Razvoj parsera za AutoSAR model sustava unutar generatora testnog okruženja

Širac, Filip

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

Sveučilišni diplomski studij

Razvoj parsera za AutoSAR model sustava unutar generatora testnog okruženja

Diplomski rad

Filip Širac

Osijek, 2021.

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:	Filip Širac
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1093R, 06.10.2019.
OIB studenta:	99172256542
Mentor:	Izv. prof. dr. sc. Marijan Herceg
Sumentor:	
Sumentor iz tvrtke:	Marko Halak
Predsjednik Povjerenstva:	Izv.prof.dr.sc. Ratko Grbić
Član Povjerenstva 1:	Izv. prof. dr. sc. Marijan Herceg
Član Povjerenstva 2:	Izv. prof. dr. sc. Mario Vranješ
Naslov diplomskog rada:	Razvoj parsera za AutoSAR model sustava unutar generatora testnog okruženja
Znanstvena grana rada:	**Telekomunikacije i informatika (zn. polje elektrotehnika)**
Zadatak diplomskog rada:	"Automatsko testiranje sastavni je dio razvoja proizvoda, bilo softverskih, bilo hardverskih, u automobilskoj industriji. Jedan od sastavnih dijelova generatora testnog okruženja je i onaj zadužen za parsiranje datoteka koje opisuju model sustava sa svim softverskim komponentama i sučeljima u okviru AutoSAR standarda. U sklopu ovog zadatka potrebno je upoznati se s trenutnom implementacijom spomenutog dijela generatora testnog okruženja, pronaći njegove nedostatke i napraviti plan implementacije novog rješenja. Potom je potrebno implementirati novo rješenje u programskom jeziku C++. Novo rješenje treba podržavati univerzalni pristup parsiranju tako da se može primijeniti na više projekata bez potrebe za adaptacijom, stvoriti osnovu glavne baze podataka i popuniti ga sa svim informacijama dostupnim iz modela sustava te pripremiti bazu podataka za sljedeći korak popunjavanja, koji će se odvijati u parseru modela za komunikacijske matrice. Nakon izrade rješenja potrebno je provjeriti njegove performanse na više primjeraka ARXML modela i usporediti ih s performansama trenutnog rješenja. Treba provesti prikladne testove koji potvrđuju ispravnost parsiranih podataka spremljenih u bazu usporedbom s podacima koje daje trenutno dostupno rješenje.(tema rezervirana za Filip Širac), (sumentor: Marko Halak, Institut RT-RK Osijek d.o.o.)
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: 1 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:	Filip Širac
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1093R, 06.10.2019.
Turnitin podudaranje [%]:	8

Ovom izjavom izjavljujem da je rad pod nazivom**: Razvoj parsera za AutoSAR model sustava unutar 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

Današnja automobilska industrija koristi visoku tehnologiju u razvoju automobila kako bi osigurala udobnost, praktičnost i sigurnost. Automobili imaju mnoštvo značajki kao što su masaža sjedala, ambijentalno osvjetljenje, digitalni zasloni koji prikazuju informacije i omogućuju kontrolu nad jedinicama klime, grijanju sjedala, hlađenju sjedala, od praktičnih značajki upravljanje sustavima u vozilu pomoću glasovnih naredbi, kamere koje snimaju određene geste za aktivaciju određenih sustava, autonomno parkiranje itd. Međutim, najbitnije su sigurnosne značajke kao što su detekcija drugog vozila u mrtvom kutu, praćenje vozila ispred te pravovremeno kočenje i mnoge druge značajke. Svim ovim kompleksnim sustavima i mnogim drugim koje posjeduju današnji automobili upravljaju elektroničke upravljačke jedinice (engl. *Electronic Control Units - ECU*), koje su poseban dio računalnih sustava nazvani ugradbeni računalni sustavi (engl. *embedded systems*). Današnji ECU-ovi visoke su složenosti s velikim brojem komunikacijskih kanala koje je potrebno testirati, a zbog količine potrebnih testova pogodno ih je automatski generirati.

Automatsko testiranje sastavni je dio razvoja proizvoda, bilo programskih, bilo sklopovskih, u automobilskoj industriji. Jedan od sastavnih dijelova generatora testnog okruženja je i onaj zadužen za parsiranje datoteka koje opisuju model sustava sa svim programskim komponentama i sučeljima u okviru AUTOSAR-a (engl. *AUTomotive Open System ARchitecture - AUTOSAR*) standarda [1]. U sklopu ovog zadatka potrebno je upoznati se s trenutnom implementacijom spomenutog dijela generatora testnog okruženja, pronaći njegove nedostatke i napraviti plan implementacije novog rješenja. Potom je potrebno implementirati novo rješenje u programskom jeziku C++. Novo rješenje treba podržavati univerzalni pristup parsiranju tako da se može primijeniti na više projekata bez potrebe za adaptacijom, stvoriti osnovu glavne baze podataka i popuniti ga sa svim informacijama dostupnim iz modela sustava te pripremiti bazu podataka za sljedeći korak popunjavanja, koji će se odvijati u parseru modela za komunikacijske matrice. Nakon izrade rješenja potrebno je provjeriti njegove performanse na više primjeraka ARXML modela i usporediti ih s performansama trenutnog rješenja. Također, treba provesti prikladne testove koji potvrđuju ispravnost parsiranih podataka spremljenih u bazu usporedbom s podacima koje daje trenutno dostupno rješenje.

2. POSTOJEĆE RJEŠENJE GENERATORA TESTNOG OKRUŽENJA

Generator testnog okruženja (engl. *Test Environment Generator - TEG*) je alat pisan u Pythonu 2.7., vlasništvo je tvrtke TTTech te je razvijen za automatsko testiranje platformi odnosno testiranje komunikacije u automobilskim sustavima koristeći pravila AUTOSAR standarda. Ovaj standard opisuje troslojna arhitektura u kojoj se TEG smjestio u posredniku (engl. *middleware*) između ulaznih signala sa sklopovlja odnosno ECU-ova i aplikacijskog sloja u AUTOSAR arhitekturi.

TEG generira testno okruženje na temelju modela jednog ECU-a i projekta kojeg testira i validira (*middleware*), odnosno generira testove za provjeru određenih komponenti sustava između ulaznih signala koji ulaze u jedan ECU i komponenti koje ih primaju. Testno okruženje se nakon generiranja stavlja na ploču i poprima oblik kao *runtime* okruženje (engl. *runtime environment - RTE*) koje se testira putem posebno dizajniranih programskih sučelja (engl. *interface*). Sučelja su u programskoj komunikacijskoj strukturi smještena u zasebni komunikacijski kanal paralelan s RTE, što omogućuje kroz kompleksno zakazivanje *runtimea*, testiranja okruženja u (gotovo) stvarnom vremenu. Postoji više oblika testnih okruženja koja se manifestiraju kao testni *build*-ovi koji se grade na temelju osnovnog tj. sistemskog *build*-a, a svaki može testirati jednu ili više funkcionalnosti te simulirati opterećenje za jedan ili više servisa koji oponašaju prave servise viđene u automobilskim uređajima u stvarnom okruženju.

Trenutno rješenje ima određene mane koje je potrebno redizajnirati i optimizirati. One će biti detaljnije objašnjene u zadnjem pod poglavlju ovog poglavlja, a one su:

- *Python 2.7*
- Problem isporuke programskog rješenja samog TEG-a
- Parser
- Spremanje podataka
- Ručno pisane putanje

2.1. AUTOSAR

AUTOSAR je skup standarda stvoren od strane svih tvrtki koji na određene načine pridonose razvoju i proizvodnji automobila. Cilj pri stvaranju standarda bio je standardizirati sloj koji je posrednik između sklopovlja ECU-a i programa. Na taj način napisani se program u idealnom

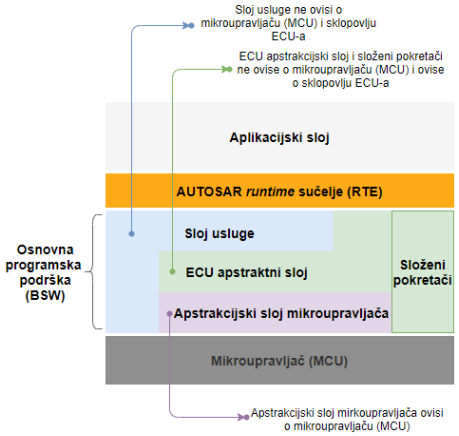
slučaju može koristiti na bilo kojem ECU neovisno o proizvođaču. AUTOSAR arhitektura ima slojeve:

- Aplikacijski sloj
- *Runtime* sučelje
- Osnovna programska podrška (engl. *Basic software – BSW)*
- Mikroupravljačka jedinica (engl. *Microcontroller unit – MCU*)

Detaljniji prikaz arhitekture AUTOSAR-a nalazi se na slici 2.1. AUTOSAR RTE je sloj koji je posrednik između aplikacijskog sloja i sloja osnovne programske podrške. Pruža usluge komunikacije za AUTOSAR programske komponente (engl. *Software Components - SWC*) i stvarnih aktuatora ili senzora u automobilu. Cilj ovog sloja je napraviti SWC-ove neovisne o konkretnom ECU.

Sloj osnovne programske podrške BSW je sloj koji se sastoji od više pod slojeva. Pruža osnovne usluge i programske module za različite aplikacije. Sloj usluga (engl. *Services Layer*) najviši je BSW sloj koji pruža funkcije:

- Mrežna komunikacija i upravljanje vozilom
- Usluge upravljanja memorijom (NVRAM upravljanje)
- Usluge dijagnostike
- Upravljanje stanjem ECU-a [1]



Sl. 2.1.Arhitektura AUTOSAR-a [1]

ECU sloj apstrakcije (engl. *Abstraction Layer*) daje pristup vanjskim jedinicama i uređajima neovisno o tome nalaze li se u mikroupravljaču ili ne. Također, pruža programsko sučelje prema mikroupravljaču. Njegov cilj je ostvariti da viši programski slojevi budu neovisni o ECU.

Sloj složenih pokretača (engl. *Complex Drivers*) je sloj koji pristupa direktno mikroupravljaču. Sadrži složene funkcije koje ostali slojevi ne sadrže kao što su kontrola ubrizgavanja goriva, kontrola električnih vrijednosti, itd. Zadatak mu je upravljati složenim funkcijama koje su strogo osjetljive na vrijeme u kojemu trebaju biti izvršene, složeni senzori i aktuatori.

Apstrakcijski sloj mikroupravljača (engl. *Microcontroller Abstraction Layer - MCAL*) je programski modul koji direktno pristupa hardveru MCU. Cilj mu je da više programske slojeve čini neovisnima o samom hardveru.

2.2. Trenutni Generator testnog okruženja

Svaki proizvođač koristi svoju konfiguraciju sklopovlja u automobilu iz čega proizlazi potreba za testiranjem novog ECU-a. Svaki ECU ima određene podatke o sebi koje treba izvući iz konfiguracije, te se pomoću TEG-a ti podatci parsiraju, pohranjuju i zatim se iz njih generiraju programske komponente pomoću kojih se testira model.

Podatci o modelu sadržani su u datoteci *.arxml* formata koji je zapravo jezik za označavanje podataka (*engl. Extensible Markup Language - XML*) uz određena ograničenja s dodatnim izmjenama prema AUTOSAR standardu. Ti podatci se parsiraju i spremaju u temeljnu klasu TEG-a koja mimikrira strukturu ARXML datoteke na sažet i smislen način za daljnje generiranje testnog okruženja u *Pythonu.*

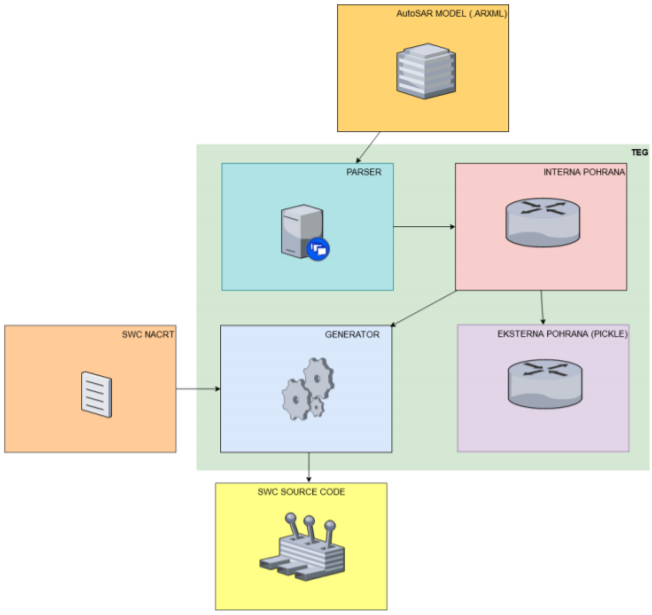
Nakon što je objekt popunjen odnosno kada je utvrđeno da su podatci parsirani, podatci se zatim serijaliziraju koristeći *Python* modul *Pickle. Pickle* je modul visoke jednostavnosti za serijalizaciju podataka, koji pruža mogućnost spremanja i kasnije učitavanja u *Python* objekt koristeći poseban binarni format. Za razliku od npr. *json-a* (engl. *JavaScript Object Notation*), nije ograničen na jednostavne objekte [2]. Postoji i modul *Marshal* koji je nešto jednostavniji, ali njegova usporedba s *Pcikle*-om bit će detaljno objašnjena u idućem pod poglavlju.

Zatim je potrebno iz prikupljenih podataka generirati testne komponente. Generira se četiri glavna testna okruženja:

- RTE sučelja za unutarnju komunikaciju

- CAN (engl. *Controller Area Network*) za vanjsku komunikaciju
- *Persistency* testovi specifični za trenutni projekt
- Prilagođeni testovi za samu platformu

Funkcija za generiranje testnog koda prima kao argumente objekt temeljne klase TEG-a i lokacije glavne *.c* datoteke s predlošcima i potrebnim SWC komponentama, također u obliku *.c* datoteka. Glavna C datoteka sadrži predloške C koda u obliku definicija varijabli, klasa i funkcija s predodređenim mjestima za smještanje vrijednosti prikupljenih iz ECU-a. Ona se metodom regularnih izraza (engl. *Regular Expressions, RegEx*), koristeći logički filtrirane podatke iz temeljnog objekta, raščlanjuje na spomenute predloške, tj. izreze koda. Predlošci se umeću na predodređena mjesta u kodu SWC *.c* datoteka, a zatim se ponovno iz temeljnog objekta na predložena mjesta umeću varijable i vrijednosti specifične za ECU koji se testira [3].



Sl. 2.2. Pojednostavljena shema rada trenutnog generatora testnog okruženja [3]

2.3. Mane trenutnog Generatora testnog okruženja

Kao što je na samom početku poglavlja navedeno osnovni problem trenutnog Generatora testnog okruženja je što je pisan u *Pythonu 2.7.* koji više nema službenu podršku, što znači da ne može koristiti novije biblioteke i poboljšane funkcionalnosti. Prema tome ne može koristiti unaprijeđene

funkcije i biblioteke koje pruža *Python* 3. Iz problema s verzijama jezika proizlazi potreba za prelaskom na noviju verziju, ali je TEG visoko kompleksan program čija bi prerada (engl. *refactoring*) bila neisplativa u omjeru dobivenih performansi i uloženog vremena, stoga je potrebno cijeli program redizajnirati i ponovno napraviti. *Python* jezik je jednostavan i lako čitljiv čovjeku, ali je jezik visoke apstrakcije te bi bilo pogodno koristiti neki niži jezik kao što su C ili C++. Niži programski jezici imaju brže vrijeme prevođenja i izvršavanja. S druge strane *Python* za prevođenje koristi prevoditelj (engl. *interpreter*) koji ovaj visoko apstraktni jezik koji je čovjeku lagan za razumijevanje prevodi u računalu razumljiv zapis. Prva komponenta unutar strukture prevoditelja je leksička analiza. Njezin zadatak je da ulazni tok podataka složi u smislene dijelove koda zvani žetoni (engl. *tokens*). Iduća komponenta je parser koji te smislene dijelove koda parsira i dalje prosljeđuje kompajleru. Problematika koja uvelike utječe na preformanse je u što se u drugim višim jezicima prethodno kompajlirani kod predaje prevoditelju, dok je u ovom slučaju kompajler integriran u prevoditelj. U idealnom slučaju kompajliranje se izvršava u vremenu izvođenja (engl. *runtime*) programa. Visoka složenost TEGa u kombinaciji s *Python* jezikom i njegovim predvoditeljom daje dosta lošije vremenske rezultate izvođenja.

Visoka modularnost koda je dobra jer je sve apstraktno i odvojeno, te je na taj način kod dosta stabilan i otporan na greške. Svi zadatci koje izvršava određena funkcija su minimalni, sve što posjeduje neku dodatnu obradu podataka izdvojeno je u novu funkciju što je razumljivije programeru koji se pokušava upoznati s ovim rješenjem. Problem je što se kod ponavlja te se rade iste stvari na više mjesta, ali s drugim imenima varijabli. Ovako velika razdvojenost koda je zahtjevna za testiranje.

Također jedan od problema dolazi s poslovne strane. Naime, kako bi se proizvod odnosno TEG isporučio klijentu, potrebno mu je dostaviti izvorni kod (engl. *source code*) kako bi ga prevoditelj mogao izvršiti. Na taj način daje se proizvod u ruke klijenta potpuno otvoren i ranjiv izmjenama. Proizvod se ne može isporučiti u stilu crne kutije (engl. *black box*) u koju tvrtke koje kupe program stave određene podatke i bez znanja kako to funkcionira dobiju rezultat što je generalni problem očuvanja integriteta izvornog koda koji je tada dostupan svima.

Nadalje, trenutni parser nije dinamičan što znači da se kod svakog novog projekta stvara potreba za dodavanjem određenih funkcionalnosti kako bi se parsirali točno određeni podatci koji mogu biti strukturirani pod drugim imenima. Svaki put potrebno je dodati nove zakrpe (engl. *patch*) koje će odraditi dio koji će pokupiti nove podatke. Na taj način se troši novac i vrijeme kako bi se TEG parser prilagodio novom projektu. Trenutni TEG nema mehanizam s kojim bi provjerio jesu li

uzeti svi potrebni podatci iz modela te bi bilo potrebno stvoriti nekakav sustav koji bi to provjeravao.

Idući nedostatak trenutnog rješenja je pohrana podataka u bazu TEG-a. Za pohranu podataka koristi se *Pythonov* modul koji se zove *Pickle. Pickle* modul implementira binarne protokole za serijalizaciju i deserijalizaciju *Python* objekata. *Pickling* je dio proces serijalizacije u kojem se hijerarhija objekta pretvara u binarni tok podataka, a *Unpickling* je obrnuta metoda odnosno pretvara podatke iz binarne datoteke nazad u hijerarhiju objekta [4]. Problemi korištenja ovog modula za spremanje podataka proizlaze iz toga da se serijalizirani objekti koji su spremljeni u binarnu datoteku ne mogu nikako provjeriti jesu li ispravni. Na taj način bilo tko može izmijeniti tu datoteku, pročitati njen sadržaj bez dozvole i ugroziti integritet podataka ili dodati neki zlonamjerni dio koda. Zatim takav kod s potencijalnim malicioznim dijelovima kod *Unpickling*-a daje određene instrukcije koje *Python* izvrši i potencijalno nanese štetu sustavu, ovisno o jačini zlonamjernosti samog izmijenjenog koda.

Marshal serijalizacija postoji uglavnom zbog podrške *.pyc* datotekama, ali generalno preferirani način serijalizacije *Python* objekata je *Pickle.* Razlike između ova dva modula su sljedeće:

- *Pickle* modul vodi računa o objektima koje je već serijalizirao tako da reference koje kasnije pokazuju na isti objekt neće serijalizirati ponovo dok *Marshal* to ne radi.
  Ovo svojstvo modula ima direktan utjecaj na rekurzivne objekte i dijeljene objekte. Rekurzivni objekti su objekti koji imaju referencu na samoga sebe i s njim se ne može rukovati pomoću *Marshal* modula, čak ako se pokušaju serijalizirati pomoću ovog modula *Pythonov* prevoditelj će se srušiti. Dijeljeni objekti su objekti koji imaju više referenci na isti objekt na različitim mjestima. *Pickle* takve objekte spremi samo jednom i osigura da sve ostale reference pokazuju na glavnu kopiju (engl. *master copy*). Na taj način dijeljeni objekti ostaju dijeljeni što je vrlo bitno za promjenjive objekte.
- *Marshal* ne može biti korišten za serijalizaciju korisnički definiranih klasa i njenih objekata. *Pickle* može spremiti i povratiti objekte transparentno, ali definicija klase mora biti moguća za uvoz (engl. *importable*) i mora „živjeti“ u istom modulu kao i kada je objekt spremljen.
- *Marshal* serijalizacijski format ne garantira primjenjivost na različitim verzijama *Pythona*, jer je njen osnovni posao podrška *.pyc* datotekama. S druge strane *Pickle* serijalizcijski format garantira kompatibilnost s prethodnim *Python* verzijama [4].

Nadalje, razlika između *Pickle* i *Json*-a je:

- JSON je tekstualno serijalizacijski format (izlaz mu je *unicode* tekst koji je u velikom broju slučajeva enkodiran u utf-8), dok je *Pickle* binarno serijalizacijski format.
- JSON je čitljiv čovjeku.
- *Pickle* je specifično korišten modul unutar *Python*-a, a JSON ima široku primjenu i interoperabilan je za korištenje i izvan *Python*-a.
- JSON može predstaviti samo dio ugrađenih tipova u *Pythonu* i ne može predstaviti korisnički definirane klase, dok *Pickle* može predstaviti velik broj *Python* tipova.[4]

Također postoji još jedan problem trenutnog rješenja, a ono je da sve datoteke potrebne za rad TEG-a imaju ručno upisane putanje do tih datoteka u jednoj XML datoteci. Na taj način ako bilo koja od tih ulaznih datoteka promijeni lokaciju TEG vrlo vjerojatno neće raditi, odnosno potrebno je ručno pronaći i upisati putanju koja je ujedno i lokacija te datoteke u XML datoteku.

3. IZRADA MODEL PARSERA

U sklopu ovog diplomskog rada zadatak je bio upoznati se s postojećim rješenjem generatora testnog okruženja, pronaći njegove nedostatke i napraviti plan implementacije novog rješenja. Kao što je u prethodnom poglavlju prikazano trenutni TEG ima brojne mane. Dvije mane koje ovaj diplomski rad rješava su programski jezik *Python 2.7.* u kojem je pisano postojeće rješenje i model parser kao cjelina unutar TEG-a.

Kako bi se riješili nedostatci postojećeg rješenja, potrebno je napraviti novo rješenje model parsera u programskom jeziku C++. Novo rješenje treba podržavati univerzalni pristup parsiranju, odnosno omogućiti da se model parser može primijeniti na više projekata bez potrebe za adaptacijom, stvoriti strukturu baze podataka i popuniti ga sa svim informacijama dostupnim iz modela sustava. Nadalje, potrebno je pripremiti bazu podataka za sljedeći korak popunjavanja koji će se odvijati u parseru modela za komunikacijske matrice.

3.1. Okolina i tehnologija razvoja Model parsera

Model parser pisan je u razvojnom okruženju *Visual Studio* u programskom jeziku C++. *Visual Studio* je integrirano razvojno okruženje (engl. *Integrated development environment – IDE*) koje je napravljeno od strane Microsofta, koje služi za razvoj računalnih programa kao i aplikacija za primjenu na webu ili mobilnim uređajima. *Visual Studio* uključuje uređivač koda koji podržava automatsko dovršavanje koda, odnosno ako se unosi ime funkcije ili određene varijable okruženje će samo prepoznati o kojoj se riječi radi i ponuditi sve koje počinju tim setom slova i dovršiti pisanje. Isto tako podržava i preradu koda, npr. prerada imena funkcije koja se poziva na deset mjesta uz pomoć prerade imena promijeni se naziv funkcije u svim njenim pozivima. Ima integrirani ispravljač pogreške (engl. *debbuger*) koji radi kao ispravljač pogreške izvorne razine i ispravljač pogreške strojne razine. Ostali ugrađeni alati koje pruža ovo razvojno okruženje jesu kod profili (engl. *code profilers*) koji služe za dinamičku analizu programa za mjerenje određenih vremena, složenost programa, broj poziva određene funkcije, korištenje memorije, u globalu glavni cilj ovog alata je analiza za optimizaciju vremena, memorije kako bi se poboljšale performanse određenog programa. Zatim dizajner za izradu aplikacija grafičkog korisničkog sučelja (engl. *graphical user interface - GUI*). GUI aplikacije imaju formu koja je prikladna za interakciju korisnika s raznim elektroničkim uređajima kroz grafičke prozore i gumbove, te kroz određenu akciju korisnika [5].

Prema [6], arhitektura Visual Studia ne podržava programski jezik, rješenja ili alate kao unutarnju komponentu IDE-a već omogućuje priključke koji uvode određene funkcionalnosti kroz VSPackage. Nakon što je instaliran, funkcionalnost je dostupna kao određeni servis. IDE pruža tri servisa:

- *SvsSolution* pruža mogućnost numeriranje projekata i rješenja
- *SVsUIShell* omogućuje UI funkcionalnosti kao što su npr. alatne trake i kartice
- *SvsShell* koji se koristi za registraciju *VSPackages*-a

Prema [7], oko 1972. se pojavljuje jezik C, koji je direktna preteča današnjeg jezika C++. To je bio prvi jezik opće namjene te je postigao neviđen uspjeh. Više je razloga tome:

- Jezik je jednostavan za učenje
- Omogućava modularno pisanje
- Sadržava samo naredbe koje se mogu jednostavno prevesti u strojni jezik
- Ima puno brže izvođenje napisanog koda

Jezik nije bio opterećen mnogim složenim funkcijama, kao na primjer skupljanje smeća (engl. *garbage collection*), a u slučaju potrebe za skupljanjem smeća programer ga je trebao sam realizirati. Programer je imao dobru kontrolu i uvid nad raspoloživim resursima što je pružalo dobra optimizacijska svojstva. Zbog sve veće složenosti programa iskazala se potreba za stvaranjem dodatnih mehanizama koji će omogućiti jednostavniju izradu i održavanje, te mogućnost korištenja napisanog koda na drugim projektima. Bjarne Stroustrup je započeo rad na novom jeziku koji će kasnije postati C++ i imati malo drugačiji pristup programiranju, uvodi novi način na koji će programeri pristupati problemu u odnosu na C. Prema [7], četiri su osnovna svojstva jezika C++ koja ga čine objektno orijentiranim:

- Enkapsulacija (engl. *encapsulation*)
- Skrivanje podataka (engl. *data hiding*)
- Nasljeđivanje (engl. *inheritance*)
- Polimorfizam (engl. *polymorphism*)

Ova svojstva su temelj objektno orijentirane paradigme programiranja. Objektno orijentirano programiranje, skraćeno OOP, prikazuje program u obliku objekata i klasa, objekti predstavljaju modele objekata u stvarnom svijetu, a klase su opisne strukture koje govore kakav je taj objekt i

što on može raditi. Ovim načinom programiranja dobiva se mogućnost ponovnog korištenja koda (engl. *code reusability*) i puno je jednostavnije razumijevanje.

Prema [8], neke od značajki ovog jezika su:

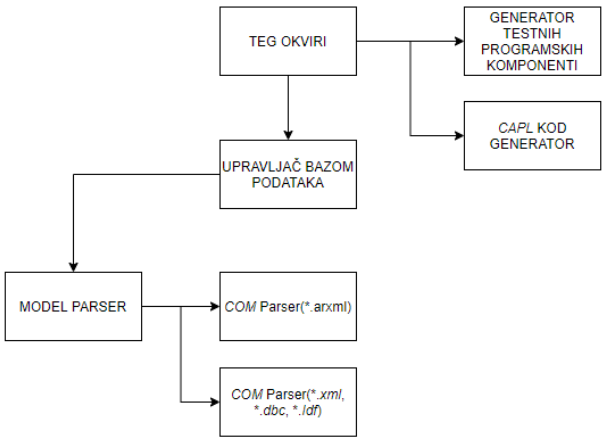
- C++ je standardiziran od strane ISO odbora za standarde 1998. godine.
- C++ se kompajlira direktno u strojni jezik što mu omogućuje veliku brzinu ako je dobro optimiziran kod.
- U programskom jeziku C++ zahtjeva se da programer pazi na resurse s kojima raspolaže odnosno upravljanje memorijom, ali s druge strane programer dobiva kontrolu nad svime što se događa.
- Podržava eksplicitno pisanje koda (engl. *explicit typing*), što znači da programer mora točno navesti tipove prilikom deklaracije, ali podržava i implicitno pisanje gdje sustav zaključuje o kojem se tipu radi npr. uz pomoć riječi *auto* odnosno programer ne piše sam o kojem se tipu podatka radi. Ova značajka omogućuje fleksibilnost i način izbjegavanja preopširnosti tamo gdje to nije potrebno.
- Također, još jedna značajka koja omogućuje fleksibilnost je ta da provjeru konverzije ili u vremenu prevođenja (engl. *compile-time*) ili u vremenu izvođenja
- Omogućuje podršku za pisanje proceduralne, generičke i objektno orijentirane paradigme, ali i drugi načini su mogući.
- C++ je jedan od najčešće korištenih jezika, ima velik broj kompajlera koji se vrte na velikom broju različitih platformi, ako kod koristi standardna biblioteka (engl. *Standard Template Library STL*) može se izvoditi na velikom broju platformi bez potrebnih izmjena ili uz male izmijene koda. Ova značajka omogućuje portabilnost koda. STL biblioteka bit će objašnjena u nastavku rada u idućem pod poglavlju.
- Ovaj jezik nastao je na temeljima C jezika stoga je kompatibilan s gotovo cijelim njegovim kodom, te može koristiti biblioteke iz C-a uz nikakve ili male modifikacije.
- Ima veliku podršku biblioteka, postoje dva tipa biblioteka koje C++ podržava to su statične i dinamične. Statične se još zovu i arhive (engl. *archive*) sastoje se od rutina koje su kompajlirane i povezane direktno u program. Dinamične biblioteke se još zovu i dijeljene biblioteke (engl. *shared library*) sastoje se od rutina koje su učitane u program za vrijeme vremena-izvođenja. Kada se koriste dinamične biblioteke one nikada ne postanu dio izvršnog programa već ostaju zasebni dio.

TinyXML2 je biblioteka koja se koristi za parsiranje XML datoteka, ali u ovom radu je poslužila za parsiranje ARXML-a. Parsira se XML dokument i iz njega kreira objektni model dokumenata (engl. *Document Object Model - DOM*) koji može biti pročitan, mijenjan i može se spremati. DOM klasa je unutar memorijska reprezentacija XML dokumenta [9].

TinyXML2 je dizajniran da bude lagan i brz za naučiti. Sastoji se od dvije datoteke od kojih je jedna .h datoteka, a druga je *.cpp*, potrebno je te datoteke dodati u projekt u kojem postoji potreba za korištenjem ovog parsera. Prema [10], TinyXML2 koristi UTF-8 (engl. *Unicode Tranformation Format 8*) kada interpretira XML. UTF-8 je način kodiranja za *Unicode*, može prevesti bilo koji *Unicode* znak u njemu jedinstveni binarni *string*, također može izvršiti i suprotnu transformaciju.

3.2. Koncept rješenja Model parsera

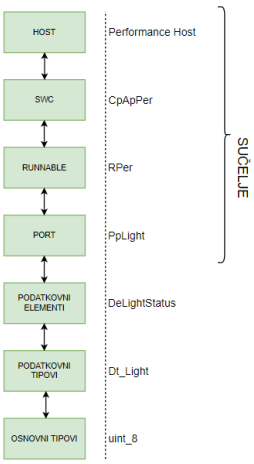
Model parser je dio TEGa. TEG se sastoji od sedam cjelina: okvir, upravljač bazom podataka, model parser, dva *com* parsera, generator *capl* koda i generator izvornog koda programskih komponenti što je prikazano na slici 3.3. Sve ove cjeline međusobno komuniciraju u ovisnosti jedna o drugoj i o raspodjeli odgovornosti unutar aplikacije. Model parser se smjestio unutar te strukture kao prvi podsustav koji dohvaća podatke iz ARXML datoteka i slaže smislen model koji se sprema u bazu. Na temelju tih podataka druga dva parsera popunjavaju podatke koje su parsirali iz svojih datoteka. Nadalje, upravljač bazom podataka pruža mogućnost generatorima za dohvaćanje podataka iz baze na temelju kojih generatori generiraju *capl* kod i kod programskih komponenti.



Sl. 3.3. Arhitektura generatora testnog okruženja

Model parseru upravljač bazom podataka prosljeđuje dvije putanje do datoteka te također model parser može biti pozvan isključivo od strane upravljača bazom podataka. Upravljač bazom podataka u sebi ima mehanizam koji provjerava je li potrebno pokretati parsiranje, a to zaključuje na temelju toga postoji li već spremljena baza. Model parser se pokreće u ovisnosti ima li promjena u bazi podataka. Iz *.arxml* datoteke model parser napraviti će strukturu baze podataka na temelju parsiranih podataka.

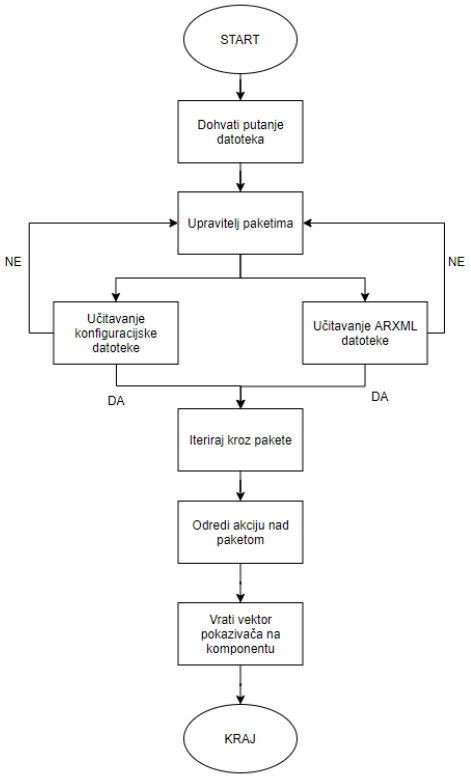
Na slici 3.4. nalazi se pojednostavljeni grafički prikaz kako komponente ovise jedna o drugoj, krenuvši od vrha strukture na ECU. U ovom radu postoje dva domaćina (engl. *hosts*), a to su domaćin sigurnosti (engl. *saftey host - SH*) i domaćin izvedbe (engl. *performance host - PH*). PH je najviši u strukturi i u njemu se nalaze programske komponente. Svaka programska komponenta se može promatrati kao funkcija koja poziva *runnable* funkciju. Postoje dvije vrste *runnabla*, *runnable init* koji se izvrše samo jednom i ciklični koji se izvršavaju svakih dvadeset mili sekundi. U tom vremenskom periodu pozivaju se portovi odnosno njegove funkcije šalji (engl. *send*), primi (engl. *receiver*), čitaj (engl. *read*), piši (engl. *write*). Navedeni dijelovi strukture su dio koji se naziva sučelje, dok ostala tri djela strukture definiraju tipove podataka. Krenuvši od najniže razine osnovni tipovi koje sam kompajler razumije zapisani su u jednoj *.h* datoteci koju koriste svi. Početni podatkovni tipovi gdje se definira da je neki podatkovni tip određenog osnovnog tipa. Zatim se ide više u strukturi gdje se podatkovni elementi opisuju kao struktura raznih podatkovnih tipova. Svi oni sadrže reference jedni na druge tako da bi se znalo npr. koji *runnable* koristi koje portove.



Sl. 3.4. Pojednostavljeni prikaz ovisnosti komponenti unutar ARXML datoteke

3.2.1. Upravljanje paketima dijagram toka

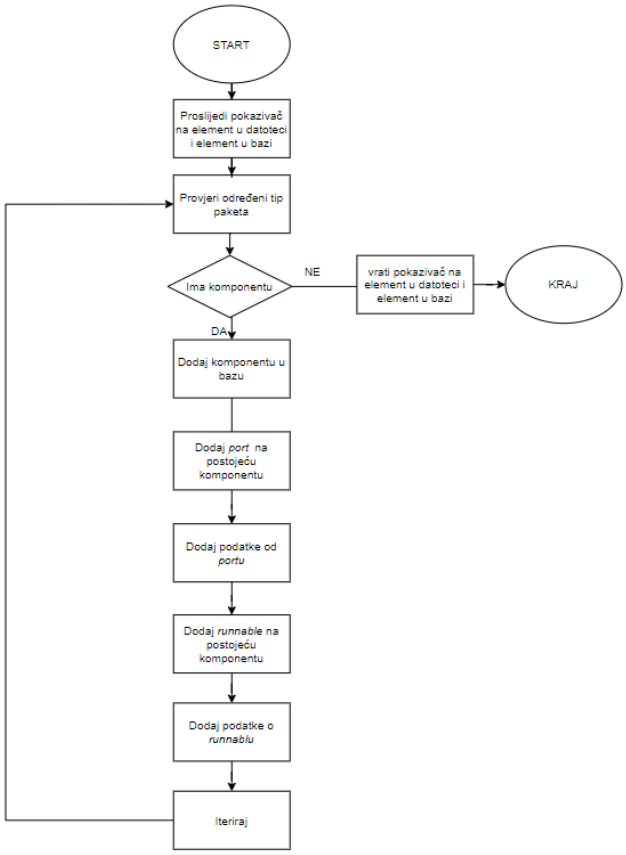
Na slici 3.5. prikazan je dijagram toka koji konceptualno opisuje učitavanje datoteka i navođenje kroz pakete. Paketi su strukture unutar *.arxml* datoteke koje opisuju programske komponente. Nakon što upravljač bazom podataka pozove model parser prvo se izvršavaju akcije opisane dijagramom na slici. Funkciji za upravljanje paketima se predaju putanje do datoteka, nakon čega upravitelj paketima učitava datoteke. Ako su datoteke učitane dohvaća se korijenski čvor u ARXML datoteci. Nakon što je dohvaćen korijenski čvor pokazivač na elemente u *.arxml* datoteci se pozicionira na dubinu paketa kako bi mogao iterirati kroz njih. Nakon provjere paketa prema ključu u konfiguracijskoj datoteci izvršava se daljnja akcija nad tim paketom. Zatim se popunjava vektor pokazivača na komponentu baze, odnosno pokazivač koji pokazuje na trenutni paket u bazi te se na kraju se vraća tako popunjen vektor.



Sl. 3.5. Dijagram toka upravljanja paketima

3.2.2. Parsiranje tipova komponenti

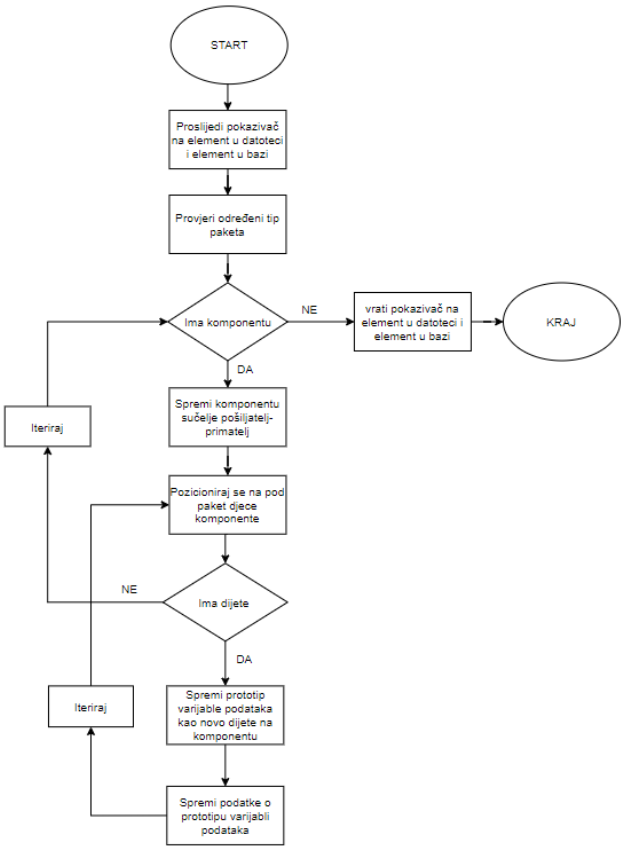
Nakon što se u funkciji koja upravlja paketima odredi da se pokazivač nalazi na paketu koji opisuje tipove komponenti, poziva se funkcija koja unutar glavnog paketa ima dva pod paketa i prima pokazivač koji pokazuje na element i pokazivač u bazi kako bi se znalo gdje se sprema element koji se parsira. Funkcija je opisana dijagramom toka koji je prikazan na slici 3.6. Ovisno o paketu na kojem se pokazivač nalazi poziva s određena funkcija za upravljanje tom vrstom paketa. Zatim se provjerava sadrži li paket programsku komponentu te ako ima, komponenta se dodaje kao novo dijete u bazu. Na tu komponentu potrebno je dodati sve *portove* i sve *runnable* koje ta komponenta ima. Prvo se sprema jedan *port* kao i njemu pripadajuće reference. Nakon što su *portovi* spremljeni spremaju se *runnable*.



Sl. 3.6. Dijagram toka parsiranja tipova komponenti

3.2.3. Parsiranje sučelja portova

Nakon što se u funkciji koja upravlja paketima odredi da se pokazivač nalazi na paketu koji opisuje sučelje portova, poziva se funkcija koja unutar glavnog paketa upravlja sučeljem pošiljatelj primatelj (engl. *sender receiver interface*) čiji je dijagram toka opisan na slici 3.7. U funkciji se provjerava ima li taj paket komponentu, te ako ima sprema komponentu u bazu s pripadajućim podatcima o prototipu varijable podataka. Funkcija vraća pokazivač na element u bazi.

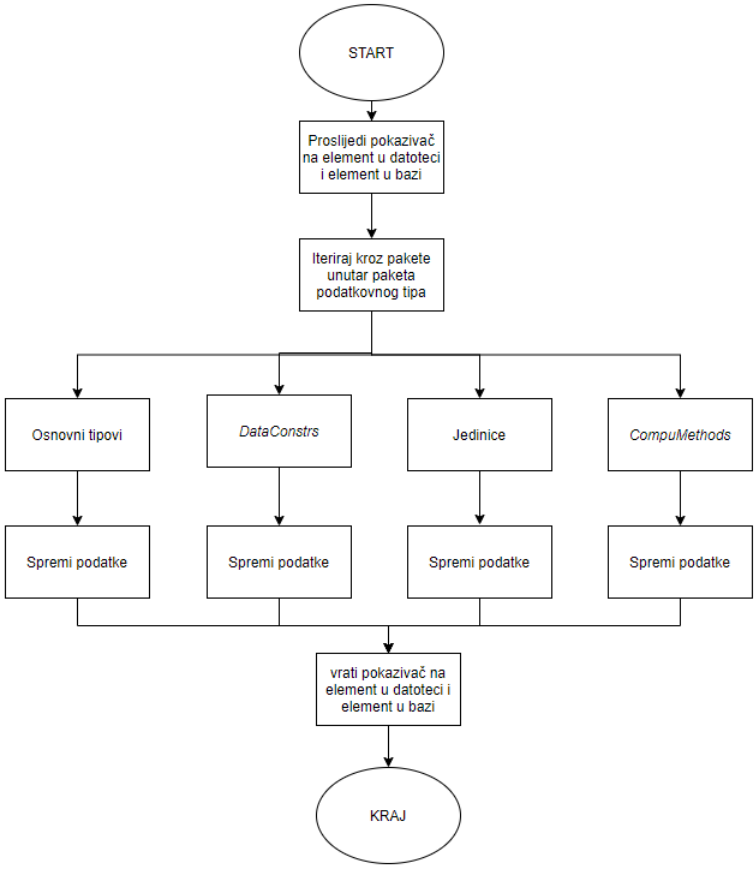


Sl. 3.7. Dijagram toka upravljanja sučeljima portova

3.2.4. Parsiranje tipova podataka

Na slici 3.8. prikazan je dijagram toka koji okvirno opisuje tijek izvođenja parsiranja i spremanja paketa koji sadrži podatke o podatkovnim tipovima. Nakon što se u funkciji upravljanja paketima odredi da se pokazivač nalazi na elementu koji je ulaz u paket podatkovnih tipova, prosljeđuje se

pokazivač na taj element i pokazivač na element u bazi funkciji koja dalje upravlja tim paketom. Zatim se iterira kroz četiri glavna pod paketa koji se nalaze u glavnom paketu, to su Osnovni tipovi, *DataConstrs*, Jedinice i CompuMethods. Prilikom svake iteracije provjerava se na kojem se paketu nalazi pokazivač te se prema tome usmjerava prema posebnoj funkciji za spremanje podataka o svakoj komponenti koja opisuje pojedini osnovni tip. Nakon što su isparsirani svi podatci iz svih pod paketa funkcija vraća pokazivač i program ide dalje.

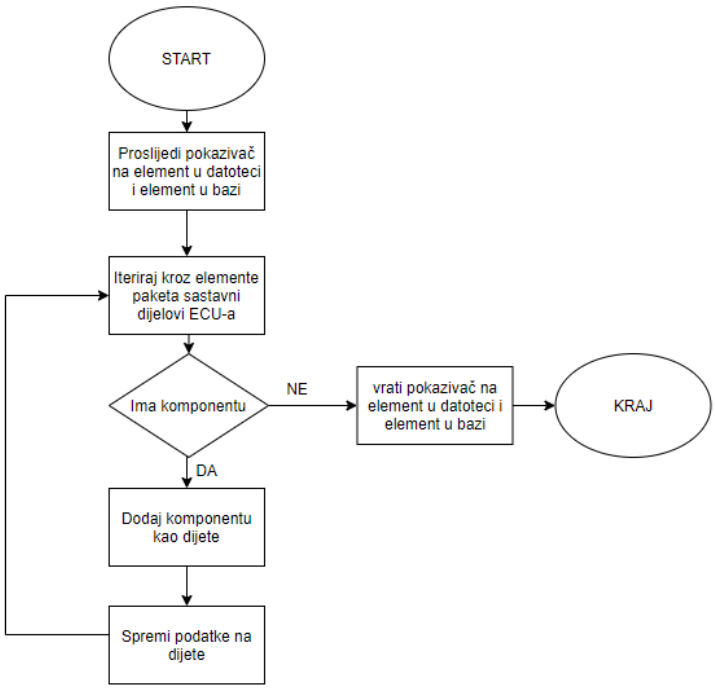


Sl. 3.8. Dijagram toka parsiranja tipova podataka

3.2.5. Parsiranje sastavnih dijelova ECU-a

Prikaz funkcije za parsiranje sastavnih dijelova ECU-a prikazana je na slici 3.9. Funkcija se poziva nakon što u funkciji za upravljanje glavnim paketima detektira da se pokazivač pri iteraciji nalazi

na ovom paketu. Funkciji se prosljeđuje pokazivač na element u bazi i pokazivač na element u *.arxml* datoteci. Zatim se iterira na novu komponentu, ako nema komponente znači da su isparsirani svi podatci ovog paketa i funkcija vraća pokazivač koji se sprema u vektor podatkovnih komponenti koji sadrži pokazivače na pakete u bazi.



Sl. 3.9. Dijagram toka parsiranja sastavnih dijelova ECU-a

3.3. Programsko rješenje Model parsera

Datoteka u kojoj su zapisani svi podatci odnosno *.arxml* datoteka pisana je prema AUTOSAR standardu, što znači da se određeni tipovi podataka koje je potrebno parsirati nalaze na istim dubinama neovisno o projektu. Parser je strukturiran tako da potrebne podatke vadi s određenih dubina na kojima se oni nalaze. Kako bi se postigla dinamičnost pri parsiranju uvedena je konfiguracijska datoteka u kojoj su opisane određene točke grananja u *.arxml* datoteci potrebne za usmjeravanja toka izvršavanja programa pri tijeku njegovog izvođenja. Podatci u konfiguracijskoj datoteci su strukturirani kao mapa. Vrijednost ovisi o trenutnom projektu, a ako se nešto promijeni

na drugom projektu mijenjat će se vrijednost u konfiguracijskoj datoteci, dok se parser neće mijenjati, već će samo na temelju ključa pokupiti novu vrijednost.

Nadalje parser je strukturiran na način da sve kreće iz jedne funkcije koja upravlja glavnim paketima. Iz funkcije za upravljanje paketima program se grana prema svakom paketu i upravlja njime. Svaki paket dalje ima nekoliko funkcija koje služe se parsiranje podataka koji se nalaze u njemu i njegovo spremanje. U svakom trenutku potrebno je znati gdje se nalazi pokazivač koji pokazuje na element u *.arxml* datoteci i pokazivač koji pokazuje na element roditelja na kojeg je potrebno spremiti dijete ili dodati podatke u bazu. U svakom trenutku ova dva pokazivača moraju biti usklađeni kako bi se točno određeni podatak stavio na pravo mjesto u bazu kako ne bi došlo do slučaja da su trenutni podatci koji se parsiraju iz datoteke. Zato je potrebno svaki put kada se doda novo dijete u bazu pomaknuti pokazivač na njega, a ne da on i dalje pokazuje na roditelja jer spremanje u tom slučaju neće biti ispravno strukturirano, te će se podatci gomilati na jednom mjestu. Još jedna bitna stavka na koju je potrebno obratiti pozornost jest da kako se ide sve dublje u strukturi podataka npr. ako se krene parsirati jedan paket, on pokazivač u bazi pomiče dublje u hijerarhiji te se sa svakim novim pod pozivom funkcije pokazivač dublje pomiče. Nakon što se izvrši neka pod funkcija prije vraćanja pokazivača na bazu potrebno ga je staviti da pokazuje ponovno na roditelja. Na taj način se osigurava kretanje po hijerarhijskoj strukturi, svaki podatak koji se vadi nekom funkcijom sprema se na mjestu u bazi koje je njemu predviđeno, pokazivač se pomiče po hijerarhiji sve dok se ne završi parsiranje paketa. Na kraju parsiranja pokazivač u bazi pokazuje na paket i sprema se u vektor objekata baze podataka.

U ovom poglavlju bit će prikazane i opisane važnije funkcije parsera pomoću pojednostavljenog C++ koda koji prikazuje grananja i izvođenje programa u malo detaljnijem prikazu nego u prethodnom poglavlju gdje su bili prikazani dijagrami toka.

3.3.1. Programsko rješenje upravljanja paketima

Na slici 3.10. prikazano je programsko rješenje funkcije za upravljanje paketima. Funkcija vraća vektor pokazivača na instancu klase baze. Funkcija prima dvije putanje gdje je jedna putanja do *.arxml* datoteke gdje se nalaze podatci, a druga putanja do konfiguracijske datoteke. Nakon toga se učitavaju datoteke, a iz *.arxml* se dohvati pokazivač na korijenski element, pozicionira se na razinu gdje se nalaze paketi istoj hijerarhijskoj razini. Nakon toga se iterira kroz pakete i provjerava poklapa li se oznaka trenutnog paketa s ključem koji opisuje taj paket prema vrijednosti koja se nalazi u mapi konfiguracijske datoteke. Funkcija vraća vektor pokazivača na svaki paket.

```
vector<shared_ptr<KlasaBaze>> upravljanjePaketima(putanjaARXML, putanjaCONFIG){
    pokazivač = ucitajARXML(putanjaARXML);
    config = ucitajCONFIG(putanjaCONFIG);

    for(dijete = pokazivač; dijete!=nullptr; dijete = dijete->brat){
        if(dijete = config[key1]){
            vector.dodajNaKraj(UpravljanjeTipovimaKomponenti(dijete,
            make_shared<KlasaBaze>(dijete)));
        }
        if(dijete = config[key2]){
            vector.dodajNaKraj(UpravljanjeSučeljimaPortova(dijete,
            make_shared<KlasaBaze>(dijete)));
        }
        if(dijete = config[key3]){
            vector.dodajNaKraj(UpravljanjeTipovimaPodataka(dijete,
            make_shared<KlasaBaze>(dijete)));
        }
        if(dijete = config[key4]){
            vector.dodajNaKraj(UpravljanjeSastavnimDijelovimaECU(dijete,
            make_shared<KlasaBaze>(dijete)));
        }
    }
    return vector;
}
```

Sl. 3.10. Programsko rješenje upravljanja paketima

3.3.2. Programsko rješenje za parsiranje tipova komponenti

Nakon što je određeno da se radi o paketu koji opisuje tipove komponenti, poziva se funkcija za upravljanje tipovima komponenti. Funkcija vraća pokazivač u bazi, a prima pokazivač na element u *.arxml* i pokazivač u bazi. Pomoću ove funkcije iterira se kroz te pakete i određuje se koji je koji pomoću uvjeta. Nakon što se odredi o kojem se paketu radi poziva se funkcija za spremanje podataka iz tog paketa. Na slici 3.11. prikazano je programsko rješenje funkcije za upravljanje tipovima komponenti.

```
shared_ptr<KlasaBaze> UpravljanjeTipovimaKomponenti(pokazivač, baza){
    for(dijete = pokazivač; dijete!=nullptr; dijete = dijete->brat){
        if(dijete = config[key_paket1]){
            baza = SpremiPaket1(dijete, baza);
        }else if{
            baza = SpremiPaket2(dijete, baza);
        }
    }
    return baza;
}
```

Sl. 3.11. Programsko rješenje upravljanja tipovima komponenti

Kada je preko uvjeta u funkciji za upravljanje tipovima podataka utvrđeno da se pokazivač nalazi na jednom od paketa poziva se npr. funkcija spremi komponentu kojoj se prosljeđuje taj pokazivač na element i pokazivač na trenutni element u bazi. Funkcija spremi komponentu također vraća pokazivač na element u bazi. Opisana funkcija prikazana je na slici 3.12.

```
shared_ptr<KlasaBaze> SpremiKomponentu(pokazivač, baza){
    if(uvjet){
        baza->dodajDijete(pokazivač);
        baza = baza->dohvatiID(pokazivač);
    }

    baza = SpremanjePortova(pokazivač, baza);

    if(uvjet){
        baza->dodajDijete(pokazivač);
        baza = baza->dohvatiID(pokazivač);
    }

    while(pokazivač){
        if(uvjet1){
            if(pokazivač = config[key5]){
                baza = SpremiRunnable(pokazivač, baza);
            }
        }
    }
    baza = baza->dohvatiRoditelja();
    return baza;
}
```

Sl. 3.12. Programsko rješenje spremanja komponenti

Funkcija za spremanje portova vraća pokazivač na element u bazi, a prima dva pokazivača koja su predana pri pozivu ove funkcije. Funkcija sprema portove i podatke koje sadrži svaki port. Programsko rješenje funkcije za spremanje portova prikazano je na slici 3.13.

```
shared_ptr<KlasaBaze> SpremanjePortova(pokazivač, baza){

    for(dijete = pokazivač; dijete!=nullptr; dijete = dijete->brat){
        baza->dodajDijete(dijete);
        baza = baza->dohvatiID(dijete);

        for(dijete1 = dijete; dijete1!=nullptr; dijete1 = dijete1->brat){
            baza->dodajPodatke({dijete1->vrijednost(), dijete1->tekst()})
        }
    }
    baza = baza->dohvatiRoditelja();
    return baza;
}
```

Sl. 3.13. Programsko rješenje za spremanje portova

Funkcija spremanje *runnable*-a vraća pokazivač na element u bazi, a prima dva pokazivača koja su predana pri pozivu ove funckije. Funkcija sprema *runnable* i pripadajuće podatke. Opisano programsko rješenje funkcije za spremanje *runnable*-a prikazano je na slici 3.14.

```
shared_ptr<KlasaBaze> SpremanjeRunnable(pokazivač, baza){

    for(dijete = pokazivač; dijete!=nullptr; dijete = dijete->brat){
        if(uvjet){
            baza->dodajDijete(dijete);
            baza = baza->dohvatiID();

            for(dijete1 = dijete; dijete1!=nullptr; dijete1 = dijete1->brat){
                if(uvje1 && uvijet2){
                    baza->dodajDijete(dijete1);
                    baza->dodajPodatke({dijete1->vrijednost(), dijete1->tekst()});
                }
            }
        }
    }
    baza = baza->dohvatiRoditelja();
    return baza;
}
```

Sl. 3.14. Programsko rješenje spremanja *runnable*-a

3.3.3. Programsko rješenje za parsiranje sučelja portova

Nakon što je određeno da se radi o paketu koji sadrži sučelja portova, poziva se funkcija za upravljanje tipovima paketom koja je prikazana na slici 3.15. Funkcija vraća pokazivač na element u bazi, a prima pokazivač na element u *.arxml* i pokazivač u bazi. Funkcija sprema portove i izvršava pozicioniranje pokazivača u strukturi kako bi se ispravni elementi slali u daljnja grananja. Nakon toga poziva se funkcija za spremanje prototipova koja će biti objašnjena u nastavku ovog pod poglavlja. Na kraju funkcije se treba vratiti na roditelja prije nego što se vrati pokazivač.

```
shared_ptr<KlasaBaze> UpravljanjeSučeljimaPortova(pokazivač, baza){
    odrediPaket();
    baza->dodajDijete(pokazivač);
    baza = baza->dohvatiID();

    do{
        pokazivač = pokazivač->Pozicioniraj();
    }while(uvjet);

    dodatneProvjere();
    Pozicioniraj();

    baza = SpremiPrototip(pokazivač, baza);
    baza = baza->dohvatiRoditelja();
    return baza;
}
```

Sl. 3.15. Programsko rješenje upravljanja sučeljima portova

Funkcija za spremanje prototipa također vraća pokazivač na element u bazi i prima pokazivač na element u podatcima i pokazivač na element u bazi. Radi se provjera ima li djeteta, ako ima dodaj dijete i pozicioniraj pokazivač na to dijete. Ponovno je potrebno pozicionirati drugi pokazivač, zatim iterirati kroz određene elemente koji trebaju sadržavati podatke, u slučaju da ne sadrže svakako se radi provjera kako ne bi došlo do greške segmentacije (engl. *segmentation fault*). Na kraju je potrebno pokazivač vratiti na roditelja. Programsko rješenje za spremanje prototipa prikazano je na slici 3.16.

```
shared_ptr<KlasaBaze> SpremiPrototip(pokazivač, baza){
    ProvjeriDijete();
    baza->dodajDijete(pokazivač);
    baza = baza->dohvatiID();

    Pozicioniraj();
    for(dijete = pokazivač; dijete!=nullptr; dijete = dijete->brat){
        if(uvjet){
            baza->dodajPodatke({dijete1->vrijednost(), dijete1->tekst()});
        }
    }

    baza = baza->dohvatiRoditelja();
    return baza;
}
```

Sl. 3.16. Programsko rješenje funkcije spremanja prototipova

3.3.4. Programsko rješenje za parsiranje tipova podataka

Nakon što je određeno da se radi o paketu koji sadrži tipove podataka, poziva se funkcija za upravljanje tipovima podataka čije je programsko rješenje prikazano na slici 3.17. Funkcija vraća pokazivač u bazi, a prima pokazivač na element u *.arxml* i pokazivač u bazi. Pokazivač na element u *.arxml* datoteci postavlja se na svaki pod paket i provjerava se nalazi li se vrijednost na koju pokazuje trenutni pokazivač u konfiguracijskoj datoteci. Za svaki pod paket dodaje se nova komponenta u bazu, a nakon dodavanja pokazivač na element u bazi se postavlja na spremljeni element. Nakon toga pozivaju se funkcije koje spremaju točno određene pod pakete, prema tome kako su strukturirani tako se i pokazivač kreće u datoteci. Na kraju funkcije se treba vratiti na roditelja prije nego što se vrati pokazivač.

```
shared_ptr<KlasaBaze> UpravljanjeTipovimaPodataka(pokazivač, baza){
    for(dijete = pokazivač; dijete!=nullptr; dijete = dijete->brat){
        if(dijete = config[key_osnovniTipovi]){
            baza->dodajDijete(dijete);
            baza = baza->dohvatiID();
            SpremiPodatkeOT();
        }
        if(dijete = config[key_DataConstr]){
            baza->dodajDijete(dijete);
            baza = baza->dohvatiID();
            SpremiPodatkeDC();
        }
        if(dijete = config[key_Jedinice]){
            baza->dodajDijete(dijete);
            baza = baza->dohvatiID();
            SpremiPodatkeJ();
        }
        if(dijete = config[key_CompuMethods]){
            baza->dodajDijete(dijete);
            baza = baza->dohvatiID();
            SpremiPodatkeCM();
        }
    }
    baza = baza->dohvatiRoditelja();
    return baza;
}
```

Sl. 3.17. Programsko rješenje upravljanja tipovima podataka

3.3.5. Programsko rješenje za parsiranje sastavnih dijelova ECU-a

Nakon što je određeno da se radi o paketu koji sadrži sastavne dijelove ECU-a, poziva se funkcija za upravljanje sastavnim dijelovima ECU-a koja je prikazana na slici 3.18. Funkcija vraća pokazivač u bazi, a prima pokazivač na element u *.arxml* i pokazivač u bazi. Funkcija sprema komponentu koja opisuje sastavne dijelove ECU-a. Nakon toga se poziva funkcija za spremanje podataka svake komponente, prikazana je na slici 3.19. Funkcija prima dva pokazivača i vraća pokazivač na element u bazi.

```
shared_ptr<KlasaBaze> UpravljanjeSastavnimDijelovimaECUav(pokazivač, baza){

    for(dijete = pokazivač; dijete!=nullptr; dijete = dijete->brat){
        if(uvjet){
            baza->dodajDijete(dijete);
            baza = baza->dohvatiID();
            baza = SpremiECUCT(dijete, baza);
        }
    }
    baza = baza->dohvatiRoditelja();
    return baza;
}
```

Sl. 3.18. Programsko rješenje upravljanja sastavnim dijelovima ECU-a

```
shared_ptr<KlasaBaze> SpremiECUCT(pokazivač, baza){

    Pozicioniraj();
    baza->dodajPodatke({pokazivač->vrijednost(), pokazivač->tekst()});
    return baza;
}
```

Sl. 3.19. Programsko rješenje spremanja podataka sastavnih dijelova ECU-a

3.3.6. Funkcija za konfiguraciju model parsra

Funkcija za učitavanje konfiguracijske datoteke prima putanju koja je proslijeđena prilikom poziva model parsera. Povratni tip je mapa koja sadrži dva *stringa*, a ti stringovi su ključ i vrijednost. Pomoću *ifstream*-a se učitava konfiguracijska datoteka. Provjerava se je li datoteka učitana, ako je učitana dohvaća se linija po linija i parsira se datoteka prilikom svake iteracije postavi pročitanu vrijednost pod određeni ključ. Programsko rješenje za učitavanje konfiguracijske datoteke prikazano je na slici 3.20.

```
map<string, string> ucitavanjeCofigDatoteke(putanja){

    ifstream datoteka(putanja);
    if(datoteka == ucitana){
        while(dohvatiLiniju(datoteka, linija)){
            parsirajDatoteku();
            config = postaviNovePodatke(ključ, vrijednost);
        }
        datoteka.zatvori();
    }
    return config;
}
```

Sl. 3.20. Programsko rješenje parsiranja konfiguracijske datoteke

4. TESTIRANJE RJEŠENJA

U ovom poglavlju prikazani su rezultati testiranja predloženog rješenja. Kako bi se postojeće rješenje evaluiralo napravljena su sljedeća testiranja; Testirano je vrijeme izvođenja model paresra u usporedbi sa prethodnim rješenjem model parsera, memorijska zahtjevnost parsiranih podataka u odnosu na stari model parser te je provjerena dinamičnost model parsera. Ono što ovo rješenje donosi je veliko poboljšanje u trajanju izvođenja zadatka kojeg ovaj model parser obavlja i dinamičnost parsera. Model parser pridonosi cjelokupnom programu na način da se pri dostavi proizvoda korisniku očuva integritet izvornog koda jer program može biti isporučen kao zatvoren sustav bez isporuke izvornog koda.

4.1. Usporedba vremena izvođenja starog i novog model parsera

Vremensko testiranje brzine izvođenja model parsera je provedeno na dva načina. Prvi način je mjerenje vremena parsiranja bez spremanja u bazu podataka, a drugi način je sa spremanjem u bazu podataka. Ova dva mjerenja su uspoređena s vremenom koje je potrebno za parsiranje model parsera s upisom u bazu starog rješenja. Izvedeno je pet mjerenja koja su prikazana u tablici 4.1. Ovaj broj mjerenja nije vjerodostojna reprezentacija za dublju statističku analizu, ali ni sam problem nije prigodan za takvu analizu. Ovih pet mjerenja je sasvim dovoljan broj kako bi se zaključilo veliko unaprjeđenje u vremenu izvođenja. Naime, novom model parseru koji je pisan u C++ jeziku potrebno je da završi parsiranje podataka ispod pola sekunde, a uz spremanje u bazu ispod četiri sekunde, starom parseru koji je pisan u *Pythonu* potrebno je oko četiri minute. Novi parser ima bržu izvedbu u odnosu na stari parser šezdeset puta, a ovakav rezultat je uvelike zasluga korištenja drugog programskog jezika. Drastična razlika u izvedbi i prevođenju jezika opisana je u prethodnim poglavljima.

Tablica. 4.1. Izmjerenih vremena izvršavanja programa

	Bez spremanja u bazu[s]	**Sa spremanjem u bazu[s]**	**Model parser stari TEG[s]**
1.	0.24109	3.79068	247.274
2.	0.258215	3.56405	246.332
3.	0.236151	3.68134	248.102
4.	0.232177	3.63074	247.820
5.	0.227863	3.63946	247.702
Srednja vrijednost	0.2390992	3.661254	247.446

4.2. Usporedba memorijske zahtjevnosti starog i novog TEG-a

Podatci koje model parser treba parsirati iz *.arxml* datoteke mogu zauzimati do nekoliko GB (engl. *gigabyte*) memorije. Samim time što je novi model parser pisan u C++-u pridonosi brzini, ali kompaktnost spremljenih podataka u odnosu na stari TEG je značajna ušteda memorije, a samim time i vremena koje je potrebno za naknadnu manipulaciju sa spremljenim podatcima. U slučaju da se pojavi projekt s većim brojem podataka u modelu razlika u količini potrebne memorije za spremanje informacija iz modela stvara sve veću razliku između starog i novog model parsera. U tablici 4.2. prikazan je odnos zauzeća memorije između starog i novog TEG-a.

Tablica. 4.2. Memorijska zahtjevnost starog i novog TEG-a

	Model parser novi TEG	**Model parser stari TEG**
Memorijsko zauzeće	71.9KB	3.7MB
Broj linija u zapisanoj datoteci	1400	58000

Usporedbom spremljenih podataka može se zaključiti značajna ušteda memorije, brzina izvođenja prilikom pristupanja podatcima iz baze kako bi generatori obavili zadatak. Iz navedenih razloga ovo svojstvo model parsera pridonosi sveukupnoj brzini izvedbe i memorijske ovisnosti programa.

4.3. Provjera dinamičnosti na više projekata

Svaki proizvođač automobila ima svoje specifikacije kako je nešto definirano u vozilu. Iz ovoga proizlazi potreba za prilagodbom TEG-a. Dodatne specifikacije su pisane prema AUTOSAR standardu unutar *.arxml* datoteke koja definira model. Prema tome bilo je potrebno dodavati nove funkcionalnosti u prethodno rješenje model parsera kako bi se novo definirani podatci mogli parsirati. Novi model parser ostvaren je dinamično što znači da nema potrebu mijenjati kod prilikom primjene na novom projektu. Ovim pristupom dinamičnost parsera realizirana je korištenjem konfiguracijske datoteke što znači da se prilikom parsiranja nove datoteke mijenjaju samo identifikatori točki grananja koje usmjeravaju tok izvođenja programa. Pri tome se programski kod parsera ne mijenja jer on pristupa ovim identifikatorima preko ključeva koji su uvijek jedinstveni, a vrijednosti koje opisuju identifikatore se mijenjaju.

Eventualno poboljšanje koje se može ostvariti u daljnjem razvoju model parsera u smislu dinamičnosti bilo bi definiranje struktura koje bi opisivale neki paket. Naime ideja je da se svaki paket opiše, što on u sebi opisuje odnosno koje komponente sadrži. Na taj način bi se prvo prolazilo

kroz pakete i ako oni sadrže neke programske komponente u sebi program bi zaključio o kojem se paketu radi i krenuo ga parsirati prema istom principu koji je ostvaren u ovom rješenju. Pomoću ovog unaprjeđenja eliminirala bi se potreba za konfiguracijskom datotekom koja trenutno služi kao mehanizam pomoću kojeg model parser ostvaruje apstrakciju odnosno izdvajanje promjenjivog dijela parsera van samog programa.

Također treba provesti prikladne testove koji potvrđuju ispravnost parsiranih podataka spremljenih u bazu usporedbom s podatcima koje daje trenutno dostupno rješenje. Ovo testiranje je planirano napraviti, ali zbog velike količine podataka nije bilo prikladno vaditi sve podatke koji nemaju značaj za rad TEG-a, nego su se podatci važni za generatore i druge parsere izvadili prema njihovoj potrebi. Zbog same razlike između baze postojećeg rješenja i baze novog TEG-a nije moguće provjeriti točnost podataka odnosno to nema smisla raditi.

5. ZAKLJUČAK

U automobilskoj industriji vrlo je važno testirati program koji se primjenjuje u automobilima, te je sigurnost na prvome mjestu. U današnje vrijeme posebnu važnost u sigurnosti donosi program u kombinaciji sa sklopovljem jer puno brže procesuira senzorske informacije iz okoline od čovjeka. Prije implementacije programa u automobil potrebno ga je testirati. Jedno od važnijih testiranja je testiranje komunikacije između ugradbenih računala u automobilu. Kako bi se testirao sloj koji je posrednik između sklopovolja i programa koristi se TEG koji prikuplja informacije o ugradbenim računalima u automobilu i automatski generira testove koji simuliraju komunikaciju i na taj način se provjerava ispravnost sustava.

Postojeće rješenje generatora testnog okruženja imalo je brojne mane od kojih je neke od njih bilo potrebno riješiti u ovome radu. Jezik u kojem je pisano postojeće rješenje, zatim dinamičnost parsera, brzina izvedbe i očuvanje izvornog koda. U ovom diplomskom radu dizajnirao se i realizirao model parser koji je sastavni dio TEG-a, koristi se za prikupljanje podataka iz *.arxml* datoteka i kreiranje modela baze. Pri odabiru tehnologije bitno je uzeti jezik koji je niže razine od *Python*-a, koji pruža više kontrole programeru, ali i više odgovornosti, iz ovih razloga kao najprikladniji jezik pokazao se C++. Prilikom testiranja dobivenog rješenja zaključeno je kako je ovim pristupom ostvareno 60 puta brže rješenje od postojećeg. Nadalje ostvarena je dinamičnost parsera u odnosu na prijašnje rješenje pomoću uvođenja mehanizma apstrakcije kroz konfiguracijsku datoteku, te više nije potrebno dodavati tzv. zakrpe u programu na novom projektu. Posljednja stvar koju donosi C++ je očuvanje integriteta izvornog koda jer prilikom isporuke programskog rješenja kupcu nije potrebno dostaviti izvorni kod već samo zatvoren sustav koji prima ulazne podatke i daje željeni izlaz.

LITERATURA

[1] Renesas, [online], Renesas Electronics Corporation, dostupno na: https://www.renesas.com/sg/en/application/automotive/autosar/autosar-layeredarchitecture, [26.7.2021.]

[2] Rushter, [online], How pickle works in Python, dostupno na: https://rushter.com/blog/pickle-serialization-internals/ , [27.7.2021.]

[3]Andrija Mihalj, Generator softverskog koda namijenjenog za testiranje ADAS sustava, Diplomski rad, Osijek, 2019. , [27.7.2021.]

[4] Python object serialization, [online], dostupno na: https://docs.python.org/3/library/pickle.html, [27.7.2021.]

[5] Microsoft Visual Studio, [online], dostupno na: https://en.wikipedia.org/wiki/Microsoft_Visual_Studio#Architecture, [28.7.2021.]

[6] Inside the Visual Studio SDK, Microsoft, [online], dostupno na: https://docs.microsoft.com/en-us/visualstudio/extensibility/internals/inside-the-visual-studio-sdk?view=vs-2019#extensibility-architecture, [28.7.2021.]

[7] Boris Motik, Julijan Šribar, Demistificirani C++, Zagreb, 1997. , [29.7.2021.]

[8] The Features of C++ as a Language, [online], dostupno na: https://www.cplusplus.com/info/description/, [30.7.2021.]

[9] XML Document Object Model (DOM), Microsoft, [online], dostupno na: https://docs.microsoft.com/en-us/dotnet/standard/data/xml/xml-document-object-model-dom, [2.8.2021.]

[10] TinyXML-2, [online], dostupno na: http://leethomason.github.io/tinyxml2/ [3.8.2021.]

SAŽETAK

Automobili imaju sve više elektroničkih komponenti i složenih sustava za pomoć u vožnji. Svim ovim kompleksnim sustavima i mnogim drugim koje posjeduju današnji automobili upravljaju ugradbeni računalni sustavi visoke složenosti. Ova računala imaju velik broj komunikacijskih kanala koje je potrebno testirati, a zbog količine potrebnih testova pogodno ih je automatski generirati. U ovom diplomskom radu analizirano je postojeće rješenje generatora testnog okruženja i njegove mane, te je predloženo programsko rješenje model parsera u sklopu generatora testnog okruženja. Model parser dohvaća podatke iz *.arxml* datoteka i stvara bazu podataka. Bilo je potrebno ostvariti dinamičnost parsera i poboljšati performanse u odnosu na već postojeće rješenje. Model parser pisan je u programskom jeziku C++. Nakon ispunjenih zahtjeva koji su definirani za ovaj model parser provedeno je testiranje, te je utvrđeno vremensko poboljšanje od šezdeset puta u odnosu na postojeći model parser.

Ključne riječi: ugradbeni računalni sustavi, generator testnog okruženja, model parser, C++, automotiv

Developing parser for AutoSAR model system within test environment generator

ABSTRACT

Commented [MH1]: Sažetak na engleskom ide na zasebnu stranicu, Naslov na engleskom ide ispred Abstracta

Cars have more and more electronical components and complex systems for driver assistance in driving. These complex systems and many others, which today's cars possess, are driven by highly complex embedded computer systems. These computers have large amounts of communication channels which need to be tested, and because of the quantity of these tests, it is suitable to generate them automatically. In this master's thesis an existing solution of the test environment generator and its flaws are analyzed and a software solution for model parser that is a part of the test environment generator is developed. Model parser gets data from *.arxml* file and creates a data base. It was required to create a dynamic parser that outperforms the existing solution. Model parser was written in C++ programming language. When requirements that are defined for this model parser were met, the solution was tested. Through testing, it was shown that the new model parser is sixty times faster than the previous one.

Key words: embedded computer systems, test environment generator, model parser, C++, automotive

ŽIVOTOPIS

Filip Širac rođen je 21.12.1996. u Osijeku. Osnovnu školu Braće Radića iz Pakraca završava 2011. godine. Nakon završetka osnovne škole upisuje Tehničku školu u Daruvaru smjer Računalstvo. Tijekom srednjoškolskog obrazovanja sudjeluje na projektu „Balon Stellar - stratosfera 30 km“ u sklopu Google Lunar Xprize, te na smotri radova na Fakultetu Elektrotehnike i Računarstva u Zagrebu sa završnim radom „Samobalansirani robot na dva kotača“. Nakon završetka srednje škole ostvaruje direktno pravo upisa na Elektrotehnički fakultet u Osijeku temeljem uspjeha ostvarenog tijekom srednjoškolskog obrazovanja. Trenutno studira Računarstvo na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek.

U Osijeku, rujan 2021.

Filip Širac

_____

Commented [MH2]: Osnovnu školu Braće Radića iz Pakraca završava XXXX godine

Commented [MH3]: Tijekom srednjoškolskog školovanja sudjeluje…..

Commented [MH4]: Koji fakultet?