Razvoj generatora testnog okruženja s fokusom na upravljanje bazom podataka

Kožul, Ivan

Master's thesis / Diplomski rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:666041

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-11-29**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

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

Sveučilišni studij

RAZVOJ GENERATORA TESTNOG OKRUŽENJA S FOKUSOM NA UPRAVLJANJE BAZOM PODATAKA

Diplomski rad

Ivan Kožul

Osijek, 2021.

SADRŽAJ

1. UVOD

Moderni automobili iz dana u dan dobivaju nove funkcionalnosti koje vozačima omogućuju jednostavniju i sigurniju vožnju, no prije negoli bi mogli izaći na tržište, potrebno je osigurati visoku kakvoću sklopovlja od kojih se sastoje i sustava kojima se koriste. Broj komunikacijskih kanala unutar sklopovlja automobila može prijeći brojku od više desetaka tisuća. Ispravna komunikacija svakim od tih kanala mora biti precizno testirana prije negoli se isti mogu primjenjivati u daljnjoj proizvodnji. S obzirom na to da je broj potrebnih testova vrlo velik, jedino smisleno rješenje je automatizirati proces testiranja u što je moguće većoj mjeri. Jedan od prihvatljivih načina testiranja je simuliranje testnog okruženja koje omogućuje testiranje komunikacije pomoću virtualnih komponenti, a ne fizičkih. Takav način testiranja provodi se u programskom alatu nazvanom generator testnog okruženja, a jedan takav alat razvila je austrijska tvrtka TTTech te ga ustupila prilikom izrade ovog diplomskog rada. Iako omogućava ispravno testiranje komunikacijskih kanala između virtualnih komponenti, alat u sebi sadržava nedostatke koje bi trebalo ukloniti, kako bi pouzdanost samog alata postala veća. U okviru ovog diplomskog rada potrebno je razviti dio osnovnog generatora testnog okruženja koji je zadužen za generiranje glavne baze podataka iz koje će generatori generirati izvorni kod. Potrebno je upoznati se s implementacijom generatora testnog okruženja kojeg je razvila tvrtka TTTech, pronaći njezine nedostatke i napraviti plan implementacije novog rješenja. Rješenje koje je razvila tvrtka TTTech implementirano je u programskom jeziku Python, a novo je rješenje potrebno implementirati u programskom jeziku C++. Potrebno je analizirati njegove performanse te ih usporediti s onima postojećeg rješenja. Nakon što su svi zahtjevi novog rješenja ispunjeni, potrebno je provesti testove koji potvrđuju ispravno funkcioniranje novostvorenog rješenja.

Diplomski je rad podijeljen u pet poglavlja. U drugom je poglavlju opisan način rada postojećeg rješenja generatora testnog okruženja te su opisani nedostatci koje novo rješenje ne bi trebalo imati. U istom je poglavlju također opisano još jedno rješenje čija je svrha testiranje AUTOSAR modela. U trećem je poglavlju opisano rješenje napravljeno u sklopu ovog diplomskog rada, koraci njegove izrada te alati korišteni za njegovu izradu. U četvrtom su poglavlju opisani rezultati testiranja provedenog rješenjem predloženim u trećem poglavlju te je napravljena usporedba performansi predloženog i postojećeg rješenja. U petom su poglavlju izneseni zaključci diplomskog rada

2. PREGLED POSTOJEĆEG RJEŠENJA GENERATORA TESTNOG OKRUŽENJA

U ovom je poglavlju ukratko opisano područje kojim se ovaj diplomski rad bavi. Predstavljeno je rješenje za problem testiranja sklopovlja u automobilskoj industriji koje je slično generatoru testnog okruženja kojeg je razvila tvrtka TTTech. Detaljnije je opisano rješenje generatora testnog okruženja kojeg je razvila tvrtka TTTech i koje je ujedno i dobiveno na proučavanje u sklopu ovog diplomskog rada. Opisani su nedostatci postojećeg rješenja koje bi novo rješenje trebalo ukloniti te je predstavljeno slično rješenje za problem testiranja sklopovlja u automobilskoj industriji.

2.1. Testiranje komunikacije između komponenti automobila

Moderni se automobili služe raznim senzorima poput kamera ili lidara kako bi percipirali okolinu oko sebe. Nakon što senzori prikupe informacije o okolini automobila, te se informacije prosljeđuju procesorima na obradu. Na temelju obrađenih informacija, procesori određuju koja će biti iduća radnja aktuatora, primjerice ako prednja kamera automobila detektira automobil koji se kreće sporije, procesori će putem aktuatora prilagoditi brzinu automobila da ne bi došlo do mogućeg sudara. S obzirom na to da se automobili u prometu kreću dovoljno velikom brzinom u kojoj bi prilikom pogrešne radnje aktuatora moglo doći do ugrožavanja ljudskih života, vrlo je važno da sustavi za pomoć pri vožnji rade ispravno, a njihov ispravan rad moguće je utvrditi provođenjem testiranja prije nego li nastupe u realanom scenariju u samoj upotrebi. Ručno pisanje testova dugotrajan je posao koji zahtijeva veliku količinu resursa, stoga je cilj automatizirati proces testiranja u što je moguće većoj mjeri. Ovaj se diplomski rad bavi jednim od načina testiranja komunikacije između komponenti automobila.

2.2. Rješenje za testiranje komunikacije između komponenti automobila na osnovu modela

Koristeći primjer iz standarda AUTOSAR za automobile, predstavljeno je programsko rješenje *„QuickCheck“* za testiranje na temelju modela [1]. Generiranjem testova iz modela, umjesto da se ručno pišu, može se pokriti proizvoljan skup scenarija ispitivanja. Kontroliranjem generatora testnih podataka može se pokrenuti proces prema određenom cilju testiranja. Slučajnost generiranja testa donosi dodatnu prednost za stvaranje tipičnih ispitnih sekvenci koje inženjer za ispitivanje

neće uzeti u obzir. Dodana vrijednost testiranja zasnovanog na modelu je ta što je moguće proizvoljno povećati broj testova koji trebaju pokriti nova ponašanja testiranog sustava uz vrlo malo dodatnog napora. Pri testiranju izmjene modela, promjena nekoliko redaka u modelu pokazala je skrivenu grešku, dok je mehanizam skupljanja *QuickChecka* pružio minimalni primjer za grešku bez dodatnih troškova razvoja, u korist dodatnog vremena rada za ispuhivanje (engl. *exhalation*) testnih podataka. Neizostavna značajka *QuickChecka* je fleksibilan mehanizam ismijavanja (engl. *mocking mechanism*) programibilnog sučelja aplikacije (engl. *Application Programming Interface* - API). Osim opsežnih softverskih apstrakcija hardverskih sučelja, što je ključno za *offline* testiranje ugrađenih softvera, pokazalo se kako ga je moguće koristiti za testiranje otpornosti na greške cijelog sustava. Ovo je postignuto ubrizgavanjem grešaka u makete komponenti, na temelju kojih se sustav testira. To omogućuje testiranje u odnosu na mnoge različite (neispravne ili ispravne) implementacije komponenti, koje u stvari ne moraju postojati, pod uvjetom da je dostupan konfigurabilan model komponente.

2.3. Opis postojećeg generatora testnog okruženja

Generator testnog okruženja (engl. *Test Environment Generator* - TEG) programski je alat čija je glavna zadaća provjera ispravnosti komunikacije između elektroničkih upravljačkih jedinica (engl. *Electronic Control Unit* - ECU) u automobilu [2]. Glavni zadatak TEG-a moguće je podijeliti u tri veća pod-zadatka, odnosno tri koraka izvršavanja programa:

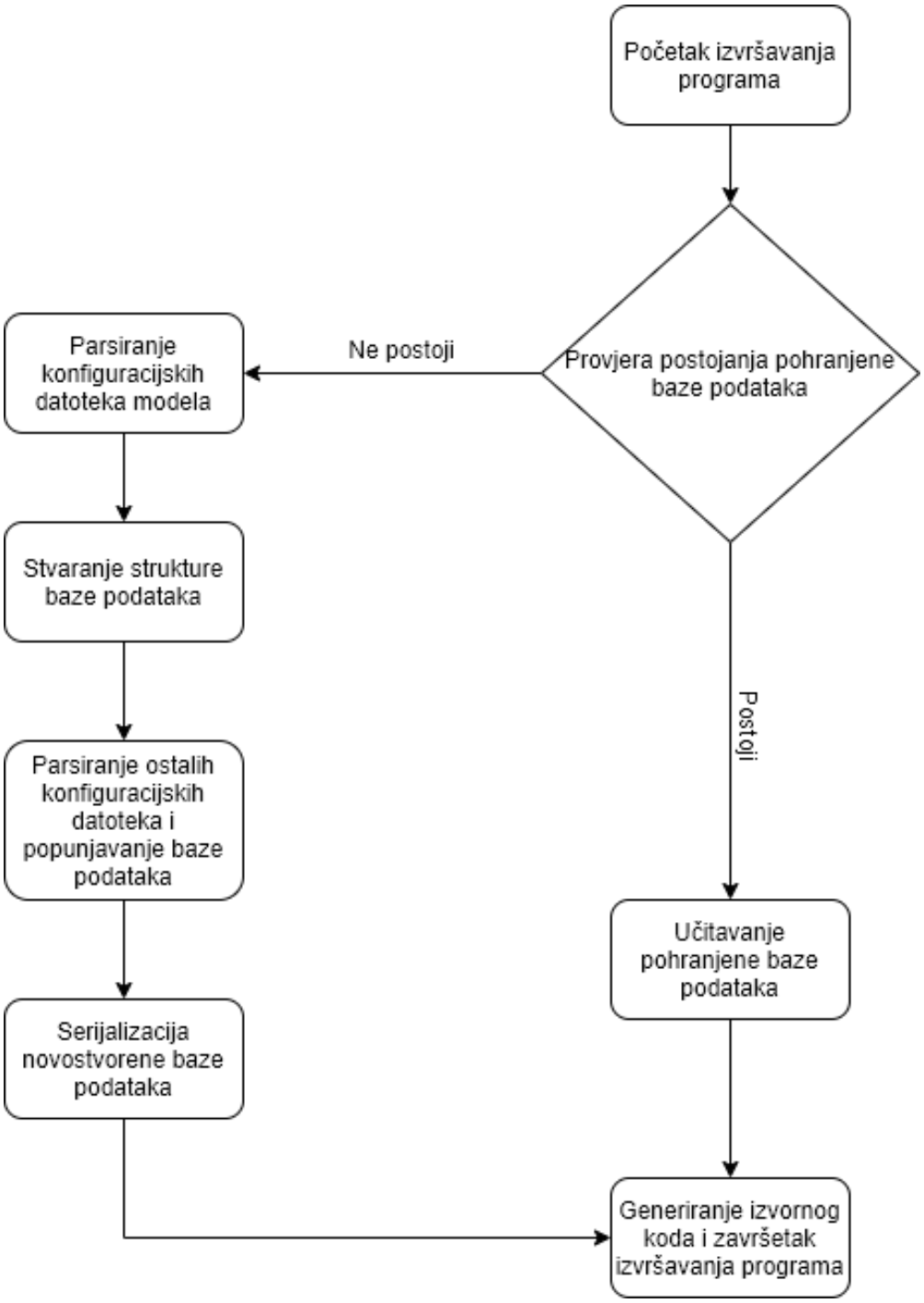
1. Parsiranje ulaznih podataka;
2. Pohrana podataka u bazu podataka;
3. Generiranje izvornog koda;

Više informacija o svakom koraku izvođenja dano je u sljedećim dijelovima rada.

2.3.1. Parsiranje ulaznih podataka

Prvi korak uključuje izdvajanje i parsiranje (raščlanjivanje) konfiguracijskih datoteka koje su propisane AUTOSAR (*engl. AUTomotive Open System ARchitecture*) standardom [3]. Datoteke su uglavnom *.arxml* (AUTOSAR XML), *.xml* i *.dbc* podatkovnog tipa. ARXML je prilagođena vrsta XML-a (*engl. Extensible Markup Language*) koja u sebi sadrži podatke o konfiguraciji i specifikacijama ECU-a [4]. Svrha parsiranja je izdvajanje konfiguracijskih podataka specifičnog ECU-a na osnovu kojih se generira model koji predstavlja strukturu baze podataka. Tu bazu

podataka čine komponente i njihove vrijednosti koje opisuju komunikaciju s ostalim komponentama sustava. Nakon što se strukturirao model, baza podataka se nadopunjava vrijednostima koje se dobiju iz ostalih datoteka koje opisuju već spomenutu komunikaciju između komponenti sustava. Na slici 2.1 nalazi se dijagram toka izvršavanja postojećeg rješenja TEG-a.



Slika 2.1. Dijagram toka izvršavanja postojećeg rješenja TEG-a

2.3.2. Pohrana podataka u bazu podataka

U drugom se koraku parsirani podatci skupljaju u smislene cjeline, a zatim se pohranjuju u spremnik (baza podataka). Pohranjeni su podatci nužni generatorima za daljnji rad TEG-a. Nakon što su svi podatci dobiveni parsiranjem pohranjeni, popunjena se baza podataka pohranjuje u zasebnu datoteku kako bi ju se moglo koristiti prilikom ponovnog pokretanja TEG-a, odnosno kako bi se prilikom ponovnog pokretanja TEG-a mogao preskočiti korak parsiranja jer je vremenski najzahtjevniji. Baza podataka pohranjuje se u *.pickle* formatu koji nije razumljiv čovjeku. Više o *pickle*-u može se pročitati u dijelu 2.4.2.

2.3.3. Generiranje izvornog koda

Treći, posljednji korak, izvršava se nakon što je formirano, odnosno popunjeno skladište sa svim potrebnim podatcima, te generatori analiziraju pohranjene podatke, a zatim ih koriste kako bi generirali izvorni C kod. Generirani se izvorni kod koristi kako bi se provjerio model AUTOSAR komunikacije na ECU-ovima.

2.4. Nedostatci postojećeg rješenja TEG-a

Postojeće rješenje TEG-a pisano je u programskom jeziku Python 2 (2 označava verziju jezika) [5]. Python se smatra skriptnim jezikom zbog povijesnog zamućenja između skriptnih jezika i programskih jezika opće namjene. U stvarnosti, Python nije skriptni jezik, već programski jezik opće namjene koji također dobro funkcionira kao skriptni jezik. Iako postojeće rješenje uspješno izvršava sve propisane zahtjeve TEG-a, postoji nekoliko nedostataka koji sprječavaju trenutno rješenje TEG-a da postane kompatibilno programsko rješenje za izlazak na tržište.

2.4.1. Nedostatci Pythona 2

Iako Python sadržava mnoge zgodne značajke koje uvelike pomažu programerima pri pisanju izvornog koda te je vrlo čitljiv za razliku od nekih programskih jezika niže razine, rješenje napisano u Pythonu nažalost dolazi s netolerantnim nedostatcima. Kao što je već navedeno, postojeće rješenje napisano je u Python verziji 2, a glavni problem verzije 2 Pythona je prestanak izdavanja daljnje podrške i novih ažuriranja samog jezika [6]. Razlog prestanka ažuriranja Pythona 2 je postojanje njegovog nasljednika, Pythona 3, kojeg Python 2 nije mogao popratiti radi brojnih

novina koje je donio Python 3, a koje nije bilo moguće ugraditi u Python 2. Iako je Python 3 izdan 2006. godine, izdavači su i dalje ažurirali Python 2 radi zahtjeva korisnika koji su se njime i dalje služili, no tome je došao kraj 1. siječnja 2020. godine.

Osim završetka podrške za Python 2, općeniti problem je niska brzina izvršavanja programa pisanog u Pythonu u odnosu na program pisan u C++-u [7]. Više detalja o usporedbi performansi TEG-a pisanog u Pythonu i TEG-a pisanog u C++-u nalazi se u 4. poglavlju ovog rada. Još jedan od nedostataka koji se tiču Pythona je nemogućnost sigurnog očuvanja izvornog koda TEG-a [8]. Iako su razvijene razne metode „skrivanja“ izvornog koda u Pythonu, niti jedna nije na dovoljno visokoj razini sigurnosti da program bude dovoljno zaštićen za plasiranje na tržište.

2.4.2. Problemi s bazom podataka

Jedan od glavnih nedostataka postojećeg rješenja TEG-a je metoda pohrane podataka, a to je korištenje Pythonove biblioteke *pickle* [9]. Biblioteka *pickle* Pythonov je modul koji rješava problem serijalizacije i deserijalizacije raznih tipova podataka i objekata kako bi bili spremni za ponovno korištenje. Proces serijalizacije provodi se na način da se Python objekt pretvara u tok binarnih podataka te se pohranjuje u datoteku tipa *.pickle*. Iako je *pickle* jednostavan za korištenje i uspješno rješava problem serijalizacije podataka, problem koji dolazi uz *pickle* je opasnost prilikom izvršavanja deserijalizacije „pickleane“ datoteke. Deserijalizacija, osim čitanja pohranjene datoteke, može okinuti aktivaciju bilo koje vanjske funkcije, što znači da postoji mogućnost da će se pozvati funkcija čiji učinak može biti maliciozan. Ovakav način napada poznat je pod nazivom „Pickle bomba“ (*engl. Pickle bomb*). Postoje slične vrste napada, kao što su primjerice „XML bomba“ ili „Tar bomba“, no za razliku od ostalih „bombi“, Pickle bombu vrlo je teško detektirati i deaktivirati [10]. Važno je napomenuti i da bazu podataka pohranjenu u *pickle* formatu čovjek ne može pročitati s razumijevanjem.

3. IZRADA RUKOVATELJA BAZOM PODATAKA TEG-A

Na osnovu nedostataka nabrojanih u potpoglavlju 2.4 koji su prisutni u postojećem rješenju TEG-a, bilo je potrebno razviti potpuno novo rješenje koje će zadovoljiti sljedeće zahtjeve:

1) Novo rješenje treba biti implementirano u programskom jeziku C++.

2) Potrebno je razviti model baze podataka iz koje će generatori moći dohvaćati potrebne podatke za generiranje izvornog C koda.

3) Potrebno je razviti metodu serijalizacije pomoću koje će biti moguće pohraniti bazu podataka u formatu u kojem će ju čovjek moći pročitati s razumijevanjem.

4) Potrebno je razviti algoritam pomoću kojeg će se detektirati promjene koje su se pojavile u konfiguracijskim datotekama koje trebaju biti parsirane u 1. koraku izvođenja TEG-a;

U ovom će poglavlju biti predstavljene korištene tehnologije pomoću kojih je novo rješenje implementirano te će detaljno biti opisani postupci koji su bili potrebni za realizaciju navedenih zahtjeva novog rješenja baze podataka TEG-a.

3.1. Tehnologije primijenjene u razvoju rukovatelja bazom podataka

U ovome će potpoglavlju ukratko biti opisane tehnologije pomoću kojih je implementirano rješenje rukovatelja baze podataka TEG-a.

3.1.1. C++

C++ programski je jezik opće namjene kojeg je stvorio Bjarne Stroustrup kao proširenje programskog jezika C ili "C s klasama". Jezik se s vremenom značajno proširio, a moderni C++ danas ima objektno orijentirane, generičke i funkcionalne značajke, osim mogućnosti za manipulaciju memorijom na niskoj razini. Gotovo se uvijek provodi kao prevedeni jezik, a mnogi dobavljači nude C++ prevoditelje [11].

C++ je dizajniran s orijentacijom prema sistemskom programiranju, programiranju ugradbenih sustava ograničenih resursa i velikim sustavima, s performansama, učinkovitošću i fleksibilnošću korištenja [12]. C++ se također našao korisnim u mnogim drugim kontekstima, a ključna prednost je softverska infrastruktura. Vrlo je primjenjiv prilikom pisanja aplikacija ograničenih resursima, uključujući aplikacije za stolna računala, video igre, poslužitelje (npr. E-trgovina, web

pretraživanje ili baze podataka) i aplikacije kritične za performanse (npr. telefonski prekidači ili svemirske sonde) [13].

C++ standardizirala je Međunarodna organizacija za standardizaciju (engl. *International Organization for Standardization* - ISO), a najnoviju standardnu verziju ISO je ratificirao i objavio u prosincu 2020 poznatu pod nazivom C++ 20 [14].

3.1.2. Microsoft Visual Studio

Microsoft Visual Studio razvojno je okruženje (engl. *Integrated Development Environment* - IDE) koje je razvila američka tvrtka Microsoft. Koristi se za razvoj računalnih programa, kao i web stranica, web aplikacija, web usluga i mobilnih aplikacija. Visual Studio koristi Microsoftove platforme za razvoj softvera kao što su Windows API, Windows Forme, Windows Trgovina i Microsoft Silverlight. Može proizvesti izvorni kod i upravljani kod. Visual Studio uključuje uređivač koda koji podržava *IntelliSense* (komponentu dovršetka koda), kao i preradu koda. Integrirani alat za ispravljanje pogrešaka radi i kao ispravljač pogrešaka na izvornoj razini i na strojnoj razini. Ostali ugrađeni alati uključuju kodiranje profila, dizajner za izradu grafičkih aplikacija, web dizajner, dizajner klasa i dizajner sheme baze podataka. Prihvaća dodatke koji proširuju funkcionalnost na gotovo svim razinama, uključujući dodavanje podrške za sustave kontrole verzije koda (poput Subversiona i Gita) i dodavanje novih skupova alata poput uređivača i vizualnih dizajnera za jezike specifične za domenu [15].

3.2. Implementacija nove baze podataka TEG-a

Kao što je već spomenuto u 2. poglavlju, uloga baze podataka kod TEG-a je pružanje mogućnosti skladištenja brojnih podataka dobivenih parsiranjem konfiguracijskih datoteka. Podatke je potrebno raspodijeliti u cjeline kako bi im se moglo lako i brzo pristupiti prilikom generiranja izvornog koda. Na slici 3.1. prikazan je isječak sadržaja jedne od konfiguracijskih datoteka tipa *.arxml* [4]. Kao i kod klasičnih *.xml* datoteka, podatci unutar prikazane konfiguracijske datoteke poredani su u hijerarhijsku strukturu. Drugim riječima, stavke unutar datoteke povezane su s ostalim stavkama roditeljsko-dječjim vezama (engl. *parent-child link*) [16]. S obzirom na to da baza podataka mora skladištiti podatke iz istovrsnih datoteka kao što je datoteka prikazana na slici 3.1., podatci unutar baze bit će također poredani u hijerarhijsku strukturu [17].

```
<AR-PACKAGE>
  <SHORT-NAME>AUTOSAR_Platform</SHORT-NAME>
  <AR-PACKAGES>
    <AR-PACKAGE>
      <SHORT-NAME>BaseTypes</SHORT-NAME>
      <ELEMENTS>
        <SW-BASE-TYPE UUID="sfjsdofdjfois45335fdsfsdf">
          <SHORT-NAME>boolean</SHORT-NAME>
          <CATEGORY>FIXED_LENGTH</CATEGORY>
          <BASE-TYPE-SIZE>8</BASE-TYPE-SIZE>
          <BASE-TYPE-ENCODING>BOOLEAN</BASE-TYPE-ENCODING>
          <NATIVE-DECLARATION>boolean</NATIVE-DECLARATION>
        </SW-BASE-TYPE>
        <SW-BASE-TYPE UUID="9854njkhnftlgdkrtnm6k5ljnflggn">
          <SHORT-NAME>float32</SHORT-NAME>
          <CATEGORY>FIXED_LENGTH</CATEGORY>
          <ADMIN-DATA>
            <SDGS>
              <SDG GID="XYZ">
                <SD GID="QWERTY">1</SD>
              </SDG>
            </SDGS>
          </ADMIN-DATA>
          <BASE-TYPE-SIZE>32</BASE-TYPE-SIZE>
          <BASE-TYPE-ENCODING>IEEE754</BASE-TYPE-ENCODING>
        </SW-BASE-TYPE>
        <SW-BASE-TYPE UUID="foigj656746456fkgldjfgkljiofj">
          <SHORT-NAME>float64</SHORT-NAME>
          <CATEGORY>FIXED_LENGTH</CATEGORY>
```

Slika 3.1. Prikaz sadržaja konfiguracijske datoteke tipa *.arxml*

Hijerarhijska baza podataka spada pod vrstu nerelacijskih baza podataka, jer su podatci unutar baze smješteni u strukturu koja ima oblik stabla (engl. *tree-like structure*), dok su podatci u relacijskim bazama podataka smješteni u tablice. Nerelacijske baze podataka imaju nekoliko prednosti nad relacijskim bazama podataka, a što se tiče TEG-a, najvažnije su prednosti bolja organizacija velikih skupova podataka i mogućnost fleksibilnog proširenja baze podataka. Bolja organizacija velikih setova podataka ne podrazumijeva samo sposobnost pohrane ogromne količine informacija, već i pretragu velikih podataka s lakoćom. Opseg i brzina pretraživanja podataka ključne su prednosti nerelacijskih baza podataka. Što se tiče mogućnosti fleksibilnog proširenja baze podataka, kako se prikuplja više informacija, nerelacijska baza podataka može apsorbirati te nove podatkovne točke, obogaćujući postojeću bazu novim razinama granularne vrijednosti (engl. *granular value*), čak i

ako se ne uklapaju u vrste podataka prethodno postojećih informacija [18]. S obzirom na to da konfiguracijske datoteke mogu biti reda veličine po nekoliko stotina tisuća linija koda, vrlo je važno da baza podataka bude u stanju brzo pretražiti tražene podatke.

3.2.1. Tip komponenti baze podataka

S obzirom na to da komponente unutar konfiguracijskih datoteka mogu biti vrlo različite po pitanju veličine, strukture i tipa informacija koje sadrže u sebi, bilo je potrebno razviti korisnički definiran tip podatka koji treba biti prilagođen za svaku komponentu unutar konfiguracijske datoteke. Na slici 3.2 prikazan je isječak koda deklaracije klase „*DataComponent*“ čije će instance biti osnovne komponente baze podataka.

```
 9  class DataComponent : public std::enable_shared_from_this<DataComponent>
10  {
11  public:
12  Constructors
30
31  Public Methods
84
85  private:
86  #pragma region Attributes
87      std::map<std::string, std::shared_ptr<DataComponent>> children;
88      std::map<std::string, std::shared_ptr<DataComponent>> siblings;
89      std::map<std::string, std::string> data;
90      std::shared_ptr<DataComponent> parent;
91      std::string ID;
92  #pragma endregion
```

Slika 3.2. Programski kod za definiranje tipa komponente u programskom jeziku C++

Kao što je navedeno ranije u poglavlju, baza podataka ima strukturu u obliku stabla (engl. *tree-like*), što znači da sve komponente u bazi moraju biti povezane roditeljsko-dječjim vezama (engl. *parent-child links*). To je postignuto tako što svaka instancirana komponenta u sebi sadrži pametne pokazivače na svoju djecu, braću i sestre te roditelja, odnosno u sebi ima zapisane memorijske adrese navedenih članova obitelji komponente. Pametni pokazivači definirani su u standardnoj biblioteci predložaka (engl. *Standard Template Library* - STL) te su službeno u primjeni od pojave inačice C++ 11 [19]. Pametni pokazivači koriste se kako bi se osiguralo da u programima ne dođe do curenja memorije (engl. *memory leak*) i resursa te su sigurni za iznimke [20]. Pametni su pokazivači ustvari omot (engl. *wrapper*) oko sirovih pokazivača (engl. *raw pointer*) koji oslobađa programera zadaće upravljanja memorijom. Pomoću spomenutih pokazivača na članove obitelji

određene komponente, moguće je od bilo koje komponente doći do neke druge u konačnom broju koraka, odnosno u konačnom broju funkcijskih poziva.

3.2.2. Tip baze podataka

Da ne bi došlo do nepreglednosti radi držanja cijele baze podataka u jednoj korijenskoj komponenti, kreirana je nova vrsta podatka koja ima sposobnost skladištenja komponenata raspoređenih u pakete kao u konfiguracijskoj datoteci. Na slici 3.3 prikazan je isječak koda deklaracije klase „*SWCDatabase*“ čija će se instanca koristiti za skladištenje svih podataka dobivenih iz konfiguracijskih datoteka. Na slici 3.4 prikazani su AUTOSAR paketi (AR-PACKAGE) unutar konfiguracijske datoteke. Radi preglednije organizacije i brže pretrage baze podataka, na isti će se način strukturirati komponente u skladištu.

```
18  class SWCDatabase
19  {
20  public:
21  deleted_functions
28
29      static std::unique_ptr<SWCDatabase> GetDynamicInstance();
30
31  private:
32      void HandleComponents(std::vector<std::string>&, std::weak_ptr<DataComponent>&, std::weak_ptr<DataComponent>&, bool&);
33      void HandleData(std::vector<std::string>&, std::weak_ptr<DataComponent>, bool&, bool&);
34      std::map<std::string, std::shared_ptr<DataComponent>> storage;
35      std::map<std::string, std::shared_ptr<DataComponent>> secondaryStorage;
36      fs::path GenerateFilename();
37      std::string GetLatestDatabase();
38      void SaveDatabaseToFile();
39      void LoadLatestDatabase();
40      void AddPackage(std::shared_ptr<DataComponent>);
41      void AddUnmatchedData(std::shared_ptr<DataComponent>);
42      friend class DatabaseHandler;
43  };
```

Slika 3.3. Programski kod za definiranje tipa baze podataka u programskom jeziku C++

```
     2  <AUTOSAR xmlns="http://autosar.org/schema/r4.0" xmlns:xsi="http
     3    <AR-PACKAGES>
     4      <AR-PACKAGE>
   152      <AR-PACKAGE>
   767      <AR-PACKAGE>
 27762      <AR-PACKAGE T="2020-07-17T15:25:40+02:00">
280608      <AR-PACKAGE T="2020-07-17T15:29:06+02:00">
290163      <AR-PACKAGE>
293933      <AR-PACKAGE>
293946      <AR-PACKAGE T="2020-07-29T10:13:40+02:00">
297566      <AR-PACKAGE UUID="5140d7ac-fafd-43c7-a751-692245b8ac4a">
297630      <AR-PACKAGE>
298077    </AR-PACKAGES>
298078  </AUTOSAR>
```

Slika 3.4. Konfiguracijska datoteka - AUTOSAR paketi

To je postignuto pomoću posebnog spremnika podataka koji se naziva mapa (engl. *map*). Mapa je sortirani asocijativni spremnik koji sadrži parove ključ-vrijednost s jedinstvenim ključevima te je poput pametnih pokazivača također dio STL-a. Ključevi se sortiraju pomoću usporedne funkcije *Compare*, a svugdje gdje STL koristi zahtjeve *Compare*, jedinstvenost se utvrđuje pomoću relacije ekvivalencije [21]. Operacije pretraživanja, dodavanja i uklanjanja elemenata mape imaju logaritamsku složenost. Mape se obično implementiraju kao crveno-crna stabla [22]. Crveno-crno stablo vrsta je samo-balansirajućeg binarnog stabla pretraživanja. Svaki čvor pohranjuje dodatni bit koji predstavlja "boju" ("crvena" ili "crna"), a služi za osiguravanje da stablo ostane balansirano tijekom umetanja i uklanjanja elemenata stabla. Kada se stablo promijeni, novo stablo se preuređuje i "prebojava" kako bi se vratila svojstva bojanja koja ograničavaju koliko stablo može postati nebalansirano u najgorem slučaju. Svojstva su projektirana tako da se ovo preuređivanje i ponovno bojanje može učinkovito izvesti [23]. Mapa, korištena za skladište baze podataka TEG-a, za ključeve koristi identifikacijsku oznaku, odnosno naziv AUTOSAR paketa koji je znakovnog tipa *string*, a element koji predstavlja vrijednost mape je pametni pokazivač na korijensku komponentu paketa čiji se naziv nalazi u ključu. Korištenje STL mape za odvajanje paketa omogućava brže pretraživanje određenih komponenata u slučaju kada korisnik zna u kojem točno paketu se željena komponenta nalazi.

S obzirom na to da instanca klase „*SWCDatabase*“ ima isključivo funkciju spremnika podataka, interakcija sa samim podatcima odvojit će se u poseban tip podatka koji će imati ulogu rukovatelja baze podataka.

3.2.3. Razvoj rukovatelja bazom podataka

Kao što je napomenuto u prethodnom potpoglavlju, kako bi se korisnike odvojilo od direktnog kontakta sa spremnikom podataka, implementiran je rukovatelj pomoću kojeg će korisnici moći pretraživati bazu te umetati nove elemente. Instanca rukovatelja je jedini objekt s kojim će korisnici imati direktan kontakt, a sve ostalo odvija se u pozadini. Na slici 3.5 prikazan je isječak koda deklaracije klase „*DatabaseHandler*“ čije su metode (funkcije) napisane u odjeljenju javnog pristupa, što znači da ih je moguće koristiti izvan opsega klase.

```
class DatabaseHandler
{
public:
    DatabaseHandler() = delete;
    DatabaseHandler(std::unique_ptr<SWCDatabase>);
    ~DatabaseHandler();

    //Fills data on a DataComponent at given path and overwrites existing data with same keys, returns false if component doesn't exist
    bool FillDataByPath(std::vector<std::string>, std::map<std::string, std::string>);

    //Fills data on DataComponent with given key and overwrites existing data with same keys, returns false if component doesn't exist
    bool FillDataByKey(std::string, std::map<std::string, std::string>);

    //Get DataComponent by path eg. Autosar/Services/Dem...
    std::shared_ptr<DataComponent> GetComponentByPath(std::vector<std::string>);

    //Returns DataComponent by partial path and component key
    std::shared_ptr<DataComponent> GetComponentByPartialPath(std::string, std::vector<std::string>);

    //Get DataComponent by key
    std::shared_ptr<DataComponent> GetComponentByKey(std::string);

    //Saves database to file
    void SaveDatabase();

    //Loads latest database from a file
    void LoadDatabase();

    //Adds filled DataComponent to the storage
    void AddPackage(std::shared_ptr<DataComponent>);

    //Fills a vector with DataComponents that contain given hint in their ID
    void GetComponentsByHint(std::vector<std::weak_ptr<DataComponent>>&, std::string);

    //Adds a new filled component to secondary storage
    void FillUnmatchedData(std::string, std::map<std::string, std::string>);
```

Slika 3.5. Programski kod za definiranje rukovatelja bazom podataka u programskom jeziku C++

Rukovatelj baze podataka nudi nekoliko mogućnosti pretraživanja podataka:

1) Pomoću identifikacijske oznake komponente;

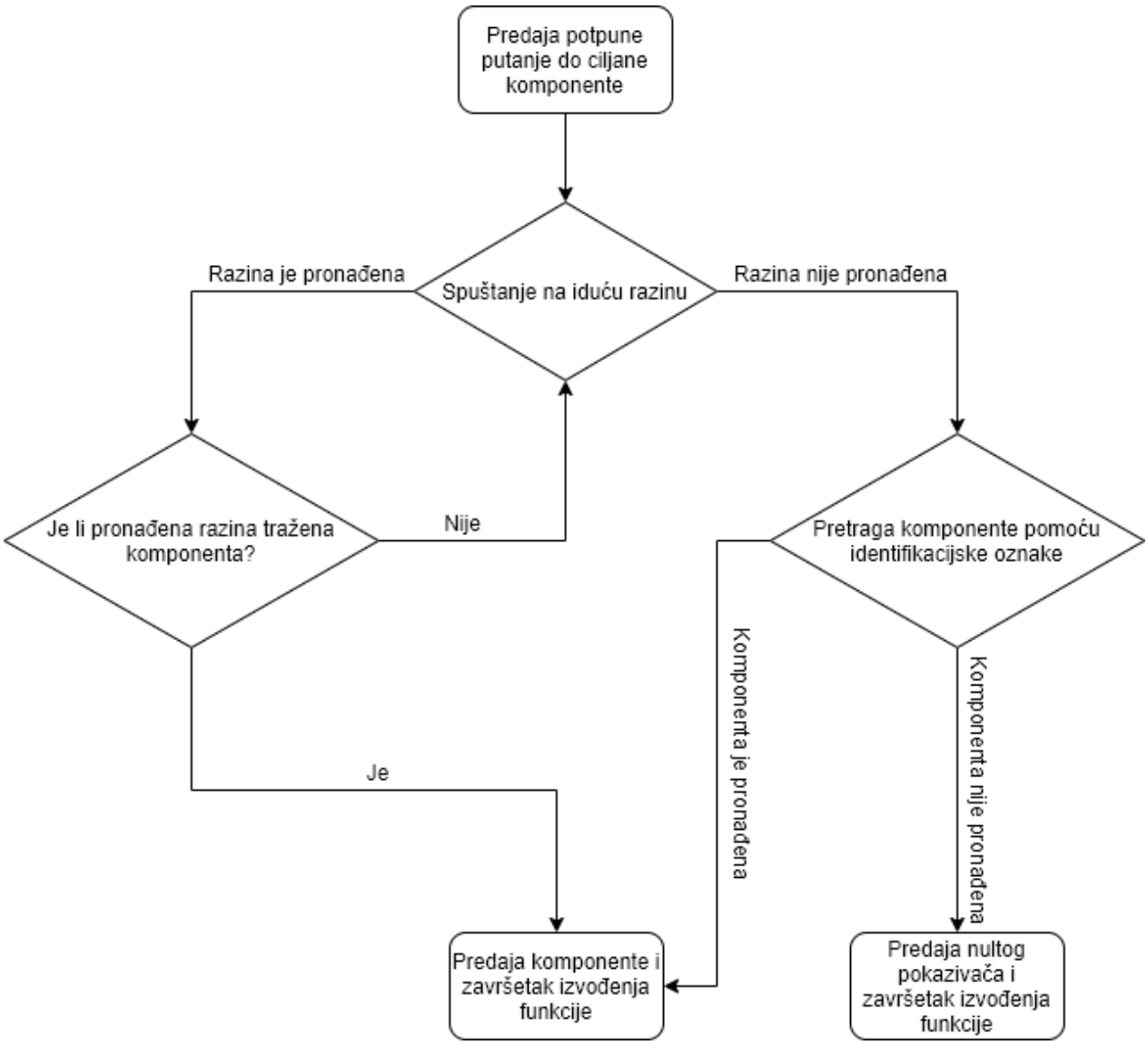
2) Pomoću potpune putanje do tražene komponente;

3) Pomoću djelomične putanje do tražene komponente;

4) Pretraživanje više istovrsnih komponenata odjednom;

Pretraživanje pomoću identifikacijske oznake komponente korisnički je najjednostavnija metoda pretraživanja, no što se tiče samog algoritma pretraživanja, najsloženije je te je složenost logaritamska (O(log N)) [24]. Traženje se izvodi obilaskom stabla unaprijed (engl. p*reorder traversal*). U ovoj se metodi prolaska prvo posjećuje korijenski čvor, zatim sva podstabla s lijeva na desno. U slučaju kada korisnik pokuša potražiti komponentu koja se ne nalazi u bazi podataka, rukovatelj će vratiti nulti pokazivač (engl. *null pointer*) [24]. Nulti pokazivač ima rezerviranu vrijednost koja se naziva konstanta nultog pokazivača, za ukazivanje da pokazivač ne upućuje ni na koji valjani objekt ili funkciju. C ++ 11 definira novu konstantu nultog pokazivača naziva „nullptr“, koja se može pretvoriti u bilo koju vrstu pokazivača [25].

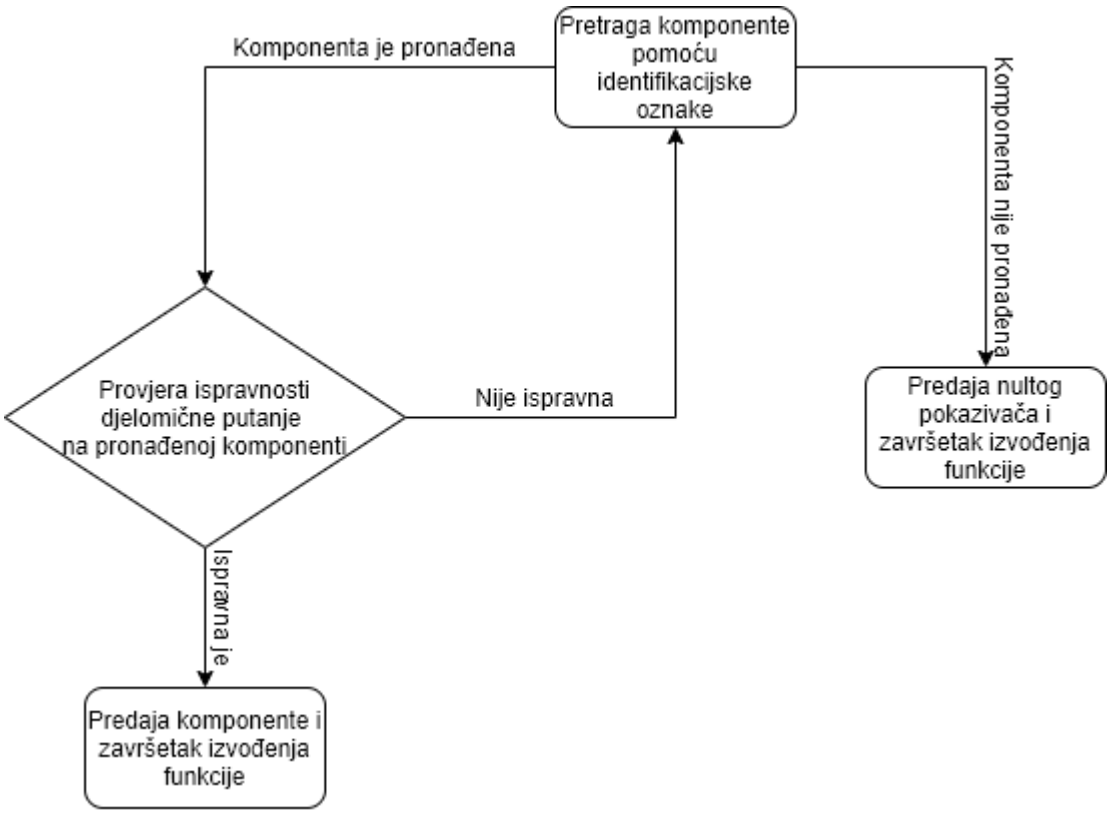
Postoje slučajevi u kojima je korisniku dostupna potpuna putanja do komponente koju želi pronaći u bazi podataka. U tom je slučaju najoptimalnije koristiti pretraživanje pomoću potpune putanje do komponente. Glavni razlog korištenja metode pretrage pomoću potpune putanje je manja složenost od pretraživanja pomoću identifikacijske oznake. Na slici 3.6 prikazani su koraci tokom izvođenja pretrage komponente pomoću potpune putanje.



Slika 3.6. Dijagram toka pretrage pomoću potpune putanje

U slučaju kada pretraga dođe do dijela putanje koji je ne postoji, pretraga se automatski prebacuje na pretragu pomoću identifikacijske oznake te se za identifikacijsku oznaku koristi zadnja razina predane putanje, koja ujedno predstavlja traženu komponentu. Ako komponenta nije pronađena ni nakon prelaska na pretragu pomoću identifikacijske oznake, rukovatelj će vratiti nulti pokazivač.

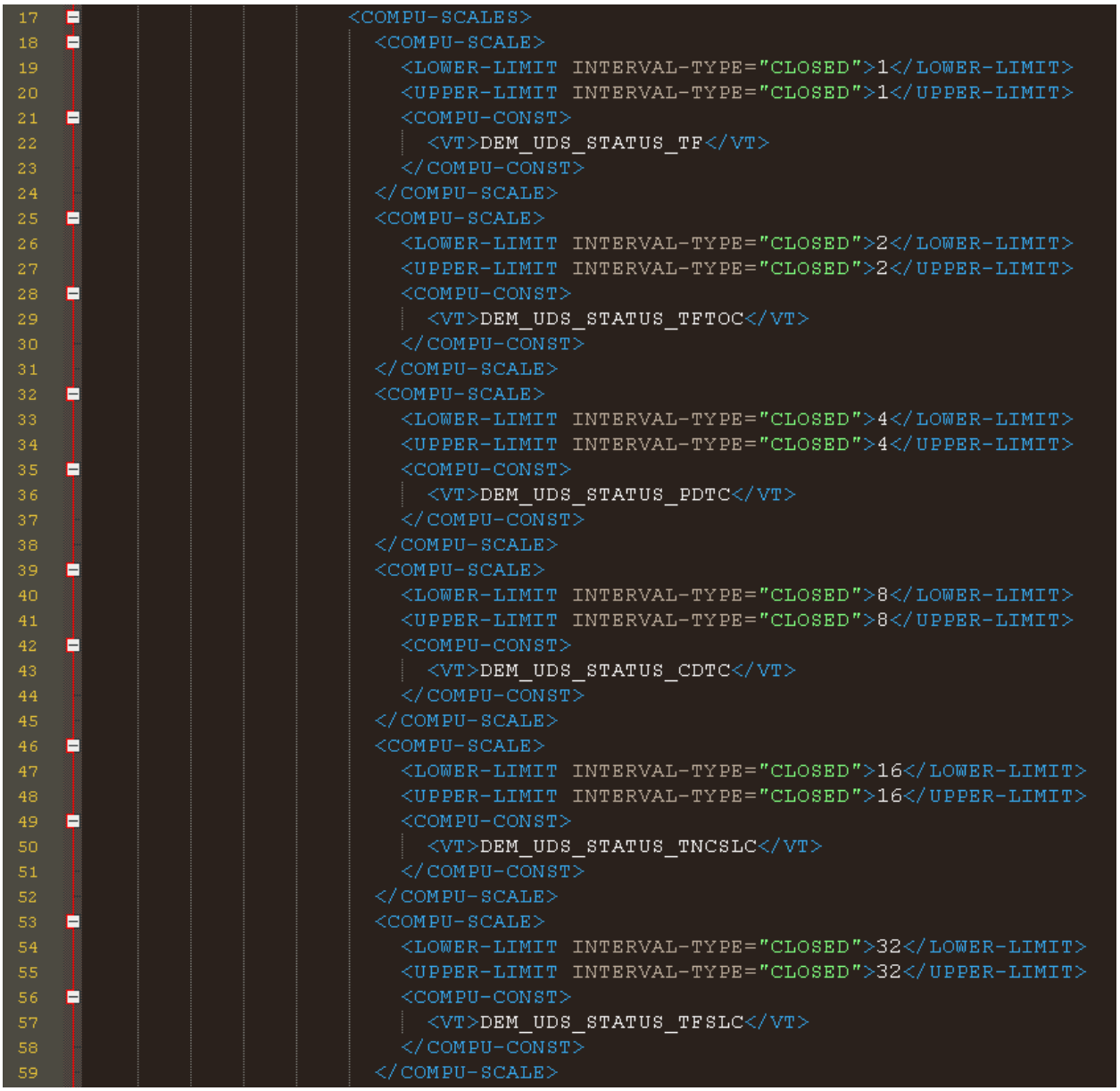
U slučaju kada bi se našle komponente jednakih identifikacijskih oznaka, no različitog smještaja što se tiče paketa u kojem se nalaze, tada pretraga pomoću djelomične putanje postaje korisna. Na slici 3.7 prikazani su koraci tijekom izvođenja pretrage pomoću djelomične putanje.



Slika 3.7. Dijagram toka pretrage pomoću djelomične putanje

U slučaju kada je pronađena komponenta koja u sebi sadržava odgovarajuću identifikacijsku oznaku, ali ne i odgovarajuću djelomičnu putanju, tada se pretraga nastavlja sve dok se ne pronađe komponenta koja sadržava oba ispunjena uvjeta pretrage. Ako niti jedna komponenta ne odgovara zadanim kriterijima pretrage, rukovatelj vraća nulti pokazivač.

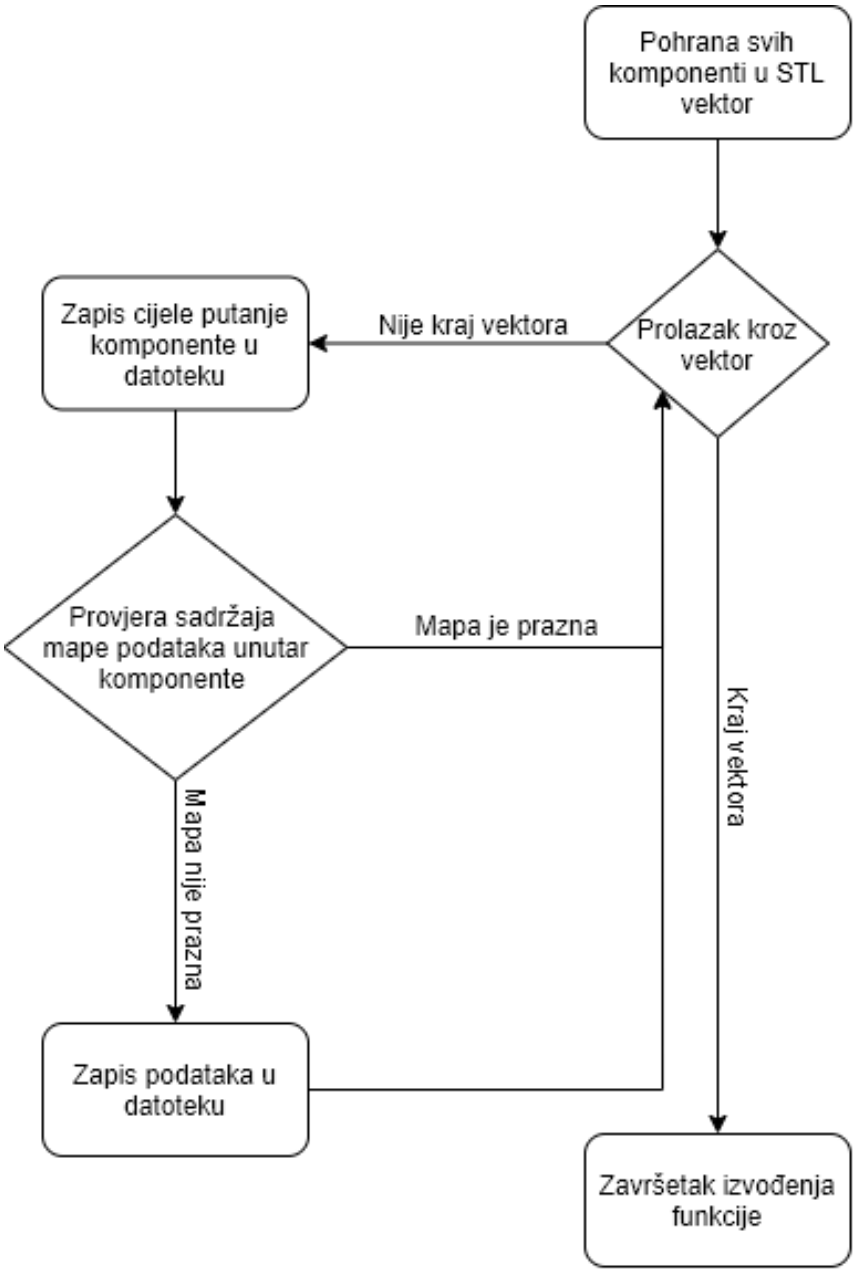
Pretraživanje više istovrsnih komponenata odjednom odgovara slučajevima kada je korisniku potrebna vektorska struktura komponenata, a ne samo jedna komponenta. Na slici 3.8 prikazan je skup komponenata istih identifikacijskih oznaka, ali različitog sadržaja. Ovakav primjer komponenata odgovara pretrazi više istovrsnih komponenata odjednom. U slučajevima kada više različitih paketa u sebi sadrže jednake vektorske skupove komponenata, tada je moguće napraviti filtriranje dobivenog pretraženog skupa na način da se izbace komponente koje ne odgovaraju putanji postavljenoj u metodi filtriranja.



Slika 3.8. Prikaz vektorske strukture komponenata unutar konfiguracijske datoteke

3.3. Implementacija serijalizacije baze podataka

S obzirom na to da pri pokretanju TEG-a parseri moraju čitati i parsirati podatke iz vrlo velikih konfiguracijskih datoteka, taj proces može potrajati po 10 sekundi ili čak i više. Kako bi se smanjilo vrijeme izvođenja narednih pokretanja TEG-a, implementirana je serijalizacija koja pohranjuje kompletnu strukturu baze podataka zajedno sa svim podatcima svake komponente. Pohranjena baza čovjeku je razumljiva te njeno učitavanje u programu traje znatno kraće od parsiranja svih konfiguracijskih datoteka. O tome će više informacija biti dano u 4. poglavlju rada. Na slici 3.9 prikazani su koraci izvođenja pohrane spremnika baze podataka u datoteku.



Slika 3.9. Dijagram toka pohrane baze podataka u datoteku

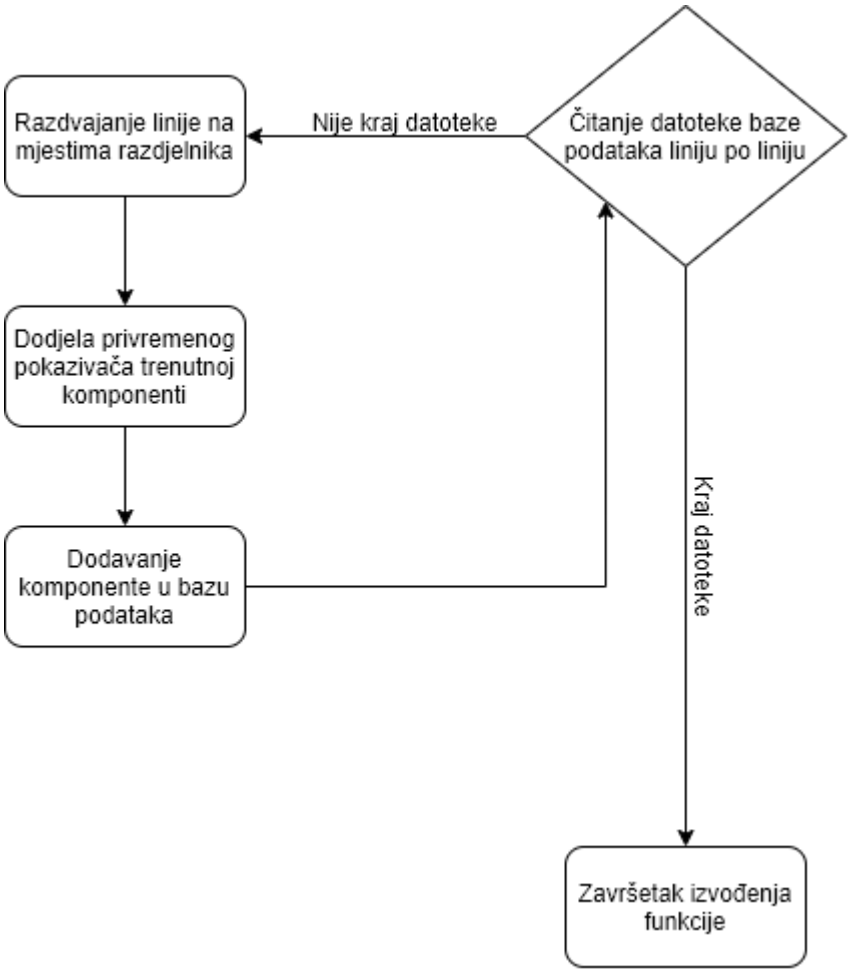
Korisnički tip komponente spomenut u dijelu 3.1.1 u sebi ima funkcionalnost dohvaćanja apsolutne putanje na kojoj se sama komponenta nalazi. Pomoću te funkcionalnosti jednostavno se može napraviti deserijalizacija prilikom učitavanja baze podataka tokom ponovnog rada TEG-a. Na slici 3.10 može se vidjeti kako baza podataka izgleda nakon pohrane u datoteku. Znak '+' na kraju linije indikator je da komponenta sadrži podatke, a ti su podatci zapisani u sljedećim linijama sve do pojave znaka '@' koji je indikator da su svi podatci komponente upisani.

```
DataTypes<|>CompuMethods<|>DataConstrs
DataTypes<|>CompuMethods<|>DataConstrs<|>DC_PiHSVMQuestion_DeQuestionType<|>+
LOWER-LIMIT<|>0
UPPER-LIMIT<|>15<|>@
DataTypes<|>CompuMethods<|>DataConstrs<|>Dt_ENUM_CLKSOURCE_DataConstr<|>+
LOWER-LIMIT<|>0
UPPER-LIMIT<|>2<|>@
DataTypes<|>CompuMethods<|>DataConstrs<|>Dt_ENUM_ContinentCodes_DataConstr<|>+
LOWER-LIMIT<|>0
UPPER-LIMIT<|>7<|>@
DataTypes<|>CompuMethods<|>DataConstrs<|>Dt_ENUM_CpEventMgr_TestResultTyp_DataConstr<|>+
LOWER-LIMIT<|>0.0
UPPER-LIMIT<|>2<|>@
DataTypes<|>CompuMethods<|>DataConstrs<|>Dt_ENUM_EHErrorID_DataConstr<|>+
LOWER-LIMIT<|>0
UPPER-LIMIT<|>65535<|>@
DataTypes<|>CompuMethods<|>DataConstrs<|>Dt_ENUM_Em_debounce_status_e_DataConstr<|>+
LOWER-LIMIT<|>0.0
UPPER-LIMIT<|>3<|>@
DataTypes<|>CompuMethods<|>DataConstrs<|>Dt_ENUM_LCS_Info_DataConstr<|>+
LOWER-LIMIT<|>0
UPPER-LIMIT<|>4<|>@
DataTypes<|>CompuMethods<|>DataConstrs<|>Dt_ENUM_LCS_State_DataConstr<|>+
LOWER-LIMIT<|>0
UPPER-LIMIT<|>10<|>@
DataTypes<|>CompuMethods<|>DataConstrs<|>Dt_ENUM_PdDatablockStatus_DataConstr<|>+
LOWER-LIMIT<|>0
UPPER-LIMIT<|>65535<|>@
DataTypes<|>CompuMethods<|>DataConstrs<|>Dt_ENUM_PerWriteAllStatus_DataConstr<|>+
LOWER-LIMIT<|>0
UPPER-LIMIT<|>255<|>@
DataTypes<|>CompuMethods<|>DataConstrs<|>Dt_ENUM_Qualifier_DataConstr<|>+
LOWER-LIMIT<|>0
UPPER-LIMIT<|>2<|>@
DataTypes<|>CompuMethods<|>DataConstrs<|>Dt_ENUM_QuestionType_DataConstr<|>+
LOWER-LIMIT<|>0
UPPER-LIMIT<|>15<|>@
DataTypes<|>CompuMethods<|>DataConstrs<|>Dt_ENUM_RequestedMesswert_BISTQMPH00_DataConstr<|>+
LOWER-LIMIT<|>0
UPPER-LIMIT<|>65535<|>@
DataTypes<|>CompuMethods<|>DataConstrs<|>Dt_ENUM_RequestedMesswert_PerPH00_DataConstr<|>+
LOWER-LIMIT<|>0
UPPER-LIMIT<|>65535<|>@
DataTypes<|>CompuMethods<|>DataConstrs<|>Dt_ENUM_SWCID_DataConstr<|>+
LOWER-LIMIT<|>0
UPPER-LIMIT<|>255<|>@
```

Slika 3.10. Isječak pohranjene baze podataka

Na slici 3.11 prikazani su koraci učitavanja baze podataka iz datoteke. Svaka linija datoteke predstavlja punu putanju do neke komponente, jedino linije između znakova '+' i '@' predstavljaju podatke u obliku parova ključ-vrijednost.

Za razliku od *pickle* serijalizacije spomenute u dijelu 2.4.2, kod serijalizacije implementirane u novome rješenju TEG-a nema nikakve opasnosti. Razlog tomu je što u procesu deserijalizacije nije moguće okinuti nikakvu vanjsku funkciju, već se samo čita datoteka baze podataka koja je u tekstualnom obliku.

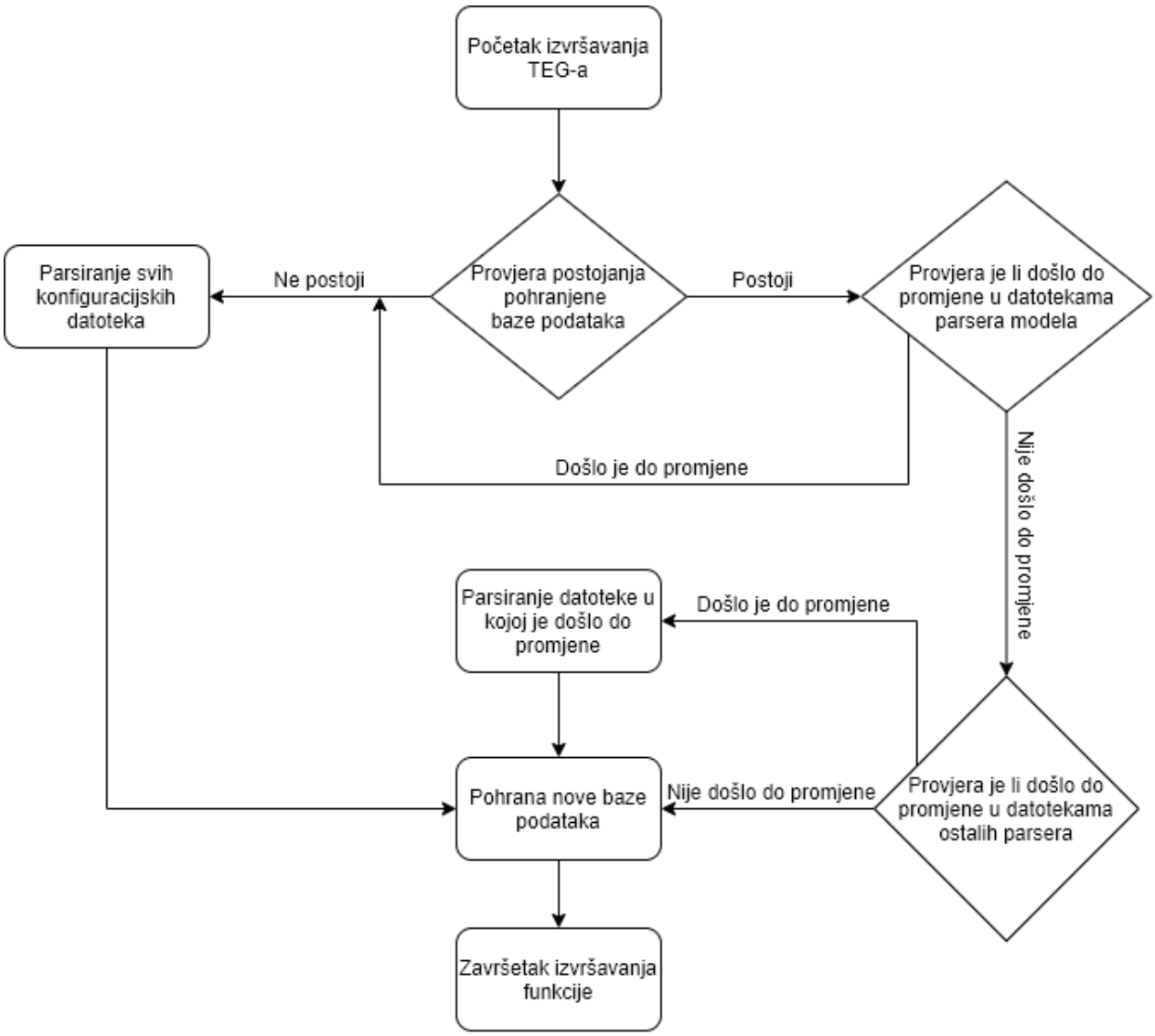


Slika 3.11. Dijagram toka učitavanja baze podataka iz datoteke

3.4. Implementacija algoritma za detekciju promjena u konfiguracijskim datotekama

Kao što je spomenuto u potpoglavlju 2.1, pri početku izvršavanja TEG-a započinje parsiranje konfiguracijskih datoteka koje mogu biti vrlo velike. S obzirom na to da je moguće koristiti deserijaliziranu bazu podataka, kao što je spomenuto u prethodnom potpoglavlju, poželjno je omogućiti provjeru je li došlo do promjene u svakoj konfiguracijskoj datoteci pojedinačno, iz razloga da se ne mora svaki put parsirati apsolutno svaka konfiguracijska datoteka, nego samo one u kojima su se pojavile promjene. Detekcija promjena postignuta je pomoću ugrađene C++ funkcije *last_write_time* koja pripada imeniku *filesystem*. Funkcija *last_write_time* radi na način da vrati posljednje vrijeme izmjene predane joj datoteke. Stoga je moguće usporediti dobiveno vrijeme s vremenom koje je datoteka dala pri zadnjem parsiranju [26]. Na slici 3.12 prikazan je tok izvođenja

1. koraka TEG-a spomenutog u dijelu 2.1.1. Pri instanciranju rukovatelja baze podataka izvršava se provjera svih konfiguracijskih datoteka.



Slika 3.12. Dijagram toka algoritma detekcije promjena

Prije negoli krene provjera konfiguracijskih datoteka, provjerava se postoji li uopće pohranjena baza podataka za učitati. Ako pohranjena baza podataka ne postoji, izvršava se parsiranje svih konfiguracijskih datoteka te se pohranjuju informacije o parsiranim datotekama i novostvorena baza podataka. U slučaju kada postoji pohranjena baza podataka, prije učitavanja treba provjeriti je li došlo do bilo kakve promjene na datotekama model parsera. Model parser parsira datoteke koje čine kostur baze podataka te ne bi imalo smisla učitavati pohranjenu bazu podataka ako njezina struktura neće odgovarati. Ako promjena na datotekama model parsera nema, učitava se pohranjena baza podataka. Nakon toga jedino preostaje provjeriti datoteke ostalih parsera te ih parsirati ako su

promijenjene od trenutka kada je stvorena pohranjena baza podataka koja je učitana. Na slici 3.13 može se vidjeti isječak datoteke u kojoj su pohranjene vrijednosti posljednjih promjena konfiguracijskih datoteka.

```
1  PerformanceHost00.arxml<|>1495834598345745
2  TTADrive.dbc<|>1495834598345295
3  TTADrive_Misc.dbc<|>1495834598347745
4  TTADrive_Object.dbc<|>1495834598345790
5  schedule_config.xml<|>1495834598316745
6  TTADrive.arxml<|>1495834598345871
7  TTADrive_Misc.arxml<|>1495834598349007
8  TTADrive_Object.arxml<|>1495834598341105
```

Slika 3.13. Prikaz posljednjih promjena konfiguracijskih datoteka

S lijeve strane simbola '<|>' nalazi se naziv konfiguracijske datoteke, a s desne se strane nalazi vrijednost koja predstavlja broj stotina nanosekundi koje su prošle od 1. siječnja 1970. godine do trenutka posljednje promjene datoteke.

4. TESTIRANJE RUKOVATELJA BAZOM PODATAKA

U ovome poglavlju opisani su rezultati testiranja novostvorenog rukovatelja bazom podataka te su rezultati uspoređeni s rezultatima dobivenim korištenjem postojećeg rješenja TEG-a. Baza podataka na kojoj je rješenje testirano sastavljena je od 8 različitih parsiranih konfiguracijskih datoteka koje odgovaraju stvarnim konfiguracijskim datotekama u automobilskoj industriji. Rješenje je testirano na Intelovom procesoru radnog takta 3.5 GHz. U tablici 4.1 prikazane su specifikacije korištenih konfiguracijskih datoteka za testiranje rada rukovatelja bazom podataka. Datoteke su odabrane iz razloga što su različitog tipa datoteke i različitog broja linija.

Tablica 4.1. Specifikacije konfiguracijskih datoteka

IME DATOTEKE	TIP DATOTEKE	BROJ LINIJA U DATOTECI	MEMORIJSKA VELIČINA DATOTEKE (MB)
PerformanceHost00	*.arxml*	298.079,00	14,10
TTADrive	*.arxml*	3.561,00	0,16
TTADrive_Misc	*.arxml*	7.090,00	0,33
TTADrive_Object	*.arxml*	3.735,00	0,21
TTADrive	*.dbc*	298,00	0,01
TTADrive_Misc	*.dbc*	434,00	0,02
TTADrive_Object	*.dbc*	156,00	0,01
schedule_config	*.xml*	3.199,00	0,16

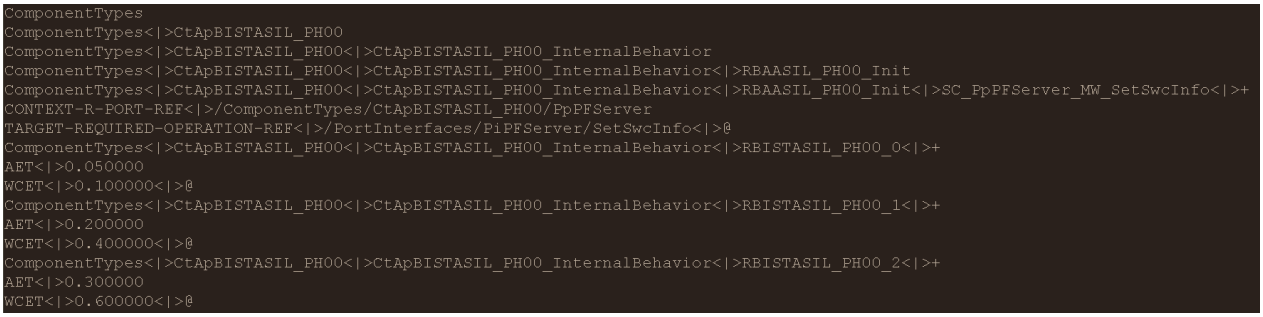
4.1. Provjera ispravnosti parsiranih podataka pohranjenih u bazu podataka

Za provjeru ispravnosti parsiranih podataka pohranjenih u bazu podataka nije bilo moguće napisati programsko rješenje iz razloga što ne postoji adekvatna baza podataka s kojom bi se nova baza podataka mogla usporediti. Stoga je ljudskim okom uspoređena struktura konfiguracijske datoteke i struktura pohranjene baze podataka.

Na slikama 4.1 i 4.2 prikazani su isječci konfiguracijske datoteke i pohranjene baze podataka. U ovom primjeru dokazano je da struktura komponenata pohranjene baze podataka odgovara strukturi komponenata u konfiguracijskoj datoteci, tj. roditeljsko-dječje veze između komponenti jednake

su u konfiguracijskim datotekama i pohranjenoj bazi podataka, što potvrđuje da su parsirani podatci pohranjeni ispravno u bazu podataka. Isto je učinjeno i za ostale konfiguracijske datoteke, čime je potvrđeno da parseri i rukovatelj bazom podataka rade ispravno.

Slika 4.1. Isječak konfiguracijske datoteke



Slika 4.2. Isječak pohranjene baze podataka

4.2. Testiranje brzine pohranjivanja parsiranih konfiguracijskih datoteka u bazu podataka

Kao što je na početku poglavlja spomenuto, testiranje brzine pohranjivanja parsiranih konfiguracijskih datoteka u bazu podataka provedeno je na računalu čiji procesor ima radni takt od 3.5 GHz. Računala s različitim specifikacijama dat će različite rezultate. Sva mjerenja odrađena su

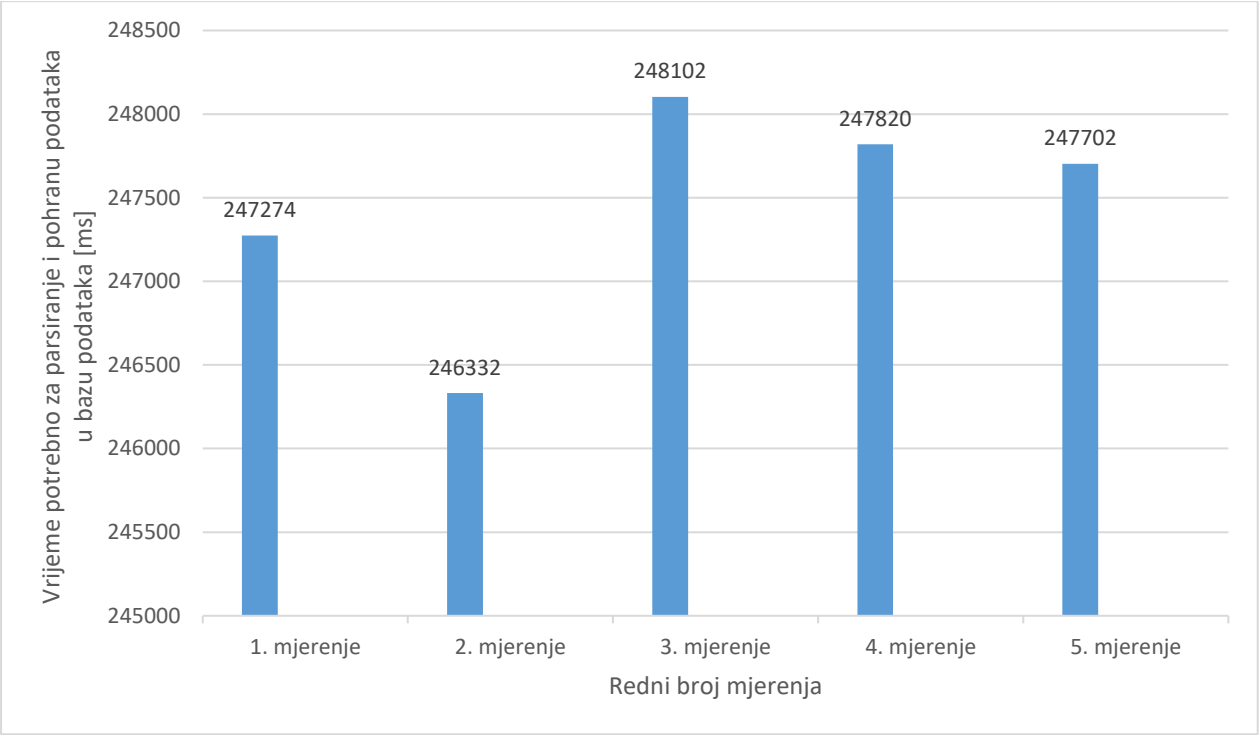
po 5 puta, a za prikaz rezultata su odabrani medijan i srednja vrijednost potrebnog vremena za izvršenje odrađenog testa.

Na slikama 4.3 – 4.10 nalaze se rezultati pojedinačnih mjerenja za svaku od 8 konfiguracijskih datoteka u postojećem rješenju i novom rješenju TEG-a. Iz rezultata je vidljivo da je vrijeme potrebno za parsiranje i pohranu podataka svih konfiguracijskih datoteka znatno manje u novoj inačici rješenja TEG-a.

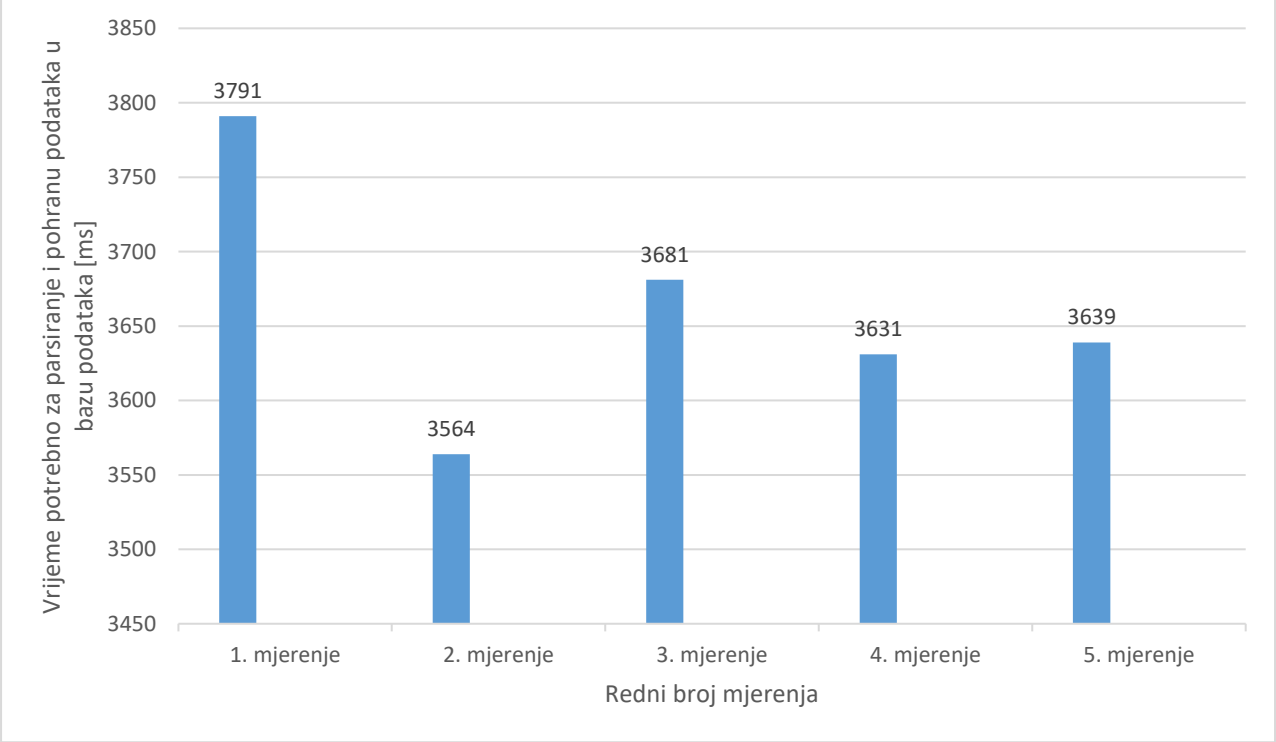
Na slici 4.11 prikazani su grafovi ovisnosti vremena potrebnog za parsiranje i pohranu podataka o logaritamskoj vrijednosti broja linija datoteka tipa *.arxml*. u postojećem i novom rješenju TEG-a. Na grafovima je korištena logaritamska vrijednost broja linija datoteka iz razloga što su velike razlike broja linija između pojedinih datoteka pa bi graf izgledao nepregledno. Što se tiče ovisnosti potrebnog vremena i broja linija, vidljivo je kako je ta ovisnost linearna. Na slici 4.12 prikazani su grafovi ovisnosti potrebnog vremena za parsiranje i pohranu podataka o broju linija datoteka tipa *.dbc* u postojećem i novom rješenju TEG-a. Kao i kod datoteka tipa *.arxml*, ovisnost potrebnog vremena za parsiranje i pohranu podataka o broja linija datoteke kod datoteka tipa *.dbc* je također linearna.

U tablici 4.2 prikazane su srednje vrijednosti i medijani vremena potrebnog za parsiranje i pohranu svih konfiguracijskih datoteka u novom i postojećem rješenju TEG-a. Iz tablice je vidljivo da se svaka konfiguracijska datoteka korištena u testiranju brže parsira i pohranjuje u bazu podataka u novom rješenju pisanom u programskom jeziku C++. Razlog zbog kojeg se program pisan u C++-u puno brže izvršava od programa pisanog u Pythonu je taj što je kod pisan u C++-u unaprijed preveden, a kod u Pythonu se prevodi tijekom izvršavanja [27].

Razlika između medijana i srednje vrijednosti potrebnog vremena za parsiranje i pohranu u bazu podataka je minimalna, iz razloga što su svi podatci u skupu bili slični, tj. nije bilo velike razlike vremena između različitih mjerenja. Vrijeme parsiranja i pohrane podataka u bazu najviše je ovisilo o broju linija konfiguracijske datoteke koja se parsirala. Kod svih različitih tipova konfiguracijskih datoteka koje su se parsirale i pohranjivale u bazu podataka, ovisnost potrebnog vremena za parsiranje i pohranu podataka u bazu podataka o broju linija datoteke bila je linearna.

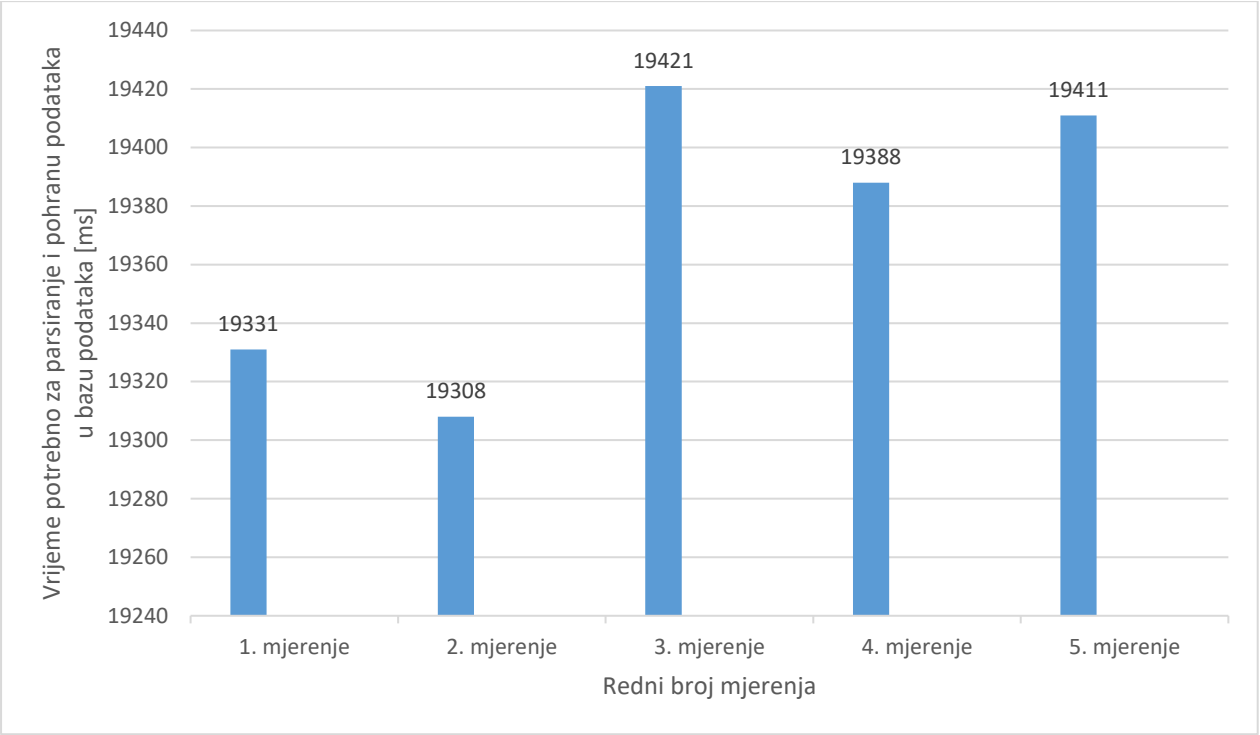


(a)

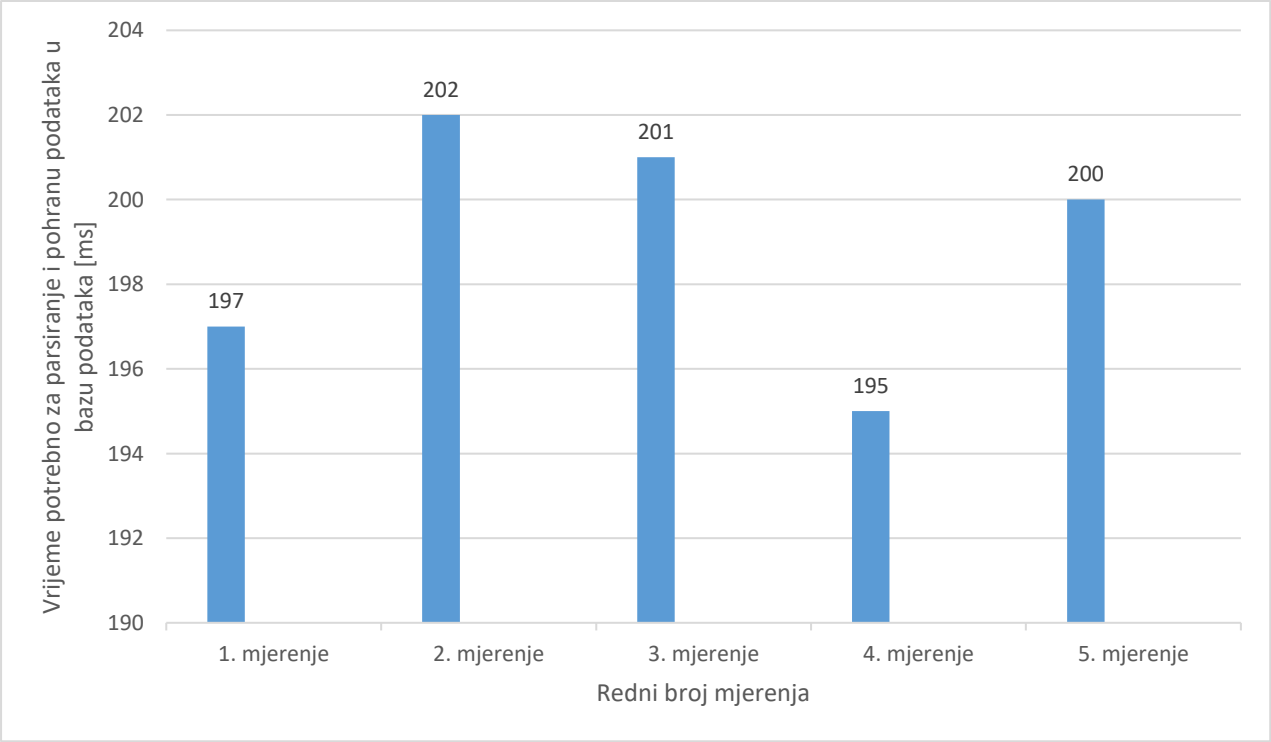


(b)

Slika 4.3. Usporedba vremena potrebnog za parsiranje i pohranu datoteke *PerformanceHost00.arxml* u (a) postojećem TEG-u realiziranom u Pythonu (b) novom TEG-u realiziranom u C++-u

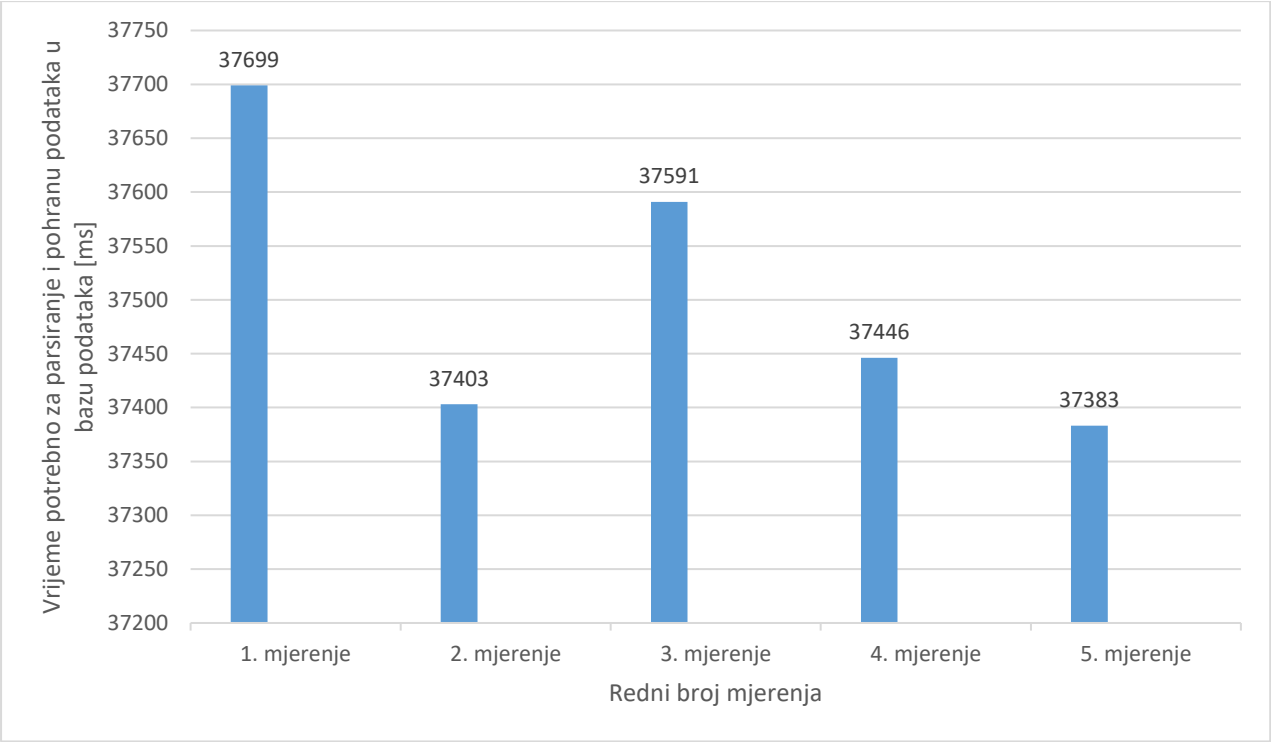


(a)

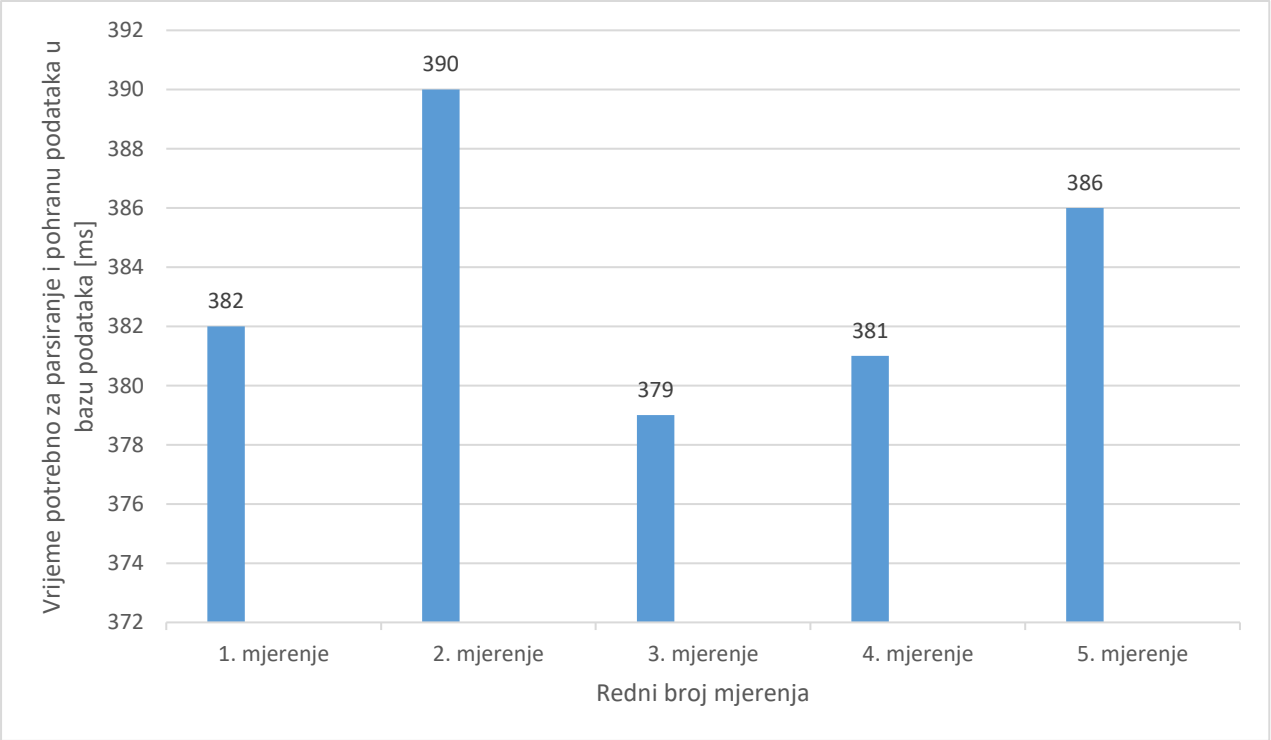


(b)

Slika 4.4. Usporedba vremena potrebnog za parsiranje i pohranu datoteke *TTADrive.arxml* u (a) postojećem TEG-u realiziranom u Pythonu (b) novom TEG-u realiziranom u C++-u

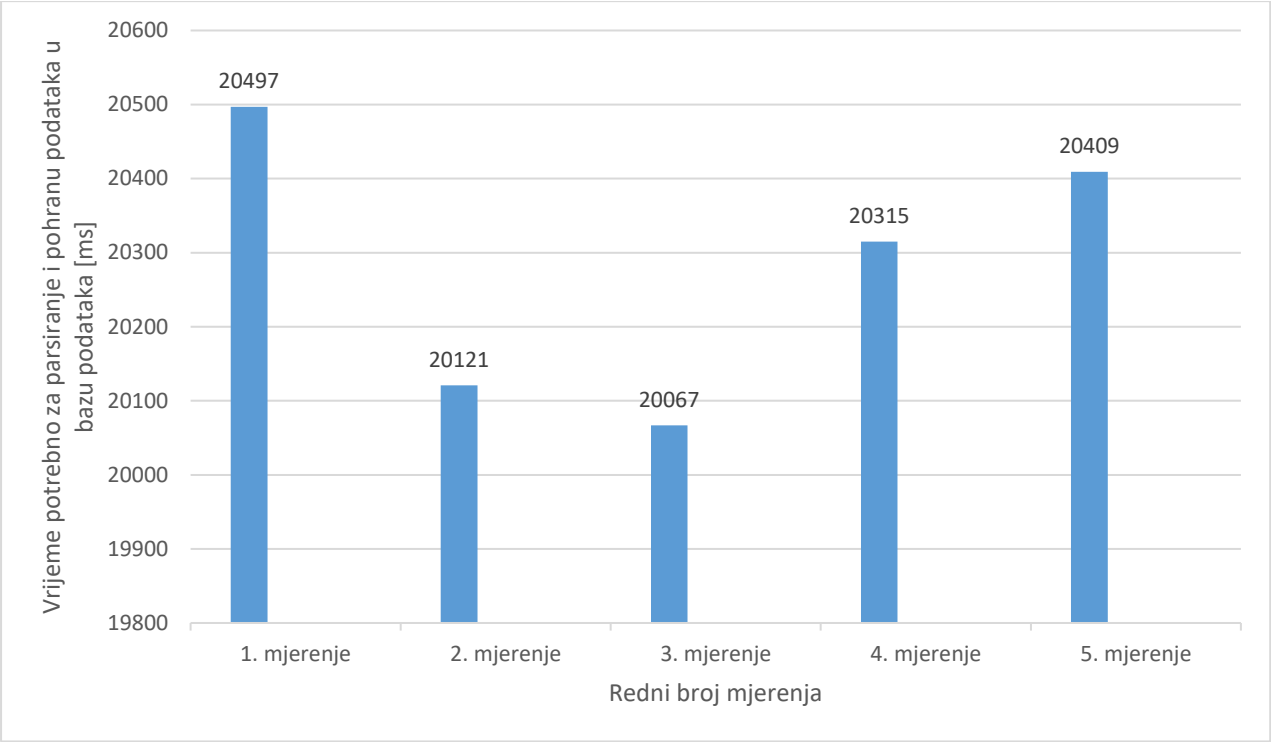


(a)

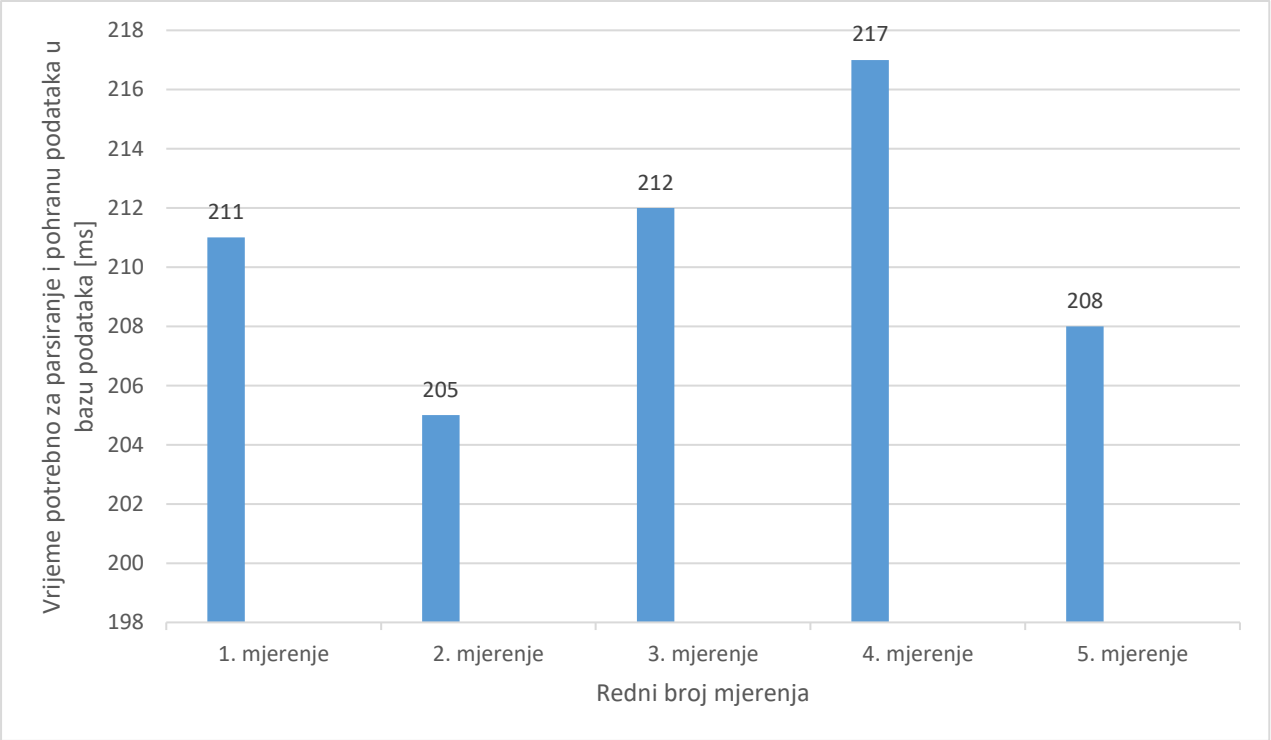


(b)

Slika 4.5. Usporedba vremena potrebnog za parsiranje i pohranu datoteke *TTADrive_Misc.arxml* u (a) postojećem TEG-u realiziranom u Pythonu (b) novom TEG-u realiziranom u C++-u

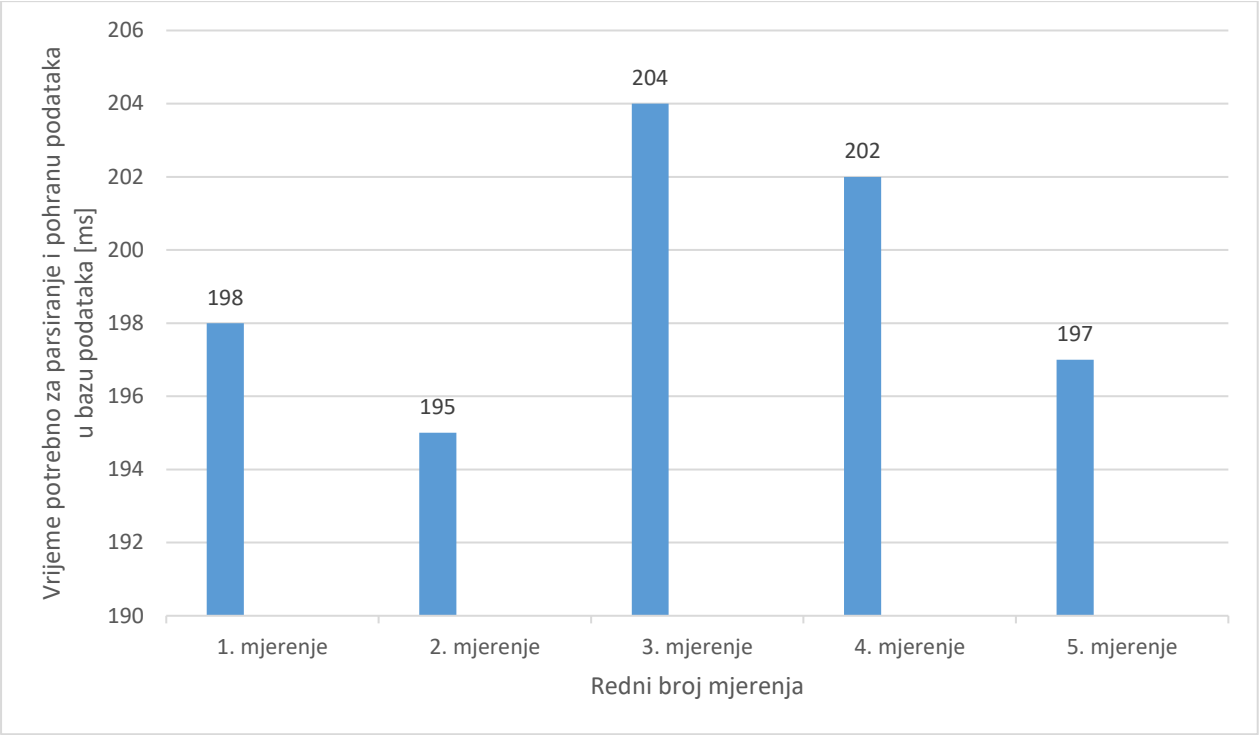


(a)

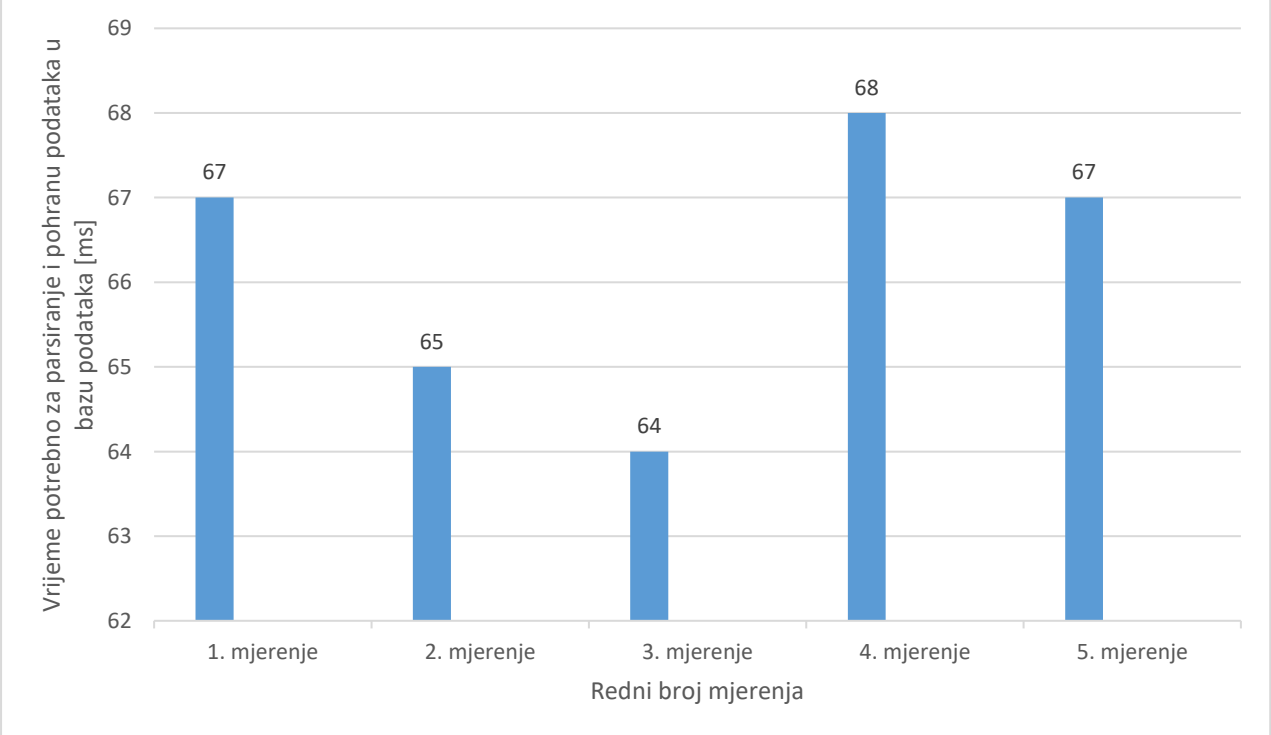


(b)

Slika 4.6. Usporedba vremena potrebnog za parsiranje i pohranu datoteke *TTADrive_Object.arxml* u (a) postojećem TEG-u realiziranom u Pythonu (b) novom TEG-u realiziranom u C++-u

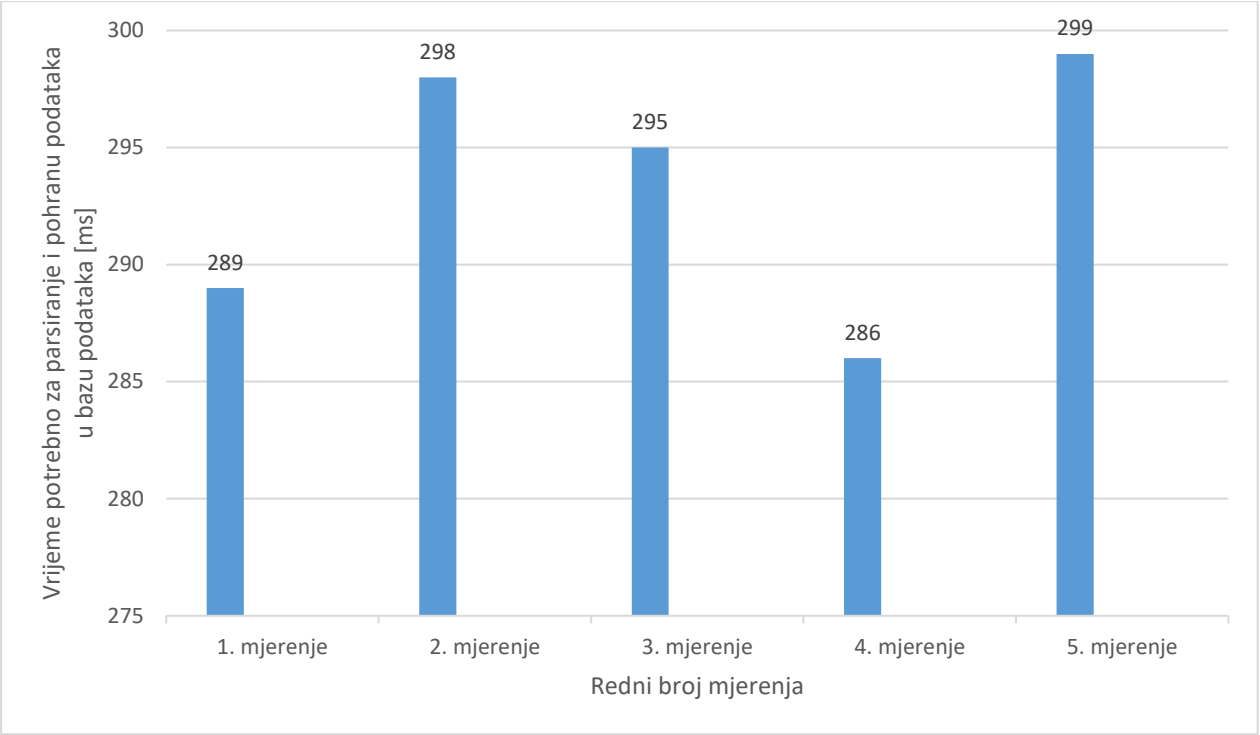


(a)

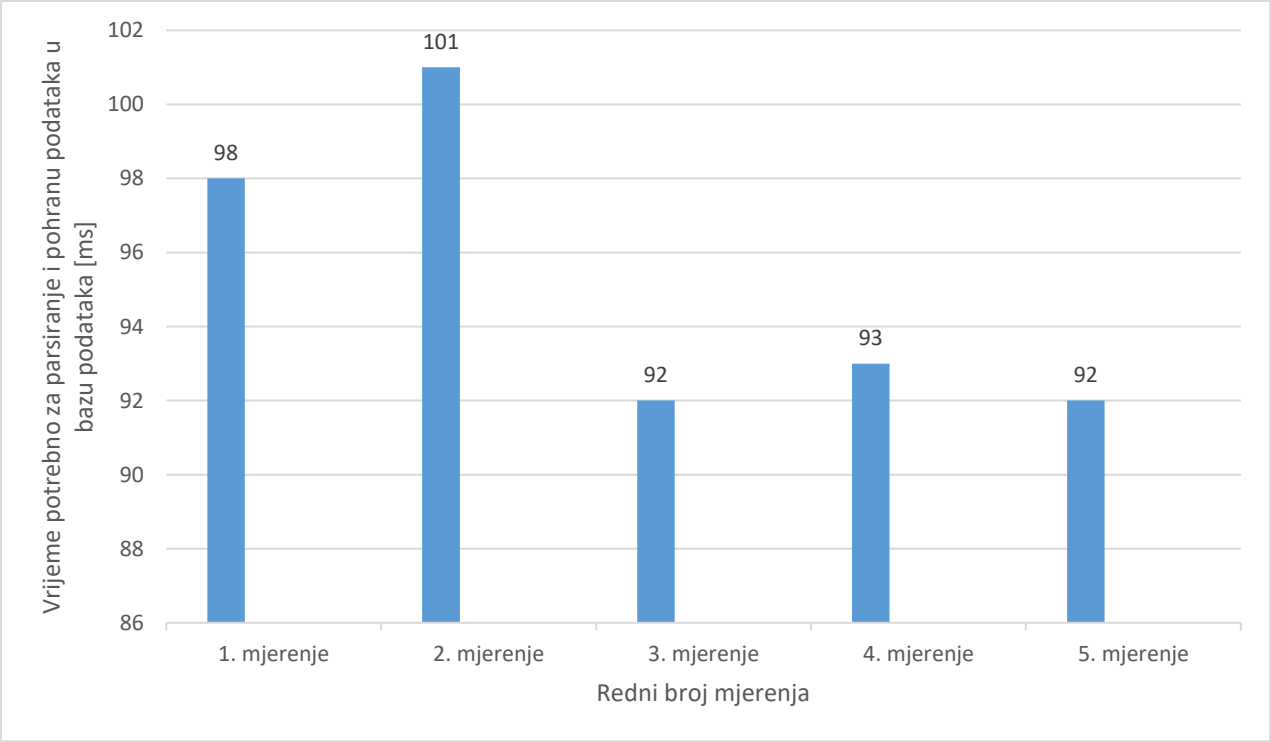


(b)

Slika 4.7. Usporedba vremena potrebnog za parsiranje i pohranu datoteke *TTADrive.dbc* u (a) postojećem TEG-u realiziranom u Pythonu (b) novom TEG-u realiziranom u C++-u

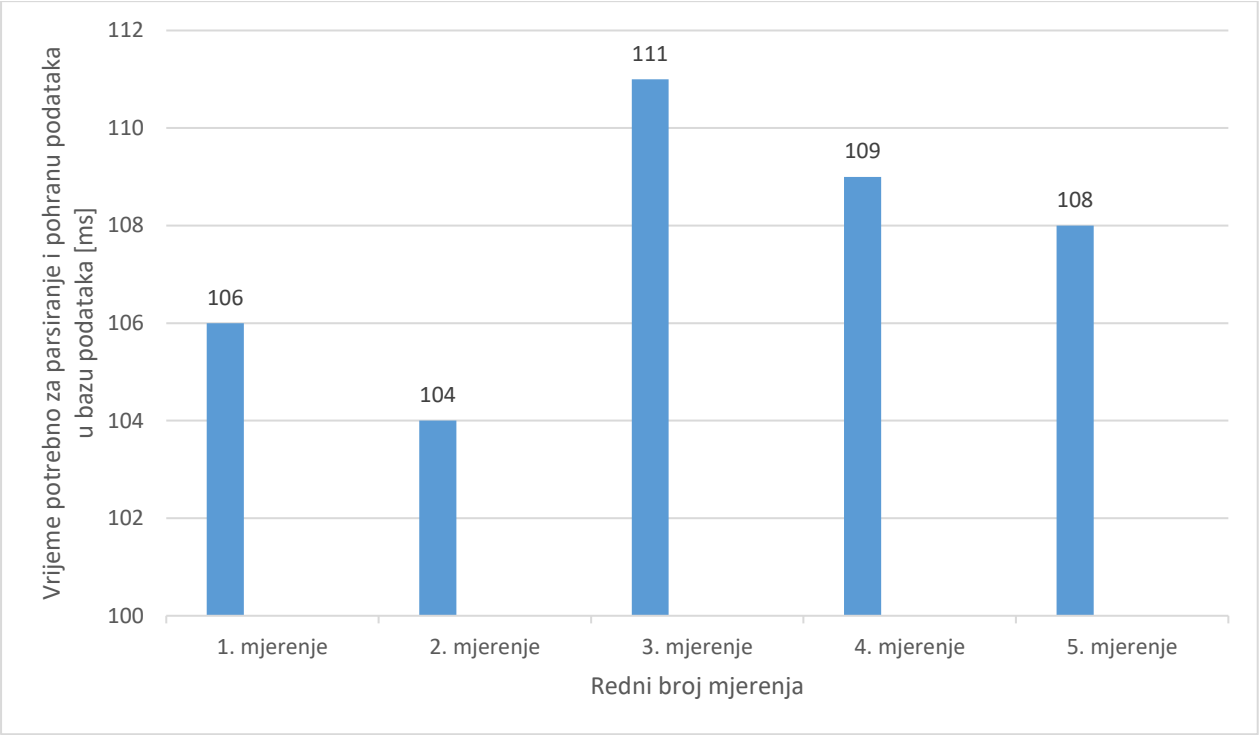


(a)

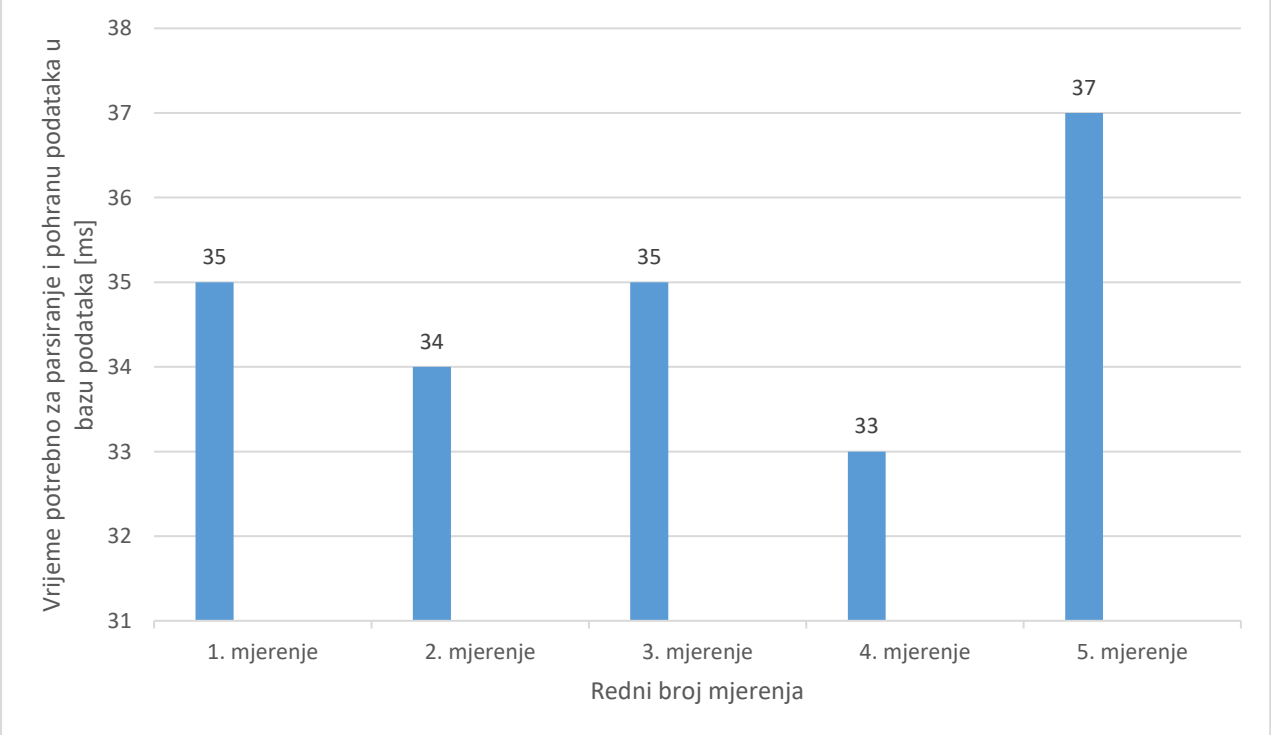


(b)

Slika 4.8. Usporedba vremena potrebnog za parsiranje i pohranu datoteke *TTADrive_Misc.dbc* u (a) postojećem TEG-u realiziranom u Pythonu (b) novom TEG-u realiziranom u C++-u

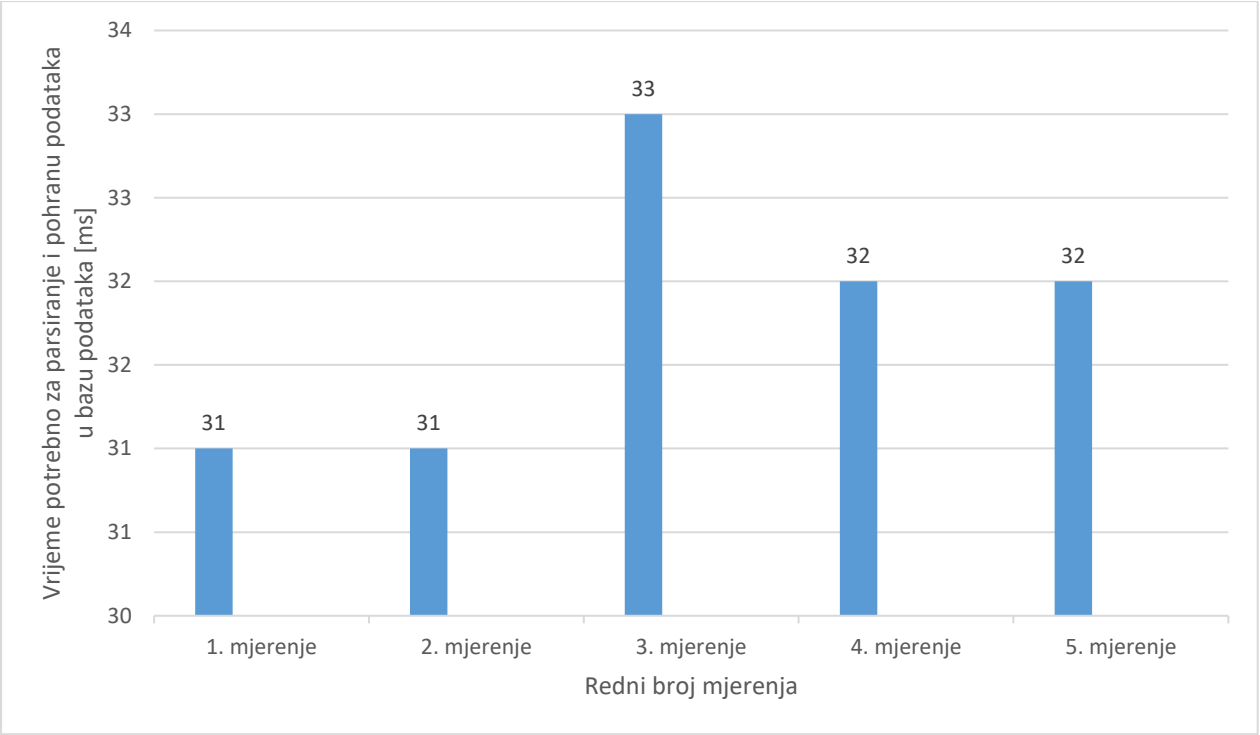


(a)

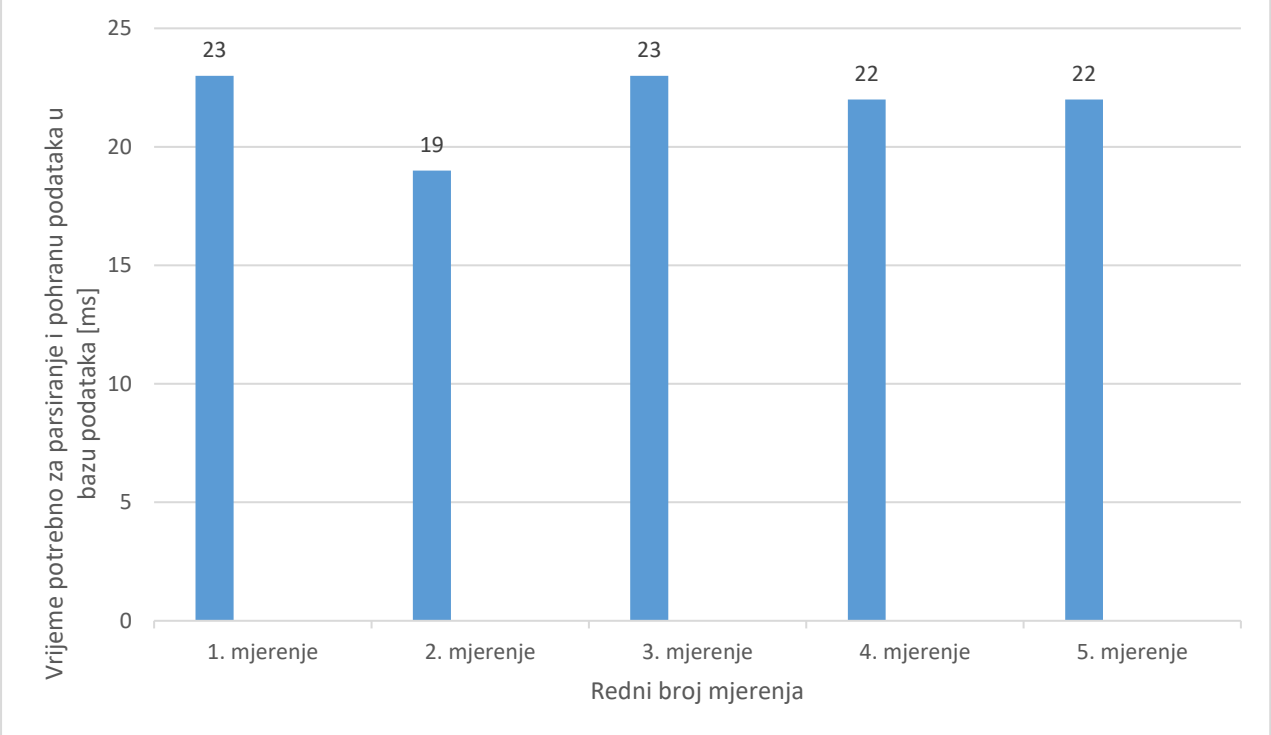


(b)

Slika 4.9. Usporedba vremena potrebnog za parsiranje i pohranu datoteke *TTADrive_Object.dbc* u (a) postojećem TEG-u realiziranom u Pythonu (b) novom TEG-u realiziranom u C++-u

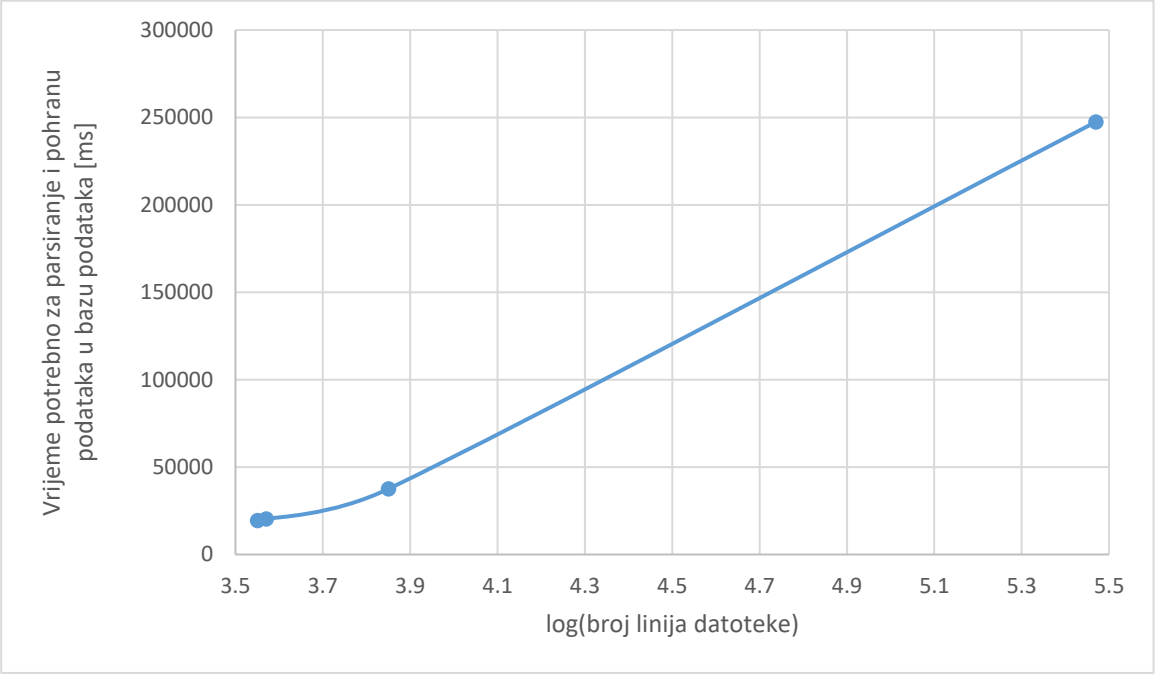


(a)

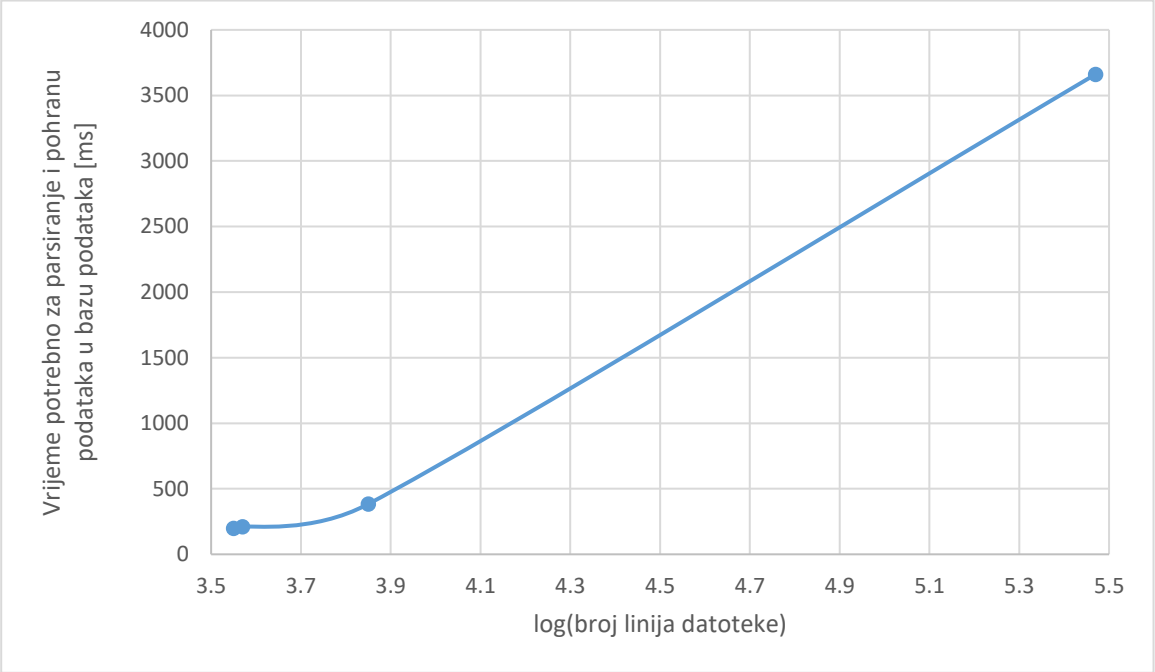


(b)

Slika 4.10. Usporedba vremena potrebnog za parsiranje i pohranu datoteke *schedule_config.xml* u (a) postojećem TEG-u realiziranom u Pythonu (b) novom TEG-u realiziranom u C++-u

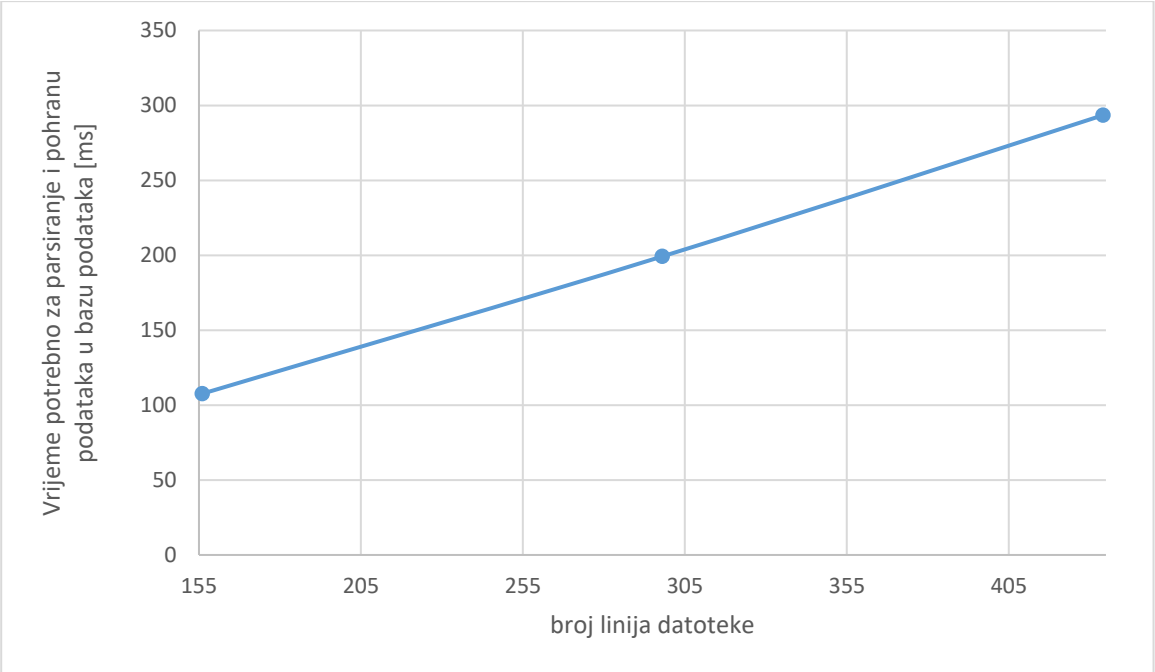


(a)

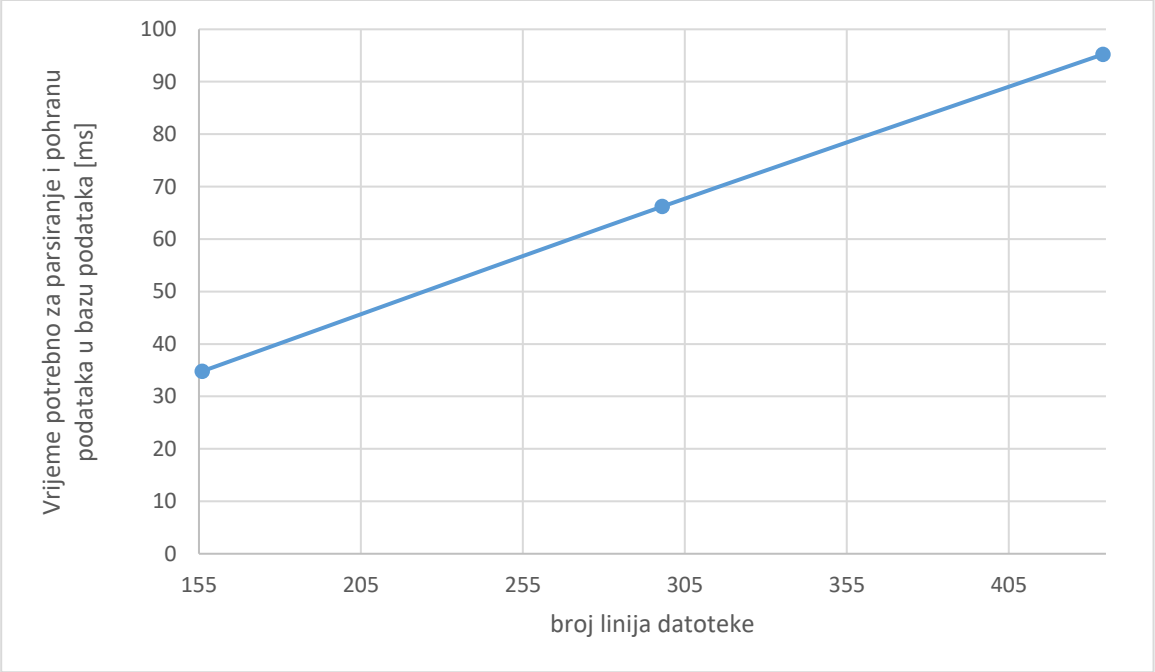


(b)

Slika 4.11. Ovisnost vremena potrebnog za parsiranje i pohranu datoteka tipa *.arxml* o broju linija datoteke u (a) postojećem TEG-u realiziranom u Pythonu (b) novom TEG-u realiziranom u C++-u



(a)



(b)

Slika 4.12. Ovisnost vremena potrebnog za parsiranje i pohranu datoteka tipa *.dbc* o broju linija datoteke u (a) postojećem TEG-u realiziranom u Pythonu (b) novom TEG-u realiziranom u C++-u

Tablica 4.2. Rezultati testiranja brzine svih konfiguracijskih datoteka u postojećem i novom rješenju TEG-a

DATOTEKA	SREDNJE VRIJEME POTREBNO ZA PARSIRANJE DATOTEKE I POHRANU U BAZU PODATAKA [ms]		MEDIJAN VREMENA POTREBNOG ZA PARSIRANJE DATOTEKE I POHRANE U BAZU PODATAKA [ms]	
	NOVO RJEŠENJE (C++)	POSTOJEĆE RJEŠENJE (PYTHON)	NOVO RJEŠENJE (C++)	POSTOJEĆE RJEŠENJE (PYTHON)
PerformanceHost00.arxml	3.661,2	247.446,0	3.639,0	247.702,0
TTADrive.arxml	199,0	19.371,8	200,0	19.388,0
TTADrive_Misc.arxml	383,6	37.504,4	382,0	37.446,0
TTADrive_Object.arxml	210,6	20.281,8	211,0	20.315,0
TTADrive.dbc	66,2	199,2	67,0	198,0
TTADrive_Misc.dbc	95,2	293,4	93,0	295,0
TTADrive_Object.dbc	34,8	107,6	35,0	108,0
schedule_config.xml	21,8	31,8	22,0	32,0

.

4.3. Verifikacija ispunjenosti zahtjeva novog rješenja rukovatelja bazom podataka

U skladu sa zahtjevima postavljenim prije izrade diplomskog rada, postojeće rješenje je unaprijeđeno i zahtjevi su u potpunosti zadovoljeni. Osim uklanjanja nedostataka postojećeg rješenja, dodane su nove funkcionalnosti poput algoritma detekcije promjena i serijalizacije baze podataka u formatu koji je razumljiv čovjeku. Potencijalna nadogradnja novom rukovatelju bazom podataka bila bi dodjela dodatnog atributa svakoj komponenti koji bi ih razlikovao po vrsti komponente koju predstavljaju. To bi omogućilo dodatan način pretraživanja baze podataka.

5. ZAKLJUČAK

Kako bi sklopovlje automobila bilo pouzdano i s potpunom se sigurnošću počelo koristiti u stvarnim automobilima, vrlo je važno provesti testove nad svom opremom. Prije negoli se počnu koristiti ECU-ovi u stvarnom automobilu, vrlo je važno provjeriti je li komunikacija među njima ispravna. Upravo se za tu svrhu koristi TEG.

Cilj diplomskog rada bio je izraditi programsko rješenje rukovatelja baze podataka pomoću kojeg će generatori izdvajati podatke potrebne za generiranje izvornoga koda. Poželjno je da novi rukovatelj baze podataka ima bolje performanse od rukovatelja baze podataka postojećeg rješenja dobivenog na raspolaganje prije početka izrade rada te da ukloni nedostatke koji su prisutni u postojećem rješenju.

Izrađeno novo rješenje rukovatelja baze podataka nadvladava postojeće rješenje u performansama, odnosno potrebno je manje vremena za izvršavanje istog zadatka. Novom vrstom serijalizacije baze podataka uklonjena je opasnost od potencijalnih napada korisnika s malicioznim namjerama koji su bili prisutni u postojećem rješenju. Konačno, novo programsko rješenje sadrži u sebi sve funkcionalnosti koje su bile definirane u zahtjevima prije izrade samog rada. Novo rješenje implementirano je u programskom jeziku C++. Moguće je dohvatiti sve potrebne podatke iz baze podataka te je bazu moguće pohraniti u formatu u kojem će ju čovjek moću pročitati s razumijevanjem. Kako bi novo pokretanje programa bilo brže, razvijen je algoritam detekcije promjena u konfiguracijskim datotekama pomoću kojeg je moguće izbjeći parsiranje već isparsiranih podataka. Provedenim testovima potvrđeno je da spomenute funkcionalnosti obavljaju svoju zadaću kako je i propisano u zahtjevima.

LITERATURA

[1] W. Mostowski, T. Arts, J. Hughes, „Modelling of Autosar Libraries for Large Scale Testing“, EPTCS, Gothenburg , Švedska, 2017.

[2] A. Vega, P. Bose A. Buyuktosunoglu, „Rugged Embedded Systems“, Elsevier Inc., 8. poglavlje, Cambridge, Massachusetts, 2017.

[3] AUTOSAR Fundamentals, https://www.autonom.com.tr/en/autosar-fundamentals/, pristupljeno 2021.

[4] AUTOSAR XML File, https://fileinfo.com/extension/arxml, pristupljeno 2021.

[5] About Python, https://www.python.org/about/, pristupljeno 2021.

[6] Sunsetting Python 2, https://www.python.org/doc/sunset-python-2/, pristupljeno 2021.

[7] How Fast Is C++ Compared to Python?, https://towardsdatascience.com/how-fast-is-c-compared-to-python-978f18f474c7, pristupljeno 2021.

[8] Concealing ("Protecting") Source Code, https://wiki.python.org/moin/Asking%20for%20Help/How%20do%20you%20protect%20Python%20source%20code%3F, pristupljeno 2021.

[9] pickle – Python object serialization, https://docs.python.org/3/library/pickle.html, pristupljeno 2021.

[10] Python Pickle is Notoriously Insecure, https://medium.com/ochrona/python-pickle-is-notoriously-insecure-d6651f1974c9, pristupljeno 2021.

[11] B. Stroustrup, „The C++ programming language (3. izdanje)“, Addison-Wesley, Boston, Massachusetts, 1997.

[12] B. Stroustrup, „Lecture:The essence of C++“, University of Edinburgh, Edinburg, Škotska, 2014.

[13] C++ Applications, https://www.stroustrup.com/applications.html, pristupljeno 2021.

[14] Programming languages – C++, https://www.iso.org/standard/79358.html, pristupljeno 2021.

[15] Visual Studio, https://www.incredibuild.com/integrations/visual-studio, pristupljeno 2021.

[16] What is Hierarchical Data?, https://www.tibco.com/reference-center/what-is-hierarchical-data, pristupljeno 2021.

[17] What is a Hierarchical Database?, https://www.omnisci.com/technical-glossary/hierarchical-database, pristupljeno 2021.

[18] What is a Non-Relational Database?, https://www.mongodb.com/non-relational-database, pristupljeno 2021.

[19] C++ 11, https://en.cppreference.com/w/cpp/11, pristupljeno 2021.

[20] Smart pointers (Modern C++), https://docs.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=msvc-160, pristupljeno 2021.

[21] C++ named requirements: *Compare*, https://en.cppreference.com/w/cpp/named_req/Compare, pristupljeno 2021.

[22] std::map, https://en.cppreference.com/w/cpp/container/map, pristupljeno 2021.

[23] T. H. Cormen, C. E. Leiserson, L. Ronald, C. Stein, „Introduction to Algorithms (2. izdanje)“, MIT Press, stranice 273. – 301., Cambridge, Massachusetts, 2001.

[24] Data Structure & Algorithms - Tree Traversal, https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal.htm, pristupljeno 2021.

[25] Null pointers, https://www.ibm.com/docs/en/i/7.3?topic=pointers-null, pristupljeno 2021.

[26] std::filesystem::last_write_time, https://en.cppreference.com/w/cpp/filesystem/last_write_time, pristupljeno 2021.

[27] Python vs C++: What’s the Difference?, https://www.guru99.com/python-vs-c-plus-plus.html#9, pristupljeno 2021.

SAŽETAK

U svrhu testiranja komunikacije među dijelovima sklopovlja automobila, definirana je potreba za izradom programskog rješenja koje će automatski vršiti zatraženo testiranje. Uz izradu spomenutog rješenja, definirana je i potreba da rješenje ukloni nedostatke dostavljenog rješenja za generiranje testnog okruženja te da bude znatno bolje u vidu performansi. Novo rješenje je ispunilo sljedeće zahtjeve: implementacija rješenja u programskom jeziku C++, mogućnost dostavljanja podataka generatorima, serijalizacija baze podataka i implementacija algoritma za detekciju promjena. Kako bi se provjerila ispravnost rada novog rješenja i usporedile performanse s onima od postojećeg rješenja, provedeni su prikladni testovi koji potvrđuju da novo rješenje ima funkcionalnosti propisane zahtjevima te da se izvodi brže od postojećeg rješenja.

Ključne riječi: automatsko testiranje, generator testnog okruženja, baza podataka

DEVELOPMENT OF ATEST ENVIRONMENT GENERATOR WITH A FOCUS ON DATABASE HANDLING

ABSTRACT

For the purpose of testing the communication between the parts of the car assemblies, the need for the development of software solution that will automatically perform the requested testing has been requested. In addition to the development of the described solutions, the need to eliminate the shortcomings of the delivered solution in order to optimize it in terms of performance was also requested. The new solution met the following requirements: implementation of the solution in the C ++ programming language, the possibility of delivering data to generators, serialization of the database and implementation of the algorithm for detecting changes. In order to verify the correct behaviour of the new solution and compare the performance with the existing solution, various tests were conducted to confirm that the new solution has all functionalities of the prescribed requirements and is performing better than the existing solution.

Keywords: automatic testing, test environment generator, database

ŽIVOTOPIS

Ivan Kožul rođen je u Osijeku 1. rujna 1997. godine. Pohađao je Isusovačku klasičnu gimnaziju s pravom javnosti u Osijeku nakon koje je upisao preddiplomski studij računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. Preddiplomski je studij uspješno završio 2019. godine te nakon toga upisuje diplomski studij računarstva, smjer robotika i umjetna inteligencija.

_____

Potpis autora