Generator softverskog koda namijenjenog za testiranje ADAS sustava

Mihalj, Andrija

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

GENERATOR SOFTVERSKOG KÔDA NAMIJENJENOG ZA TESTIRANJE ADAS SUSTAVA

Diplomski rad

Andrija Mihalj

Osijek, 2019.

SADRŽAJ

1. UVOD

ADAS sustavi (engl. ***A**dvanced **D**river-**A**ssistance **S**ystems*) elektronički su sustavi koji pomažu vozaču pri vožnji automobila. Koriste se da bi automatizirali, prilagodili i poboljšali sustave unutar vozila u svrhu veće sigurnosti i kvalitetnije vožnje na cesti [1]. ADAS sustavi mogu direktno utjecati na niz komponenti automobila s ciljem razvijanja raznih korisnih značajki sa stajališta sigurnije i komfornije vožnje: od automatskog uključivanja svjetala i prepoznavanja prometnih znakova do integracije s pametnim telefonima.

Ključan faktor u radu ADAS sustava komunikacija je među njegovim komponentama. Pomoću senzora računalni sustav automobila dobiva informacije o neposrednoj okolini automobila, tj. percipira okolinu. Ti senzori su primjerice LIDAR, radar, kamere i ultrazvučni senzori. Informacije dobivene senzorima se odgovarajućim algoritmima obrađuju te se dobiva željeni izlaz – automatsko kočenje, usporavanje vozila na temelju detektiranih znakova za ograničenja brzine, nadzor vozača, detekcija prometnih traka i upozorenje vozaču na prelazak linija i sl. [2] Budući da ADAS sustavi kao takvi mogu imati značajan utjecaj na promet, vozilo i samog vozača, moraju prolaziti kroz detaljna testiranja i prilagođavati se rigoroznim standardima. Sustavi moraju biti robusne strukture i otporni na promjene, imati standardizirane prakse u kôdu i prije svega biti konzistentni u svojim mjerenjima i rezultatima.

Standardiziranu komunikaciju nužno je testirati, što se najbolje odrađuje izradom testnih kôdova za provjeru modela AUTOSAR (engl. ***AUT**omotive **O**pen **S**ystem **AR**chitecture*) komunikacije. AUTOSAR predstavlja međunarodno partnerstvo automotiv tvrtki čija je svrha uvođenje standardizacije u stvaranju softverske i hardverske arhitekture u automotiv razvojnom okruženju. Testno okruženje omogućuje inženjerima koji rade na računalnim sustavima automobila izradu testova nad sustavima bez da moraju koristiti stvarne komponente i riskirati kvarove. Kvalitetno testno okruženje iznimno je bitno jer ako ne simulira vjerno komponente automobila ili krivo obrađuje podatke može doći do katastrofalnih kvarova u praktičnoj implementaciji kôda. Simuliranje testnih okruženja stoga je pedantan i vremenski intenzivan, ali nadasve nužan proces pri izradi ADAS sustava. U ovom radu unaprjeđen je jedan od takvih sustava s fokusom na AUTOSAR komunikaciju u svrhu testiranja ADAS okruženja putem generiranja testnog okruženja.

U okviru diplomskog rada unaprijeđeno je softversko rješenje za testiranje ADAS sustava, s fokusom na komunikaciju. Predloženo rješenje testira komunikaciju između upravljačkih jedinica

(engl. *Electronic Control Unit - ECU*) automobila ili unutar samog CPU-a ECU-a pomoću izrađenog generatora testnog okruženja.

Predloženo rješenje zasniva se na implementiranju načina da se postojeći proces generiranja testnog okruženja ubrza, ponajviše upotrebom metoda za raspodjelu rada programa na više računalnih procesa odjednom, ali i upotrebom rješenja za uklanjanje redundancije u višestrukim izvođenjima programa. Unaprjeđeni program tako omogućuje višestruko ubrzanje sveukupnog rada programa na modernim višejezgrenim procesorima i mogućnost preskakanja dijelova procesa radi uštede vremena.

Diplomski rad sastoji se od pet poglavlja. U drugom poglavlju prikazano je postojeće rješenje generatora testnog okruženja i metode korištene za njegovu realizaciju. Treće poglavlje opisuje unaprjeđenja učinjena nad postojećim rješenjem koja prate zadatak diplomskog rada, a rezultati unaprjeđenja testirani su i prikazani u četvrtom poglavlju i slijednom analizom.

2. GENERATOR TESTNOG OKRUŽENJA

Generator testnog okruženja (engl. *Test Environment Genrator – TEG*) zasniva se na AUTOSAR okruženju i pomaže u testiranju komunikacije automobilskih sustava. AUTOSAR koristi troslojnu arhitekturu:

- osnovni softver: skup standardiziranih softverskih modula koji sadrži servise potrebne za funkcioniranje gornjeg, aplikacijskog sloja,
- RTE (engl. ***R**un**t**ime **e**nvironment*): posrednički sloj koji služi za razmjenu informacija, tj. komunikaciju između osnovnog i aplikacijskog sloja,
- aplikacijski sloj: aplikacijske softverske komponente koje komuniciraju s RTE [3].

Generator testnog okruženja nalazi se upravo u posredničkom sloju AUTOSAR komunikacije. TEG na ulazu prikuplja podatke u obliku signala za specifične ECU-ove na temelju kojih na izlazu generira modele komunikacijskog okruženja. Ovo poglavlje opisuje rad TEG-a, odnosno jednog specifičnog postojećeg rješenja koje ovaj diplomski rad unaprjeđuje.

2.1. AUTOSAR komunikacija

ARXML (**A**UTOSA**R XML**) posebna je vrsta XML datoteke koja se koristi specifično u automotivnom razvojnom okruženju za spremanje podataka o specifičnim komunikacijskim ulazima i izlazima (engl. *ports*) koji se koriste, paketima koji se šalju i softverskim komponentama (engl. ***s**oft**w**are **c**omponents – SWCs*) koje ih obrađuju. ARXML matrica (engl. *schema*) specijalna je definicija XML podatkovnog jezika za razmjenu AUTOSAR komunikacijskih modela koji sadrže podatke o vrstama signala, komponentama i ulazno-izlaznim *portovima*. To je W3C matrica koja definira jezik za razmjenu AUTOSAR modela. Matrica je izvedena iz primarnog opisnog AUTOSAR modela te definira format razmjene podataka u AUTOSAR-u [4].

ARXML dokumenti predstavljaju konfiguraciju jednog ECU-a objašnjenu sve do primitivnih tipova podataka koje koristi. Konfiguracija ECU-a zavisi od situacije do situacije, ali se često nailazi na redundantna testiranja. Ona predstavlja specifične *portove,* softverske komponente i signale koji se koriste u pojedinom ECU u komunikaciji između dva ili više ECU-a. ARXML dakle definira sve tipove podataka potrebne za specifičnu ECU komunikaciju, od osnovnih tipova (*int, float, string*) do onih složenijih (liste, objekti). Koristeći osnovne tipove, ARXML gradi

podatkovni model trenutne komunikacije u obliku signala i pripadajućih *portova* kojima se šalju. Također sadrži i popis softverskih komponenti u koje se generira kôd testnog okruženja.

ECU ARXML datoteke su XML datoteke visoke dubinske složenosti čija veličina može varirati ovisno o ECU-u i potrebama, od par tisuća kilobajta do nekoliko stotina megabajta. Podaci u tim datotekama opisani su počevši od primitivnih tipova podataka na koje se zatim lančano referenciraju sve ostale definicije podataka u ARXML-u.

2.2. Princip rada generatora testnog okruženja

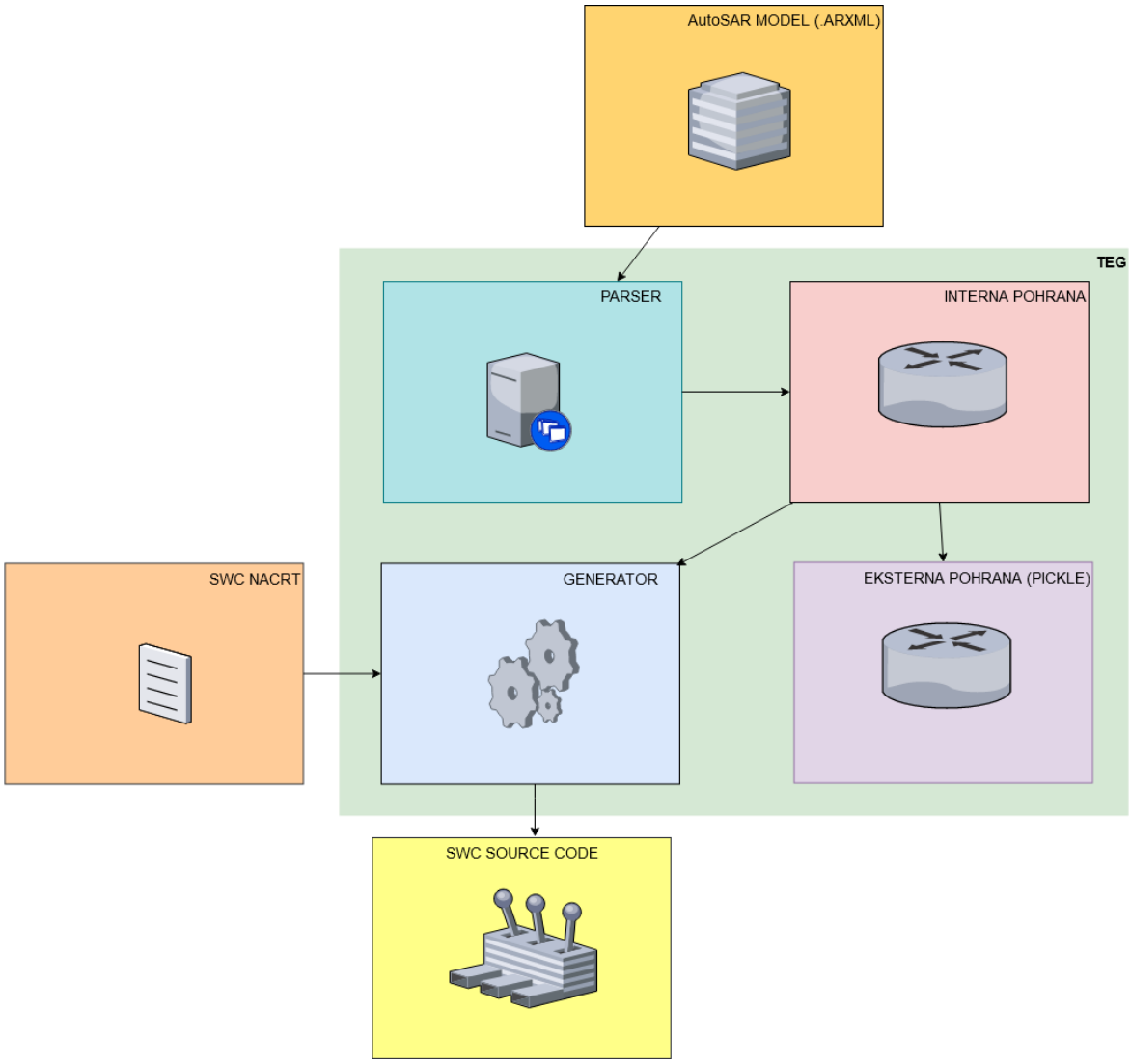
Promjenom konfiguracije ECU-a, odnosno korištenjem različitih konfiguracija, mogu se provoditi različita testiranja i simulacije u AUTOSAR razvojnom okruženju, ovisno o konkretnoj primjeni. Kako bi se podaci prikupljeni o ECU-ovima iz konfiguracije mogli eventualno upotrijebiti za testiranje novog ECU modela, potrebno je iste raščlaniti na smislene cjeline, pohraniti ih te na temelju njih izgenerirati softverske komponente kojima se model testira. To je zadatak generatora testnog okruženja.

Raščlanjeni ARXML podaci prvo se prikupljaju u jednako složenu bazu podataka koja se stvara unutar okruženja programa radi lakšeg zapisa i razmjene podataka između različitih dijelova TEG-a. Stvorena baza, kojoj se brže pristupa negoli *.arxml* datotekama, tvori interni izvor podataka potrebnih za identifikaciju specifičnog kôda koji je potrebno automatizmom unijeti (engl. *code injection*) u specifične i za to predodređene SWC-ove koje skupa tvore model. Među ulaznim datotekama u TEG-u nalazi se *.c* datoteka koja sadrži predloške C kôda koji se unosi u SWC-ove predodređene specifičnom ECU konfiguracijom. Na temelju podataka prikupljenih iz ARXML datoteka identificiraju se potrebni predlošci (specifični za svaki SWC), a zatim iz ostalih prikupljenih podataka unose varijable i vrijednosti u predloške u SWC-ovima.

2.3. Arhitektura generatora testnog okruženja

Generiranje testnog okruženja AUTOSAR komunikacije opisano u prethodnom poglavlju sastoji se od tri temeljna dijela, prikazanih na slici 2.1.:

1. parsera
2. pohrane podataka
3. generatora kôda.



Slika 2.1. *Pojednostavljena načelna shema rada generatora testnog okruženja.*

2.3.1. Parser

Svrha parsera izdvajanje je konfiguracijskih podataka specifičnog ECU-a na temelju kojega se generira model. Parser prima podatke tipa *.arxml*, prolazi kroz njihovu strukturu i pohranjuje parsirane podatke u obliku složene podatkovne strukture koja se dalje koristi kroz cijeli TEG.

2.3.2. Pohrana podataka

Podaci raščlanjeni iz *.arxml* dokumenata pohranjuju se u svrhu daljnjih provjera ispravnosti unešenih podataka u slučaju grešaka ili podešavanja te kao sigurnosna kopija. Prikupljeni podaci predstavljaju sve podatke potrebne za daljnji rad TEG-a, odnosno generatora.

2.3.3. Generator kôda

Podaci raščlanjeni iz *.arxml* dokumenata filterskom se logikom analiziraju kako bi se specifični izrezi C kôda implementirali u predloške softverskih komponenti i, zajedno s pripadajućim vrijednostima varijabli, tvorili testni kôd za provjeru modela AUTOSAR komunikacije za ECU-ove.

2.4. Postojeće rješenje generatora testnog okruženja

Postojeći generator testnog okruženja ima raščlanjenu strukturu datoteka, s kôdom programa odvojenim od generiranih podataka, baze podataka i ulaznih podataka. Pri svakom pokretanju TEG provjeri raspored strukture datoteka koja je konfigurirana u pripadnoj *.ini* datoteci.

Sâm kôd generatora podijeljen je u više povezanih funkcija koje čine ranije navedene cjeline i implementiran je u Python programskom jeziku. Python programski jezik odabran je zbog svoje raznolikosti u mogućnostima pristupa različitim tipovima podataka te dostupnim modulima koji omogućuju brzo raščlanjivanje (parsiranje) *.xml* i *.c* datoteka i jednostavno generiranje C kôda. Korištena je Python inačica 2.7 sa svojom standardnom bibliotekom, uz dodatni modul za parsiranje XML-a i strukturno sličnih tipova datoteka – *LXML*. Pohrana parsiranih podataka odrađena je putem Pythonovog standardnog *Pickle* modula uz popratni XML koji se generira za dodatnu preglednost nad raščlanjenim podacima. Generatorski dio programa koristi pohranjene podatke da u predloške kôda iz glavne *.c* datoteke, napisane u C programskom jeziku, na predodređena mjesta u tzv. fragmentima (manjim C dokumentima koji predstavljaju komponente koje čine model) unese isti kôd, a zatim i specifične varijable, odnosno vrijednosti na mjesta u predlošcima.

2.4.1. Parser

Temeljna komponenta postojećeg TEG-a visoko je složeni Python objekt koji se inicijalizira po predlošcima strukturno predodređenih klasa koje se sve referenciraju jedna na drugu i tvore temeljni objekt, tzv. TOM (engl. ***T**est **O**bject **M**odel*). TOM, koji svojom hijerarhijom odražava hijerarhiju ARXML-a, počinje najvišim objektom u hijerarhiji koji se naziva korijenskim – *TOM Root.*

TOM Root zajedno s lokacijom ARXML datoteka prosljeđuje se kao argument funkciji za raščlanjivanje koja iterativno prolazi kroz sve podatke u ARXML-u i kroz logička grananja pohranjuje ih u predodređene podobjekte TOM-a. Arhitektura TOM-a usko je vezana uz

arhitekturu ARXML-ova, što ograničava izvođenje parsera (engl. *parser*) na sekvencijalni prolaz kroz XML format i paralelno spremanje u odgovarajuće objekte TOM-a.

Rad parsera smatra se gotovim kada su svi relevantni podaci iz ARXML-a preneseni u *TOM Root* koji se onda pohranjuje unutara RAM-a za daljnje korištenje. Za parsiranje XML datoteka koristi se LXML modul za raščlanjivnaje XML datoteka koji je nadogradnja standardnog XML modula u Pythonu, a donosi znatna ubrzanja u radu.

LXML

LXML modul alat je za povezivanje C biblioteka *libxml2* i *libxslt* u Python programskom jeziku. Ovaj modul spaja prednosti navedenih biblioteka, prvenstveno u vidu brzine izvođenja, s prednostima Python aplikacijskog programskog sučelja (engl. ***A**pplication **P**rogramming **I**nterface, API*), prvenstveno u vidu jednostavnosti. LXML unaprjeđuje postojeću standardnu XML Python biblioteku, specifično njezin *ElementTree* API [5]. LXML se koristi u svrhu ubrzanja parserskih operacija i stvaranja novih XML datoteka, a da pritom ostaje unutar Python okruženja.

Na slici 2.2.***2.2.*** [6] prikazana je usporedba performansi pri parsiranju i serijalizaciji podataka između standardnog i LXML *ElementTree* (*eTree*) API-ja. Iz slike je vidljiva jasna prednost korištenja LXML (na slici *lxe*) modula u odnosu na standardni XML (na slici *cET*) modula u svrhu uštede vremena. Modul podržava rad s *.arxml* datotekama. Za postojeće rješenje korištena je inačica 4.2.5 LXML vanjskog modula.

```
lxe: tostring_utf16  (S-TR T1)    7.9958 msec/pass
cET: tostring_utf16  (S-TR T1)   83.1358 msec/pass

lxe: tostring_utf16  (UATR T1)    8.3222 msec/pass
cET: tostring_utf16  (UATR T1)   84.4688 msec/pass

lxe: tostring_utf16  (S-TR T2)    8.2297 msec/pass
cET: tostring_utf16  (S-TR T2)   87.3415 msec/pass

lxe: tostring_utf8   (S-TR T2)    6.5677 msec/pass
cET: tostring_utf8   (S-TR T2)   76.2064 msec/pass

lxe: tostring_utf8   (U-TR T3)    1.1952 msec/pass
cET: tostring_utf8   (U-TR T3)   22.0058 msec/pass
```

Slika 2.2. *Usporedba performansi pri parsiranju i serijalizaciji podataka između standardnog i LXML Element Tree.*

2.4.2. Pohrana podataka

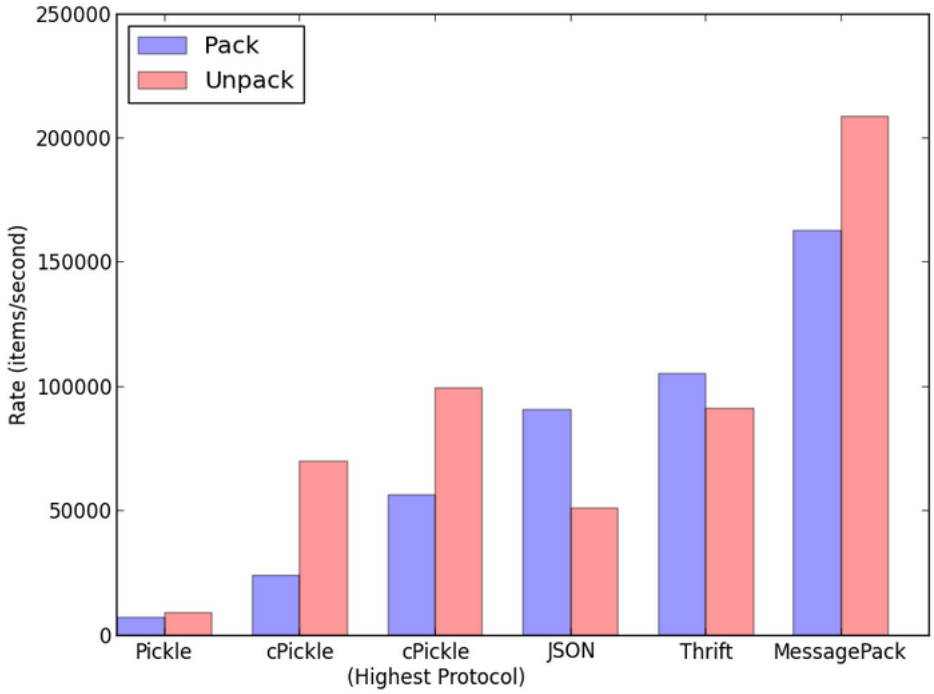
Postojeće rješenje TEG-a oslanja se na standardni Python modul za pohranu podataka serijalizacijom – *Pickle*. Funkcija za pohranu pokreće se nakon parser funkcije, a kao argumente prima *TOM Root* i željenu lokaciju *.pickle* datoteke u direktoriju za pohranu i izlazne podatke. Tom funkcijom cijeli se *TOM Root* jednostavnim slijedom naredbi serijalizira u spomenutu datoteku koja ostaje u direktoriju za pohranu i izlazne podatke i nakon izvršenja programa.

Uz *pickle* funkciju, *TOM Root* se u složenom XML obliku šalje kao argument (uz lokaciju direktorija za pohranu) i sporednoj funkciji za pohranu TOM-a čija je primarna funkcija spremanje TOM-a u preglednijem format u trajnu memoriju, a u svrhu podešavanja rada programa i lakšeg otklanjanja potencijalnih grešaka. Funkcija za pohranu podataka u XML-u, slično kao parser, raščlanjuje sadržaj TOM-a u povezane sekcije XML-a koristeći logička grananja i rekurziju. XML se generira također koristeći LXML modul.

Pickle

Pickle modul za spremanje podataka koristi serijalizaciju. Serijalizacija je metoda pohranjivanja podataka u obliku binarnog toka znakova. Suprotno tome, deserijalizacija je proces pri kojem se iz serijaliziranog *.pickle* dokumenta izvlači originalna struktura (objekt). *Pickle* serijalizaciju provodi pomoću virtualnog stroja zasnovanog na stogu koji sprema upute potrebne za rekonstrukciju objekta pri deserijalizaciji [7].

Pickle modul iznimno je popularan zbog svoje jednostavnosti izvedbe i lakoće s kojom se nosi i s najsloženijim Python objektima, ali se zbog svojih nedostataka također naširoko smatra previše nestabilnim za profesionalan rad. Problem stabilnosti *Picklea* leži u tome što u slučaju kritičnog kvara na računalu (pad sustava, nestanak struje i sl.) dolazi do potpunog gubitka pohranjenih podataka. Serijalizirani podaci u *.pickle* datoteci čitljivi su samo Pythonovom *Pickle* modulu zbog svojeg specifičnog binarnog zapisa. *Pickle* je samim time, osim nepreglednosti, iznimno nesiguran za korištenje jer je nemoguće obaviti pretpregled podataka u datoteci, što znači da je moguće ubaciti pogrešan ili maliciozan kôd u sustav. Također, kao metoda serijalizacije podataka u svrhu pohrane, *Pickle* je najsporija metoda u odnosu na JSON, Thrift i ostale slične biblioteke kao što se vidi na slici 2.3. gdje se u sekundama mjerio proces spremanja (*Pack)* i čitanja (*Unpack*) podataka iz pohrane [8]. Pohranjeni podaci u *.pickle* datoteci služe samo kao mjera za praćenje rada TEG-a izvan samog izvođenja programa i, nakon što se datoteka stvori, TOM se (nepromijenjene strukture i sadržaja) šalje generatoru istovremeno kada je i poslan *Pickle* modulu.



Slika 2.3. *Usporedba brzine rada Picklea i drugih modula slične namjene* [8].

2.4.3. Generator

Funkcija za generiranje testnog kôda prima kao argumente *TOM Root* i lokacije glavne *.c* datoteke s predlošcima i potrebnim SWC komponentama, također u obliku *.c* datoteka. Glavna C datoteka sadrži predloške C kôda u obliku definicija varijabli, klasa i funkcija s predodređenim mjestima za smještanje vrijednosti prikupljenih iz ECU-a. Ona se metodom regularnih izraza (engl. ***Reg****ular* ***Ex****pressions*, *RegEx*) koristeći logički filtrirane podatke iz TOM-a raščlanjuje na spomenute predloške, tj. izreze kôda. Predlošci se umeću na predodređena mjesta u kôdu SWC *.c* datoteka, a zatim se ponovno iz TOM-a na predložena mjesta umeću varijable i vrijednosti specifične za ECU koji se testira.

Rad generatora vremenski je daleko najzahtjevnija komponenta TEG-a čije vrijeme izvođenja ovisi o veličini ulaznih datoteka. Generator je također dizajniran da sekvencijalno koristi *RegEx* dok sastavlja komponente modela što negativno utječe na performanse programa (u vremenskom smislu) kako se iznos ulaznih podataka povećava.

Python RegEx modul

Regularni izrazi nizovi su znakova koji definiraju uzorak za pretragu. Koriste se uglavnom za pretraživanje i/ili izmjenu teksta (*stringa*) ili za validaciju unesenog teksta, a temelje se na teoriji

formalnih jezika [9]. Funkcionalnost regularnih izraza u generatoru testnog okruženja postignuta je Pythonovim modulom RegEx (točnije, standardna inačica *re*).

Generator logičkim grananjima (IF-ELSE naredbama) i regularnim izrazima pronalazi početke i krajeve predložaka u glavnoj *.c* datoteci s predlošcima. Generator zatim logičkim grananjima prolazi redom kroz svaku SWC komponentu i u nju smješta predloške na specifične lokacije, također pronađene regularnim izrazima. Zatim se unutar novostvorenih predložaka u SWC komponentama smještaju vrijednosti varijabli specifičnih za trenutni ECU nad kojim se izvodi TEG. U listingu 2.1. prikazan je isječak kôda regularnog izraza koji pomoću Python biblioteke *re* pronalazi početak definicije C funkcije.

```
import re

regexStartOfFunctionDefinition = re.compile \
        ( r"<< Start of function definition area >>"
         r".*?\*/."
        , re.DOTALL | re.MULTILINE
        )
```

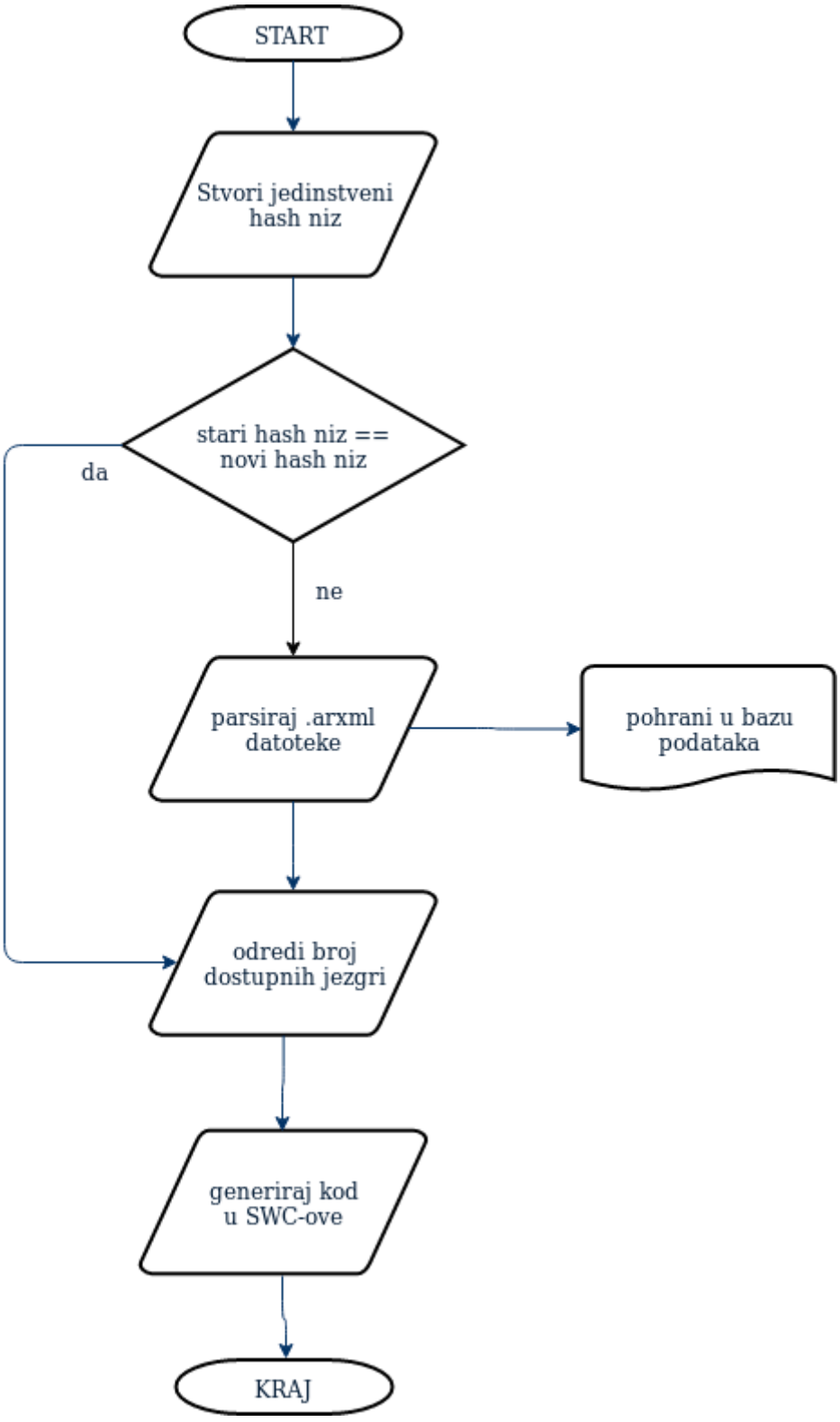
Listing 2.1. *Isječak kôda regularnog izraza koji pronalazi početak definicije C funkcije.*

3. UNAPRJEĐENJE GENERATORA TESTNOG OKRUŽENJA

Temeljni cilj ovog diplomskog rada ubrzanje je postojećeg TEG-a. S povećanjem veličine ulaznih podataka značajno se povećava vrijeme izvođenja ukupnog programa što predstavlja praktičan problem. Osim toga, cilj je i oformiti stabilniji oblik pohrane podataka u obliku baze podataka umjesto serijalizacije podataka kao načina pohrane. Konkretno, zahtjevi za unaprjeđenu verziju TEG-a su:

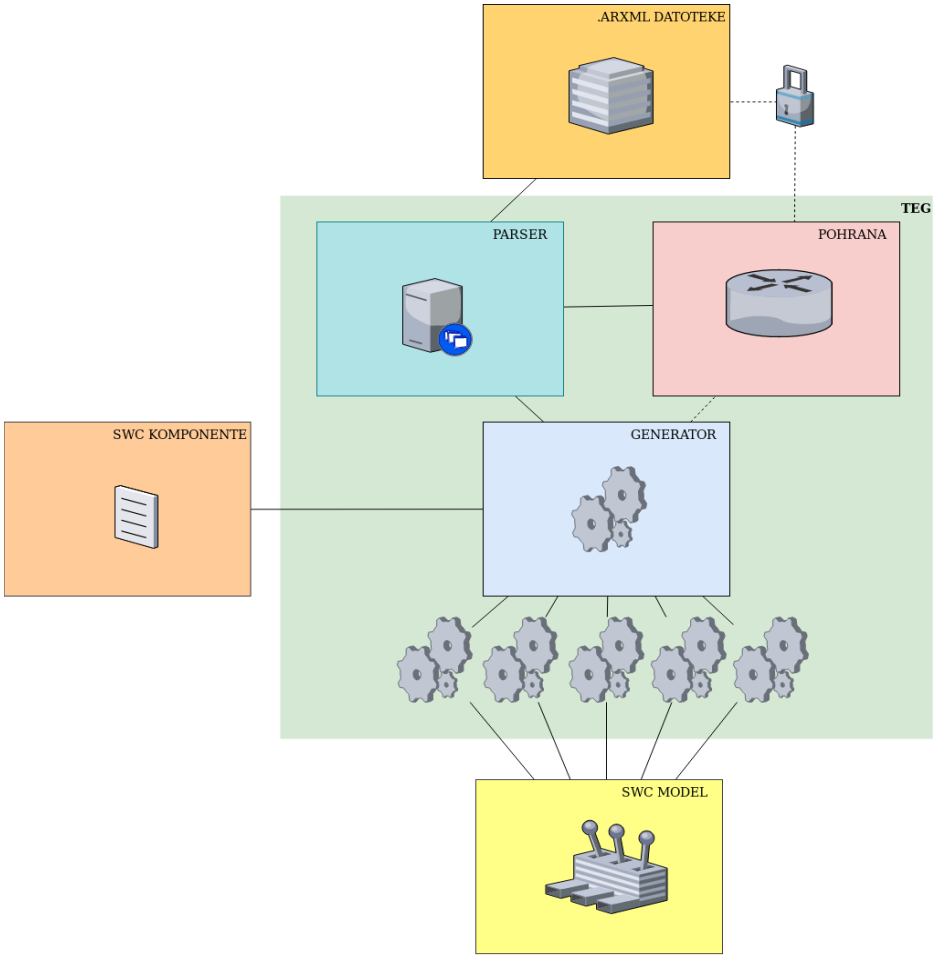
1. ubrzanje postojećeg TEG-a tako da mu je potrebno manje vremena za raščlanjivanje i generiranje izlaznih datoteka,
2. promjena načina pohrane podataka kako bi ista bila stabilnija i preglednija.

Predloženo rješenje odrađeno je implementiranjem promjena kôda u postojeće rješenje. Promjene su specifične za svaki pojedini dio programa, ali je njegova struktura i redoslijed izvođenja ostali nepromijenjeni. Predloženo rješenje tako koristi Python 2.7 inačicu. TEG pri pokretanju provjerava *.ini* konfiguracijsku datoteku čiji se sadržaj zatim predstavlja jedinstvenim *hash* nizom znakova. Kao prikaz datoteka u TEG-u odabran je *hash* niz jer je to jedan od najbržih načina zapisivanja podataka s velikim brojem mogućih varijabli, koji se kreira i provjerava jednostavnim funkcijama. Primarna komponenta TEG-a, parser, pokreće se na identičan način – putem dodatnog Python LXML modula za parsiranje ARXML-a. ARXML se parsira u incijalizirani TOM – temeljni objekt. TOM se zatim putem Python *multiprocessing* modula usporedno šalje u generator i sprema u SQL (engl. ***S****tructured* ***Q****uery* ***L****anguage*) bazu podataka pomoću Python SQLite modula. Generator prima TOM i putem nove *multiprocessing* instance obavlja generiranje SWC-ova u više procesa odjednom. Pri rekurzivnom pokretanju TEG-a nad identičnim skupom podataka, TEG ima mogućnost preskakanja procesa parsiranja u svrhu uštede vremena. Dijagram rada unaprjeđenog TEG-a prikazan je na slici 3.1. zajedno s rekurzivnom funkcionalnošću putem *hash* niza.



Slika 3.1. *Dijagram toka rada unaprjeđenog TEG-a.*

Dodatno, radi lakše vizualizacije unaprjeđenja TEG-a, slika 3.2. prikazuje pojednostavljenu načelnu shemu rada unaprjeđenog TEG-a na temelju sheme postojećeg.



Slika 3.2. *Pojednostavljena načelna shema rada unaprjeđenog TEG-a.*

3.1. Parsiranje uz provjeru *hash* niza

Zbog arhitekture ARXML datoteka i prateće arhitekture TOM temeljnog objekta, nije bilo moguće implementirati znatne promjene u radu parsera s ciljem ubrzanja bez provođenja temeljitog restrukturiranja cijelog postojećeg rješenja. Korišteni LXML modul za parsiranje ARXML-a nije imao znatne nadogradnje u novijim inačicama od vremena implementacije u postojeće rješenje TEG-a u vidu ubrzanja rada. Prilikom istraživanja i izrade samog diplomskog rada, a prije prvog pristupa kôdu postojećeg rješenja, razvijene su gotovo identične metode prolaska kroz gustu ARXML strukturu i pratećeg TOM-a. Stoga ubrzati rad samog parsera nije bilo izvedivo u ovom diplomskom radu, ali je zato razvijena metoda ubrzanja rada cijelog TEG-a – točnije, da se uopće izbjegne pokretanje parsera u svrhu uštede vremena u vidu vremena rada čitavog programa.

Ubrzanje je postigunto validacijom strukture nove inačice TEG-a sa strukturom zadnje zapisane u baze podataka. Za tu potrebu korišten je standardni Python modul *hashlib*. Nakon parsiranja, *Tom Root* se šalje kao argument paralelno generatoru i funkcijama za spremanje u bazu podataka.

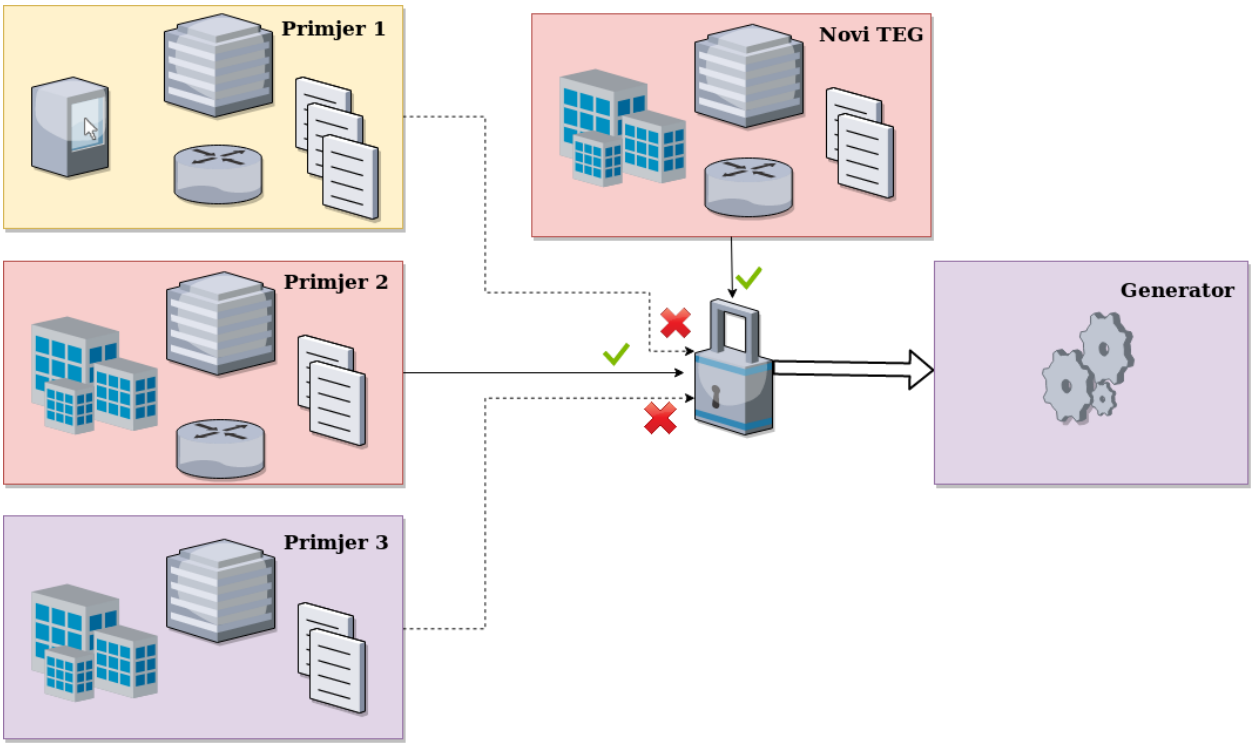
Hashlib modul

Kod parsiranja *.ini* konfiguracijske datoteke, Python kôd stvara jedinstveni *hash* niz koji predstavlja točnu strukturu TEG-a, odnosno njegovih datoteka. Kada se TOM sprema u SQL bazu podataka u drugom koraku, također se pohranjuje i *hash* niz. Pri drugom pokretanju generatora, TEG uspoređuje prethodno spremljeni *hash* niz u datoteci baze podataka (ako ona uopće postoji) s *hash* nizom generiranim nad trenutnom strukturom datoteka. U slučaju poklapanja *hash* nizova, što potvrđuje identičnu strukturu novog i starog TEG-a, TEG preskače proces parsiranja ARXML datoteka i TOM se samo ispuni podacima iz baze.

Proces validacije ilustriran je na slici 3.3.***3.3.*** Prikazana su tri primjera koji predstavljaju tri moguće situacije pri validaciji *hash* niza:

- Primjer 1 predstavlja TEG strukturu koja je različita od novog (trenutno pokrenutog) TEG-a stoga se *hash* nizovi neće poklapati pa se TEG pokreće iznova,
- Primjer 2 predstavlja TEG strukturu koja je identična novoj stoga se *hash* nizovi poklapaju pa TEG preskače direktno na generatorski dio programa, i
- Primjer 3 predstavlja TEG strukturu koja je identična novoj, ali nema bazu podataka sa spremljenim nizom stoga se *hash* nizovi ne mogu poklapati; ovaj primjer tehnički je nulti slučaj, ali ilustrira jednostavnost, a samim time i manu ovako jednostavne provjere.

Složenija provjera bila bi moguća ako bi se odredila dubina do koje bi se u bazu podataka trebalo ući koja se smatra dostatnom da se s tom količinom informacija pouzdano može preskočiti parsiranje. Takva provjera nužno bi zahtijevala i preliminarno parsiranje ARXML-ova do iste dubine, što dovodi do pitanja: do koje se dubine isplati ići, a da se ne žrtvuje previše vremena? Također bi se moglo reći i da preskakanje parsiranja kada ono nije trebalo biti preskočeno dovodi do još većeg gubitka vremena i odmicanja od cilja. U praksi, takvu odluku će uvijek točnije obaviti sama osoba koja koristi generator testnog okruženja negoli trenutni kôd.



Slika 3.3. *Proces validacije hash niza na više primjera.*

3.2. Pohrana podataka u relacijsku bazu

Pohrana podataka prebačena je sa serijalizacijske na relacijsku. Relacijske baze podataka temelje se na relacijskom modelu podataka. U relacijskom modelu podataka podaci su prikazani kao n-torke grupirane u tablice, tj. relacije. Dakle, svaka n-torka u relacijskom modelu podataka zapravo je jedan redak u tablici s određenim brojem imenovanih stupaca koji predstavljaju imena atributa objekata koji se u tablicu spremaju. Povezanost tablica u relacijskim bazama podataka postiže se stranim ključevima (engl. *foreign* key) koji služe kao paralela između redaka dviju povezanih tablica. Sustavi koji upravljaju relacijskim bazama podataka zovu se RDBMS (engl. ***R****elational* ***D****atab****a****se* ***M****anagement* **S**ystem). Kao relacijska baza podataka, odnosno RDBMS, u ovom radu korišten je SQLite. SQLite omogućuje stvaranje, zapisivanje u, interakciju i brisanje SQL baza podataka koristeći Python naredbe. SQLite je odabran prvenstveno zato što za TEG nije potrebno podizati poslužitelj, nego se baza podataka koristi samo za internu uporabu te može funkcionirati i bez internetske veze (*offline RDBMS*). Također, baza podataka spremljena u SQLite datoteci lako je prenosiva budući da se radi o samo jednoj datoteci koja je generalno vrlo male veličine i ne ovisi o drugim, vanjskim bibliotekama i programima. Osim toga, SQLite je lako dostupan (otvorenog je kôda) te je vrlo popularan, što znači i odličnu softversku podršku.

Prije pokretanja funkcije za pohranu u SQLite bazu podataka, u roditeljskoj se funkciji postojećeg rješenja TEG-a stvori sâm temelj baze podataka s inicijalnom tablicom koja predstavlja *TOM Root*. Funkcija za pohranu podataka prima kao argumente *Tom Root,* inicijalnu tablicu i lokaciju baze podataka u datotečnom sustavu. Funkcija slijedno prolazi kroz svaki objekt u TOM-u i zasebno validira svaki pojedini atribut. Prvotno svaki atribut prolazi kroz logički filter za detekciju grešaka (pogrešni tipovi podataka, greške u sintaksi, prazni podaci) koje se izbacuju. Zatim, kroz logičko grananje se utvrđuje radi li se o primitivnim tipovima podataka (*integer, float, string, long*), nepromjenjivim popisima n-torki (*tuple*), nizovima podataka (*lists*) ili ugniježđenim objektima. Ako se radi o prvom slučaju, baza u trenutnoj tablici (koja dijeli naziv s objektom) stvara novi stupac (ako isti već ne postoji). Za nizove podataka, popise i ugniježđene objekte pokreće se rekurzivna funkcija za pohranu podataka unutar zasebne petlje. Funkcija tada za argumente prima objekt koji treba ući u rekurziju, njegovu tablicu i tablicu roditeljskog objekta na koju će se referencirati putem stranog ključa. Funkcija za pohranu podataka tako rekurzijom prolazi kroz cijeli TOM i pohranjuje njegov sadržaj u složenu relacijsku bazu podataka.

Unutar ARXML standarda, nazivi komponenti se često ponavljaju. Iz tog razloga, pri izradi naziva tablice, zbog specifičnosti rada SQL-a koji ne dopušta dvije tablice s istim nazivom, svakoj tablici dodaje se numerički prefiks da bi se izbjegle greške zbog istih naziva tablica. Prefiks se dobiva pomicanjem varijable tipa integer svakim stvaranjem nove tablice, počevši od nule za *TOM Root* tablicu. U bazu se također sprema i *hash* niz koji služi za izbjegavanje redundancije pri rekurzivnom pokretanju TEG-a. Pisanje u bazu odvija se usporedno s generatorom.

SQLite modul

SQLite Python modul sustav je za upravljanje relacijskim bazama podataka, tj. RDBMS sadržan u jednoj biblioteci. Za razliku od većine drugih sustava za upravljanje bazama podataka, SQLite se ne temelji na komunikaciji klijent-poslužitelj, nego je ugrađen u sâm program koji se pokreće [10], u ovom slučaju TEG. SQLite većinom koristi SQL standard i sintaksu, no, budući da koristi dinamičku sintaksu, nema potpune provjere integriteta te je moguće, primjerice, unijeti podatak tipa *string* u stupac definiran kao *integer*. Zato SQLite, gdje je to moguće, radi konverziju u zadani tip podatka, a u suprotnom sprema podatak kako mu je pružen [11].

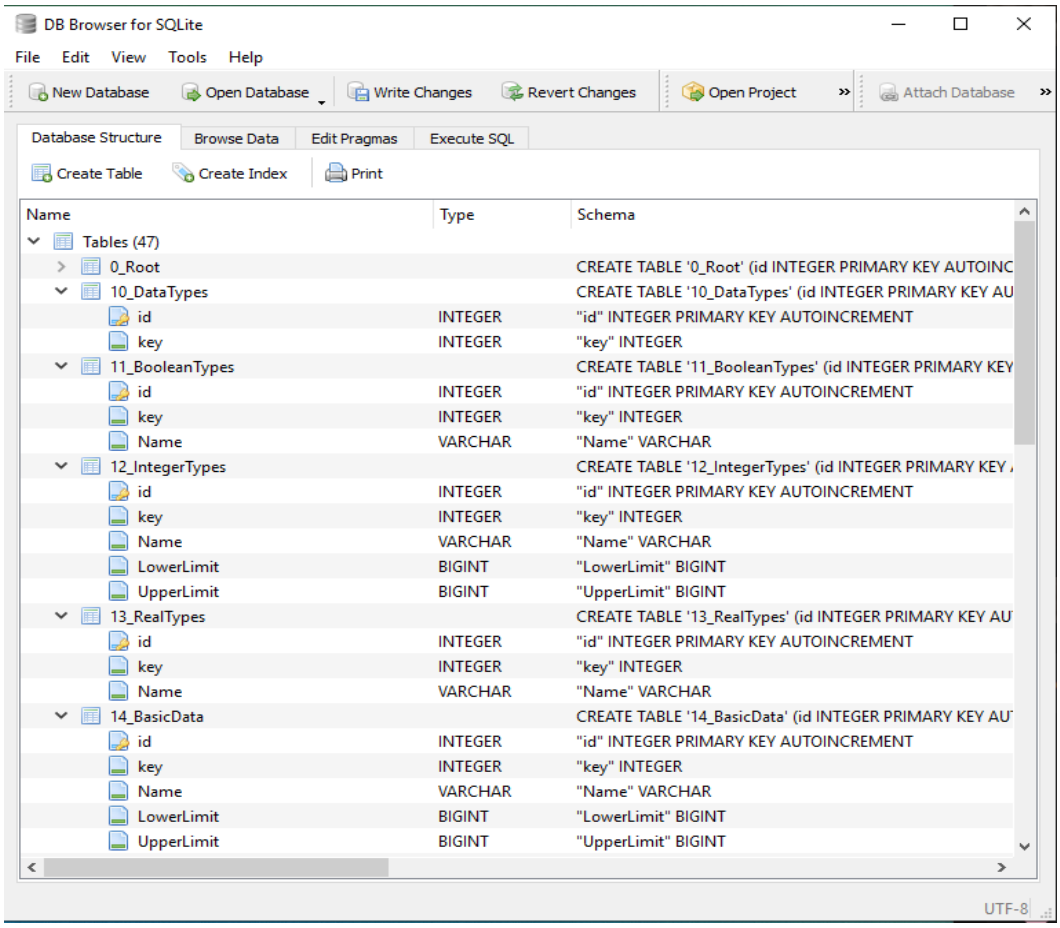
SQLite datoteka čitljiva je bilo kojim SQL alatom, a za potrebe ovog diplomskog rada korišten je besplatni alat otvorenog kôda *DB Browser for SQLite*. U listingu 3.1. vidljiv je primjer SQLite Python kôda u upotrebi. Unutar kôda se uspostavlja veza s novom datotekom baze podataka i šalje se SQL upit (engl. *query*) u obliku *stringa* pisanog SQL jezikom za upite.

```
import sqlite

# putanja odredista SQL baze podataka u projektu
database_location = '''C:\\Mihalj\TEG\\Data\\Database-sql.db'''
# stvaranje SQL baze podataka
conn = sqlite3.connect( database_location)
c = conn.cursor()
# stvaranje temeljne/inicijalne tablice u TOM-u
query = "CREATE TABLE IF NOT EXISTS '{}' (id INTEGER PRIMARY KEY
AUTOINCREMENT)".format( '0_' + tomRoot.GetTag())
c.execute(query)
conn.commit()
# inicijalno pokretanje funkcije za pohranu podataka
storeDataToSql( tomRoot, '0_' + tomRoot.getTag(), None, None, 0 )
```

Listing 3.1. *Isječak kôda koji predstavlja stvaranje baze podataka i temeljne tablice putem SQLite Python modula.*

Na slici 3.4. ***3.4.*** vidljiv je pogled u gotovu bazu podataka pomoću *DB Browser for SQLite* koja u sebi sadrži kompletne podatke iz *.arxml* datoteka prikupljene iz TOM temeljnog objekta.



Slika 3.4. *Pogled u gotovu SQL bazu podataka unutar* DB Browser for SQLite *alata.*

3.3. Paralelizacija zadaća unutar generatora

Analizom postojećeg rješenja identificirano je da postoji problem gomilanja rada generatora koji usporava rad TEG-a pri većem broju podataka koje treba obraditi. Zato je odlučeno da implementacija vlastitog rješenja u postojeći generator neće iziskivati kvalitetniji algoritam, već raspodjelu rada postojećeg algoritama na više procesorskih jezgri kako bi se ubrzalo generiranje testnog koda. Paralelizacijom kôda, odnosno korištenjem više procesa, očekuje se znatno ubrzanje izvođenja TEG-a zbog generatora kao najznačajnije komponente TEG-a.

Stoga je implementiran, na bazi cijelog TEG-a, Pythonov *multiprocessing* modul koji na više računalnih procesa rastavlja rad jedne ili više funkcija. Rješenje zasebno za svaki SWC grupira generiranje kôda na više istovremenih procesa. SWC-ovima se dodjeljuju računalne jezgre ovisno o ukupnom broju dostupnih jezgri na računalu. Fragmenti, odnosno predlošci, iz glavnog se C dokumenta paralelno unose kao komponente korištenjem regularnih izraza, a zatim se unose i vrijednosti vezane uz specifični ECU na predložena mjesta.

Multiprocessing modul

Multiprocessing modul je biblioteka koja omogućuje stvaranje procesa korištenjem API-ja sličnog standardnom *threading* modulu, ali se ne bazira na radu s više niti, nego na radu s više procesa. *Multiprocessing* modul omogućuje i lokalni i udaljeni (engl. *remote*) istovremeni rad [12]. Za ovaj diplomski rad odabran je upravo *multiprocessing* modul umjesto *threading* modula. Kod višenitnog rada (*threading* modul) niti rade s istom memorijskom hrpom, dok procesi rade s više memorijskih hrpa. To otežava dijeljenje informacija tj. podataka među procesima. Međutim, korištenje iste memorijske hrpe može dovesti do pisanja u istu memorijsku lokaciju. Također, kompliciranije je implementirati višenitni rad unutar Pythona zbog mjera sigurnosti ugrađenih u njegov *GIL* (engl. *Global Interpreter Lock*) u odnosu na korištenje više jezgri koje je samo hardversko ograničenje koje na modernim procesorima više ne postoji. Osim toga, raspoređivanjem procesa bavi se operacijski sustav, a raspoređivanjem niti bavi se sama threading biblioteka što može znatno usporiti rad programa. Budući da TEG koristi *multiprocessing* za veliku količinu zapisivanja u datoteke, iznimno je bitno da ne dođe do pogrešaka u zapisu. Uzevši u obzir sve navedeno, zaključeno je da za potrebe ovog diplomskog rada *multiprocessing* ima znatnu prednost nad *threadingom*.

Na početku rada generatorskog dijela TEG-a provjerava se broj jezgri procesora računala te se ovisno o tom broju pokreću procesi za paralelni rad. Istovremeno se stvara lista svih SWC

komponenti i nad procesima se pokreću iteracije iste funkcije za generaciju testnog kôda, svaka s jednim SWC-om s liste – dok se lista ne iscrpi. U listingu 3.2. vidljiv je primjer kôda koji u redu (engl. *queue*) procesa prolazi kroz popis SWC-ova u obliku strukture *lstSWCs* i dodjeljuje ih idućim dostupnim procesima.

```
import multiprocessing

# Prolazak kroz listu SWC-ova i dodjeljivanje procesa
objTOM = Queue()

for SWC in lstSWCs :
        proc = Process \
                ( target = iterateThroughSWCs
                , name = "process_" + str( SWC.name )
                )
        lstProcesses.append( proc )
```

Listing 3.2. *Isječak kôda prolaska kroz popis SWC-ova i dodjele istih procesima.*

4. VERIFIKACIJA UNAPRJEĐENJA GENERATORA

Na temelju promjena, odnosno unaprjeđenja napravljenih u okviru diplomskog rada u vidu dodavanja logike za višestruko procesiranje (*multiprocessing*), logike koja rukuje postojećim funkcijama, preskakanja postojećih koraka u izvođenju programa te prelaska na drugačiji oblik pohrane identičnih podataka bilo je potrebno ispitati unaprjeđenja na sljedeći način:

- verifikacijom ispravnosti zapisa prenesenih podataka u SQLite bazu podataka
- mjerenjem brzine izvođenja modificiranog TEG-a i usporedbom s brzinom procesiranja postojećeg TEG-a.

Sva testiranja za postojeći i unaprjeđeni generator testnog okruženja obavljena su na istom setu podataka u obliku dva para *.arxml* datoteka veličine 5.2 i 5.6 MB koje su obrađivane istovremeno.

Testiranja su odrađena na računalu s instaliranim *Windows 10* operacijskim sustavom, Python inačicom 2.7, i hardverskih specifikacija prikazanih u tablici ***4.1.***

Tablica 4.1. *Specifikacije hardverskih komponenti korištenih za testiranje.*

Vrsta komponente	Komponenta
Procesor	Intel i7, 4.5 GHz, 4 jezgre
Radna memorija	16 GB, DDR4, 2666 MHz
Pohrana (R/W)	NVMe SSD, 1500/1000 MB/s

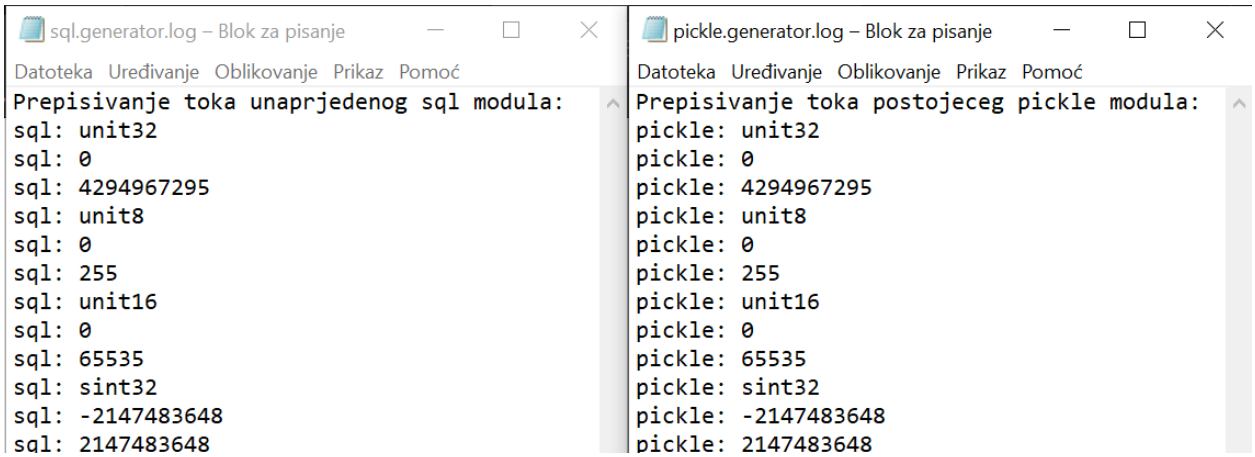
4.1. Opis korištenih testova

Verifikacija ispravnosti prenesenih podataka u SQLite bazu podataka odrađena je ručnom usporedbom vrijednosti prenesenih iz baze na generator za dva dostupna testa. Na kôd generatora unesene su privremene funkcije za ispis svih vrijednosti koje je isti prošao u zasebnom zapisniku (engl. *log*) za slučajeve postojećeg i novog rješenja i zatim uspoređene kako bi se utvrdila ispravnost.

Verifikacija uspješnosti ubrzanja TEG-a odrađena je putem zasebnog *Python* modula koji u *Windows* naredbeni redak ispisuje vremena izvođenja pojedinih dijelova TEG-a (parser, baza, generator) te ukupno vrijeme izvođenja. Vrijeme je izraženo u formatu h:mm:ss.sss i izlistava se u naredbenom retku kako se koja komponenta TEG-a izvrši. Za mjerenje vremena korišten je standardni Python *time* modul.

4.2. Rezultati testiranja ispravnosti zapisa u bazu podataka

Budući da se SQLite baza podataka sastoji od tablica i stupaca ispunjenih podacima, a *Pickle* podaci su samo serijalizirani kôd TOM-a, nije moguće direktno uspoređivati sadržaj dvaju rješenja. Usporedbu je stoga potrebno raditi na idućem koraku rada generatora testnog okruženja. Na slici 4.1. vidi se podudaranje nasumično odabranih pohranjenih upisa iz toka prebacivanja podataka iz TOM temeljnog objekta u generatorsku komponentu TEG-a koji se izvodio nad identičnim podacima uz različit kôd.



Slika 4.1. *Usporedba zapisnika generatora postojećeg i unaprjeđenog rješenja TEG-a.*

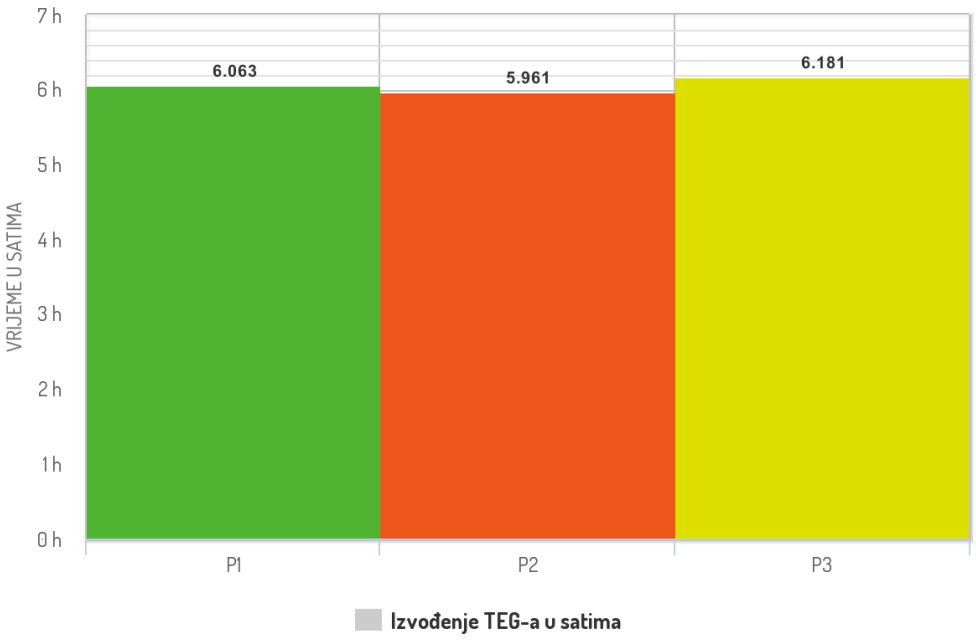
4.3. Rezultati mjerenja brzine postojećeg rješenja generatora testnog okruženja

Mjerenja su prikazana na unutar Windows naredbenog retka i na slici 4.2.***4.2.*** prikazan je primjer izvođenja postojećeg generatora testnog okruženja. Ovaj prikaz u Windows naredbenom retku industrijski je standard, a odvaja bitne dijelove izvedbe TEG-a (parser s bazom od generatora) i mjeri brzinu izvođenja svake posebno, s ukupnim mjerenjem prikazanim na kraju ispisa.

```
TestEnvGenerator ( Version 3.2.02 / 2019-07-21 )
    DataModelParser
    .. == VehicleBusFR  done ( 0:00:00 )
    .. == SignalMapping  done ( 0:00:00 )
    .. == VehicleBusCAN  done ( 0:00:00.220000 )
    .. == AutosarModel  done ( 0:00:10.167000 )
    .. DatabaseTEG  done ( 0:00:00.045000 )
    .. DatabaseTEG  done ( 0:00:00.700000 )
    SourceFileGenerator
    .. TestCaseList  done ( 1:31:13.183000 )
    .. == TestModuleBUS  done ( 0:48:13.670000 )
    .. == TestModulePFF  done ( 0:47:83.382000 )
    .. == TestModulePER  done ( 0:48:17.641000 )
    .. == TestModuleITF  done ( 1:22:10.111000 )
    .. == VerifyTemplates  done ( 0:20:00.875000 )
.. TestEnvGenerator  done ( 6:00:12.005000 )
```

Slika 4.2. *Ispis dobiven prilikom izvođenja postojećeg generatora testnog okruženja.*

Na slici 4.3. prikazan je graf vremena izvođenja postojećeg generatora testnog okruženja. Svaki *P* predstavlja jednu instancu izvođenja TEG-a u satima, zaokruženim na tri decimalna mjesta. Razlike u izvedbama svakog pojedinog primjera TEG-a (*Pn*, *n=1,2,3*...) unutar su tolerancije zbog nepredvidljivosti okruženja eksperimenta, u ovom slučaju računala – nije realno očekivati da će računalo svaki put izvoditi program u identičnom vremenskom roku zbog razlika izvedbe operacijskog sustava. Vidljivo je da postojeće rješenje zauzima znatan dio vremena korisniku programa (preko šest sati) i zašto se zadatak ovog diplomskog rada fokusirao isključivo na vrijeme izvedbe.



Slika 4.3. *Vrijeme izvođenja postojećeg generatora testnog okruženja.*

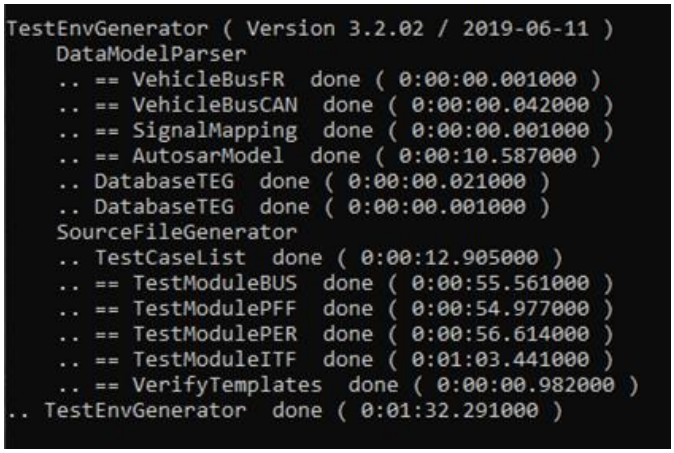
4.4. Rezultati mjerenja brzine unaprjeđenog rješenja generatora testnog okruženja

Mjerenja brzine izvođenja unaprjeđenog TEG-a podijeljena su u dvije smislene cjeline:

- mjerenje brzine izvođenja TEG-a uz primjenu *multiprocessing* modula i *hash* niza
- mjerenje brzine izvođenja TEG-a uz korištenje SQLite baze podataka te mjerenje utjecaja različitih postavki SQLite baze podataka na brzinu izvođenja unaprjeđenog TEG-a.

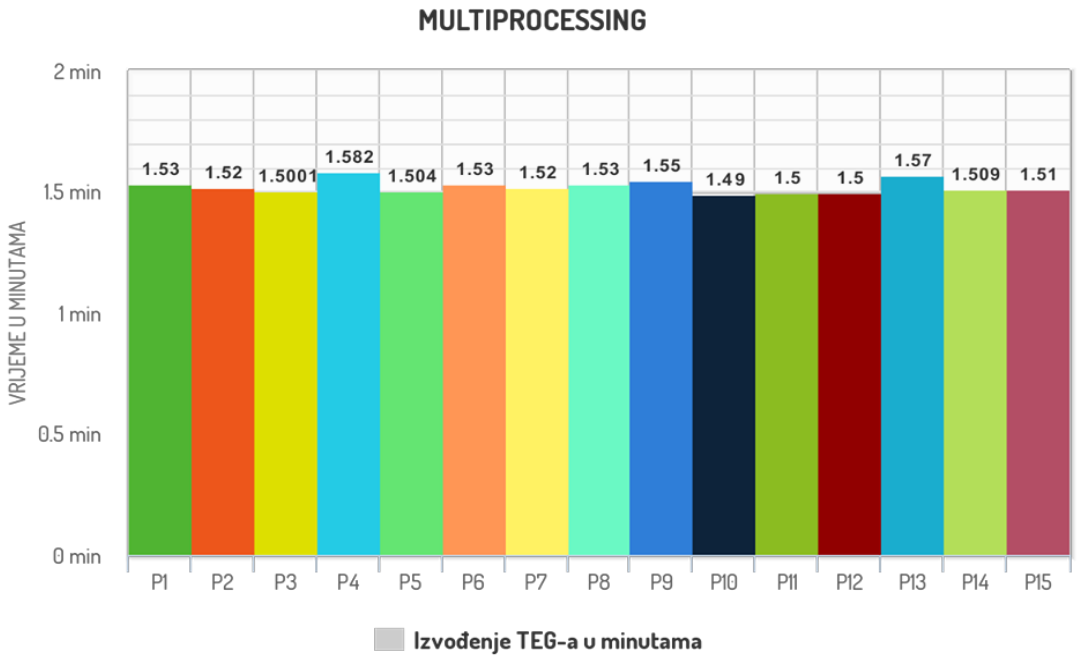
4.4.1. Mjerenja brzine s unaprjeđenom arhitekturom

Sljedeća mjerenja obavljena su prilikom izrade unaprjeđenog generatora testnog okruženja i predstavljaju postojeće rješenje s promijenjenom arhitekturom te originalnim *Pickle* rješenjem za pohranu podataka. Prva mjerenja, prikazana na sljedeće dvije slike, odnose se na izvođenje generatora testnog okruženja s uvedenim standardnim *multiprocessing* Python modulom nad generatorskom komponentom. Na slici 4.4.***4.4.*** prikazan je jedan primjer izvođenja TEG-a s *multiprocessing* modulom. Sam način izvođenja ispisa unutar Windows naredbenog retka je nepromijenjen, ali vidljive su iznimne razlike u vremenima izvođenja TEG-ova. Vrijeme izvođenja na primjeru je približno jedna minuta i trideset sekundi naspram dosadašnjih šest sati (slika 4.3.***4.3.***). Kao što je vidljivo iz usporedbe, dio TEG-a koji generira podatke znatno je skraćen zbog upotrebe *multiprocessing* modula (u *SourceFileGenerator* dijelu).



Slika 4.4. *Primjer izvođenja TEG-a s multiprocessing modulom.*

Na slici 4.5. prikazan je graf vremena izvođenja TEG-a s *multiprocessing* modulom. Iz grafa je vidljivo da je prosječno trajanje izvođenja TEG-a približno jednu minutu i 31 sekundu, odnosno približno 180 puta manje od postojećeg rješenja TEG-a.



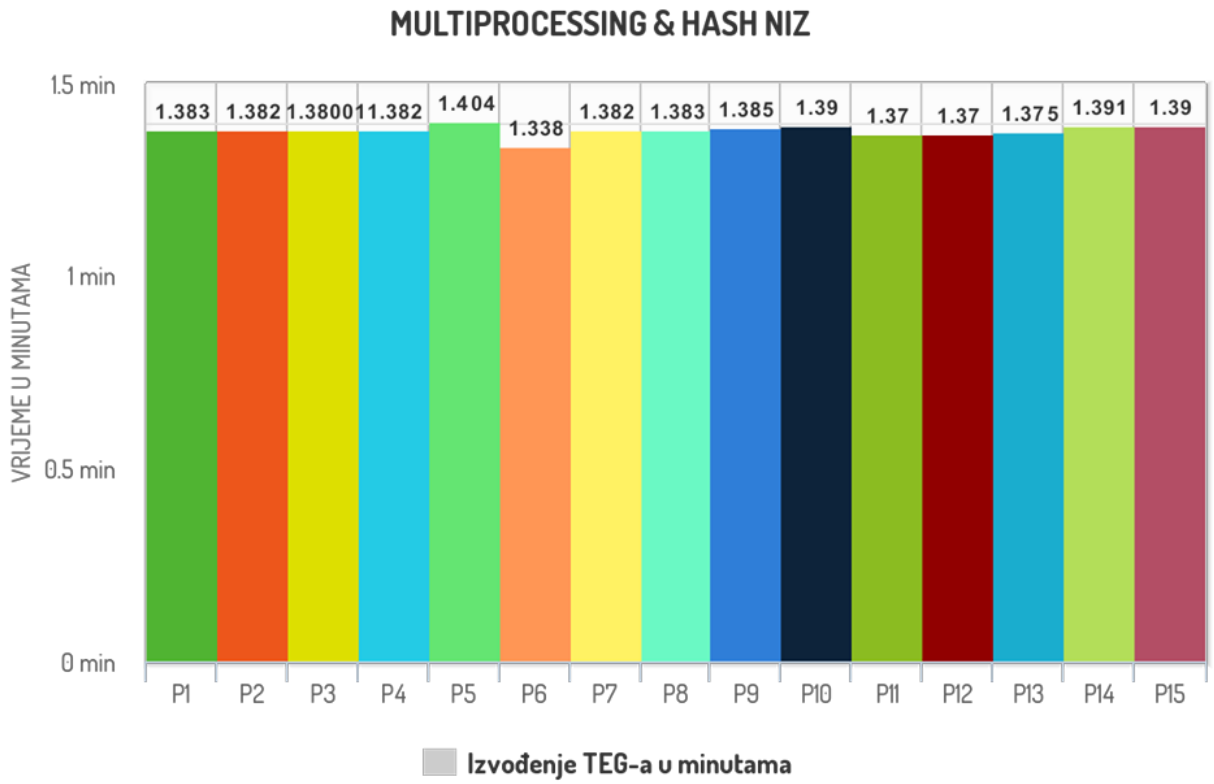
Slika 4.5. *Vrijeme izvođenja TEG-a s multiprocessing modulom.*

Zatim, provedena su mjerenja rada generatora testnog okruženja s usvojenim *multiprocessing* modulom, ali uz aktivirane preduvjete sigurnosnog *hash* niza koji programu omogućuje da preskoči izvođenje primarne komponente – parsera. Na slici 4.6. ***4.6.*** prikazan je jedan primjer izvođenja TEG-a s *multiprocessing* modulom i uspješno prepoznatim *hash* nizom. Iz slike je vidljivo da je uvođenjem *hash* niza vrijeme rada TEG-a skraćeno za dodatnih 10%.

```
TestEnvGenerator ( Version 3.2.02 / 2019-06-11 )
    DataModelParser
    .. == VehicleBusFR  done ( 0:00:00.005000 )
    .. == VehicleBusCAN  done ( 0:00:00.020000 )
    .. == SignalMapping  done ( 0:00:00 )
    .. == AutosarModel  done ( 0:00:02.322000 )
    .. DatabaseTEG  done ( 0:00:00 )
    .. DatabaseTEG  done ( 0:00:00 )
    SourceFileGenerator
    .. TestCaseList  done ( 0:00:11.889000 )
    .. == TestModuleBUS  done ( 0:00:54.284000 )
    .. == TestModulePFF  done ( 0:00:55.683000 )
    .. == TestModulePER  done ( 0:00:58.801000 )
    .. == TestModuleITF  done ( 0:01:05.003000 )
    .. == VerifyTemplates  done ( 0:00:00.810000 )
.. TestEnvGenerator  done ( 0:01:23.493000 )
```

Slika 4.6. *Primjer izvođenja TEG-a s multiprocessing modulom i uspješno prepoznatim* hash *nizom.*

Na slici 4.7.***4.7.4.7.4.7.*** prikazana su vremena izvođenja TEG-a s *multiprocessing* modulom i uspješno prepoznatim *hash* nizom. Vidljivo je da s implementiranim *hash* nizom TEG uspio preskočiti parserski dio izvođenja i stoga prosječno vrijeme izvođenja iznosi jednu minutu i 22 sekunde.



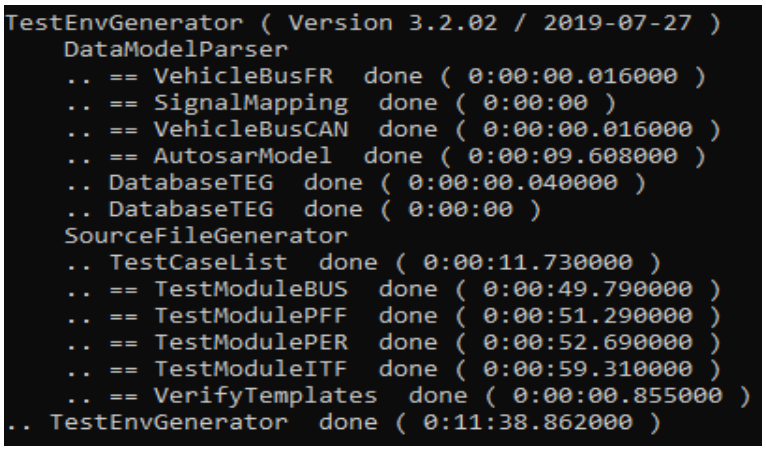
Slika 4.7. *Vrijeme izvođenja TEG-a s multiprocessing modulom i uspješno prepoznatim hash nizom.*

4.4.2. Mjerenje permutacija u postavkama SQL baze podataka

Pri izradi SQLite baza podataka postoje određene mjere sigurnosti koje se mogu poduzeti da bi se utjecalo na brzinu, ali i kvalitetu izvođenja pisanja ili čitanja iz baze. Unutar SQLite Python modula na te mjere se može utjecati koristeći se jedinstvenim izrazom *PRAGMA*. *PRAGMA* se poziva kao ključna riječ iza koje slijedi postavka baze podataka na koju se želi utjecati te nova vrijednost (naspram standardne) postavke. Na sljedećim mjerenjima postoje tri permutacije:

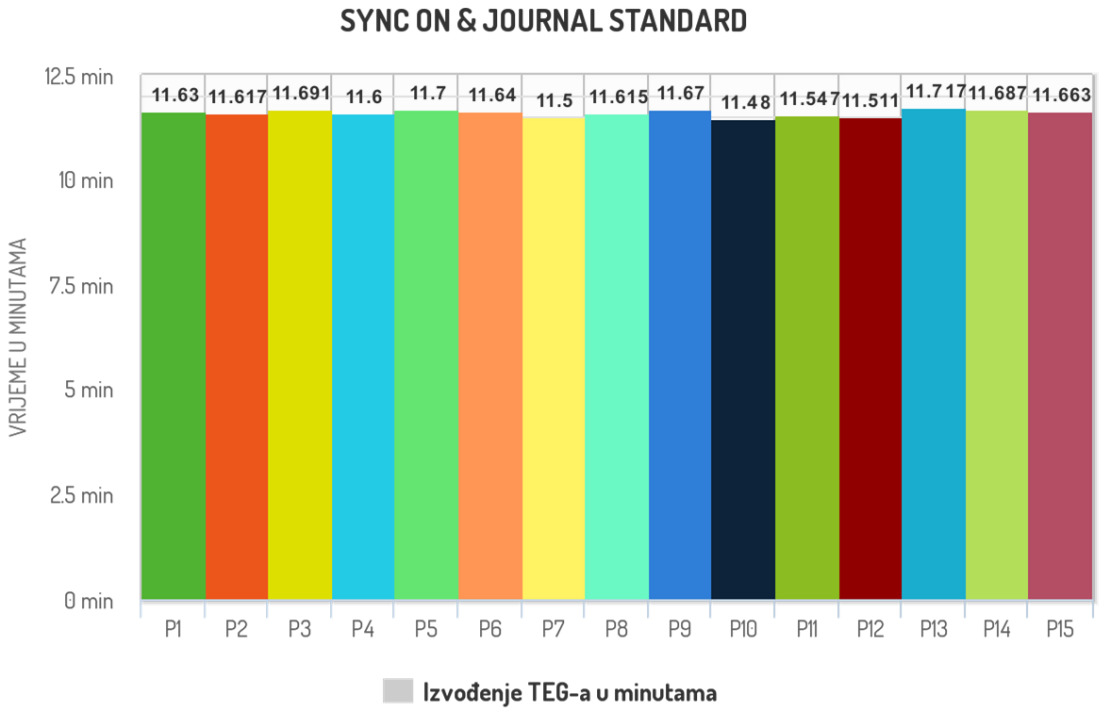
- standardne postavke SQL baze podataka
- sinkrono pisanje upita u bazu (*SYNCHRONOUS*)
- vođenje dnevnika zapisa u bazu (*JOURNAL_MODE*).

Prva mjerenja izvedena su uz standardne postavke gdje su, među ostalima, aktivirani sinkrono pisanje i dnevnik zapisa. Na slici 4.8. prikazan je primjer izvođenja TEG-a s usvojenim arhitekturalnim unaprjeđenjima i implementiranom SQLite bazom podataka putem SQLite Python modula. Vidljivo je da potpuno neoptimizirana SQLite baza sa standardnim postavkama poveća izvedbu pohrane podataka TEG-a na jedanaest, gotovo dvanaest minuta naspram postojećeg rješenja, odnosno serijalizacije *Pickleom* čija je izvedba trajala samo par sekundi, usporedno s generatorskim dijelom programa koji je trajao više sati.



Slika 4.8. *Primjer izvođenja TEG-a s multiprocessing modulom i implementiranom SQL bazom podataka sa standardnim postavkama.*

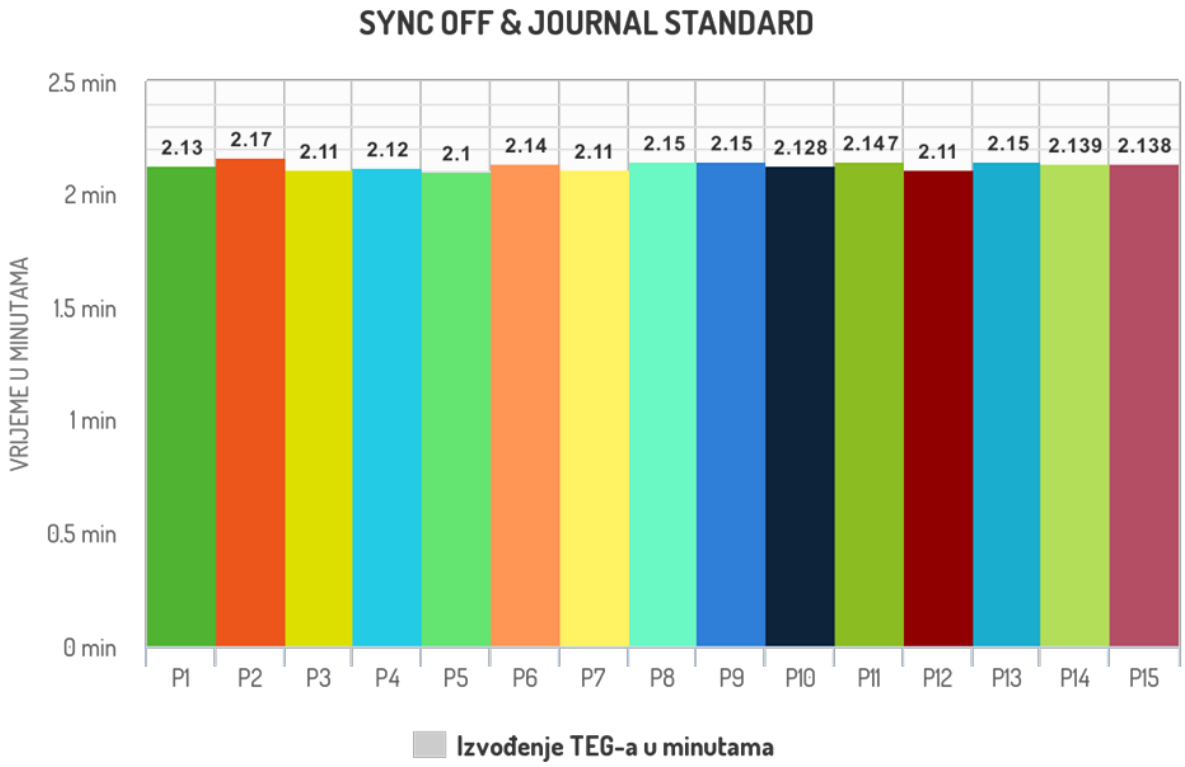
Na slici 4.9. prikazan je graf vremena izvođenja TEG-a s usvojenim arhitekturalnim unaprjeđenjima i implementiranom SQLite bazom podataka. Iz grafa je vidljivo da prosječno vrijeme izvođenja iznosi jedanaest minuta i trideset šest sekundi, što je deset puta dulje nego postojeće rješenje TEG-a s *Pickle* modulom.



Slika 4.9. *Vrijeme izvođenja TEG-a s multiprocessing modulom i usvojenim arhitekturalnim unaprjeđenjima i implementiranom SQL bazom podataka.*

U sljedećem eksperimentu onemogućen je uvjet za sinkrono pisanje podataka u bazu podataka te je pisanje prepušteno operacijskom sustavu. Ova mjera znatno ubrzava broj upita po sekundi, ali dovodi stabilnost baze u pitanje jer je moguće da se dogodi greška u zapisu podataka. Doduše, greške u zapisima, odnosno negativne posljedice promjene ove mjere nisu primijećene pri testiranju, barem ne pri ovoj veličini testnih podataka.

Na slici 4.10. prikazan je graf vremena izvođenja TEG-a s usvojenim arhitekturalnim unaprjeđenjima i implementiranom SQL bazom podataka uz izmijenjenu *SYNCHRONOUS* postavku. Iz grafa se može izračunati prosječno vrijeme izvođenja TEG-a koje iznosi dvije minute i osam sekundi. Poduzeta mjera ubrzanja pisanja u bazu prepuštanjem kontrole operacijskom sustavu spustila je vrijeme izvođenja na ono usporedno izvedbi TEG-a s *Pickle* modulom. Također je i najstabilnija metoda izuzev standardne konfiguracije.

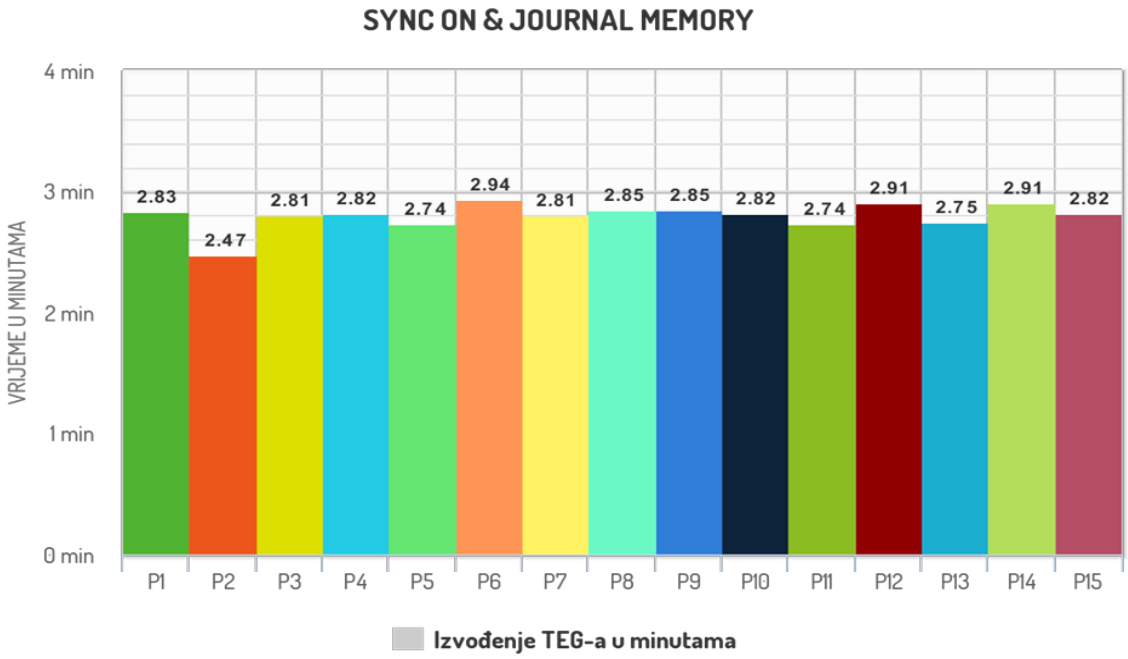


Slika 4.10. *Vrijeme izvođenja TEG-a uz izmijenjenu* SYNCHRONOUS *postavku na* OFF.

Naposljetku, izmijenjena je postavka za pisanje dnevnika zapisa u bazu. Ova postavka može poprimiti više vrijednosti. Standardna vrijednost je zapis dnevnika usporedno sa zapisom u bazu, u isti direktorij u obliku dokumenta. Vrijednost korištena u ovom testiranju je postavku dnevnika s *ON* promijenila na *MEMORY*. *MEMORY* postavka pomiče izvođenje i zapis dnevnika u radnu memoriju računala gdje se isti obriše pri završetku rada s bazom podataka.

Ova mjera također znatno povećava broj upita u sekundi, ali izmjena ove postavke utječe na sigurnost stvaranja i pisanja SQL baze podataka jer u slučaju kritičnog kvara računala (pad sustava, nestanak struje, ručni prekid programa i sl.) može doći do potpunog gubitka baze, odnosno podataka u njoj.

Na slici 4.11.***4.11.*** prikazan je graf vremena izvođenja TEG-a s usvojenim arhitekturalnim unaprjeđenjima i implementiranom SQL bazom podataka uz izmijenjenu *JOURNAL_MODE* postavku. Iz grafa se može izračunati prosječno vrijeme izvođenja TEG-a koje iznosi dvije minute i četrdeset osam sekundi. Poduzeta mjera držanja dnevnika zapisnika u bazi unutar radne memorije računala (bez spremanja na disk) dovodi do nešto sporije izvedbe od prošle, ali je ujedno i najsigurnija metoda izvan standardne.



Slika 4.11. *Vrijeme izvođenja TEG-a uz izmijenjenu JOURNAL_MODE postavku na MEMORY.*

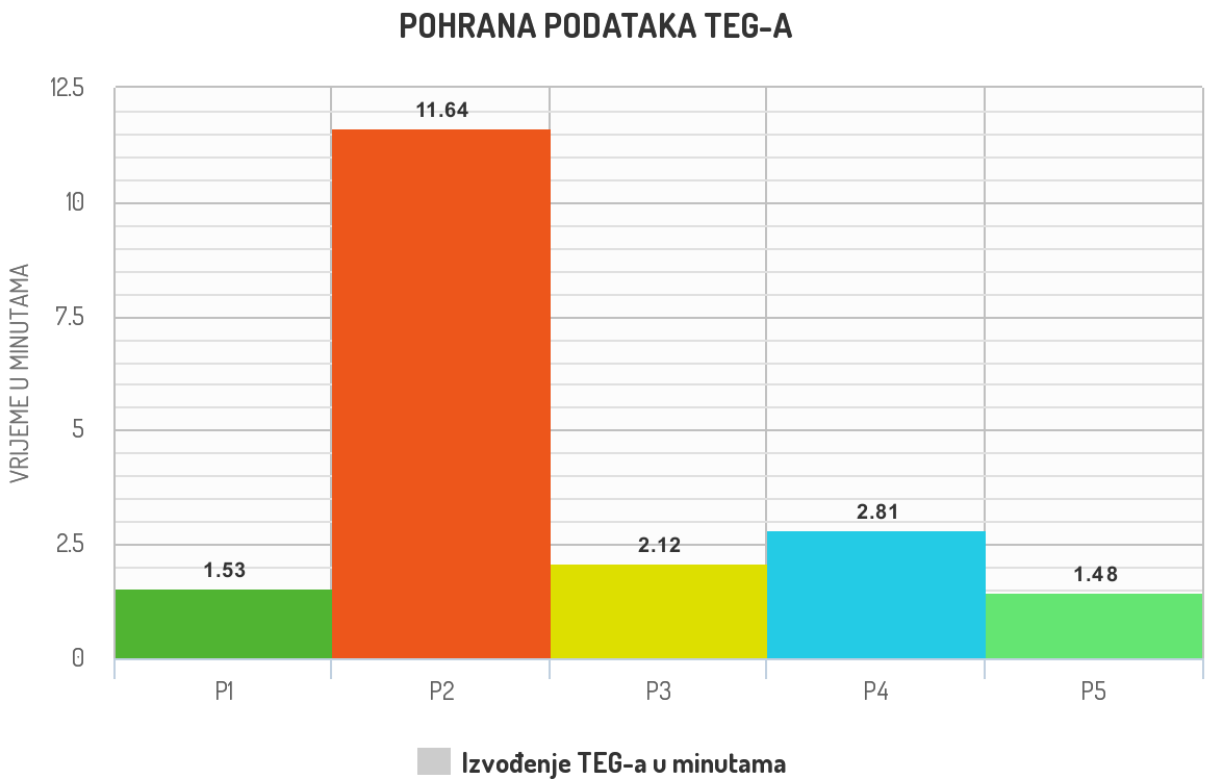
Postoji i četvrta mogućnost promjene obiju postavki na brže vrijednosti. Mjera bi *SYNCHRONOUS* postavku stavila na *OFF*, a *JOURNAL_MODE* na *MEMORY*. Na slici 4.12. prikazan je jedan primjer izvođenja generatora testnog okruženja s ovim mjerama. Budući da ovaj rezultat postiže najmanje vrijeme izvršavanja na danim datotekama, može se zaključiti se da ovaj rezultat najbolje parira *Pickle* verziji TEG-a, uz trajanje od jedne minute i dvadesetdevet sekundi, ali ova mjera se ne preporuča iz razloga znatnog gubitka sigurnosti i stabilnosti baze podataka.

```
TestEnvGenerator ( Version 3.2.02 / 2019-07-21 )
    DataModelParser
    .. == VehicleBusFR  done ( 0:00:00 )
    .. == SignalMapping  done ( 0:00:00.005000 )
    .. == VehicleBusCAN  done ( 0:00:00.030000 )
    .. == AutosarModel  done ( 0:00:10.455000 )
    .. DatabaseTEG  done ( 0:00:00.025000 )
    .. DatabaseTEG  done ( 0:00:00 )
    SourceFileGenerator
    .. TestCaseList  done ( 0:00:11.231000 )
    .. == TestModuleBUS  done ( 0:00:53.039000 )
    .. == TestModulePFF  done ( 0:00:53.432000 )
    .. == TestModulePER  done ( 0:00:55.168000 )
    .. == TestModuleITF  done ( 0:01:01.280000 )
    .. == VerifyTemplates  done ( 0:00:00.795000 )
.. TestEnvGenerator  done ( 0:01:29.015000 )
```

Slika 4.12. *Primjer izvođenja TEG-a uz izmijenjenu* JOURNAL_MODE *postavku na* MEMORY *i* SYNCHRONOUS *postavke na* OFF.

Pisanje u novu bazu podataka, tj. korištenje SQLite baze umjesto serijalizacije donijelo je slične rezultate kao i postojeće rješenje (točnije, da je nezamjetno kraj izvođenja ukupnog programa bez *multiprocessing* modula). Dakle, ne može se reći da je sam proces pohrane podataka ubrzan, ali, ispravno konfiguriran, brzinom parira postojećem rješenju, s dodatnim benefitom veće sigurnosti, stabilnosti i preglednosti baze. Osim toga, kako se baza sprema usporedno s radom generatora, koji traje znatno dulje, vremensko izvođenje baze ne utječe znatno na ukupno vremensko izvođenje TEG-a ako se SQLite baza optimalno konfigurira.

Slika 4.13. prikazuje usporedno sve mjere izvođenja zapisa, odnosno pohrane podataka u TEG-u i njihova prosječna vremena izvođenja. Primjer 1 (P1) predstavlja izvođenje postojeće verzije TEG-a s *Pickle* i *multiprocessing* modulima; primjer 2 (P2) predstavlja izvođenje TEG-a uz standardne postavke SQLite baze podataka; primjer 3 (P3) predstavlja izvođenje TEG-a uz postavke SQLite baze bez *SYNCHRONOUS* mjere; primjer 4 (P4) predstavlja izvođenje TEG-a uz postavke SQLite baze s *JOURNAL_MODE* mjerom postavljenom na RAM umjesto na disk; primjer 5 (P5) predstavlja izvođenje TEG-a uz postavke SQLite baze sa svim mjerama ubrzanja gore navedenim, odjednom. Gledajući usporedno sve performanse različitih mjera ubrzanja SQLite baze podataka, može se preporučiti opcija P4 odnosno promjena pisanja dnevnika pohrane u radnu memoriju računala, odnosno da mjera *SYNCHRONOUS* ostane uključena. Zbog iznimno kratke izvedbe unaprjeđenog TEG-a, gubitak vremena radi ponovne izvedbe je zanemariv naspram problema mogućih od neispravno zapisanih podataka koje druge mjere mogu donijeti.



Slika 4.13. *Usporedba svih mjera pohrane podataka TEG-a unaprjeđenog multiprocessing modulom.*

4.5. Buduća unaprjeđenja generatora testnog okruženja

Što se tiče prijedloga za moguća buduća unaprjeđenja TEG-a, predlaže se:

- redizajn ARXML dokumenata koji ulaze u TEG na manje *.arxml* datoteke gdje svaka predstavlja jednu SWC komponentu (ili barem više sličnih odjednom),
- automatski i nužan redizajn strukture TOM temeljnog objekta da bude obradiv na više procesa odjednom,
- uvođenje potrebnih atributa i kôda u strukturu TOM temeljnog objekta koje bi omogućilo robusniji i efikasniji prijenos podataka i vrijednosti iz TOM-a u SQL bazu podataka,
- uz to i uvođenje dodatnog modula za rad s objektno-relacijskim mapiranjima objekata u bazu podataka u Pythonu (engl. ***O****bject* ***R****elational* ***M****apping*, *ORM*), gdje se preporučuje korištenje modula poput *SQLAlchemy ORM*-a,
- moguć prelazak potpunog TEG-a na C programski jezik gdje se očekuju još manja vremena izvođenja.

5. ZAKLJUČAK

Kako bi ADAS sustavi bili u potpunosti sigurni, vrlo ih je važno detaljno testirati. Testiranje se odrađuje na više načina, kako softverski, tako i hardverski. Jedno od važnih testiranja predstavlja provjeru komunikacije između ECU-ova u automobilu koristeći simulacije, odnosno testna okruženja. U svrhu testiranja *middleware* sloja između hardvera i softvera računala automobila koristi se generator testnog okruženja, odnosno TEG. Korištenjem različitih konfiguracija TEG-a moguće je izvesti višestruka testiranja i simulacije i time kvalitetno provjeriti rad ADAS sustava. TEG uzima podatke o ECU-ovima koji komuniciraju i unutar svojeg testnog okruženja generira simulaciju komunikacije kojom provjerava ispravnost rada ADAS sustava.

Raspoloživi generator testnog okruženja ima tri glavne komponente: parser, pohranu podataka i generator. Zadatak ovog diplomskog rada bio je unaprijediti svaku pojedinu komponentu. Unaprjeđenje je moglo biti i algoritamske i arhitekturne prirode, s glavnim ciljem skraćivanja ukupnog vremena izvođenja TEG-a i uvođenjem stabilnije i sigurnije pohrane podataka. Temeljna komponenta TEG-a složeni je Python objekt (TOM) koji je dizajniran da svojom strukturom prati strukturu ARXML datoteka koje se u njega parsiraju. TOM se predaje kao primarni argument svakoj glavnoj funkciji koja predstavlja tri glavne komponente TEG-a i stoga je redizajn istog bio nemoguć bez duljeg pristupa postojećem rješenju i vjerojatno izvan opsega jednog diplomskog rada. Rad postojećeg parsera bio je vezan uz petlju koja prolazi kroz ARXML i jednako tako inicijalizira i piše u TOM. Pohrana podataka bila je odrađena dvostruko: zasebnom XML datotekom radi uvida u podatke i *Pickle* serijalizacijom. Generator je također iterativno prolazio kroz predloške *.c* kôda i smještao ih redom u SWC komponente.

Ubrzanje rada parsera odrađeno je zaobilaženjem rada parsera ako nije potreban putem *hash* niza koji uspoređuje strukturu novog TEG-a sa starim. Promijenjen je način pohrane podataka sa serijalizacije *Pickle* modulom na relacijsku bazu podataka SQLite modulom. Time ubrzanje pohrane podataka nije postignuto, ali može se smatrati usporednim s postojećim rješenjem. Dodatni benefiti u obliku stabilnije i sigurnije baze koja je čitljiva i pregledljiva čine bitnu razliku, pogotovo jer se baza sprema usporedno sa znatno dugotrajnijim radom generatora tako da nema utjecaja na ukupno vremensko izvođenje TEG-a. Rad generatora značajno je ubrzan uvođenjem paralelnog obrađivanja pojedinih SWC komponenata, s prvotno generirane liste SWC-ova, podijeljene na zasebne računalne procese. Na temelju svih mjerenja i primjera, jasno je vidljivo da su ciljevi zadani na početku ovog diplomskog rada uspješno izvršeni budući da se vrijeme izvođenja skratilo 180 puta.

LITERATURA

[1] Shaout A., Colella, D. Awad, S., Advanced Driver Assistance Systems – Past, Present and Future, The University of Michigan Dearborn, 2011.

[2] AutoSens, https://auto-sens.com/ (pristupljeno 27.7.2019.)

[3] AUTOSAR, Layered Software Architecture, https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf (pristupljeno 5.8.2019.)

[4] Automotive Wiki, AUTOSAR XML Schema, https://automotive.wiki/index.php/AUTOSAR_XML_Schema (pristupljeno 24.7.2019.)

[5] lxml, Processing XML and HTML with Python, https://lxml.de/index.html#introduction (pristupljeno 26.7.2019.)

[6] lxml, Benchmarks and Speed, https://lxml.de/performance.html (pristupljeno 26.7.2019.)

[7] Rushter, How pickle works in Python, https://rushter.com/blog/pickle-serialization-internals/ (pristupljeno 27.7.2019.)

[8] Ben Fredrickson, Don't Pickle Your Data, https://www.benfrederickson.com/dont-pickle-your-data/ (pristupljeno 26.7.2019.)

[9] Watt, A., Beginning Regular Expressions, Wiley Publiching, Inc., 2005.

[10] SQLite, About, https://www.sqlite.org/about.html (pristupljeno 26.7.2019.)

[11] Owens, M.: The Definitive Guide to SQLite, Chapter 4: SQL, Apress, 2006.

[12] Python Software Foundation, multiprocessing - Process-based "threading" interface, https://docs.python.org/2/library/multiprocessing.html (26.7.2019.)

SAŽETAK

Moderni automobili koriste napredne elektroničke sustave koji pomažu vozaču u prometu – tzv. ADAS sustavi (engl. ***A**dvanced **D**river-**A**ssistance **S**ystems*). ADAS sustavi koriste se da bi automatizirali, prilagodili i poboljšali sustave unutar vozila u svrhu veće sigurnosti i kvalitetnijeg iskustva vožnje automobila na cesti. Ključan faktor u njihovom radu komunikacija je između pojedinih komponenata sustava koju je nužno testirati, što se najbolje odrađuje izradom testnih kôdova za provjeru modela usmjerenih na AUTOSAR (engl. ***AUT**omotive **O**pen **S**ystem **AR**chitecture*) komunikaciju. U ovom diplomskom radu unaprjeđivan je jedan od takvih sustava testiranja ADAS okruženja koji generira testno okruženje za simulaciju komunikacije u srednjem sloju (engl. *Middleware*) arhitekture AUTOSAR-a. Generator testnog okruženja (engl. ***T**est **E**nvironment **G**enerator,* TEG) Python je program za obradu testnih ARXML datoteka na temelju kojih generira model testnog okruženja u obliku zasebnih komponenti u C programskom jeziku. Program se sastoji od parsera ulaznih podataka, pohrane parsiranih podataka i generatora. U okviru ovog diplomskog rada postojeći TEG je unaprijeđen u vidu brzine vremena izvođenja i načina pohrane podataka. Vrijeme izvođenja programa ubrzano je uvođenjem višeprocesorskog načina rada u sve komponente programa i procedurama za izbjegavanje redundantnih operacija unutar programa. Način pohrane podataka postojećeg rješenja serijalizacijom zamijenjen je relacijskom bazom podataka u SQL upitnom jeziku, rezultirajući stabilnijim i robusnijim načinom pohrane podataka. Testiranjem predloženog rješenja postignuto je iznimno skraćivanje ukupnog vremena izvođenja postojećeg programa sa par sati na nekoliko minuta nad istim podacima, a komponenta pohrane podataka svojim izvođenjem parira onom postojećeg rješenja – uz dodanu sigurnost i preglednost podataka.

Ključne riječi: ADAS, AUTOSAR, ARXML, SQL, višestruko procesiranje, generator kôda, testno okruženje

ABSTRACT

Code generator for ADAS systems testing

Modern cars use advanced electronic systems that help the driver in traffic - so-called ADAS systems (**A**dvanced **D**river-**A**ssistance **S**ystems). ADAS systems are used to automate, customize and improve systems within a vehicle for greater safety and better road driving experience. The key factor in their work is communication between individual system components which is necessary to test. This is best done by developing AUTOSAR (**O**pen **S**ystem **AR**chitecture) communication tests. In this Master's thesis, one of these ADAS environment testing systems is being developed, which generates a test environment for the simulation of communication in the middle layer (*Middleware*) of AUTOSAR architecture. Test Environment Generator (TEG) is a Python program for processing ARXML test files based on which it generates a test environment model in the form of separate components in the C programming language. The program consists of a data parser, parsed data storage and code generator. Within this Master's thesis, the existing TEG has been improved in terms of speed of execution and manner of data storage. Program runtime is accelerated with the introduction of multiprocessing into all program components and by using special procedures to avoid redundant operations within the program. The storage method of the existing solution by serialization has been replaced with a relational database in SQL query language, resulting in a more stable and robust data storage method. The testing of the proposed solution showed it achieved an exceptional reduction of the total execution time of the existing program from a few hours to a few minutes over the same data, and the data storage component, by its execution, pairs with the existing solution - with the added security and transparency of the data.

Keywords: ADAS, AUTOSAR, ARXML, SQL, multiprocessing, code generator, test environment

ŽIVOTOPIS

Andrija Mihalj rođen je 1. kolovoza 1994. Godine u Osijeku, Hrvatska. Pohađa Osnovnu školu Josipa Antuna Ćolnića u Đakovu do 2009. godine te iste godine upisuje opći smjer Gimnazije A. G. Matoša u Đakovu. Srednju školu završava s vrlo dobrim uspjehom. Po završetku srednje škole, 2013. godine, upisuje tadašnji Elektrotehnički fakultet Osijek, danas Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek, smjer računarstvo. Preddiplomski studij završava 2017. godine te na istom fakultetu upisuje diplomski studij računarstva, smjer Procesno računarstvo.

Izvrsno se služi engleskim jezikom te ima srednju razinu znanja s Microsoft Office alatima. Ima visoku razinu znanja s programskim jezicima C i Python, srednju razinu znanja s SQL-om kao i srednju razinu znanja s jezicima PHP i JavaScript te opisnim jezikom HTML i stilskim jezikom CSS.

Dobitnik je stipendije od strane privatnih poduzetnika za potporu obrazovanja studenata na elektrotehničkim smjerovima za sve godine studiranja. Radno iskustvo ima kao radnik u skladištu za HEMCO d.o.o. u Đakovu te kao agent u teleprodajnom centru za Hrvatski Telekom d.d. u Osijeku u trajanju od dvije godine. Također, akademske godine 2018./19. dobitnik je stipendije RT-RK iz Osijeka te se trenutno u istoj tvrtci obučava za daljnji rad nakon diplome.

Andrija Mihalj

_____