Razvoj generatora C koda u sklopu generatora testnog okruženja

Brkić, Dino

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

Sveučilišni studij diplomski studij

RAZVOJ GENERATORA C KODA U SKLOPU GENERATORA TESTNOG OKRUŽNJA

Diplomski rad

Dino Brkić

Osijek, 2021.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 17.09.2021.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime studenta:	Dino Brkić
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1040R, 06.10.2019.
OIB studenta:	18447908650
Mentor:	Izv. prof. dr. sc. Marijan Herceg
Sumentor:	
Sumentor iz tvrtke:	Marko Halak
Predsjednik Povjerenstva:	Izv. prof. dr. sc. Mario Vranješ
Član Povjerenstva 1:	Izv. prof. dr. sc. Marijan Herceg
Član Povjerenstva 2:	Izv.prof.dr.sc. Ratko Grbić
Naslov diplomskog rada:	Razvoj generatora C koda u sklopu generatora testnog okruženja
Znanstvena grana rada:	**Telekomunikacije i informatika (zn. polje elektrotehnika)**
Zadatak diplomskog rada:	Generator testnog okruženja nužna je komponenta cijelog procesa automatskog testiranja u automobilskoj industriji. Jedan dio spomenutog generatora je i onaj zadužen za generiranje datoteka izvornog koda softverskih komponenata sadržajem generiranih testova. Potrebno je analizirati trenutno dostupnu implementaciju jednog postojećeg rješenja te izdvojiti njezine prednosti i nedostatke. Na osnovu dobivenih zaključaka treba implementirati novo rješenja za generiranje datoteka izvornog koda softverskih komponenata sadržajem generiranih testova. Rješenje treba osmisliti na način da izvorni kod zauzima što manje prostora i da pritom uspješno generira izvorni kod koji se može izvršavati na elektroničkoj upravljačkoj jedinici
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	17.09.2021.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 22.09.2021.

Ime i prezime studenta:	Dino Brkić
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1040R, 06.10.2019.
Turnitin podudaranje [%]:	2

Ovom izjavom izjavljujem da je rad pod nazivom**: Razvoj generatora C koda u sklopu generatora testnog okruženja**

izrađen pod vodstvom mentora Izv. prof. dr. sc. Marijan Herceg

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD

Generator testnog okruženja nužna je komponenta cijelog procesa automatskog testiranja u automobilskoj industriji. U današnjoj automobilskoj industriji, pri izradi vozila, sve se više oslanja na elektroniku u vozilima za optimalno upravljanje motorom, korisničko sučelje, sustav kočenja, mjerenje razine goriva, itd. Takvo upravljanje se izvodi na specijaliziranim ugradbenim računalnim sustavima poznatijih kao elektroničke upravljačke jedinice (engl. *Electronic Control unit - ECU*). Izradom ECU javlja se potreba za testiranjem ispravnosti rada komunikacijskih kanala, kako unutar ECU, tako i unutar cijeloga automobila. Postupnim razvojem računalnih tehnologija, ECU su postali vrlo složeni, a broj komunikacijskih kanala može biti i više desetaka tisuća. Time se pojavljuje problem pisanja testova za svaki individualni kanal, koji bi u slučaju ručnog pisanja bio vrlo vremenski zahtjevan i čovjeku naporan posao. U svrhu skraćenja vremena testiranja komunikacijskih kanala osmislio se alat za testno okruženje, kojim se automatizira proces izrade programa za testiranje komunikacijskih kanala na ECU, prateći standarde arhitekture otvorenih auto motiv sustava (engl. ***Aut****omotive* ***O****pen* ***S****ystem* ***Ar****chitecture* - *AUTOSAR*) [1]. AUTOSAR je otvoreni standard za izradu programske podrške u automobilskoj industriji te ujedno i naziv partnerstva kojeg sačinjavaju vodeći proizvođači automobilskih dijelova, opreme i tehnologije. Alat osmišljen za automatizaciju testova naziva se generator testnog okruženja (engl. *Test Environment Generator - TEG*). TEG je alat koji se sastoji od svih potrebnih dijelova za uspješno generiranje testova, a to su: TEG okvir, TEG model parser, parseri TEG ulaznih podataka, rukovatelj bazom TEG podataka te najvažniji, generator izvornog C koda i generator liste testnih slučajeva.

Generator izvornog C koda testnog okruženja generira izvorni kod koji se izvodi na samom ECU-u, ali također je i dio testnog okruženja, koji uz testne liste služi za pokretanje samog automatski generiranog testa. Generiranje testnog okruženja izvodi se u minimalno tri glavna koraka:

1. Parsiranje svih datoteka koje sadrže informacije o kanalima ECU
2. Spremanje podataka u bazu podataka u lako dostupnom i čitljivom obliku
3. Generiranje izvornih kodova testnih modula i ostalih dijelova testnog okruženja na temelju podataka iz baze podataka

Zadatak ovoga rada je izraditi generator izvornog C koda u sklopu TEG-a. Kako bi se realizirao generator u sklopu TEG-a, potrebno je proučiti skup programskih alata, koji će omogućiti daljnji razvoj, te generiranje datoteka sa izvornim kodom koji se može izvršavati na ECU. Također

jedan od zadataka je proučiti postojeće rješenje generatora izvornog C koda te na temelju njegovih prednosti i nedostataka implementirati novo i efikasnije rješenje u programskom jeziku C++.

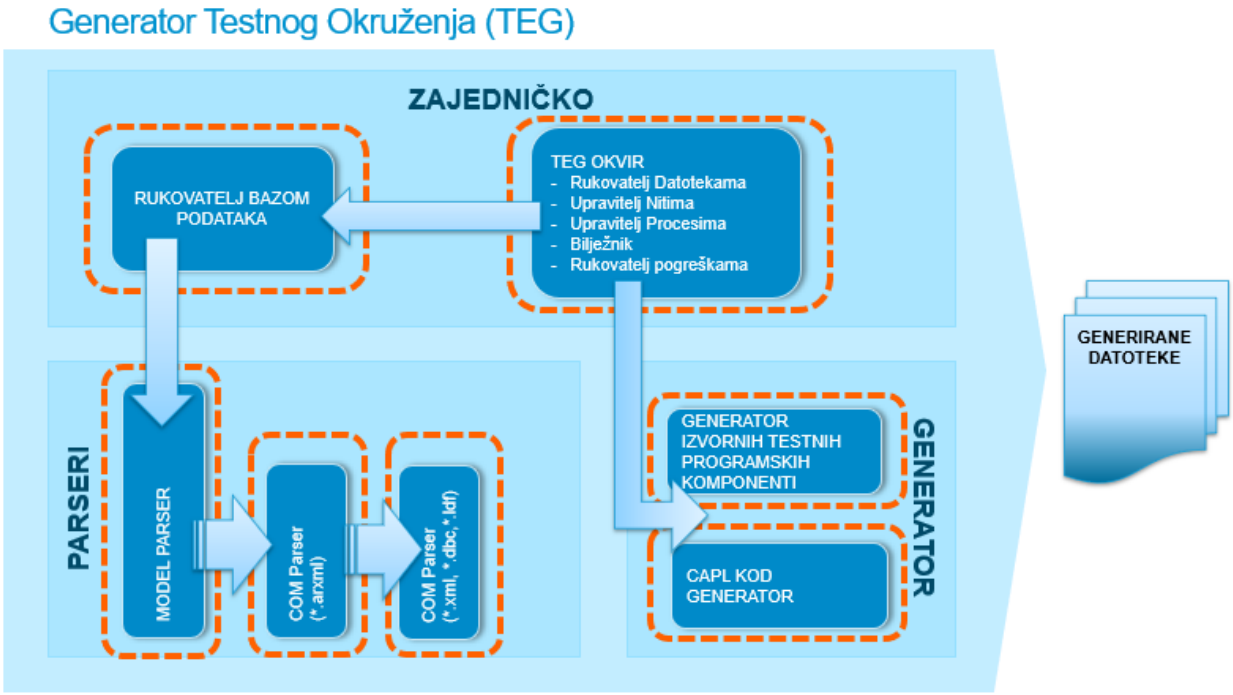
Generator izvornog koda bi stoga trebao imati više značajki:

- Generiranje izvornog koda s naglaskom na što manje zauzeće memorije
- Vremenska optimizacija procesa generiranja izvornog koda
- Generiranje izvornog koda korištenjem TEG okvira i rukovatelja baze podataka

U ovome radu najprije će se usporediti rješenje iz ovoga rada s već postojećim te će biti istaknute prednosti ovoga rješenja u odnosu na postojeće. Nakon toga, detaljno će biti opisane sve značajke, moduli i funkcionalnosti TEG generatora C koda, a rješenje će biti opisano u obliku pseudo koda i dijagrama toka programa. Na kraju rada bit će dan osvrt na cilj ovoga zadatka i postignutih rezultata.

2. GENERATOR TESTNOG OKRUŽENJA

Generator testnog okruženja je alat razvijen u programskom jeziku *Python* (verzija 2.7), kojeg je razvila tvrtka TTTech. Glavna zadaća TEG-a je da na temelju podataka komponenata iz ulaznog modela TEG-a generira datoteke izvornog C koda (Slika 2.1). Podaci modela su najčešće pohranjeni u modificiranim XML datotekama koje se nazivaju ARXML datoteke, te se njihov sadržaj parsira i sprema u bazu podataka, na temelju čega se popunjavaju implementacijski predlošci koji će naposljetku predstavljati generiranu datoteku s izvornim C kodom. Trenutna verzija TEG-a sastoji se od većeg broja datoteka napisanih u programskim jezicima *Python* i CAPL, koje pomažu u generiranju koda i za pokretanje testnih modula.



Slika 2.1: *Blok dijagram konceptualne sheme TEG-a* [2]

Tijekom rada, TEG generira sljedeća četiri testna okruženja propisana AUTOSAR standardom:

1. Sučelja u okruženju izvođenja (engl. *Runtime Environment - RTE*) – služe za komunikaciju unutar ECU-a
2. Sučelje za sabirnicu mreža u području upravljača (engl. *Controller Area Network - CAN*) – služi za vanjsku komunikaciju sa ostalim ECU-ima te definiranje kritičnih vremena izvršavanja.
3. Testovi izdržljivosti sustava (engl. *Persistency*) – testiranje postojanosti podataka u slučaju kvara ili nedostatka napona

4. Platformski testovi – sklopovlje, dijagnostika, nadzor aktivnosti, itd.

Trenutno rješenje TEG-a radi na način da se prvo pronađu unaprijed definirane putanje ulaznih podataka TEG-a u *config.ini* datoteci. Nakon toga se parsiraju podaci modela i ostalih XML i ARXML datoteka te se njihov sadržaj sprema u bazu podataka. Zatim se analizira baza podataka i povezuju se signali i podaci iz modela. Nakon toga se na temelju podataka u bazi generiraju programske komponente u obliku C koda, testni moduli za navedena četiri testna okruženja i testni kod koji se izvršava za svaki signal na sabirnici.

2.1. Nedostaci trenutnog rješenja

Kako je prethodno spomenuto, postojeće rješenje generatora testnog okruženja pisano je u skriptnom jeziku *Python*, verzija 2.7. Iako *Python* omogućava mnoge mogućnosti te je vrlo jednostavan za pisanje programa, rješenje napisano u *Pythonu* ima svoje nedostatke.

Glavni nedostatak *Pythona* u odnosu na C++ programski jezik jest njegova brzina. *Python* programski jezik koristi interpretator pri prevođenju koda u računalu razumljiv oblik. *Python* interpretator koristi paradigmu dinamičkog prevođenja linije po linije u stvarnome vremenu (engl. *Just In Time - JIT*), pri čemu gubi dodatno vrijeme za interpretiranje linije po linije izvornog koda. Prema tome, pisanje velikih programa koji imaju puno linija koda može narušiti brzinu tijekom izvršavanja *Python* programa.

Još jedna velika mana *Pythona* kao programskog jezika, koja također utječe na performanse programa, je mehanizam zaključavanja globalnog interpretatora (engl. *Global Interpreter Lock - GIL*) [3]. GIL omogućava izvršavanje *Python* programa u samo jednoj niti u vremenu, čak i kada postoji mogućnost rada u više niti u više jezgara na procesoru istovremeno. Kao rezultat toga, izvođenje *Python* programa u više niti je često sporije, nego u jednoj niti, pa se više isplati u *Pythonu* ne koristiti mogućnost višenitnog rada [4]. Trenutno rješenje TEG-a, koje je implementirano u *Python* jeziku koristi mehanizme pokretanja programa u više procesa (engl. *Multiprocessing*), što dodatno povećava složenost same implementacije.

Iako *Python* pošteđuje programere ručnog upravljanja memorijom, jer koristi sakupljač smeća (engl. *Garbage Collector*), ipak nije najbolji izbor za memorijski zahtjevne probleme, pogotovo probleme koji se trebaju izvršavati na sklopovlju sa ograničenim resursima, gdje je svaki bajt u memoriji dragocjen. Programer, kada piše kod u *Pythonu*, nema kontrolu pri rukovanju memorijom. Konačno, zadnja mana, koja se odnosi na sam *Python* jezik je ta što programi

napisani u *Pythonu* nisu pogodni za komercijalne uporabe u poslovnom sektoru. Naime, program napisan u *Pythonu* zahtjeva posjedovanje izvornog koda (*.py* datoteka), kako bi *Pythonov* interpretator mogao izvršavati linije naredbi. Takav pristup može izložiti program neželjenim izmjenama, jer ne razdvaja izvršnu verziju od izvornog koda, kao što je to slučaj kod programskih jezika koji prevode izvorni kod u strojni jezik, kao C++.

Ipak, činjenica da je prethodno rješenje pisano u *Python* jeziku, nije jedini nedostatak, pogotovo jer je *Python* jezik dovoljno brz na današnjim računalima opće namjene u većini slučajeva. Trenutno rješenje također ima nedostataka u implementaciji te u izboru verzije *Python* jezika, kao i korištenje nekih njegovih alata, odnosno modula.

Prvi nedostatak trenutnog rješenja jest činjenica da je program pisan u verziji *Pythona* koja je zastarjela i više se ne ažurira. Kao posljedica toga, *Python* verzija 2.7 ne može koristiti novije i naprednije biblioteke, koje su napisane za noviju verziju *Python* jezika (verzija 3), nego se mora oslanjati na zastarjele biblioteke, koje imaju nekih ne ispravljenih grešaka, ili nedovršenih i ne poboljšanih funkcionalnosti.

Nadalje, drugi nedostatak koje ima trenutno rješenje je da se za spremanje podataka u TEG-ovu bazu podataka koristi biblioteka *pickle,* kao alat za pohranu podataka. Biblioteka *pickle* je *Pythonov* modul, koji implementira binarne protokole za serijalizaciju i deserijalizaciju *Pythonovih* struktura podataka i objekata. Tijekom procesa serijalizacije *Pythonov* objekt, tj. hijerarhija tog objekta pretvara se u binarni tok podataka [5]. Iako je *pickle* brz i lagan za korištenje, ipak nije najpouzdanije rješenje za spremanje podataka. Naime, biblioteka *pickle* je ranjiva na izmjene. Drugim riječima, podaci koji su pohranjeni pomoću *pickle* biblioteke mogu se neovlašteno čitati, te se mogu izmijeniti. Moguće je čak i nadodati vlastiti kod u *pickle* datoteku, koji se nesmetano može izvršiti tijekom učitavanja *pickle* datoteke tijekom rada *Python* programa, jer *pickle* biblioteka nema nikakve mehanizme detekcije i sprječavanja izmijenjenog sadržaja *pickle* datoteke [6]. Zbog ovog velikog nedostatka *pickle* biblioteke, dovoljno je da neovlašteni korisnik ima pristup *pickle* datoteci, u koju može dodati svoj kod i može doći do toga da *Python* program počne izvršavati zadatke, za koje nije predviđen te tako uzrokovati štetu na računalu, ovisno koliko je zloćudan prikriveni kod u izmijenjenoj (engl. *hacked*) *pickle* datoteci.

Također, treći nedostatak u implementaciji trenutnog rješenja je da nema načina da se potvrdi je li cijeli opseg generiranog sadržaja ispunjen. Drugim riječima, nema načina da se provjeri je li svaka stavka uzeta iz modela, što znači da postoji mogućnost da nije uvijek generirano sve što je moglo biti u modelu, koji je dinamično pohranjen u ARXML datoteci. Ovaj nedostatak proizlazi

iz činjenice da trenutni parser modela nije dinamičan, nego svaki put kada se ARXML datoteka ažurira novim sadržajem modela, potrebno je nadograditi parser da može obrađivati nove stavke, odnosno pokrpati trenutne nedostatke koje ima (engl. *Patch*). Naravno, prilikom izrade zakrpi za nove stavke modela, postoji mogućnost da one stare ne budu pročitane. Međutim, ovaj problem neće biti riješen u ovome radu.

Još jedan problem je da sama pretraga potrebnih datoteka, koje su potrebne kako bi TEG izradio konačne izlazne datoteke, nije dinamična, nego sve putanje potrebnih datoteka ručno su upisane u jednu XML datoteku. Ovo predstavlja mogući problem, jer korisnik ne drži uvijek sve potrebne datoteke na istome mjestu. Također, postoji mogućnost da dođu nove datoteke, koje je potrebno obraditi, kao i one datoteke, koje nisu više potrebne. Drugim riječima, svaki put kada se dodaje nova datoteka, ili ako se uklanja, promjene je potrebno ručno zapisati u pripadajuću XML 6 datoteku, što je naporan proces, te je takav ručni pristup podložan pogreškama.

Konačno, četvrti nedostatak u implementaciji trenutnog rješenja je da sav generirani kod, koji je nastao kao rezultat TEG-ove obrade ulaznih datoteka, previše redundantan, odnosno nije optimiziran. Naime, u generiranom kodu postoji puno dijelova koji se nepotrebno ponavljaju i nepotrebno nadodavaju. Posljedica toga je da prilikom izvršavanja generiranih kodova nepotrebno se troši vrijeme na višestrukom ispitivanju pojedinog kanala na ECU. Također, vrijeme generiranja datoteka izvornog C koda je do nekoliko desetaka minuta što nije pogodno za testiranje i konačno rješenje.

Budući da trenutno rješenje, napisano u *Pythonu* 2.7, ima velik broj nedostataka, samo će neki od tih nedostataka biti riješeni u ovome radu pomoću generatora izvornog C koda. Za ostale nedostatke, koji se tiču parsiranja podataka i rukovanjem baze podataka, potrebno je implementirati dijelove TEG-a specijalizirane za rješavanje spomenutih nedostataka.

3. IZRADA TEG GENERATORA IZVORNOG C KODA

Generator izvornog C koda je jedna od zadnjih stavki u sklopu TEG-a te obavlja važan zadatak kojeg generator testnog okruženja treba odraditi. Predstavlja neizostavan dio programa jer je neophodan alat za ispravno i učinkovito obavljanje zadaće generatora testnog okruženja. Budući da je generator izvornog C koda među zadnjim modulima u TEG-u, oslanja se na ostale dijelove TEG-a i implementacija mu ovisi isključivo o dostupnim podacima proslijeđenih iz ostalih dijelova programa. Tijekom izrade generatora izvornog C koda funkcionalnosti su podijeljene unutar samog modula za generiranje te ih se može opisati u više značajnih funkcionalnosti.

Generator izvornog C koda sastoji se od tri veća modula:

1. Parsiranje fragmenata programskih komponenti i konfiguracijske tablice generatora te njihovo spremanje u memoriju.
2. Zamjena vrijednosti semantičkih oznaka unutar fragmenata sa relevantnim vrijednostima iz TEG baze podataka.
3. Popunjavanje implementacijskih predložaka odgovarajućim fragmentima na temelju podataka iz TEG baze podataka.

Parsiranje fragmenata programskih komponenti te njihovo spremanje u memoriju obuhvaća dio modula kojim se implementira parsiranje ručno pisanih sučelja u obliku fragmenata koji se smještaju u određena mjesta unutar implementacijskih predložaka. Parsiranje konfiguracijske tablice generatore te njeno spremanje u memoriju obuhvaća dio modula kojim se implementira parsiranje podataka sadržanih u konfiguracijskog tablici generatora, koji služe kao jedan od načina uvjetovanja smještanja fragmenata u određene implementacijske predloške i određena mjesta unutar implementacijskog predloška. Zamjena vrijednosti semantičkih oznaka unutar fragmenata s relevantnim vrijednostima iz baze podataka obuhvaća modul kojim se implementira zamjena podataka fragmenta podacima iz baze podataka relevantnim za određenu programsku komponentu. Popunjavanje implementacijskih predložaka odgovarajućim fragmentima na temelju podataka iz baze podataka obuhvaća modul kojim se implementira generiranje datoteke izvornog C koda na temelju prethodnih modula i na temelju ostalih podataka iz baze podataka.

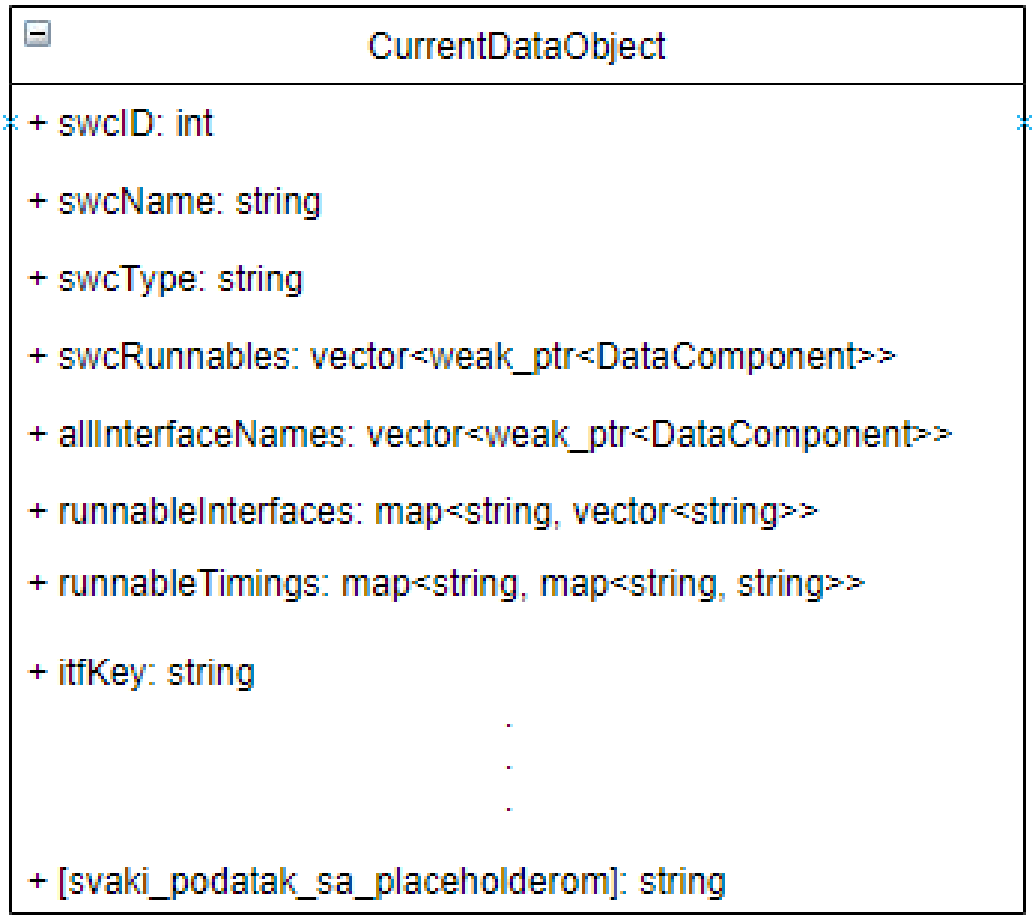
Ovakav način implementacije omogućuje modularnost funkcionalnosti samog generatora u slučaju potrebe za većim izmjenama. Budući da se ovaj program radi u C++ programskom jeziku, bilo je potrebno uzeti u obzir koji alati i biblioteke će se koristiti. Generator izvornog C koda, koji treba raditi uz pomoć ostalih dijelova programa, koristi biblioteke TEG okvira, TEG

rukovatelja bazom podataka te bibliotekama parsera modela, komunikacijskih matrica te ostalih XML datoteka u projektu. Također, jedna od biblioteka koju koriste parseri modela, komunikacijskih matrica i ostalih XML datoteka, *tinyxml2* se koristi i za parsiranje konfiguracijske tablice generatora. Kao sam jezik koristi se C++ programski jezik standard 17 (C++17), jer sadrži neke biblioteke i alate koje starije verzije ne posjeduju ili su nadograđene.

3.1. Pristup pri izradi generatora izvornog C koda

Generator izvornog C koda kao jedan od dijelova TEG-a, u C++ programskom jeziku osmišljen je u obliku više korisničkih tipova podataka sadržanih u većoj jedinstvenoj strukturi koji će obavljati većinu funkcionalnosti unutar sebe. Postojeći TEG koristi mnoštvo korisnički definiranih podataka koji se u *Pythonu* mogu implementirati vrlo lako i dinamično, što u programskom jeziku C++ nije slučaj. Također, *Python* TEG rješenje zbog velikog broja klasa unosi znatnu redundanciju u spremanju podataka. Time može nastati velika količina dupliciranog koda ili jednostavno praznih spremnika, ali svejedno alociranih, što unosi memorijska ograničenja.

Iz spomenutih nedostataka osmišljena je jedna klasa generatora izvornog C koda, koja u sebi sadrži sve potrebne podatke za generiranje jedne programske komponente. Konceptualno, programske komponente se generiraju u petlji, gdje jedna iteracija petlje predstavlja generiranje jedne programske komponente. Korištenjem jedinstvene klase generatora izvornog C koda ograničava se spremanje redundantnih i ponavljajućih podataka u memoriji. Time se unosi memorijska efikasnost kroz iteracije te svi mogući podaci nisu trenutno dostupni, već kada su potrebni. Uz glavnu klasu generatora izvornog C koda napravljena je pomoćna struktura koja se koristi kao spremnik za podatke koji se učitavaju iz baze podataka te su njeni podaci relevantni samo za trenutnu programsku komponentu u tekućoj iteraciji. Također, pomoćna struktura, slično kao glavna klasa generatora izvornog C koda, nastoji uvesti efikasnost među podacima koji se povlače iz baze podataka. Kako je pomoćna struktura dio glavne klase generatora izvornog C koda, dodatna efikasnost se unosi nad kompletnom klasom. To se ostvaruje ponavljajućim upitima nad bazom podataka za svaku jedinstvenu programsku komponentu, te se time osigurava da nema redundantnih ili dupliciranih podataka u memoriji tokom generiranja programskih komponenti.

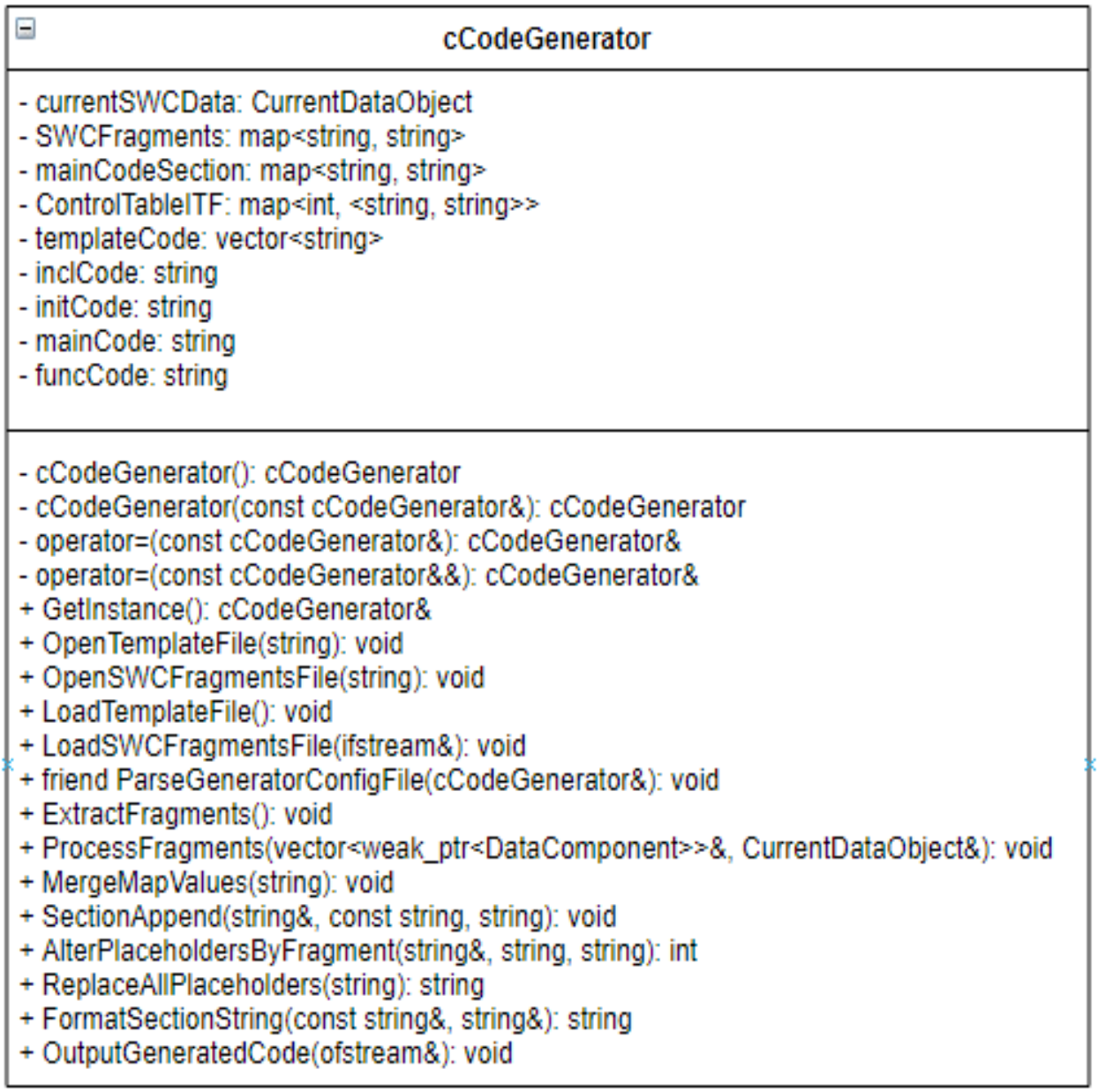


Slika 3.1: *Prikaz pomoćne strukture za spremanje trenutnih podataka o programskim komponentama*

Iz slike 3.1 može se vidjeti da struktura trenutnih podataka sadrži uglavnom složene tipove podataka koji spremaju:

- imena modula koji se izvode po programskoj komponenti
- imena svih sučelja u jednoj programskoj komponenti
- imena svih sučelja po modulima koji se izvode
- vremena kritičnih završetaka i prosječnog izvršenja po modulima koji se izvode

Isprva izgleda da ima više složenih tipova podataka no zapravo ima više jednostavnih tipova podataka u strukturi, tipa cjelobrojnih brojeva, brojeva sa dvostrukom preciznošću te polja znakova. Takvi podatci su označeni u općem obliku na slici 3.1 sa [*svaki_podatak_sa_placeholderom*] te je kao primjer napisan *itfKey* koji sadržava ključ sučelja za zamjenu u blokovima koda za trenutnu programsku komponentu. Takvih podataka ima blizu stotinu te nisu svi navedeni, ali svima je uloga čuvanje podataka relevantnih za trenutnu programsku komponentu, kako bi se mogli zamijeniti podaci unutar fragmenata, gdje je to naznačeno.



Slika 3.2: *Prikaz klase za generiranje izvornog C koda*

Slika 3.2 prikazuje strukturu klase koja sadrži sve potrebne podatke i apstrakcije za generiranje izvornog C koda. Pošto je klasa složena, napravljena je u obliku obrasca stvaranja *singleton-a*. *Singleton* je obrazac kojim se ograničava instanciranje više od jednog objekta klase te se na svaki pokušaj kreiranja više od jednog objekta uvijek vraća referenca na prvi kreirani objekt. Stoga je klasa generatora izvornog C koda kao „skupa“ klasa implementirana kao *singleton*. S time na umu, klasi su dodatno onemogućeni operatori dodjeljivanja referenciranih objekata generatora i dvostruko referenciranih objekata generatora izvornog C koda. Klasa sadrži sve atribute potrebne za generiranje koda po implementacijskim predlošcima, vidljivih na slici 3.2. Tablicom 3.1 dan je detaljan opis svih atributa klase.

Ime varijable	Opis
currentSWCData	Objekt pomoćne strukture koja sadrži sve podatke relevantne za trenutnu programsku komponentu
SWCFragments	Mapa koja sadržava uređene parove svih isparsiranih fragmenata. – (ime_fragmenta, blok_koda)
mainCodeSection	Mapa koja sadržava uređene parove komada koda koji se dodaje u module koji se izvode. – (ime_modula_koji_se_izvodi, pripadajući_kod)
ControlTableITF	Mapa koja sadržava uređene parove indeksa i mape koja sadržava uređene parove vrijednosti sučelja koji se nadodaju u module koji se izvode. – (indeks, (ime_sučelja, blok_koda))
templateCode	Vektor znakovnih polja koji sadržava cijelu učitanu datoteku implementacijskog predloška
inclCode	Znakovno polje koje sadržava sav kod koji pripada u dio za uključivanje zaglavlja u implementacijskom predlošku
initCode	Znakovno polje koje sadržava sav kod koji pripada u dio za inicijalizaciju u implementacijskom predlošku
mainCode	Znakovno polje koje sadržava sav kod koji pripada u glavni izvršni dio u implementacijskom predlošku
funcCode	Znakovno polje koje sadržava sav kod koji pripada u dio za funkcijske definicije u implementacijskom predlošku

Tablica 3.1: *Tablica atributa klase generatora izvornog C koda*

Klasa također sadrži sve apstrakcije potrebne za generiranje izvornog C koda unutar implementacijskih predložaka. Tablicom 3.2 dan je detaljan opis svih funkcija klase.

Ime funkcije	Opis
cCodeGenerator	Zadani konstruktor – koristi automatsku inicijalizaciju svih varijabli unutar klase
operator=	Preopterećen operator pridruživanja – definiran samo kako bi se onemogućio i time spriječio dodatno referenciranje na više od jedne reference
GetInstance	Funkcija singletona koja kreira objekt iz zadanog konstruktora ako on već ne postoji
OpenTemplateFile	Funkcija koja otvara datoteku implementacijskog predloška koristeći rukovatelj datotekama TEG okvira
OpenSWCFragmentsFile	Funkcija koja otvara datoteku fragmenata programske komponente koristeći rukovatelj datotekama TEG okvira
LoadTemplateFile	Funkcija koja učitava sve podatke datoteke implementacijskog predloška u trenutni spremnik memorije
LoadSWCFragmentsFile	Funkcija koja učitava sve podatke datoteke fragmenata programske komponente u trenutni spremnik memorije
ParseGeneratorConfigFile	Funkcija koja parsira podatke konfiguracijske tablice generatora i sprema ih u trajni spremnik memorije u obliku mape
ExtractFragments	Funkcija koja parsira fragmente iz trenutnog spremnika fragmenata te ih sve sprema u trajni spremnik memorije u obliku mape
ProcessFragments	Funkcija koja filtrira i puni varijable znakovnih polja svim kodom koji pripadaju u određene dijelove u implementacijskom predlošku
MergeMapValues	Funkcija koja stavlja kôd napunjenih varijabli znakovnih polja u predodređena mjesta u implementacijskim predlošcima

SectionAppend	Funkcija koja u formatiranom obliku nadodaje fragmente u varijablu znakovnog polja
AlterPlaceholdersByFragment	Funkcija koja mijenja niz znakova zadanim nizom znakova unutar niza znakova
ReplaceAllPlaceholders	Funkcija koja koristi funkciju *AlterPlaceholdersByFragment* za mijenjanje svih mogućih kombinacija zadanih znakova unutar fragmenata.
FormatSectionString	Funkcija koja vraća formatirani oblik fragmenta
OutputGeneratedCode	Funkcija koja ispisuje generiran kod u datoteku koristeći rukovatelj datotekama TEG okvira

Tablica 3.2: *Tablica funkcija klase generatora izvornog C koda*

3.2. Implementacija parsera fragmenata programskih komponenti

Parser fragmenata programskih komponenti jedan je od ulaznih dijelova generatora izvornog C koda. Za pripremu podataka parseru prvo se moraju učitati podaci koji sadrže fragmente generatora testnog okruženja. Učitavanje podataka je dio koji se odvija unutar klase za generiranje izvornog C koda te koristi rukovatelj ulaznog toka proslijeđenog od TEG okvira. Datoteke s fragmentima programskih komponenti sadrže blokove koda koji imaju određeni naziv i imaju pripadajuća mjesta unutar implementacijskih predložaka. Takve datoteke su ručno napisane te se njeni fragmenti koriste za višestruko umetanje unutar implementacijskih predložaka.

Fragmenti unutar datoteka su podijeljeni na četiri dijela:

1. Blokovi koda koji pripadaju u zaglavlje implementacijskih predložaka (engl. *Include*)
2. Blokovi koda koji pripadaju u inicijalizacijski dio implementacijskih predložaka (engl. *Initialization*)
3. Blokovi koda koji pripadaju u glavni izvršni dio implementacijskih predložaka (engl. *Main*)

4. Blokovi koda koji pripadaju u dio funkcijskih definicija unutar implementacijskih predložaka (engl. *Function Implementations*)

```
1:
// ####### TEMPLATE [ime_regije_fragmenta] ################
##################
2:
3: // ####### TEMPLATE [ime_fragmenta]
4:
5: /*Blok koda za uključivanje zaglavlja*/
6:
7: // ####### END
```

Slika 3.3: *Primjer izgleda generičkih fragmenata*

Iz slike 3.3 može se vidjeti kako je svaki fragment, koji se nalazi u datoteci, opisan:

- Oznakom regije fragmenta (uvijek počinje s //###### TEMPLATE i završava s redom *hashtag* znakova (#))
- Oznakom imena samoga fragmenta (uvijek počinje s //###### TEMPLATE)
- Blokom koda koji fragment izvršava
- Oznakom za kraj fragmenta (uvijek počinje s //###### END)

U postojećem *Python* rješenju TEG-a, datoteke koje sadrže fragmente programskih komponenti u sebi ne sadrže oznaku za kraj fragmenta, nego je to nešto što je naknadno dodano u svrhu lakšeg parsiranja fragmenata u C++ programskom jeziku.

Učitavanjem datoteka fragmenata programskih komponenti u memoriju započinje se sam proces parsiranja individualnih fragmenta. Sve datoteke s fragmentima programskih komponenti se učitavaju u petlji te se učitavaju u trenutni spremnik, koji sadrži cijeli sadržaj učitane datoteke. Nakon učitavanja u trenutni spremnik, nad njime se poziva funkcija za parsiranje fragmenata nad svakom datotekom. Bitno je napomenuti da postoji poseban slučaj fragmenata (Slika 3.4) u kojima se, umjesto bloka koda koji izvršava, nalazi oznaka sadržavanja identične vrijednosti fragmenta po imenu fragmenta.

Za parsiranje fragmenata korišten je pristup izvlačenja koda između dvaju oznaka uz uvjet da se svakome fragmentu mora parsirati i pripadajuće ime (Slika 3.5). Znajući da svaki fragment počinje s početnom oznakom te završava sa završnom oznakom, iterira se svaki redak u

spremniku koji sadrži sadržaj datoteke s fragmentima programskih komponenti, te se ispituje ako se u trenutnom retku nalazi oznaka za početak fragmenta. Ako se nalazi oznaka za početak fragmenta, parsira mu se ime, te se podiže zastavica za signalizaciju da se u sljedećim iteracijama nalazi blok koda fragmenta. Dok je zastavica podignuta, redci bloka koda za trenutni fragment se spremaju u trenutni spremnik te nadodaju jedan na drugi kako se prolaze redci. Kada iteracija dostigne redak koji sadrži oznaku za završetak fragmenta, zastavica za izlazak iz fragmenta se spušta, te se uređeni par – (ime_fragmenta, pripadajući_blok_koda) sprema u trajni spremnik memorije u obliku mape.

Poredak parsiranja datoteka fragmenata programskih komponenti je također bitan jer su potrebne dodatne operacije nad fragmentima iz datoteka sa zamjenskim fragmentima programskih komponenti specifične za verziju programa koji se izvršava. Naime, naknadne datoteke sa zamjenskim fragmentima programskih komponenti mogu imati fragmente istoga naziva kao u datoteci osnovnih testnih fragmenata programskih komponenti. Takve fragmente je potrebno prepisati sa onima iz datoteka osnovnih testnih fragmenata.

```
1: // ####### TEMPLATE [ime_fragmenta]
2:
// ~~~~ equal to :  [ime_nekakvog_drugog_fragmenta]
3:
4: // ####### END
```

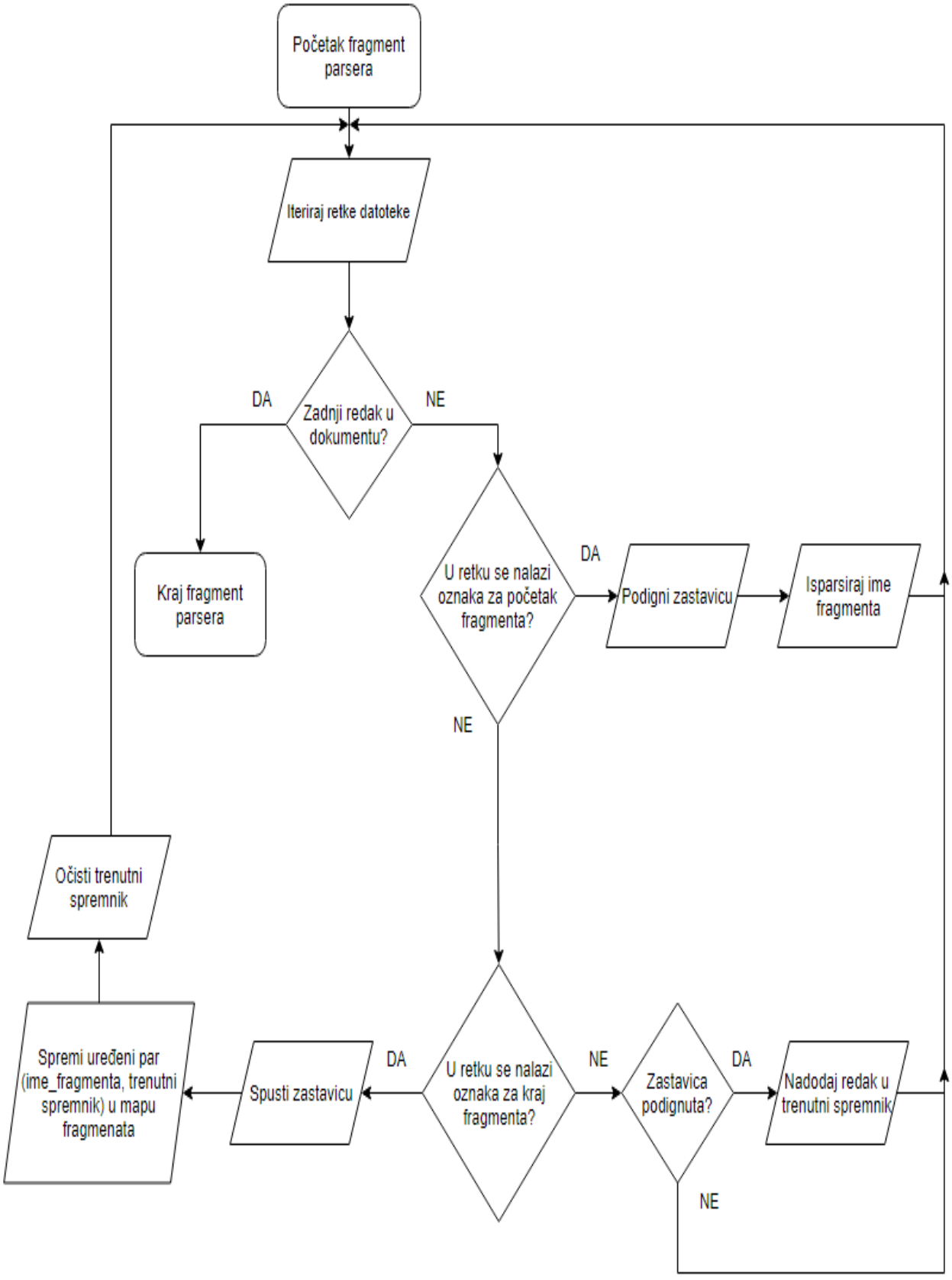
Slika 3.4: *Poseban slučaj fragmenta sa vrijednosti drugog fragmenta*

Prilikom procesiranja fragmenata provjerava se je li njegova vrijednost sadrži oznaku koja označava da fragment ima vrijednost drugoga fragmenta. Ako se dođe do takvog fragmenta, njegovo ime unutar oznake se parsira, te se vrijednost trenutnog fragmenta prepisuje sa vrijednošću fragmenta po ključu imena isparsiranog iz oznake (Slika 3.6).

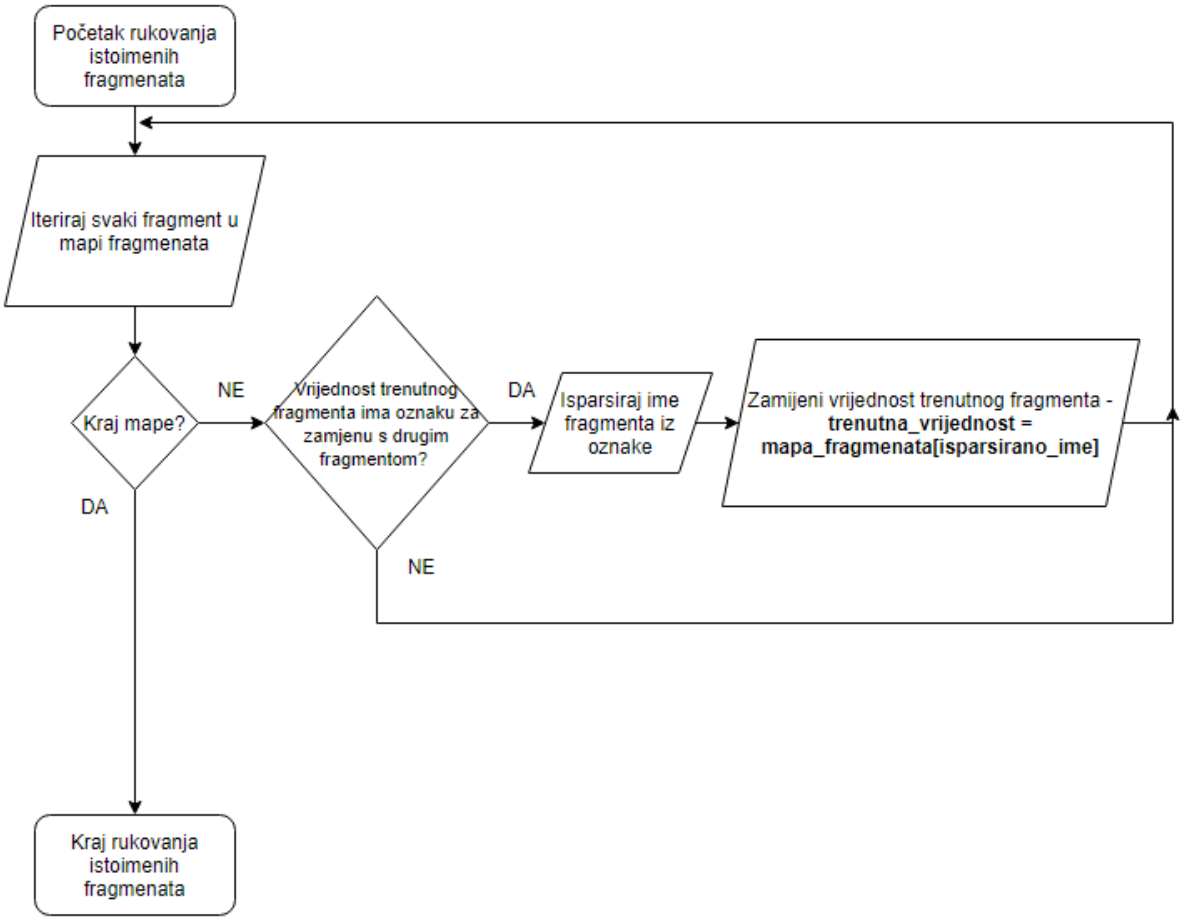
Redoslijed parsiranja datoteka programskih komponenti je:

- Osnovna datoteka testnih fragmenata
- Datoteka testnih fragmenata specifičnih za platformu
- Datoteka prilagođenih testnih fragmenata
- Datoteka zamjenskih testnih fragmenata,

gdje redoslijed poslije parsiranja osnovne datoteke testnih fragmenata nije bitan, ali je nužan.



Slika 3.5: *Algoritam parsiranja datoteka fragmenata programskih komponenti*



Slika 3.6: *Postupak rukovanja fragmenata u posebnim slučajima*

3.2.1 Parsiranje konfiguracijske tablice generatora

Konfiguracijska tablica generatora (Slika 3.7), kao jedna od ulaznih komponenti generatora izvornog C koda, važan je dio za generiranje koda u implementacijske tablice. Tablica sadrži indeksirane podatke za svaku verziju testnog modula tokom pokretanja programa. Podaci tablice zapisani su u obliku prijenosnog tipa podataka u XML formi te sadrže veze između sučelja koja se moraju izvršavati u određenim dijelovima implementacijskih predložaka. U narednom poglavlju, kada se spominje filtriranje fragmenata po modulima koji se izvode, baš podaci iz ove tablice određuju koja točno sučelja se moraju izvršavati te u kojim modulima koji se izvode. Tablica u sebi sadrži ugniježđene vrijednosti po vrijednostima indeksa, koji su poveznica podacima koji se dohvaćaju iz baze podataka. Indeks podatka za jedan tip sučelja sadrži u sebi podatke o:

- tipu sučelja

- bloku koda/sučelja koje se stavlja u dio implementacijskog predloška za uključivanje zaglavlja
- bloku koda/sučelja koje se stavlja u dio implementacijskog predloška za inicijalizaciju
- bloku koda/sučelja koje se stavlja u glavni izvršni dio implementacijskog predloška
- bloku koda/sučelja koje se stavlja u dio implementacijskog predloška za funkcijske definicije
- tipu filtra

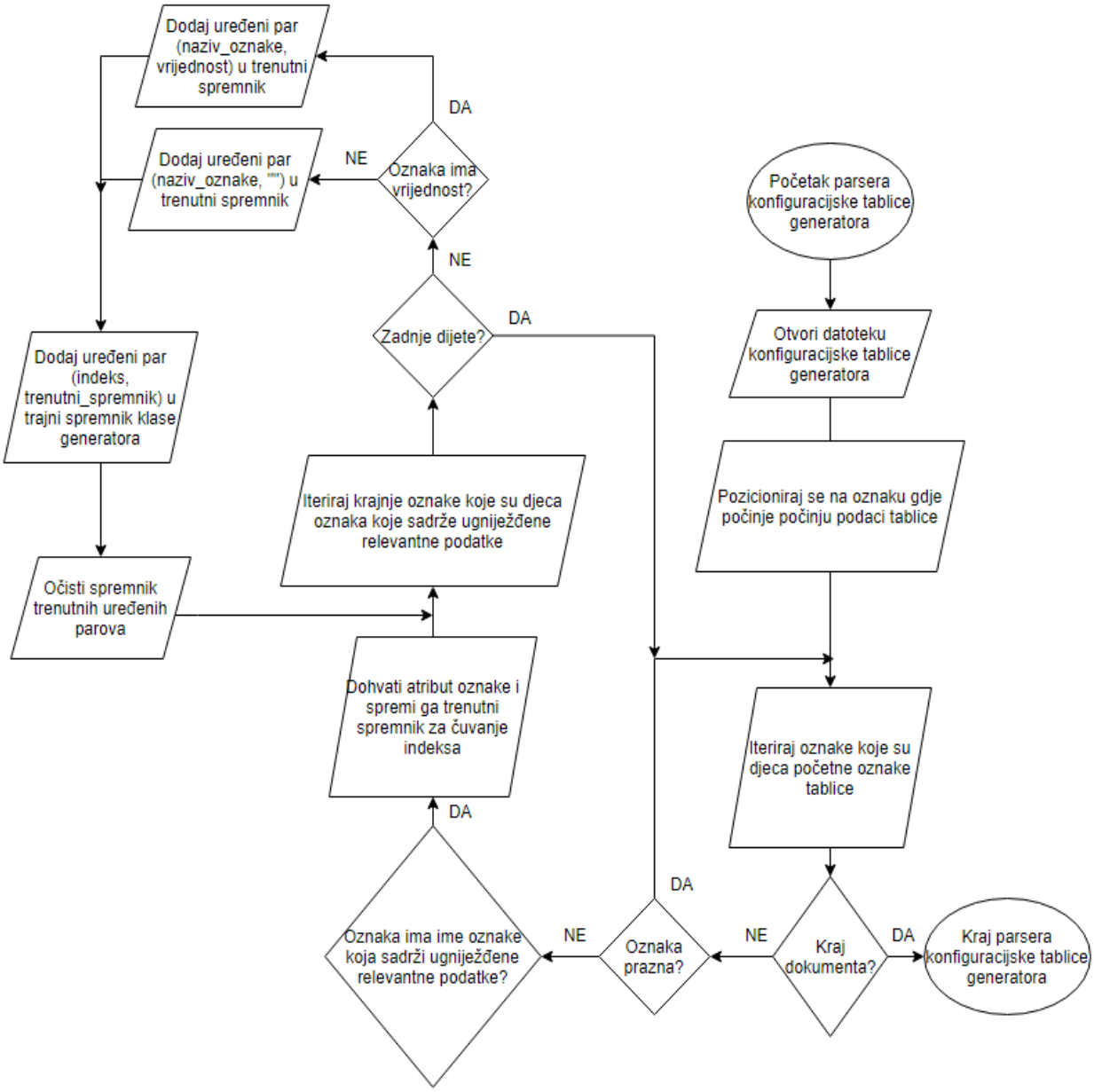
```
1: <VALUE idx="[broj_indexa]">
2: <TYPE>[tip_sučelja]</TYPE>
3: <INCL>[sučelje_koje_se_stavlja_u_dio_za_uključivanje_zaglavlja]</INCL>
4: <INIT>[sučelje_koje_se_stavlja_u_dio_za_inicijalizaciju]</INIT>
5: <FUNC>[sučelje_koje_se_stavlja_u_dio_za_funkcijske_definicije]</FUNC>
6: <MAIN>[sučelje_koje_se_stavlja_u_glavn_izvršni_dio]</MAIN>
7: <FILTERS>[tip_filtera]</FILTERS>
8: </VALUE>
```

Slika 3.7: *Prikaz oblika jednog podatka u tablici konfiguracije generatora*

Za parsiranje tablice konfiguracije generatora korištena je biblioteka koji koriste model parser, parser komunikacijskih matrica te ostalih XML dokumenata, *tinyxml2* za programski jezik C++. Kako ne postoji univerzalni parser za XML tip datoteka, moraju se manualno uz pomoć programskih sučelja biblioteke parsera, parsirati željeni podaci iz datoteke i spremati u predodređene spremnike.

Parser konfiguracijske tablice generatora (Slika 3.8) implementiran je u obliku jednostavne apstrakcije, koja je prijateljski dio klase generatora izvornog C koda, te vrši promjene nad kreiranom referencom iste. Funkcija počinje otvaranjem datoteke tablice konfiguracije generatora statičnom putanjom, koje su raspoređene projektom u smislene mape. Zatim se postavlja prva oznaka u XML dokumentu kao početna referenca te se iterira po ostalima. Tijekom iteracija oznaka u XML dokumentu provjerava se je li ime trenutne oznake početna oznaka bloka koji sadrži ugniježđene podatke specifičnih sučelja. U trenutku kada se pronađe tražena oznaka, parsira se njen atribut, koji sadrži vrijednost indeksa za trenutnu oznaku ugniježđenih podataka, te se sprema u trenutni spremnik. Time se započinje nova iteracija koja će prolaziti po oznakama po dubini unutar nađene početne oznake. Unutar dublje iteracije se ispituje sadržaj krajnjih oznaka te njihovih vrijednosti. U slučaju postojanja sadržaja unutar oznake, dohvaća se ime trenutne oznake i njena pripadajuća vrijednost, čime se kreira trenutni uređeni par njihovih vrijednosti u obliku: (naziv_oznake, vrijednost_oznake). Suprotno, ako

unutar oznake ne postoji vrijednost, dohvaća se samo ime trenutne oznake, dok se za njenu vrijednost stavlja prazno znakovno polje, te se kreira trenutni uređeni par njihovih vrijednosti slično prethodnom slučaju. Završetkom dublje petlje i nakon iteracija svih krajnjih oznaka, u mapu sa spremanje kontrolne tablice unutar klase generatora izvornog C koda, nadodaju se uređeni parovi indeksa i trenutnih uređenih parova dublje petlje sa njihovim nazivima oznaka i vrijednosti u obliku: (indeks, (naziv_oznake, vrijednost_oznake)). Nakon dodavanja uređenog para u mapu klase, čisti se trenutni spremnik uređenih parova te time završava proces parsiranja konfiguracijske tablice generatora.



Slika 3.8: *Algoritam parsiranja konfiguracijske tablice generatora*

3.3. Implementacija zamjene semantičkih oznaka unutar fragmenata

Fragmenti, kao podaci datoteke fragmenata implementacijskih predložaka, u sebi sadrže semantičke oznake varijabli kojima se vrijednost treba promijeniti ovisno o više uvjeta. Semantičke oznake su jedinstvenog oblika i prvotno su ručno napisane unutar fragmenata. Oblik semantičkih oznaka varijable je takav da oznaka sadržava:

1. Početak semantičke oznake – uvijek počinje sa: $###
2. Semantička vrijednost oznake – opisna vrijednost varijable, npr. *swc-name*
3. Završetak semantičke oznake – uvijek završava sa: ###$

```
1: // ####### TEMPLATE [ime_fragmenta]
2: // $###[opis_sučelja]###$
3:
4:  static void fInitItfBuffer_$###[ključ_sučelja]###$()
5:  {
6:  $###[ime_početnog_fragmenta_za_slobodno_izvršavanje]##
#$
7:  #if defined(test_init_$###[tip_podatka]###$_$###[ime_p
odatkovnog_elementa]###$)
8:      Sl_MemCpy( &var_$###{ključ_sučelja}###$, &init_$##
#[tip_podatka]###$_$###[ime_podatkovnog_elementa]###$, siz
eof($###[tip_podatka]###$) );
9:  #else
10:      Sl_MemSet( &var_$###[ključ_sučelja]###$, 0x00, si
zeof($###[tip_podatka]###$) );
11:  #endif
12:  $###[ime_završnog_fragmenta_za_slobodno_izvršavanje]#
##$
13:  }
```

Slika 3.9: *Prikaz generičkog fragmenta sa zadanim semantičkim oznakama varijabli*

Kao što je vidljivo slikom 3.9, fragment koji će eventualno biti stavljen u implementacijski predložak sadržava nekolicinu semantičkih oznaka unutar bloka koda, kojima vrijednost mora biti promijenjena kako bi se ostvarila pravilna funkcionalnost generiranog implementacijskog predloška. Vrijednosti kojima se mijenjaju semantičke oznake varijabli dolaze i formiraju se

isključivo i na temelju podataka iz baze podataka. Osnovni podaci poput imena programskih komponenti, imena modula koji se izvode su dijelovi baze koji su isprasirani model parserom TEG-a, te su takvi podaci statični kroz cijelo izvođenje programa. Podaci poput kritičnih vremena izvođenja i srednjih vremena izvođenja modula koji se izvode su dijelovi baze koji se mijenjaju kroz program i isparsirani su CAN parserom TEG-a. Također, podaci koji sadrže podatke za povezivanje komunikacijskih kanala, a u semantičkim oznakama varijabli imaju naznaku imena *signal*, dijelovi su baze koji se dinamički određuju tokom izvođenja programa i isparsirani su parserom komunikacijske matrice TEG-a. Svi takvi podaci potrebni su za zamjenu semantičkih oznaka varijabli te se moraju dohvatiti iz baze podataka i pripremiti u prihvatljivom obliku. Dohvaćanje podataka iz baze podataka obavlja se u glavnom izvršnom dijelu TEG-a te se dohvaćeni podaci kolektivno prosljeđuju u obliku pomoćne strukture koja sadrži sve relevantne podatke za trenutnu programsku komponentu. U glavnom izvršnom dijelu TEG-a upitima prema bazi dohvaćaju se podaci:

- Imena svih programskih komponenti TEG projekta
- Svi moduli koji se izvode po programskim komponentama
- Sva sučelja po svim programskim komponentama
- Sučelja po modulima koji se izvode, programskih komponenti
- Kritična vremena izvođenja i prosječna vremena izvođenja po modulima koji se izvode
- Tipovi podataka po programskim komponentama

Na temelju spomenutih podataka formiraju se podaci za zamjenu semantičkih oznaka varijabli augmentacijom raspoloživih podataka iz baze podataka. Zamjena semantičkih oznaka varijabli provodi se kroz tri povezane operacije, koje čine funkcionalni algoritam zamjene semantičkih oznaka varijabli po programskim komponentama. Prva i osnovna operacija (Slika 3.10) za zamjenu oznaka je zamjena polja znakova unutar polja znakova drugim poljem znakova. Operacija je vrlo jednostavna te u suštini mijenja riječ u rečenici drugom zadanom rečenicom. Parametri su joj: referenca na polje znakova u kojem se treba zamijeniti, znakovno polje koje se mijenja unutar reference na polje znakova i znakovno polje kojim će se zamijeniti vrijednost znakovnog polja unutar reference na polje znakova. Također, povratna vrijednost joj je logički tip podatka gdje u slučaju ako se desila zamjena polja znakova funkcija vraća istinu, u suprotnom vraća neistinu. Ta povratna vrijednost služit će ostalim operacijama za njihove proširene funkcionalnosti.

```
1: inline bool ZamijeniSubstring(referenca_na_polje
_znakova,
 znakovno_polje_koje_se_mijenja,
 znakovno_polje_kojim_se_mijenja)
2: {
3:     početna_pozicija = referenca_na_polje_znakov
a.pronađi(znakovno_polje_koje_se_mijenja);
4:
5:    Ako je (pozicija == -1)
6:        Vrati neistinu;
7:
8:    referenca_na_polje_znakova.zamijeni(početna_p
ozicija, znakovno_polje_koje_se_mijenja.duljina(),
znakovno_polje_kojim_se_mijenja);
9:    Vrati istinu;
10: }
```

Slika 3.10: *Pseudo kod funkcije za zamjenu pod polja znakova polja znakova*

Na temelju prve operacije implementira se druga operacija (Slika 3.11) koja proširuje mogućnosti zamijene polja znakova. Kako je vidljivo na Slika 3., fragment u sebi može imati više semantičkih oznaka varijabli. Semantička oznaka varijabli se u fragmentu može višestruko pojavljivati i unutar jednog retka fragmenta ili jednog cjelovitog izraza. Korištenjem samo prve operacije nad fragmentom promijenit će samo jednu semantičku oznaku varijable unutar fragmenta. Druga operacija koristi funkcionalnost prve operacije kako bi se omogućila zamjena svih mogućih oznaka unutar jednog fragmenta. Proširenje se očituje uvođenjem petlje nad prvom operacijom. Petlja se izvršava sve dok se u referenci fragmenta pronalazi semantička oznaka varijable. Svakim pronalaskom semantičke oznake varijable se izvršava prva operacija nad referencom fragmenta, koja također sprema svoju povratnu vrijednost i vraća ju unutar druge operacije. Slično kao u prvoj operaciji, druga operacija vraća povratni kod 1 ako je bilo zamjene semantičkih oznaka varijabli, te suprotno, 0 ako nije bilo zamjena. Također, povratna vrijednost prve operacije prosljeđuje se drugoj kako bi se omogućila pravilna funkcionalnost daljnjih operacija.

```
1: int ZamijeniOznakePoFragmentu(referenca_na_fragment,
 semantička_oznaka_varijable,
 znakovno_polje_kojim_se_mijenja)
2: {
3:     povratni_kod = 0;
4:     Sve dok (referenca_na_fragment.pronađi(semantička_oznaka_varijable) !=
-1)
5:     {
6:         Ako je (ZamijeniSubstring(referenca_na_fragment, znakovno_polje_koj
im_se_mijenja))
7:         {
8:             povratni_kod = 1;
9:         }
10:     }
11:     Vrati povratni_kod;
12: }
```

Slika 3.11: *Pseudo kod funkcije za zamjenu svih oznaka u jednom fragmentu*

Implementacijom druge operacije ostvaruju se sve potrebne funkcionalnosti kako bi se mijenjale semantičke oznake varijabli definiranih u trećoj operaciji, koja se nadovezuje na drugu operaciju. Treća operacija (Slika 3.12) je operacija koja vrši operaciju nad kopijom fragmenta umjesto reference na fragment. Iako nije optimalno, izuzetno je bitno koristiti kopiju fragmenta jer zamijene semantičkih oznaka varijabli nisu iste za svaki fragment i različite programske komponente. Ovisno o programskoj komponenti i modulima koji se izvode, semantičke oznake varijabli poprimaju svoje stvarne vrijednosti. Stoga fragmenti u trajnoj memoriji moraju ostati nepromijenjeni te spremni za generiranje ostalih implementacijskih predložaka. S time na umu, treća operacija zamijene semantičkih oznaka varijabli kao atribut prima kopiju fragmenta te vraća izmijenjeni fragment. Operacija je zamišljena kao rekurzija no napravljena je u obliku beskonačne petlje sa brojačem kao signalizacijom za prekid petlje. Operacija koristi beskonačnu petlju unutar koje se nalaze sve moguće semantičke oznake varijabli kroz sve moguće fragmente projekta. Unutar petlje poziva se druga operacija nad kopijom fragmenta, semantičke oznake varijable i relevantne zamjenske vrijednosti iz strukture podataka programskih komponenti, te se taj poziv ponavlja za svaku semantičku oznaku varijable izvučenu iz svih kolektivnih fragmenata TEG projekta, kojih se broji na nešto manje od jedne stotine. Svaka povratna vrijednost tokom poziva druge operacije unutar beskonačne petlje treće operacije dodaje se na brojač koji će signalizirati izlazak iz petlje. Ovakva funkcionalnost je implementirana jer postoji mogućnost da fragment u sebi sadrži semantičku oznaku varijable koja predstavlja vrijednost drugog fragmenta te nadalje u dubinu. Kako bi se osigurala zamjena svih takvih ugniježđenih semantičkih oznaka varijabli, koriste se povratne vrijednosti prethodnih operacija kako bi se detektirale promjene

unutar fragmenata te ako postoji daljnjih ugniježđenih promjena. Brojač se pri svakoj ponovnom ulasku u beskonačnu petlju resetira te time osigurava rekurzivnu funkcionalnost beskonačne petlje. Uvjet za prekidanje petlje je ako je vrijednost brojača jednaka nuli, tj. ako u svim prijeđenim dubinama fragmenta više nema zamjena, prekini petlju. Prekidom petlje vraća se vrijednost izmijenjenog fragmenta u obliku polja znakova.

```
1: string ZamijeniSveOznake(vrijednost_fragmenta)
2: {
3:    Sve dok (istina)
4:     {
5:         brojač = 0;
6:         brojač += ZamijeniOznakePoFragmentu(vrijednost_fragmenta, "$###swc-
name###$", strukturaTrenutnihPodataka.swcName);
7:
8:                                      .
9:                                      .
10:                                     .
11:                                     .
12:                                     .
13:                                     .
14:
15:         vrijeme = trenutno_vrijeme()
16:
17:         brojač += ZamijeniOznakePoFragmentu(vrijednost_fragmenta, "$###year###$", pretvori_u
_string(vrijeme->godina));
18:         brojač += ZamijeniOznakePoFragmentu(vrijednost_fragmenta, "$###month###$", pretvori_
u_string(vrijeme->mjesec));
19:         brojač += ZamijeniOznakePoFragmentu(vrijednost_fragmenta, "$###day###$", pretvori_u_
string(vrijeme->dan));
20:         brojač += ZamijeniOznakePoFragmentu(vrijednost_fragmenta, "$###hour###$", pretvori_u
_string(vrijeme->sat));
21:         brojač += ZamijeniOznakePoFragmentu(vrijednost_fragmenta, "$###minute###$", pretvori
_u_string(vrijeme->minuta));
22:         brojač += ZamijeniOznakePoFragmentu(vrijednost_fragmenta, "$###second###$", pretvori
_u_string(vrijeme->sekunda));
23:
24:                                     .
25:                                     .
26:                                     .
27:                                     .
28:                                     .
29:                                     .
30:
31:         Ako je (!brojač)
32:         {
33:             prekini_petlju;
34:         }
35:     }
36:
37:     Vrati vrijednost_fragmenta;
38: }
```

Slika 3.12: *Pseudo kod funkcije za zamjenu svih (samo par navedenih na slici) oznaka u fragmentu njegovim relevantnim podacima*

3.4. Implementacija smještanja fragmenata u odgovarajuća mjesta implementacijskih predložaka

Smještanje fragmenata u točna i odgovarajuća mjesta implementacijskih predložaka je među zadnjim funkcionalnostima potrebnim za uspješno generiranje datoteka izvornog C koda TEG-a. Implementacija smještanja fragmenata u odgovarajuća mjesta implementacijskih predložaka izvedena je u dva dijela:

- Procesiranje fragmenata
- Stavljanje fragmenata u trenutne spremnike implementacijskih predložaka

Nakon procesiranja i smještanja fragmenata u trenutne spremnike implementacijskih predložaka preostaje ispisivanje popunjenih implementacijskih predložaka u datoteku. Pod procesiranjem fragmenata podrazumijeva implementacija glavne logike povezivanja podataka fragmenata, podataka iz baze podataka i podataka relevantnih za trenutnu programsku komponentu. Takav način je potreban za povezivanje svih razdijeljenih podataka u jednu apstrakciju. Procesiranje fragmenata kao jedna od funkcionalnosti popunjavanja implementacijskih predložaka, implementira se kao jedna velika apstrakcija koja govori koji će se fragmenti smještati u implementacijski predložak te na koja mjesta unutar implementacijskog predloška. Pošto su za procesiranje fragmenata potrebni podaci svih prethodnih parsera TEG-a, apstrakcija za procesiranje fragmenata prima listu programskih komponenti i referencu na pomoćnu strukturu relevantnih podataka za trenutnu programsku komponentu. Kako je već spomenuto, pomoćna struktura će sadržavati sve potrebne podatke dohvaćene iz baze podataka te će se ti podaci koristiti za kreiranje logike procesiranja fragmenata za njihovo pravilno smještanje unutar implementacijskog predloška. Samo procesiranje fragmenata radi popunjavanje odgovarajućih fragmenata u trenutne varijable dijelova implementacijskih predložaka. Time se radi međukorak, koji omogućuje optimirano generiranje programskih komponenti bez potrebe za višestrukim otvaranjem implementacijskih predložaka ili direktnog nadodavanja koda u spremnik implementacijskog predloška u memoriji. Fragmenti se određenom logikom (Slika 3.13 i Slika 3.14) nadodavaju u znakovna polja koja sadržavaju sav kod koji pripada u: dio za uključivanje zaglavlja, dio za inicijalizaciju, glavni izvršni dio i dio za funkcijske definicije. Fragmenti koji se nadodaju u znakovna polja su promijenjeni fragmenti vraćeni funkcijom za zamjenu semantičkih oznaka varijabli.

```
1: Ako
je (uvjet_za_stavljanje_fragmenta_u_određeni_dio)
2: {
3:     određeni_dio.append(ReplaceAllPlaceholders(S
WCFragments["[ime_fragmenta]"]));
4: }
5: inače
6: {
7:     ostali_dijelovi.append(ReplaceAllPlaceholder
s(SWCFragments["[ime_fragmenta]"]));
8: }
```

Slika 3.13: *Pseudo kod nadodavanja fragmenata u spremnike znakovnih polja dijelova implementacijskih predložaka*

Tokom procesiranja fragmenata i definiranja logike za smještanje fragmenata u odgovarajuće spremnike dijelova implementacijskih tablica, koriste se razni hakovi i nepovezani uvjeti preuzeti iz logike postojećeg *Python* TEG-a, no potrebni za uspješno generiranje. Primjer takvih su jednostavni uvjeti tipa ako ime programske komponente završava nekim poljem znakova.

```
1: Ako
je (završava_sa(currentData.swcName, "[dio_imena_sw
ca]"))
2: {
3:     SectionAppend(inclCode, "[određeni_swc]", Re
placeAllPlaceholders(SWCFragments["[određeni_swc]"]
));
```

Slika 3.14: *Primjer nadodavanja fragmenta pod čvrsto ožičenim uvjetima*

Također, dijelovi implementacijskih predložaka uvijek počinju početnim fragmentom te se mora omogućiti da se on prvi smjesti na početak svakog dijela implementacijskog predloška (Slika 3.15). Takav je slučaj u svakom implementacijskom predlošku osim onih koji završavaju određenim imenom.

```
1: Ako je (završava_sa(currentData.swcName, "[dio_imena_swca]"))
2: {
3:     inclCode = "//---
 [ime_početnog_fragmenta_u_dijelu_uključivanja_zaglavlja] ---\n";
4:     inclCode.append(ReplaceAllPlaceholders(SWCFragments["[ime_početnog_fra
gmenta_u_dijelu_uključivanja_zaglavlja]"]));
5:
6:     initCode = "//--- [ime_početnog_fragmenta_u_dijelu_inicijalizacije] --
-\n";
7:     initCode.append(ReplaceAllPlaceholders(SWCFragments["[ime_početnog_fra
gmenta_u_dijelu_inicijalizacije]"]));
8:
9:     funcCode = "//---
 [ime_početnog_fragmenta_u_dijelu_funkcijskih_definicija] ---\n";
10:     funcCode.append(ReplaceAllPlaceholders(SWCFragments["[ime_početnog_fr
agmenta_u_dijelu_funkcijskih_definicija]"]));
11: }
12: inače
13: {
14:     inclCode = "//---
 [ime_početnog_fragmenta_u_dijelu_uključivanja_zaglavlja] ---\n";
15:     inclCode.append(ReplaceAllPlaceholders(SWCFragments["[ime_početnog_fr
agmenta_u_dijelu_uključivanja_zaglavlja]"]));
16:
17:     initCode = "//--- [ime_početnog_fragmenta_u_dijelu_inicijalizacije] -
--\n";
18:     initCode.append(ReplaceAllPlaceholders(SWCFragments["[ime_početnog_fr
agmenta_u_dijelu_inicijalizacije]"]));
19:
20:     mainCode = "//--- [ime_početnog_fragmenta_u_galvnom_izvršno_dijelu] -
--\n";
21:     mainCode.append(ReplaceAllPlaceholders(SWCFragments["[ime_početnog_fr
agmenta_u_galvnom_izvršno_dijelu]"]));
22:
23:     for (auto a : currentData.swcRunnables)
24:     {
25:         SectionAppend(mainCodeSection[a.lock()-
>GetID()],"[ime_početnog_fragmenta_u_galvnom_izvršno_dijelu]",ReplaceAllPlace
holders(SWCFragments["[ime_početnog_fragmenta_u_galvnom_izvršno_dijelu]"]));
26:     }
27:
28:     funcCode = "//---
 [ime_početnog_fragmenta_u_dijelu_funkcijskih_definicija] ---\n";
29:     funcCode.append(ReplaceAllPlaceholders(SWCFragments["[ime_početnog_fr
agmenta_u_dijelu_funkcijskih_definicija]"]));
30: }
```

Slika 3.15: *Pseudo kod nadodavanja početnih fragmenata u spremnike dijelova implementacijskih predložaka ovisno o programskoj komponenti*

Iz slike 3.15 može se vidjeti da se početni fragmenti dodaju u sve dijelove implementacijskih predložaka osim onih koji završavaju određenim imenom. To je jedan od posebnih slučajeva jer programske komponente koje završavaju takvim imenom u svom glavnom izvršnom dijelu trebaju sadržavati fragment kojemu je dio semantičke oznake sav kod glavnog izvršnog dijela. Iz tog razloga se ne dodaje direktno u dio glavnog izvršnog dijela već se dodaje u obliku zamjene semantičke oznake varijable. Također postoje posebni slučajevi fragmenata koji se nadodaju u spremnike dijelova implementacijskih predložaka koji završavaju imenom programske komponente koja se trenutno generira (Slika 3.16 i Slika 3.17).

```
1: Za svaki (fragment : mapa_svih_fragmenata)
2: {
3:     Ako je (fragment.first.substr(0, početno_ime.length()) == "[početno_ime_dijela_za_u
ključivanje_zaglavlja]")
4:     {
5:         Ako je (fragment.first.find("[početni_fragment_za_dio_uključivanja_zaglavlja]")
 != string::npos)
6:         {
7:             nastavi;
8:         }
9:         Ako je ((fragment.first.find("[početno_ime_dijela_za_uključivanje_zaglavlja_]"
+ ime_programske_komponente != string::npos)
10:         {
11:             SectionAppend(inclCode, fragment.first, ReplaceAllPlaceholders(fragment.se
cond));
12:         }
13:         Ako je (fragment.first.find("[početno_ime_dijela_za_uključivanje_zaglavlja_]"
+ ime_programske_komponente + ".ITF") != string::npos)
14:         {
15:             SectionAppend(inclCode, fragment.first, ReplaceAllPlaceholders(fragment.se
cond));
16:         }
17:     }
18:     Inače ako je (fragment.first.substr(0, početno_ime.length()) == "[početno_ime_dije
la_za_inicijalizaciju]")
19:     {
20:         Ako je (fragment.first.find("[početni_fragment_za_dio_inicijalizacije]") != st
ring::npos)
21:         {
22:             nastavi;
23:         }
24:         Ako je ((fragment.first.find("[početno_ime_dijela_za_inicijalizaciju_]" + ime_
programske_komponente != string::npos)
25:         {
26:             SectionAppend(initCode, fragment.first, ReplaceAllPlaceholders(fragment.se
cond));
27:         }
28:         Ako je (fragment.first.find("[početno_ime_dijela_za_inicijalizaciju_]" + ime_p
rogramske_komponente + ".ITF") != string::npos)
29:         {
30:             SectionAppend(initCode, fragment.first, ReplaceAllPlaceholders(fragment.se
cond));
31:         }
32:     }
```

Slika 3.16: *Pseudo kod nadodavanja fragmenata koji sadržavaju ime programske komponente (1/2)*

```
33:     Inače ako je (fragment.first.substr(0, početno_ime.length()== "[počet
no_ime_glavnog_izvršnog_dijela]")
34:     {
35:         Ako je (fragment.first.find("[početni_fragment_za_glavni_izvršni_
dio]") != string::npos)
36:         {
37:             nastavi;
38:         }
39:         Ako je  ((fragment.first.find("[početno_ime_glavnog_izvršnog_dije
la_]" + ime_programske_komponente != string::npos)
40:         {
41:             SectionAppend(mainCode, fragment.first, ReplaceAllPlaceholder
s(fragment.second));
42:         }
43:         Ako je  (fragment.first.find("[početno_ime_glavnog_izvršnog_dijel
a_]" + ime_programske_komponente + ".ITF") != string::npos)
44:         {
45:             SectionAppend(mainCode, fragment.first, ReplaceAllPlaceholder
s(fragment.second));
46:         }
47:     }
48:     Inače ako je  (0, početno_ime.length()== "[početno_ime_dijela_funkcij
skih_definicija]")
49:     {
50:         Ako je  (fragment.first.find("[početni_fragment_za_dio_funkcijski
h_definicija]") != string::npos)
51:         {
52:             nastavi;
53:         }
54:         Ako je  ((fragment.first.find("[početno_ime_dijela_funkcijskih_de
finicija_]" + ime_programske_komponente != string::npos)
55:         {
56:             SectionAppend(funcCode, fragment.first, ReplaceAllPlaceholder
s(fragment.second));
57:         }
58:         Ako je  (fragment.first.find("[početno_ime_dijela_funkcijskih_def
inicija_]" + ime_programske_komponente + ".ITF") != string::npos)
59:         {
60:             SectionAppend(funcCode, fragment.first, ReplaceAllPlaceholder
s(fragment.second));
61:         }
62:     }
63: }
```

Slika 3.17: *Pseudo kod nadodavanja fragmenata koji sadržavaju ime programske komponente (2/2)*

Smještanje ostalih fragmenata i sučelja obavlja se pomoću podataka parsirane konfiguracijske tablice generatora i samih imena sučelja. Svako sučelje pod modulom koji se izvodi ima specifičan naziv kojim se može prepoznati kakav je tip sučelja, a to su:

- Čitač
- Pisač
- Pošiljatelj
- Primatelj

Iz naziva sučelja i podudarajućeg podatka iz konfiguracijske tablice generatora može se uvjetovati koji će se fragment staviti u određeni izvodljivi dio implementacijskog predloška (Slika 3.18 i Slika 3.19).

```
1: Za svako (sučelje : currentData.allInterfaceNames)
2: {
3:     Ako je (sučelje.substr(0, 4) == "READ")
4:     {
5:         SectionAppend(inclCode, ControlTableITF[currentRunnableIdx][INCL],
 ReplaceAllPlaceholders(SWCFragments[ControlTableITF[currentRunnableIdx][INCL
]]));
6:         SectionAppend(initCode, ControlTableITF[currentRunnableIdx][INIT],
 ReplaceAllPlaceholders(SWCFragments[ControlTableITF[currentRunnableIdx][INIT
]]));
7:         SectionAppend(mainCode, ControlTableITF[currentRunnableIdx][MAIN],
 ReplaceAllPlaceholders(SWCFragments[ControlTableITF[currentRunnableIdx][MAIN
]]));
8:         SectionAppend(funcCode, ControlTableITF[currentRunnableIdx][FUNC],
 ReplaceAllPlaceholders(SWCFragments[ControlTableITF[currentRunnableIdx][FUNC
]]));
9:     }
10:
11:     Inače Ako je (sučelje.substr(0, 5) == "WRITE")
12:     {
13:         SectionAppend(inclCode, ControlTableITF[currentRunnableIdx][INCL]
, ReplaceAllPlaceholders(SWCFragments[ControlTableITF[currentRunnableIdx][INC
L]]));
14:         SectionAppend(initCode, ControlTableITF[currentRunnableIdx][INIT]
, ReplaceAllPlaceholders(SWCFragments[ControlTableITF[currentRunnableIdx][INI
T]]));
15:         SectionAppend(mainCode, ControlTableITF[currentRunnableIdx][MAIN]
, ReplaceAllPlaceholders(SWCFragments[ControlTableITF[currentRunnableIdx][MAI
N]]));
16:         SectionAppend(funcCode, ControlTableITF[currentRunnableIdx][FUNC]
, ReplaceAllPlaceholders(SWCFragments[ControlTableITF[currentRunnableIdx][FUN
C]]));
17:     }
```

Slika 3.18: *Pseudo kod nadodavanja fragmenata ovisno o tipu sučelja (1/2)*

```
18:
19:     Inače ako je (sučelje.substr(0, 4) == "SEND")
20:     {
21:         SectionAppend(inclCode, ControlTableITF[currentRunnableIdx]
[INCL], ReplaceAllPlaceholders(SWCFragments[ControlTableITF[currentRunn
ableIdx][INCL]]));
22:         SectionAppend(initCode, ControlTableITF[currentRunnableIdx]
[INIT], ReplaceAllPlaceholders(SWCFragments[ControlTableITF[currentRunn
ableIdx][INIT]]));
23:         SectionAppend(mainCode, ControlTableITF[currentRunnableIdx]
[MAIN], ReplaceAllPlaceholders(SWCFragments[ControlTableITF[currentRunn
ableIdx][MAIN]]));
24:         SectionAppend(funcCode, ControlTableITF[currentRunnableIdx]
[FUNC], ReplaceAllPlaceholders(SWCFragments[ControlTableITF[currentRunn
ableIdx][FUNC]]));
25:     }
26:
27:     Inače ako je (sučelje.substr(0, 3) == "REC")
28:     {
29:         SectionAppend(inclCode, ControlTableITF[currentRunnableIdx]
[INCL], ReplaceAllPlaceholders(SWCFragments[ControlTableITF[currentRunn
ableIdx][INCL]]));
30:         SectionAppend(initCode, ControlTableITF[currentRunnableIdx]
[INIT], ReplaceAllPlaceholders(SWCFragments[ControlTableITF[currentRunn
ableIdx][INIT]]));
31:         SectionAppend(mainCode, ControlTableITF[currentRunnableIdx]
[MAIN], ReplaceAllPlaceholders(SWCFragments[ControlTableITF[currentRunn
ableIdx][MAIN]]));
32:         SectionAppend(funcCode, ControlTableITF[currentRunnableIdx]
[FUNC], ReplaceAllPlaceholders(SWCFragments[ControlTableITF[currentRunn
ableIdx][FUNC]]));
33:     }
34: }
```

Slika 3.19: *Pseudo kod nadodavanja fragmenata ovisno o tipu sučelja (2/2)*

Postoji još nekolicina uvjeta smještanja posebnih slučaja fragmenata no specifični su i preuzeti iz postojećeg TEG-a te se smatraju hakovima koji se ne bi trebali pokazivati javno jer su pod ugovorom o tajnosti (engl. *Non-disclosure agreement - NDA*).

3.4.1. Smještanje fragmenata u spremnik implementacijskog predloška i ispis u datoteku

Procesiranjem fragmenata svi izmijenjeni fragmenti potrebni za trenutnu programsku komponentu nalaze se u trenutnim spremnicima znakovnih polja za svaki dio unutar implementacijske tablice. Jedina stvar koja preostaje je smjestiti kod unutar spremnika u predodređene dijelove implementacijskih predložaka. Popunjavanje spremnika implementacijskog predloška implementirano je relativno jednostavno i to u obliku petlje koja prolazi po redcima spremnika implementacijskog predloška (Slika 3.20). Svaki redak spremnika ispituje se da li sadrži oznaku koja označava ulazak u određeni dio implementacijske tablice. Dijelovi implementacijske tablice, kako je spomenuto ranije, mogu sadržavati:

- Dio za uključivanje zaglavlja
- Glavni izvršni dio
- Dio funkcijskih definicija

Primjetno je da nedostaje inicijalizacijski dio implementacijskog predloška. Razlog tomu je jer se on nalazi kao pod regija glavnog izvršnog dijela. Inicijalizacijski dijelovi konkretno, označeni su kao dio modula koji se izvodi programske komponente te se njima posebno rukuje. Dijelovi unutar glavnog izvodljivog dijela se popunjavaju na temelju dohvaćenog simbola koji je prisutan samo u glavnom izvršnom dijelu implementacijskog predloška. Time se pomoću simbola stavljaju pripadajući dijelovi koda glavnog izvršnog dijela. Osim posebnih slučajeva programskih komponenti gdje u glavnom izvršnom dijelu postoji samo jedan izvršni modul, te je ujedno potreban samo jedan spremnik cjelokupnog izvršnog koda, u ostalim programskim komponentama potrebno je stavljati različiti kod u različite module koji se izvode, glavnog izvršnog dijela implementacijskog predloška. Kako bi se to ostvarilo implementiran je dodatni spremnik koji sadržava povezane informacije naziva modula koji se izvode i fragmenata koji pripadaju modulu koji se izvodi te se takvim pristupom uspješno popunjavaju dijelovi unutar glavnog izvršnog dijela implementacijskog predloška. Ostale dijelove implementacijskog predloška, pošto nemaju dodatnih dijelova unutar sebe, jednostavno se popunjavaju pripadajućim spremnicima koda.

```
1: Za svaki (redak : templateCode)
2: {
3:     Ako je (redak.find(startLabel) != string::npos && startLabel == mai
nLabel)
4:     {
5:         Ako je (sljedeći_redak.find(symbolDelimiter) != string::npos)
6:         {
7:             symbol = sljedeći_redak.substr(symbolDelimiter.length(), sl
jedeći_redak.length());
8:
9:             Ako je (symbol.substr(symbol.length() -
 4, symbol.length()) == "[početak_oznake_za_inicijalizaciju]")
10:             {
11:                 templateCode.insert(templateCode.begin() + redak.pozic
ija() + 3, initCode);
12:             }
13:             Inače ako je (mainCodeSection.find(symbol) != mainCodeSect
ion.end())
14:             {
15:                 templateCode.insert(templateCode.begin() + redak.pozic
ija() + 3, mainCodeSection[symbol]);
16:             }
17:         }
18:     }
19:     Inače ako je (redak.find(startLabel) != string::npos && startLabel
 == includeLabel)
20:     {
21:         templateCode.insert(templateCode.begin() + redak.pozicija() +
2, inclCode);
22:     }
23:     Inače ako je (redak.find(startLabel) != string::npos && startLabel
 == functionLabel)
24:     {
25:         templateCode.insert(templateCode.begin() + redak.pozicija() +
2, funcCode);
26:     }
27:     Inače
28:     {
29:         nastavi;
30:     }
31: }
```

Slika 3.20: *Pseudo kod smještanja fragmenata u spremnik implementacijskog predloška*

Popunjavanjem implementacijskih predložaka esencijalno je generirana programska komponenta kojoj sve što preostaje za napraviti je ispis u datoteku. Pošto je datoteka implementacijskog

predloška učitana u spremnik memorije te su se sva smještanja obavljala nad tim spremnikom, jednostavno se može obaviti ispis redak po redak u datoteku.

```
1: void IspišiGeneriraniKodUDatoteku(TEG_filehandler)
2: {
3:     Za svaki (redak : templateCode)
4:     {
5:         TEG_fileHandler << redak << endl;
6:     }
7:     fileHandler.close();
8:     fileHandler.clear();
9:     templateCode.clear();
10:    inclCode.clear();
11:    initCode.clear();
12:    mainCode.clear();
13:    mainCodeSection.clear();
14:    funcCode.clear();
15: }
```

Slika 3.21: *Pseudo kod ispisa programske komponente u datoteku*

Slikom 3.21 prikazan je ispis datoteke koja će sačinjavati generiranu programsku komponentu te čišćenje spremnika klase generatora u svrhu pripreme za generiranje sljedeće programske komponente u iteraciji. Generiranjem datoteka programskih komponenti završava se dio generatora izvornog C koda TEG-a.

4. REZULTATI RADA TEG GENERATORA IZVORNOG C KODA

U ovome poglavlju objašnjen je način na koji se ispitivao rad TEG generatora izvornog C koda. Najprije se ispituje ispravnost pojedinog modula TEG generatora, gdje će se naposlijetku generirati testna programska komponenta koja obuhvaća sve testove i potvrđuje cjelokupan rad generatora izvornog C koda. Svaki test sastoji se od implementacije koda za ispitivanje samog koda, te priloženog rezultata ispitivanja. TEG generator ispitivan je na tri različita sustava: prijenosnom računalu sa procesorom *„Intel Core i5-8300H“*, radnog takta 2.3GHz s turbom na 4.0GHz kroz 4 jezgre i 8 niti, 16GB DDR4-SDRAM i pogonu čvrstog stanja (*Solid-state drive - SSD*) sa brzinom pisanja i čitanja od 500*mb/s* , stolnom računalu sa procesorom *„Intel Core i7-7700“*, radnog takta 3.6GHz kroz 4 jezgre i 8 niti , 8GB DDR4-RAM i tvrdom disku sa brzinom okretaja od 7200 revolucija po minuti i prosječnom brzinom čitanja i pisanja od 70*mb/s* te stolnom računalu sa procesorom *„Intel Core i5-10400“,* radnog takta 2.9GHz sa turbom na 4.0GHz kroz 6 jezgri i 12 niti, 16GB DDR4-RAM i tvrdom disku sa brzinom okretaja od 7200 revolucija po minuti i prosječnom brzinom čitanja i pisanja od 70*mb/s*. Testovi su također provedeni na različitim diskovima tj. SSD i tvrdim diskovima no performanse su identične jer je usko grlo uglavnom procesor tokom izvršavanja programa te nema tolike razlike u čitanju i pisanju datoteka.

4.1. Rezultati rada parsera fragmenata programskih komponenti

Rad parsera relativno je jednostavno ispitati. Potrebno je kao i u glavnom TEG programu dohvatiti pomoću TEG rukovatelja datoteka datoteku fragmenata unutar navedenog direktorija te pozivati funkciju za parsiranje fragmenata nad otvorenom datotekom. U ovome testiranju, za prikaz rada parsera napravljena je testna datoteka fragmenata *„fragmentTest.txt“* koja ne sadrži konkretne fragmente projekta već ima identičnu strukturu, ali bez pravilnog izvršnog koda (Slika 4.1). U testnom programu (Slika 4.2) priložena je putanja gdje se nalaze datoteke fragmenata te petlja u kojoj se po nađenim datotekama izvršava parsiranje fragmenata. Za potrebu testiranja rada parsera fragmenata koristi se statično napisana putanja do testne datoteke fragmenata te se unutar direktorija nalazi i parsira samo jedna testna datoteka fragmenata, no u konkretnoj TEG implementaciji se dinamički prosljeđuje putanja do datoteka fragmenata te se dinamički parsiraju sve datoteke nađene unutar direktorija. Za provjeru rezultata, svi fragmenti spremljeni u memoriji se ispisuju u datoteku kako bi se utvrdila ispravnost parsiranja.

```
// ####### TEMPLATE dummy_replacement

#include "Rte_Type.h"

#if defined(_HOST_PH01_)
    #define _HOST_PH00_
#endif

// ####### END

// ####### TEMPLATE funcDummy

#if defined(_HOST_$###ime-hosta###$_) && defined(TEST_$###akronim-testnog-
modula###$)
void TestUnpack_$###ime-swca###$( const uint8 *Payload, uint32 FrameId )
{}
#endif

// ####### END
```

Slika 4.1: *Sadržaj datoteke fragmentTest.txt*

```
1: string fragmentPath_ = "CodeGenerator\\dummyFragments\\";
2: string outputPath = "CodeGenerator\\dummyOutputs\\";
3: vector<string> fragments_path_vector;
4: fragments_path_vector.push_back(fragmentPath);
5:
6: cCodeFenerator& CodeGenerator = cCodeGenerator::GetInstance();
7:
8: FileHandlerInitialize(".");
9: FileHandler &fh = FileHandler::GetInstance();
10:
11: vector<string> fragment_names = fh.FindFiles(fragments_path_vector,
 "*", false);
12:
13: for(const auto& file : fragment_names)
14: {
15:     ifstream temp = fh.OpenFile(file);
16:     CodeGenerator.LoadSWCFragmentsFile(temp);
17:     CodeGenerator.ExtractFragments();
18: }
19:
20: CodeGenerator.OutputFragments(outputPath + "test_fragments_output.c
");
```

Slika 4.2: *Testni program parsera fragmenata programskih komponenti*

Rezultat parsera fragmenata je izlazna datoteka *test_fragments_output.c* u kojoj se nalazi sadržaj memorije, čiji je sadržaj identičan ulaznoj datoteci (Slika 4.4). Prema tome parser radi ispravno. Slika 4.3 i Slika 4.4 potvrđuju ispravnost modula.

```
12-08-2021 17:01:21.927 [  INFO  ] New search directory added: .
12-08-2021 17:01:21.929 [  INFO  ] File Handler: Initialized successfully!
12-08-
2021 17:01:22.083 [  INFO  ] Finding files with pattern: * (Case Sensitive)
12-08-
2021 17:01:22.085 [  INFO  ] File found: CodeGenerator\dummyFragments\fragment
Test.txt
12-08-
2021 17:01:22.343 [  INFO  ] Opened file in write mode: CodeGenerator\dummyOut
puts\test_fragments_output.c
```

Slika 4.3: *Ispis konzole testnog programa za parser fragmenata*

```
//--- dummy_replacement ---

#include "Rte_Type.h"

#if defined(_HOST_PH01_)
    #define _HOST_PH00_
#endif

//--- funcDummy ---

#if defined(_HOST_$###ime-hosta###$_) && defined(TEST_$###akronim-testnog-
modula###$)
void TestUnpack_$###ime-swca###$( const uint8 *Payload, uint32 FrameId )
{}
#endif
```

Slika 4.4: *Sadržaj datoteke test_fragments_output.c*

4.1.1. Rezultati rada parsera konfiguracijske tablice generatora

Rad parsera konfiguracijske tablice generatora provodi se slično parseru fragmenata no bez ispisa tablice u datoteku, već samo u konzolu izvršenog programa. Testiranje se provodi kao i u glavnom TEG programu gdje se pomoću TEG rukovatelja datoteka nad datotekom konfiguracijske tablice generatora unutar navedenog direktorija poziva funkcija za parsiranje konfiguracijske tablice generatora (Slika 4.6). U ovome testiranju, za prikaz rada parsera korištena je konkretna konfiguracijska tablica generatora no bit će prikazano samo nekoliko primjera podataka kako bi se mogla dati usporedba sa parsiranim vrijednostima. Pošto je

parsiranje datoteke konfiguracijske tablice generatora uvijek isto parsiranje jedne datoteke i to samo jednom na početku programa, otvaranje datoteke se odvija u samoj apstrakciji parsiranja. Za provjeru rezultata, svi podaci spremljeni u memoriji se ispisuju u konzolu programa kako bi se izvšila vizualna usporedba i time utvrdila ispravnost parsiranja (Slika 4.5).

```
<?xml version='1.0' encoding='UTF-8'?>
<PACKAGES>
    <PACKAGE>
        <SHORT-NAME>ControlTables</SHORT-NAME>
        <OPTIONS>
            <OPTION dest="tblControlITF">
                <SHORT-NAME>ControlITF</SHORT-NAME>
                <VALUE idx="0">
                    <TYPE></TYPE>
                    <INCL></INCL>
                    <INIT></INIT>
                    <FUNC></FUNC>
                    <MAIN>mainStart</MAIN>
                    <FILTERS></FILTERS>
                </VALUE>

                            .
                            .
                            .

                <VALUE idx="19">
                    <TYPE>iwriterefDS</TYPE>
                    <INCL>inclRteIWriteRef</INCL>
                    <INIT>initRteInterface</INIT>
                    <FUNC>funcRteIWriteRefDS.ITF</FUNC>
                    <MAIN>mainRteIWriteRef</MAIN>
                    <FILTERS></FILTERS>
                </VALUE>
```

Slika 4.5: *Sadržaj datoteke konfiguracijske tablice generatora*

```
1: cCodeFenerator& CodeGenerator = cCodeGenerator::GetInstance();
2: ParseGeneratorConfigFile(CodeGenerator);
3: CodeGenerator.printTableControlITF();
```

Slika 4.6: *Testni program parsera konfiguracijske tablice generatora*

Rezultat parsera konfiguracijske tablice generatora je ispis sadržaja memorije, čiji je sadržaj identičan ulaznoj datoteci. Prema tome parser radi ispravno. Slika 4.7 potvrđuje ispravnost modula.

```
12-08-2021 17:11:21.927 [  INFO  ] New search directory added: .
12-08-
2021 17:11:21.929 [  INFO  ] File Handler: Initialized successfully!
12-08-
2021 17:11:22.083 [  INFO  ] Finding files with pattern: * (Case Sen
sitive)
12-08-
2021 17:11:22.085 [  INFO  ] File found: CodeGenerator\configure_gen
erator.xml
Printing TableControlITF...
idx: 0
FILTERS =
FUNC =
INCL =
INIT =
MAIN = mainStart
TYPE =

        .
        .
        .

idx: 19
FILTERS =
FUNC = funcRteIWriteRefDS.ITF
INCL = inclRteIWriteRef
INIT = initRteInterface
MAIN = mainRteIWriteRef
TYPE = iwriterefDS
```

Slika 4.7*: Ispis konzole testnog programa za parser konfiguracijske tablice generatora*

4.2. Rezultati rada zamjene semantičkih oznaka varijabli fragmenata

Rad ispravne zamjene semantičkih oznaka varijabli ispituje se nad testnim fragmentima korištenim u prethodnim testovima. Pošto takvi podaci unutar semantičkih oznaka ne postoje unutar baze podataka, bit će demonstrirana zamjena varijabli sa proizvoljnim vrijednostima no za trenutno odgovarajuću testnu programsku komponentu. Kako bi se testirao rad zamjene oznaka potrebno je uklopiti i test parsiranja fragmenata gdje su fragmenti isparsirani i spremljeni u memoriju. Slično testu parsiranja fragmenata, test će se izvesti nad naknadno napravljenom datotekom testnih fragmenata i proizvoljnih podataka za zamjenu semantičkih oznaka varijabli. Test će biti dodatak glavnom programu TEG-a gdje je razlika samo dodatni testni fragment dodan u spremnik memorije te se samo on izmjenjuje i printa u konzolu (Slika 4.8). Sadržaj datoteke fragmenata je isti kao na Slika 4.1.

```
1: string fragmentPath_ = "CodeGenerator\\dummyFragments\\";
2: string outputPath = "CodeGenerator\\dummyOutputs\\";
3: vector<string> fragments_path_vector;
4: fragments_path_vector.push_back(fragmentPath);
5: cCodeFenerator& CodeGenerator = cCodeGenerator::GetInstance();
6: FileHandlerInitialize(".");
7: FileHandler &fh = FileHandler::GetInstance();
8: vector<string> fragment_names = fh.FindFiles(fragments_path_vector, "*"
, false);
9: for(const auto& file : fragment_names)
10: {
11:     ifstream temp = fh.OpenFile(file);
12:     CodeGenerator.LoadSWCFragmentsFile(temp);
13:     CodeGenerator.ExtractFragments();
14: }
15: for (const auto& file : fs::directory_iterator(path))
16: {
17:     string SWC_name;
18:     string file_extension = file.path().extension().string();
19:     string file_name = file.path().filename().string().substr(0, file.
path().filename().string().length() - file_extension.length());
20:     CurrentDataSuperObject currentData{ swcPairs["SWCID_" + file_name]
, SWC_name, file_name, InternalBehaviors, fragRunnables, runnableInterface
s, runnableTimings };
21:     CodeGenerator.ProcessFragments(names, currentData);
22: }
```

Slika 4.8: *Testni program zamjena semantičkih oznaka varijabli fragmenata*

```
cnt += AlterPlaceholdersByFragment(fragmentValue, "$###ime-
swca###$", currentSWCData.swcName);
cnt += AlterPlaceholdersByFragment(fragmentValue, "$###ime-
hosta###$", "PH00");
cnt += AlterPlaceholdersByFragment(fragmentValue, "$###akronim-testnog-
modula###$", "ITF");
```

Slika 4.9: *Dodatak funkciji za mijenjanje svih semantičkih oznaka varijabli fragmenata u svrhu testiranja*

Slikom 4.9 može se vidjeti dodatak koji je potrebno dodati u funkciju za mijenjanje oznaka kako bi se dobio željeni rezultat testa. Također, u funkciji procesiranja fragmenata, dodan je ispis fragmenta nakon zamjene u konzolu kako bi se vidio rezultat testa (Slika 4.10). Za testne potrebe, ispisuje se samo jedan testni fragment jer on u sebi sadrži semantičke oznake varijabli.

```
cout << FormatSectionString("funcDummy", ReplaceAllPlaceholders(SWCF
ragments["funcDummy"])) << endl;
```

Slika 4.10: *Dodatak funkciji za procesiranje fragmenata*

```
12-08-
2021 17:54:21.527 [  INFO  ] New search directory added: .
12-08-
2021 17:54:21.529 [  INFO  ] File Handler: Initialized successf
ully!
12-08-
2021 17:54:21.586 [  INFO  ] Finding files with pattern: * (Cas
e Sensitive)
12-08-
2021 17:54:21.589 [  INFO  ] File found: CodeGenerator\dummyFra
gments\fragmentTest.txt
//--- funcDummy ---

#if defined(_HOST_PH00_) && defined(TEST_ITF)
void TestUnpack_fragmentTest( const uint8 *Payload, uint32 Fram
eId )
{}
#endif
```

Slika 4.11: *Ispis konzole testnog programa za mijenjanje semantičkih oznaka varijabli fragmenata*

Iz priloženog ispisa sa slike 4.11 se da vidjeti da su se originalnom fragmentu semantičke oznake varijabli promijenile u proizvoljne vrijednosti definirane u funkciji za zamjenu semantičkih oznaka varijabli.

4.3. Rezultati rada smještanja fragmenata u implementacijski predložak generiranjem testne programske komponente

Smještanje fragmenata u odgovarajuća mjesta implementacijskog predloška će se prikazati u obliku generirane testne datoteke programske komponente, koja može konceptualno pokazati kako su generirane konkretne programske komponente TEG projekta. Kako bi se testirao rad generiranja programskih komponenti bit će potrebno koristiti prethodne testove, proizvoljne podatke koji će imitirati podatke iz baze podataka te pomoćne testne datoteke iz prethodnih testova uz dodatak testnih implementacijskih predložaka (Slika 4.12 i Slika 4.13). Testiranje se provodi glavnim dijelom TEG programa koji će uz programske komponente stvarnog projekta dodatno generirati testnu programsku komponentu. U ovome testiranju, za prikaz rada generiranja programskih komponenti, potrebno je nadodati podatke testnog implementacijskog predloška u funkciju za procesiranje fragmenata. Dodavanje podataka u svrhu testa je potrebno jer se u TEG projektu programske komponente generiraju pomoću podataka koji se nalaze u bazi podataka. Rezultat generiranja programske komponente je popunjeni testni implementacijski predložak u kojem će se nalaziti fragmenti smješteni na temelju proizvoljnih testnih podataka. Popunjen testni implementacijski predložak će predstavljati oblik programskih komponenti TEG projekta te time potvrditi rad generatora izvornog C koda.

```
/**********************************************************************************************************
****************
 * DO NOT CHANGE THIS COMMENT!           << Start of include and declaration area >>        DO NOT CHA
NGE THIS COMMENT!
 **********************************************************************************************************
***************/

/**********************************************************************************************************
****************
 * DO NOT CHANGE THIS COMMENT!           << End of include and declaration area >>          DO NOT CHA
NGE THIS COMMENT!
 **********************************************************************************************************
***************/

FUNC(void, dummyTemplate_CODE) RdummyTemplate_Init(void)
{
/**********************************************************************************************************
****************
 * DO NOT CHANGE THIS COMMENT!           << Start of runnable implementation >>            DO NOT CHA
NGE THIS COMMENT!
 * Symbol: RdummyTemplate_Init
 **********************************************************************************************************
***************/

/**********************************************************************************************************
****************
 * DO NOT CHANGE THIS COMMENT!           << End of runnable implementation >>              DO NOT CHA
NGE THIS COMMENT!
 **********************************************************************************************************
***************/
```

Slika 4.12: *Izgled praznog testnog implementacijskog predloška (1/2)*

```
FUNC(void, dummyTemplate_CODE) RdummyTemplate_0(void)
{
/*********************************************************************
*********************************************
 * DO NOT CHANGE THIS COMMENT!           << Start of runnable implem
entation >>             DO NOT CHANGE THIS COMMENT!
 * Symbol: RdummyTemplate_0
 *********************************************************************
*********************************************/

/*********************************************************************
*********************************************
 * DO NOT CHANGE THIS COMMENT!           << End of runnable implemen
tation >>               DO NOT CHANGE THIS COMMENT!
 *********************************************************************
*********************************************/
}
/*********************************************************************
*********************************************
 * DO NOT CHANGE THIS COMMENT!           << Start of function defini
tion area >>            DO NOT CHANGE THIS COMMENT!
 *********************************************************************
*********************************************/

/*********************************************************************
*********************************************
 * DO NOT CHANGE THIS COMMENT!           << End of function definiti
on area >>              DO NOT CHANGE THIS COMMENT!
 *********************************************************************
*********************************************/
```

Slika 4.13: *Izgled praznog testnog implementacijskog predloška (2/2)*

Iz praznog implementacijskog predloška prikazanih slikom 4.12 i slikom 4.13 se mogu vidjeti dijelovi sa oznakama koje služe za usmjeravanje i stavljanje pripadajućih dijelova koda unutar dijelova datoteke. Simboli i ostali dijelovi testnog implementacijskog predloška preimenovani su kako bi se mogao uspješno generirati bez mijenjanja rješenja gotovog generatora već uz par testnih dodataka. Stoga je prilikom procesiranja fragmenata dodan podatak iščitan iz testnog implementacijskog predloška te se kreira njegov odgovarajući uređeni par (Slika 4.14).

```
currentData.swcRunnables.push_back(DataComponent("RdummyTemplate_Init"));
currentData.swcRunnables.push_back(DataComponent("RdummyTemplate_0"));
currentData.runnableInterfaces["RdummyTemplate_Init"] = "READ_test";
currentData.runnableInterfaces["RdummyTemplate_0"] = "READ_test";
```

Slika 4.14: *Dodatak testnih podataka funkciji za procesiranje fragmenata*

```
/*********************************************************************************************************
****************
 * DO NOT CHANGE THIS COMMENT!           << Start of include and declaration area >>        DO NOT CHA
NGE THIS COMMENT!
 *********************************************************************************************************
***************/
//inclStart
//blok_koda (prevelik je za primjer)

//--- inclRteRead ---
// 1.PH00-0.dummyTemplate.0.RdummyTemplate_0-0.read-0.Rte_Read_RdummyTemplate_0

#define dAsilLevel_read0  eAsilLevel_0
static void fInitItfBuffer_read0( void );

//--- inclMemMap_swcStart ---
#define dummyTemplate_START_SEC_VAR_INIT_UNSPECIFIED
#include "dummyTemplate_MemMap.h"

// Interface specific data
static Dt_INT    var_read0;
static Dt_INT    bak_read0;
//--- inclMemMap_swcStop ---
#define dummyTemplate_STOP_SEC_VAR_INIT_UNSPECIFIED
#include "dummyTemplate_MemMap.h"

//--- inclMemMap_globStart ---
#define GlobalShared_START_SEC_VAR_INIT_UNSPECIFIED
#if defined(_HOST_SH00_)
    #include "GlobalShared_MemMap.h"
#else
    #include "MemMap.h"
#endif

static tItfStatusData       vItfStatusData_read0 = mInitItfStatusData(&bak_read0, sizeof(Dt_INT), 0, d
InvalidItfIndex);
//--- inclMemMap_globStop ---
#define GlobalShared_STOP_SEC_VAR_INIT_UNSPECIFIED
#if defined(_HOST_SH00_)
    #include "GlobalShared_MemMap.h"
#else
    #include "MemMap.h"
#endif
/*********************************************************************************************************
****************
 * DO NOT CHANGE THIS COMMENT!           << End of include and declaration area >>          DO NOT CHA
NGE THIS COMMENT!
 *********************************************************************************************************
***************/
```

Slika 4.15: *Rezultat dijela za uključivanje zaglavlja unutar generirane testne programske komponente*

Iz slike 4.15 može se vidjeti da se sadržaj varijable koja čuva sav kod za uključivanje zaglavlja trenutne programske komponente pravilno smješta u predodređeno mjestu unutar testnog implementacijskog predloška.

```
FUNC(void, dummyTemplate_CODE) RdummyTemplate_Init(void)
{
/*********************************************************
*********************************************************
***
 * DO NOT CHANGE THIS COMMENT!           << Start of runna
ble implementation >>             DO NOT CHANGE THIS COMME
NT!
 * Symbol: RdummyTemplate_Init
 *********************************************************
*********************************************************
**/
//initStart
//blok_koda (prevelik je za primjer)

//--- initRteInterface ---
// 1.PH00-0.dummyTemplate.0.RdummyTemplate_0-0.read-
0.Rte_Read_RdummyTemplate_0

    fInitItfBuffer_read0();
/*********************************************************
*********************************************************
***
 * DO NOT CHANGE THIS COMMENT!           << End of runnabl
e implementation >>               DO NOT CHANGE THIS COMME
NT!
 *********************************************************
*********************************************************
**/
}
```

Slika 4.16: *Rezultat dijela za inicijalizaciju unutar generirane testne programske komponente*

Prikazanim rezultatom na slici 4.16 može se vidjeti kako je sav sadržaj varijable koji čuva kod za inicijalizaciju trenutne programske komponente pravilno smješten između predodređenih oznaka unutar implementacijskog predloška.

```
/**********************************************************
**********************************************************
*******
 * DO NOT CHANGE THIS COMMENT!           << Start of fun
ction definition area >>             DO NOT CHANGE THIS C
OMMENT!
 **********************************************************
**********************************************************
******/
//funcStart
//blok_koda (prevelik je za primjer)

//--- funcRteRead ---
// 1.PH00-0.dummyTemplate.0.RdummyTemplate_0-0.read-
0.Rte_Read_RdummyTemplate_0

static void fInitItfBuffer_read0()
{
#if defined(test_init_Dt_INT_De_TestVar)
    Sl_MemCpy( &bak_read0, &init_Dt_INT_De_TestVar, size
of(Dt_INT) );
#else
    Sl_MemSet( &bak_read0, 0x00, sizeof(Dt_INT) );
#endif
}
/**********************************************************
**********************************************************
*******
 * DO NOT CHANGE THIS COMMENT!             << End of funct
ion definition area >>               DO NOT CHANGE THIS C
OMMENT!
 **********************************************************
**********************************************************
******/
```

Slika 4.17: *Rezultat dijela funkcijskih definicija unutar generirane testne programske komponente*

Iz slike 4.17 može se vidjeti da se sadržaj varijable koja čuva sav kod za funkcijske definicije trenutne programske komponente pravilno smješta u predodređeno mjestu unutar testnog implementacijskog predloška

```
FUNC(void, dummyTemplate_CODE) RdummyTemplate_0(void)
{
/*****************************************************************************************
***************************
 * DO NOT CHANGE THIS COMMENT!           << Start of runnable implementation >>
 DO NOT CHANGE THIS COMMENT!
 * Symbol: RdummyTemplate_0
 *****************************************************************************************
*************************/
//mainStart
//blok_koda (prevelik je za primjer)

//mainDebugModeCheck
// 1.PH00-0.dummyTemplate.0.RdummyTemplate_0-0.read-0.Rte_Read_RdummyTemplate_0

    {
        tMsgTestResponse       *vpTestRsp               = &vTransmitToTestPC.Data;

        fResetTestResponse( vpTestRsp, read + dOffsetMiscItfType, 0, vpTimeMeasure-
>StartEGT );
        switch (vItfStatusData_read0.ItfActivity)
        {
            case eItfActivity_PerformFunction:
            {
                fDebugModeCheck( ownSWCID );
                vpTestRsp->Common.Info = E_OK;
                vpTestRsp->Status.ErrorDetail = E_OK;

                vItfStatusData_read0.ItfActivity = eItfActivity_None;
                vpTestRsp->Common.Usage = eRspMisc_withoutData;
                mSendToTestPC( &vTransmitToTestPC );
                break;
            }
            default:
            {   // if unexpected, send notification to TestPC and clear command
                fCheckUnexpectedItfActivity( &vTransmitToTestPC, &vItfStatusData_read0.ItfA
ctivity );
                break;
            }
        }
    }
/*****************************************************************************************
***************************
 * DO NOT CHANGE THIS COMMENT!           << End of runnable implementation >>
 DO NOT CHANGE THIS COMMENT!
 *****************************************************************************************
*************************/
}
```

Slika 4.18: *Rezultat glavnog izvršnog dijela unutar generirane testne programske komponente*

Prikazanim rezultatom na slici 4.18 može se vidjeti kako je sav sadržaj varijable koji čuva kod za glavni izvršni dio trenutne programske komponente pravilno smješten između predodređenih oznaka unutar implementacijskog predloška.

Iz popunjenog implementacijskog predloška, odnosno generirane programske komponente prikazane slikom 4.15, slikom 4.16, slikom 4.17 i slikom 4.18 može se konceptualno vidjeti kako se generiraju i ostale programske komponente projekta. Kompletan popunjeni testni implementacijski predložak, odnosno programska komponenta nalazi se u prilogu P.4.1.

4.4. Rezultati vremena izvršavanja TEG projekta

Trenutno *Python* rješenje TEG-a izvršava se sporo za program koji esencijalno parsira nekolicinu datoteka i piše tekst u datoteke. Vrijeme izvršavanja *Python* TEG-a je u prosjeku oko pet minuta, gdje je prosjek četiri minute za verziju programa koji testira sučelja, što radi TEG u ovome radu. Vrijeme izvršavanja *Python* implementacije je mjereno *Python* bibliotekom *time,* a vrijeme izvršavanja C++ implementacije je mjereno C++ bibliotekom *chrono.* U *Python* implementaciji vrijeme je mjereno početkom generiranja određene verzije programa te se time lako može izdvojiti traženo vrijeme izvršavanja. Kod C++ implementacije, vrijeme se mjeri kroz cijeli raspon *main* funkcije, čime se obuhvaća izvršavanje svakog modula TEG-a. Testiranje se provodi na tri različite platforme za što bolju usporedbu relativnog ubrzanja C++ implementacije nad *Python* implementacijom. Razlog tomu je što TEG mora imati mogućnost izvršavanja na strani bilo kojeg klijenta, te se ovim testiranjem može dati prosječno vrijeme izvršavanja ovisno o platformi i eventualna procjena vremena izvršavanja na ostalim platformama.

Broj testa	Procesor	TEG implementacija	Vrijeme (h:m:s)
1.	Intel Core i5 10400 @4.00 GHz	*Python*	0:04:19,776
		C++	0:00:11,9771
	Intel Core i5 8300H @2.30 GHz	*Python*	0:06:56,218
		C++	0:00:27,2209
	Intel Core i7 7700 @3.6GHz	*Python*	0:04:51,396
		C++	0:00:15,2437
2.	Intel Core i5 10400 @4.00 GHz	*Python*	0:04:33,832
		C++	0:00:12,4894
	Intel Core i5 8300H @2.30 GHz	*Python*	0:06:54,898
		C++	0:00:28,3851
	Intel Core i7 7700 @3.6GHz	*Python*	0:05:07,024
		C++	00:00:15,8956
3.	Intel Core i5 10400	*Python*	0:04:38,587

3.	@4.00 GHz	C++	00:00:12,2597
	Intel Core i5 8300H @2.30 GHz	*Python*	0:07:02,102
		C++	0:00:27,8631
	Intel Core i7 7700 @3.6GHz	*Python*	0:05:12,355
		C++	00:00:15,6033
4.	Intel Core i5 10400 @4.00 GHz	*Python*	0:04:37,047
		C++	00:00:11,8783
	Intel Core i5 8300H @2.30 GHz	*Python*	0:06:59,769
		C++	0:00:26,9963
	Intel Core i7 7700 @3.6GHz	*Python*	0:05:10,629
		C++	00:00:15,1179
5.	Intel Core i5 10400 @4.00 GHz	*Python*	0:04:34,369
		C++	00:00:11,6097
	Intel Core i5 8300H @2.30 GHz	*Python*	0:06:55,711
		C++	0:00:26,3859
	Intel Core i7 7700 @3.6GHz	*Python*	0:05:07,726
		C++	00:00:14,7761
6.	Intel Core i5 10400 @4.00 GHz	*Python*	0:04:32,978
		C++	00:00:11,7051
	Intel Core i5 8300H @2.30 GHz	*Python*	0:06:53,604
		C++	0:00:26,6026
	Intel Core i7 7700 @3.6GHz	*Python*	0:05:06,067
		C++	00:00:14,8974
7.	Intel Core i5 10400 @4.00 GHz	*Python*	0:04:31,871
		C++	00:00:11,4181
	Intel Core i5 8300H @2.30 GHz	*Python*	0:06:51,926
		C++	0:00:25,9504
	Intel Core i7 7700 @3.6GHz	*Python*	0:05:04,825
		C++	00:00:14,5322
8.	Intel Core i5 10400 @4.00 GHz	*Python*	0:04:36,858
		C++	00:00:12,1714
	Intel Core i5 8300H	*Python*	0:06:59,482

8.	@2.30 GHz	C++	0:00:27,6623
	Intel Core i7 7700 @3.6GHz	*Python*	0:05:10,416
		C++	00:00:15,4908
9.	Intel Core i5 10400 @4.00 GHz	*Python*	0:04:33,663
		C++	00:00:11,7131
	Intel Core i5 8300H @2.30 GHz	*Python*	0:06:54,641
		C++	0:00:26,6207
	Intel Core i7 7700 @3.6GHz	*Python*	0:05:06,834
		C++	00:00:14,9075
10.	Intel Core i5 10400 @4.00 GHz	*Python*	0:04:35,538
		C++	00:00:12,106
	Intel Core i5 8300H @2.30 GHz	*Python*	0:06:57,482
		C++	0:00:27,5137
	Intel Core i7 7700 @3.6GHz	*Python*	0:05:08,936
		C++	00:00:15,4076

Tablica 4.1: *Rezultati mjerenja vremena izvršenja implementacije Python i C++ TEG-a*

Iz rezultata tablice 4.1 može se primijetiti kako je C++ implementacija znatno brža u izvođenju projekta nego *Python* implementacija.

U deset mjerenja, C++ implementacija je na sustavu sa:

- Intel Core i5 8300H brža za:

$$P_{AVG} = 1 - \frac{\sum_{i=1}^{10} t_{cpp,8300H,i}}{\sum_{i=1}^{10} t_{py,8300H,i}} \times 100 = 93{,}49\ \%$$

- Intel Core i5 10400 brža za:

$$P_{AVG} = 1 - \frac{\sum_{i=1}^{10} t_{cpp,10400,i}}{\sum_{i=1}^{10} t_{py,10400,i}} \times 100 = 95{,}636\ \%$$

- Intel Core i7 7700 brža za:

$$P_{AVG} = 1 - \frac{\sum_{i=1}^{10} t_{cpp,8300H,i}}{\sum_{i=1}^{10} t_{py,8300H,i}} \times 100 = 95{,}047\ \%$$

Implementacija u C++ je za razliku od *Python* implementacija, kroz sva tri sustava, brža za:

$$P_{AVG} = 1 - \frac{\sum_{j=1}^{3} \sum_{i=1}^{10} t_{cpp,j,i}}{\sum_{j=1}^{3} \sum_{i=1}^{10} t_{py,j,i}} \times 100 = 94{,}558\ \%$$

5. ZAKLJUČAK

Kako bi elektronika vozila bila pouzdana i sa potpunom sigurnošću se počela koristiti u autima, od velike je važnosti detaljno testirati svu potrebnu opremu. Prije samog korištenja ECU-ova u automobilu, potrebno je najprije odraditi testiranje njihovog rada koristeći razne simulacije, odnosno testna okruženja, kako bi se provjerilo radi li komunikacija među kanalima ECU-ova kako je očekivano. Upravo za tu svrhu koristi se generator testnog okruženja, odnosno TEG. Korištenjem TEG-a moguće je izrađivati različita testna okruženja i simulacije i na taj način učinkovito provoditi veliki broj testova, te kao posljedica toga povećati pouzdanost rada sustava. TEG generator izvornog C koda predstavlja dio TEG-a koji omogućuje glavnu funkcionalnost i povezivanja ostalih dijelova TEG-a kako bi se uspješno generirale programske komponente. TEG generator izvornog C koda se dijeli na tri veća modula, koja obavljaju različite zadaće. To su: parser fragmenata i konfiguracijske tablice generatora, apstrakcije zamjene semantičkih oznaka varijabli i apstrakcije popunjavanja implementacijskih predložaka određenim fragmentima. Pomoću parsera fragmenata postignuto je čitanje podataka iz datoteka fragmenata te njihovo spremanje u memoriju, a pomoću parsera konfiguracijske tablice generatora postignuto je čitanje strukturiranih podataka unutar datoteke konfiguracijske tablice generatora te njihovo spremanje u memoriju. Uz pomoć apstrakcija za zamjenu semantičkih oznaka varijabli omogućena je jedna od funkcionalnosti koja je potrebna za pravilnu obradu fragmenata programskih komponenti u ovisnosti o programskoj komponenti koja se generira. Uvođenjem alternative rekurzije, petljom se postigla dinamička zamjena semantičkih oznaka varijabli kroz bilo koju dubinu zamjena semantičkih oznaka varijabli. Apstrakcijama za smještanje fragmenata postignuta je funkcionalnost kojim se generira sama programska komponenta. Također, smještanje fragmenata omogućuje dinamičko generiranje programskih komponenti kroz dva koraka i njihov rad obuhvaća sve prethodne implementacije za uspješno popunjavanje implementacijskih predložaka i generiranje programskih komponenti. Implementacija TEG generatora izvornog C koda u C++ programskom jeziku se za razliku od postojeće *Python* TEG implementacije izvršava većom brzinom u prosjeku 15 puta. Također, testovi brzine izvršavanja TEG-a provedeni su na tri različita sustava, gdje svaki daje približne vremenske omjere rezultata uz *Python* implementaciju. Testovi validacije programskih komponenti nisu provedeni jer takva funkcionalnost nije implementirana niti u postojećem *Python* TEG-u. Postoji moguće unaprjeđenje generatora izvornog C koda opisanoga u ovome radu kojim bi se pametno generirala sučelja programskih komponenti uvođenjem funkcija za izvršavanje sučelja nalik

objektno orijentiranoj paradigmi. Time bi se smanjio broj poziva sučelja u samoj programskoj komponenti te bi se ostvarila dodatna efikasnost veličine generiranog koda.

LITERATURA

[1] Vector Informatik GmbH, "AUTOSAR Fundamentals," pp. 26-32.

[2] TTTech prezentacija zatvorenog izvora, slajd 3/11.

[3] A. Joy, »5 Main Disadvantages Of Python Programming Language,« Pythonista Planet, [Mrežno]. Available: https://pythonistaplanet.com/disadvantages-of-python.

[4] L. Zou, »Parallelising in Python (mutithreading and mutiprocessing) with practical templates,« Medium, 16 June 2019. [Mrežno]. Available: https://medium.com/pythonexperiments/parallelising-in-python-mutithreading-and-mutiprocessing-with-practicaltemplates-c81d593c1c49.

[5] Python Software Foundation, »pickle — Python object serialization,« Python Software Foundation, 20 July 2021. [Mrežno]. Available: https://docs.python.org/3/library/pickle.html.

[6] F. Hocutt, »Using Pickle,« Python Software Foundation, 15 December 2019. [Mrežno]. Available: https://wiki.python.org/moin/UsingPickle.

SAŽETAK

U ovom radu predložen je i opisan TEG generator izvornog koda napisanog u C++ programskom jeziku koji nastoji riješiti problem ručnog pisanja testova komunikacijskih sučelja automobila. Generator izvornog C koda jedan je od glavnih dijelova TEG-a koji za zadaću ima generirati datoteke programskih komponenti s izvornim C kodom kojima se testiraju funkcionalnosti komunikacijskih sučelja automobila. Sastoji se od tri veća modula koji su predviđeni za obavljanje cjelokupnog generiranja programskih komponenti no podjelom na više dijelova. To su: parser fragmenata programskih komponenti i konfiguracijske tablice generatora, apstrakcije za zamjenu semantičkih oznaka varijabli fragmenata i apstrakcije procesiranja fragmenata i ispis popunjenih implementacijskih predložaka. Ovim radom ostvarena je funkcionalnost generiranja programskih komponenti u C++ programskom jeziku za verziju programa koji testira sučelja u svrhu bržeg i efikasnijeg generiranja datoteka programskih komponenti. Rješenja postignuta ovim radom su brzo i memorijski efikasno generiranje testnih programskih komponenti TEG-a.

Ključne riječi: ECU, generator testnog okruženja, TEG, AUTOSAR, C++ programski jezik, automatizacija testova, programski alat

Development of a C code generator within a test environment generator

ABSTRACT

In this thesis, a TEG source code generator solution written in C++ programming language is suggested and described which tends to solve the problem of having to manually write tests of communication interfaces of a car ECU. Source code generator is one of the main parts of TEG whose task is to generate software component files filled with source code which is used to test the functionalities of the communication interfaces of a car ECU. TEG source code generator is built of three major modules which are used to do the work of all software component generations but are divided in multiple parts. There are: software component fragments parser and generator configuration table parser, abstractions for the replacement of semantic tags of variables present in fragment code and abstractions for processing fragments and outputting filled out implementation templates. With this thesis, a functionality of generating software components in C++ programming language for the program version of program for interface testing, is achieved, for faster and more efficient generation of software component source code files. Solutions achieved by this thesis are fast and memory efficient generation of test software components of the TEG.

Keywords: ECU, test environment generator, TEG, AUTOSAR, C++ programming language, automated tests, software tool

ŽIVOTOPIS

Dino Brkić rođen je 24. prosinca 1997. u Požegi. Završio je Osnovnu školu Dobriše Cesarića u Požegi. Nakon završene osnovne škole upisuje Tehničku školu Požega smjer elektroenergetika. Po završetku srednje škole 2016. godine upisuje sveučilišni preddiplomski studij računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. Trenutno je student druge godine diplomskog studija računarstva modul DRD – Informacijskih i podatkovnih znanosti.

_____

Potpis autora

PRILOZI

Prilog P.4.1 – Generirana testna programska komponenta, nalazi se u digitalnom formatu na CD-u priloženom u mapi *test_output* pod nazivom *dummyTemplate_lidliTestSWC_replacedPlaceholders.c*.