Alat za grafički prikaz kompleksnosti izvršnog koda

Udovičić, Ana

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

Sveučilišni studij

ALAT ZA GRAFIČKI PRIKAZ KOMPLEKSNOSTI IZVRŠNOG KÔDA

Diplomski rad

Ana Udovičić

Osijek, 2020.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 19.09.2020.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime studenta:	Ana Udovičić
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1022R, 24.09.2019.
OIB studenta:	11053173644
Mentor:	Izv.prof.dr.sc. Ratko Grbić
Sumentor:	
Sumentor iz tvrtke:	Nemanja Nikić
Predsjednik Povjerenstva:	Izv. prof. dr. sc. Mario Vranješ
Član Povjerenstva 1:	Izv.prof.dr.sc. Ratko Grbić
Član Povjerenstva 2:	Izv.prof.dr.sc. Tomislav Matić
Naslov diplomskog rada:	Alat za grafički prikaz kompleksnosti izvršnog koda
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	Razvijena programska podrška u automotive projektima često je vrlo složena. Kako bi se složenost dobivene programske podrške po mogućnosti reducirala, potrebni su alati koji omogućuju analizu kompleksnosti programskog rješenja, pri čemu se kao mjere kompleknosti najčešće uzimaju modularnost i definirana sučelja. U okviru ovog diplomskog rada potrebno je analizirati specificirana sučelja u okviru dokumentacije i na osnovu njih napraviti graf toka podataka između komponenata programske podrške. Rezultat analize treba biti grafički prikaz cijelog sustava s mogućnošću jednostavne navigacije po nivoima (npr. ako se ECU (engl. Electronic Control Unit))
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: 2 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	19.09.2020.

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, 29.09.2020.

Ime i prezime studenta:	Ana Udovičić
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1022R, 24.09.2019.
Turnitin podudaranje [%]:	2

Ovom izjavom izjavljujem da je rad pod nazivom**: Alat za grafički prikaz kompleksnosti izvršnog koda**

izrađen pod vodstvom mentora Izv.prof.dr.sc. Ratko Grbić

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

Automobilska industrija jedna je od trenutno najbrže rastućih industrija u svijetu. Razvojem automobilske industrije postavljaju se sve veći zahtjevi na programsku podršku razvijenu za automobile i implementirane funkcionalnosti unutar te programske podrške. U modernim vozilima prisutan je veći broj elektroničkih upravljačkih uređaja (engl. ***E**lectronic **C**ontrol **U**nit* – ECU) koji upravljaju radom različitih podsustava unutar vozila. Ovo podrazumijeva i veliki broj programa napisanih za svaki pojedini podsustav. Arhitektura ovakve programske podrške izrađuje se modularno, odnosno ona se gradi od većeg broja zasebnih modula, kako bi se mogla nadograđivati i mijenjati. Programski moduli su dijelovi programskog kôda koji se mogu samostalno prevoditi uz pomoć odgovarajućeg prevoditelja [1]. Jedan program može se sastojati od više programskih modula, a upravo je to najčešći slučaj u praksi, te se time postiže velika kompleksnost programskog kôda. Svaki modul sadrži određeni broj sučelja. Sučelje je skup funkcija koje omogućuju komunikaciju i interakciju s vanjskim jedinicama sustava, poput senzora i izvršnih članova. Arhitektura ovakvih programa najčešće je u obliku hijerarhije, pri čemu postoji jedan glavni modul, koji sadržava druge module koji su također sastavljeni od drugih modula. Broj ovakvih razina može biti vrlo velik te je pregledavanje ovakvog kôda vrlo zamršeno, a uočavanje i otklanjanje pogrešaka u kôdu otežano.

Iz samih specifikacija programskog kôda ili iz generiranih datoteka čovjek teško uočava važne značajke o specifikaciji sustava i njegovoj građi. Pregled kôda u vizualnom obliku, najčešće grafom ili dijagramom, omogućava čovjeku lakši pregled ovih značajki, a time i uočavanje i otklanjanje eventualnih pogrešaka. Nadalje, kompleksnost programske podrške ponekad je nepotrebno velika te se reduciranjem njezine veličine postiže ušteda vremena i novca. Kako bi korisnicima bilo omogućeno bolje razumijevanje programa i njegove arhitekture razvijaju se alati koju omogućuju vizualizaciju i analizu kompleksnosti programskog rješenja. Vizualizacija programskog kôda prvenstveno je namijenjena programerima koji razvijaju programski kôd i arhitektima koji osmišljavaju arhitekturu programa na velikim projektima.

Zadaća grafičkog prikaza je korisniku omogućiti efikasnu analizu arhitekture programskog kôda, pridonijeti uštedi vremena i novca, povećati kvalitetu i produktivnost programera te im olakšati upravljanje kompleksnošću kôda i ranom otkrivanju postojećih i potencijalnih pogrešaka.

Kako bi se mogao razviti kvalitetan alat koji bi omogućio detaljnu analizu, važno je izraditi kvalitetnu dokumentaciju nakon izrade programskog kôda. Dokumentacija mora sadržavati informacije o svim implementiranim modulima i sučeljima koje program koristi te mora biti

napisana u standardnom formatu unutar tvrtke. Ako je dokumentaciju izradio arhitekt prije implementacije kôda, programer se prilikom izrade programskog kôda mora pridržavati naziva i podataka napisanih u dokumentaciji. Nedovršena, netočna ili nedostajuća dokumentacija značajno otežava izradu ovakvog alata, jer se on temelji upravo na njoj i ona je ishodište i izvor podataka za prikaz na grafičkom sučelju.

U ovome diplomskom radu analizirana su specificirana sučelja i moduli u okviru dokumentacije izrađene u *JSON* formatu i na osnovu njih je izrađen graf toka podataka između modula programske podrške. Kao rezultat analize dan je grafički prikaz cijelog sustava s mogućnošću navigacije po nivoima. Zatim je izrađena analiza generiranog kôda (*map* datoteke). Analiza *map* datoteke provodi se ekstrahiranjem naziva pojedinih programskih modula i sučelja definiranih unutar tih modula. Grafička vizualizacija generiranog kôda provodi se na sličan način kao i prethodna. Na kraju se korisniku omogućava opcija usporedbe prethodna dva grafa, koja iscrtava usporedni graf. Usporedni graf nastaje na osnovu spajanja prethodna dva te tako omogućuje jasan prikaz razlika i odstupanja. Upravo ovaj usporedni graf omogućuje lakše uočavanje pogrešaka nastalih prilikom implementacije kôda.

Diplomski rad strukturiran je na način da se u drugom poglavlju prikazuju postojeća rješenja i aktualne ideje u području analize i vizualizacije kompleksnosti kôda. Treće poglavlje detaljno prikazuje i opisuje izradu alata za vizualizaciju kompleksnosti kôda. U četvrtom poglavlju demonstriran je način rada predloženog rješenja s pripadajućom raspravom. Na kraju je dan zaključak rada sa smjernicama za daljnja poboljšanja izrađenog alata.

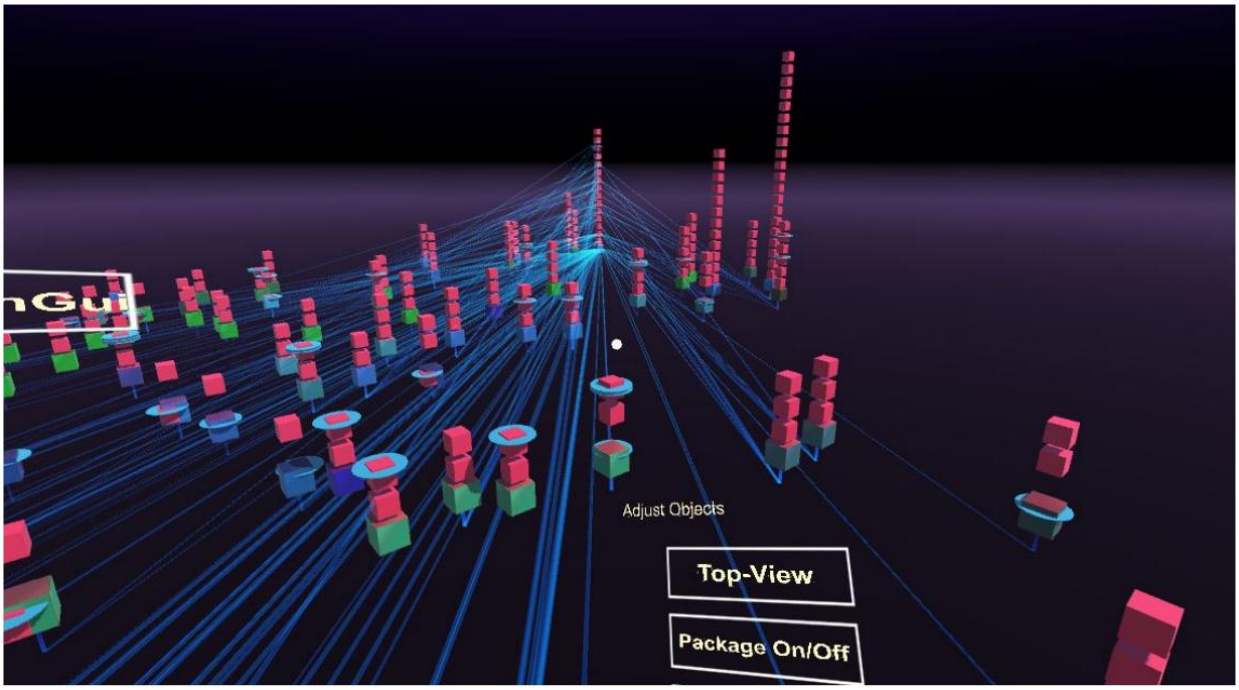
2. PREGLED POSTOJEĆIH RJEŠENJA ZA ANALIZU I VIZUALIZACIJU KOMPLEKSNOSTI IZVRŠNOG KÔDA

Problemu analiziranja programskog kôda i izradi grafičkih prikaza arhitekture kôda u praksi se pristupa sa različitih stajališta. Programi i alati koji se razvijaju u ovu svrhu su najčešće specifične namjene, prilagođeni određenoj vrsti programskog kôda i programskom jeziku u kojem je taj program napisan. Razlog tome je što se za svaku vrstu analize odabire što se želi analizirati i prikazati. Tako se primjerice, pregled performansi sustava najčešće prikazuje dvodimenzionalnim grafovima ili stupčastim dijagramima, dok se arhitektura programskog rješenja prikazuje najčešće usmjerenim grafovima ili trodimenzionalnim grafovima. Iako se vizualizacija programskog kôda koristi već godinama, u novije vrijeme sve su popularniji trodimenzionalni prikazi te korištenje virtualne stvarnosti jer dodavanjem nove dimenzije omogućuju detaljniji prikaz. Izrada ovakvog grafičkog prikaza podrazumijeva analiziranje i dekompoziciju programskog kôda, odnosno obrnuti inženjering (engl. *reverse engineering*) . Pri tome se kao ulaz koristi prevedeni programski kôd koji se nastoji prikazati u obliku koji će čovjeku biti lako razumljiv.

Radovi prikazani u ovome poglavlju opisuju grafičke alate za analizu i vizualizaciju kôda razvijene za različite namjene i u različitim industrijama. Neovisno o tome za koju su namjenu izrađeni, naglasak je u svima stavljen na korisniku što pristupačnije sučelje kojim se može jednostavno kretati.

Jedno od rješenja problema reduciranja kompleksnosti prikaza je korištenje virtualne stvarnosti. U radu [2] predlaže se trodimenzionalni grafički prikaz i virtualna stvarnost za analizu kompleksnosti arhitekture programskih rješenja. Navodi se kako trodimenzionalni prikaz, za razliku od dvodimenzionalnog, pruža bolji uvid u detalje. Grafičko sučelje, prikazano slikom **2.1.** prikazuje programske pakete, module i njihove međusobne veze.

Programski paketi i moduli se prikazuju kvadrima različitih boja i veličina. Programski paketi i moduli pripadaju određenim snopovima, a svaki snop ima određenu visinu u ovisnosti o broju paketa i modula koje sadržava. Paketi mogu sadržavati servise, a u tom slučaju servisi se prikazuju kao elipse oko kvadra koji ih sadržava. Klase su predstavljene cilindrima, a prikazuju se samo za odabrani paket u spiralnoj formi oko tog paketa.

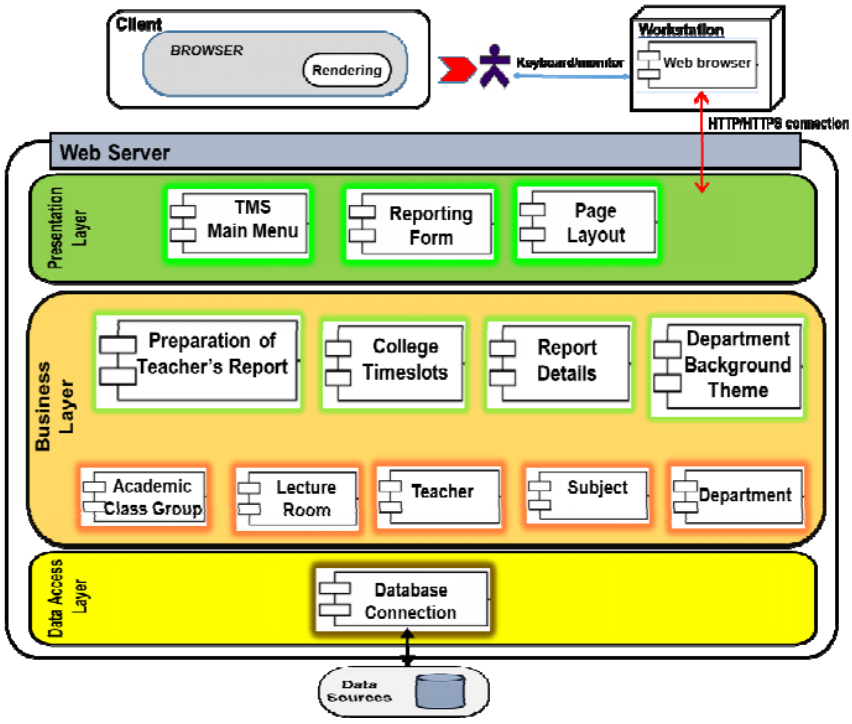


Sl. 2.1. *Izgled grafičkog sučelja realiziranog u radu [8]*.

Veze između određenih paketa prikazane su ravnim linijama, pri čemu crvena boja prikazuje uvezene (engl. *imported*) veze, a plava izvezene (engl. *exported*) veze. Cijeli grafički prikaz smješten je u trodimenzionalni koordinatni sustav, a programski paketi i moduli se grupiraju (eng. *cluster*) po sličnosti. Korisničko sučelje je interaktivno, a omogućuje osnovne funkcionalnosti i dodatne. U osnovne funkcionalnosti pripadaju odabir prikaza, kretanje između snopova te bočni prikaz ili pogled odozgo. Dodatne funkcionalnosti su kretanje po sceni, rotiranje geometrijskih objekata na sceni, uključivanje i isključivanje prikazivanja veza te uključivanje i isključivanje opcije virtualne stvarnosti. Kako se ovom prikazu pristupa preko naočala za virtualnu stvarnost, za korištenje funkcionalnosti koristi se tipka na naočalama te tipke unutar grafičkog prikaza. Za izradu ovog alata, korišten je *Unity*, skup alata za izradu računalnih igara. Arhitektura koja je prikazana bazirana je na OSG-i (engl. ***O**pen **S**ervices **G**ateway*) aplikacijama. Vizualizacija je testirana na *Oculus Rift* naočalama za virtualnu stvarnost i *Google Cardboard* platformom. To je platforma koju je razvio *Google* koja koristi pametni telefon za prikaz virtualne stvarnosti. Ovakav prikaz ima veliku prednost u situacijama kada je korisniku potreban uvid u veličinu programskog rješenja i količinu memorije koju zauzima to programsko rješenje ili neki njegovi pojedini dijelovi. Iako je vrlo pregledan, korištenje virtualne stvarnosti može biti komplicirano i nepraktično pri korištenju ovakvih alata na dnevnoj bazi.

Obrnuti inženjering je način analize sustava koji se koristi za prepoznavanje njegovih komponenata i njihovih ovisnosti te za izdvajanje i kreiranje apstrakcije sustava i informacija o dizajnu [3]. U radu [4] se naglašava važnost obrnutog inženjeringa prilikom analize arhitekture programskog kôda i njegove optimizacije. U ovome kontekstu, postupcima obrnutog inženjeringa se nastoji od strojnog kôda dobiti izvorni kôd u kojem je program napisan. Na ovaj način mogu se optimizirati dijelovi kôda i otkloniti pogreške.

Rad [4] se bavi djelomičnom vizualizacijom programskog kôda. Prikazivanje ukupnog programskog kôda je teško izvesti jer je pri tome količina informacija prevelika za kvalitetan prikaz. Kako bi se izbjegli problemi nedostajućih informacija koji bi rezultirali nezadovoljavajućim prikazom, koristi se djelomična vizualizacija, koja prikazuje dio kôda za koji su poznate sve potrebne informacije. Nadalje, često se dokumentacija programskog kôda, koja je polazište u izradi svih alata ove i slične namjene, razlikuje od razvijenog programskog kôda ili je uopće nema. Za izbjegavanje ovakvog problema, rad [4] predlaže prikupljanje dokumentacije ne samo od programera koji je izradio programski kôd, već i od arhitekta koji je zamislio programski kôd te osobe zadužene za održavanje. Sama vizualizacija prikazana u ovome radu sastoji se od dva koraka. Prvi korak podrazumijeva preslikavanje dijelova kôda dobivenih obrnutim inženjeringom u arhitektonske komponente koje se prikazuju na sceni. U drugom koraku se izrađuje grafički prikaz arhitekture kôda korištenjem slojevitog modela prikazanog slikom **2.2.**

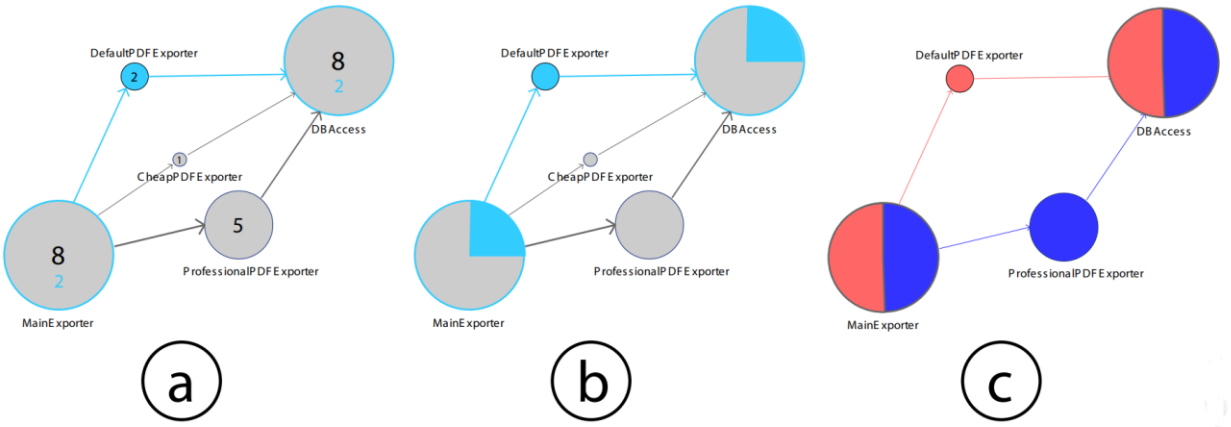


Sl. 2.2. *Slojeviti model predložen u radu [4] za prikaz arhitekture programske podrške.*

Metode izvlačenja podataka iz generiranog kôda često nisu dovoljno automatizirane ili su izrađene za programe pisane u objektno orijentiranoj notaciji. Rad [5] predlaže izradu arhitekture kao izvršne datoteke, koja bi predstavljala dinamičku simulaciju arhitekture programa. Za izradu ovakve datoteke potrebno je izvorni kôd transformirati u određeni zapis koji bi se mogao koristiti i u drugim implementacijama kôda. Za izradu ovakvog alata odabrana je metoda statičkog obrnutog inženjeringa. Statička metoda analizira sve korake programskog kôda i njihove međusobne povezanosti i putanje. Time omogućuje maksimalni opoziv (sposobnost pronalaska svih relevantnih primjera u zadanom skupu podataka) i minimalne gubitke informacija do kojih bi moglo doći ukoliko se ne uzmu u obzir svi koraci i putanje.

Metodologija opisana u radu [5] iz izvornog kôda kreira apstraktno sintaksno stablo (engl. *abstract syntax tree*). Apstraktno sintaksno stablo predstavlja softverski artefakt nastao korištenjem struktura podataka koje predstavljaju tipove konstruktora, njihove kompozicijske odnose prema drugim konstrukcijama te skup izravnih i izvedenih svojstava povezanih sa svakom jezičnom konstrukcijom [6]. Ova metoda testirana je na različitim *JAVA* projektima. Testiran je samo prvi krug obrnutog inženjeringa, a drugi krug, koji uspoređuje dobivene rezultate s originalnim kôdom, se tek predlaže kao jedno od mogućih dodatnih funkcionalnosti. Korišten je *Jabref* alat za upravljanje bazama podataka za jedan od testova, koji je rezultirao pogreškama do kojih je došlo zbog zanemarivanja alokacije resursa.

Većina vizualizacija i analiza kreiranih za potrebe optimizacije kôda i ispravljanje grešaka dolazi u tekstualnoj formi, neprilagođenoj u potpunosti čovjeku i njegovu načinu opažanja. Stoga se u radu [7] istražuju nove tehnike i strategije za vizualizaciju programskog kôda. Kao osnovne metode vizualizacije ističu se skice raspršivanja (engl. *scatter plot*), stupčasti dijagrami (engl. *bar chart*), dijagrami paralelnih koordinata (engl. *parallel coordinate plot*) te radarske karte (engl. *radar chart*). Predlažu se poboljšanja ovakvih metoda, u smislu dodavanja novih funkcionalnosti. Tako se primjerice, za samu vizualizaciju arhitekture programskog rješenja, predlažu usmjereni grafovi u kojima radijus čvora govori o memorijskoj veličini tog programskog modula kojeg taj čvor predstavlja, kao što je prikazano na slici 2.3.



Sl. 2.3. *Varijacije grafikona koji prikazuju komponente i ovisnosti o veličini te (a) eksplicitno navedenu veličinu, (b) implicitna reprezentacija veličine pomoću strukturnog kruga (engl. pie chart) i (c) preslikavanje pomoću boja [7].*

Za poboljšanje grafičkih prikaza općenito, predlaže se dodavanje veličina povećavanjem ili smanjivanjem geometrijskih likova koji prikazuju određeni dio kôda, korištenje boja za prikaz odnosa između dijelova programskog kôda, isticanjem važnih dijelova ili razlika povećavanjem ili bojanjem te korištenjem interaktivnosti. Kao kvalitetnu interaktivnost, rad ističe mogućnost navigacije, odnosno kretanja po grafovima, povećavanja i smanjivanja prikaza te odabira dijela grafa koji se želi prikazati. Informacija o veličini nekog dijela programskog kôda može se predstaviti veličinom geometrijskog lika.

Kao rezultat ovog istraživanja izrađen je prototip vizualizacije, koristeći *JavaScript* i *HTML*. Prototip se sastoji od vizualizacijskog servera za *back-end*, a grafičko sučelje za korisnika se prikazuje u internetskom pregledniku. Testiranje prototipa je provedeno na način da je programsko rješenje predloženo u radu dano različitim programerima na korištenje i uvid i prikupljena su njihova mišljenja. Testiranje je pokazalo kako je potrebna dorada na pregledu arhitekture programskog kôda jer je za potrebe optimizacije i ispravljanja pogrešaka neophodan iznimno detaljan prikaz.

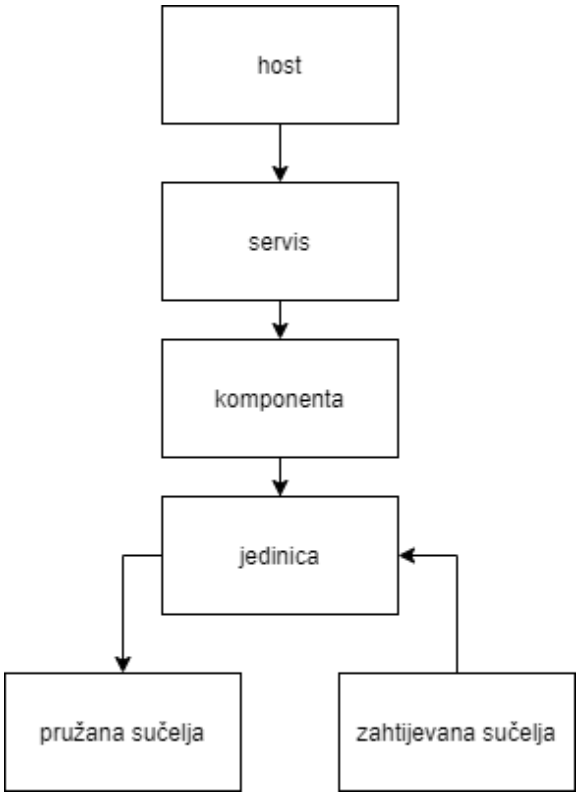
U radu [8] predložena je ideja o sinkrono kolaborativnom modeliranje softverskih sustava pomoću trodimenzionalnog *UML* (engl. *the **U**nified **M**odeling **L**anguage*) dijagrama u stvarnom vremenu. Ovakav prikaz trebao bi uštedjeti vrijeme, poboljšati prikaz kompleksnih sustava i omogućiti korisniku pregled prethodnih promjena. Osnovna ideja sinkrono kolaborativnog modeliranja je da jednom grafičkom prikazu istovremeno može pristupiti više korisnika u

stvarnom vremenu. Na taj način svaki pojedinac može vidjeti sve promjene koje se događaju u tom trenutku. Realizirano je rješenje u kojem su kreirani *UML* dijagrami dvodimenzionalni, ali su smješteni u trodimenzionalni prostor koji olakšava navigaciju i pregled prethodnih rješenja. Kako je omogućen rad više korisnika u isto vrijeme, dodana je mogućnost prijave korisnika i njihove međusobne komunikacije. Kolaborativni pristup je inovativan, ali nije testirano koliko je uspješan i koliko se na ovaj način povećava efikasnost programera.

Trenutna programska rješenja predlažu različite pristupe i stavljaju naglasak na različite aspekte programskog kôda koji analiziraju. Trenutno postoje brojni načini i alati koji se koriste za vizualizaciju programskog kôda, no svi su i dalje podložni optimizaciji i dodavanju novih funkcionalnosti. Razlog tome je što su programska rješenja sve kompleksnija i naprednija te je iste potrebno pratiti i omogućiti što detaljniji, ali istovremeno i jednostavniji, prikaz cjelokupnog sustava. Ovo je područje koje se kontinuirano razvija i u koji se kontinuirano unose nove tehnologije koje omogućuju napredniji i kvalitetniji prikaz.

3. PREDLOŽENI ALAT ZA VIZUALIZACIJU KOMPLEKSNOSTI IZVRŠNOG KÔDA

Alat za vizualizaciju kompleksnosti kôda predložen u okviru ovog rada temelji se na strukturi programske podrške prikazane slikom **3.1**.



Sl. 3.1. *Arhitektura programskog rješenja koje se analizira u radu.*

Host predstavlja središnji program na kojem se povezuju svi njemu podređeni programi. U arhitekturi programskog rješenja izrađenog u ovome radu nalaze se dva *host*-a. Servisi su programi organizirani u grupe koje se izvode unutar različitih instanci na nadređenom *host*-u. Svaki servis sadrži više komponenti. Komponenta je programski modul koji se može višekratno koristiti i kombinirati s drugim komponentama. Svaka komponenta ima određeni broj jedinica koje sučeljima komuniciraju s vanjskim jedinicama, poput senzora i izvršnih članova. Sučelja mogu biti pružana i zahtijevana. Pružana sučelja su sučelja implementirana u jedinici i prikazuju informacije o toj jedinici. Zahtijevana sučelja su sučelja koja su jedinici nužna za ispravan rad, a implementirana su unutar druge jedinice. Na taj način su jedinice, komponente i servisi međusobno

povezani. Programsko rješenje može imati više servisa, komponenata, jedinica i sučelja koja se izvode na nekom *host*-u.

Alat kao ulazni skup podataka koristi prikupljene i obrađene podatke iz dokumentacije programskog kôda i podatke iz same izvršne datoteke. Tako pripremljen skup se zatim prikazuje grafički na zaslonu uzimajući u obzir jednostavnost i preglednost prikaza, koja će korisniku olakšati pregled i analizu. Problem reduciranja kompleksnih grafičkih prikaza rješava se mogućnošću navigacije po grafu pritiskom miša ili kao zasebna funkcionalnost. Korištenje alata omogućava pregled kompleksnosti kôda te uvid u potencijalne pogreške koje su naznačene na grafu.

Alat za grafički prikaz i analizu kompleksnosti kôda objašnjen u ovom radu koristi podatke spremljene u *JSON* (engl. ***J**ava**S**cript **O**bject **N**otation*) formatu, analizira ih te prikazuje grafički na zaslonu u interaktivnom obliku. Ovakav prikaz korisniku omogućuje samo grafički prikaz podataka definiranih u dokumentaciji. Drugi graf nastaje nakon implementacije programskog rješenje na temelju podataka iz generiranih datoteka. U ovome radu koriste se generirane *map* datoteke iz kojih se izvlače znakovi i nazivi. Ponavlja se sličan postupak kao i u prethodnom koraku, podaci se obrađuju i prikazuju na zaslonu. Ova dva grafa usporedno se prikazuju jedan pokraj drugoga. Kako bi bilo lakše uočiti razlike između njih, oni se mogu usporediti, pri čemu se generira treći graf, graf usporedbe. Na grafu usporedbe istaknute su razlike između prethodna dva. Detaljan opis razvoja ovoga alata objašnjen je i prikazan u nastavku.

3.1. Korišteni alati i tehnologije

3.1.1. JSON

JSON je otvoreni standardni tip datoteke koji omogućava razmjenu podataka u čitljivom obliku. Neovisan je o programskom jeziku koji se koristi. *JSON* datoteka je tekstualna, a pogodna je za rad sa *Javascript* jezikom jer se lako čita iz i sprema u *Javascript* objekte. Budući da se podaci između severa i internetskog preglednika mogu prenositi samo u tekstualnom obliku, *JSON* je najlakši i najbrži način, bez potrebe za dodatnim transformacijama ili parsiranjem.

3.1.2. HTML i CSS

HTML (engl. ***H**yperText **M**arkup **L**anguage*) standardni je prezentacijski jezik za izradu internetskih stranica. On opisuje strukturu sadržaja koji je potrebno prikazati na internetskoj stranici, odnosno govori internetskom pregledniku kako prikazati sadržaj. Uz *JavaScript* i *CSS*

predstavlja temelj izrade internetskih stranica. *CSS* (engl. ***C**ascading **S**tyle **S**heets*) stilski je jezik koji dodatno opisuje izgled sadržaja koji će se prikazati na web stranici. Korištenjem *CSS*-a određuju se fontovi, veličine elemenata, boje, razmaci i slično.

3.1.3. JavaScript

JavaScript je programski jezik korišten za razvoj aplikacija internetskih aplikacija. Programi napisani u *JavaScript* programskom jeziku nazivaju se skripte i moguće ih je pokrenuti iz *HTML* kôda. Njihovo pokretanje je automatsko pri učitavanju internetske stranice te nije potrebno nikakvo dodatno prevođenje (engl. *compiling*) ili obrađivanje skripti. Važna funkcionalnost *JavaScript*-a u kontekstu ovoga rada je da omogućuje dinamične događaje, interaktivnost između sadržaja i korisnika te prilagodbu grafičkog izgleda.

3.1.4. D3.js [9]

D3.js (engl. *3D - **D**ata-**D**riven **D**ocuments*) je *JavaScript* biblioteka koja omogućava izradu interaktivnih vizualizacija u internetskim aplikacijama. Ova biblioteka omogućava manipuliranje datotekama sa podacima, kreiranje vizualizacija, animacija i interaktivnih događaja na objektima. Biblioteka sadrži unaprijed izgrađene funkcije (engl. *pre-built functions*) za kreiranje objekata, oblikovanje, dodavanje animacija i prijelaza, detekciju događaja na prijelaz ili odabir mišem i slično. *D3.js* biblioteka je pogodna za rad s podacima u *JSON* formatu, koji su i korišteni u ovom radu. U radu je korištena *D3.js v3* verzija biblioteke za učitavanje podataka, kreiranje grafova animacija i interaktivnih događaja.

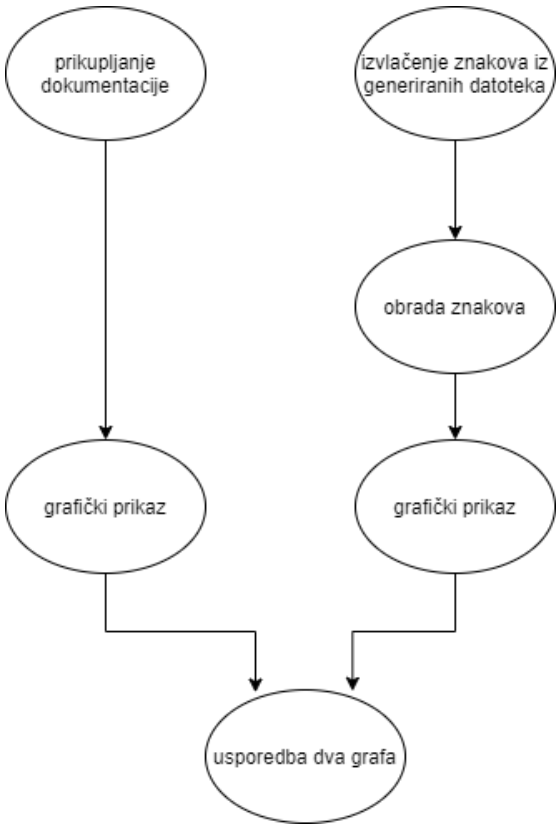
3.1.5. *Python* i dodatne biblioteke

Python je programski jezik opće namjene koji se koristi za pisanje različitih programa. Čitanje datoteka i pisanje datoteka je jedna od funkcionalnosti koje se lako implementiraju u *Pythonu* te je zato korišten u izradi ovog alata.

Uz *Python* korištene su i softverska biblioteke (engl. *software library*) *pandas* [10] i modul *difflib* [11] napisane za Python. Bibiloteka *pandas* se koristi za manipulaciju i analizu podataka, a pruža strukture podataka i operacije za manipulaciju tih podataka. Modul *difflib* sadrži alate za računanje različitosti i rad s različitostima između sekvenci, najčešće tekstualnih. U ovom radu se koristi za usporedbu teksta.

3.2. Koncept predloženog alata za vizualizaciju kompleksnosti izvršnog kôda

Alat za grafički prikaz kompleksnosti kôda sastoji se od nekoliko koraka, koji su prikazani blok dijagramom na slici **3.2.**



Sl. 3.2. *Konceptualno rješenje alata za vizualizaciju kompleksnosti programskog kôda.*

Najprije se prikuplja dokumentacija koju svaki programer piše nakon izrade programske podrške. Dokumentacija može biti pisana u različitim oblicima i formatima, a u ovom slučaju izrađena je u obliku *JSON* datoteke. Zatim je iz takve dokumentacije potrebno izvući relevantne podatke. Naposljetku, relevantni podaci se iscrtavanju grafički. Iz generiranih datoteka se izvlače znakovi koji nose relevantne podatke. Generirane datoteke mogu biti u *elf* (engl. ***E**xecutable and **L**inkable **F**ormat*) formatu, objektne datoteke ili *map* datoteke. U ovome radu korištene su *map* datoteke. Iz njih se izvlače znakovi i prevode u čovjeku razumljiv format te se grafički prikazuju. Zadnji korak ovog algoritma je usporedba dva grafa. Ovaj alat treba u konačnici korisniku na zaslonu prikazati tri grafa – jedan graf je dobiven iz dokumentacije, drugi iz generiranih datoteka

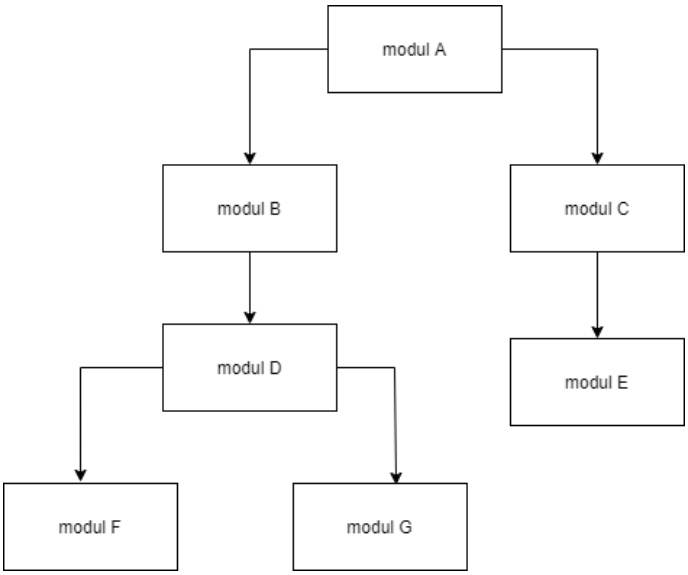
i treći nastaje kao usporedba prethodna dva. Prvi graf omogućuje uvid u kompleksnost programskog rješenja i predstavlja kako programsko rješenje zaista treba izgledati i od kojih se programskih modula treba sastojati. Drugi graf nastaje iz generiranih datoteka te pruža uvid u programske module koje je programer izradio tijekom pisanja kôda te daje uvid u njegov rad. Treći graf je usporedni graf i on prikazuje odstupanje programskog rješenja od dokumentacije te na taj način pruža uvid u pogreške nastale prilikom izgradnje programskog kôda.

3.3. Prikupljanje dokumentacije programskog rješenja

Prvi korak podrazumijeva izradu dokumentacije programskog rješenja. Dokumentacija programskog rješenja može se pisati u različitim formatima i različitim programskim alatima. Važno je definirati način na koji se dokumentacija piše i standardizirati taj način jer je dokumentacija polazišna točka za izradu ovakvoga alata. Prilikom izrade diplomskog rada, originalna dokumentacija nije bila ispravno napisana, odnosno nije bila u potpunosti standardizirana. Time je bilo onemogućeno automatizirano čitanje i raščlanjivanje podataka iz dokumentacije i rezultiralo je nepotpunom informiranošću o potrebnim programskim modulima i sučeljima. O ovome problemu pisano je i u radu [4] gdje je predloženo rješenje za djelomičan prikaz koji bi prikazivao samo dijelove za koje je dokumentacija ispravno izrađena. Pri izradi alata predloženog u okviru ovog rada korišten je drugačiji pristup u kojem se dokumentacija izradila u standardiziranom ugniježdenom *JSON* formatu, koji će biti prilagođen prikazu na internetskoj aplikaciji.

Objekti u *JSON* datotekama uvijek sadržavaju par vrijednosti, ključ (engl. *key*) i vrijednost (engl. *value*). Ključ uvijek mora biti znakovni niz (engl. *string*), dok vrijednost može biti broj, znakovni niz, objekt, niz (engl. *array*), *Boolov* tip ili *null*. Ako je vrijednost nekog objekta drugi objekt, tada se takva struktura naziva ugniježdenom (engl. *nested*) i korisna je za prikaz hijerarhije.

Ugniježdena struktura podataka omogućava prikaz čvorova roditelja i potomaka. Gornji čvor predstavlja korijenski čvor i on nema čvora roditelja. Svi ostali čvorovi imaju točno jednog roditelja čvora, koji im je nadređen. Svaki čvor može imati neograničeni broj čvorova potomaka. Glavni modul, *host*, će tako biti korijenski čvor, a kao njegovi čvorovi potomci bit će prikazani svi programski moduli koje on poziva, odnosno koristi. Primjer ovakve strukture dan je slikama **3.3.** i **3.4**, na kojima je prikazana dijagramom hijerarhija modula te zatim pripadno rješenje u *JSON* formatu.



Sl. 3.3. *Primjer dijagram hijerarhije programskog rješenja.*

Modul A predstavlja glavni modul - korijenski, koji ima dva čvora djeteta – modul B i C. Modul B koristi modul D, koji se nadalje grana na module F i G. Modul C koristi modul E. Ovakva hijerarhija prikazuje se ugniježdenim *JSON* formatom kao na slici **3.4**.

Linija ***Kôd***

```
 1:   [
 2:   {
 3:   "key": "Modul A",
 4:   "value": [
 5:   {
 6:   "key": "Modul B",
 7:   "value": [
 8:   {
 9:   "key": "Modul D",
10:   "value": [
11:   {
12:   "key": "Modul F",
13:   "value": ""
14:   },
15:   {
16:   "key": "Modul G",
17:   "value": ""
18:   }
19:   ]
20:   }
21:   ]
22:   },
23:   {
24:   "key": "Modul C",
25:   "value": [
26:   {
27:   "key": "Modul E",
28:   "value": ""
29:   }
30:   ]
31:   }
32:   ]
33:   }
 34   ]
```

Sl. 3.4. *Primjer ugniježdene strukture u JSON formatu.*

Format prikazan slikom **3.4** daje pregledan i jednostavan uvid u hijerarhiju kôda te je iz njega lako pročitati koje programske module sadrži svaki pojedini modul. Dokumentacija napisana u ovom formatu bit će tako pogodna za prikaz u internetskoj aplikaciji, čija je izrada objašnjena u idućim potpoglavljima.

3.4. Generirane datoteke i izvlačenje podataka iz njih

Većina programskih rješenja u automobilskoj industriji napisana je u MISRA C programskom jeziku. MISRA C predstavlja skup smjernica za razvoj programskih rješenja pisanih u programskom jezik C. Cilj ovih smjernica je povećati sigurnost, pouzdanost i stabilnost programskog kôda u ugradbenim sustavima koji se koriste u automobilskoj industriji. Za izradu alata u ovome radu iz tog razloga korišten je programski kôd pisan u C programskom jeziku. Pisanju kôda pristupano je modularno te je za svaku datoteku generirana i pripadna datoteka zaglavlja (engl. *header file*). Programskim prevoditeljem *GCC* (engl. ***G**NU **C**ompiler **C**ollection*) kreira se potrebna *map* datoteka. Za te potrebe najprije su korištene različite zastavice prikazane na slici **3.5**.

```
gcc -O2 -g -c -Wall -Werror -Wextra -std=gnu99 -O0 modulA.c modulB.c
modulC.c
```

Sl. 3.5. *Kôd za kreiranje map datoteke, prvi korak.*

Prva zastavica, *-O2*, označava drugu razinu optimizacije, koja uključuje gotovo sve optimizacije koje ne obuhvaćaju kompromis brzine i prostora. Druga zastavica, *-g*, označava da je potrebno izraditi podatke za otklanjanje pogrešaka (engl. *debugging information*) u izvornom formatu operativnog sustava za koji se provodi prevođenje. Sljedeća zastavica, *-c*, označava da se izvorne datoteke sastavljaju bez povezivanja. Zastavica *–Wall* omogućuje sva upozorenja programa prevoditelja. Zastavica *–Werror* pretvara sva upozorenja u pogreške. Zastavica *–Wextra* omogućava dodatne poruke upozorenja. Zastavica *-std=gnu99* označava opcije dijalekta programskog jezika, pri čemu *gnu99* označava *C99* standard s *GNU* dodacima. Zastavica *–O0* smanjuje vrijeme sastavljanja (engl. *compilation time*) te osigurava da otklanjanje pogrešaka daje rezultate u očekivanom formatu. Nakon toga navode se sve kreirane programske datoteke. Ovo rezultira kreiranjem objektnih datoteka.

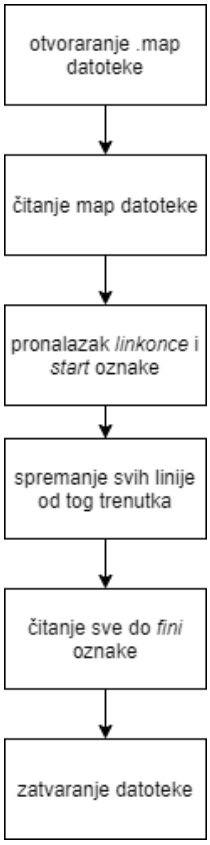
U drugom koraku, prikazanom slikom **3.6**. ponovno se koriste različite zastavice.

```
gcc -o full -g -Wall -Werror -Wextra -O0 modulA.o modulB.o modulC.o
-Wl,-Map=results.map
```

Sl. 3.6. *Kôd za kreiranje map datoteke, drugi korak.*

Zastavica *–o full* kreira izlaznu datoteku s optimiziranom veličinom kôda i vremenom izvršavanja. Nakon navođenja ostalih zastavica, objašnjenih u prethodnom odlomku, redom se navode kreirane objektne datoteke. Zastavica *–Wl* koristi se za povezivanje te rezultate sprema u *map* datoteku. Time je kreirana potpuno sastavljena (engl. *compiled*) i povezana (engl. *linked*) datoteka.

Generirana *map* datoteka sastoji se od nekoliko dijelova, kao što su ulazni odjeljci, podaci o konfiguraciji memorije te skripta povezivanja (engl. *linker script*) i memorijske karte (engl. *memory map*). Podaci potrebni za rekonstrukciju arhitekture programskog kôda nalaze se u dijelu skripte povezivanja i memorijske karte. Koraci izvlačenja podataka prikazani su slikom **3.7**.



Sl. 3.7. *Koraci izvlačenja podataka iz generiranih datoteka.*

Skripta kojom su implementirani koraci prikazani na slici **3.7**. izrađena je u programskom jeziku *Python* pomoću *pandas* biblioteke, opisane u potpoglavlju 3.1.5. Prvi korak je otvaranje datoteke. *Python map* datoteku čita kao tekstualnu datoteku. Zatim je datoteku potrebno čitati sve dok se ne dođe do oznake *linkonce* koja označava početak dijela u kojem su navedeni programski moduli i sučelja koje ti moduli koriste. Dio *linkonce* započinje oznakom *start*, a završava oznakom

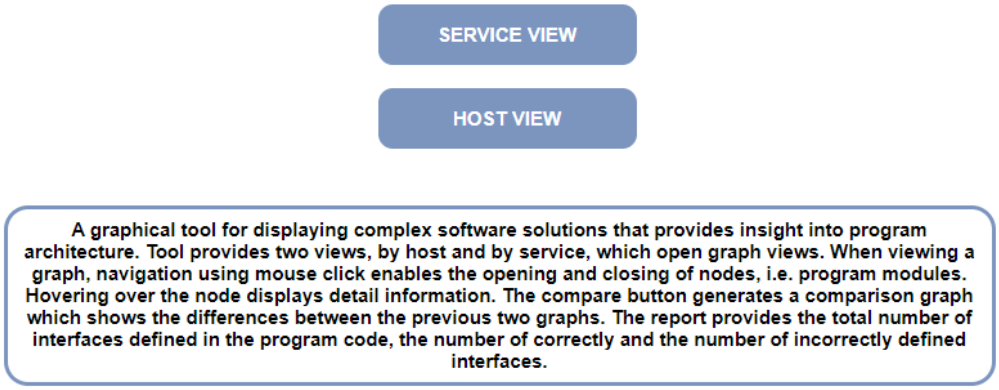
fini. Svi redovi između predstavljaju redove od interesa te je te redove potrebno spremati. Na kraju se *map* datoteka zatvara. Linije od interesa se spremaju u *Python* listu.

3.5. Izrada grafičkog prikaza

Grafički prikaz izrađen je u *JavaScriptu* pomoću biblioteke *D3.js*. Grafičko sučelje za korisnika podijeljeno je u nekoliko dijelova, koji su izrađeni kao zasebne *HTML* stranice međusobno povezane gumbima (engl. *button*). Najprije se korisniku otvara prozor u kojem može odabrati vrstu željenog pregleda, prema *host*-u ili prema servisima. Prvi pregled pruža pregled svih servisa koji se izvode na određenom *host*-u, a drugi daje pregledniji uvid u pojedini servis. Nakon odabira željenog prikaza, korisniku se otvara novo sučelje u kojemu je zaslon podijeljen na dva dijela te prikazuje grafove usporedno jedan kraj drugoga. Lijevi je graf nastao iz dokumentacije, a desni je graf s podacima iz generiranih programskih datoteka. Korisniku se omogućava i opcija usporedbe, koja otvara novo sučelje u kojem se prikazuje graf usporedbe i izvješće. Pojedini dijelovi grafičkih sučelja detaljno su objašnjeni u nastavku.

3.5.1. Prozor za odabir sučelja

Prozor za odabir sučelja prikazan slikom **3.8** korisniku omogućava odabir željenog pregleda preko *host*-a ili preko servisa pomoću dva gumba (engl. *button*). Ispod dva gumba nalazi se opis programskog alata koji korisniku objašnjava njegov način rada i funkcionalnosti koje pruža.



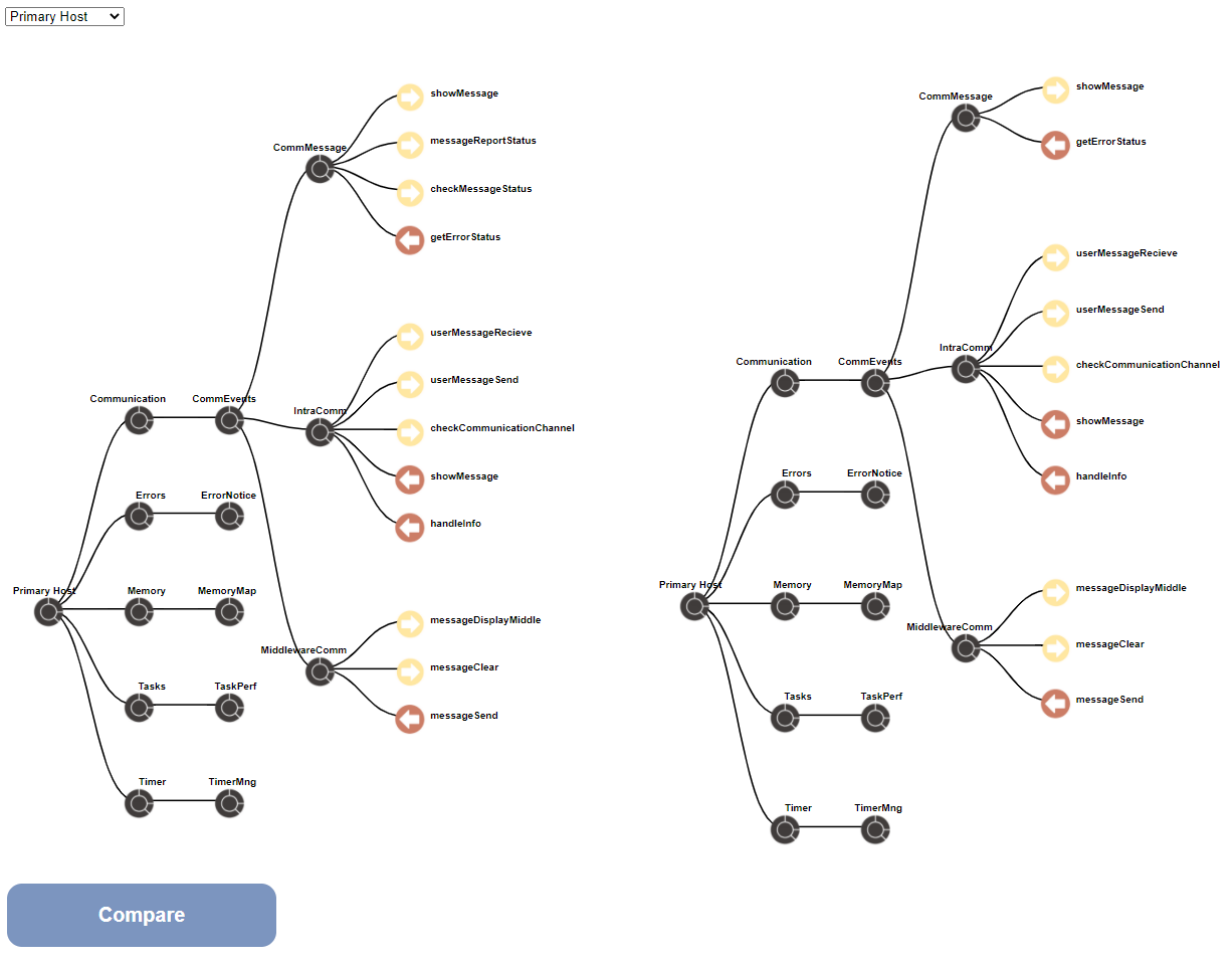
Sl. 3.8. *Grafički prikaz prozora za odabir sučelja.*

Oba gumba predstavljaju poveznicu na novu *HTML* stranicu na kojoj će se prikazati grafovi koje korisnik odabere. Korisnik se u svakom trenutku može ponovno vratiti na početnu stranicu.

3.5.2. Sučelje za prikaz grafova

Grafovi koji daju uvid u strukturu programskog kôda u obliku stabla se mogu prikazivati prema *host*-u ili prema servisima, ovisno o odabiru korisnika. Pregled prema *host*-u prikazuje sve programe koji se izvode na odabranom *host*-u, a pregled prema servisima daje uvid u komponente, jedinice i sučelja koji se izvode na tom servisu (vidi poglavlje 2.). Temeljni način prikaza je isti, ali postoje određene razlike.

Pregled programskog kôda prema *host*-u prikazan slikom **3.9**. korisniku daje mogućnost uvida u sve programske kôdove koji se izvode na *Primary Host*-u.



Sl. 3.9. *Prikaz strukture programskog kôda koji se izvodi na Primary host-u, s lijeve strane nalazi se graf generiran s podacima iz dokumentacije, a s desne graf generiran s podacima izvučenim iz generiranih datoteka.*

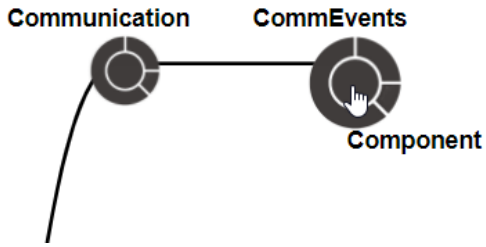
Korisnik pomoću padajućeg izbornika odabire koji će se graf prikazivati na zaslonu. Ovisno o odabranom grafu, s lijeve i s desne strane prema podacima iz *JSON* datoteke generiraju se dva grafa. Postavlja se korijenski čvor (engl. *root node*) te redom njemu pripadajući čvorovi s lijeva na desno. Graf je horizontalno orijentiran te se otvara s lijeve na desnu stranu. Nakon kreiranja čvorova, oni se međusobno povezuju linijama. Nakon osnovnog iscrtavanja grafa, dodaju mu se dodatne informacije u ovisnosti o podacima unesenima u *JSON* datoteku. Primjer *JSON* objekta sa svim potrebnim svojstvima za generiranje grafa prikazan je slikom **3.10**.

Linija ***Kôd***

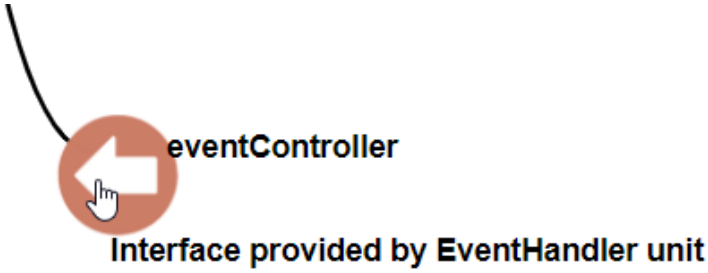
```
1:   {
2:   "name": "getErrorStatus",
3:   "info": "Interface provided by ErrorStatus unit",
4:   "icon": "required.png"
5:   }
```

Sl. 3.10. *JSON objekt pripremljen za prikaz na grafu.*

Ime modula ili sučelja (engl. *name*) prikazuje se iznad čvora koji prikazuje taj modul. Informacije o modulu ili sučelju (engl. *info*) prikazuju se prelaskom miša preko čvora. Ukoliko se radi o programskom modulu, prikazuje se informacija radi li se o *host*-u, komponenti, servisu ili jedinici kao što je i prikazano slikom **3.11**. , a ukoliko se radi o sučelju prikazuje se informacija je li sučelje pružano ili ako je zahtijevano, prikazuje se koji programski modul pruža to sučelje, kao što se vidi na slici **3.12**.

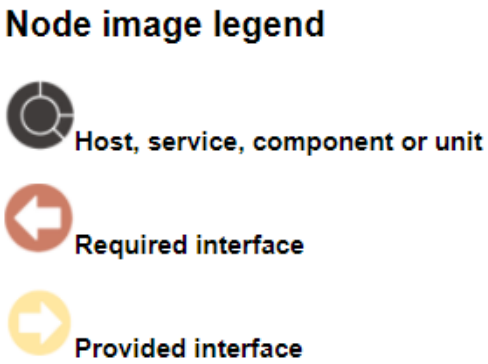


Sl. 3.11. *Prikaz dodatnih informacija o programskom modulu prelaskom miša.*



Sl. 3.12. *Prikaz dodatnih informacija o sučelju prelaskom miša.*

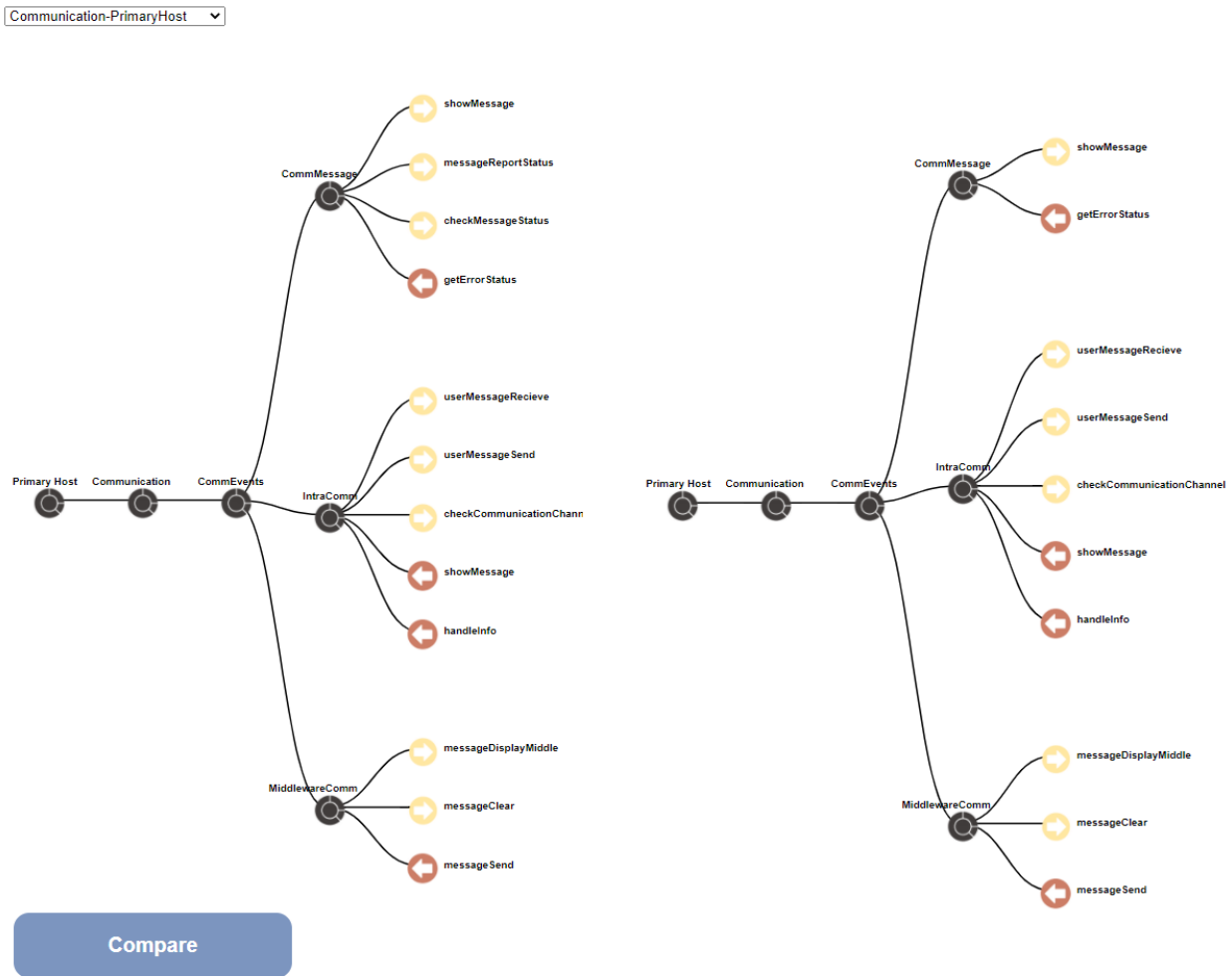
Sličica modula (engl. *icon*) prikazuje se na mjestu gdje je čvor postavljen. Sličice su jednake za sve programske module, dok se kôd programskih sučelja razlikuju ovisno o tome je li sučelje zahtijevano ili pružano, a prikazane su slikom **3.13**.



Sl. 3.13. Legenda koja prikazuje sličice u ovisnosti o tome koji čvor prikazuju

Pregled programskog kôda prema servisima prikazan na slici **3.14**. daje prikaz odabranog servisa, u ovom primjeru *Communication Service*. U gornjem dijelu grafičkog prikaza nalazi se padajući izbornik koji korisniku omogućava odabir željenog servisa. Ovisno o odabiru grafa u padajućem izborniku mijenja se grafički prikaz s obje strane. S lijeve strane nalazi se graf, koji prikazuju podatke definirane u dokumentaciji nakon pisanja programskog kôda. S desne strane nalazi se pripadajući graf generiran podacima izvučenim iz programskih datoteka.

U donjem dijelu grafičkog prikaza nalazi se gumb za prikaz usporednog grafa koji će prikazati razlike između lijevog (nastalog iz dokumentacije) i desnog (nastalog iz *map* datoteke) grafa. On predstavlja poveznicu na novu *HTML* stranicu koja će prikazati usporedbu nakon pritiska mišem.



Sl. 3.14. *Prikaz strukture programskog kôda koji se izvodi na odabranom servisu, s lijeve strane nalazi se graf generiran s podacima iz dokumentacije, a s desne graf generiran s podacima izvučenim iz generiranih datoteka.*

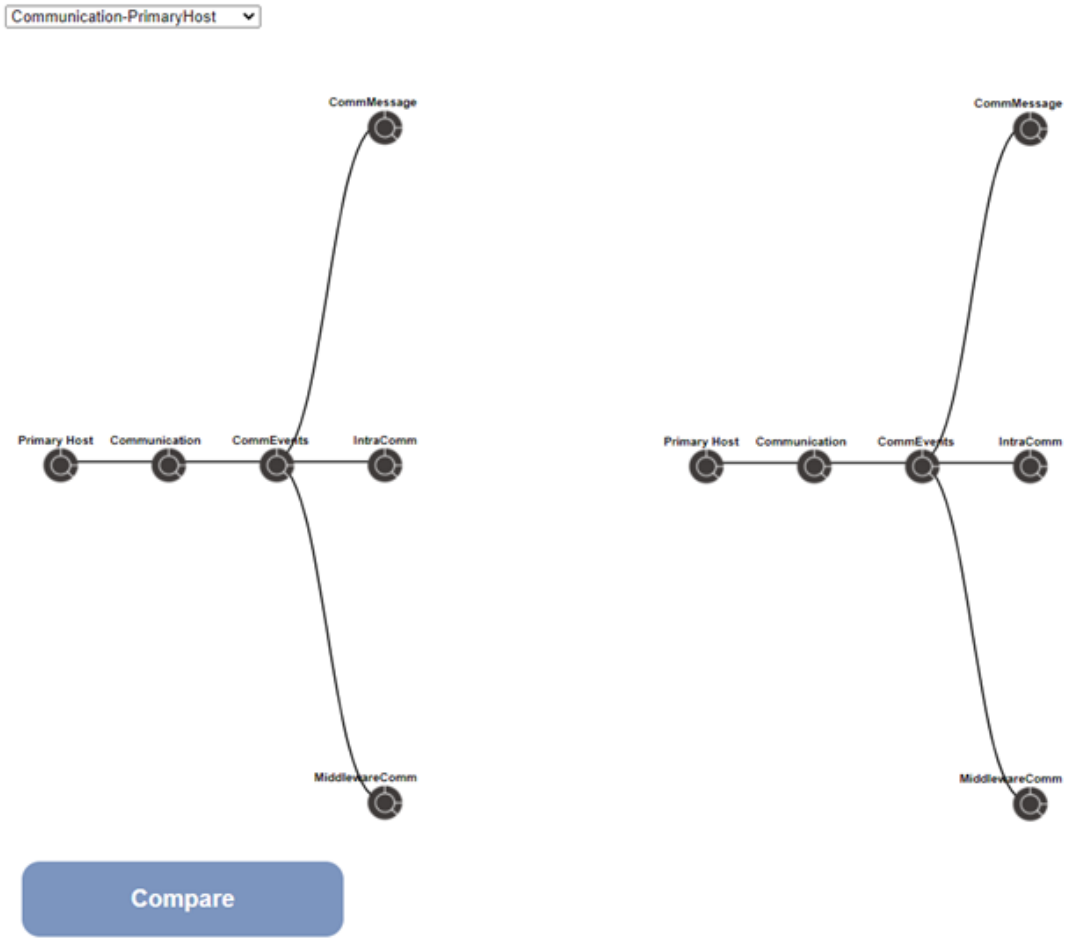
Graf je oblika stabla i sastoji se od čvorova i veza. Čvor je prikazan okruglom sličicom crne boje ukoliko se radi o programskom modulu, sličicom žute boje ukoliko se radi o sučelju koje modul pruža te sličicom narančaste boje ukoliko se radi o sučelju koje modul zahtjeva za normalan rad. Svaki čvor uz sličicu ima i oznaku (engl. *label*) koja prikazuje ime tog programskog modula ili sučelja. Veze između čvorova prikazane su linijama.

Prelaskom miša (engl. *hover*) preko sličica koje predstavljaju modul ili sučelje prikazuje se kratka animacija te dodatne informacije o tom modulu ili sučelju. Za modul se prikazuje informacija o vrsti programskog modula. Vrste modula su *host*, servis, komponenta te jedinica (vidi poglavlje 2.). Dodatne informacije koje se prikazuju o sučelju su vrsta sučelja (pružano ili zahtijevano) te ukoliko se radi o zahtijevanom sučelju prikazuje se naziv programskog sučelja koje

ga pruža. Prikaz informacija koje se dobivaju prelaskom miša preko pojedinog dijela grafa prikazane su slikama **3.11**. i **3.12**.

Primjerice na slici **3.11**. vidi se crni čvor naziva *CommEvents* koji predstavlja komponentu, a ta informacija prikazuje se prelaskom miša preko sličice. Na slici **3.12**. vidi se narančasti čvor naziva *eventController* koji predstavlja zahtijevano sučelje. Prelaskom miša preko sličice ovog čvora prikazuje se informacija kako ovo sučelje pruža jedinica naziva *EventHandler*.

Kao što je već rečeno, programski kôd je često kompleksan, a time i grafovi koji ga prikazuju mogu imati mnogo razina. Kako bi se postignula jednostavnost prikaza, pri otvaranju stranice grafovi nisu potpuno razgranati, već se korisniku omogućuje kretanje po grafu. Kretanje po grafu ostvaruje se lijevim pritiskom miša na određeni čvor koji zatim otvara njemu pripadne čvorove. Ponovnim pritiskom miša na taj isti čvor, on se zatvara, odnosno ne prikazuju se više njemu pripadni čvorovi. Na slikama **3.9**. i **3.14**. prikazani su otvoreni čvorovi jedinica *CommMessage*, *IntraComm* i *MiddlewareComm*, a slika **3.15**. prikazuje zatvorene čvorove.



Sl. 3.15. *Prikaz strukture programskog kôda sa zatvorenim čvorovima*.

3.6. Usporedni graf

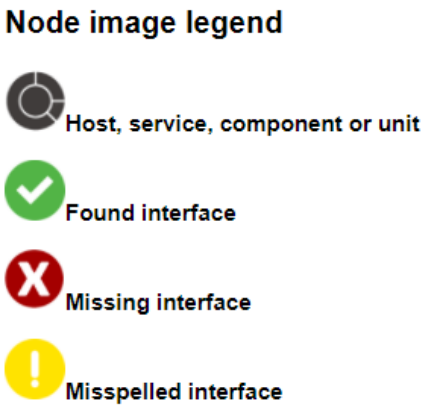
Podatke možemo usporediti tek nakon što prikupimo sve podatke iz dokumentacije i sve podatke iz generiranih datoteka. Algoritam za usporedbu izrađen je u programskom jeziku *Python* pomoću *pandas* biblioteke (vidi točku 3.1.5). Podaci iz dokumentacije spremaju se u prvu listu, a podaci iz generiranih datoteka u drugu listu. Ove liste sada sadrže nazive svih programskih modula i sučelja koja se nalaze u dokumentaciji, odnosno u generiranim datotekama. Algoritam se u suštini sastoji od prolaska kroz liste te pretraživanja kojim se utvrđuje nalaze li se elementi prve liste u drugoj, odnosno nalaze li se sva imena programskih sučelja i modula iz dokumentacije u generiranim datotekama. Pronađene vrijednosti spremaju se u novu listu. Nova lista može sadržavati ponavljajuće vrijednosti te se one moraju filtrirati kako ne bi bilo ponavljanja. Filtriranje se vrši na način da se spremi samo prvo ponavljanje, dok se ostala postavljaju kao nedostajuće vrijednosti (engl. *nan values*) te se kao takve uklanjaju.

Ispod grafa nalazi se izvještaj u obliku tablice, prikazan slikom **3.18**. U izvještaju su prikazana tri retka koja redom prikazuju ukupan broj sučelja, broj ispravno implementiranih sučelja te broj pogrešno implementiranih sučelja, odnosno broj nedostajućih sučelja. Prikazuje se i pripadni postotak. Izvještaj je prikazan u obliku tablice, a podaci potrebni za prikaz u tablici također su učitani pomoću *JSON* formata. Podaci se dobivaju unutar *Python* skripte u kojoj se kreiraju i podaci za usporedni graf. Iz dokumentacije se prebroji ukupan broj definiranih sučelja. Zatim se prebrojavaju imena sučelja koja se poklapaju u podacima iz dokumentacije i u podacima iz generiranih datoteka te tako nastaju pronađeni podaci. Nedostajuće vrijednosti se računaju kao razlika ukupnog broja sučelja i broja pronađenih podataka. Izračunavaju se pripadni postoci za pronađene i nedostajuće podatke.

3.6.1. Grafički prikaz usporednog grafa

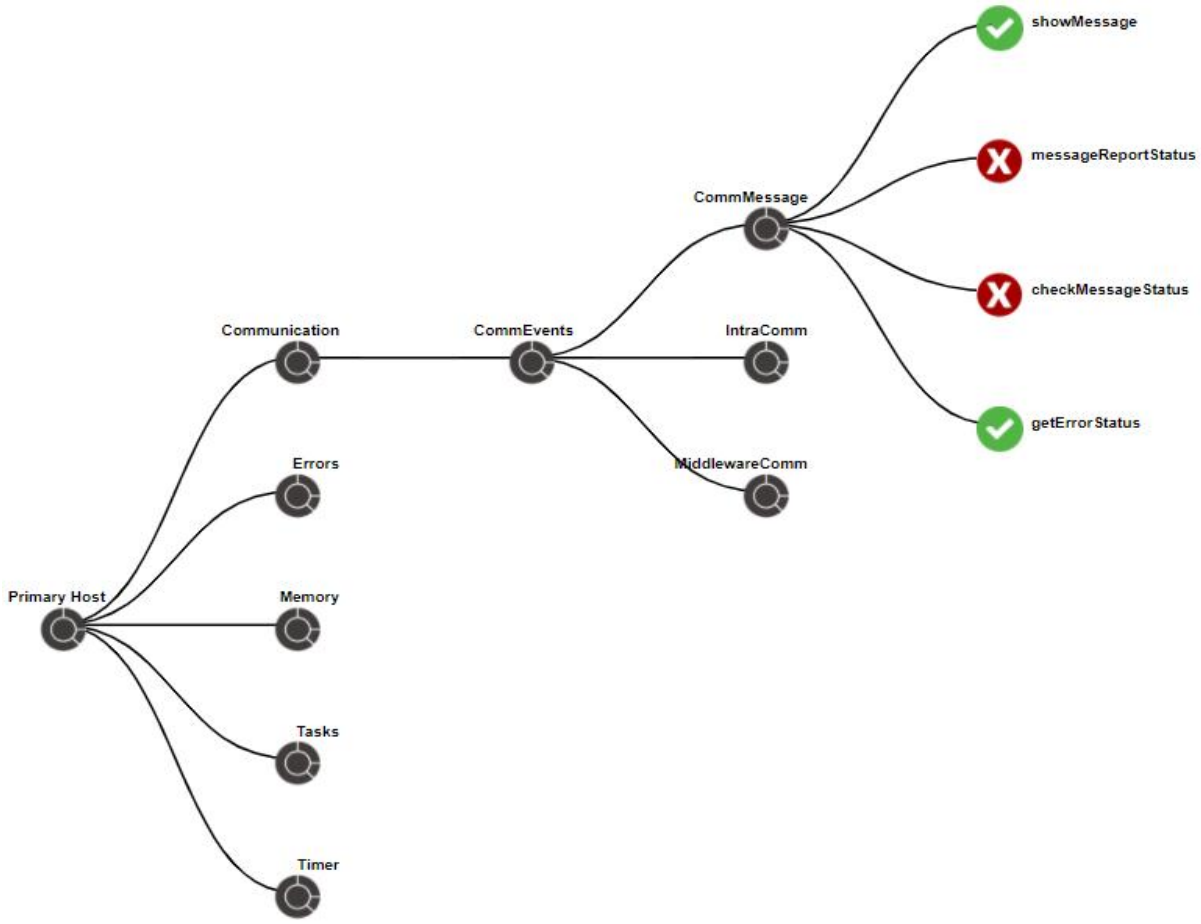
Jedna od najvažniji funkcionalnosti ovog alata je usporedba dva grafa. Ova usporedba omogućava korisniku uvid u nastale pogreške, odnosno sučelja koja nisu implementirana u programskom rješenju, a u dokumentaciji su navedena kao nužna. Time korisnik može ispraviti te pogreške i prevenirati daljnje probleme. Također, ovom usporedbom može dobiti uvid u kompleksnost programskog kôda te ga po potrebi reducirati.

Grafičko sučelje za prikaz usporednog grafa prikazano slikom **3.183.17**. korisniku prikazuje samo jedan graf. Na njemu su odgovarajućim bojama i sličicama istaknute potencijalne razlike između prethodna dva grafa koje korisnik ima na uvid u prethodnom koraku (vidi točku 3.5.2). Sličice koje su do sada žutom i narančastom bojom prikazivale sučelja sada su zamijenjene zelenim i crvenim. Zelena boja označava da je programsko sučelje zahtijevano u dokumentaciji ispravno implementirano i u samom kôdu. Ukoliko to sučelje nije implementirano ili ima pogrešan naziv, njegov čvor je prikazan sličicom crvene boje te je dodatno naglašeno crvenim oznakama uz čvor. Sličice su prikazane slikom **3.16**.



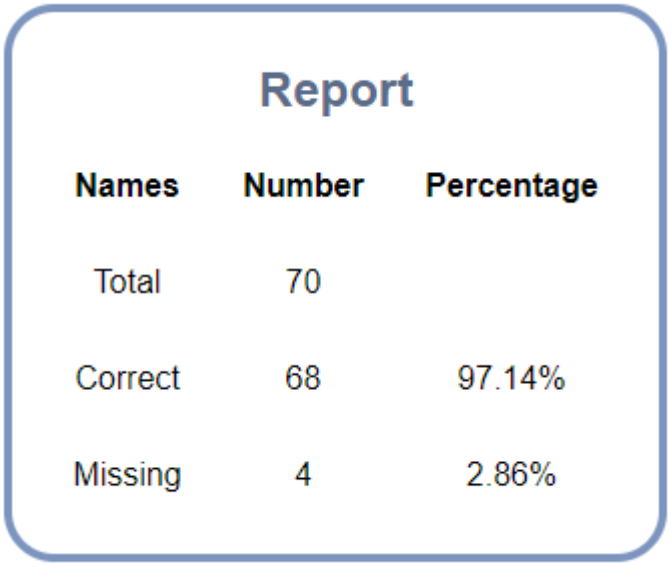
Sl. 3.16. *Legenda koja prikazuje sličice u ovisnosti o tome kakavi čvor prikazuju.*

Prelaskom miša preko čvora i dalje se prikazuju informacije o tom čvoru kao i u prethodnom prikazu. Na čvorovima na kojima je otkrivena pogreška sada se ispisuje i informacija o imenu programskog modula koji je to sučelje trebao pružati, odnosno implementirati, kao što je i prikazano slikom **3.18**. Primjerice, na ovome grafu je prikazano kako nedostaju sučelja *messageReportStatus* i *checkMessageStatus*. Također, i dalje je omogućena navigacija, odnosno kretanje po grafu pritiskom lijeve tipke miša.



Sl. 3.17. *Graf nastao na temelju usporedbe grafova prikazanih slikom **3.9.***

Ispod grafa na ovoj stranici nalazi se tablica prikazana slikom **3.18**. koja prikazuje izvještaj koji se temelji na usporedbi podataka iz dokumentacije i podataka izvučenih iz *map* datoteke. U izvještaju se prikazuje ukupni broj sučelja implementiranih na tom *host*-u ili servisu, broj ispravno implementiranih programskih sučelja te broj pogrešno implementiranih sučelja. Također prikazuje i postotak ispravno i pogrešno implementiranih sučelja (vidi točku 3.6).



Report

Names	Number	Percentage
Total	70	
Correct	68	97.14%
Missing	4	2.86%

Sl. 3.18. *Izvještaj sa stranice prikazan tablicom, odgovara grafičkom prikazu sa slike* ***3.17.***

Na samom dnu stranice nalaze se dva gumba. Prvi omogućava povratak na prethodnu stranicu, a drugi omogućava izvoz (engl. *export*) generiranog grafa usporedbe i izvješća u *pdf* formatu. Generirana izvješća mogu poslužiti za usporedbu novijih i starijih programskih rješenja. Izvještaj se generira na jednoj stranici. U gornjem dijelu se nalazi graf, a ispod njega se nalazi tablica sa izvještajem.

3.7. Pokretanje alata za vizualizaciju programskog rješenja

Alat za vizualizaciju se pokreće pomoću naredbe prikazane slikom **3.19**. pri čemu *main.py* označava naziv programa, a *map_file* je parametar koji se predaje funkciji, a označava generiranu *map* datoteku. Map datoteka generira se iz .c datoteke prema koracima opisanim u potpoglavlju 163.4.

```
python main.py map_file
```

Sl. 3.19. *Kôd za pronalaženje pogrešaka u nazivu datoteke.*

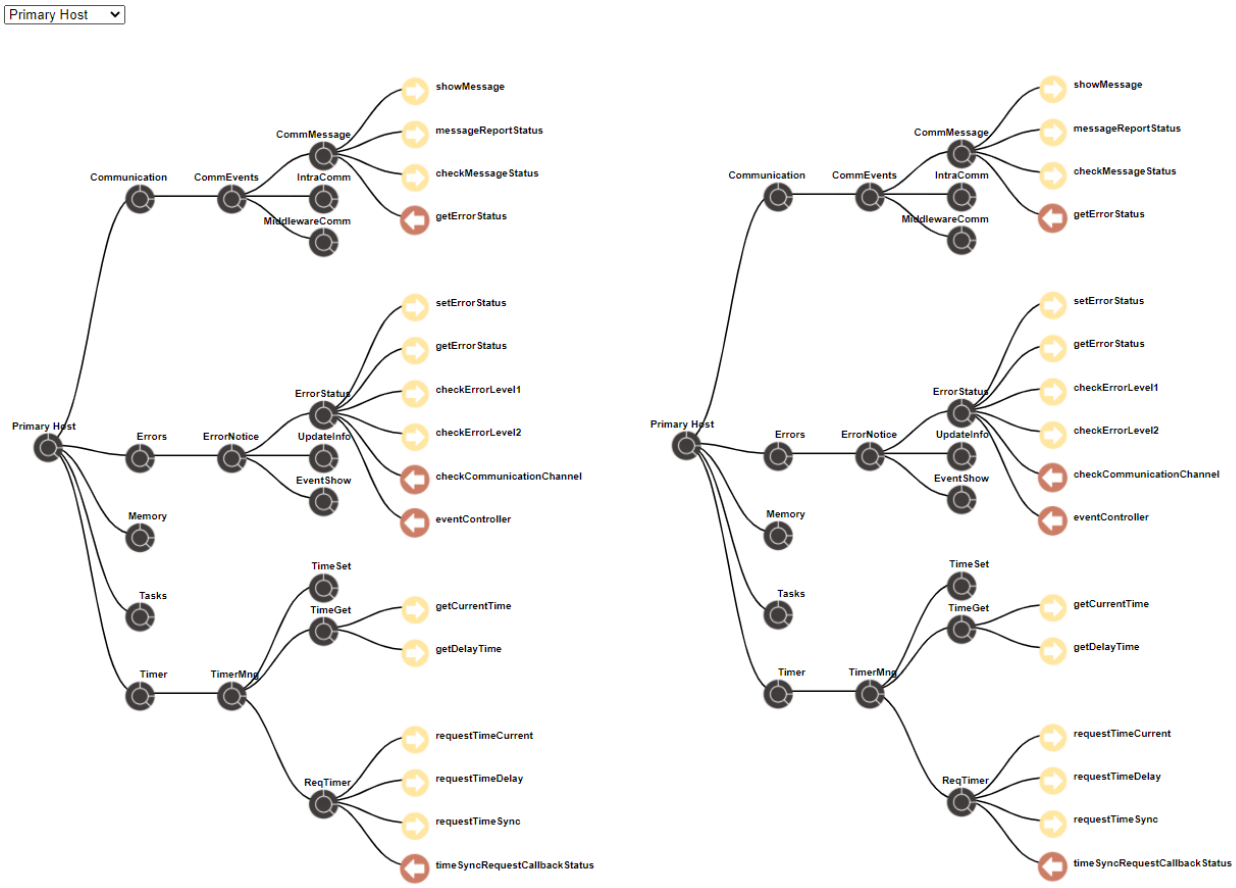
Korisniku se zatim otvara prozor s početnom *HTML* stranicom prikazanom slikom **3.8**. Glavna funkcija *main.py* poziva sve potrebne funkcije za čitanje *map* datoteke predane funkciji i za kreiranje grafova.

4. VERIFIKACIJA PREDLOŽENOG ALATA ZA VIZUALIZACIJU KOMPLEKSNOSTI KÔDA

Kako je već ranije navedeno, glavna funkcionalnost ovoga alata za grafički prikaz kompleksnosti izvršnog kôda je uspoređivanje grafova. U ovome poglavlju prikazani su različiti scenariji koji se mogu pojaviti prilikom usporedbe. Prvi je najpovoljniji, onaj u kojemu su ispravno definirana sva sučelja u programskom kôdu te nema potencijalnih pogrešaka. U drugom scenariju je moguća pogreška pri upisu naziva pojedinog sučelja te se tada prikazuje upozorenje. Posljednji, treći scenarij, prikazuje situaciju u kojoj nedostaju pojedina sučelja. Za potrebe prikazivanja ovih scenarija generirani su umjetni podaci, odnosno dokumentacija i programski kôd koji su prikazani.

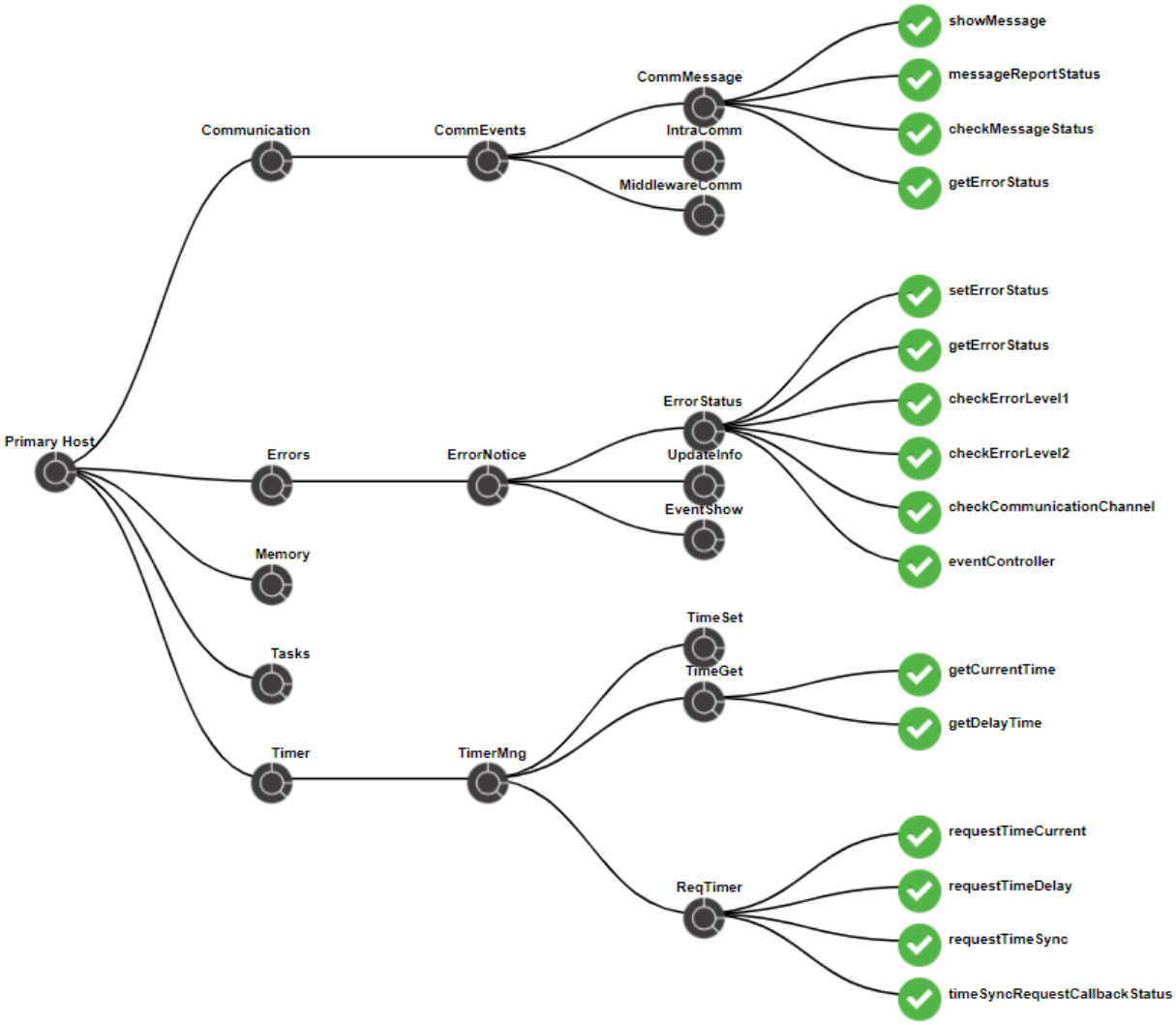
4.1. Prvi scenarij

Jedan od mogućih rezultata usporedbe je graf u kojem se u potpunosti podudaraju sva imena sučelja i programskih modula. Na slici **4.1** možemo uočiti da su lijevi i desni graf potpuno jednaki,



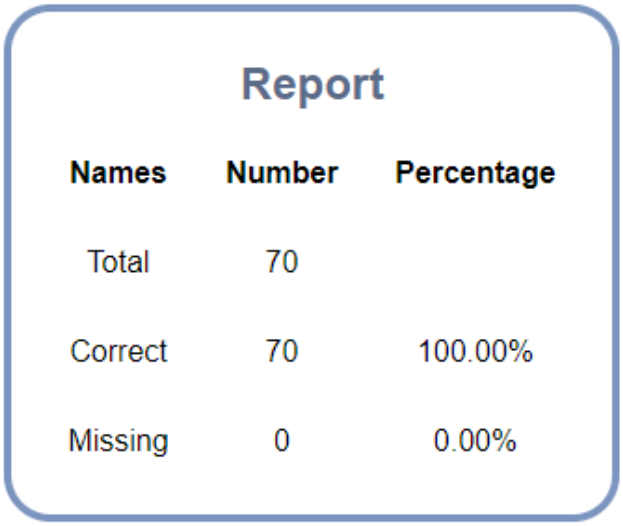
Sl. 4.1. *Prvi scenarij u kojemu je lijevi (graf generiran s podacima iz dokumentacije) jednak desnom (graf generiran podacima iz map datoteke).*

odnosno nazivi svih sučelja im se podudaraju. Ovo se možemo provjeriti opcijom usporedbe, koja generira graf prikazan slikom **4.2**.



Sl. 4.2. *Graf usporedbe za slučaj kada su ispravno definirani nazivi svih sučelja.*

U ovome slučaju sve sličice su prikazane zelenom bojom što označava ispravno definirane nazive sučelja. Izvještaj za pripadni graf prikazan je slikom **4.3**. Možemo uočiti kako je ukupan broj sučelja definiranih na ovome grafu 70 te da je svih 70 ispravno definirano.



Report

Names	Number	Percentage
Total	70	
Correct	70	100.00%
Missing	0	0.00%

Sl. 4.3. *Izvještaj za slučaj kada su ispravno definirani nazivi svih sučelja, odgovara grafičkom prikazu sa slike **4.2***

4.2. Drugi scenarij

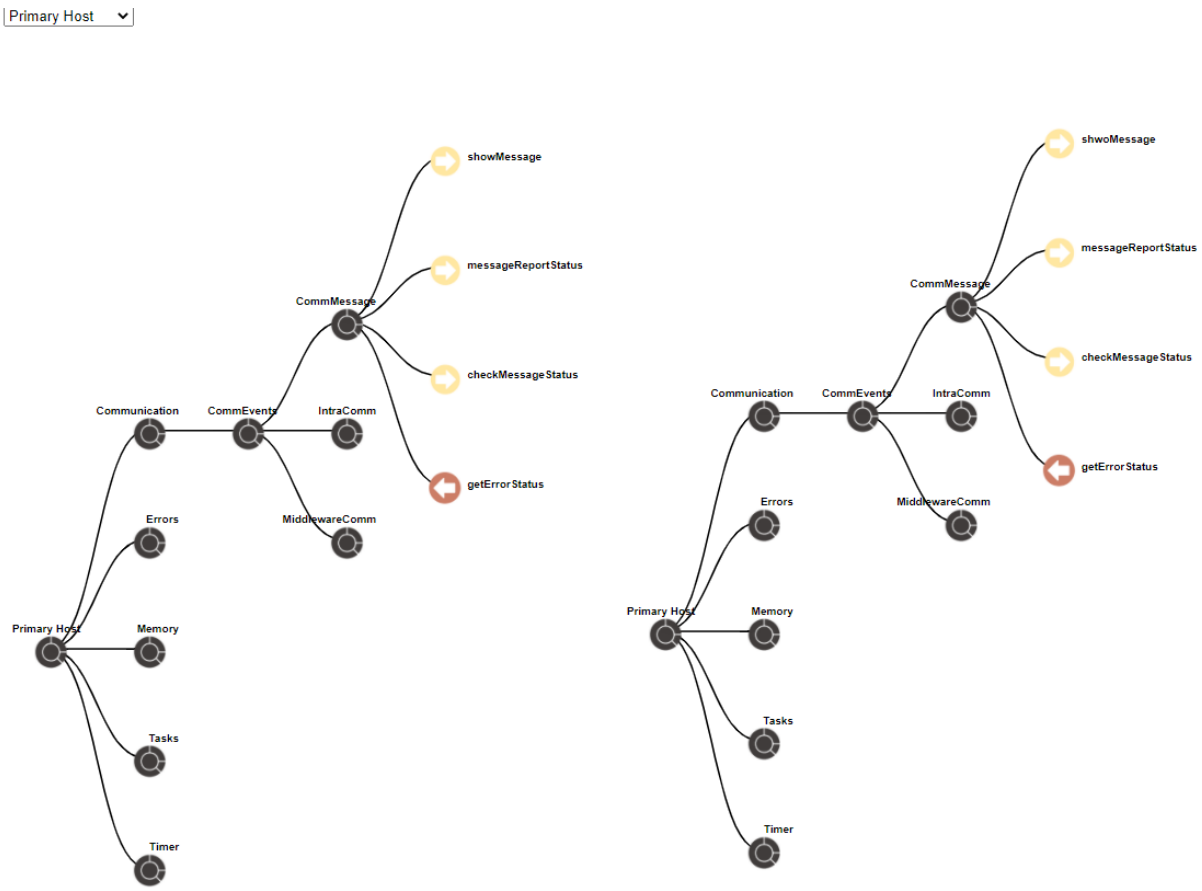
Drugi slučaj podrazumijeva pogrešku u nazivu sučelja. Primjer ovakve pogreške prikazan je slikom **4.5**. na kojoj možemo uočiti da je naziv funkcije definiran kao „*shwoMessage*“ umjesto „*showMessage*“.

Pogreška u nazivu funkcije ubraja se u ovaj slučaj samo ako su zamijenjena dva slova prilikom pisanja naziva funkcija. Sve veće pogreške smatraju se kao nedefinirana, odnosno nedostajuća sučelja. Detektiranje ove pogreške izrađeno je pomoću programskog jezika *Python* i biblioteke *difflib*. Biblioteka *difflib* pruža funkcije za usporedbu slijeda znakova. Korištena je funkcija *SequenceMatcher* prikazana na slici **4.4**. koja računa najdužu kontinuiranu podudarnost među znakovima. Funkcija *ratio* vraća rezultat sličnosti (decimalni broj između 0 i 1) između dva znakovna niza. Što je veći rezultat sličnosti, dva znakovna niza su sličnija, pri čemu rezultat sličnosti od 1.0 označava potpuno poklapanje, a rezultat sličnosti 0.0. označava potpunu različitost. Za slučaj kada se toleriraju samo pogreške u nazivu funkcija, kao zadovoljavajući rezultat sličnosti (engl. *ratio*) uzima se svaki koji iznosi više od 0.8. Ovaj prag tolerancije obuhvaća samo slučajeve u kojima su dva slova zamijenjena te će svaka veća pogreška rezultirati nezadovoljavajućim rezultatom sličnosti. U slučaju prikazanom slikom **3.5.4.4**. rezultat sličnosti iznosi 90.9%.

```
ratio = SequenceMatcher(a='showMessage',b='shwoMessage').ratio()
```

Sl. 4.4. *Kôd za pronalaženje pogrešaka u nazivu datoteke.*

Zamjena dva slova u nazivu funkcija uzrokovat će prikaz upozorenja. Upozorenje prikazuje žutu ikonicu i obavijest o zamijeni slova pri prelasku mišem preko ikonice kao što je i prikazano slikom **4.6**. Pogledamo li izvještaj za generiranu usporedbu prikazan slikom **4.7**. možemo uočiti da je od ukupno 70 sučelja, njih 69 ispravno definirano, a sučelje kojem je krivo definiran naziv je pridodano nedostajućim sučeljima. Ovaj slučaj je izdvojen kao poseban, jer je sučelje definirano, tako da ne pripada trećem scenariju, ali mu je naziv krivo napisan, tako da se ne može ubrojiti u ispravno definirana sučelja.



Sl. 4.5. *Prvi scenarij u kojemu je lijevi (graf generiran s podacima iz dokumentacije) različit od desnog (graf generiran podacima iz map datoteke) u kojemu je krivo napisan naziv funkcije showMessage.*

Sl. 4.6. *Graf usporedbe za slučaj kada su u nazivu funkcije zamijenjena dva slova.*

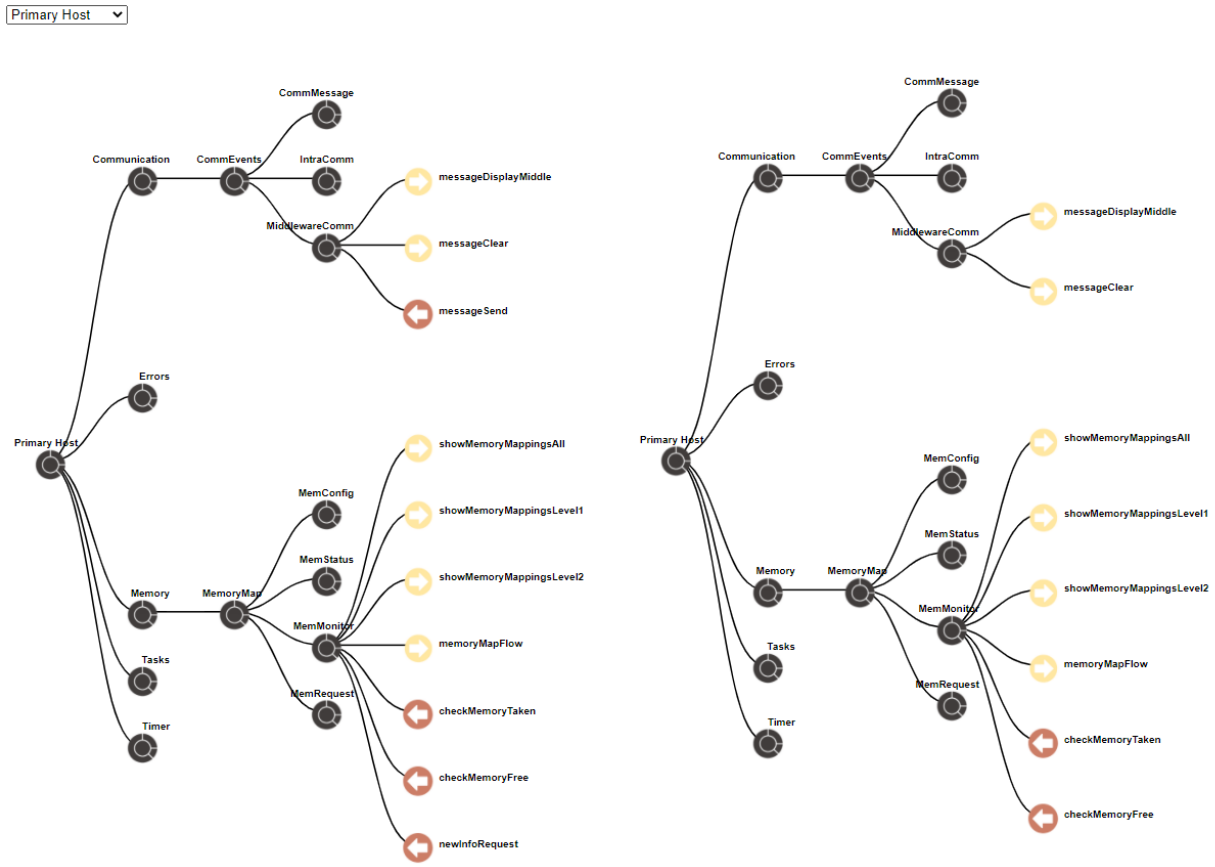
Report

Names	Number	Percentage
Total	70	
Correct	69	98.57%
Missing	1	1.43%

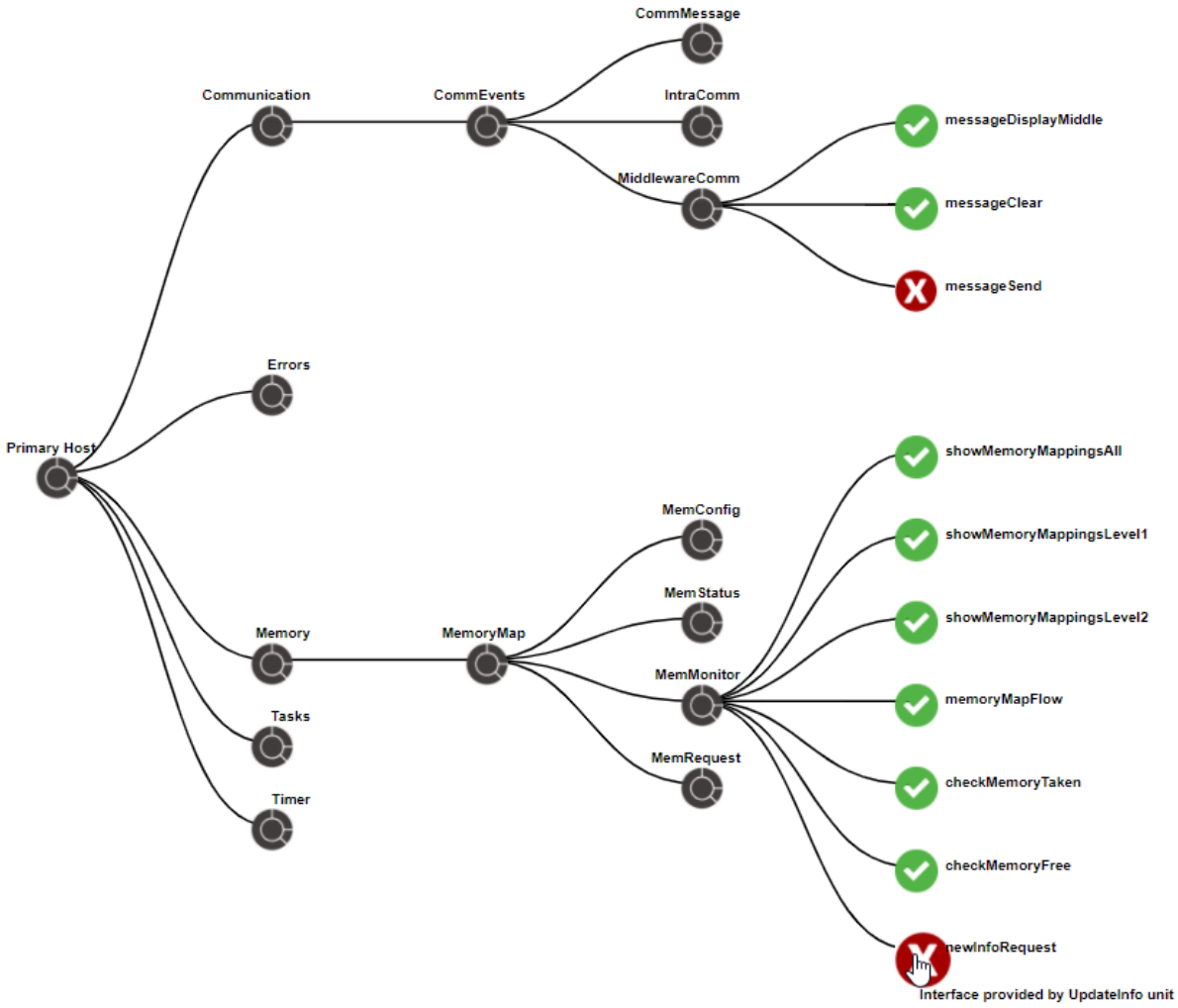
Sl. 4.7. *Izvještaj za slučaj kada su u nazivu funkcije zamijenjena dva slova, odgovara grafičkom prikazu sa slike* **4.7.*4.6.***

4.3. Treći scenarij

Posljednji slučaj koji se prikazuje je onaj u kojemu nisu definirana pojedina sučelja, a jedan ovakav scenarij prikazan je slikom **4.8**. Prilikom usporedbe ova dva grafa sličice nedostajućeg čvora iscrtavaju se crvenom bojom, a prelaskom miša preko sličice prikazuje se informacija o programskom modulu koji bi trebao pružati nedostajuće sučelje. Ovaj slučaj prikazan je na slici **4.9** gdje se može uočiti da nedostaju sučelja *messageSend* i *newInfoRequest*. Izvještaj prikazan slikom **4.10**. prikazuje kako od ukupno 70 definiranih sučelja, njih 68 ih je ispravno definirano, a 2 nedostaju.



Sl. 4.8. *Prvi scenarij u kojemu je lijevi (graf generiran s podacima iz dokumentacije) različit od desnog (graf generiran podacima iz map datoteke) kojemu nedostaju sučelja messageSend i newInfoRequest.*



Sl. 4.9. *Graf usporedbe za slučaj kada nedostaju sučelja messageSend i newInfoRequest.*

Report

Names	Number	Percentage
Total	70	
Correct	68	97.14%
Missing	2	2.86%

Sl. 4.10. *Izvještaj za slučaj kada su dva sučelja nedostajuća, odgovara grafičkom prikazu sa slike **4.9**.*

Ovaj alat prilagođen je programskom rješenju za koje je izrađena dokumentacija u obliku json datoteke. Time radi i stabilno brzo jer su sva područja od interesa jasno definirana. Ukoliko bi se koristio za drugačija programska rješenja bile bi potrebne izmjene kako bi se automatiziralo čitanje i raščlanjivanje dokumentacije. Također, bila bi potrebna i nadogradnja kako bi mogao čitati i druge objektne datoteke, osim *map* datoteka.

5. ZAKLJUČAK

Automobilska industrija svakodnevno se razvija i napreduje, a to podrazumijeva i razvoj i unaprjeđenje programske podrške namijenjene projektima u ovoj industriji. Programska podrška je iz tih razloga često kompleksna. Kako bi se programerima olakšao uvid u arhitekturu programskog rješenja i pomoglo pri otklanjanju potencijalnih pogrešaka, potrebno je razviti alat koji će omogućiti grafičku analizu kompleksnosti kôda. Korištenjem ovog alata moguće su uštede vremena i novca pri izradi projekata, te je moguće povećati i efikasnost i produktivnost programera.

U okviru ovog diplomskog rada predložen je alat za grafički prikaz arhitekture programskog rješenja. Alat se izrađuje na temelju podataka iz dokumentacije i na temelju podataka generiranih iz programskih izvršnih datoteka. Na taj način alat omogućava lakši pregled kompleksnosti programskog kôda i ispravljanje potencijalnih pogrešaka, poput nedostajućih ili krivo napisanih naziva pojedinih dijelova kôda.

Grafički izgled alata je kreiran pomoću *JavaScript* jezika uz korištenje dodatne biblioteke *D3.js*, a prikazuje dva grafa, od kojih jedan prikazuje arhitekturu programskog rješenja prema podacima iz dokumentacije, a drugi graf prikazuje arhitekturu prema generiranoj *map* datoteci. Opcijom usporedbe kreira se novi graf kao usporedba prethodna dva na kojemu se mogu uočiti razlike između njih te se prikazuje izvještaj koji prikazuje ukupni broj sučelja, broj ispravno implementiranih sučelja te broj nedostajućih sučelja. Prikupljanje i obrada podataka potrebnih za generiranje grafa izrađeno je u programskom jeziku *Python* uz biblioteke *pandas* i *difflib*.

Funkcionalnosti izrađene u ovom diplomskom radu rade fluidno i prilagođene su čovjeku. Hijerarhijskim prikazom i navigacijom po čvorovima grafa omogućuju korisniku analizu kompleksnosti kôda. Usporedni graf omogućuje uvid u potencijalne pogreške kao što su pogrešno napisani nazivi sučelja u programskom kôdu, odnosno njihovo odstupanje od naziva u dokumentaciji te uvid u nedostajuća sučelja, odnosno sučelja koja nisu implementirana u programskom rješenju, a navode se u dokumentaciji kôda. Funkcionalnosti su ilustrirane na tri jednostavna primjera koja prikazuju moguće scenarije korištenja alata.

Moguće nadogradnje predloženog alata su mogućnosti označavanja područja od interesa na grafu ili opcija povećavanja prikaza, koje bi olakšale prikaz kada je broj programskih modula, odnosno čvorova na grafu, izrazito velik. Također, moguće je prilagoditi alat kako bi radio sa

drugačijim tipom ulaznih podataka, odnosno kako bi mogao čitati dokumentaciju zapisanu u nekom drugom formatu ili koristiti druge generirane datoteke.

LITERATURA

[1] P. Oreški, »Modularnost i kvaliteta softvera,« *Journal of information and organizational sciences*, svez. 15, pp. 161-171, 1991..

[2] M. B. Andreas Schreiber, »Interactive Visualization of Software Components,« u *IEEE Working Conference on Software Visualization*, Njemačka, 2017..

[3] E. J. C. Chikofsky, »Reverse Engineering and,« *IEEE Computer Society*, pp. 13-17, 1990.

[4] H. A. Hind Alamin, »Concerns-Based Reverse Engineering for Partial Software,« *International Journal on Informatics Visualization*, svez. 4, 2020.

[5] R. A. Khan, »Extracting Executable Architecture From Legacy Code Using Static Reverse Engineering,« u *ICSEA*, Atena, 2017.

[6] W. Ulrich, »Modernization Standards Roadmap,« u *Information Systems Transformation*, MK/OMG Press, 2010, pp. 45-64.

[7] S. Frank, *Techniques for Visualization and Interaction in Software Architecture Optimization*, Stuttgart, 2019..

[8] I. P. J. V. Matej Ferenc, »Collaborative Modeling and Visualization of Software Systems Using Multidimensional UML,« u *IEEE Working Conference on Software Visualization*, Bratislava, 2017.

[9] »D3.js - Data-Driven Documents,« [Mrežno]. Available: https://d3js.org/. [Pokušaj pristupa 16. 9. 2020.].

[10] »Pandas,« [Mrežno]. Available: https://pandas.pydata.org/. [Pokušaj pristupa 16 9 2020].

[11] »difflib — Helpers for computing deltas,« [Mrežno]. Available: https://docs.python.org/3/library/difflib.html. [Pokušaj pristupa 16 9 2020].

SAŽETAK

Programska podrška koja se razvija za automobilske sustave je najčešće izrazito kompleksna i sastoji se od brojnih programskih modula. Alati koji omogućuju prikaz kompleksnosti programskog kôda moraju čovjeku omogućiti analizu arhitekture kôda i uvid u potencijalne pogreške, te tako povećati efikasnost programera i softverskih arhitekata te u konačnici omogućiti uštedu vremena i novca. Postojeća rješenja i izrađeni alati obično su razvijeni za specifičnu namjenu te često nisu dovoljno automatizirani. Podložni su optimizaciji i poboljšanjima jer je programska podrška za koje su ti alati namijenjeni sve složenija, a time je i alate potrebno nadograđivati. Alat predstavljen u ovom radu se temelji na podacima dobivenim iz dokumentacije programskog kôda čija se arhitektura želi prikazati i podacima izvučenim iz generirane *map* datoteke programskog kôda. Dokumentacija programskog kôda izrađena je u *JSON* formatu, koji je pogodan za prikaz u internetskom pregledniku. Izvlačenje podataka iz generiranih *map* datoteka se obavlja pomoću programskog jezika *Python* i biblioteka *pandas* i *difflib*. Grafički prikaz izrađen je pomoću programskog jezika *JavaScript* i biblioteke *D3*.js te uz korištenje *HTjML*-a i *CCS*-a. Alat omogućuje paralelan prikaz grafova koji opisuju isti programski modul na način da je jedan graf generiran na temelju podataka iz dokumentacije, a drugi na temelju podataka iz *map* datoteke. Usporedba grafova je najvažnija funkcionalnost ovog alata, a omogućava prikazivanje razlika između grafa nastalog iz dokumentacije i grafa nastalog iz *map* datoteke. Na taj način omogućava uočavanje potencijalnih pogrešaka te generiranje odgovarajućeg izvještaja. Funkcionalnosti alata prikazane su na tri primjera koji predstavljaju realne scenarije moguće pri korištenju ovoga alata.

Ključne riječi: automobilska programska podrška, kompleksnost kôda, alat za grafički prikaz, arhitektura programskog rješenja

ABSTRACT

A tool for visualization of the executable code complexity

The software developed for automotive systems is usually extremely complex and consists of numerous software modules. Tools that enable the visualization of the complexity of program code must provide code architecture analysis and insight into potential errors, and thus increase the efficiency of programmers and software architects and ultimately save time and money. Existing solutions and manufactured tools are usually developed for a specific purpose and are often not sufficiently automated. They are subject to optimization and improvements because the software for which these tools are intended is becoming more complex, and thus the tools need to be upgraded. The tool presented in this paper is based on the data obtained from the program code documentation whose architecture has to be displayed and the data extracted from the generated map file. The program code documentation is made in *JSON* format, which is suitable for viewing in a web browser. Extraction of data from generated map files is done using the *Python* programming language and the *pandas* and *difflib* libraries. The visualization was created using the *JavaScript* programming language and the *D3.js* library, as well as using *HTML* and *CCS*. The tool allows parallel display of graphs describing the same program module in such a way that one graph is generated with the data from the documentation and the other one with data from the map file. Graph comparison is the most important functionality of this tool, and it shows the differences between the graph created from the documentation and the graph created from the *map* file. In this way, tool enables the detection of potential errors and it generates the corresponding report. The functionalities of the tool are shown in three examples that represent realistic scenarios possible when using this tool.

Keywords: automotive software, code complexity, graphical display tool, software solution architecture

ŽIVOTOPIS

Ana Udovičić rođena je 5. veljače 1996. godine u Osijeku. Završila je Osnovnu školu Ivana Filipovića u Osijeku. Nakon toga upisuje II. gimnaziju u Osijeku koju završava 2015. godine. Iste godine upisuje redovni preddiplomski sveučilišni studij Računarstvo na Fakultetu Elektrotehnike, Računarstva i Informacijskih Tehnologija koji završava 2018. godine. Trenutno pohađa drugu godinu Diplomskog sveučilišnog studija Računarstvo, izborni blok Robotika i umjetna inteligencija.

_____

Potpis autora