Razvoj generatora CAPL koda u sklopu generatora testnog okruženja

Kostić, Abraham

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

Sveučilišni diplomski studij

Razvoj generatora CAPL koda u sklopu generatora testnog okruženja

Diplomski rad

Abraham Kostić

Osijek, 2021.

SADRŽAJ

1. UVOD

Automobilska industrija svakim danom sve više napreduje. Danas su se u automobilima počele koristiti visoke tehnologije, ponajviše u sigurnosti i novim funkcijama automobila koje prije nisu postojale (npr. zračni jastuci, detektor objekta u mrtvom kutu, pomoć za ostajanje u svojoj prometnoj traci (engl. *lane assists*) i sl.), ali i u udobnosti vozila. Jedan od primjera je i tzv. *Infotainment* sustav, sustav koji se koristi za prikupljanje informacija (navigacija, trenutna brzina, temperatura unutar i izvan vozila) i za zabavni sadržaj (glazba, glasnoća, ambijentalno osvjetljenje i sl.). Kako bi svi navedeni sustavi mogli funkcionirati, potrebna je komunikacija između upravljačkih jedinica (engl. *Electronic Control Unit*, skraćeno ECU) svakog podsustava automobila, koja se odvija putem komunikacijskih kanala. Ukupan broj upravljačkih jedinica, komponenti sustava i komunikacijskih kanala unutar modernih automobila može biti od nekoliko tisuća do nekoliko desetaka tisuća. Svaki ECU, svaka komponenta i svaki komunikacijski kanal moraju biti testirani prije konkretne upotrebe u automobilskoj industriji. Generator testnog okruženja (engl. *Test Environment Generator*, skraćeno TEG) je upravo način za testiranje navedenih elemenata komunikacije. TEG simulira testno okruženje te testira virtualne komponente koje ostvaruju komunikaciju, kao i same komunikacijske kanale putem kojih se ta ista komunikacija odvija. Na taj način nije potreban fizički sustav (npr. automobil) na kojem će se testirati upravljačke jedinice i komunikacija, već je to moguće napraviti prije stavljanja u navedeni fizički sustav, odnosno automobil.

Ovaj diplomski rad podijeljen je u pet poglavlja. Prvo poglavlje predstavlja uvod, a peto poglavlje zaključak. U drugom poglavlju opisan je način rada postojećeg TEG-a i navedeni su njegovi nedostaci. Nadalje, u trećem poglavlju predstavljeno je novo rješenje CAPL (engl. ***C**ommunication **A**ccess **P**rogramming **L**anguage*) generatora i generatora lista testnih slučajeva (engl. *TestCase lists*), dok je u četvrtom poglavlju napravljena usporedba između novog i postojećeg rješenja.

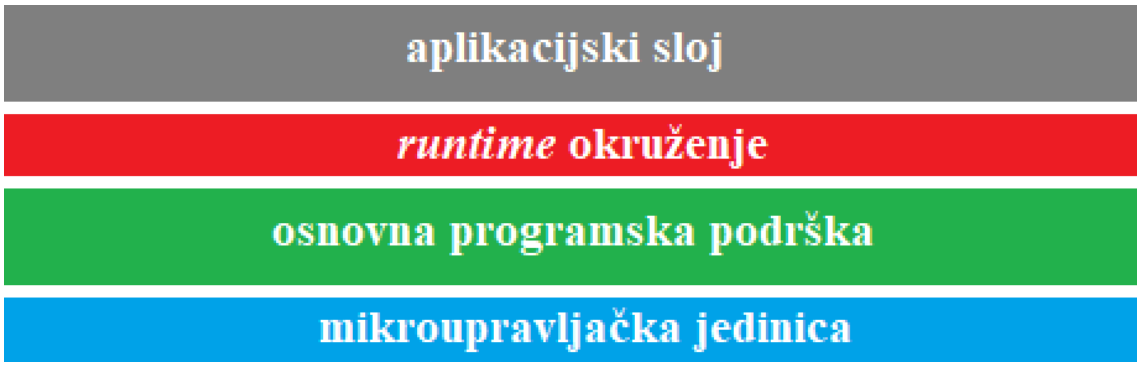
2. POSTOJEĆI GENERATOR TESTNOG OKRUŽENJA

Postojeći generator testnog okruženja je alat za testiranje komunikacije unutar automobilskog sustava koristeći standard AUTOSAR (engl. ***AUT****omotive* ***O****pen* ***S****ystem* ***AR****chitecture*). TEG je razvijen u programskom jeziku *Python* (verzija 2.7). Vlasništvo je kompanije *TTTech* koja ga je ustupila za izradu ovog diplomskog rada.

2.1. AUTOSAR standard

AUTOSAR je standard koji predstavlja međunarodni dogovor između kompanija u automobilskoj industriji. Svrha uvođenja takvog dogovora je standardiziranje u stvaranju sklopovlja i programske podrške. Uspostavljanjem standarda ostvarena je mogućnost korištenja programske podrške na bilo kojem sklopovlju, bez obzira na proizvođača. Prema [1], AUTOSAR ima hijerarhijsku arhitekturu:

- mikroupravljačka jedinica (engl. *microcontroller unit*)
- osnovna programska podrška (engl. *basic software*) – skup standardizirane programske podrške koja sadrži usluge potrebne za funkcioniranje aplikacijskog sloja
- *RTE* (engl. *runtime environment*) – posrednički sloj koji služi za komunikaciju između osnovne programske podrške i aplikacijskog sloja
- aplikacijski sloj (engl. *application layer*) – komponente koje komuniciraju s posredničkim slojem.



Sl. 2.1. AUTOSAR arhitektura [1].

2.2. Postojeći generator testnog okruženja

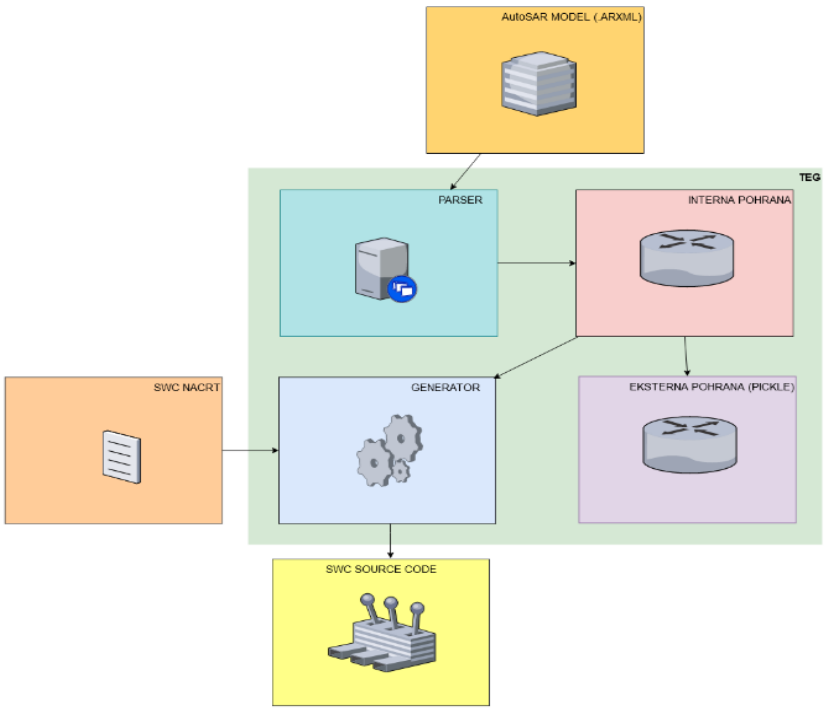
U AUTOSAR arhitekturi, prema slici 2.1., TEG se nalazi u posredničkom (*runtime*) sloju. Na ulazu prikuplja podatke o ECU te na temelju istih generira model za testiranje i simulaciju. Promjenom bilo kojih postavki upravljačke jedinice (primjena različitih konfiguracija), mogu

se provesti različita testiranja prema AUTOSAR standardu. Svaki proizvođač automobila će, prema svojim potrebama, koristiti drugačiju konfiguraciju te će TEG generirati različite modele za testiranje. Konfiguracija svakog modela sprema se u datoteku formata *.arxml*. ARXML (***AUTOSAR XML***) je posebna vrsta XML (engl. ***E**xtensible **M**arkup **L**anguage*) datoteke koja se koristi u razvoju programske podrške za automobilsku industriju. Pod konfiguraciju se podrazumijevaju svi jednostavni (*int*, *float*) i složeni (klase, strukture) tipovi podataka. Oni se raščlanjuju u cjeline i na taj način pohranjuju. U TEG-u se takva datoteka kasnije parsira i sprema u novu, jednako složenu, ali smisleniju (čitljiviju ljudskim okom, engl. *human-readable*) bazu podataka. To se radi zbog lakšeg čitanja i pisanja unutar te iste baze podataka, koja će se koristiti kao izvor podataka potrebnih za daljnje testiranje [2].

2.3. Arhitektura TEG-a

Prema [2], postojeći TEG se sastoji od tri osnovna dijela:

1. parser ulaznih podataka
2. pohrana podataka
3. generatori izvornog koda



Sl. 2.2. Arhitektura generatora testnog okruženja [2].

Svaki od spomenutih dijelova je podijeljen na manje zadatke, a jedan od podzadataka trećeg dijela, generatora izvornog koda, je ovaj diplomski rad koji se bavi generiranjem CAPL koda i generiranjem lista testnih slučajeva. Na slici 2.2. prikazan je pojednostavljeni izgled trenutne arhitekture generatora testnog okruženja. Osim navedena tri glavna dijela, u arhitekturi se još nalaze ranije spomenuta ARXML datoteka, koja sadrži podatke koji se parsiraju, eksterna pohrana u koju će rukovatelj bazom zapisivati podatke koji se traže od njega (zahtjev za dohvaćanjem) te „SWC nacrt“. On predstavlja oblik u kojem će generator .c koda zapisati podatke o svakoj programskoj komponenti (engl. ***S**oftware **C**omponent*, skraćeno SWC), odnosno služi kao osnovni predložak za generiranje .c koda o programskim komponentama.

2.3.1. Parser ulaznih podataka

Prvi korak TEG-a je parsirati konfiguracijske datoteke i izdvojiti bitne informacije za konkretnu upravljačku jedinicu. Kao što je ranije navedeno, konfiguracije su spremljene u *.arxml* formatu. Parser će proći kroz *.arxml* datoteku, izdvojiti podatke i presložiti ih (npr. po komponentama, po komunikacijskim kanalima, sučeljima i sl.) kako bi se mogli zapisati u novu strukturu (bazu). Datoteka sadrži podatke kao što su konfiguracija upravljačke jedinice, imena komponenti prisutnih u sustavu, imena komunikacijskih kanala, kao i podatke koji se nastavljaju na svaku od komponenti (npr. vrijeme potrebno za prijenos poruke između dvije komponente, sučelje preko kojeg se odvija komunikacija ili sl.).

2.3.2. Pohrana podataka

Podaci koje je parser izdvojio iz *.arxml* datoteke pohranjuju se, na zahtjev, u bazu podataka. Nova struktura je lakša za čitanje u odnosu na *.arxml* datoteku. Takva se baza koristi u daljnjem radu TEG-a, ponajviše u generatorima izvornog (engl. *source*) koda, koji potom čitaju potrebne podatke, vrše operacije nad njima te ih spremaju u neku od datoteka, kao što je lista testnih slučajeva.

2.3.3. Generatori izvornog koda

U ovom dijelu generatora testnog okruženja nalazi se generator izvornog koda za svaku programsku komponentu, CAPL generator te generator *TestCase* listi. Druga dva generatora tema su ovog diplomskog rada. Postojeća baza podataka ima nekoliko testnih *buildova* te svaki od njih u sebi sadrži nekoliko testnih grupa. Nova baza podataka u sebi trenutačno sadrži samo jedan testni *build*, dok su u budućnosti moguće dopune. Postojeći *build*, skraćenog naziva ITF, koje predstavlja sučelje posredničkog sloja, u postojećem TEG-u sadrži nekoliko testnih grupa:

- testnu grupu pošiljatelj – primatelj (engl. *sender-receiver,* pa se grupa onda označava ITF-SendRec),
- testnu grupu pisač – čitač (engl. *writer-reader,* oznake ITF-WriteRead),
- testnu grupu velikih podataka (engl. *big data,* oznake ITF-BigData),
- testnu grupu skupa podataka (engl. *data set* oznake ITF-DataSet) itd.

Novo rješenje generatora će se verificirati na istom testnom *buildu*, ITF-u, te na jednoj testnoj grupi unutar *builda,* grupi pošiljatelj-primatelj.

2.4. Nedostaci postojećeg generatora testnog okruženja

Prvi i osnovni nedostatak postojećeg TEG-a je taj što je pisan u *Pythonu* 2.7. *Python* 2.0 je programski jezik objavljen 2000. godine. Nedugo nakon toga su kreatori *Pythona* 2.0, kompanija *Python Software Foundation*, na čelu s dizajnerom Guidom van Rossumom, uvidjeli su puno nedostataka verzije 2.0, ali i načina na koji ga mogu poboljšati. Vođeni željom za kreiranje verzije s boljom učinkovitošću, 2006. godine predstavljaju *Python* 3.0, koji se tada minorno razlikovao od verzije 2.0. Bez obzira na takve minimalne razlike u programskom jeziku, ukoliko se htjelo preći na verziju 3.0, bilo je potrebno mijenjati većinu koda. Veliki broj korisnika nije preveo kod na novu verziju, već je nastavio koristiti staru, što je spriječilo brzu tranziciju s verzije 2.0 na verziju 3.0. Godinama su kreatori nastavljali poboljšavati i objavljivati nadogradnje za oba *Pythona*, dok nisu uvidjeli da to koči stvarni napredak ovog programskog jezika, jer su trošili vrijeme i resurse na obje verzije, umjesto da se fokusiraju na jednu. Zbog toga su 1. siječnja 2020. godine odlučili stati na kraj izdavanju nadogradnji za sve verzije *Pythona* 2 [3]. Takav događaj je loše utjecao na postojeći generator testnog okruženja jer *Python* 2 više nema podršku, korištene funkcije i biblioteke neće biti nadograđivane, postojeći ili budući *bugovi* i greške neće biti otklonjeni te potrebne izmjene neće biti napravljene. Uzmemo li u obzir prethodno navedene stavke, postojeći generator nema mogućnost korištenja novo osmišljene tehnologije u vidu funkcija i biblioteka, kao niti unaprijeđene postojeće biblioteke, a koje su sve dostupne za korištenje u *Pythonu* 3. S obzirom da je *Python* kao programski jezik općenito jezik visoke razine, čitljiviji je ljudskom oku i lakši za upotrebu, ali ima duže vrijeme izvođenja za razliku od jezika niskih ili srednjih razina kao što su programski jezici C i C++. Kao sljedeće unaprjeđenje TEG-a odabrano je pisanje u drugom programskom jeziku, C++.

Trenutni generatori nemaju izričitih mana, već se nedostaci pojavljuju ranije u dijelovima TEG-a koje generatori koriste. Odnosi se na vrijeme izvođenja parsera, baze podataka i rukovatelja

bazom podataka koji su u sklopu novog rješenja morali biti redefinirani i ponovo implementirani, kako bi se ubrzali i enkapsulirali. S druge strane, generatori ostaju iste arhitekture, ali koriste novi rukovatelj bazom, koji koristi novi parser.

Jedna od mana TEG-a je njegova otvorenost. TEG je prije svega vlasništvo tvrtke TTTech te ga kao takvim smatramo proizvodom. Pri isporuci TEG-a klijentu, u *Python* 2.7. verziji, klijent izvršava program prevođenjem izvornog koda. Na taj način, izvorni kod je otvoren, vidljiv svima koji mu imaju pristup. Takvo što nije prihvatljivo kada je u pitanju proizvod koji se želi komercijalizirati. Potrebno je načiniti proizvod koji će biti enkapsuliran (učahuren) tako što će klijent moći koristiti samo krajnji proizvod, odnosno izvoditi program gdje mu izvorni kod neće biti javno dostupan. C++ je programski jezik koji to omogućava.

3. IZRADA CAPL GENERATORA I GENERATORA TESTCASE LISTA

CAPL generator i *TestCase* generator su funkcionalno vrlo slični postojećim, dok je najveća razlika to što su pisani u drugom programskom jeziku, C++. Njihov rad u postojećem TEG-u je direktan (engl. *straight-forward*), generiraju točno određene datoteke, pa su eventualne nadogradnje skoro pa nepostojeće u vidu funkcionalnosti.

Osim zahtjeva pisanja u programskom jeziku C++, generatori imaju sljedeće zahtjeve:

1. koristeći postojeći rukovatelj bazom (engl. *database handler*), dohvatiti potrebne podatke (komponente, sučelja) iz postojeće baze podataka,

2. filtrirati dohvaćene podatke (komponente, sučelja) i izdvojiti one na temelju kojih će se generirati testni slučajevi i CAPL kod,

3. generirati (zapisati u datoteku) ih u istom obliku kao što generiraju postojeći generatori pisani u *Pythonu*, kako bi se održao format.

3.1. Korištene tehnologije u razvoju CAPL generatora i generatora *TestCase* lista

U ovom poglavlju opisani su programski jezici korišteni za izradu oba generatora te razvojno okruženje koje je korišteno.

3.1.1. C++

C++ je programski jezik opće namjene koji je stvoren kao proširenje programskog jezika C. Njegovo razvijanje Bjarne Stroustrup započeo je 1979. godine, a razvija se i danas [4]. Za razliku od C-a, C++ podržava objektno orijentirano programiranje (OOP), što znači da mu je osnovna jedinica objekt – instanca klase. C je jezik visokih performansi, izvodi se puno brže u odnosu na *Python* te zbog toga C++ sadrži slične performanse. C++ ima sljedeća osnovna obilježja:

- enkapsulacija,
- nasljeđivanje,
- polimorfizam.

Enkapsulacija (čahurenje) predstavlja pojam koji označava skrivanje članova i metoda jedne klase. Članovi mogu biti podatkovni, a mogu biti i funkcijski. Skriveni članovi se u OOP-u nazivaju privatnima (engl. *private*), a mogu se definirati i javni članovi (engl. *public*).

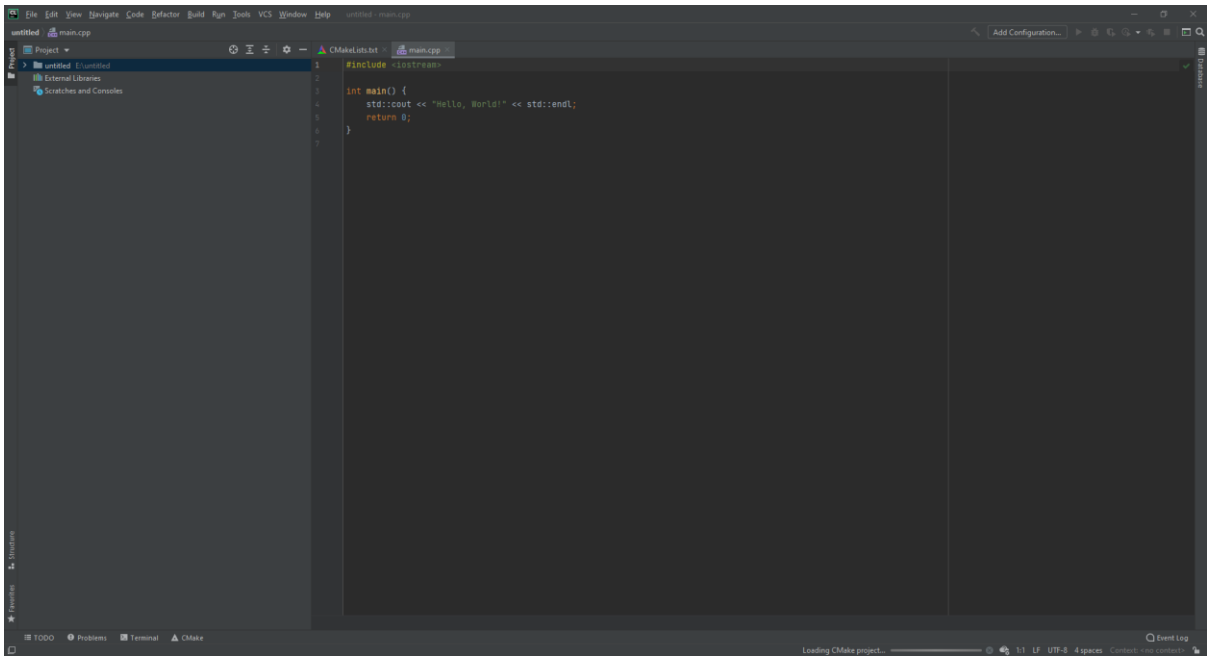
Nasljeđivanje je svojstvo klase da bude definirana iz neke postojeće klase, odnosno da nasljedi već postojeće članove, podatkovne i funkcijske, a da se novi članovi dodaju.
Polimorfizam kao općeniti pojam označava svojstvo oblika da bude promijenjen. U sklopu OOP-a ono znači da jedan naziv funkcije može poprimiti više oblika funkcije, odnosno više različitih funkcija se može jednako zvati. Tada se te funkcije razlikuju u broju parametara, u povratnom tipu i/ili tijelu funkcije.
C++ je zbog svojih svojstava najčešće korišten u ugradbenim sustavima i u alatima kao što je generator testnog okruženja. Također je jezik koji se prevodi izravno u strojni kod, što mu omogućava brzo izvođenje programa te je kao takav idealan za korištenje u ugradbenim sustavima [5].

3.1.2. CAPL

CAPL je proceduralni programski jezik baziran na C/C++ programskim jezicima te mu je sintaksa vrlo slična navedenima. Koristi se u razvoju automatskih testova i simulacija u ugradbenim sustavima u automobilskoj industriji [6]. Zapisuje se u datoteku *.cin* formata, osnovni format datoteke za alat *CANoe*, razvijen od strane tvrtke *Vector Informatik*, a koristi za testiranje u automobilskoj industriji. *CANoe* vraća povratnu vrijednost nakon pokrenutog testa (prolaz/pad, engl. *pass/fail*) [7]. U sklopu ovog rada nije rađeno s alatom *CANoe*, već su generatori priprema za testiranje programskih komponenti (SWC-ova) na ploči.

3.1.3. CLion

CLion je višeplatformsko razvojno okruženje (engl. *Integrated Development Environment,* skraćeno IDE) korišteno ponajviše za razvoj projekata u programskim jezicima C i C++. Višeplatformsko je jer se može koristiti na *Linux* distribucijama, *Windows* operacijskim sustavima i *MacOS-u*. *CLion* podržava prelazak na deklaracije i korištenja varijabli i funkcija unutar koda, odnosno ranije definiranim prečacima moguće je preskočiti s mjesta gdje se koristi funkcija na mjesto gdje je implementirana. Također, omogućava automatsko generiranje preopterećenih operatora, konstruktora i destruktora. Sadržava velik broj alata za analizu koda i automatsko prerađivanje koda, od kojih je najpoznatiji *Clang-Tidy*, alat koji prepoznaje mogućnost poboljšanja određene funkcije (efikasnije, čitljivije, korisnije), primjerice korištenje referenci gdje je moguće [8]. Na slici 3.1. vidljiv je izgled razvojnog okruženja *CLion* kod otvaranja novog, praznog projekta.



Sl. 3.1. Izgled programskog okruženja *CLion*.

3.2. Programsko rješenje CAPL generatora

Zadatak ovog dijela diplomskog rada je razviti generator koji će pomoću funkcija implementiranih u C++ programskom jeziku, zapisivati CAPL kod u datoteku željenog formata. Na postojećem primjeru generirane datoteke, iz starog TEG-a, generirat će se ista takva datoteka kako bi se usporedio format i uspješnost generatorskih funkcija. Odabrani format je *.cin* datoteka, osnovni format datoteke koja se koristi u alatu *CANoe* za pokretanje testova. Podaci su preuzeti iz primjera generirane datoteke starog TEG-a, koja sadrži sve potrebne varijable, konstante, strukture i ostale tipove podataka koji su potrebni za buduće izvođenje testova. Bitno je usporediti format zapisanog CAPL koda s postojećim te ukoliko je format održan, moguće je generator u budućnosti koristiti za generiranje CAPL koda po potrebi.

CAPL kod ima sličnu sintaksu kao C/C++ programski jezici te je potrebno da ukoliko se, primjerice, pozove funkcija za zapis strukture u datoteku, ta struktura bude istog formata kao da je pisana u CAPL kodu. Na slici 3.2. vidimo da je sličnost između dva jezika velika, ali da se razlikuje u sintaksnim pravilima, ponajviše u tipovima podataka. U četvrtom poglavlju usporedbom s datotekom generiranom u starom TEG-u, verificirat će se rješenje, a ovdje će se prezentirati rezultat pozivanja funkcija.

```
struct CommonTestData
{
  byte                  Usage;
  byte                  Info;
  byte                  Flags;
  byte                  HostId;
  word                  SwcId;
  word                  RunnableNum;
  word                  ITFgroup;
};
```

(a)

```
typedef struct
{
    uint8               Usage;
    uint8               Info;
    uint8               Flags;
    uint8               HostId;
    uint16              SwcId;
    uint16              RunnableNum;
    uint16              ITFgroup;
}
CommonTestData;
```

(b)

Sl. 3.2. Usporedba strukture pisane u CAPL (a) i C programskom jeziku (b).

Na slici 3.3. vidljivo je da sve funkcije generatora imaju prvi argument isti, a on predstavlja datoteku u koju će se generirati kod. Po potrebi se datoteka u koju se zapisuje željeni CAPL kod vrlo lako mijenja predajom neke druge datoteke kao argumenta.

```
void WriteInfo(std::ofstream &);
void WriteStart(std::ofstream &);
void Write(std::ofstream &, std::string);
void WriteComment(std::ofstream &, std::string);
void WriteConst(std::ofstream &, std::string , std::string, std::string);
void WriteEnum(std::ofstream &, std::string, std::vector<std::string>);
void WriteStruct(std::ofstream &, std::string, std::vector<std::pair<std::string,std::string>>);
void WriteChar(std::ofstream &, std::string, int, int, std::vector<std::string>);
void WriteFunction(std::ofstream &, std::string);
```

Sl. 3.3. Prototipi funkcija za generiranje CAPL koda.

Prva u nizu funkcija je *WriteInfo()*. Prima jedan argument, a to je naziv datoteke, tipa *std::ofstream*, koji predstavlja postojeću klasu izlaznog toka za rad s datotekama. Izlaznog toka znači da se u nju zapisuje, dok postoji još i *std::ifstream* za čitanje iz datoteke. Funkcija se koristi najčešće jednom, na početku programa, za zapisivanje osnovnih informacija o programu. Po potrebi se taj zapis može promijeniti. Poziv funkcije se vidi na slici 3.4.

```
std::ofstream CAPL_cin( s: R"(CAPL.cin)");

WriteInfo( &: CAPL_cin);
```

Sl. 3.4. Poziv funkcije *WriteInfo()*.

Na slici 3.5. se nalazi rezultat pozivanja funkcije *WriteInfo()*. Ispisane su konstante tipa *byte* koje predstavljaju verzioniranje TEG-a te jedna konstanta tipa *word* koja predstavlja zapis pune verzije TEG-a.

```
// current TEG version
const byte              TEGversionMajor         = 1;
const byte              TEGversionMinor         = 0;
const byte              TEGversionPatch         = 0;

//============================================================================
// Hardware

const word              TEGversion              = 1.0.0.;
```

Sl. 3.5. Rezultat poziva funkcije *WriteInfo()*.

Ova funkcija u svom tijelu poziva druge funkcije sa slike 3.3., *Write()*, *WriteComment()* i *WriteConst()*. Prva u nizu, *Write()*, omogućava ispis željenog niza znakova u datoteku po želji. Općenitija je funkcija od ostalih, koristi se za pisanje „bilo čega“ u datoteku, te služi kao pomoćna funkcija za popunjavanje komentarima, dodavanjima novog reda i sl. Na slici 3.6. nalazi se poziv navedene funkcije, a na slici 3.7. rezultat njenog pozivanja.

```
std::ofstream CAPL_cin( s: R"(CAPL.cin)");

Write( &: CAPL_cin,  word: "// created by  TEG-cpp");
```

Sl. 3.6. Poziv funkcije *Write()*.

CAPL.cin

```
// created by  TEG-cpp
```

Sl. 3.7. Rezultat pozivanja funkcije *Write()*.

Sljedeća korištena funkcija je jednostavni zapis komentara u datoteku, koji za razliku od funkcije *Write()* automatski dodaje oznake „//“ na početak željenog ispis. U pozivu funkcije (slika 3.8.) kao argument se predaje datoteka u koju želimo zapisati komentar te željeni komentar. Ispis u datoteci vidljiv je na slici 3.9.

```
WriteComment( &CAPL_cin, comment: " current TEG version");
```

Sl. 3.8. Poziv funkcije *WriteComment()*.

```
variables
{
  //============================================================================

  // current TEG version
  const byte              TEGversionMajor       = 1;
  const byte              TEGversionMinor       = 0;
  const byte              TEGversionPatch       = 0;
```

Sl. 3.9. Rezultat funkcije *WriteComment()*.

Treća funkcija korištena u tijelu prve funkcije, *WriteInfo()*, naziva se *WriteConst()*. Ona u željenu datoteku zapisuje konstantu koja prati sintaksu CAPL programskog jezika. Za argumente prima datoteku, podatkovni tip konstante, naziv konstante i njenu vrijednost. Tip podatka koju funkcija prima kao vrijednost konstante je tipa *string*, radi poopćavanja funkcije. Konstanta byte će primati cjelobrojnu vrijednost, dok će konstanta word primati *string*. Dodaje potrebne razmake, nove redove i tab-ove kako bi se zadržao format CAPL koda. Drugi argument koji funkcija prima je tip podatka, a na slici 3.10. prikazan je poziv ispisa konstanti tipa byte. Na slici 3.11. prikazan je ispis u datoteci.

```
WriteConst( &CAPL_cin, "byte",  "TEGversionMajor", "1");
WriteConst( &CAPL_cin, "byte",  "TEGversionMinor", "0");
WriteConst( &CAPL_cin, "byte",  "TEGversionPatch", "0");
```

Sl. 3.10. Poziv funkcije *WriteConst()*.

```
// current TEG version
const byte              TEGversionMajor       = 1;
const byte              TEGversionMinor       = 0;
const byte              TEGversionPatch       = 0;
```

Sl. 3.11. Rezultat poziva funkcije *WriteConst()*.

Sljedeća funkcija koju program koristi je *WriteStart()*. Jednostavna funkcija koja u sebi sadrži pozivanje funkcije *WriteInfo()*, za ispis potrebnih informacija, te poziv funkcije *Write()*, koja će ispisati ključnu riječ *variables* i znak *„{“*, kako bi bio moguć zapis varijabli. Na kraju ispisivanja varijabli je potrebno, koristeći *Write*() funkciju ponovo, ispisati znak *„}“* koji će označiti kraj sekcije s varijablama.

Nadalje, slijedi funkcija *WriteEnum()*. Funkcija ispisuje željeni *enum* u datoteku. Kao argumente prima izlazni tok datoteke, naziv *enuma* te njegove članove. Zadnji argument je vektor *stringova*, a predstavlja vektor naziva članova *enuma*, u proizvoljnoj duljini. Može sadržavati jedan, dva ili više *stringova*. Na slici 3.12. poziva se funkcija *WriteEnum()*.

```
WriteEnum( & CAPL_cin,  nameOfEnum: "Hosts", hosts);
```

Sl. 3.12. Poziv funkcije *WriteEnum()*.

Na slici 3.13. nalazi se zadnji argument u primjeru poziva, koji predstavlja vektor *stringova* naziva *hosts*. Vidljivo je da je vrlo lako dodati nove *hostove* upisivanjem novih nizova znakova u vektor te će generator pri sljedećem pozivanju funkcije ili ponovnim pokretanjem generatora ispisati još jedan ili više *hostova*.

```
std::vector<std::string> hosts {
        "SH00", "PH00"
};
```

Sl. 3.13. Vektor nizova znakova naziva *hosts*.

Funkcija iterira po vektoru, ispisuje svaki niz znakova te dodaje zarez i novi red na kraju ispisa svake varijable. Kad je funkcija došla do kraja, ispisuje znak „}“ i znak „;“ . Tada ispis datoteke izgleda kao na slici 3.14.

```
enum Hosts
{
  SH00,
  PH00
};
```

Sl. 3.14. Rezultat poziva funkcije *WriteEnum()*.

Iduća u nizu funkcija je funkcija pod nazivom *WriteStruct()*. U izlazni tok datoteke šalje ispis strukture koja prati sintaksu CAPL programskog jezika. Pri pozivu kao argumente prima još i naziv strukture te vektor parova nizova znakova. Poziv funkcije je vidljiv na slici 3.15., a ispis u datoteku na slici 3.17.

```
WriteStruct( &: CAPL_cin,  nameOfStruct: "CommonTestData", CommonTestData);
```

Sl. 3.15. Poziv funkcije *WriteStruct()*.

Zadnji argument, sa slike 3.16., predstavlja par tip varijable – naziv varijable. Vektor je proizvoljne duljine te se po potrebi može dodavati u njega, a kao i kod *WriteEnum()*, nije potrebno mijenjati argumente poziva funkcije, već samo dodati (ili izbrisati) varijablu i njen tip u/iz već postojećeg vektora.

```
std::vector<std::pair<std::string,std::string>> CommonTestData{
        { x: "byte",  y: "Usage"}, { x: "byte",  y: "Info"}, { x: "byte",  y: "Flags"}, { x: "byte",  y: "HostId"},
        { x: "word",  y: "SwcId"}, { x: "word",  y: "RunnableNum"}, { x: "byte",  y: "ITFgroup"}
};
```

Sl. 3.16. Vektor parova nizova znakova naziva *CommonTestData*.

```
struct CommonTestData
{
  byte                  Usage;
  byte                  Info;
  byte                  Flags;
  byte                  HostId;
  word                  SwcId;
  word                  RunnableNum;
  word                  ITFgroup;
};
```

Sl. 3.17. Rezultat poziva funkcije *WriteStruct()*.

Funkcija *WriteChar()* ispisuje dvodimenzionalno polje znakova, podatkovnog tipa *char*. Za argumente prima, kao i sve funkcije do sada, najprije izlazni tok datoteke. U nju će zapisati naziv polja, što predstavlja drugi argument funkcije, podatkovnog tipa *string*, te će ono biti veličine trećeg argumenta funkcije, tipa *int*. Svaki od niza znakova će biti najviše duljine četvrtog predanog argumenta, također tipa *int*.

```
WriteChar( &: CAPL_cin, "operations", operations.size(), 10, operations);
```

Sl. 3.18. Poziv funkcije *WriteChar()*.

Zadnji argument koji prima je vektor nizova znakova (slika 3.19.). Kao i kod prethodnih funkcija, po potrebi je omogućeno jednostavno proširivanje sadržaja koji će se generirati u datoteku, na način da se doda neka nova operacija u vektor nizova znakova naziva *operations.* Funkcija će po promjeni učitati veličinu tog niza znakova te ispisati dimenziju tog polja, zajedno s drugim *intom*, koji predstavlja najveću moguću duljinu niza znakova.

```
std::vector<std::string> operations {
        "send", "receive", "write", "read"
};
```

Sl. 3.19. Vektor parova *stringova* naziva *operations.*

Pri pozivu funkcije sa slike 3.18., u datoteci se ispisuje sadržaj sa slike 3.20.

```
char                    operations[4][10] =
{ "send", "receive", "write", "read" };
```

Sl. 3.20. Rezultat poziva funkcije *WriteChar().*

Na slici 3.21. prikazano je korištenje implementiranih funkcija u programskom kodu, unutar glavne funkcije koju će se koristiti za generiranje CAPL koda, *GenerateCommonTypes().*

```
void GenerateCommonTypes(){

    std::ofstream CAPL_cin( s: R"(CAPL.cin)");

    Write( &: CAPL_cin,  word: "// created by  TEG-cpp");

    WriteStart( &: CAPL_cin);

    WriteComment( &: CAPL_cin, sectionBreak);

    WriteComment( &: CAPL_cin,  comment: "Hardware");

    WriteConst( &: CAPL_cin, "word",  "TEGversion", "1.0.0.");

    WriteEnum( &: CAPL_cin,  nameOfEnum: "Hosts", hosts);

    WriteStruct( &: CAPL_cin,  nameOfStruct: "CommonTestData", CommonTestData);

    WriteChar( &: CAPL_cin, "operations", operations.size(), 10, operations);

    Write( &: CAPL_cin,  word: "\n}");
}
```

Sl. 3.21. Glavna funkcija koja generira CAPL kod.

Pozivanjem funkcije sa slike 3.21., u datoteci *CAPL.cin* zapis izgleda ovako:

CAPL.cin

```
// created by  TEG-cpp

variables
{
  //============================================================================
  // current version of TestEnvironmentGenerator
  const byte              TEGversionMajor        = 1;
  const byte              TEGversionMinor        = 0;
  const byte              TEGversionPatch        = 0;

  //============================================================================
  //Hardware

  const word              TEGversion             = 1.0.0.;

  enum Hosts
  {
    SH00,
    PH00
  };

  struct CommonTestData
  {
    byte                   Usage;
    byte                   Info;
    byte                   Flags;
    byte                   HostId;
    word                   SwcId;
    word                   RunnableNum;
    word                   ITFgroup;
  };

  char                    operations[4][10] =
  { "send", "receive", "write", "read" };

}
```

Sl. 3.22. Rezultat pozivanja funkcije *GenerateCommonTypes()*.

Na slici 3.22. je vidljivo da struktura *CommonTestData* zadržava svoj format kao na slici 3.2. (a). Prelazi se na verifikaciju rješenja generiranjem veće datoteke, odnosno generiranjem konstanti, *enuma*, struktura i ostalih tipova podataka koji se nalaze u datoteci generiranoj starim TEG-om. Važno je napomenuti da postoji još puno struktura i funkcija unutar CAPL programskog jezika, primjerice struktura *includes*, te je u budućnosti moguće proširivanje generatora ukoliko to bude potrebno. Međutim, u ovom radu, takve strukture nisu razvijane jer ih postojeći generator testnog okruženja do sada nije koristio.

3.3. Programsko rješenje generatora *TestCase* lista

U sklopu rada potrebno je razviti i generator testnih slučajeva, odnosno implementirati funkcije koje će generirati potrebne testne slučajeve. Funkcije su implementirane u C++ programskom jeziku.

Pomoću postojećeg rukovatelja bazom potrebno je pronaći sve podatkovne komponente podsustava. Nova baza trenutno sadrži testni *build* ITF. Iako postoji nekoliko testnih okruženja u staroj bazi, za novu bazu je odabran ITF jer unutar njega se testira postojanje komunikacije između dvije programske komponente. Važno je naglasiti da se ovim okruženjem samo

provjerava postoji li komunikacija, dok druga testna okruženja onda mogu testirati točnost poruke (sadržaja) u komunikaciji, zbog čega treba prvo testirati ITF. Ukoliko nije moguće ostvariti komunikaciju, nema smisla provjeravati sadržaj poruka koje je pošiljatelj poslao. Testni *build* ITF se sastoji od nekoliko testnih grupa. Neki od primjera testnih grupa su:

- pošiljatelj - primatelj,
- pisač – čitač,
- testna grupa velikih podataka itd.

Unutar ovog rada generirat će se testni slučajevi za prvu od navedenih testnih grupa, pošiljatelj-primatelj. Razlog tome je što nova baza podataka trenutno sadrži testni *build* ITF te cjelovito sadrži jedino grupu pošiljatelj-primatelj. U budućnosti je moguće proširenje baze podataka na nekoliko testnih grupa i/ili *buildova*, pa je samim time, uz manje preinake, moguće i proširenje programa za generiranje ostalih testnih grupa i *buildova*.

Program će pomoću rukovatelja bazom učitati sve postojeće podatkovne komponente (engl. *data component*) sustava u program, koristeći vektore. Rukovatelj bazom istu pretražuje putem ključa. Dogovoreno je da sve što predstavlja komponentu sustava započinje ključem „Component“ te na taj način rukovatelj bazom zna što je podatkovna komponenta. Vraća sve podatkovne komponente koje je pronašao i sprema svaku u instancu klase te objekt komponente sprema u vektor podatkovnih komponenti. C++ programski jezik omogućuje objektni pristup rješavanju ovog problema.

Nakon što su sve komponente spremljene u vektor, kreće se u njihovu analizu. Na slici 3.23. je prikazano što će svaki objekt (komponenta) sadržavati kad rukovatelj bazom zapiše jednu od komponenti u klasu te stvori objekt. Redom sadrži podatke o: nazivu komponente, o domaćinu (engl. *host*), nazivu *runnable* funkcije te o sučelju (engl. *interface*). Sučelje se sastoji od oznake komponente (npr. pošiljatelj ili primatelj) te porta preko kojeg se odvija komunikacija. Dogovorno se na početku sučelja nalazi oznaka funkcije komponente. U ovom radu su obrađeni pošiljatelji (engl. *sender*) i primatelji (engl. *receiver*). Nastavak sučelja predstavlja *port.*

```
ComponentTypes<|>ComponentDisplayProcessing_PH00<|>Runnable_1_PH00<|>SENDER_PortDiagnostic_IntraHostPH00<|>+
PORT-PROTOTYPE-REF<|>/ComponentTypes/ComponentDisplayProcessing_PH00/PortDiagnostic_IntraHostPH00
TARGET-DATA-PROTOTYPE-REF<|>/PortInterfaces/PortDiagnostic/IntraHostPH00<|>@

ComponentTypes<|>ComponentTimeMonitor_PH00<|>Runnable_1_PH00<|>RECEIVER_PortDiagnostic_IntraHostPH00<|>+
PORT-PROTOTYPE-REF<|>/ComponentTypes/ComponentTimeMonitor_PH00/PortDiagnostic_IntraHostPH00
TARGET-DATA-PROTOTYPE-REF<|>/PortInterfaces/PortDiagnostic/IntraHostPH00<|>@
```

Sl. 3.23. Primjer komponenti unutar baze.

Generator testnog okruženja trenutno sadrži dva *hosta*, ***s**afety **h**ost (SH)* i ***p**erformance **h**ost* (*PH*). Budući da svaki od *hostova* ima svoju bazu, iz slike 3.23., iz naziva komponente, vidimo da one pripadaju *performance hostu*, jer u imenu sadrže „*PH00“*. Pri izradi testnih slučajeva, ključno je sučelje. Ono daje informaciju koji pošiljatelj komunicira s kojim primateljem. Primjeri sučelja nalaze se zaokruženi na slici 3.24.

```
ComponentTypes<|>ComponentDisplayProcessing_PH00<|>Runnable_1_PH00<|>SENDER_PortDiagnostic_IntraHostPH00<|>+
PORT-PROTOTYPE-REF<|>/ComponentTypes/ComponentDisplayProcessing_PH00/PortDiagnostic_IntraHostPH00
TARGET-DATA-PROTOTYPE-REF<|>/PortInterfaces/PortDiagnostic/IntraHostPH00<|>@

ComponentTypes<|>ComponentTimeMonitor_PH00<|>Runnable_1_PH00<|>RECEIVER_PortDiagnostic_IntraHostPH00<|>+
PORT-PROTOTYPE-REF<|>/ComponentTypes/ComponentTimeMonitor_PH00/PortDiagnostic_IntraHostPH00
TARGET-DATA-PROTOTYPE-REF<|>/PortInterfaces/PortDiagnostic/IntraHostPH00<|>@
```

Sl. 3.24. Primjer sučelja koji se podudaraju.

Kada se iz postojećeg niza znakova (engl. *string*) odbaci oznaka komponente, koja može biti ili „*SENDER_*“ ili „*RECEIVER_*“, vidljivo je koje je sučelje u pitanju. Kada se dva sučelja podudaraju, tada program zna da se između te dvije komponente treba testirati komunikacija, odnosno treba se ispisati testni slučaj u datoteku *.csv* formata.

Sljedeći korak je ispisati testni slučaj u datoteku naziva *TestCaseList_ITF.csv*. Potrebno je zadržati postojeći format jednog testnog slučaja koji je preuzet iz starog generatora testnog okruženja.

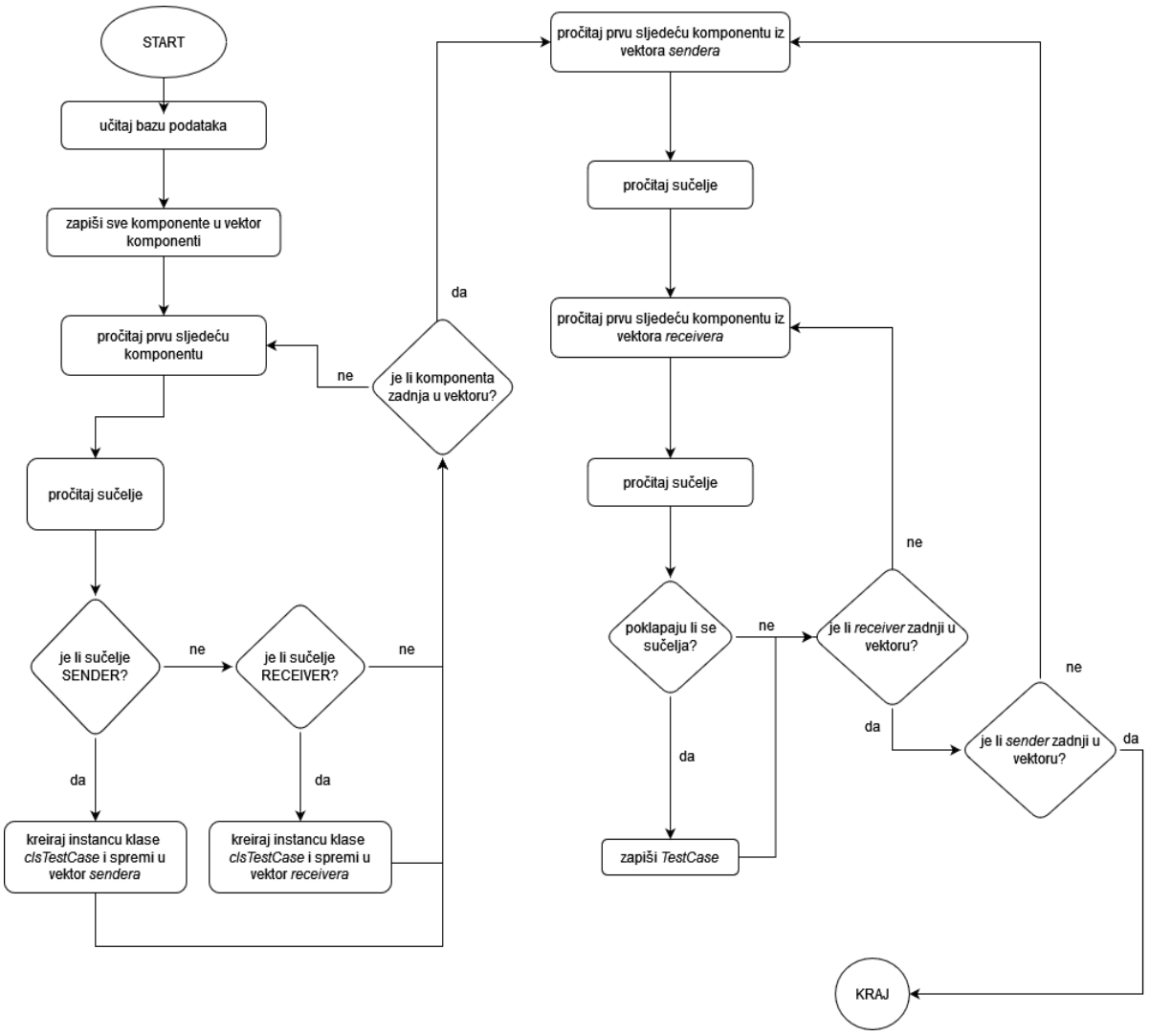
```
#
# created by  TestEnvGenerator ( Version 1.x.x )
#
TEG-1.x.x; ITF
#
TestGroup_SenderReceiver;
ITF.SendRec-001;
                ABCD;
                SH00-3.ComponentPositioningInterface_SH00-send-1.SENDER_PortGlobalVehicleState;
                SH00-12.ComponentGlobalDataReceiver-receive-1.RECEIVER_PortGlobalVehicleState
```

Sl. 3.25. Primjer testnog slučaja iz starog TEG-a.

Važno je napomenuti da se jedan testni slučaj piše u jedan red, ali zbog jednostavnosti prikaza je prelomljen u više redova. Na slici 3.25. se nalazi prvi po redu testni slučaj iz starog TEG-a. Za prvu pošiljateljsku programsku komponentu (na slici *ComponentPositioningInterface_SH00*) algoritam čita sučelje bez oznake pošiljatelja (odbacuje se „SENDER_“) te pronalazi prvu primateljsku komponentu koja ima jednako sučelje, ali oznake „RECEIVER_“. Kada je komponenta pronađena, zapisuje se cijeli testni slučaj u datoteku *TestCaseList_ITF.csv*. Također je moguće postojanje više primatelja za

jednog pošiljatelja. *ABCD* predstavlja ASIL (***A**utomotive **S**afety **I**ntegrity **L**evel*) razine, odnosno razine sigurnosti u cestovnim vozilima, gdje *A* predstavlja najmanje riskantnu razinu, a *D* najriskantniju. Primjerice, ASIL razinom *A* može se označiti neispravna žarulja na zadnjem svjetlu automobila. To je nedostatak, ali najmanjeg rizika, za razliku od neispravnog zračnog jastuka, kojeg će se označiti ASIL oznakom *D*.

Sljedeći korak je svakoj programskoj komponenti pridijeliti njen brojčani ID koji predstavlja redni broj komponente u bazi. Tako će komponenta *ComponentPositioningInterface_SH00* dobiti ID 3, jer je ona treća pronađena programska komponenta u bazi. Komponenta *ComponentGlobalDataReceiver* će dobiti ID 12.



Sl. 3.26. Dijagram toka za algoritam ispisivanje *TestCase* liste.

Algoritam iterira po svim komponentama dokle god pronalazi isto sučelje. Kada više ne pronalazi ista sučelja, prelazi na sljedeću komponentu. Ponavlja iteriranje, kada nađe pošiljatelja, traži njegove primatelje. Kada prođe kroz sve komponente, zapisao je sve i završava se program. Dijagram toka je vidljiv na slici 3.26.

U C++ programskom jeziku, u sklopu rukovatelja bazom, već postoje funkcije i algoritmi za dohvaćanje i čitanje podatkovnih komponenti iz baze. Implementirano je čitanje podataka iz svake komponente baze za traženje preklapajućih sučelja te ispisivanje u *.csv* datoteku. Kao glavnu prednost C++ naveden je objektno orijentirani pristup. Zbog toga je implementirana klasa *clsTestCase* koja će pri izradi testnog slučaja služiti za pohranu svih podataka potrebnih za ispisivanje.

```
class clsTestCase {
    string host;
    string component;
    string runnable;
    string operation;
    string interface;
    int nComponent;
    int nRunnable;
    int nInterface;

public:
    void setHost(const string&);
    void setComponent(const string&);
    void setRunnable(const string&);
    void setOperation(const string&);
    void setInterface(const string&);
    void setNumberComponent(int);
    void setNumberRunnable(int);
    void setNumberInterface(int);

    const string& getHost() const;
    const string& getComponent() const;
    const string& getRunnable() const;
    const string& getOperation() const;
    const string& getInterface() const;
    int getNumberComponent() const;
    int getNumberRunnable() const;
    int getNumberInterface() const;

    clsTestCase();
    clsTestCase(string h, string cp, string r, string o, string i, int nC, int nR, int nI);
    ~clsTestCase();
};
```

Sl. 3.27. Klasa koja u sebi sadrži informacije o svakoj komponenti.

Klasa sa slike 3.27. je implementirana kako bi se svi podaci o testnom slučaju spremili na jedno mjesto. Sadrži redom:

- *string host* – naziv hosta (*PH/SH),*
- *string component* – naziv programske komponente (SWC),
- *string runnable* – naziv *runnable* funkcije,
- *string operation* – operacija, odnosno označava je li komponenta pošiljatelj ili primatelj,
- *string interface* – sučelje, po kojem se uspoređuju imaju li pošiljatelj i primatelj isto sučelje kako bi se mogli zapisati u obliku testnog slučaja u datoteku.

Osim navedenih privatnih članova, klasa sadrži javne konstruktore i destruktore te funkcije za postavljanje vrijednosti prethodno navedenih varijabli i njihovo dohvaćanje (čitanje) iz klase.

```
void Write_Info(std::ofstream &, const std::string &);

void Write_TestGroup(std::ofstream &, const std::string &);

string FindHost(const std::string &, std::string &);

string FindOperation(const std::string &);

string FindInterface(const std::string &);

void eraseSubStr(std::string &, const std::string &);

void WriteSingleTestCase(std::ofstream &, const std::string &, const std::string &, int, const std::string &);

void CreateTestCases(DatabaseHandler *, std::vector<weak_ptr<DataComponent>> &, std::vector<weak_ptr<DataComponent>> &,
                     std::vector<clsTestCase> &, std::vector<clsTestCase> &, std::ofstream &, std::string Type);
```

Sl. 3.28. Funkcije korištene u programskom kodu za generiranje testnih slučajeva.

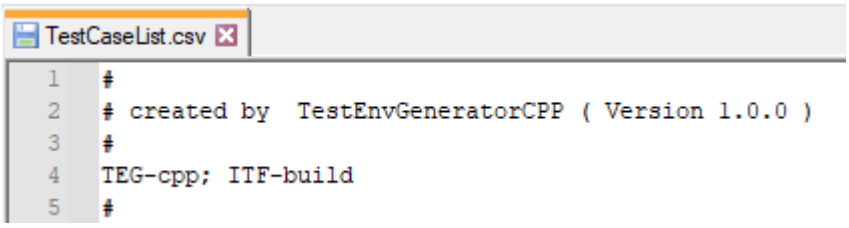
Prva u nizu implementiranih funkcija, koje se nalaze na slici 3.28. je jednostavna funkcija za ispisivanje informacija, *Write_Info().* Prima jedan argument, izlazni tok datoteke u koju želimo upisati informacije. Poziv funkcije vidljiv je na slici 3.29.

```
ofstream TestCaseList_csv( s: R"(TestCaseList.csv)");

Write_Info( &: TestCaseListITF_csv,  build: "ITF");
```

Sl. 3.29. Poziv funkcije *WriteInfo()*.

Zapis u datoteci *TestCaseList.csv* onda izgleda kao na slici 3.30.:



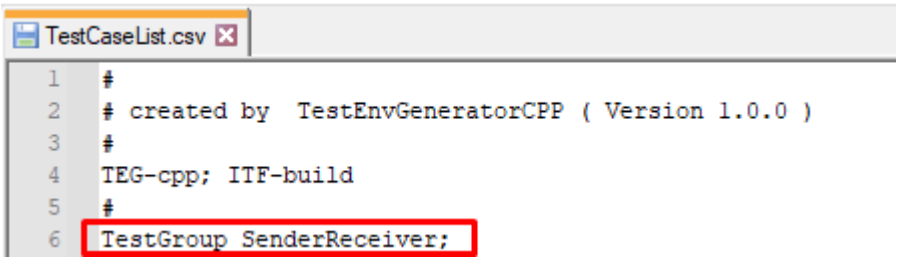
Sl. 3.30. Rezultat pozivanja funkcije *WriteInfo()*.

Sljedeća funkcija je ispis testne grupe u datoteku pomoću funkcije *Write_TestGroup()*. Funkcija ispisuje niz znakova koji predstavlja testnu grupu za koju će se ispisati testni slučajevi. Osim testne grupe, funkcija pri pozivu (slika 3.31.) prima izlazni tok datoteke u koji ćemo upisati testnu grupu.

```
Write_TestGroup( &: TestCaseListITF_csv, TestGroup: "SenderReceiver");
```

Sl. 3.31. Poziv funkcije *Write_TestGroup()*.

Funkcija ima jednostavnu primjenu, upisivanjem željene testne grupe, ispisuje se niz znakova zajedno sa nizom znakova „TestGroup_“ te se ispis vidi na slici 3.32.



Sl. 3.32. Rezultat poziva funkcije *Write_TestGroup()*.

Sljedeća funkcija je *FindHost()*. Funkcija od podataka prima niz znakova koji predstavlja naziv komponente za koju tražimo njen host. Dogovorno se zna da svaka komponenta na kraju naziva ima skraćenicu hosta. Funkcija ne ispisuje ništa, već vraća niz znakova kao povratnu vrijednost. Ukoliko je *performance host* u pitanju, pronaći će *PH00*, i vratiti „PH“. Ukoliko je *safety host*¸ pronaći će *SH00* i vratiti „SH“. Poziv funkcije nalazi se na slici 3.33.

```
string host = FindHost(component);
```

Sl. 3.33. Poziv funkcije *FindHost()*.

Funkcije *FindOperation()* i *FindInterface()*, čiji pozivi se nalaze na slici 3.34., kao argument primaju isti niz znakova. Radi se o zaokruženom nizu sa slike 3.24., a njega dohvaća postojeći rukovatelj bazom. Razlika između funkcija je što prva vraća niz znakova o kojoj se operaciji radi (*send* ili *receive*), a odbacuje ostatak niza. Druga, *FindInterface()*, vraća taj ostatak niza znakova, odnosno sučelje.

```
operation = FindOperation(It.lock()->GetID());

interface = FindInterface(It.lock()->GetID());
```

Sl. 3.34. Pozivanje funkcija *FindOperation()* i *FindInterface()*.

Funkcija se koristi pri izradi testnih slučajeva, kao način za pridodjeljivanjem vrijednosti varijablama *operation* i *interface* sa slike 3.27.

Zadnja funkcija koja se koristi kao pomoćna funkcija je ispisivanje jednog testnog slučaja. Ta se funkcija onda koristi u glavnoj funkciji *CreateTestCases()* u iteracijama te njenim pokretanjem funkcija *WriteSingleTestCase()* se pokrene koliko god puta se pronađe preklapajuće sučelje, a ispisuje jedan po jedan testni slučaj.

```
TestCaseString += sender.getHost() + "-" +
                  to_string(sender.getNumberComponent()) + "." +
                  sender.getComponent() + "-" +
                  to_string(sender.getNumberInterface()) + "." +
                  sender.getInterface() + "; "
```

(a)

```
TestCaseString += receiver.getHost() + "-" +
                  to_string(receiver.getNumberComponent()) + "." +
                  receiver.getComponent() + "-" +
                  to_string(receiver.getNumberInterface()) + "." + receiver.getInterface() + "; ";
```

(b)

Sl. 3.35. Slaganje niza znakova pošiljatelja (a) i primatelja (b) za ispis testnog slučaja.

Slaganje niza znakova koji će se predati funkciji *WriteSingleTestCase()* pri pozivu (slika 3.36.) kao argument za ispis testnog slučaja nalazi se na slici 3.35.

```
WriteSingleTestCase( &: TestCaseListITF_csv, Type, shortTestGroup: "SendRec", TCNumber, TestCaseString);
```

Sl. 3.36. Pozivanje funkcije *WriteSingleTestCase()*.

Zadnja funkcija koju će glavna funkcija koristiti je *CreateTestCases()*. Pri pozivu prima postojeći, korišteni, rukovatelj bazom, listu komponenti, listu sučelja, testnu grupu, dva vektora klasa *clsTestCase* od kojih jedan predstavlja pošiljatelje (*clsSend*), a drugi primatelje (*clsReceive)*, datoteku u koju zapisuje testne slučajeve te testni *build* o kojemu je riječ (ITF). Poziv funkcije vidljiv je na slici 3.37. Funkcija kreira sve testne slučajeve u obliku instance klase *clsTestCase*.

```
CreateTestCases(DatabaseHandler, &Components, &Interfaces, testGroup: "SenderReceiver", &clsSend, &clsReceive,
                &TestCaseList_csv, Type: "ITF");
```

Sl. 3.37. Pozivanje funkcije *CreateTestCases()*.

Glavna funkcija koju pokreće TEG okvir je *GenerateTestCases()*. Iz ove funkcije se poziva rukovatelj bazom koji funkcijom *GetComponentsByHint()* pronalazi sve komponente po ključu „*Components*“. Ispisuju se informacije o programu i testna grupa *SenderReceiver*. Poziva se funkcija za izradu testnih slučajeva, odnosno čitanje svih podataka iz baze podataka za svaku podatkovnu komponentu te spremanje potrebnih informacija u klasu *clsTestCase*. Zbog filtriranja pošiljatelja i primatelja definirana su dva vektora objekata *clsTestCase*. Na taj način se mogu lakše uspoređivati njihova sučelja, koja na slici 3.27. predstavlja varijabla *string interface.*

```
void GenerateTestCases(DatabaseHandler *DatabaseHandler) {

    ofstream TestCaseList_csv( s: R"(TestCaseList.csv)");

    Write_Info( &TestCaseList_csv, build: "ITF");

    vector<weak_ptr<DataComponent>> Components;
    vector<weak_ptr<DataComponent>> Interfaces;

    vector<clsTestCase> clsSend;
    vector<clsTestCase> clsReceive;

    DatabaseHandler->GetComponentsByHint( &Components, "Components");

    Write_TestGroup( &TestCaseList_csv, TestGroup: "SenderReceive");

    CreateTestCases(DatabaseHandler, &Components, &Interfaces, testGroup: "SenderReceiver", &clsSend, &clsReceive,
                    &TestCaseList_csv, Type: "ITF");

    TestCaseList_csv.close();
}
```

Sl. 3.38. Glavna funkcija za generiranje testnih slučajeva.

Nakon pozivanja funkcije *GenerateTestCases(),* datoteka *TestCaseList.csv* izgleda kao na slici 3.39., na kojoj je prikazan dio liste testnih slučajeva. Funkcija *GenerateTestCases()* izgenerirala je 13 testnih slučajeva, odnosno točno toliko podudarajućih pošiljatelj-primatelj sučelja je pronašla. Napomena je da postojeći, stari, generator testnih slučajeva generira 39

slučajeva za istu testnu grupu, a to se pripisuje većoj bazi podataka. Također, stari generator u sebi sadrži nekoliko testnih grupa, dok novo rješenje u novoj bazi pronalazi samo testnu grupu pošiljatelj-primatelj. U budućnosti je moguće novo rješenje prilagoditi bazi podataka, tako što će program prepoznavati postoji li više testnih grupa, i ako postoje, o kojima se radi. Također je moguće prilagoditi program za prepoznavanje više *hostova*, više sučelja (varijabla *interface*) i slično.

TestCaseList.csv

```
#
# created by  TestEnvGeneratorCPP ( Version 1.0.0 )
#
TEG-cpp; ITF-build
#
TestGroup_SenderReceiver;
ITF.SendRec-001;
                ABCD;
                PH00-2.ComponentDisplayProcessing_PH00-send-1.SENDER_PortDiagnostic_IntraHostPH00;
                PH00-18.ComponentTimeMonitor_PH00-receive-1.RECEIVER_PortDiagnostic_IntraHostPH00;
ITF.SendRec-002;
                ABCD;
                PH00-3.ComponentEthernet_PH00-send-1.SENDER_PortEth_IntraHostPH00;
                PH00-20.ComponentCommTest1_PH00-receive-1.RECEIVER_PortEth_IntraHostPH00;
ITF.SendRec-003;
                ABCD;
                PH00-5.ComponentTest1_PH00-send-1.SENDER_PortTest101_IntraHostPH00;
                PH00-21.ComponentCommTest2_PH00-receive-1.RECEIVER_PortTest101_IntraHostPH00;
ITF.SendRec-004;
                ABCD;
                PH00-5.ComponentTest1_PH00-send-2.SENDER_PortTest102_IntraHostPH00;
                PH00-27.ComponentGlobalPosition_PH00-receive-1.RECEIVER_PortTest102_IntraHostPH00;
ITF.SendRec-005;
                ABCD;
                PH00-5.ComponentTest1_PH00-send-3.SENDER_PortTest103_IntraHostPH00;
                PH00-28.ComponentClock_PH00-receive-1.RECEIVER_PortTest103_IntraHostPH00;
ITF.SendRec-006;
                ABCD;
                PH00-6.ComponentTest2_PH00-send-1.SENDER_PortTest201_IntraHostPH00;
                PH00-29.ComponentSensor01_PH00-receive-1.RECEIVER_PortTest201_IntraHostPH00;
ITF.SendRec-007;
                ABCD;
                PH00-6.ComponentTest2_PH00-send-2.SENDER_PortTest202_IntraHostPH00;
                PH00-30.ComponentSensor02_PH00-receive-1.RECEIVER_PortTest202_IntraHostPH00;
ITF.SendRec-008;
                ABCD;
                PH00-15.ComponentComm1_PH00-send-1.SENDER_PortCommChannel1_IntraHostPH00;
                PH00-31.ComponentSensor03_PH00-receive-1.RECEIVER_PortCommChannel1_IntraHostPH00;
ITF.SendRec-009;
                ABCD;
                PH00-16.ComponentPositioning_PH00-send-1.SENDER_PortPositionPort1_IntraHostPH00;
                PH00-33.ComponentASILCom_PH00-receive-1.RECEIVER_PortPositionPort1_IntraHostPH00;
```

Sl. 3.39. Dio zapisa testnih slučajeva u datoteci *TestCaseList.csv*.

4. VERIFIKACIJA RJEŠENJA

Budući da je najveće unaprjeđenje novog generatora to što je razvijan u C++ programskom jeziku, potrebno je usporediti stari i novi generator kako bi bilo vidljivo jesu li datoteke održale isti format. Bitno je da datoteka CAPL generatora (*.cin*) te testni slučajevi (liste, *.csv*) budu istog formata kao postojeći generatori.

4.1. Usporedba postojećih i novih generiranih datoteka

U ovom potpoglavlju uspoređen je format postojećeg i novog generatora.

4.1.1. CAPL generator

U CAPL generatoru je bitno generirati sve potrebne podatke, od najnižih tipova podataka kao što su *byte* i *word*, do struktura (*struct*). Za usporedbu generiranih podataka, preuzeta je generirana datoteka iz starog generatora. Funkcije koje su napisane omogućavaju po potrebi dodavanje još konstanti, varijabli, struktura ili *enuma*. Na slici 4.1. se nalazi usporedba starog (lijevo) i novog (desno) rješenja, gdje se poklapaju apsolutno sve linije koda, koje predstavljaju strukturu *CommonTestData*, a predstavlja informacije o općenitim podacima.

CAPL.cin

```
struct CommonTestData
{
  byte                  Usage;
  byte                  Info;
  byte                  Flags;
  byte                  HostId;
  word                  SwcId;
  word                  RunnableNum;
  byte                  ITFgroup;
};
```

(a)

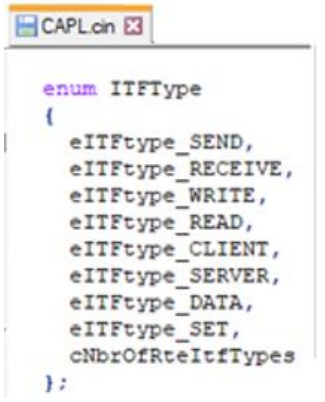
CAPL.cin

```
struct CommonTestData
{
  byte                  Usage;
  byte                  Info;
  byte                  Flags;
  byte                  HostId;
  word                  SwcId;
  word                  RunnableNum;
  byte                  ITFgroup;
};
```

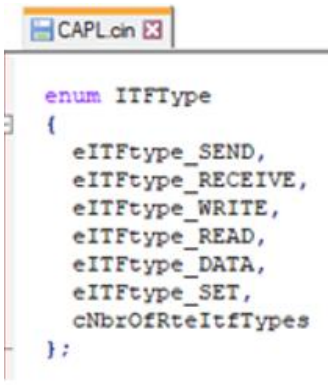
(b)

Sl. 4.1. Stari (a) i novi CAPL kod.

Također se događa da se neki tipovi podataka ne poklapaju. Na slici 4.2. vidljiva su manja odstupanja u sadržaju *enuma* koji predstavlja moguće funkcije koje komponenta može imati.



(a)



(b)

Sl. 4.2. Stari (a) i novi (b) CAPL kod.

Iz ranije navedenih sučelja prepoznaje se *SEND* i *RECEIVE*. Vidi se da staro rješenje (na slici 4.2. a) ima više postojećih funkcija, ali u novom programu se može, po potrebi, generirati još varijabli unutar *enuma* zbog dinamičnosti funkcija. Potrebno je samo proširiti vektor nizova znakova koji predstavljaju naziv člana *enuma*.

Nadalje, na slici 4.3. jasno je da se dva generatora znatno razlikuju u *enumu* naziva *TestType*. *TestType* predstavlja informaciju o kojem testnom *buildu* se radi i o kojoj testnoj grupi. Razlog tome je postojanje nekoliko *buildova* (ITF, BUS, PER) u staroj bazi te nekoliko testnih grupa za svaki od *buildova*, a postojanje jednog *builda* (ITF) i jedne testne grupe (pošiljatelj-primatelj, skraćeno SR) u novoj bazi. Važno je naglasiti da nepodudaranje u ovom obliku nije greška, dokle god je format (*enum*, struktura) jednakog oblika. Njegovi članovi se mogu po potrebi dodavati ili brisati

CAPL.cin

```
enum TestType
  {
    TestType_ITF_SR,
    TestType_ITF_WR,
    TestType_ITF_ST,
    TestType_ITF_DS,
    TestType_ITF_BD,
    TestType_ITF_MT,
    TestType_BUS_FO,
    TestType_BUS_FI,
    TestType_BUS_CO,
    TestType_BUS_CI,
    TestType_BUS_CS,
```

(a)

CAPL.cin

```
enum TestType
{
  TestType_ITF_SR,
  TestType_ITF_WR,
  TestType_ITF_DS,
  TestType_ITF_CS,
};
```

(b)

Sl. 4.3. Stari (a) i novi (b) CAPL kod.

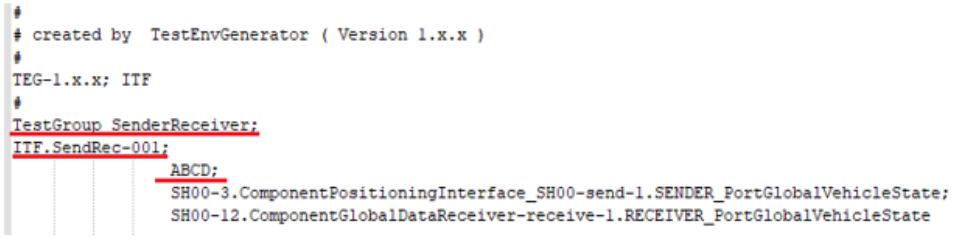
4.1.2. Generator testnih slučajeva

Slika 3.25. prikazuje jedan testni slučaj iz stare liste testnih slučajeva. Na slici 4.4. prikazan je testni slučaj novog generatora.

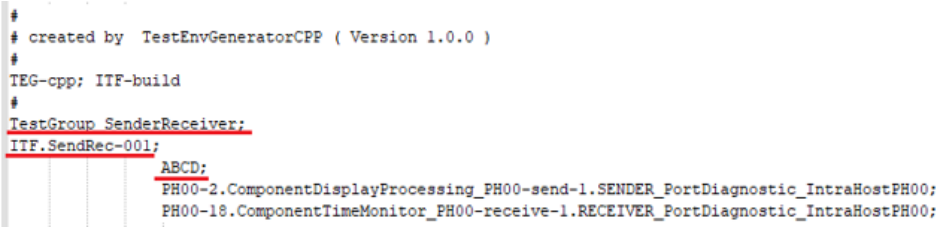
```
#
# created by  TestEnvGeneratorCPP ( Version 1.0.0 )
#
TEG-cpp; ITF-build
#
TestGroup_SenderReceiver;
ITF.SendRec-001;
                ABCD;
                PH00-2.ComponentDisplayProcessing_PH00-send-1.SENDER_PortDiagnostic_IntraHostPH00;
                PH00-18.ComponentTimeMonitor_PH00-receive-1.RECEIVER_PortDiagnostic_IntraHostPH00;
```

Sl. 4.4. Testni slučaj novog generatora.

Kako bi se verificiralo novo rješenje, na slici 4.5. nalaze se stari i novi testni slučaj koji su stavljeni jedan ispod drugog te su označene informacije koje je potrebno usporediti.



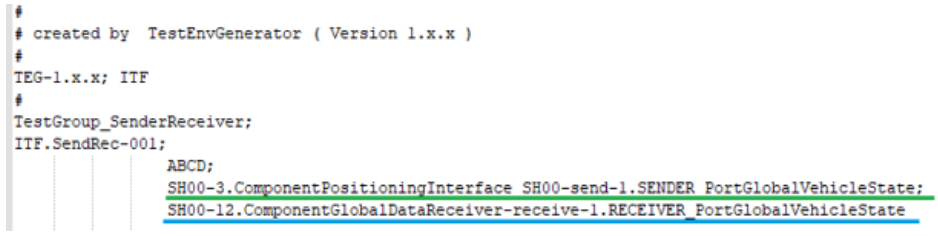
(a)



(b)

Sl 4.5. Usporedba starog (a) i novog (b) testnog slučaja.

Crvenom bojom na slici 4.5. označen je prvi od tri dijela testnog slučaja. Sadrži testnu grupu, redni broj testnog slučaja, njegovu oznaku te ASIL razine. Iz slike je vidljivo da su potrebne informacije istog sadržaja te se prelazi na analizu preostalog dijela testnog slučaja. U ovom dijelu promjenjiv je samo redni broj testnog slučaja.



(a)

(b)

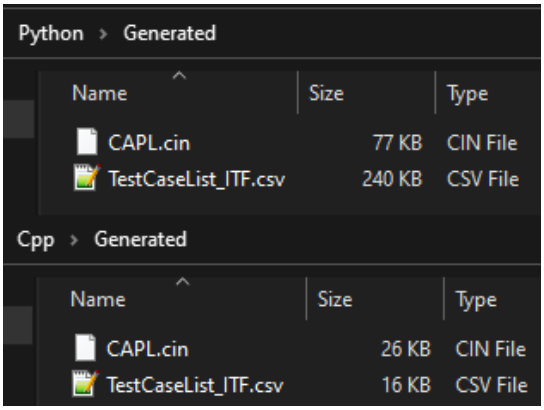
Sl. 4.6. Usporedba starog (a) i novog (b) testnog slučaja.

Na slici 4.6. zelenom bojom podvučen je drugi od tri dijela testnog slučaja. U njemu se nalaze podaci o pošiljatelju. Ono što obje strane (pošiljatelj i primatelj) sadrže je: *host*, redni broj komponente, naziv programske komponente, funkcija pošalji (engl. *send*) te sučelje preko

kojeg se odvija komunikacija, pomoću kojeg se nalaze par pošiljatelj-primatelj. Na slici je vidljivo da se razlikuju skoro sve informacije, ali su istog formata. Postojeći generator (na slici gore), sadrži *safety host* te je u njemu zapisano „*SH00*“, dok novi generator iz baze čita *performance host* komponente te je u njemu zapisano „*PH00*“. Svaka informacija je odvojena znakom „ – „. Ostatak testnog slučaja se razlikuje u sadržaju, ali u njemu se nalaze sve potrebne informacije koje jedan testni slučaj treba imati, u redoslijedu u kojem mora biti. Sadržaj se razlikuje jer stari generator ima svoju bazu podataka, a novi generator novu bazu podataka. Analogno, analizom primatelja (podvučeno plavom bojom), vidljivo je da se format poklapa sa starim testnim slučajem. Kao i u pošiljateljskom dijelu, sadržaj je različit. Za pokretanje novih testnih slučajeva potrebno ih je pokrenuti u alatu *CANoe* te se to predlaže za buduće radove u sklopu TEG-a. *CANoe* će onda vratiti povratnu vrijednost prolaza ili pada na testiranju.

4.2. Usporedba memorijske zahtjevnosti i vremena izvođenja

U ovom potpoglavlju uspoređene su veličine generiranih datoteka postojećih i novih generatora, a prikazana je i usporedba vremena potrebnih za izvođenje oba generatora. Stari generatori su imali pristup većoj bazi podataka. Stara baza podataka strukturno je drugačija od nove baze te su u njima nalazi već navedeni veći broj testnih *buildova*, unutar kojih se nalazi veći broj testnih grupa. Novi generatori rade samo na jednom testnom *buildu* i na jednoj testnoj grupi unutar tog *builda*, u sklopu nove baze podataka.



Sl. 4.7. Usporedba veličine datoteka starog i novog generatora.

Na slici 4.7. prikazana je usporedba veličina generiranih datoteka. Jasno je da su datoteke generirane u *Python* TEG-u veće jer se radi o većoj bazi podataka. Budući da se format datoteka podudara, razlika je samo u broju postojećih programskih komponenata, kojih u staroj bazi podataka ima i do nekoliko puta više. Međutim, pošto se radi o redu veličine kB, obje datoteke,

iz starog i iz novog generatora, smatraju se malima. Veličina postojećih datoteka prikazuje da čak i s velikim brojem komponenti, memorijska zahtjevnost neće biti velika.

Nadalje se testira vremenska zahtjevnost. Na staroj bazi u sklopu starog generatora pokrenuta su tri generiranja iz kojih se vidi da su otprilike jednaka vremena izvođenja. Na novoj bazi je isto toliko puta generirano u sklopu novog generatora. Vremena generiranja prikazana su u tablici 4.1. i tablici 4.2.

Tablica 4.1. Usporedba vremena izvođenja starog i novog CAPL generatora.

	Stari generator CAPL [ms]	***Novi CAPL generator [ms]***
1.	1851	704
2.	2204	698
3.	1627	652
Srednja vr.	1894	684

Tablica 4.2. Usporedba vremena izvođenja starog i novog generatora testnih slučajeva.

	Stari generator testnih slučajeva [s]	***Novi generator testnih slučajeva [s]***
1.	59,05	11,34
2.	65,12	11,87
3.	70,78	11,56
Srednja vr.	64,98	11,59

U tablici 4.1. vidljivo je da stari generator otprilike tri puta sporije generira CAPL kod. Budući da je datoteka generirana starim generatorom skoro pa točno tri puta veća od nove (stara datoteka je veličine 77kB, dok je nova 26kB), može se zaključiti da novi generator CAPL koda nije dobio na performansama, već je razlika u vremenu izvođenja isključivo zbog toga što novo rješenje generira iz manje baze podataka.

Prema tablici 4.2., odnosno srednjoj vrijednosti iz tablice, novi generator testnih slučajeva je brži otprilike šest puta od starog. Međutim, prema slici 4.7., staro rješenje generira datoteku veličine 240kB, dok novo generira datoteku 16kB, dakle otprilike petnaest puta manju datoteku za šest puta manje vrijeme izvođenja. Razlog tomu je pristup generiranju testnih slučajeva sa spremanjem u klase. Radi jednostavnosti čitanja i točnosti informacija o svakoj programskoj komponenti, drugi i treći dio (informacije o pošiljateljskoj komponenti i informacije o primateljskoj komponenti) su spremane u klasu *clsTestCase* te su instance te klase spremane u vektore. Također, rukovatelj bazom sve pronađene komponente zapisuje u vektor klasa podatkovnih komponenti. U budućnosti je moguće generator testnih slučajeva isprobati na nekom drugačijem principu, primjerice bez spremanja u klasu, već spremanjem naziva

komponenti, sučelja i *hostova* u tip podatka *string* te ih povezati sve putem istog ključa (ID-a) kako bi se usporedile performanse.

Uspoređivanjem prethodne dvije tablice vidljivo je da je novo rješenje nekoliko puta brže. Međutim, budući da novo rješenje ima puno manju bazu podataka, može se zaključiti da će vremena izvođenja biti skoro pa ista za generator CAPL koda, odnosno i sporija za novi generator testnih slučajeva, ukoliko se nova baza poveća nekoliko puta. Zaključuje se da novi generator testnih slučajeva nije unaprjeđen po pitanju performansa, ali budući da su vremena i dalje u sekundama za puno veću bazu podataka, nije prioritet ubrzati program, već ga sakriti od korisnika. To je upravo najveća prednost novog generatora, što je razvijen u C++ programskom jeziku.

5. ZAKLJUČAK

Razvijanjem tehnologije u automobilima dolazi do potrebe za testiranjem svih komponenti sustava. Generator testnog okruženja je jedan od alata za testiranje komponenti i komunikacijskih kanala putem kojih se odvija komunikacija između komponenti. Generator simulira testno okruženje na kojemu se može testirati bilo koji dio sustava, ovisno o potrebi. Ovaj diplomski rad se bavio razvojem generatora CAPL koda i generatora testnih slučajeva. Za razliku od postojećeg rješenja pisanog u *Python* programskom jeziku, ovdje je zadatak bio implementirati generatore u programskom jeziku C++. S obzirom da se prilikom generiranja datoteka, u drugom programskom okruženju i jeziku, uspio očuvati isti format podataka, moguće je u budućnosti koristiti nove datoteke prilikom pokretanja testnih slučajeva u alatu za testiranje. Uspoređena vremena i memorijska zahtjevnost daju zaključak da je novo rješenje malo brže od starog, ali budući da je krajnji zadatak zapisivanje u datoteku, vremena se neznatno razlikuju. Vremenski zahtjevi novog i starog TEG-a se razlikuju u ranijim fazama generatora, a to su parsiranje podataka, spremanje u bazu podataka i čitanje iz baze, gdje novi TEG daje puno bolje rezultate. Memorijska zahtjevnost novog generatora je manja, ali samo zato što je njegova baza podataka manja. Ukoliko bi se baza podataka proširila, zahtjevnost bi bila slična ili jednaka postojećoj. Zaključuje se da je najveća prednost novog rješenja njegova enkapsulacija, odnosno čahurenje, te će u sklopu novog generatora testnog okruženja, izvorni kod biti nevidljiv korisnicima generatora.

LITERATURA

[1] AUTOSAR, Layered Software Architecture, https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf , pristupljeno 2021.

[2] Andrija Mihalj, Generator softverskog koda namijenjenog za testiranje ADAS sustava, diplomski rad, Osijek, 2019., pristupljeno 2021.

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

[4] History of C++, https://www.cplusplus.com/info/history/ , pristupljeno 2021.

[5] A Brief Description of C++, https://www.cplusplus.com/info/description/ , pristupljeno 2021.

[6] The Use of CAPL, https://www.vector.com/int/en/know-how/capl/ , pristupljeno 2021.

[7] CANoe, ECU and Network Testing, https://www.vector.com/hr/en/products/products-a-z/software/canoe/ , pristupljeno 2021.

[8] CLion, https://snapcraft.io/clion , pristupljeno 2021.

SAŽETAK

Današnji automobili imaju jako velik broj složenih elektroničkih sustava kojima je potrebno upravljati. Upravljanje se odvija komunikacijom putem komunikacijskih kanala između dvije ili više komponenti, ili između elektroničkih sustava. Svaka od komponenti i svaki od komunikacijskih kanala moraju biti testirani prije upotrebe. Za potrebe testiranja izrađen je generator testnog okruženja. U ovom diplomskom radu napravljena je analiza postojećeg generatora koji je pisan u programskom jeziku *Python* 2 te se implementiralo novo rješenje u programskom jeziku C++. Razvijen je novi generator CAPL koda, a uspoređen je s postojećim generatorom, kako bi se održala sintaksa CAPL programskog jezika. *TestCase* generator generira testne slučajeve na temelju postojeće baze podataka. Testni slučajevi se zapisuju u datoteku *.csv* formata. Nakon generiranja utvrđeno je da su sve datoteke istog formata kao i postojeći TEG pisan u *Pythonu*, što je bio i cilj jer se mora održati standard pri razvijanju novog rješenja u C++ programskom jeziku.

Ključne riječi: generator testnog okruženja, CAPL generator, *TestCase* generator, C++

Developing CAPL generator within test environment generator

ABSTRACT

Today’s vehicles have a very large number of advanced electronic systems which need to be handled. Handling is managed by communication through communication channels between two or more components, or between electronic systems. Each of components and each of communication channels need to be tested before use. Test environment generator was developed for testing purposes. In this master's thesis, an analysis of the existing generator was made. Generator was written in *Python* 2 programming language, while new solution was developed in programming language C++. New CAPL generator was developed and compared to existing generator, in order to maintain CAPL programming language syntax. *TestCase* generator generates test cases from existing database. Test cases are written in file format *.csv*. After generating, it was determined that all of generated files are matching generated files from existing test environment generator. That was the goal because standard must be maintained when developing a new solution in the C++ programming language.

Key words: test environment generator, CAPL generator, *TestCase* generator, C++

ŽIVOTOPIS

Abraham Kostić rođen je u Osijeku 4. listopada 1995. godine. Pohađao je I. gimnaziju u Osijeku koju je završio 2014. godine. Nakon toga upisuje preddiplomski studij računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. Preddiplomski studij završava 2018. godine, kada upisuje diplomski studij računarstva, smjer informacijske i podatkovne znanosti.

Abraham Kostić

_____