potom vozač može usporediti snimke s podacima senzora kako bi provjerio odgovaraju li senzorski podaci okolini. Nadalje, neobrađeni senzorski podaci s određenih vrsta senzora mogu se snimiti i spremiti. To omogućuje provođenje regresijskih testova novih algoritama i softvera na već spremljenim podacima kako bi se potvrdilo da nova funkcionalnost funkcionira onako kako se i očekivalo. Na funkcionalnoj razini senzori se često ispituju odvojeno kako bi se ispitala radna svojstva, nakon čega se različiti senzori spajaju te se provjerava kompletna ADAS funkcija.

Testiranje na potpunoj razini prilikom primjene novih ADAS funkcija u vozilima moguće je provoditi metodom crne kutije (engl. *black – box testing*) ili metodom sive kutije (engl. *grey – box testing*). U metodi testiranjem crne kutije tester, odnosno, ispitivač nema znanje o unutarnjim sustavima, a mjerenjem ulaza i izlaza testiraju se samo osnovni aspekti sustava. Kod metode testiranjem sive kutije ispitivač ima ograničeno znanje o internom radu, ali može mjeriti unutarnje signale na različitim mjestima u sustavu. Na primjer, provjerom ima li funkcija ispravan izlaz i radi li kako je očekivano, tada je dovoljno testirati metodom crne kutije. Međutim, ako su signali unutar ADAS funkcije od interesa, npr. podaci sa senzora, potrebno je provesti testiranje metodom sive kutije [14].

2.3. *Python* programski jezik

Danas u svijetu inženjeri sve više koriste programski jezik *Python*. Zbog svoje jednostavnosti pri učenju, učinkovitosti i izvodivosti na različitim operacijskim sustavima *Python* postaje sve popularniji među programskim jezicima. *Python* predstavlja interpretirani i objektno orijentirani programski jezik visoke razine koji je stvorio Guido van Rossum 1990. godine. Danas postoje dvije verzije *Pythona*, a to su *Python* verzija 2 (verzija koja se koristi u diplomskom radu) i *Python* verzija 3. U *Pythonu* su dozvoljeni razni stilovi programiranja, kao što su: objektno orijentirano, strukturno, funkcijsko te aspektno orijentirano programiranje. Njegova velika prednost je rad na različitim operacijskim sustavima kao što su: Windows, Mac, Linux, itd. *Python*

se može koristiti u različitim domenama kao što su [15]: internet programiranje, razvoj softvera, matematika, edukacija itd.

Python često u svojim naredbama koristi engleske ključne riječi te, za razliku od ostalih programskih jezika, ne koristi vitičaste zagrade za razgraničavanje blokova naredbi. Umjesto vitičastih zagrada upotrebljava uvlake razmaka. Isto se tako točka zarez ne koristi te nije obavezna nakon završetka naredbe.

Python je odabran za realizaciju poboljšanja postojećeg rješenja zbog mogućnosti izvođenja na različitim operacijskim sustavima te zbog toga što je postojeće testno okruženje već implementirano u *Python* programskom jeziku. Na taj način će biti jednostavnija integracija vlastitog rješenja u postojeće testno okruženje. Primjer jednostavne *Python* funkcije koja na osnovu unesenog broja provjerava je li broj pozitivan, negativan ili nula prikazan je na slici 2.1.

```
def check_number():
    number = float(input("Enter a number: "))
    if number > 0:
        print "Positive number"
    elif number == 0:
        print "Zero"
    else
        print "Negative number"
```

Slika 2.1. Primjer jednostavne Python funkcije.

2.4. *SQLite* baza podataka u *Pythonu*

SQLite predstavlja C biblioteku koja pruža bazu podataka pohranjenu u jednu datoteku na disku. U *Pythonu* je dostupan modul *sqlite3* kojeg je napisao Gerhard Häring. *SQLite* baza podataka koristi se u mnogim aplikacijama kao interna pohrana podataka. Kako bi se koristio *sqlite3* modul potrebno je na početku kreirati objekt za povezivanje s bazom te pokazivač (engl. *cursor*) koji se koristi za izvršavanje SQL naredbi. Neke od naredbi koje sadržava *sqlite3* modul su [16]:

- *sqlite3.connect()* – služi za povezivanje sa *SQLite* bazom podataka
- *cursor.execute()* – izvršavanje određenih naredbi prema bazi
- *cursor.fetchall()* – dohvaćanje redaka iz tablice po nekom parametru
- *connection.close()* – zatvaranje povezivanja sa bazom

SQLite je danas najkorištenija baza podataka. Takva baza podataka je brza, samostalna te ima visoku pouzdanost. Osim toga *SQLite* baza podataka je besplatna te se može lako koristiti i stvarati te je zbog toga korištena u diplomskom radu. Neke prednosti *SQLite* baze podataka su:

- za pokretanje nije potreban dodatni program ili komponente (*SQLite* ne zahtjeva pokretanje zasebnog računalnog procesa),
- dijeljenje baze podataka je vrlo jednostavno (potrebno je samo kopirati jednu datoteku),
- *SQLite* se temelji na standardnom SQL jeziku.

2.5. Paralelna obrada, višeprocesni i višenitni način rada

Paralelna obrada (engl. *parallel processing*) predstavlja istovremeno izvršavanje više instrukcija. Paralelna obrada koristi dva ili više procesora (procesorskih jezgri) u kombinaciji kako bi se riješio zadani problem. Kod paralelne obrade program se dijeli na manje dijelove koji se obrađuju istovremeno. Primarni cilj paralelne obrade je povećati raspoloživu računalnu snagu za bržu obradu podataka i rješavanje zadataka [17].

Višenitni način rada (engl. *mltithreading*) predstavlja rad u kojemu više niti radi u istom procesu i istom memorijskom prostoru. Svaka nit obavlja svoj zadatak, ima svoj vlastiti kod, vlastitu memoriju stogova, pokazivač instrukcija i dijeljenu memoriju. Niti koje pripadaju istom procesu moraju dijeliti komponente tog procesa kao što su kodovi, podaci i resursi sustava. Ako kod niti dođe do curenja memorije može se dogoditi oštećenje ostalih niti i nadređenih procesa. Za razliku od višenitnog načina rada, višeprocesni način rada (engl. *multiprocessing*) koristi odvojeni memorijski prostor i više procesorskih jezgri. Najveća prednost u odnosu na višenitni način rada je to što se zadaci mogu odvijati na više različitih procesorskih jezgri čime se postiže brže rješavanje zadataka [18].

Višprocesni način rada zapravo predstavlja paket koji podržava pokretanje procesa koristeći API sličan onome u višenitnom načinu rada. Kod višenitnog načina rada, povećavanjem broja niti povećava se i linearno vrijeme izvršavanja. Također u *Pytho*n programskom jeziku dolazi do problema u slučaju ako neki zadatak treba CPU na duže vrijeme. Takav problem se naziva *Global Interpreter Lock* (GIL). GIL je mehanizam koji koristi interpreter za sinkronizaciju izvršavanja niti kako bi se osiguralo da se samo jedna nit može istovremeno izvršiti. Način na koji se zaobilazi problem GIL-a je korištenje većeg broja procesa umjesto većeg broja niti. Primjer toga je objekt *Pool* koji nudi mogućnost paralelizacije izvršavanja funkcija, odnosno, paralelizma podataka u procesima [19].

3. POSTOJEĆE OKRUŽENJE ZA TESTIRANJE ADAS PROGRAMSKIH RJEŠENJA

Postojeće rješenje za testiranje ADAS programskih rješenja predstavlja alat koji se koristi za pronalazak, pripremu te izvođenje automatiziranih testnih slučajeva. Testni slučaj je specifikacija ulaza, uvjeta izvršenja, postupka testiranja i očekivanih rezultata koji definiraju jedan test koji treba izvršiti kako bi se postigao određeni cilj testiranja softvera, kao što je provjera sukladnosti s određenim zahtjevima. Postojećim okruženjem za testiranje ADAS programskih rješenja obavlja se testiranje na funkcionalnoj razini, što je već opisano u poglavlju 2.2., odnosno, testiranje se svodi na ispitivanje ispravnosti rada pojedinih ADAS funkcija.

Alat kao cjelina sastoji se od nekoliko *Python* skripti koje su potrebne za ispravan rad. Glavna skripta je *TestExecutor.py* i ta skripta pokreće alat zvan *Test Executor*. Neke skripte koje *Test Executor* koristi su: *TE_LoadConfig*, *TE_FileHandler*, *TE_Optimizator*, *TE_Obj*, *TE_ErrorHandler* i *TE_DeviceHandler*. Najbitnija datoteka za ovaj diplomski rad je *TE_FileHandler* u kojoj se trenutno izvršava spremanje svih testova u jedan veliki rječnik.

Prije pokretanja *Test Executora* potrebno je ispravno podesiti datoteku *config.ini*. Slika 3.1. prikazuje datoteku config.ini u kojoj se nalaze sve važne informacije o projektu koje su potrebne za pokretanje *Test Executora*. Datoteka je također važna za postavljanje putanje do direktorija gdje se nalaze testovi.

```
# List of global variables available for user to edit
[Globals]
#Project = zFAS_series, MotionWise, iECU_EP21H....
project='MotionWise'
#Test System
VTSystem=True
#USB Switch device ID
devID=""
#HTML report file name
html_reprot_filename=""
#Console printout width
screenEdge=110
#Path to tests
pathToTests=r"E:\TestExe\test_executor\Input"
```

Slika 3.1. Datoteka config.ini.

Testne slučajeve moguće je izraditi ručno, ali također ih je moguće kreirati direktno iz *Test Executora*. Za svaki testni slučaj nužno je imati pravilno ispunjeno zaglavlje testa („*tc_header*“) u njemu se nalaze sve potrebne informacije o pojedinom testu kako bi se on mogao izvršiti.

```
####################
# TC Header
####################
tc_header = {
    'ptc_id': [2643784],
    'service': "LifeCycle",
    'build': 'System',
    'data_set': "DB_ALL",
    'host': ['PH00'],
    'document': 'ITS',
    'tag':'TBD'
}
```

Slika 3.2. Zaglavlje testa.

Svaka pojedina datoteka (test) sastoji se od funkcija potrebnih za ispravan rad testa te zaglavlja testa u obliku rječnika (engl. *dictionary*) nazvanog „*tc_header*“ koji sadrži podatke o testu. Slika 3.2. prikazuje izgled zaglavlja testa koje je spremljeno u jedan rječnik naziva „*tc_header*“. Informacije su specifične za svaki test. Duljina zaglavlja te nazivi i vrijednosti pojedinih ključnih riječi specifični su za svaki test.

Uvidom u postojeće rješenje zabilježeni su nedostaci kao što su spremanje testova u jedan veliki rječnik te korištenje istog tog rječnika kroz cijelo okruženje, što se može opisati kao nepraktično, nesigurno i nepregledno. Osim spremanja testova u jedan veliki rječnik, nedostatak je bio i pokretanje testova na nitima (engl. *thread*). Pokretanje velikog broja testova u višenitnom načinu rada dovodi do zagušenja u programu. Kako je već spomenuto u poglavlju 2.5., rad s nitima se pokreće u istoj jedinstvenoj memoriji, dok se procesi izvode u odvojenim memorijama. Izvođenje u odvojenim memorijama otežava dijeljenje informacija s procesima i instancama objekata te je potrebno istražiti kako to na što lakši način odraditi. Problem kod niti nastaje kada više niti želi istovremeno pisati u istu memoriju. Kada se pokrene postojeće testno okruženje, koje radi u višenitnom načinu, vidljivo je da je samo jedna procesorska jezgra opterećena.

4. POBOLJŠANJE OKRUŽENJA ZA TESTIRANJE ADAS PROGRAMSKIH RJEŠENJA

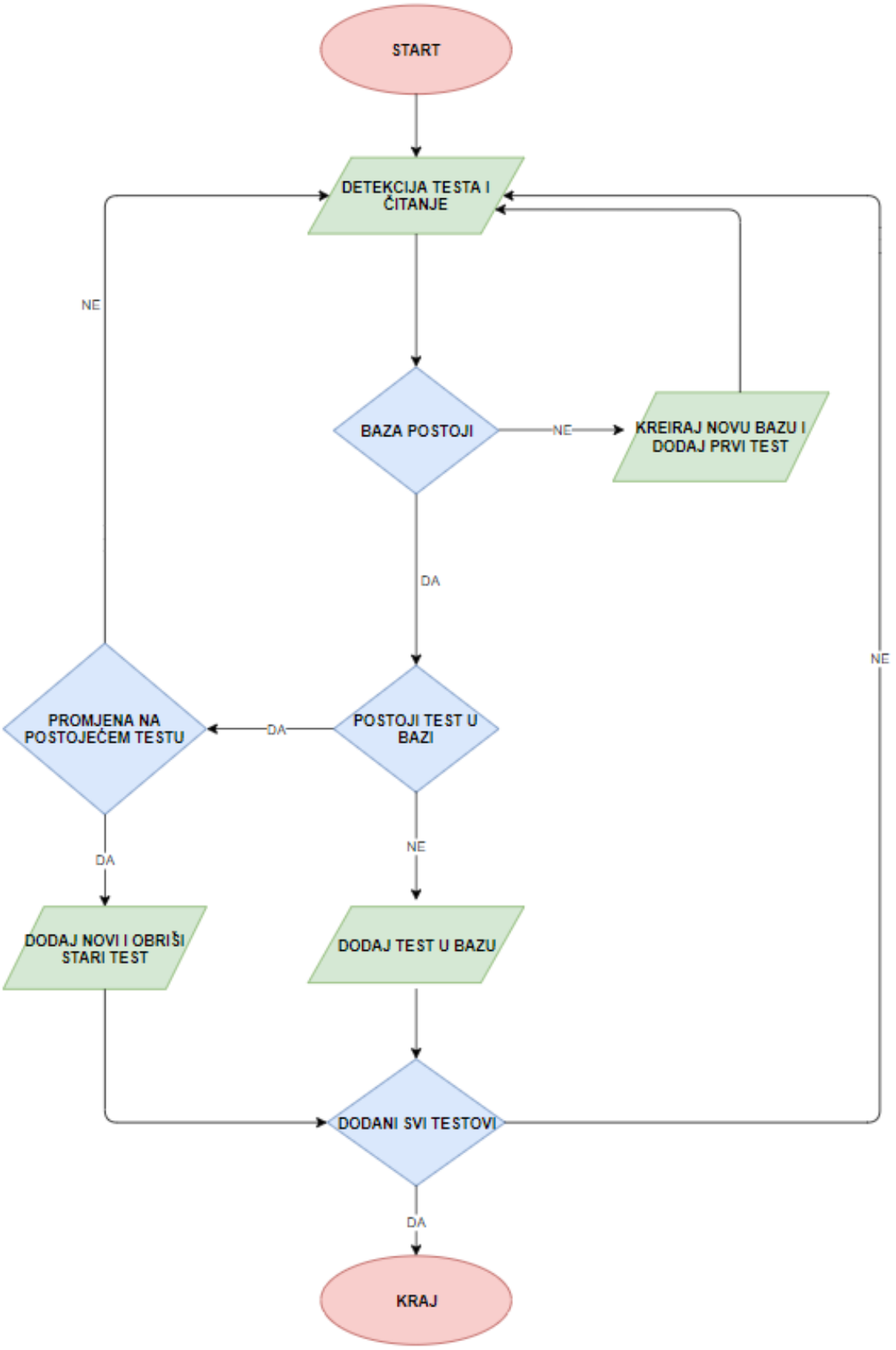
Trenutno okruženje sastoji se od prolaska kroz sve testove, čitanja rječnika u zaglavlju *(„tc_header“)* iz testa te spremanja tih testova u jedan novi veliki rječnik. Cijelo okruženje radi s takvim rječnikom. Prilikom testiranja postojećeg okruženja uočeno je da je takav način dosta nepraktičan te nepregledan korisniku. Kako bi se ostvarilo poboljšanje, potrebno je koristiti bazu podataka koja predstavlja znatno robusnije, stabilnije, sigurnije i preglednije rješenje za pohranu podataka od postojećeg. Poboljšanja postojećeg okruženja za testiranje ADAS programskih rješenja mogu se predstaviti sljedećim zahtjevima:

- pronalazak svih datoteka (testova) u mapama i podmapama,
- ubrzanje izvođenja programa na način da se čita samo dio datoteke gdje se nalaze podaci o testu,
- projektiranje i izgradnja odgovarajuće baze podataka za pohranu podataka o testovima,
- spremanje podataka o pojedinom testu (zapisano u „*tc_headeru*“) u bazu, detekcija promjene na pojedinom testu te izmjena u bazi i dohvaćanje podataka o pojedinom testu iz baze te rukovanje istim (engl. *CRUD – create, read, update and delete*).
- ubrzanje rada programa prelaskom na rad s procesima, umjesto dosadašnjeg rada s nitima.

4.1. Ideja i model rješenja

Za početak bilo je potrebno osmisliti na koji način najbolje zamijeniti trenutno spremanje testova u rječnik. Osim toga bilo je potrebno na što brži način pronaći gdje se u testu nalazi zaglavlje kako se ne bi čitala cijela datoteka, već samo onaj dio gdje je zaglavlje. Zbog svoje robusnosti, *SQLite* baza podataka se pokazala kao idealno rješenje za spremanje testova jer i nakon gašenja aplikacije podaci o svim testovima ostaju spremljeni. Podaci o testovima spremaju se u dvije tablice. U jednoj tablici nalaze se opći podaci o testu, dok se u drugoj nalaze vrijednosti iz zaglavlja „*tc_hedaer*“. Svaki ključ u rječniku predstavljao bi stupac u tablici, dok bi pripadna vrijednost ključa predstavljala redak u tablici u bazi podataka. Odlučeno je da bi bilo dobro napraviti i mogućnost prepoznavanja promjena na svakom testu te da ta promjena bude osvježena u bazi podataka.

Prije početka implementacije testnog okruženja bilo je potrebno napraviti dijagram toka kako bi se olakšao sami postupak programiranja. Slika 4.1. prikazuje dijagrama toka predloženog rješenja. Program otvara jedan po jedan test te u njemu traži gdje se nalazi zaglavlje.



Slika 4.1. Dijagram toka dodavanja testova u bazu podataka.

Kada je pronađeno zaglavlje poziva se funkcija za stvaranje *SQLite* baze podataka ako ona već ne postoji. Stoga će se kod prvog pokretanja programa stvoriti nova *SQLite* baza podataka te dodati prvi test koji se pročita. U suprotnom je potrebno provjeravati postoji li određeni test u bazi ili je došlo do promjene na već postojećem testu u bazi. Kada program završi s dodavanjem testova, korisnik može pretraživati bazu podataka. To znači da korisnik može dohvatiti testove po nekom određenom parametru iz baze podataka.

Drugi dio diplomskog rada bio je prebacivanje objekta između različitih procesa, odnosno, jednostavnog rječnika ili složenijeg objekta u obliku klase koja sadrži više rječnika i lista. Trenutno rješenje je obuhvaćalo rad s nitima pa je ideja bila da se istraži prebacivanje objekta između različitih procesa te da su promjene na objektu vidljive u svakom procesu. Objekt je trebalo kreirati u obliku jednostavnog rječnika na početku, a kasnije na složeniji način, odnosno, u obliku klase koja će sadržavati više rječnika i lista.

Slika 4.2. prikazuju kako se zaglavlje testa „*tc_header*“ sprema u tablicu *Header* baze podataka, dok slika 4.3. prikazuje spremanje općih podataka o testu u tablicu *Tests*.

Database Structure | Browse Data | Edit Pragmas | Execute SQL

Table: Header

New Record | Delete Re

	File_Name	path	Last_Modified	host	domain	document	build	ptc_id
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	EH_2861841.py	E:\TestExe\te...	18-07-2019 1...	['PH']	Error Handler	ITS	PFF	[2861841]
2	EH_2861844.py	E:\TestExe\te...	18-07-2019 1...	['PH']	Error Handler	ITS	PFF	[2861844]
3	EH_2861847.py	E:\TestExe\te...	18-07-2019 1...	['PH']	Error Handler	ITS	PFF	[2861847]
4	EH_2861849.py	E:\TestExe\te...	18-07-2019 1...	['PH']	Error Handler	ITS	PFF	[2861849]
5	EH_2875674.py	E:\TestExe\te...	18-07-2019 1...	SH	NULL	ITS	System	[2875674]
6	EH_2875676.py	E:\TestExe\te...	18-07-2019 1...	PH	NULL	ITS	System	[2875676]
7	EH_2893685.py	E:\TestExe\te...	12-07-2019 1...	PH	NULL	ITS	System	[2893685]
8	EH_2893687.py	E:\TestExe\te...	12-07-2019 1...	SH	NULL	ITS	System	[2893687]
9	EH_2893689.py	E:\TestExe\te...	12-07-2019 1...	SH	NULL	ITS	System	[2893689]
10	EH_2861841.py	E:\TestExe\te...	18-07-2019 1...	['PH']	Error Handler	ITS	PFF	[2861841]

Slika 4.2. Struktura baze podataka – tablica Header.

Database Structure | Browse Data | Edit Pragmas | Execute SQL

Table: Tests

New Record

	File_Name	path	size	Last_Modified	Created_Time
	Filter	Filter	Filter	Filter	Filter
1	EH_2861841.py	E:\TestExe\test_executor\Input\...	2993	18-07-2019 13:48:49	Sat Jul 27 22:02:40 2019
2	EH_2861844.py	E:\TestExe\test_executor\Input\...	3037	18-07-2019 13:26:30	Sat Jul 27 22:02:40 2019
3	EH_2861847.py	E:\TestExe\test_executor\Input\...	3298	18-07-2019 13:26:37	Sat Jul 27 22:02:40 2019
4	EH_2861849.py	E:\TestExe\test_executor\Input\...	3034	18-07-2019 13:27:07	Sat Jul 27 22:02:40 2019
5	EH_2875674.py	E:\TestExe\test_executor\Input\...	2478	18-07-2019 13:27:16	Sat Jul 27 22:02:40 2019
6	EH_2875676.py	E:\TestExe\test_executor\Input\...	2491	18-07-2019 13:27:21	Sat Jul 27 22:02:40 2019
7	EH_2893685.py	E:\TestExe\test_executor\Input\...	2463	12-07-2019 13:21:02	Sat Jul 27 22:02:40 2019
8	EH_2893687.py	E:\TestExe\test_executor\Input\...	2487	12-07-2019 13:21:02	Sat Jul 27 22:02:40 2019
9	EH_2893689.py	E:\TestExe\test_executor\Input\...	2487	12-07-2019 13:21:02	Sat Jul 27 22:02:40 2019
10	EH_2861841.py	E:\TestExe\test_executor\Input\...	2993	18-07-2019 13:48:49	Sat Jul 27 22:02:40 2019

Slika 4.3. Struktura baze podataka – tablica Tests.

4.2. Programski dio rješenja

Nakon pojednostavljenja pomoću dijagrama toka potrebno je sve opisane korake realizirati pomoću programskom jezika *Python*. Za početak, programski dio se radio odvojeno od postojećeg okruženja, a ideja je bila da se nakon završetka doda u postojeće okruženje. Integracija s postojećim okruženjem treba biti izvedena tako da se oni dijelovi koda gdje se testovi spremaju u rječnik zamjene sa spremanjem testova u *SQLite* bazu podataka. Također je bilo potrebno dohvaćanje iz rječnika zamijeniti sa dohvaćanjem iz baze podataka. Zbog preglednosti, lakše integracije s postojećim okruženjem, boljih performansi i lakše mogućnosti dodatnog proširenja, novo rješenje je podijeljeno u funkcije od kojih svaka ima specifičnu ulogu. Rješenje se sastoji od pet funkcija koje su opisane u nastavku.

- *detect_all_files*(*te_obj*) – funkcija za prolazak kroz sve datoteke u mapama i podmapama. Funkcija kao parametar prima *te_obj* koji predstavlja putanju do mape testova.
- *read_all_files(test_file)* – funkcija za čitanje svake datoteke (testa) te pronalazak dijela gdje se nalazi zaglavlje i spremanje istog kako bi bilo spremno za dodavanje u *SQLite* bazu podataka. Funkcija kao parametar prima test koji je potrebno pročitati.
- *start_database()* – funkcija za kreiranje nove baze podataka te za provjeru postojanja baze podataka na disku.
- *refresh_Database(valueOfColumnList, nameOfColumnList, valueInTests, nameInTests)* – funkcija za dodavanje prvog testa u bazu, novih testova i provjeru promjena na testu. Funkcija kao parametre prima četiri liste. Lista *nameOfColumnList* sadrži nazive stupaca koji će se zapisati u tablici *Header,* dok lista *nameInTests* sadrži nazive stupaca za tablicu *Tests.* Vrijednosti stupaca za tablicu *Header* zapisani su u listi *valueOfColumnList* dok su za tablicu *Tests* vrijednosti stupaca zapisani u listi *valueInTests.*
- *get_item_db(**kwargs)* – funkcija koja omogućuje korisniku dobivanje odgovarajućih testova iz baze podataka za bilo koji broj parametara. Parametar treba biti uređeni par stupac i redak, a funkcija vraća vrijednost za određeni stupac i redak.

Svakoj od gore navedenih funkcija dano je upravo takvo ime zbog lakšeg shvaćanja što pojedina funkcija radi. U daljnjem tekstu svaka od tih funkcija bit će detaljnije razrađena, odnosno, bit će objašnjeno kako je koji dio programa realiziran i zašto je realiziran na takav način.

Dio koji se bavi prebacivanjem objekta između različitih procesa, odnosno, razmjenom podataka između dva procesa, realiziran je na početku kao jednostavan rječnik, no zbog složenosti postojećeg rješenja takav način bio je previše pojednostavljen. Kasnije je takav način zamijenjen

složenijim kako bi bio spreman za lakšu integraciju. Složeniji način sastoji se od jedne klase koja je sadržavala više lista i rječnika. Tako složeni objekt potrebno je prebacivati između različitih procesa.

4.2.1. Detekcija promjena i dodavanje testova u bazu podataka

Prva funkcija koja se poziva na početku programa je *detect_all_files*. Prvi zadatak ove funkcije je dohvaćanje putanje glavne mape u kojoj se nalazi svi testovi. Svaki test predstavljen je jednom *Python* datotekom. Nakon toga funkcija prolazi jednu po jednu datoteku u mapi, uz uvjet da je datoteka (test) pisan u *Python* programskom jeziku, te poziva funkciju *read_all_files* koja je zadužena za daljnji dio (slika 4.4.). Funkcija kao povratnu vrijednost vraća sve testove koji su dostupni u obliku liste. Funkcija također broji koliko testova je pronađeno.

```
def detect_all_files(te_obj):
    file_list = list()
    te_obj.log.progress("Mapping Python files...\n")
    if te_obj.globals.get('pathToTests') == '':
        for path, subdirs, files in os.walk("TestCase"):
            for name in [x for x in files if x.endswith('.py')]:
                file_list.append((path, name))
                test_file = (os.path.join(path, name))
                read_all_files(test_file)

    else:
        for path, subdirs, files in os.walk(te_obj.globals.get('pathToTests')):
            for name in [x for x in files if x.endswith('.py')]:
                file_list.append((path, name))
                test_file = (os.path.join(path, name))
                print "opening file", counter_file
                counter_file += 1
                read_all_files(test_file)

    return file_list
```

Slika 4.4. Detekcija dostupnih testova.

Nakon detekcije pojedine datoteke potrebno je tu istu datoteku otvoriti i pronaći „*tc_header*“ unutar nje. Ta funkcionalnost obavlja se u funkciji *read_all_files*. Po pozivanju, funkcija prima datoteku te započinje čitanje. U datoteci (testu) je bitno pronaći gdje se nalazi zaglavlje („*tc_header*“), odnosno, potrebno je osmisliti kako pronaći samo taj dio, a da se ne čita cijela datoteka, kako bi se ubrzao proces čitanja iz datoteke. Ovo je učinjeno na način da je datoteka čitana red po red te su brojane vitičaste zagrade jer rječnik, u ovom slučaju „*tc_header*“, započinje otvorenom, a završava zatvorenom vitičastom zagradom. Cijeli pročitani dio iz datoteke, odnosno, „*tc_header*“ se sprema u jedan niz znakova (engl. *string*). Nakon toga se koristi naredba *exec()* koja ima mogućnost dinamičkog izvršavanja dijela *Python* programa koji je u ovom slučaju niz

znakova. Izvršavanjem naredbe *exec()* niz znakova je pretvoren u rječnik kako bi daljnji rad sa zaglavljem bio olakšan i ubrzan.

Bilo je potrebno kreirati i prazne liste kako bi u njih mogli spremiti nazive stupaca koji se koriste u bazi podataka i pripadne vrijednosti koje su zapisane u „*tc_headeru*“. Kako je niz znakova izvršen kao rječnik pomoću naredbe *exec()*, možemo izvući ključeve (engl. *keys*) i pripadne vrijednosti ključeva (engl. *values*). U jednu listu spremaju se ključevi iz rječnika, dok u drugu pripadne vrijednosti ključeva. U slučaju da pročitani test ne sadrži zaglavlje, program automatski prelazi na novi test, a prazni test ne dodaje se u bazu podataka. Takav test se odbacuje i smatra se nevaljanim. Na slici 4.5. prikazana je funkcija *read_all_files()* koja služi za traženje zaglavlja „*tc_header*“ u testu.

```
def read_all_files(test_file):
    cntUlazZagrada = 0      # counter of "{" (open bracket)
    cntIzlazZagrada = 0     # counter of "}" (closed bracket)
    with open(test_file, 'rb'):
        my_string = ""
        for i in open(test_file).readlines():
            if '}' in i:
                cntIzlazZagrada = cntIzlazZagrada + i.count('}')
                my_string += i
                break

            if cntUlazZagrada != cntIzlazZagrada:
                my_string += i

            if '{' in i:
                cntUlazZagrada = cntUlazZagrada + i.count('{')
                my_string +=i

        nameOfColumnList = list()
        valueOfColumnList = list()
        valueInTests = list()
        nameInTests = list()

        locals()['tc_header'] = None
        print my_string
        exec(my_string)
        if locals()['tc_header'] is None:
            print("##################")
            print("EMPTY file, WITHOUT TC_HEADER")
            print("##################")
            return 0

        columns = (locals()['tc_header'].keys())
        values = (locals()['tc_header'].values())
        for i in columns:
            nameOfColumnList.append(i)
        for j in values:
            valueOfColumnList.append(str(j))
```

Slika 4.5. Čitanje datoteka i spremanje u liste.

Same vrijednosti iz „*tc_header*“ korisniku baze ne daju previše informacija. Kako bi korisnik baze podataka znao o kojem se točno testu radi bilo je potrebno izvući i podatke o testu, kao što su naziv, putanja i veličina testa te podatke o tome kada je zadnji put test izmijenjen i kada je kreiran.

Slika 4.6. prikazuje dodavanje tih podataka na početak liste. Nakon što su popunjene liste, spremne su za slanje u funkciju *refresh_Database,* gdje se odlučuje što će se raditi s listama. Liste *nameOfColumnList, valueOfColumnList* se koriste za popunjavanje tablice *Header,* dok se liste *valueInTests* i *nameInTests* koriste za popunjavanje tablice *Tests*.

```
base = os.path.basename(test_file)
path = os.path.abspath(test_file)
createdTime = time.ctime(os.path.getctime(test_file))
size = os.path.getsize(test_file)
getTime = os.path.getmtime(test_file)
modificationTime = time.strftime('%d-%m-%Y %H:%M:%S', time.localtime(getTime))

valueOfColumnList[0:0] = [base, path, modificationTime]
nameOfColumnList[0:0] = ["File_Name", "path", "Last_Modified"]

valueInTests[0:0] = [base, path, size, modificationTime, createdTime]
nameInTests[0:0]    =    ["File_Name",    "path",    "size",    "Last_Modified",
"Created_Time",]

refresh_Database(valueOfColumnList,      nameOfColumnList,      valueInTests,
nameInTests)

return nameOfColumnList,valueOfColumnList
```

Slika 4.6. Dodavanje pojedinosti o svakoj datoteci u odgovarajuću listu.

Kako bi se koristila *SQLite* baza podataka potrebno je uključiti u *Pythonu* modul pomoću naredbe *import sqlite3.* Novostvorena baza naziva se *Testovi*. Slika 4.7. prikazuje funkciju koja ima ulogu povezivanja na bazu *Testovi* i provjere je li baza kreirana. Baza *Testovi* sastoji se od 2 tablice *Header i Tests*. Kod integracije je došlo do problema povezivanja na bazu jer se objekt *SQLite* baze podataka mogao koristiti samo u jednoj niti. Kako bi se objekt *SQLite* baze podataka mogao koristiti u više niti dodan je uvjet *check_same_thread = False*, nakon postavljanja imena baze podataka.

```
def start_database():
    global conn, c, counter
    database_is_new = not os.path.exists('Testovi.db')
    conn = sqlite3.connect('Testovi.db', check_same_thread=False)
    c = conn.cursor()
    c.execute("PRAGMA synchronous = OFF")
    c.execute("PRAGMA journal_mode = MEMORY")

    # check new or existing DB
    if database_is_new:
        counter = 0
        print("New DATABASE, Counter je: ", counter)
    else:
```

```
        counter = 2
        print("DATABASE Exist, Counter je: ", counter)
    return counter, conn, c
```

Slika 4.7. Stvaranje SQLite baze podataka.

Kada je baza podataka stvorena i test proslijeđen, potrebno je provjeriti postoji li tablica u bazi podataka ili ju je potrebno stvoriti. Nakon toga program odlučuje što će raditi s listama koje su proslijeđene u funkciju. Pri prvom pokretanju baze podataka potrebno je stvoriti dvije nove tablice u bazi podataka u koju će se upisati podaci o prvom testu. Zbog lakše preglednosti koriste se dvije tablice u bazi *Testovi.db* i to *Header* i *Tests*. *Header* tablica služi za prikaz podataka koji su zapisani u zaglavlju testa, odnosno „*tc_headeru*“, te naziv testa, putanju i zadnju promjenu na pojedinom testu. Tablica *Tests* sadrži općenite podatke o svakom testu, poput naziva, vremena kada je stvoren test, koja je njegova putanja, kolika je veličina u bajtovima i kada je zadnji put došlo do promjene. Nakon stvaranja tablica dodaje se prvi test u obje tablice.

Nakon što je kreirana baza *Testovi.db*, stvorene tablice *Header* i *Tests* i dodan prvi pročitani test, aplikacija će na svakom sljedećem testu preći na sljedeći dio (slika 4.8.) jer baza postoji i nije ju potrebno ponovno kreirati i otvoriti. Prilikom dodavanja testova u bazu podataka došlo je do problema kada je više testova stvoreno u točno isto vrijeme. Taj problem je prvotno bio riješen tako što se dodavao test u bazu po imenu testa, no ispostavilo se da to nije dobar način jer je moguće imati više istih naziva testova, ali u različitim podmapama. S obzirom na to, ovaj je slučaj uspješno riješen tako da se testovi dodaju u bazi po njihovim putanjama. Bilo je potrebno stvoriti dvije prazne liste zbog provjere postojanja testa i dodavanja istog u bazu podataka. Lista *list_pathFile* koristi se za spremanje putanja svakog testa, dok se lista *list_LastModified* koristi za spremanje vremena zadnje promjene na nekom od testova. Ukoliko putanja određenog testa ne postoji u bazi podataka potrebno je taj test dodati, ali također je potrebno provjeriti ima li taj test nazive stupaca kao i zadnji dodani test u bazi podataka. Izvršava se usporedba stupaca prošlog testa s trenutnim te, u slučaju da testovi imaju različite stupce, dodaje se razlika stupaca, odnosno, dodaje se stupac (i pripadna vrijednost) koju prošli test nema. U isto vrijeme kada se dodaju vrijednosti u tablicu *Header*, dodaju se i općeniti podaci o testu u tablicu *Tests*.

```
    if counter == 2:
        list_pathFile = []
        for last_path in conn.execute("SELECT path FROM Header"):
            list_pathFile.append(last_path[0])

        list_LastModified = []
        for last in conn.execute("SELECT Last_Modified FROM Header"):
            list_LastModified.append(last[0])
```

```
        if valueOfColumnList[1] not in list_pathFile:
            print("NEMA TESTA u DataBase, DODAJ!!!")
            columns = [i[1] for i in c.execute('PRAGMA table_info(Header)')]

            result = (set(nameOfColumnList) - set(columns))
            print(result)
            # printa i sprema iteme koji nisu duplikati u listama(nema ih u
tablici)
            # set() nema razlike između dvije liste
            if result != set():
                for i in result:
                    c.execute("ALTER TABLE Header ADD COLUMN '%s' " % i)


            sql_insert = 'INSERT INTO Header(' + ','.join(nameOfColumnList) +
') values(' + ','.join(['?'] * len(valueOfColumnList)) + ')'
            c.execute(sql_insert, valueOfColumnList)

            sql_insert_1 = 'INSERT INTO Tests(' + ','.join(nameInTests) + ')
values(' + ','.join(['?'] * len(valueInTests)) + ')'
            c.execute(sql_insert_1, valueInTests)
```

Slika 4.8. Dodavanje svih ostalih testova u bazu podataka.

U slučaju da već imamo putanju testa spremljenu u bazi, moramo provjeriti je li došlo do promjene na tom testu. Slika 4.9. predstavlja listu *list_LastModified* u koju spremamo vremena zadnjih promjena na pojedinom testu. Ako je došlo do promjene, test se briše, a upisuju se nove vrijednosti u obje tablice.

```
        elif valueOfColumnList[2] not in list_LastModified:
            print("NEMA TESTA u DataBase, DODAJ!!!")
            columns = [i[1] for i in c.execute('PRAGMA table_info(Header)')]

            result = (set(nameOfColumnList) - set(columns))
            if result != set():
                for i in result:
                    c.execute("ALTER TABLE Header ADD COLUMN '%s' " % i)

            sql_insert = 'INSERT INTO Header(' + ','.join(nameOfColumnList) +
') values(' + ','.join(['?'] * len(valueOfColumnList)) + ')'
            c.execute(sql_insert, valueOfColumnList)

            sql_insert_1 = 'INSERT INTO Tests(' + ','.join(nameInTests) + ')
values(' + ','.join(['?'] * len(valueInTests)) + ')'
            c.execute(sql_insert_1, valueInTests)


        else:
            pass
            print("POSTOJI TEST u DB, Sve ISTO!!!")

    c.execute("DELETE FROM Tests WHERE rowid NOT IN (SELECT MAX(rowid) FROM
Tests GROUP BY path);")
    c.execute("DELETE FROM Header WHERE rowid NOT IN (SELECT MAX(rowid) FROM
Header GROUP BY path);")
    conn.commit()
```

Slika 4.9. Provjera izmjena na postojećem testu.

4.2.2. Dohvaćanje testova iz baze podataka

Po završetku dodavanja svih testova u bazu podataka korisniku je omogućeno dobivanje informacije o testovima po nekom parametru. Tu zadaću ima funkcija *get_item_db* (slika 4.10.). Ključ i njegova pripadna vrijednost, odnosno ime stupca i vrijednost u retku, smatraju se parametrom koji funkcija prima. Funkcija, u ovisnosti o broju parametara, izvršava upit u bazu podataka. Nakon sastavljenog upita dohvaćaju se svi elementi iz baze koji zadovoljavaju takav uvjet. U liste *list_keys* i *list_keysTests* spremaju se nazivi stupaca iz baze podataka kako bi se kasnije mogla svaka vrijednost dodati odgovarajućem nazivu stupca.

```
def get_item_db(**kwargs):
    counter, conn, c = start_database()
    item_db = dict()
    neasted_dict = dict()
    list_keys = list()
    nova = dict()

    cnt_kwarg = 0
    query = "SELECT * FROM Header"
    if len(kwargs)!= 0:
        query += " WHERE"
    for key, value in kwargs.items():
        cnt_kwarg += 1
        query += " " + key +  " = " + str(value)
        if cnt_kwarg < len(kwargs):
            query += " " + "AND"
        else:
            break

    list_keys = [i[1] for i in c.execute('PRAGMA table_info(Header)')]
    c.execute(query)
    rows = c.fetchall()

    list_keysTests = [i[1] for i in c.execute('PRAGMA table_info(Tests)')]
```

Slika 4.10. Upit u bazu za dohvaćanje testova.

S obzirom na to da je ovakvo rješenje zahtijevalo prilagodbu na postojeće rješenje, odnosno integraciju, bilo je potrebno dohvaćene elemente iz baze spremiti u jedan rječnik (slika 4.11.). Taj rječnik treba se sastojati od više različitih rječnika od kojih svaki predstavlja jedan test. Jedan takav veliki, odnosno, ugniježđeni rječnik u takvom formatu spreman je za integraciju u postojeće rješenje. Kako bi u takvom rječniku znali koji se rječnik odnosi na koji test, bilo je potrebno svaki rječnik nazvati imenom testa.

```
    counter = 0
    for row_value in rows:
        nameOfDict = str(rows[counter][0])
        pathValue = str(rows[counter][1])
        query1 = "SELECT * FROM Tests WHERE path = '" + pathValue + "'"

        c.execute(query1)
```

```
        path = c.fetchall()[0]

        counter += 1 # sluzi za naziv dictionarya u dictionaryu => prvi dict
= test1.py{} drugi = test2.py{}...
        item_db = dict(zip(list_keys,row_value)) # two list in one dict
        nova = map(item_db.pop, ['File_Name','Last_Modified', 'path'])
        neasted_dict[nameOfDict] = dict()
        neasted_dict[nameOfDict]['header'] = item_db

        for index, key in enumerate(list_keysTests):
            neasted_dict[nameOfDict][key] = path[index]

    return neasted_dict
    conn.close()
```

Slika 4.11. Spremanje testova u neasted_dict.

4.2.3. Rad s procesima

Nakon obavljenog zadatka s dodavanjem testova u bazu i dohvaćanjem elemenata iz baze podataka, bilo je potrebno preći na drugi dio zadatka, odnosno, rad s procesima. Postojeće rješenje svodilo se na rad s nitima, dok je zadatak obuhvaćao istraživanje prebacivanja objekta između procesa zbog ubrzanja rada. Prvi način realiziran je na način da je kreiran jednostavan rječnik (x) koji se prebacivao između različitih procesa pomoću *multiprocessing.Manager()* funkcije. *Manager* je služio za stvaranje podataka koji se mogu dijeliti između različitih procesa odnosno obavljao je sinkronizaciju među njima. Funkcija *Manager.dict()* služila je za kreiranje dijeljenog rječnika. Definirane su tri funkcije prikazane na slici 4.12.: *process1*, *process2* i *process3* koje su proširivale početni rječnik.

```
def process2(shared_dictionary):
    print("From p2: Usao u Process2")
    print("From p2:{}".format(shared_dictionary))
    shared_dictionary['y'] = 20
    print("From p2:{}".format(shared_dictionary))

def process1(shared_dictionary):
    print("From p1: Usao u Process1")
    time.sleep(1)
    print("From p1:{}".format(shared_dictionary))

def process3(shared_dictionary):
    print("From p3: Usao u Process3")
    time.sleep(2)
    shared_dictionary['z'] = 50
    print("From p3:{}".format(shared_dictionary))
```

Slika 4.12. Funkcije za proširenje rječnika.

Svaki od procesa radio je s tom određenom funkcijom. Na kraju je uspješno realiziran uvid svakog procesa u događanja unutar ostala dva procesa. Tri procesa su pokrenuta u isto vrijeme te je svaki od njih čekao da svi ostali procesi završe (slika 4.13.). Krajnji rezultat bio je uspješno prošireni rječnik s kojim se radilo na tri odvojena procesa, odnosno, tri odvojene procesorske jezgre.

```
    print("Usao u glavni main i priprema za start Threada/Processa...\n")
    x = {'x': 10}
    new_manager = multiprocessing.Manager()
    shared_dictionary = new_manager.dict(x)
    p1 = multiprocessing.Process(target=process1, args=(shared_dictionary,))
    p2 = multiprocessing.Process(target=process2, args=(shared_dictionary,))
    p3 = multiprocessing.Process(target=process3, args=(shared_dictionary,))
    p1.start()
    p2.start()
    p3.start()
    p1.join()
    p2.join()
    p3.join()
```

Slika 4.13. Pokretanje i čekanje na završetak svih procesa.

Kako je ovo rješenje bilo prejednostavno za integraciju, odnosno, objekt koji se prebacuje bio je prejednostavan, trebalo je osmisliti način kako napraviti prebacivanje složenijih objekata između procesa. Ovaj način bio bi pogodniji za integraciju u postojeće okruženje pa se trenutno napravljeni dio s procesima trebao modificirati. Odlučeno je da se napravi samo probni program (*demo*) koji će se kasnije integrirati i prilagoditi postojećem okruženju. Trebalo je napraviti objekt koji će biti složeniji na način da se stvori klasa koja će sadržavati nekoliko rječnika te nekoliko listi (slika 4.14.). Također je klasa sadržavala funkcije potrebne za rad klase.

```
class Person(object):
    def __init__(self):
        self.Value = [24, 7]
        self.Height_Weight = [184, 90]
        self.Contact = {
            "FirstName": "Davor",
            "LastName": "Baric",
            "Faculty": "FERIT"
        }
        self.ContactNumbers = {
            "Mob": "+38595xxxxxxx",
            "Country": "Croatia"
        }

    def append_List_HeigtWeight(self, x):
        self.Height_Weight.append(x)

    def append_List_Value(self, x):
        self.Value.append(x)

    def update_Dict_Contacts(self, x):
        self.Contact.update(x, )

    def update_Dict_ContactNumbers(self, x):
        self.ContactNumbers.update(x, )

    def printing(self):
        print("LISTA(HW) je: ", self.Height_Weight, "\nDICTIONARY(Contact)
je:", self.Contact, "\nLISTA(Value) je:", self.Value,
"\nDICTIONARY(ContactNumbers) je:", self.ContactNumbers, "\n")
```

Slika 4.14. Složenija klasa Person.

Svaka od tih klasnih funkcija korištena je u funkcijama za prebacivanje objekta koje su modificirane u odnosu na prvo rješenje. Osim toga kreiran je i *Manager* koji je modificiran u odnosu na staro rješenje, ali je imao istu ulogu – sinkronizaciju među procesima (slika 4.15).

```
def process3(shared_object):
    print("From p3: Usao u Process3")
    time.sleep(4)
    shared_object.update_Dict_Contacts({'YearOfBirth': 1995})
    print("From p3:")
    shared_object.printing()

def main():
    mymanager = MyClassManager()
    mymanager.start()
    shared_object = mymanager.Person()

    p1 = multiprocessing.Process(target=process1, args=(shared_object,))
    p2 = multiprocessing.Process(target=process2, args=(shared_object,))
    p3 = multiprocessing.Process(target=process3, args=(shared_object,))

    p1.start()
    p2.start()
    p3.start()

    p1.join()
    p2.join()
    p3.join()
```

Slika 4.15. Funkcija za rad sa objektom i pokretanje Managera i procesa.

5. VERIFIKACIJA POBOLJŠANOG OKRUŽENJA ZA TESTIRANJE ADAS PROGRAMSKIH RJEŠENJA

Nakon pisanja programa potrebno je provesti testiranja kako bi se vidjelo radi li zamišljeno rješenje upravo onako kako bi trebalo. Testiranja su provedeno na neintegriranom rješenju i to na osobnom računalu te na postojećem okruženju prije i nakon integracije predloženog rješenja i to na računalu tvrtke „*Institut RT-RK*“.

Za početak testiranje dodavanja testova u bazu podataka je provedeno na neintegriranom rješenju, odnosno, rješenju koje je implementirano na osobnom računalu prije integracije u postojeće okruženje. Testiranje neintegriranog rješenja provedeno je na osobnom računalu s procesorom *Intel Core i5-4200M CPU @ 2.50 GHz*, *RAM* memorije od *4GB* te grafičkom karticom *NVIDIA GeForce 710M.* Rad s procesima testiran je samo na neintegriranom rješenju također na osobnom računalu.

Testiranja na postojećem okruženju prije i poslije integracije rađena su na računalu tvrtke „*Institut RT-RK*“ s procesorom *Intel Core i7-6700HQ CPU @ 2.60 GHz, RAM* memorije od *16GB,* grafičkom karticom *NVIDIA GeForce GTX 950M* te diskom *HFS256G32TND-N210A.*

5.1. Testiranje postojećeg okruženja na računalu „Institut RT-RK“

Kako bi se mogla obaviti verifikacija rješenja bilo je potrebno prvo napraviti testiranje na već postojećem okruženju koji ima način spremanja testova u jedan veliki rječnik. Rezultati testiranja postojećeg okruženja na različitom broju testova prikazani su tablicom 5.1. Iz rezultata je vidljivo da se porastom broja testova približno linearno povećava i vrijeme dodavanja u rječnik.

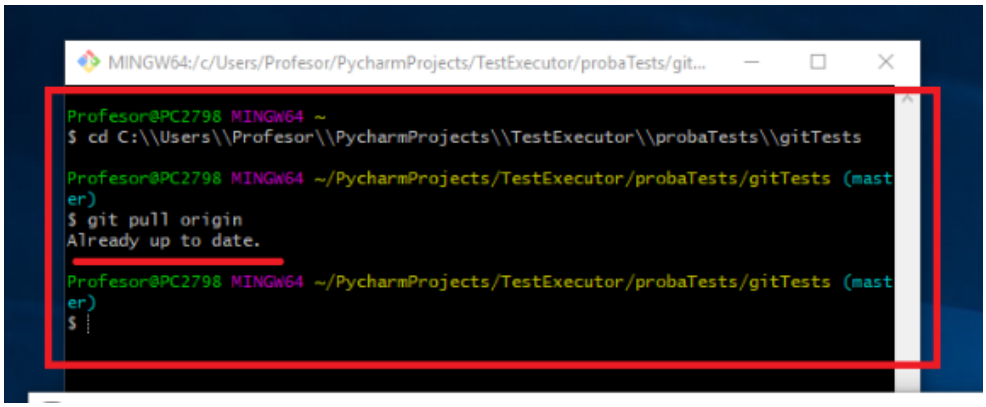
Tablica 5.1. Postojeće okruženje testiran na računalu „Institut RT-RK“.

	POSTOJEĆE OKRUŽENJE					
Broj testova	20	100	500	1000	3000	5000
Vrijeme dodavanja testova (sekunde)	0,07	0,28	1,75	3,49	10,91	17,35

5.2. Testiranje rada *SQLite* baze podataka na osobnom računalu

Prije integracije rješenja u postojeće okruženje, testiranje je provedeno na testovima poput onih koji se koriste u postojećem okruženju. Napravljen je manji set testova od 20 testova i to podijeljenih u mape i podmape. Kasnije je taj set testova postupno povećavan.

Prvi zadatak testiranja bio je pomoću naredbe „*git clone*“ dohvatiti testove sa *GIT-a*. Nakon toga trebalo je pokušati odraditi naredbu „*git pull*“ i vidjeti hoće li doći do promjene na testu u smislu vremena zadnje promjene na testu. Testiranjem se ustanovilo da nije došlo do nove promjene te se samim time nije zahtijevalo nepotrebno ažuriranje u bazu (slika 5.1.).



Slika 5.1. Testiranje nepotrebnog ažuriranja testa u bazu podataka.

Osim provjere „*git*“ naredbi, na početku je bilo potrebno provjeriti je li program uspješno kreirao bazu podataka, odnosno, postoji li baza podataka. Kako je program uspješno kreirao bazu podataka moglo se preći na sljedeće testiranje.

Sljedeći test uključivao je provjeru ispravnog dodavanja svakog testa u bazu podataka. Testiranjem je utvrđeno da se zaglavlje odnosno „*tc_header*“ ispravno zapisuje u bazu podataka. Prva verzija neintegriranog rješenja nije bila podijeljena na funkcije te se neznatno razlikovala od krajnjeg rješenja koje je implementirano u postojeće okruženje. Rezultati prve verzije neintegriranog rješenja provedenih na osobnom računalu prikazani su tablicom 5.2.

Tablica 5.2. Prva verzija neintegriranog rješenje testirana na osobnom računalu.

	NEINTEGRIRANO RJEŠENJE					
Broj testova	20	100	500	1000	3000	5000
Vrijeme pri prvom dodavanju (sekunde)	3,03	15,14	97,96	208,29	610,89	1125,67
Vrijeme nakon što su svi testovi dodani (sekunde)	0,04	0,21	1,17	2,51	20,71	43,18

Kako rezultati nisu bili približni onima kakve daje postojeće okruženje bilo je potrebno u kratkom vremenu pokušati otkriti probleme u programu te probati do neke mjere smanjiti vrijeme. Način

kojim se uspjelo smanjiti vrijeme, odnosno, dodatno ubrzati vrijeme dodavanja testova je bila kombinacija dvije optimizacije u bazi podataka i to postavkama:

- *PRAGMA synchronous = OFF*
- *PRAGMA journal_mode = MEMORY*

Prema zadanim postavkama, *SQLite* baza podataka se pauzira nakon izdavanja naredbe za pisanje na razini operacijskog sustava. Tim načinom se jamči da će svi podaci biti zapisani na disk. Postavljanjem *synchronous = OFF*, upućuje se *SQLite* bazi podataka da jednostavno preda podatke za pisanje operacijskom sustavu i nakon toga nastavi. Druga optimizacija u bazi podataka je *journal_mode = MEMORY* kojom se postavlja način rada koji kontrolira kako se *journal* datoteka pohranjuje i obrađuje. Ovim načinom se ta datoteka sprema u memoriju i na taj način su zapisi u bazu podataka puno brži. Kombinacijom ove dvije optimizacije dobiva se na velikom povećanju brzine zapisa podataka u bazu, iako postoji mali rizik u slučaju pada sustava. Tablicom 5.3. prikazani su rezultati neintegriranog rješenja nakon dodavanja kombinacije dvije optimizacije u bazu podataka. Kako su rezultati nakon dodavanje kombinacije optimizacija izgledali zadovoljavajuće započeta je integracija u postojeće okruženje. Iz tablice 5.3. vidljivo je da krajnja verzija neintegriranog rješenja pokazuje dobre rezultate, kako na malom skupu testova, tako i na velikom. Rezultati su čak i neznatno bolji u odnosu na rezultate postojećeg okruženja koji su prikazani u tablici 5.1.

Tablica 5.3. Krajnja verzija neintegriranog rješenja nakon dodavanja optimizacije za bazu podataka testirana na osobnom računalu.

	NEINTEGRIRANO RJEŠENJE					
Broj testova	20	100	500	1000	3000	5000
Vrijeme pri prvom dodavanju (sekunde)	0,1	0,11	0,52	1,15	9,94	16,27
Vrijeme nakon što su svi testovi dodani (sekunde)	0,06	0,19	1,19	2,5	20,18	40,23

Gledajući rezultate testiranja pri prvoj i krajnjoj verziji kod neintegriranog rješenja vidljivo je da se vrijeme dodavanja testova u novostvorenu bazu dosta smanjilo nakon poboljšanja, odnosno, dodavanja optimizacije. Tom optimizacijom dovelo se do toga da krajnja verzija neintegriranog rješenja daje čak i bolje rezultate u odnosu na postojeći sustav.

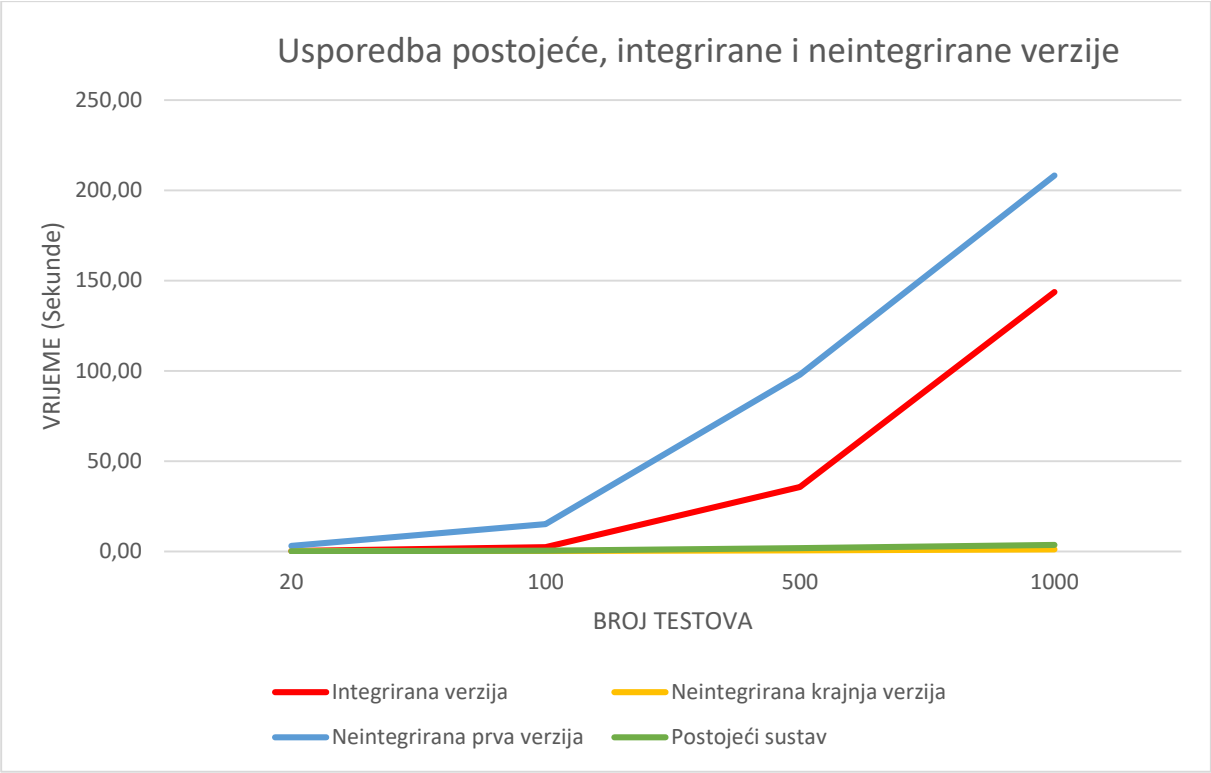
5.3. Testiranje rada *SQLite* baze podataka na računalu „Institut RT-RK“

Kako je krajnja verzija neintegriranog rješenja davala jako dobre rezultate, smatralo se da će isti takvi rezultati biti i nakon integracije u postojeće okruženje. Integracija u postojeće okruženje zahtijevala je promjenu u cijelom programu te prilagodbu novome rješenju. Također, cijeli program je podijeljen u funkcije. Nakon provedene promjene i uspješne integracije bilo je moguće testirati na različitim skupovima testova. Već prvi rezultati nakon uspješne integracije u postojeće okruženje davali su lošije rezultate na malom skupu testova u odnosu na krajnju verziju neintegriranog rješenja. Integrirano rješenje u postojeće okruženje testirano je na računalu tvrtke *„Institut RT-RK“.*

Tablica 5.4. Integrirano rješenje u postojeće okruženje testirano na računalu „Institut RT-RK“.

	INTEGRIRANO RJEŠENJE					
Broj testova	20	100	500	1000	3000	5000
Vrijeme pri prvom dodavanju (sekunde)	0,24	2,27	35,71	143,58	1179,84	3148,32
Vrijeme nakon što su svi testovi dodani (sekunde)	0,19	3,11	64,86	255,15	2147,26	6073,52

Usporedbom postojećeg okruženja i integriranog rješenja u postojeće okruženje vidljivi su veći skokovi u vremenu nakon 100 testova kod integriranog rješenja. Slika 5.2. najbolje prikazuje porast vremena s brojem testova koji se dodaju u bazu.



Slika 5.2. Usporedba vremena izvođenja integrirane i neintegrirane verzije.

Iako je vrijeme dodavanja u bazu podataka smanjeno kod krajnje verzije neintegracijskog rješenja, odnosno, značajno ubrzano u odnosu na prvu neintegriranu verziju, prilikom integracije u postojeće okruženje došlo je do znatnog usporenja. Potrebno je detaljnije usporediti krajnju verziju neintegriranog rješenje sa onim rješenjem koje je integrirano u postojeće okruženje kako bi se integracijsko rješenje dovelo do vremena kakvo se postiže neintegracijskim rješenjem.

Iako su programska rješenja dosta slična, očito je prilikom integracije, prebacivanja koda po funkcijama i uspostavljanja da sve dobro radi s postojećim okruženjem došlo do usporenja rada. Također je moguće da je razlog usporenja stalno provjeravanje postoji li baza i stalnog povezivanja na nju. Neki od razloga također mogu biti nedovoljno dobra optimizacija koda prilikom integracije i slično.

5.4. Testiranje rada s procesima

Nakon završetka testiranja dodavanja testova u novostvorenu bazu podataka bilo je potrebno testirati ispravnost rada rješenja prebacivanja rječnika i liste pomoću procesa. Kako napravljeno rješenje nije integrirano u postojeće okruženje, testiranje ispravnosti rada provedeno je samo na neintegriranom rješenju i to na osobnom PC računalu. Testiranjem se pokazalo kako procesi međusobno dobro komuniciraju te je uspješno izvršeno prebacivanje objekta između procesa.

Pokretanjem upravitelja zadataka na računalu bilo je vidljivo da se rad odvija na više procesorskih jezgri. Takvo rješenje, uz određene modifikacije, moguće je integrirati u postojeće okruženje.

Kao budući dio ovog dijela potrebno je detaljno proći kroz postojeće okruženje te osmisliti kako što lakše prilagoditi ovakvu verziju. Postojeće okruženje trenutno radi pomoću niti, tako da je potrebno određeno vrijeme kako bi se na svim potrebnim mjestima u postojećem okruženju prešlo na rad sa procesima.

6. ZAKLJUČAK

Automobilska industrija svakim danom sve više napreduje. Može se reći da ADAS sustavi predstavljaju osnovu novih vozila. Sustavi kao što su ADAS su složeni te je potrebno provesti detaljno testiranje prije upotrebe kako bi se provjerila ispravnost njihovog rada. Svakim danom testiranje takvih sustava sve je složenije jer se i veličina programskog koda povećava. Kako bi se inženjerima pomoglo u takvim testiranjima te kako bi se smanjili potrebni ljudski resursi i vrijeme provođenja takvih testova, potrebno je imati odgovarajuće programsko okruženje koje će te poslove obavljati umjesto ljudskog rada.

Diplomski rad bavi se poboljšanjem postojećeg okruženja za testiranje ADAS programskih rješenja. Cilj je bio poboljšati trenutni način spremanja testova koji je na postojećem okruženju izveden na način da se testovi spremaju u jedan veliki rječnik. Takav način je dosta nepraktičan, nepregledan i dosta nesiguran i nestabilan te je predložen novi način spremanja u *SQLite* bazu podataka. Još jedan od ciljeva bio je istražiti način na koji je moguće trenutno pokretanje testova na nitima zamijeniti pokretanjem na procesima. Takvim načinom dodatno bi se ubrzalo izvođenje testova jer se testovi na taj način izvode na različitim procesorskim jezgrama za razliku od dosadašnjeg, gdje se niti izvode na jednoj procesorskoj jezgri.

Dio zadatka, koji se odnosi na korištenje *SQLite* baze podataka je uspješno odrađen na način da se testovi koji su se dosad spremali u jedan veliki rječnik, spremaju u *SQLite* bazu podataka. Rješenje prije integracije ima približno jednako vrijeme izvođenja kao i postojeće okruženje koje koristi rječnike kao metodu pohrane podataka. Pokazalo se da je nakon integracije u postojeće okruženje došlo do značajnog usporenja rada, za razliku od rješenja prije integracije u okruženje. Iako su vremena spremanja u bazu podataka bila i znatno veća kod integracijskog rješenja, uspjelo se do neke mjere smanjiti to vrijeme pomoću postavki za optimizaciju baze podataka, ali još uvijek nedovoljno. Kao buduće poboljšanje ovog programskog koda potrebno je detaljnije usporediti programski kod s krajnjom verzijom neintegracijskog rješenja, neznatno drugačijom od integracijskog rješenja koje je trebalo prilagoditi cijelom postojećem okruženju. Detaljnim pregledom trebalo bi optimizirati integracijsko rješenje. Kada se optimizira integrirano rješenje koje je implementirano u postojeće okruženje te se performanse dovedu do onih kakvih daje krajnja verzija neintegracijskog rješenja, cijelo okruženje će biti na zavidno boljoj razini od postojećeg okruženja. Drugi dio zadatka, koji je obuhvaćao rad s procesima, uspješno je izvršen te kako bi se rad unaprijedio potrebno je napraviti integraciju u postojeće okruženje.

LITERATURA

[1] E., Frazzoli, A., Munther, E. Feron, Real-time motion planning for agile autonomous vehicles, Journal of Guidance, Control, and Dynamics, br. 25, sv. 1, str. 116-129, 2002. https://arc.aiaa.org/doi/abs/10.2514/2.4856

[2] D. J., Fagnant, K. Kockelman, Preparing a nation for autonomous vehicles: opportunities, barriers, and policy recommendations, Transportation Research Part A, br. 77, str. 167-181, travanj 2015. https://doi.org/10.1016/j.tra.2015.04.003

[3] C.Y., Chan, Advancements, prospects, and impacts of automated driving systems, International Journal of Transportation Science and Technology, br. 6, sv. 3, str. 208-216, srpanj 2017. https://www.tandfonline.com/doi/full/10.1080/01441647.2018.1524401

[4] Europska komisija, Komunikacija Komisije Europskom parlamentu, Vijeću, Europskom gospodarskom i socijalnom odboru te Odboru regija, Na putu prema automatiziranoj mobilnosti: strategija EU-a za mobilnost budućnosti, str. 1-17, Bruxelles, svibanj 2018. https://eur-lex.europa.eu/legal-content/HR/TXT/PDF/?uri=CELEX:52018DC0283&from=EN

[5] D., Tokody, A., Albini, L., Ady, Z., Rajnal, F., Pongrácz, Safety and security through the design of autonomous intelligent vehicle systems and intelligent infrastructure in the smart city, Interdisciplinary Decsription of Coplex Systems : INDECS, br. 16, sv. 3-A, str. 384-396, kolovoz 2018. https://doi.org/10.7906/indecs.16.3.11

[6] ERTRAC Working Group „Connectivity and Automated Driving“, Automated Driving Roadmap, str.1-54. https://www.ertrac.org/uploads/images/ERTRAC_Automated_Driving_2017.pdf

[7] M., Hasenjäger, H., Wersing, Personalization in advanced driver assistance systems and autonmous vehicles: a review, 2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC), STR. 1-7, Yokohama, listopad 2017.

[8] Oxford Technical Solutions Ltd. https://www.oxts.com/what-is-adas/

[9] S., Singh, S., Critical reasons for crashes investigated in the National Motor Vehicle Crash Causation Survey, Traffic Safety Facts Crash•Stats. Report No. DOT HS 812 506, str. 1-3., Washington, DC: National Highway Traffic Safety Administration, ožujak 2018. https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812506

[10] How ADAS is making autonomous driving a reality https://www.sneci.com/how-adas-is-making-autonomous-driving-a-reality/

[11] Cypress 3.0. https://www.cypress.com/solutions/advanced-driver-assistance-systems-adas

[12] SAE International, SAE International Releases Updated Visual Chart for Its "Levels of Driving Automation" Standard for Self-Driving Vehicles, Warrendale, PA, prosinac 2018. https://www.sae.org/news/press-room/2018/12/sae-international-releases-updated-visual-chart-for-its-%E2%80%9Clevels-of-driving-automation%E2%80%9D-standard-for-self-driving-vehicles

[13] M.Tudor, D.Martinović, Testiranje programske podrške, Pomorstvo, br. 19. sv. 1, str. 285-293, mjesec ? 2005. https://hrcak.srce.hr/3963

[14] A.Cioran, System integration testing of advanced driver assistance systems, Degree project, in automatic control, second level, str. 1-48, Stockholm, lipanj 2015. http://www.diva-portal.se/smash/get/diva2:860599/FULLTEXT01.pdf

[15] *Python* programski jezik https://www.python.org/doc/essays/blurb/ (pristupio 30.7.2019.)

[16] *SQLite* baza podataka https://www.sqlite.org/about.html (pristupio 30.7.2019.)

[17] Paralelna obrada https://www.techopedia.com/definition/8777/parallel-computing (pristupio 1.8.2019.)

[18] Višenitni i višeprocesni način rada https://dev.to/nbosco/multithreading-vs-multiprocessing-in-python--63j (pristupio 1.8.2019.)

[19] Global interpreter lock https://docs.python.org/2/glossary.html#term-global-interpreter-lock (pristupio 24.8.2019.)

SAŽETAK

Danas testiranje u automobilskoj industriji predstavlja vrlo bitnu stavku. Diplomski rad se temelji na već postojećem okruženju za testiranje ADAS programskih rješenja koje je sadržavalo određene mane pri svakodnevnom radu. Unutar diplomskog rada pomoću *Python* programskog jezika bilo je potrebno ispraviti mane na postojećem programskom rješenju. Umjesto spremanja testova u jedan veliki rječnik (engl. *dictionary*), predloženo rješenje koristi *SQLite* baza podataka. Prije same integracije u postojeće okruženje napravljeno je neintegracijsko rješenje koje je rađeno neovisno o postojećem okruženju. Nakon izrađenog neintegracijskog rješenja, bilo je moguće integrirati takvo rješenje u postojeće okruženje. Integracijom u postojeće okruženje cijeli rad s testovima prebačen je na rad s bazom podataka umjesto trenutnog rada s rječnikom. Vremena dodavanja testova, nakon integracije u postojeće okruženje su nešto veća nego u slučaju postojećeg okruženja koji radi s rječnikom, ali s bazom podataka se dobila sigurnost, robusnost te stabilnost cijelog okruženja. Osim rada s bazom podataka, istražen je i način rada s procesima koji bi trebao ubrzati izvođenje testova, jer trenutno rješenje koristi niti.

Ključne riječi: testiranje, Python programski jezik, SQLite baza podataka, niti, procesi

ABSTRACT

TESTING ENVIRONMENT FOR ADAS SOFTWARE SOLUTIONS

Today, testing in the automotive industry is a very important item. The thesis is based on the already existing environment for testing ADAS software solutions, which contained certain disadvantages in everyday work. Within the paper, it was necessary to correct the defects in the existing software solution using Python programming language. Instead of storing tests in one large dictionary, the proposed solution uses an SQLite database. Before integration into the existing environment, a non-integration solution was created that was made independently of the existing environment. Once the non-integration solution was developed, it was possible to integrate such a solution into the existing environment. By integrating into the existing environment, all the test work was moved to database work instead of the current dictionary work. After the integration into the existing environment, the times of adding tests are slightly higher than in the case of the existing environment that works with the dictionary, but the database provides security, robustness, and stability of the entire environment. In addition to working with the database, the process of working with processes that should speed up the execution of tests is explored as the current solution uses threads.

Keywords: testing, Python programming language, SQLite database, threads, processes

ŽIVOTOPIS

Davor Barić rođen je 24.srpnja 1995. godine u Osijeku, Hrvatska. Živi u Đakovu, na adresi Augusta Šenoe 35. 2002. godine započinje osnovnoškolsko obrazovanje u OŠ Ivana Gorana Kovačića u Đakovu. Nakon završenog osnovnoškolskog obrazovanja, 2010. godine upisuje Srednju strukovnu školu Braće Radića u Đakovu, smjer računalni tehničar za strojarstvo. U srednjoj školi se iskazuje na županijskim i državnim natjecanjima iz područja strojarstva. 2013. godine u Osijeku osvaja 2. mjesto na županijskom natjecanju iz predmeta Tehnička mehanika, te samim time osigurava i nastup na državnom natjecanju u Zagrebu iz istog predmeta gdje osvaja 7. mjesto. Godinu dana kasnije osvaja 3. mjesto u Osijeku na županijskom natjecanju iz predmeta Strojarske konstrukcije. Preddiplomski sveučilišni studij računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku upisuje 2014. godine. 2017. godine završava preddiplomski studij te upisuje diplomski studij Automobilsko računarstvo i komunikacije na istom fakultetu.

Od vještina i znanja posjeduje određeno znanje u govoru, čitanju i pisanju engleskog jezika, vozačku dozvolu B kategorije. Posjeduje vještine stečene srednjoškolskim obrazovanjem u dizajniranju i crtanju pomoću CAD alata kao što su AutoCAD i CATIA. Također posjeduje znanje u radu na računalu, rad s Microsoft Office alatima, određeno znanje programskih jezika Python, C, C++ i C# te rad s bazama podataka.

Od akademske godine 2018./2019. stipendist je tvrtke Institut RT-RK iz Osijeka, gdje se obučava za daljnji rad u automobilskoj industriji nakon diplomiranja.

Potpis:

_____