Razvoj okvira generatora testnog okruženja u programskom jeziku C++

Garmaz, Filip

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

Sveučilišni studij

RAZVOJ OKVIRA GENERATORA TESTNOG OKRUŽENJA U PROGRAMSKOM JEZIKU C++

Diplomski rad

Filip Garmaz

Osijek, 2021.

Sadržaj

1. UVOD

U današnjim automobilskim industrijama, pri izradi vozila, automobilske firme sve se više oslanjaju na elektroniku u vozilima za optimalno upravljanje motorom, korisničko sučelje, sustav kočenja (ABS), mjerenje razine goriva, itd. Navedene funkcionalnosti izvode se na specijaliziranim ugradbenim računalnim sustavima, koje se nazivaju elektroničke upravljačke jedinice, (engl. *Electronic Control Unit* – ECU). Tijekom izrade ECU javlja se potreba za testiranjem ispravnosti rada svih komunikacijskih kanala, kako unutar ECU, tako i prema ostatku automobila. Danas, uz tehnološki razvijenu arhitekturu i male dimenzije čipova, ECU su postali jako složeni, a broj komunikacijskih kanala može biti više desetaka tisuća. Kao posljedica toga, ručno pisanje programa za testiranje svakog kanala vrlo je vremenski zahtjevan i naporan posao. Kako bi se skratilo vrijeme testiranja komunikacijskih kanala, javlja se potreba za testnim okruženjem, koje bi automatiziralo proces izrade programa za testiranje komunikacijskih kanala na ECU. Generiranje testnog okruženja svodi se na tri glavna zadatka:

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

Pomoću testnog okruženja omogućuje se izrada testova bez potrebe za korištenjem stvarnog hardvera i riskiranjem mogućih kvarova. Prema tome, dobro izrađeno testno okruženje od velike je važnosti jer ukoliko nepouzdano simulira ECU automobila može doći do katastrofalnih kvarova pri implementaciji koda na stvarnome ECU [1].

Zadatak ovog rada je izraditi okvir (engl. *Framework*) za generator testnog okruženja (engl. *Test Environment Generator* – TEG) u programskome jeziku C++. Razlog odabira programskog jezika C++ je njegova brzina izvođenja i mogućnost izravnog upravljanja računalnim resursima što omogućava programiranje na niskoj razini. Nadalje, C++ ima mogućnosti apstrakcije koda pomoću klasa te tako omogućava i programiranje na visokoj razini.

Okvir u sklopu generatora testnog okruženja nudi skup alata za razvoj ostalih dijelova generatora na efikasan i siguran način. Pri tome okvir ima neke od značajki, kao što su:

- Mogućnost detekcije svih datoteka u željenoj radnoj mapi, na osnovu zadanih djelomičnih putanja,
- Mogućnost rada u višestrukim nitima (engl. *Multithreading*),
- Mogućnost rada u višestrukim procesima (engl. *Multiprocessing*),

- Mogućnost bilježenja stanja programa na ekran i u datoteku (engl. *Logging*),
- Mogućnost rukovanja pogreškama u programu, tj. detekcija pogreški te poduzimanje različitih akcija za detektiranu pogrešku.

Okvir je namijenjen isključivo programerima i inženjerima TEG okvira za učinkovit razvoj testnog okruženja. Korištenje TEG okvira programeru olakšava razvoj generatora jer ne mora voditi računa o rukovanju resursima na računalu.

Rad je strukturiran na slijedeći način. U drugom poglavlju najprije je definirana potrebna terminologija i opisana su potrebna znanja i alati. Zatim su naznačeni nedostaci postojećeg rješenja i istaknuti su dijelovi rješenja koji se mogu unaprijediti. U trećem poglavlju dan je prijedlog rješenja TEG okvira te detaljan opis svih značajki, funkcionalnosti i modula. Uz predložena rješenja priložena je njihova implementacija pomoću dijagrama toka. U četvrtom poglavlju objašnjen je postupak testiranja i validacije rada svih implementiranih značajki te su priloženi rezultati svakog testa uz pojašnjenje dobivenih rezultata. Na kraju ovoga rada dan je zaključak u kojem se nalazi osvrt na cilj ovoga rada i postignutih rezultata. Također dane su preporuke za primjenu programskog rješenja u ovome radu te moguća daljnja poboljšanja okvira.

2. GENERATOR TESTNOG OKRUŽENJA

Dostupni generator testnog okruženja, odnosno TEG, je alat razvijen u programskom jeziku Python (verzija 2), kojeg je prvotno razvila firma TTTech. Zasniva se na AUTOSAR okruženju te pomoću njega se testiraju komunikacije ECU-ova u automobilskim sustavima.

2.1 AUTOSAR

Automotive Open System Architecture – AUTOSAR je svjetski poznata razvojna zajednica raznih automobilskih industrija i ostalih industrija na područjima elektrotehnike i razvoja softvera. Cilj AUTOSAR zajednice je standardizirati arhitekturu softvera, koji se razvija za ECU-ove. Standardizacijom se postiže razvoj softvera koji je nezavisan o hardveru te je moguće isti softver koristiti na više hardverskih platformi i na vozilima različitih proizvođača. Svrha toga je poboljšati kvalitetu i učinkovitost elektroničkih komponenata u vozilu, a time i smanjiti troškove ponovnog dizajniranja i osmišljavanja programskih rješenja za svaki hardver zasebno.

2.1.1 AUTOSAR format za razmjenu podataka

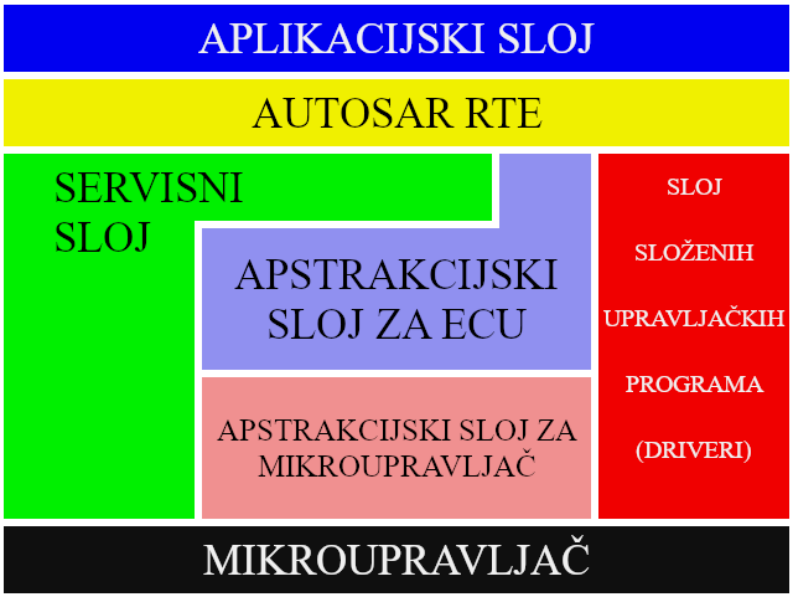
AUTOSAR definira standardni format za razmjenu podataka imena *AUTOSAR Schema*, koji je po svojoj strukturi sličan XML formatu. Ovakav format prema AUTOSAR terminologiji još je poznat i kao AUTOSAR XML ili ARXML. Sadržaj ARXML-a čini baza podataka koja sadrži informacije o ECU, tj. sve njegove dijelove i karakteristike njegovih ulaznih i izlaznih signala. Informacije koje sadrži ARXML format mogu se podijeliti u tri kategorije [2]:

- Ograničenja sustava (engl. *System Constraints*) – slikoviti prikaz komunikacijske mreže unutar vozila,
- Softverske komponente (engl. *Softvare Component, SWC*) – uključuje detalje ECU-a, kao što su tipovi podataka signala, koji sudjeluju u komunikaciji te detalji o međusobnoj povezanosti više SWC-ova,
- Detalji o resursima ECU-a (engl. *ECU Resources*) – detalji specifični za hardver, kao što su jedinice za obradu podataka, radna memorija, vanjske jedinice (engl. *Peripherals*) i detalji o nožicama (pinovima) hardvera.

2.1.2 AUTOSAR arhitektura

AUTOSAR arhitektura je slojevita. Glavni cilj ovakve slojevite arhitekture jest apstrakcija hardverskih komponenata od aplikacijskog sloja. Slojevita arhitektura sastoji se od tri glavna sloja, kako je prikazano na slici 2.1:

- Sloj osnovnog softvera (engl. *Basic Software Layer* – BSW) – upravlja resursima hardvera i omogućava dijeljene servise za aplikativni softver,
- *Runtime Environment* – RTE – sloj između aplikacijskog sloja i BSW-a, omogućava apstrakciju aplikacijskog sloja od BSW-a. Pomoću ovog sloja moguća je razmjena podataka između više SWC-a. Također, prema [1] TEG se nalazi u ovome sloju, budući da generira virtualno testno okruženje pomoću informacija konkretnog hardvera ECU-a kojeg parsira iz ARXML datoteke,
- Aplikacijski sloj (engl. *Application Layer*) – najviši sloj, služi za razvoj aplikacija, koje su specifične za određeni ECU. Aplikacije su sačinjene od većeg broja SWC-a, a pojedini SWC čini najjednostavniju formu funkcionalnosti aplikacije.



Slika 2.1: *Pojednostavljena AUTOSAR slojevita arhitektura. Svi blokovi između blokova „*Mikro upravljač*“ i „*AUTOSAR (RTE)*“ odnose se na BSW.*

2.2 Princip rada generatora testnog okruženja

Krajnji zadatak TEG-a je izrada testnog okruženja, koje oponaša rad ECU-a, te izrada mnoštva datoteka izvornog koda u programskom jeziku C, koji predstavljaju pojedini testni slučaj (eng. *test case*) za ECU. Ulazni podaci, koje TEG treba parsirati, čine model testnog okruženja i pohranjeni su u ARXML datoteci. U ARXML datoteci nalaze se sve komponente potrebne za

izgradnju testnog okruženja. Na temelju pročitanih (parsiranih) podataka iz ARXML datoteke generiraju se datoteke C koda, pomoću kojih je moguće ispitati svaki kanal ECU-a. Tijekom rada, TEG generira četiri testna okruženja, propisana AUTOSAR standardom [3]:

1. RTE sučelje – služi za komunikaciju unutar ECU-a, omogućuje apstrakciju prema aplikativnom sloju,
2. Sučelje za CAN sabirnicu – služi za vanjsku komunikaciju među ECU-ima, ostalim elektroničnim komponentama i modulima u automobilu (npr. moduli za upravljanje vratima, jedinice za upravljanje klimom, itd.),
3. *Persistency* – testiranje postojanosti podataka u slučaju kvara ili gubitka napajanja,
4. Platformski testovi – hardver, dijagnostika, nadzor aktivnosti, itd.

Trenutno rješenje TEG-a radi na način da se prvo pronađe unaprijed definirane putanje u `config.ini` datoteci. Zatim se parsira ARXML datoteka i na temelju toga se izrađuje baza podataka koja predstavlja jednostavniji model strukture ECU-a sa kojim se može upravljati tijekom rada TEG programa. Nakon toga na temelju podatka u bazi se generiraju testne programske komponente odnosno SWC-i, testni slučajevi za navedena testna okruženja i testni kod za svaki signal na sabirnici. Svi generirani izlazi za testiranje predstavljaju kompletno testno okruženje, koje se može simulirati pomoću *CANoe* programa [4].

2.3 Nedostaci postojećeg rješenja

Kako je spomenuto, postojeće rješenje TEG-a i TEG okvira pisano je u skriptnom jeziku Python, verzija 2. Python jezik poznat je po tome što ima vrlo jednostavnu sintaksu pa je jednostavan za pisanje i čitanje izvornog koda. Također, kako je vrlo popularan jezik posjeduje veliki broj modula, koji se lako mogu preuzeti preko Interneta i jednostavno uključiti u izvorni kod. Kako je Python jezik bogat modulima gotovo za svaki problem postoji već napisan modul, kojega se lako može iskoristiti na bilo kojoj platformi, što je velika prednost u odnosu na ostale jezike. Međutim, iako Python posjeduje mnoge mogućnosti, postojeće rješenje napisano u Pythonu ima nedostatke.

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

Još jedna velika mana Pythona kao programskog jezika, koja također utječe na performanse programa, je mehanizam globalnog zaključavanja interpretera (engl. *Global Interpreter Lock* – GIL) [5]. GIL omogućava izvršavanje Python programa u samo jednoj niti, čak i kada postoji mogućnost rada u više niti na više procesorskih jezgara istovremeno. Kao rezultat toga, izvođenje Python programa u više niti je često sporije, nego u jednoj niti [6]. Postojeće rješenje TEG-a, koje je implementirano u Python jeziku koristi mehanizme pokretanja programa u više procesa (engl. *Multiprocessing*). Kao posljedica toga je nemogućnost korištenja dijeljene memorije, što omogućuju niti. Također, uporaba niti je vremenski i resursno ekonomičnija [7].

Nadalje, Python koristi sakupljač smeća (engl. *Garbage Collector*) kako bi olakšao pisanje koda bez potrebe za ručnim upravljanjem memorijskim resursima. Ipak njegovo korištenje nije poželjno za memorijski zahtjevne probleme, pogotovo za probleme koji se trebaju izvršavati na hardveru sa ograničenim resursima. Programer prilikom pisanja koda u Pythonu, nema izravnu i potpunu kontrolu pri rukovanju memorijom.

Ipak, arhitektura i performanse programskog jezika Python nisu jedini nedostatak. Trenutno rješenje također ima nedostataka u implementaciji te u izboru verzije Python jezika, kao i korištenje nekih njegovih alata, odnosno modula.

Prvi nedostatak trenutnog rješenja jest činjenica da je program pisan u verziji Pythona koja je zastarjela i više se ne ažurira. Kao posljedica toga, Python verzija 2 ne može koristiti novije i naprednije biblioteke, koje su napisane za noviju verziju Python jezika (verzija 3), nego se mora oslanjati na zastarjele biblioteke, koje se više ne ažuriraju i ne nadopunjuju se s novim funkcionalnostima, te mogu sadržavati pogreške (engl. *bug*).

Nadalje, drugi nedostatak trenutnog rješenja je da se za spremanje podataka u TEG-ovu bazu podataka koristi biblioteka *pickle* [8]. Biblioteka *pickle* je Pythonov modul, koji implementira binarne protokole za serijalizaciju i deserijalizaciju Pythonovih struktura podataka i objekata. Tijekom procesa serijalizacije Python objekta, njegova hijerarhija pretvara se u binarni tok podataka. Iako je *pickle* brz i jednostavan za korištenje, ipak nije najpouzdanije rješenje za spremanje podataka. Naime, biblioteka *pickle* je ranjiva na izmjene. Drugim riječima, podaci koji su pohranjeni pomoću *pickle* biblioteke mogu se neovlašteno čitati, te se mogu izmijeniti. Moguće je čak i nadodati vlastiti kod u *pickle* datoteku, koji se nesmetano može izvršiti tijekom učitavanja *pickle* datoteke tijekom rada Python programa, jer *pickle* biblioteka nema nikakve mehanizme detekcije i sprječavanja izmijenjenog sadržaja *pickle* datoteke [9]. Zbog ovog velikog nedostatka *pickle* biblioteke, dovoljno je da neovlašteni korisnik ima pristup *pickle* datoteci, u koju može dodati svoj kod i može doći do toga da Python program počne izvršavati zadatke za koje nije predviđen. Najčešće je takav kod zloćudan pa može uzrokovati štetu na računalu.

Treći nedostatak u implementaciji trenutnog rješenja je nemogućnost provjere cjelovitosti generiranog sadržaja. Drugim riječima, nema načina da se provjeri je li svaka stavka uzeta iz modela, što znači da postoji mogućnost da neke stavke mogu biti izostavljene iz modela. Ovaj nedostatak proizlazi iz činjenice da postojeći parser modela nije dinamičan, nego svaki put kada se ARXML datoteka ažurira novim sadržajem modela, također je potrebno nadograditi parser da može obrađivati nove stavke, odnosno pokrpati trenutne nedostatke koje ima (engl. *Patch*). Naravno, prilikom izrade zakrpi za nove stavke modela, postoji mogućnost da one stare ne budu pročitane.

Valja još napomenuti da postojeće rješenje nema mogućnost dinamičkog pronalaska potrebnih datoteka, koje TEG koristi kao ulaz. Sve putanje potrebnih datoteka ručno su upisane u jednu XML datoteku. Korištenjem dinamičke pretrage potrebnih datoteka bilo bi praktičnije jer bi u tome slučaju korisnik mogao pohraniti datoteke na bilo kojem mjestu datotečnog sustava umjesto da ih pohranjuje na točno određenom mjestu. Također postoji mogućnost da je potrebno obraditi nove datoteke sa kojima TEG još nije radio i mogućnost da neke od postojećih datoteka više nisu potrebne pa ih je potrebno ukloniti iz okruženja. Drugim riječima, svaki put kada se dodaje nova datoteka ili ako se uklanja, te promjene potrebno je ručno zapisati u pripadajuću XML datoteku, što je naporan proces te je takav ručni pristup podložan pogreškama.

Konačno, četvrti nedostatak u implementaciji trenutnog rješenja je da generirani kod, koji je nastao kao rezultat TEG-ove obrade ulaznih datoteka, previše je previše redundantan, odnosno nije optimiziran. Naime, u generiranom kodu postoji puno dijelova koji se nepotrebno ponavljaju i nepotrebno nadodavaju. Posljedica toga je da se prilikom izvršavanja generiranih kodova nepotrebno troši vrijeme na višestrukom ispitivanju pojedinog kanala na ECU.

2.4 Prijedlog novog rješenja

Budući da postojeće rješenje ima velik broj nedostataka, samo će neki od tih nedostataka biti riješeni u ovome radu pomoću TEG okvira. Za ostale nedostatke, koji se tiču parsiranja i generiranja koda, potrebno je implementirati dijelove TEG-a specijalizirane za rješavanje spomenutih nedostataka.

U ovome radu korištenjem programskog jezika C++ otkloniti će se nedostaci postojećeg rješenja vezanih uz performanse programa i upravljanja memorijom te nitima i procesima. Također, poboljšat će se pretraga datoteka implementacijom dinamičke pretrage izravno na datotečnom sustavu, bez upotrebe dodatnih konfiguracijskih datoteka. Ovakva dinamička pretraga koristit će mehanizme pretraga imena datoteka korištenjem djelomičnih imena i putanja.

3. IZRADA OKVIRA ZA GENERATOR TESTNOG OKRUŽENJA

TEG okvir je najniži sloj TEG-a. Zadatak TEG okvira je omogućiti apstrakciju upravljanja resursima na operacijskom sustavu, praćenje stanja programa i rukovanje mogućim pogreškama, nastalih tijekom izvršavanja TEG programa. Prema tome TEG okvir oslobađa programera od vođenja računa o resursima i potpuno ručnom rukovanju pogreškama. TEG okvir je neizostavan dio TEG-a, jer je neophodan za učinkovito i sigurno obavljanje zadaća TEG-a pa se svi ostali dijelovi programa oslanjaju na TEG okvir.

Kako bi TEG okvir u potpunosti zadovoljio postavljene zahtjeve potrebni su mu mehanizmi za upravljanje resursima sustava, kao što je upravljanje datotečnim sustavom, nitima i procesima. Također kako bi osigurao stabilan rad programa i oporavak od pogrešaka javlja se potreba za uvođenjem rukovatelja za rješavanje nastalih pogrešaka. Također, budući da se radi o okviru koji treba samostalno raditi bez upotrebe biblioteka treće strane, ovaj rad koristio se u većini slučajeva isključivo standardnim bibliotekama C++ programskog jezika.

Tijekom izrade TEG okvira, uzelo se u obzir da svaka veća funkcionalnost okvira bude razdvojena u zaseban modul. Tako se TEG okvir sastoji od ukupno šest modula:

1. TEG Framework (glavni modul TEG okvira),
2. Rukovatelj datotekama (engl. *File Handler*),
3. Bilježnik (engl. *Logger*),
4. Rukovatelj pogreškama (engl. *Error Handler*),
5. Upravitelj nitima (engl. *Thread Manager*),
6. Upravitelj procesima (engl. *Process Manager*).

Ovakav način implementacije omogućuje modularnost okvira te uključivanje pojedinih modula okvira u izvorni kod po potrebi.

3.1 Korišteni alati i tehnologije

3.1.1 Programski jezik C++

C++ je programski jezik opće namjene i glavni je programski alat koji se koristi za izradu nove verzije TEG-a. C++ se najčešće koristi za sustavno programiranje (engl. *system programming*) i za ugradbene računalne sustave, kao što je npr. ECU. Osim toga C++ ima mogućnosti apstrakcije koda pomoću klasa te tako omogućava programiranje na visokoj razini. Nadalje, C++ ostvaruje velike brzine jer se sav kod napisan u C++ programskom jeziku izravno prevodi u strojni jezik.

Prevedeni strojni jezik se tada može izravno izvršavati na hardveru. Zbog svoje brzine i visoke apstrakcije, odabran je kao programski jezik za izgradnju novog rješenja TEG programa.

Prvo izdanje C++ programskog jezika 1985. godine izradio je Bjarne Stroustrup. Budući da se C++ koristi mnogo godina od njegovog početnog izdanja, do danas je razvijeno mnogo novih verzija i standarda. Za ovaj rad koristi se C++ verzija 17, odnosno C++17. C++17 je relativno moderan standard, te ima značajke u skladu sa novijim računalnim arhitekturama i operacijskim sustavima. Neke od C++17 značajka koje se koriste za izradu TEG okvira su npr. podrška za višenitni rad i rad sa datotekama i mapama.

3.1.2 Biblioteka Boost

Biblioteka *Boost* je jedna od najrazvijenijih biblioteka programskog jezika C++. Dizajnirana je s naglaskom da radi uz standardne biblioteke, s ciljem proširivanja mogućnosti C++ programskog jezika, te da se može koristiti na bilo kojoj platformi. Zbog svoje univerzalnosti lako je prenosiva i može se koristiti u bilo kojem projektu.

U ovome radu se *Boost* biblioteka koristi za upravljanje procesima. Svaki operacijski sustav ima drugačiji mehanizam rukovanja procesima što rezultira višestrukim pisanjem izvornog koda, koji je specifičan za pojedini operacijski sustav. Upravo se biblioteka *Boost* koristi kako bi se izbjeglo dupliciranje koda, tj. kako bi se jednom napisan kod mogao koristiti na bilo kojem operacijskom sustavu.

3.2 Dizajn i implementacija bilježnika

Bilježnik (engl. *Logger*) je najvažniji dio TEG okvira. Njime se koriste ostali moduli unutar TEG okvira te svi ostali dijelovi TEG-a. Glavna svrha bilježnika je omogućiti jednostavan i informativan način bilježenja trenutnog stanja programa, te na taj način omogućiti programeru uvid u tijek izvršavanja programa. Pomoću bilježnika moguće je ispisivati poruke na ekran i u datoteku. Ime i mjesto datoteke definiraju se po izboru, a može se bilježiti i bez korištenja datoteke, odnosno samo na ekran. U bilježniku postoji mehanizam razlikovanja važnosti poruka koje se trebaju ispisati. Važnost poruke predstavlja koliko je nužno da program obavijesti programera o stanju u kojem se nalazi ili o događaju koji je nastao. Također, pomoću razina važnosti, poruke se mogu filtrirati na način da se manje važne poruke zanemaruju, a one važnije da se ispisuju. Razine važnosti određuju se posebno za ispis na ekran, a posebno za zapis u datoteku. Tako se primjerice može definirati da se sve poruke zapisuju u datoteku, dok se važnije poruke ispisuju na ekran.

Bilježnik posjeduje 5 razina ozbiljnosti:

1. DEBUG – na ovoj razini ispisuju se poruke koje mogu sadržavati određene rezultate pojedinih međukoraka u cilju što lakšeg otklanjanja pogrešaka u kodu tijekom razvoja softverskog rješenja,
2. INFO – razina namijenjena za ispisivanje obavijesti programa, te informiranje programera
3. WARNING – razina namijenjena za ispisivanje upozorenja, koja ukazuju na moguće pogreške u programu, ali uz koje program može nastaviti sa radom,
4. ERROR – razina namijenjena za ispisivanje pogrešaka koje su nastale tijekom rada programa. Ovakve poruke ukazuju da se dio programa ne može izvršiti te da je potrebna adekvatna reakcija programera. Ovakvim porukama koristi se rukovatelj pogreškama, kada je potreban oporavak od ovakvih pogrešaka,
5. CRITICAL – najozbiljnije pogreške, koje onemogućavaju rad programa i oporavak, a ponekad i nasilno prekidaju rad program.

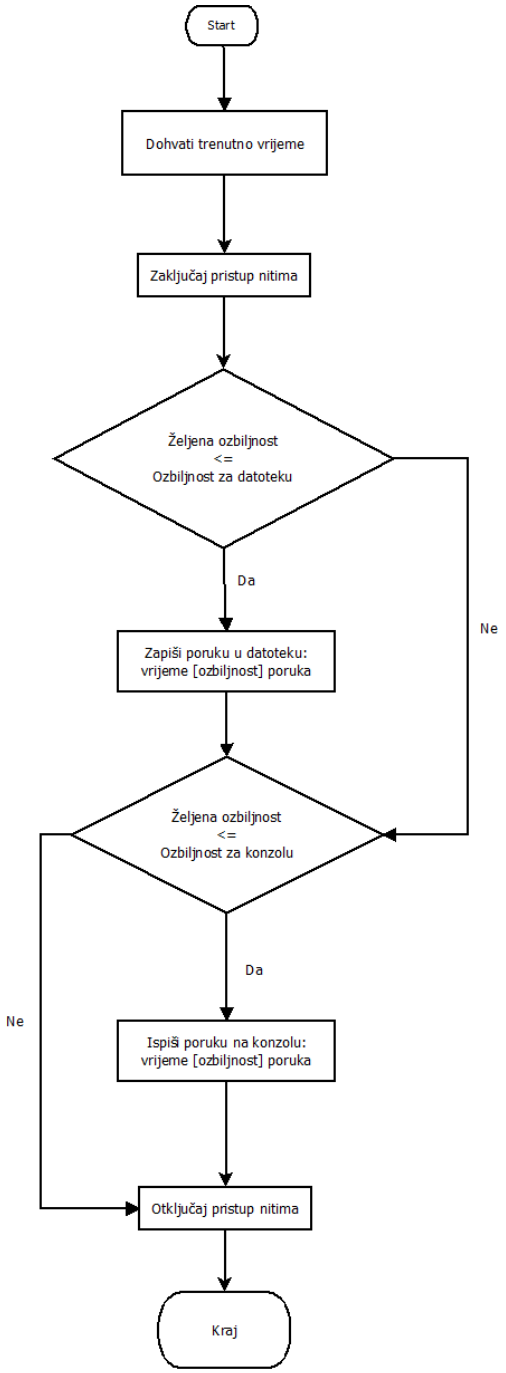
Tijekom implementacije bilježnika uzimalo se u obzir jednostavnost korištenja i dostupnost kroz cijeli program, ali također i sigurnost pristupa metodama i atributima, kako bi se izbjegli mogući slučajevi nepažljivih izmjena kritičnih dijelova bilježnika. Također, uzimalo se u obzir automatsko oslobađanje memorijskih resursa nakon završetka rada programa kako bi program pravilno i uredno završio s radom. Kako bi se osiguralo oslobađanje resursa i jednostavnost korištenja modula, za implementaciju je odabran *singleton* obrazac. *Singleton* je stvarajući obrazac dizajna (engl. *design pattern*) programskog koda, koji osigurava da u bilo kojem trenutku tijekom rada programa postoji samo jedan jedinstveni objekt. Takav objekt je raspoloživ preko samo jedne točke pristupa, preko bilo kojeg mjesta u programu [10]. Upravo pomoću *singleton* obrasca moguće je ispuniti sve navedene zahtjeve.

Za bilježnik je također potrebno da ima mogućnost višenitnog rada, a da pri tome ne dolazi do konflikata tijekom rada nad zajedničkim resursom (datoteka, standardni izlaz) istovremeno. Kako bi se spriječila asinkronost u slučaju višenitnog rada, koriste se muteksi (engl. *mutex*), koji omogućavaju siguran rad pa čak i u više niti (engl. *thread safety*). Na slici 3.1 dan je blok dijagram algoritma za bilježenje u datoteku i na ekran.

Tijekom izvršavanja algoritma za bilježenje najprije se dohvaća trenutno vrijeme na način da se najprije navede datum, a onda vrijeme, tj. sat, minuta i sekunda zaokružena na tri decimale. Nakon dohvaćanja vremena zaključavaju se niti. Ovaj mehanizam osigurava da točno jedna nit izvršava kritičan dio koda. Druge niti pri tom moraju čekati dok zaključana nit ne oslobodi kritičan dio koda. Kritičan dio koda predstavlja bilježenje. Bilježenje se provodi na način da se uspoređuju razine ozbiljnosti ispisa na ekran i u datoteku sa razinom ozbiljnosti poruke. Ukoliko je razina

ozbiljnosti poruke manja od ozbiljnosti ispisa na ekran ili u datoteku, tada će program zanemariti ispis poruke na spomenuta odredišta. U suprotnom poruka se ispisuje na ekran ili u datoteku zajedno sa vremenom i razinom ozbiljnosti.

Za ostvarenje mogućnosti sigurnog rada u više niti, koriste se standardne C++ biblioteke *mutex* i *condition_variable*, dok se za dohvaćanje trenutnog datuma i vremena koriste biblioteke: *chrono*, *iomanip*.



Slika 3.1: *Blok dijagram algoritma za bilježenje poruka u datoteku i na standardni izlaz (ekran), kao i vremena ispisa poruke, te ozbiljnosti poruke.*

3.3 Dizajn i implementacija rukovatelja datotekama

Rukovatelj datotekama (engl. *File Handler*) je modul TEG okvira, koji obavlja radnje vezane za pristup datotekama u datotečnom sustavom. Pomoću rukovatelja datotekama moguće je manipulirati datotečnim sustavom te izvršavati jednostavnije operacije, kao što su otvaranje, zatvaranje, pisanje i čitanje datoteka, te učitavanje putanja radnih mapi, koje su potrebne TEG-u. Također pomoću rukovatelja datotekama mogu se postići složenije operacije, kao što je dinamička pretraga datoteka po sustavu korištenjem djelomičnih putanji i naziva datoteka. Operacije koje podržava rukovatelj datotekama su:

1. Rječito (engl. *Verbose*) otvaranje datoteka za čitanje,
2. Rječito otvaranje datoteka za pisanje i to u načinu za nadodavanje (engl. *Append*) ili za nanovo pisanje (engl. *Overwrite*),
3. Dinamička i dubinska pretraga datoteka po punom ili djelomičnom imenu u zadanoj putanji (najčešće radna mapa TEG programa),
4. Dinamička i dubinska pretraga datoteka po punom ili djelomičnom imenu uz korisnički definirane putanje mapa za pretragu,
5. Dinamička i dubinska pretraga datoteka prema djelomično zadanim putanjama, i
6. Mogućnost dodavanja i uklanjanja novih putanja mapa u listu za pretraživanje.

Isto kao i bilježnik, rukovatelj datotekama je također implementiran koristeći *singleton* obrazac. Time se postiže enkapsulacija članova tog modula, koji se nalaze unutar klase. Drugim riječima, očuva se objektno orijentirana paradigma, dok se postiže mogućnost korištenja istog objekta unutar cijelog programa, kao i u slučaju korištenja globalnih varijabli.

Rukovatelj datotekama implementiran je kako bi nadogradio ograničenu funkcionalnost upravljanja datotekama postojećeg rješenja. Predloženi rukovatelj datotekama implementiran je na način da rekurzivno pretražuje datoteke na samome datotečnom sustav u stvarnome vremenu, bez prethodnog čitanja unaprijed definiranih podataka iz datoteke. Prednost ovakvog pristupa je ta što je moguće pohraniti datoteke projekta u bilo kojem direktoriju. Samim time moguće je datoteke premještati, mijenjati im ime, stvarati nove datoteke potrebne za rad projekta, te uklanjati one datoteke koje više nisu potrebne, i to bez potrebe vođenja računa o izmjenama na datotečnom sustavu, kao što je bio slučaj u postojećem rješenju. Prema tome, nema potrebe za izradom zasebne XML datoteke, u kojoj su ručno definirane putanje do svih potrebnih datoteka. Dovoljno je dodati putanje za pretragu, ako postoje datoteke koje se ne nalaze u radnoj mapi programa.

Za realizaciju ovakvog rješenja u C++ programskom jeziku koristi se standardna C++ biblioteka *filesystem*. Biblioteka *filesystem* nudi osnovne funkcionalnosti za izvršavanje operacija

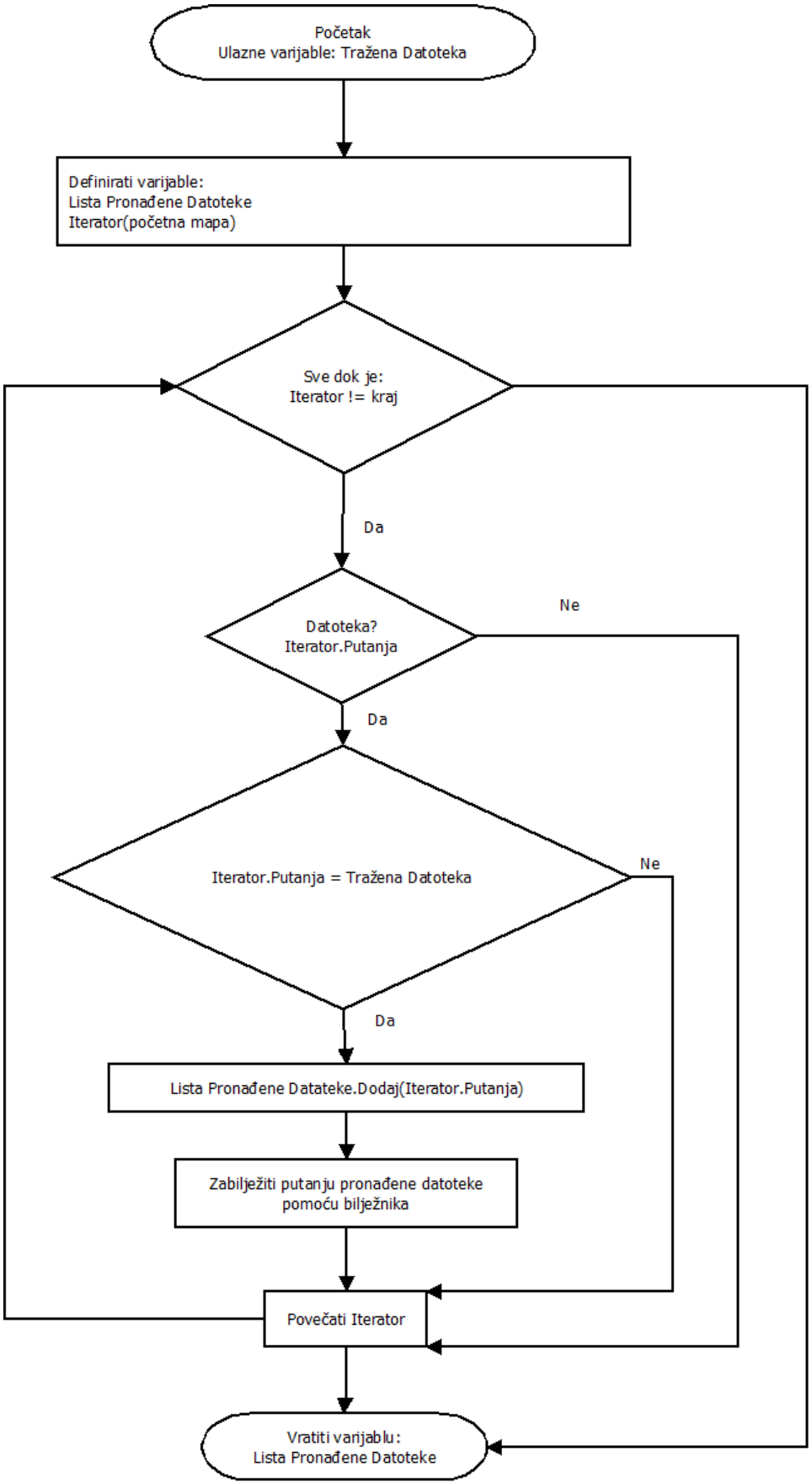
na datotečnom sustavu, tj. njegovim objektima, kao što su putanje, datoteke i mape [11]. Biblioteka *filesystem* omogućava slijedeće značajke:

1. Rad sa datotekama,
2. Rad sa mapama,
3. Upravljanje simboličnim poveznicama (engl. *symbolic link*),
4. Mogućnost manipuliranjem imenom datoteke i datotečnim nastavcima,
5. Upravljanje putanjama
    - Apsolutna putanja (engl. *absolute path*),
    - Kanonična putanja (engl. *canonical path*),
    - Relativna putanja (engl. *relative path*).

Jedna od najznačajnijih mogućnosti biblioteke *filesystem* je mogućnost rekurzivnog obilaska datotečnog sustava u dubinu. Ovakva mogućnost može se izvesti pomoću rekurzivnog iteratora, odnosno *recursive_directory_iterator*. Rekurzivni iterator je objekt, koji sadrži pokazivač na implementacijski objekt koji sadrži stog svih mapa koje još nisu otvorene, te brojač dubine u kojoj se rekurzija trenutno nalazi, odnosno koliko je mapa rekurzivni iterator otvorio [12].

Pomoću rekurzivnog obilaska datotečnog sustava u dubinu, moguće je u svakoj mapi i u onim mapama unutar polazne pohraniti datoteku. Iz datoteke moguće je očitati njezino puno ime (ime uz datotečni nastavak) i na taj način provjeriti odgovara li pročitano ime trenutne datoteke nazivu datoteke koju je potrebno pronaći. Također, uporabom liste znakovnih nizova (engl. *string*), moguće je pohraniti sve pronađene datoteke, koje odgovaraju traženoj datoteci, koje je rekurzivni iterator pronašao. Na slici 3.2 dan je blok dijagram jednostavnog algoritma pretrage datoteka. Algoritam započinje deklaracijom dviju varijabli, tj. deklaracijom samoga iteratora za obilazak datoteka i prazne liste u koju se pohranjuju sve pronađene putanje željenih datoteka. Tada algoritam pomoću iteratora u petlji pretražuje sve datoteke od početne mape i uspoređuje svaki naziv datoteke sa željenim nazivom. Ako se nazivi podudaraju pohranjuje cijela putanja do datoteke se pohranjuje u listu. Nakon obrade zadnje datoteke unutar početne mape algoritam završava sa radom i vraća listu putanji pronađenih datoteka.

Međutim, u mnogim slučajevima potrebna je pretraga koristeći općenitije izraze, umjesto točnog imena datoteke. Naime, postoji mogućnost da su potrebne npr. sve datoteke sa određenim datotečnim nastavkom, ili pak neke datoteke koje imaju isti početak imena (engl. *prefix*), dok se u drugim dijelovima razlikuju. U tome slučaju bilo bi potrebno znati točno svako ime datoteke sa njezinim datotečnim nastavkom, te ih pretraživati jednu po jednu, što nije ništa manje mukotrpno i vremenski zahtjevno od postojećeg rješenja u Python jeziku.



Slika 3.2: *Blok dijagram algoritma rekurzivne pretrage datoteka u dubinu unutar datotečnog sustava.*

Kako bi se riješio navedeni problem, potrebno je implementirati mehanizam za provjeru valjanosti djelomičnih naziva bilo kojeg imena datoteke ili mape. Ideja je da se pomoću posebnog znaka (engl. *wildcard characters*), zanemare dijelovi imena datoteka ili putanja koji ne utječu na odluku poklapanja traženog izraza sa imenom konkretne datoteke. Time, moguće je predati rukovatelju datotekama samo one dijelove naziva datoteka koji su mu od interesa, a ostale dijelove zamijeniti posebnim znakom. Izraz kojeg se navede bez posebnih znakova podrazumijeva se kao točno ime datoteke. Također, može se navoditi više posebnih znakova ako posebni znakovi nisu odmah jedan pokraj drugoga. Za posebni znak odabran je znak „*“ (zvjezdica, engl. *asterisk*), jer je uvelike rasprostranjen u mnogim programima koji rješavaju probleme pretrage (npr. Linux naredba *find*, [13] Windows Explorer, itd.). U tablici 3.1 dan je popis načina na koje je moguće koristiti posebne izraze, a koji daju različite rezultate. Postoji još mnogo mogućnosti, kako iskoristiti posebne znakove u nazivu. Može se primijetiti da su posebni znakovi moćan alat i imaju širok spektar uporabe, a koliko se mogu napredno iskoristiti, ovisi o samome programeru.

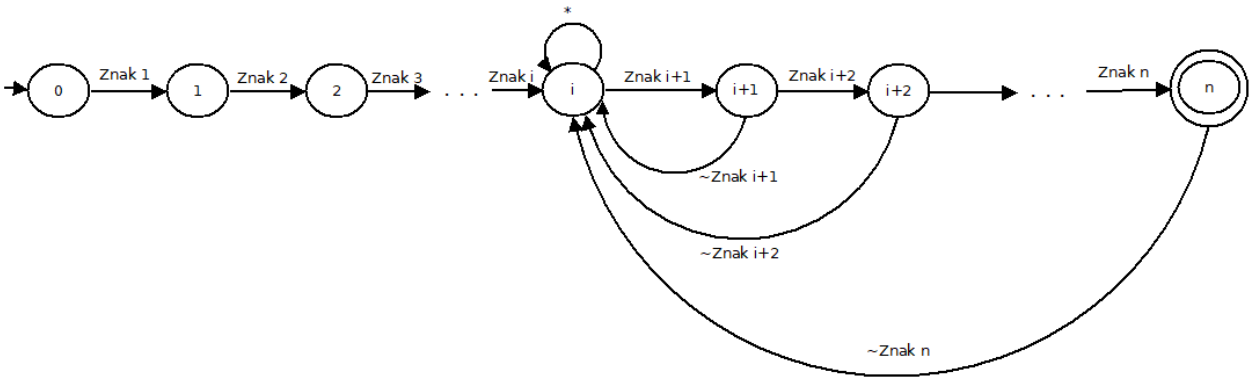
Čitanje, odnosno parsiranje djelomičnih naziva datoteka implementirano je kao funkcija, koja prima tri argumenta. Prvi argument je trenutno ime datoteke ili mape, u obliku niza znakova (engl. *string*), kojeg treba usporediti sa djelomičnim nazivom, tj. utvrditi zadovoljava li predano ime željeni djelomični naziv. Drugi argument upravo predstavlja djelomični naziv, također u obliku niza znakova koji može, ali ne mora sadržavati specijalne znakove, te on služi kao kriterij koji prvi argument treba zadovoljiti. Naravno, kako je spomenuto, ako djelomični naziv nema specijalnih znakova, gleda se kao puni naziv. Na kraju, treći argument predstavlja osjetljivost na velika i mala slova, on je logičkog, odnosno *bool* tipa. Pomoću osjetljivosti na slova određuje se treba li se tijekom čitanja imena trenutne datoteke ili mape, kao i čitanja djelomičnog naziva, razlikovati velika i mala slova. Kada osjetljivost na slova ima vrijednost istine, prilikom čitanja naziva velika i mala slova biti će različita, dok u slučaju kada je vrijednost laž odnositi će se isto prema slovima neovisno jesu li velika ili mala. Npr. ukoliko osjetljivost na slova je istinita, slijedeći nazivi biti će različiti iako znače isto:

$$\text{ECU} \neq \text{Ecu} \neq \text{eCu} \neq \text{ecU}$$

U suprotnome, ako je osjetljivost na slova laž, navedeni nazivi biti će isti.

Prilikom parsiranja imena datoteke ili mape, uobičajeno je koristiti automat sa konačnim brojem stanja (engl. *Deterministic Finite Automata* – DFA), pomoću kojeg se opisuje točan izraz koji se treba zadovoljiti. Međutim, DFA definira točno jedan izraz, odnosno gramatiku, dok u slučaju provjere imena pomoću djelomičnih naziva, sam naziv definira programer, te svaki put kada se čita ime, može biti drugačiji djelomični naziv, koji se opisuje u potpunosti drugačijim DFA. Da bi se riješio ovakav problem potrebno je pomoću zadanog djelomičnog naziva dinamički

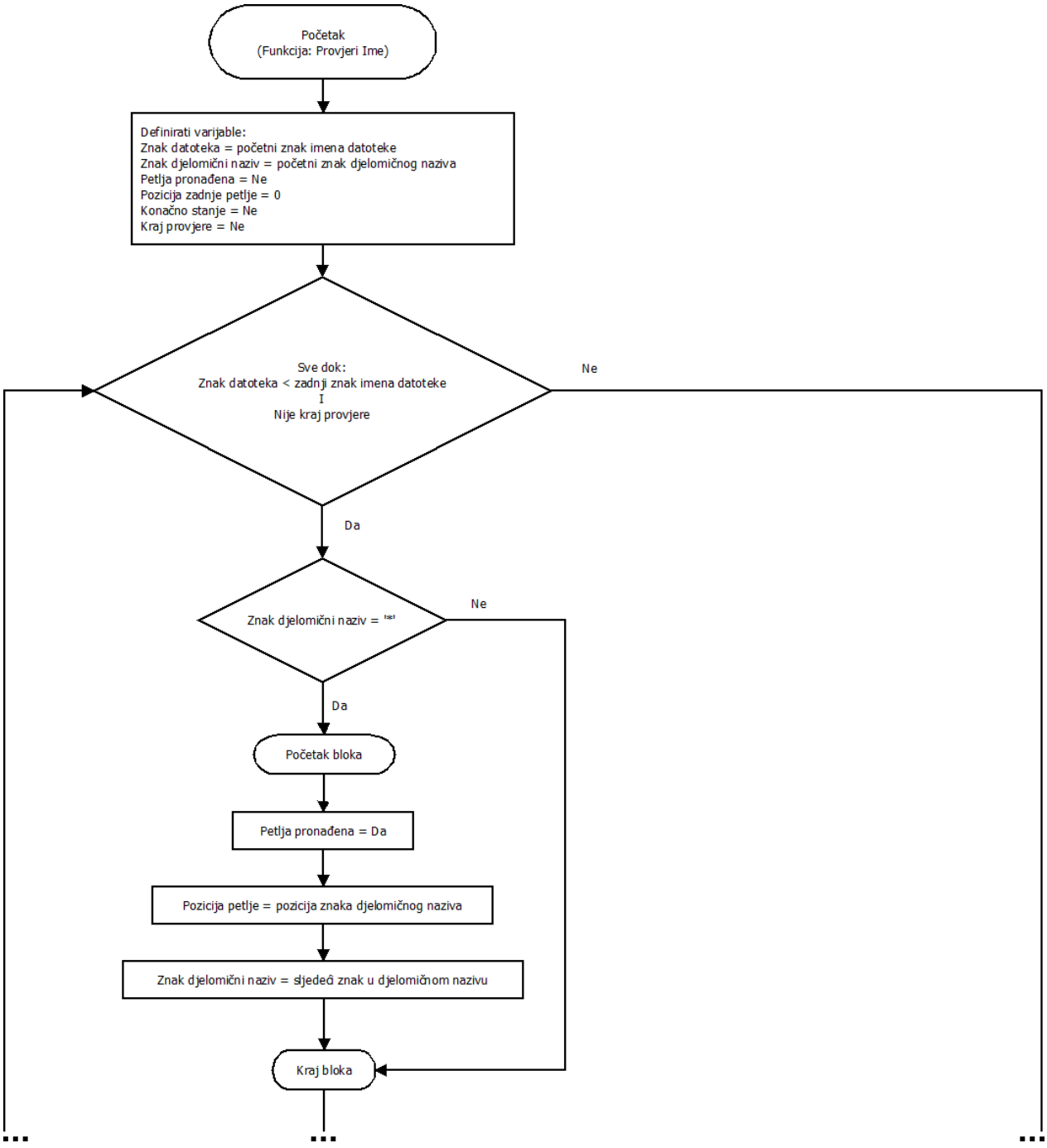
implementirati DFA za vrijeme rada programa. Algoritam radi na takav način da gleda pojedine znakove u djelomičnom nazivu kao prijelaze u nova stanja automata. Budući da broj stanja unaprijed nije poznat ovaj algoritam radi stanje po stanje u hodu dok čita novi znak unutar djelomičnog naziva. Najprije se čita znak po znak imena trenutne datoteke ili mape, pa se tada uspoređuje sa trenutnim znakom djelomičnog naziva. Ukoliko su znakovi isti tek tada se može pročitati slijedeći znak djelomičnog naziva, odnosno prijeći u slijedeće stanje dinamičkog automata. Budući da se ne zna unaprijed broj stanja, ipak postoji uvjet zaustavljanja čitanja. Naime, algoritam završava sa parsiranjem onda kada mu ponestane novih znakova iz imena ili iz djelomičnog naziva. Kako bi se ime podudaralo sa djelomičnim nazivom, oba naziva moraju iscrpiti sve znakove u isto vrijeme. Tada je automat došao do konačnog stanja, a i ne treba ništa više čitati, što rezultira ispravnim imenom datoteke ili mape. Iako je uvjet ispravnosti imena da oba niza znakova moraju doći do kraja u isto vrijeme, to ne mora značiti da trebaju imati isti broj znakova, te jednostavna provjera broja znakova ne jamči da je ime netočno. Razlog tome su upravo specijalni znakovi „*“, koji se tretiraju kao petlja u DFA, pa je ime datoteke ili mape gotovo uvijek veće od djelomičnog naziva. Kada se pročita znak „*“ iz djelomičnog naziva potrebno je zapamtiti poziciju tog znaka unutar djelomičnog naziva, jer ukoliko se nađe barem jedan slijedeći znak imena, koji ne odgovara slijedećem znaku djelomičnog naziva, potrebno je vratiti se na mjesto znaka „*“, te uspoređivati daljnje znakove trenutnog imena datoteke ili mape ponovo sa mjesta na kojem se pojavio znak „*“, sve dok ne ponestane znakova imena za čitanje. Na slici 3.3 dan je shematski prikaz DFA koji se gradi tijekom izvršavanja funkcije za provjeru valjanosti imena datoteke. Na slikama 3.4 i 3.5 dan se blok dijagram dinamičke izgradnje DFA. Budući da se može uspoređivati trenutno ime datoteke ili mape sa djelomičnim nazivom, umjesto da se uspoređuje sa točnim željenim imenom. Na slici 3.6. prikazan je blok dijagram sličan blok dijagramu na slici 3.2 koji umjesto izravne usporedbe koristi funkciju `Provjeri Ime`.



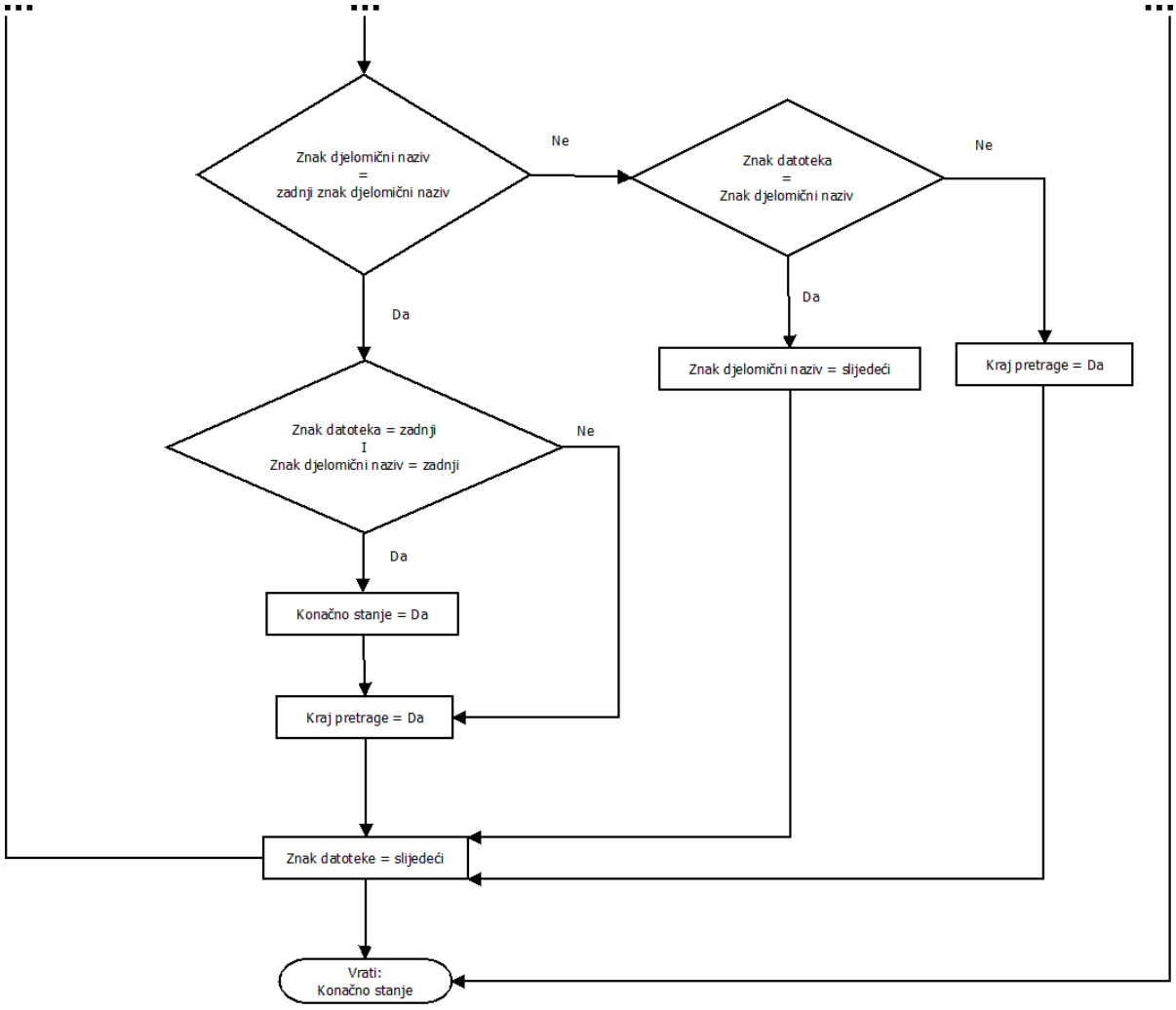
Slika 3.3: *Dinamička izgradnja automata sa konačnim brojem stanja na osnovu izraza djelomičnog naziva.*

Primjer izraza	*Primjeri ispravnih naziva za dani izraz*	*Opis*
*.xml	file1.xml dir1/file2.xml dir2/file3.xml	Pronalazi sve XML datoteke, koje se nalaze na bilo kojem mjestu u odnosu na radnu mapu.
file*.txt	file.txt file1.txt file2.txt dir3/file123.txt	Pronalazi sve datoteke, koje počinju nazivom „file“, a završavaju sa „.txt“ (sve tekstualne datoteke).
file*	file.txt file.xml dir1/dir2/file.exe	Pronalazi sve datoteke imena „file“, bez obzira na datotečni nastavak.
dir*subdir*file*.ini	dir1/subdir2/file1.ini dir2/subdir3/file2.ini dir3/subdir1/file3.ini dirx/x/subdiry/y/filez.ini dirfakesubdirfakefile.ini	Primjer djelomične putanje. Najprije se traži mapa koja počinje imenom „dir“. Nakon toga pronalazi se mapa koja počinje imenom „subdir“, a nalazi se unutar mape „dir*“. Konačno pronalazi se INI datoteka, koja počinje imenom „file“. Također, važno je napomenuti kako i složeni nazivi datoteke, koji u svom nazivu sadržavaju i „dir“, i „subdir“, i „file“, i „.ini“ mogu biti valjani.

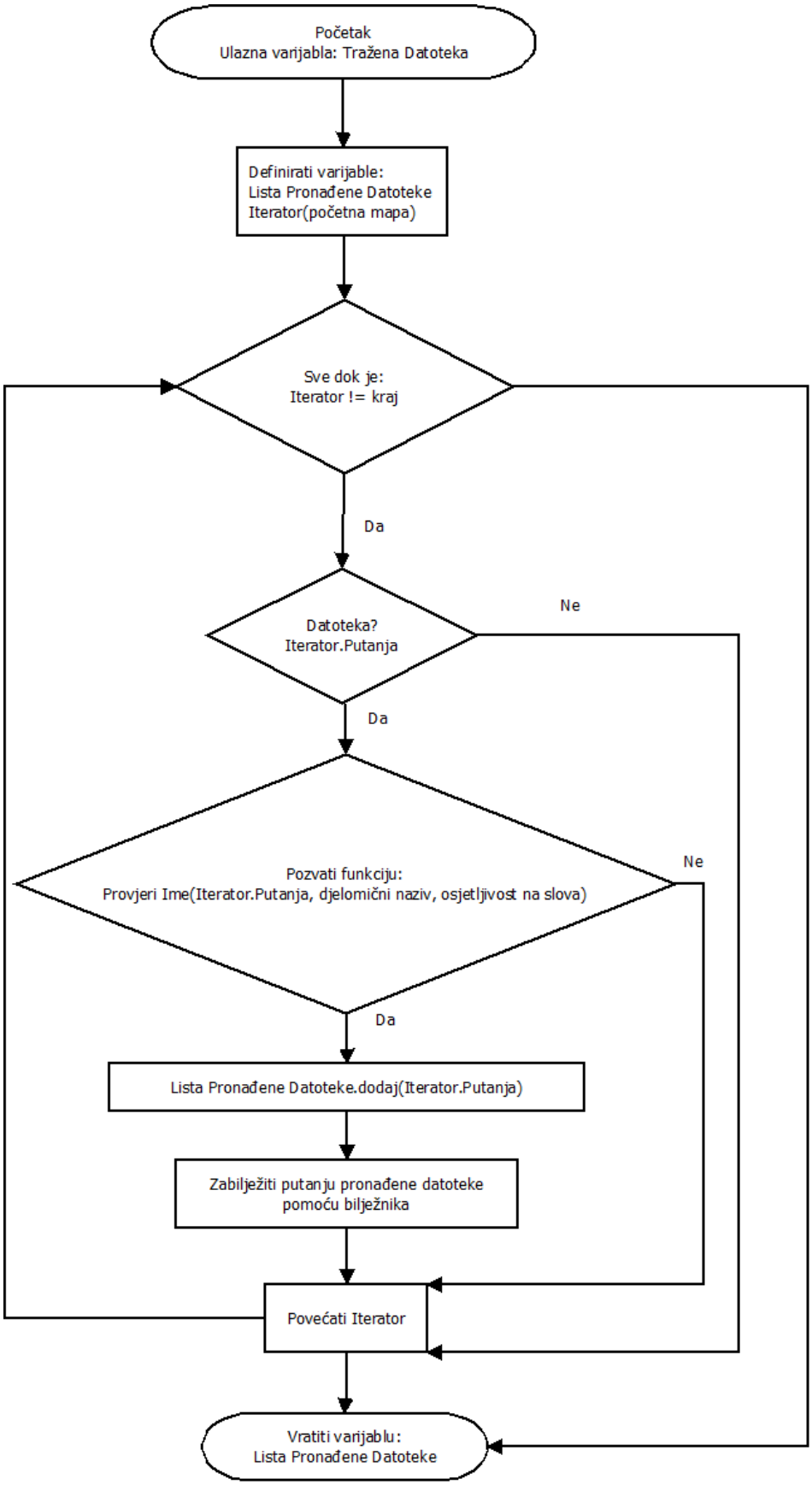
Tablica 3.1: *Popis mogućih načina korištenja posebnog znaka „*“.*



Slika 3.4: *Blok dijagram algoritma dinamičke validacije imena datoteke ili mape na osnovu djelomičnog naziva (prvi dio).*



Slika 3.5: *Blok dijagram algoritma dinamičke validacije imena datoteke ili mape na osnovu djelomičnog naziva (drugi dio).*



Slika 3.6: *Blok dijagram algoritma rekurzivne pretrage datoteka u dubinu korištenjem djelomičnih naziva.*

3.4 Implementacija rukovatelja pogreškama

Rukovatelj pogreškama također je jedan od modula TEG okvira. Glavna zadaća rukovatelja pogreškama je olakšati rukovanje pogreškama na način da se ne mora voditi računa o ručnoj izradi vlastitih klasa pogrešaka niti na kojem mjestu je potrebno rukovati pogreškom, odnosno kada je potrebno pokušati oporaviti sustav od nastale pogreške. Dovoljno je da se pozove rukovatelj pogreškama kako bi se registrirala vlastita pogreška sa vlastitim opisom, dok sve ostalo radi sam rukovatelj pogreškama. Nadalje, rukovatelj pogreškama ima automatizirani mehanizam bilježenja pronađenih pogrešaka pomoću bilježnika. Informacije o pogreškama se detaljno bilježe kako bi se programeru omogućio detaljan uvid u nastalu pogrešku te kako bi se ta pogreška učinkovito mogla otkloniti. Informacije koje se bilježe o pogreškama su opis pogreške, ime datoteke izvornog koda, linija u izvornom kodu i opseg (engl. *scope*) u kojem je pogreška nastala. Za implementaciju rukovatelja pogreškama koriste se biblioteke:

- *exception*
- *stdexcept*
- *vector*
- *map*

U navedenim bibliotekama nalaze se klase sa svim standardnim pogreškama kao i osnovne funkcionalnosti za rad sa iznimkama (engl. *exceptions*). Iznimke su poseban mehanizam kojeg posjeduje programski jezik C++, a koje nastaju kao reakcija na nastanak pogreške u izvornom kodu. Iznimka prekida normalan rad programa i omogućava način prijenosa upravljanja jednim djelom programa na drugi dio programa. Taj drugi dio programa je upravitelj pogreškama.

Rukovatelj pogreškama treba biti dostupan u bilo kojem dijelu koda, jer ostali dijelovi generatora testnog okruženja će ga često koristiti pogotovo za rukovanje mogućim pogreškama koje nastaju tijekom parsiranja ili tijekom generiranja datoteka. Zbog toga se također pri implementaciji rukovatelja pogreškama koristi *singleton* obrazac.

Sadržaji pogrešaka pohranjuju se u obliku dinamički alociranih listi u memoriju. Kako bi se sadržajima pogrešaka moglo pristupati, dodjeljuje im se jedinstveni brojevi (*ID*) te jedinstvena imena. Prema tome, moguće je pristupiti željenom sadržaju pogreške prema jedinstvenom broju ili prema jedinstvenom imenu pogreške. Jedinstveni brojevi pohranjeni su u obliku mape u kojoj je ključ jedinstveni broj, a vrijednost je pokazivač na sadržaj pogreške. Slično tome su pohranjena i jedinstvena imena, samo što mapa kao ključ koristi niz znakova, koji predstavlja jedinstveno ime. U isječku koda 3.1 dana je struktura podataka koja opisuje sadržaj pogreške, te liste i mape koje pohranjuju sadržaje pogreški.

```
struct SadrzajPogreske {
        int Reakcija;
        int Ozbiljnost;
        std::string Opis;
};

std::list<struct SadrzajPogreske> ListaPogresaka;
std::map<std::string, struct SadrzajPogreske*> ImenaPogresaka;
std::map<unsigned int, struct SadrzajPogreske*> IDPogresaka;
```

Isječak Koda 3.1: *Strukture podataka za pohranjivanje pogrešaka.*

Iako postoje strukture podataka, koje sadržavaju sve potrebne informacije o pogreškama, ipak na ovakav način pogreške se ne mogu tretirati kao iznimke. U tome slučaju implementira se prilagođena klasa, koja nasljeđuje standardnu klasu za iznimke tijekom rada programa (engl. *runtime error*), odnosno klasa `std::runtime_error` iz biblioteke *stdexec*. Za vrijeme registriranja prilagođene klase pogrešaka, za što informativniji opis klasa, konstruktor treba primiti četiri argumenta za vrijeme konstrukcije, a to su: opis pogreške, datoteka izvornog C++ koda u kojoj je nastala pogreška, linija u kojoj je nastala pogreška i naziv funkcije, odnosno doseg (engl. *scope*) u kojemu je nastala pogreška. Na slici 3.7 prikazan je blok dijagram za inicijalizaciju prilagođene klase za iznimke.

Samu klasu iznimki nije moguće izravno registrirati naredbom `throw`, nego pomoću pomoćne metode. Ova metoda prima sadržaj pogreške, a ostali argumenti su slični argumentima konstruktora prilagođene klase iznimki, odnosno: datoteka izvornog C++ koda u kojoj se nastala pogreška, linija u kojoj je nastala pogreška i naziv funkcije u kojoj je nastala pogreška. Na slici 3.8 dan je blok dijagram pomoćne metode za registriranje pogrešaka.

Može se primijetiti da pozivanjem ove metode na slici 3.8 programer mora ručno unositi putanju do datoteke izvornog koda, liniju i naziv funkcije, što može biti otežano zbog čestih izmjena izvornog koda. Ovakvo pozivanje često je podložno pogreškama i ako se ručno ne ažurira npr. broj linije, može doći do pogrešnog ispisa mjesta pogreški za vrijeme rada programa. Ipak, korišteni C++ prevoditelj (g++) ima posebne makro izraze, koji automatski postavljaju putanje, linije i nazive funkcije tijekom prevođenja izvornog koda u program. To su:

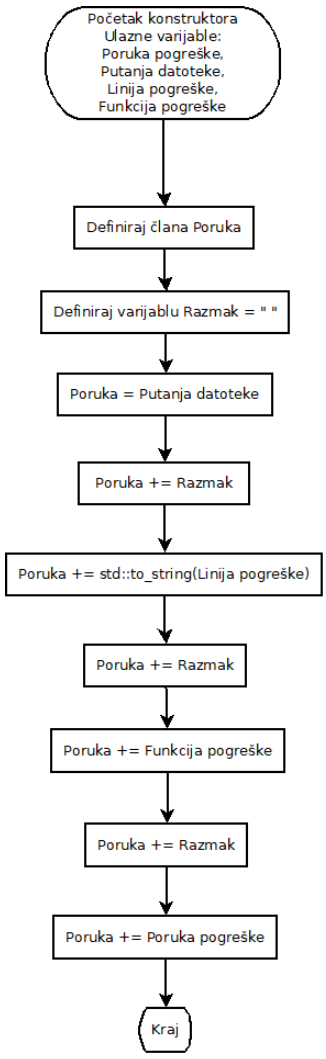
- `__FILE__` – daje informaciju prevoditelju da na mjesto gdje je postavljen ovaj makro izraz zamijeni sa putanjom do datoteke izvornog koda, u kojoj se nalazi taj makro izraz,
- `__LINE__` – daje informaciju prevoditelju da na mjesto gdje je postavljen ovaj makro izraz zamijeni sa linijom, na kojoj se nalazi taj makro izraz,
- `__FUNCTION__` – daje informaciju prevoditelju da na mjesto gdje je postavljen ovaj makro

izraz zamijeni sa nazivom funkcije, unutar koje se nalazi ovaj makro izraz.

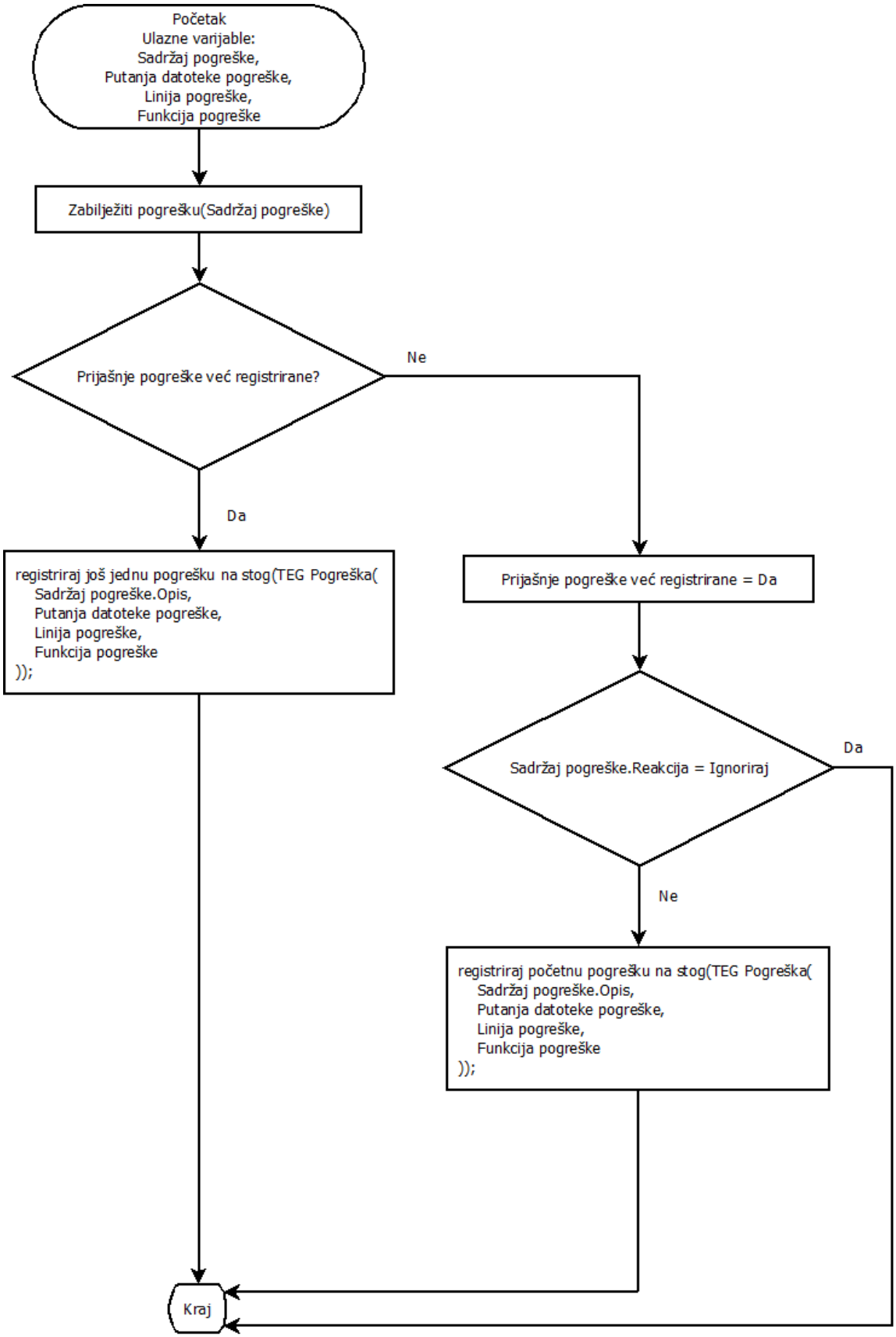
Na ovakav način moguće je umjesto ručnog upisivanja upisati navedena tri makro izraza, a prevoditelj će ih zamijeniti konkretnim vrijednostima tijekom prevođenja. Međutim, može se dalje pojednostaviti pozivanje funkcije registriranja pogreški, tako da se ne mora uopće unositi niti jedan makro izraz, nego samo sadržaj željene pogreške. Ovo se može postići definiranjem novog makro izraza, koji će obaviti upis ovih triju makro izraza, kao što se vidi na isječku koda 3.2. Umjesto izravnog pozivanja funkcije, programer samo treba navesti makro izraz.

```
#define  Podigni(x)  ErrorHandler::PodigniPogresku(x,  __FILE__,  __LINE__,
__FUNC__);
```

Isječak Koda 3.2: *Makro izraz za jednostavnije pozivanje metode za podizanje pogrešaka.*

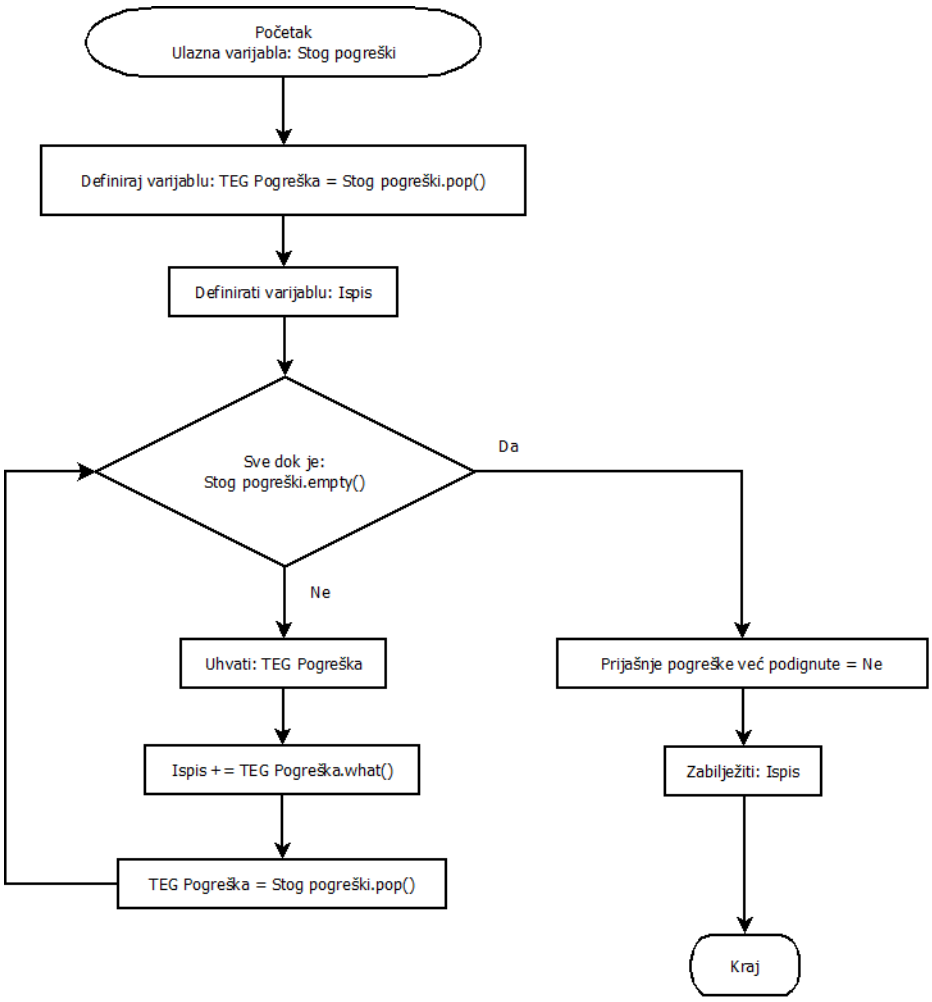


Slika 3.7: *Blok dijagram algoritma konstruktora prilagođene klase za iznimke.*



Slika 3.8: *Blok dijagram algoritma za podizanje vlastito definirane pogreške.*

Osim što može registrirati prilagođene pogreške, rukovatelj pogreškama također mora moći rješavati ovakve pogreške. Za takav slučaj implementirana je još jedna metoda, koja sa stoga čita pogreške, te ih obrađuje i pri tome ispisuje cijeli stog pogrešaka, uključujući njihove opise, izvorne datoteke, linije i funkcije u kojima su se dogodili. Na ovakav način se osigurava jednostavan i informativan mehanizam registriranja pogrešaka, te njihovo rješavanje pozivanjem za to odgovarajućih metoda. Na slici 3.9 dan je blok dijagram koji rukuje pogreškama te ih ispisuje pomoću bilježnika.



Slika 3.9: *Blok dijagram algoritma koji čita pogreške sa stoga i ispisuje njihove detalje pomoću bilježnika.*

3.5 Dizajn i implementacija upravitelja nitima

Upravitelj nitima (engl. *Thread Manager*) je modul TEG okvira, kojemu je zadaća raspodijeliti i ubrzati rad programa, tako da iskoristi prednosti i mogućnosti procesora da izvodi više zadataka istovremeno, tj. da iskoristi mogućnosti paralelizma. Upravitelj nitima paralelizam

ostvaruje korištenjem niti (engl. *threads*), te korištenjem mogućnosti višenitnosti (engl. *multithreading*) procesora. Prednost rada u više niti u odnosu na više procesa je ta što izvođenje programa je fleksibilnije. To podrazumijeva bolju kontrolu nad programom, tj. niti imaju mogućnost pauziranja, sinkronizacija među nitima je lakša, te svu memoriju i resurse međusobno dijele sa drugim nitima. Također uz pomoću niti moguće je izvršavati samo dio programa (npr. funkcije, metode, dijelove koda), dok procesi izvode cijeli program, te se dodatno moraju implementirati mehanizmi za reguliranje toka programa (više o tome će biti objašnjeno u potpoglavlju 3.6). Pri korištenju niti, također je općenito olakšana komunikacija među nitima korištenjem dijeljenih resursa, semafora i muteksa [14]. Upravitelj nitima predstavlja novo, jednostavnije i elegantnije rješenje u odnosu na trenutno rješenje, uz veliki stupanj paralelizacije, kao i za slučaj korištenja višestrukih procesa, jer C++ nema ograničenja što se tiče višenitnog rada za razliku od Pythona (vidi potpglavlje 2.3).

Za implementaciju upravitelja nitima koristi se poseban obrazac za dizajn koda, tzv. skup niti (engl. *thread pool*) kako bi se potpuno iskoristile sve prednosti niti i omogućila se što veća njihova fleksibilnost i iskoristivost. Skup niti je obrazac, koji održava više niti koje čekaju da im se dodjele zadaci i koje mogu izvršavati te zadatke neovisno o drugim nitima, stvarajući konkurentni način izvršavanja [15]. Ovakav pristup korištenja višestrukih niti povećava učinkovitost izvođenja zadataka minimizirajući kašnjenje tijekom izvršavanja. Naime, niti unutar skupa niti su prisutne tijekom cijelog rada glavnog procesa i, umjesto da budu uništene kada završe zadatak, one ostaju u stanju čekanja, sve dok im se ne dodjeli novi zadatak. Na taj način se ne gubi na vremenu višestrukim uništavanjem i stvaranjem niti. Za najveću brzinu izvršavanja zadataka preporuča se stvaranje onoliko niti koliko procesor ima logičkih jezgri. Upravitelj nitima osigurava da se taj broj ne prekorači, jer tada opada brzina izvođenja. Za implementaciju upravitelja nitima, koriste se slijedeće standardne C++ biblioteke:

- *queue*
- *thread*
- *mutex*
- *conditional_variable*
- *functional*
- *vector*

Kako bi mogao upravljati nitima, upravitelj nitima treba sadržavati određenu strukturu podataka za pohranu stvorenih niti. Niti se u memoriju pohranjuju u obliku vektora, odnosno C++ dinamički alociranog polja. Također uz niti, pohranjuju se i zadaci, koji čekaju da se dodijele prvoj slobodnoj niti. Za upravitelj nitima zadaci su funkcije ili procedure, koje se trebaju izvršiti u nitima. Ovo

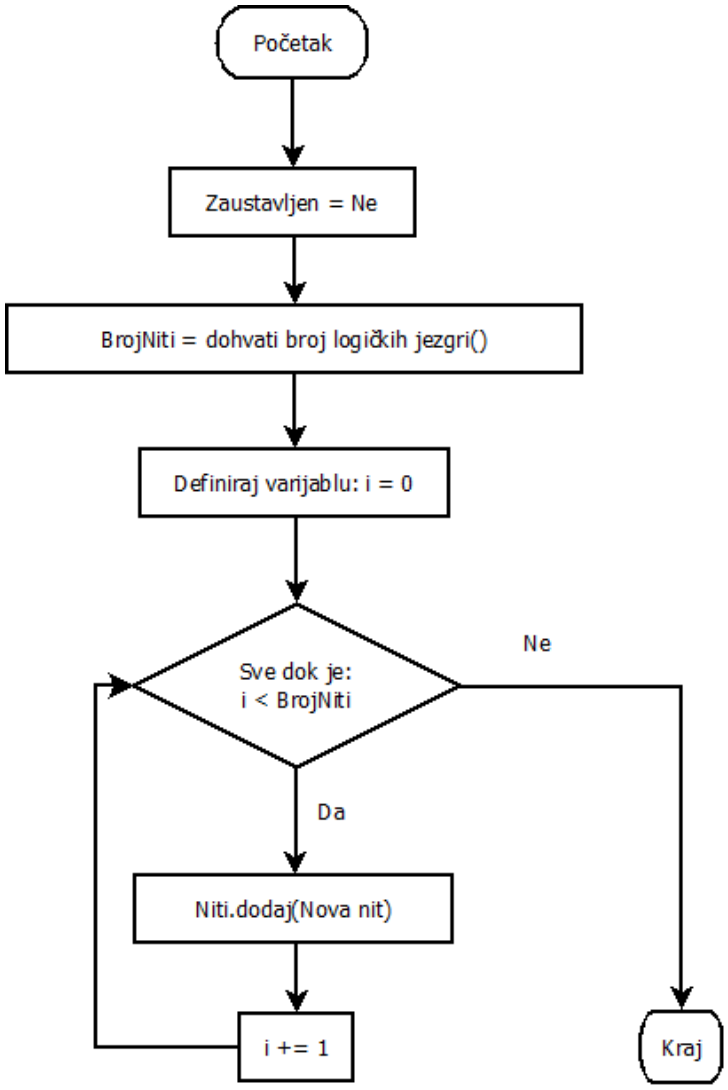
predstavlja problem, jer funkcije i procedure nisu strukture podataka, nego su dijelovi koda i kao takve ne mogu se pohranjivati u memoriju. Kako bi se riješio taj problem koristi se struktura podataka `std::function`, koja je dio *functional* biblioteke. Ova struktura podataka je klasa koja se može „omotati“ oko bilo kojeg elementa u kodu koji se može pozivati (kao što su npr. funkcije, metode, procedure, pokazivači na funkcije, itd.) i na taj način može ih pretvoriti u objekte koji se mogu kopirati [16]. Na ovakav način zadaci koje treba obaviti upravitelj nitima mogu se pohraniti kao funkcijski objekti u memoriju. Zadaci se pohranjuju u strukturu podataka red (engl. *queue*), koji šalje zadatke nitima na način da se prvi pohranjeni zadatak dodjeljuje prvoj slobodnoj niti nakon čega se uklanja iz reda. Tada onaj sljedeći zadatak postaje prvi element u redu kojeg će sljedeća slobodna nit iskoristiti, itd. Slijedi isječak koda 3.3 u kojem su definirane tipovi podataka kao npr. brojači niti, podatak o stanju upravitelja nitima, i navedene strukture podataka koje sadržavaju niti i zadatke.

```
bool Zaustavljen;
unsigned int BrojNiti;
unsigned int BrojZaposlenihNiti;

std::vector<std::thread> Niti;
std::queue<std::function<void(void)>> Zadaci;
```

Isječak Koda 3.3: *Strukture podataka koje koristi upravitelj nitima za pohranu niti i zadataka.*

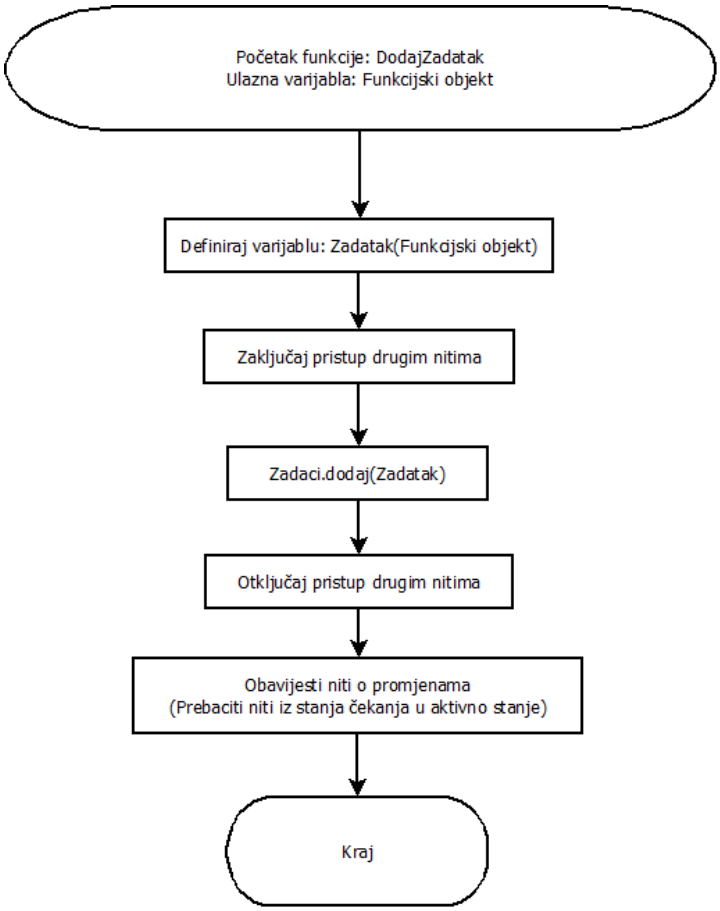
Prije početka rada u više niti, upravitelja nitima je potrebno inicijalizirati. Inicijalizacija niti ostvaruje se očitavanjem broja logičkih jezgri na procesoru (koristeći biblioteku *thread*) i pomoću petlje stvaraju se prazne niti, koje su u stanju čekanja. Broj stvorenih niti odgovara broju logičkih jezgri procesora. Red zadataka se također inicijalizira i u početku je prazan. Na slici 3.10 dan je blokovski prikaz algoritma za stvaranje niti. Važno je napomenuti da za razliku od niti, čiji je broj ograničen hardverskim mogućnostima procesora, najveći mogući broj zadataka nije ograničen. U tom slučaju broj zadataka može vrlo lako nadmašiti broj slobodnih niti. Tada će niti uzeti prvih n zadataka (gdje je n broj niti), dok će ostalih $z - n$ zadataka ostati u redu i čekati prvu nit koja završi sa radom na trenutnom zadatku (gdje je z broj zadataka, $z > n$). Kako niti rade konkurentno, može doći do toga da dvije niti pokušavaju u isto vrijeme dohvatiti isti zadatak iz reda, što bi moglo rezultirati da više niti u isto vrijeme pristupa zadatku iz reda, i tako narušiti rad programa, dovodeći ga u neodređeno stanje. Ipak, ovaj problem moguće je riješiti uporabom muteksa. Muteks osigurava da uvijek točno jedna nit ima pristup redu zadataka. Na slici 3.12 dan je mehanizam

sigurnog pristupa niti redu zadataka u obliku blok dijagrama. Nit pomoću muteksa „zaključa“ dio koda u kojem dohvaća zadatak iz reda. U tome trenutku ako ostale niti pokušavaju pristupiti zadacima moraju čekati da nit koja koristi muteks završi sa uzimanjem zadatka iz reda i „otključa“ kritičan dio koda kojeg mogu druge niti izvršavati, također jedna po jedna. Nadalje, postoji slučaj u kojem je red prazan, a niti pokušavaju dohvatiti zadatak iz reda, što bi rezultiralo pogreškom i nepotrebnim gubljenjem vremena na otklanjanju pogreške ili pak prekid rada programa. U tome slučaju niti koriste uvjetnu varijablu, koja je definirana u biblioteci *conditional_variable*. Pomoću uvjetne varijable stvara se uvjet u kojem niti čekaju u petlji sve dok se ne popuni red sa barem jednim zadatkom. Normalno, uvjetna varijabla šalje nit u stanje čekanja, koja pri tom otključava muteks kako bi druge niti neometano radile i na taj način se izbjegavaju mogući zastoji (engl. d*eadlock*).



Slika 3.10: *Blok dijagram algoritma za stvaranje niti i inicijalizacija reda zadataka.*

Osim što upravlja paralelnim radom niti, upravitelj nitima treba omogućiti dodavanje novih zadataka u red. Na slici 3.11 opisan je mehanizam dodavanja zadataka u red u obliku blok dijagrama. Već je spomenuto da su zadaci funkcije zapakirane u obliku funkcijskog objekta te se kao takve trebaju pohranjivati na stog. Da bi se omogućila jednostavna obrada proizvoljne funkcije i pohrana u red, funkcije se pakiraju bez povratnog tipa, odnosno funkcijski objekt tretira se uvijek kao funkcija tipa `void` koja ne prima argumente. Pohrana funkcijskih objekata kao zadataka u red također je kritična operacija te se također koristi muteks kako bi se zabranilo nitima istovremeno korištenje reda (za čitanje zadataka, koji su prvi redu; istovremeno čitanje i pisanje po redu uzrokuje nepredvidivo ponašanje programa).



Slika 3.11: *Blok dijagram algoritma za dodavanje novog zadatka u red, koristeći funkcijski objekt.*

Iako je moguće pohranjivati zadatke u red, postavlja se pitanje kako poslati proizvoljnu funkciju upravitelju nitima za izvršavanje u paralelu. Na prvi pogled moguće je u funkcijski objekt pakirati samo funkcije tipa `void`, koje ne primaju nikakve argumente, što je vrlo ograničena mogućnost. Međutim, postoji način kako zaobići ovaj problem. U C++ standardu 17 postoje posebne vrste funkcija, imenom lambda izrazi (engl. *lambda expressions*). Lambda izrazi (također

poznati pod nazivom anonimne funkcije) su posebne linijski definirane funkcije, koje omogućavaju jednostavan način definiranja anonimnog funkcijskog objekta unutar linije gdje se planira koristiti [17]. Pomoću lambda izraza može se enkapsulirati proizvoljan kod koji se tada izvršava kao funkcija. Upravo njihove mogućnosti mogu se iskoristiti kako bi se bilo koja funkcija mogla poslati upravitelju nitima. U slučaju da funkcija prima argumente, pomoću lambda izraza mogu se željene varijable proslijediti kao argumenti željenoj funkciji pomoću tzv. inicijalizacijske liste (engl. *capture clause*), te sam lambda izraz poslati upravitelju nitima. Isječak koda 3.4 prikazuje algoritam koji demonstrira dodavanje funkcije sa argumentima u red zadataka.

```
int arg1, arg2, arg3, ...;

DodajZadatak([arg1, arg2, arg3, ...]() {
        MojZadatak(arg1, arg2, arg3, ...);
});
```

Isječak Koda 3.4: *Dodavanje zadatka u red pomoću lambda izraza, koji sadrži funkciju koja prima argumente.*

Na sličan način, pomoću lambda izraza može se dodati funkcija koja ima povratnu vrijednost. Isječak koda 3.5 prikazuje algoritam koji demonstrira dodavanje funkcije sa povratnom vrijednosti u red zadataka.

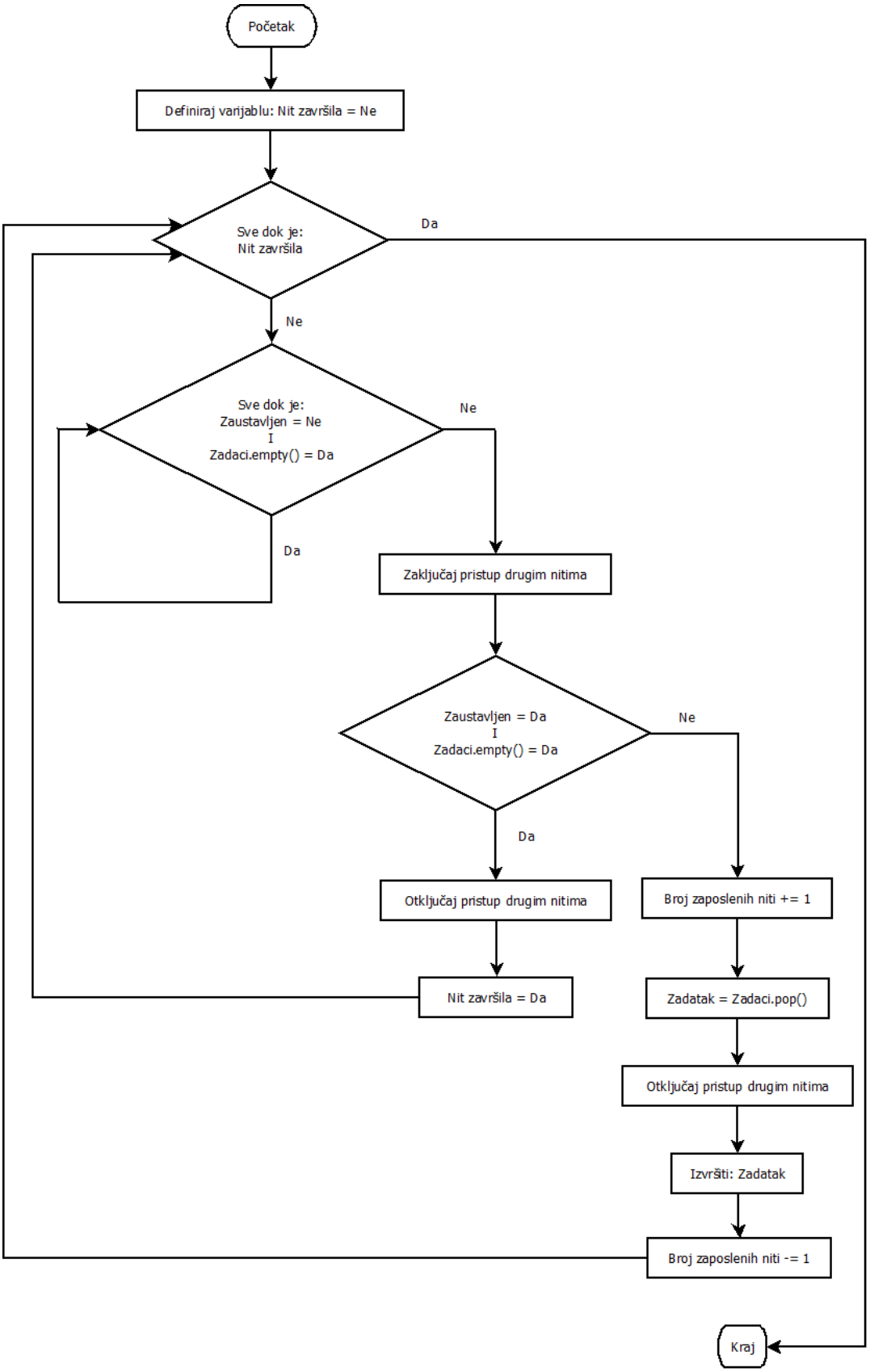
```
int ret;

DodajZadatak([&ret]() {
        ret = MojZadatak();
});
```

Isječak Koda 3.5: *Dodavanje zadatka u red pomoću lambda izraza, koji sadrži funkciju sa povratnom vrijednosti.*

Pomoću lambda izraza može se poslati funkcija koja ima i argumente i povratnu vrijednost, kako je prikazano na isječku koda 3.6.

```
int ret;
string arg1;
float arg2;
...

DodajZadatak([&ret, arg1, arg2, ...]() {
        ret = MojZadatak(arg1, arg2, ...);
});
```

Isječak Koda 3.6: *Dodavanje zadatka u red pomoću lambda izraza, koji sadrži funkciju koja ima argumente i povratnu vrijednost.*



Slika 3.12: *Blok dijagram algoritma rada niti u petlji, unutar koje ili čekaju novi zadatak ili izvršavaju zadatak.*

3.6 Dizajn i implementacija upravitelja procesima

Upravitelj procesima (engl. *Process Manager*) je modul TEG okvira, koji treba ubrzati rad izvođenje programa paralelizacijom. U odnosu na upravitelj nitima, paralelizam se ostvaruje na drugačiji način. Upravitelj procesima omogućuje paralelno izvođenje programa stvaranjem više procesa (engl. *process*), odnosno pokrećući program istodobno više puta za obavljanje više zadatka istovremeno. Iako upravitelj nitima ima prednosti u odnosu upravitelja procesa, ipak upravitelj procesima nadopunjuje neke značajke koje upravitelj nitima ne posjeduje. Glavna prednost pokretanja programa u više procesa od pokretanja jednog programa u više niti je ta što broj procesa ne ovisi samo o broju jezgri jednog procesora. Naime, programi koji rade u više procesa nisu ograničeni na samo jedan procesor na kojemu su pokrenuti, nego mogu nezavisno jedan o drugome raditi preko višeprocesorskih sustava, kao npr. polje servera (engl. *server farm*), te na taj način mogu ostvariti daleko veću procesnu moć u odnosu na jedan višenitni proces. Ipak kako je TEG namijenjen za pokretanje na osobnim računalima, za očekivati je da u praksi višenitni i višeprocesni rad programa ostvaruju približno jednake rezultate pa se naglasak daje na upravitelj nitima. Upravitelj procesima implementiran je da posluži u iznimnim slučajevima, gdje je potrebna velika procesna moć za obradu velikog broja testnih okruženja za višestruke ECU-e istovremeno. Time se ostvaruje „pravi paralelizam“. Programski jezik C++ ne posjeduje standardne biblioteke, koje omogućavaju rad u više procesa, nego je potrebno oslanjati se na biblioteke API-ja (engl. *Application Programming Interface*), koje nudi sam operacijski sustav, ili na biblioteka treće strane. Prilikom implementiranja TEG okvira jedan od zahtjeva je mogućnost pokretanja generatora testnog okruženja na što je moguće više platformi (Microsoft Windows, macOS, GNU/Linux, FreeBSD, OpenBSD, ...). Zbog toga, za implementaciju upravitelja procesima koristi se biblioteka *boost.process*, koja je nezavisna o platformi, a omogućuje rad više procesa [18]. Glavne značajke upravitelja procesima su slijedeće:

- Mogućnost inicijalizacije na proizvoljan broj procesa,
- Nadzor pokrenutih podređenih procesa,
- Mogućnost razlikovanja podređenih procesa pomoću lokalnih identifikacijskih brojeva.

Upravitelj procesima implementiran je koristeći tzv. „gazda – sluga“ (engl. *master - slave*) paradigmu. Ovakva paradigma koristi jedan glavni proces (engl. *master process*), koji preko upravitelja procesa pokreće podređene procese (engl. *slave process*), dodjeljujući im lokalne identifikacijske brojeve i čeka da svi podređeni procesi završe sa radom. Lokalni brojevi služe za lakše identificiranje i podjelu podređenih procesa. Operacijski sustav dodjeljuje identifikacijske brojeve globalno, kako bi mogao razlikovati različite procese. Isto tako funkcionira i upravitelj

procesima, samo što on vidi samo procese koji su pokrenuti preko upravitelja procesima. Za razliku od glavnog procesa, podređeni procesi obavljaju konkretan zadatak, pomoću argumenata koje dobivaju od glavnog procesa pomoću upravitelja procesima. Također kao argument primaju svoj lokalni identifikacijski broj, kako bi pokrenuti proces lako utvrdio da je podređen i izvršavao dio koda namijenjen podređenom procesu. Glavni proces uvijek ima identifikacijski broj 0, a ostali podređeni procesi mogu imati cjelobrojni broj veći od nule. Prema tome, upravitelj procesima sadrži dvije varijable koje sadržavaju informacije o lokalnom PID-u i o broju procesa. U isječku koda 3.7 prikazane su varijable kojima se koristi upravitelj procesima.

```
const unsigned int PIDGazda = 0u;

static unsigned int LokalniPID;
static unsigned int BrojProcesa;
```

Isječak Koda 3.7: *Strukture podataka koje koristi upravitelj procesima za pohranu informacija o vlastitom PID-u i broju procesa, koje može pokretati.*

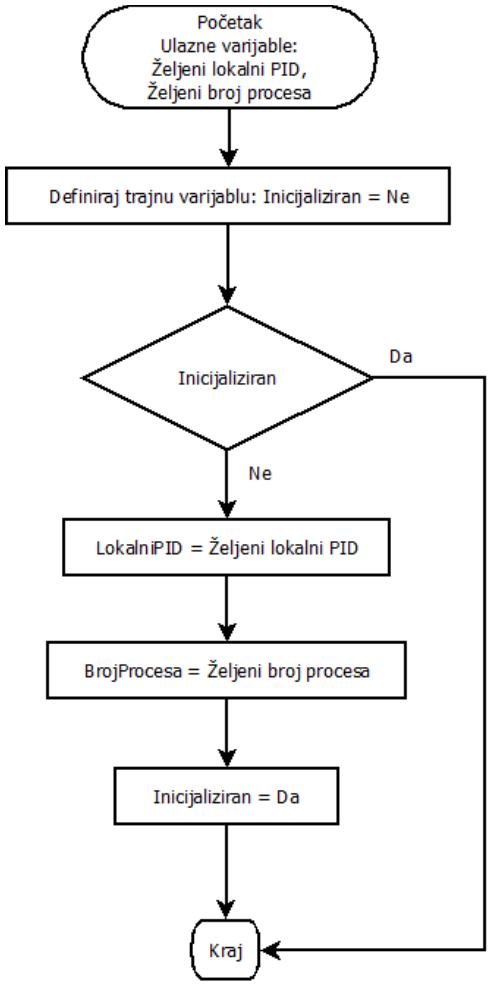
Inicijalizacija upravitelja procesima moguća je samo jednom kako bi se izbjegla višestruka inicijalizacija na glavnom procesu i kao posljedica toga postojanje više glavnih procesa. Višestruka inicijalizacija se izbjegava korištenjem trajne varijable (engl. *static variable*), koja je prisutna na svakoj instanci upravitelja procesima. Na slici 3.13 prikazano je kako je ostvarena sigurna inicijalizacija upravitelja procesima.

Da bi se program pokrenuo u više procesa potrebno je najprije provjeriti je li proces glavni, odnosno ima lokalni PID postavljen na 0, kako je prikazano na slici 3.14. Ako PID nije postavljen na 0, proces je sluga te takav proces ne može stvarati nove procese. Za slučaj glavnog procesa algoritam nastavlja sa stvaranjem podređenih procesa. Unutar petlje stvara se željeni broj procesa, koji se pohranjuju u listu procesa. Tada upravitelj procesima čeka da se izvrši svaki proces unutar liste. Dok se podređeni procesi izvršavaju, upravitelj procesima zaustavlja rad glavnog procesa dok svi podređeni procesi ne završe sa radom. Kada svi podređeni procesi završe sa radom, lista oslobađa zauzeti prostor u memoriji i glavni proces može nastaviti sa svojim radom.

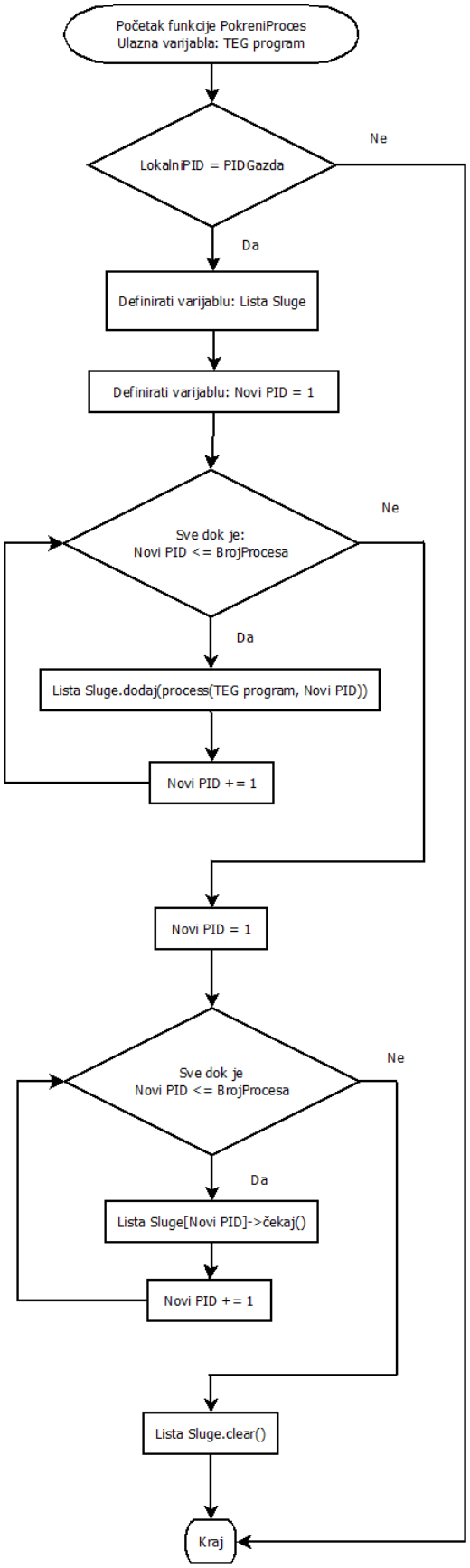
Ipak, kako je upravitelj procesima napravljen koristeći paradigmu „gazda – sluga“, isto tako programer treba imati na umu tu paradigmu prilikom korištenja upravitelja procesima. U isječku koda 3.9 prikazan je primjer pravilnog razdvajanja programa na dio koje obavlja glavni proces i dio kojeg obavljaju podređeni procesi. Razdvajanje programa na dijelove postiže se pomoću naredbe grananja.

```
UpraviteljProcesima Upravitelj(LokalniIDIzArgumenata);

// Gazda
if(Upravitelj.LokalniPID == 0u) {
        Upravitelj.PokreniProces(**argv);
}

// Sluga
else {
        // Mogućnost mapiranja ID-ova za određeni zadatak sluge procesa
        if(Upravitelj.LokalniPID < 100u)
                Zadatak1(arg1, arg2, ...);

        else if(Upravitelj.LokalniPID >= 100u && ID < 1000u)
                Zadatak2(arg3, arg4)

        else
                Res = Zadatak3();
}
```

Isječak Koda 3.8: *Pravilna implementacija višeprocesnog rada.*



Slika 3.13: *Blok dijagram inicijalizacije parametara za pokretanje procesa, kao što je željeni broj podređenih procesa koji rade u paraleli.*



Slika 3.14: *Pokretanje i čekanje podređenih procesa i dodjela identifikacijskih brojeva.*

4. EVALUACIJA PREDLOŽENOG OKVIRA TESTNOG OKRUŽENJA

U ovome poglavlju objašnjen je način evaluacije predloženog TEG okvira. Najprije se ispituje ispravnost rada pojedinog modula TEG okvira. Svaki test sastoji se od implementacije koda za ispitivanje rada, te priloženog rezultata ispitivanja. Nadalje, dobiveni rezultati se komentiraju i pojašnjavaju.

TEG okvir je evaluiran na računalu, sa procesorom *Intel Core i7-7700*, radnog takta frekvencije $3.6GHz$, uz dostupnih $8GB$ radne memorije, te uz tvrdi disk *Western Digital WD10EZEX Blue, Sata 64 MB Cache*, sa prosječnom brzinom čitanja i pisanja $130\ MB/s$ [19].

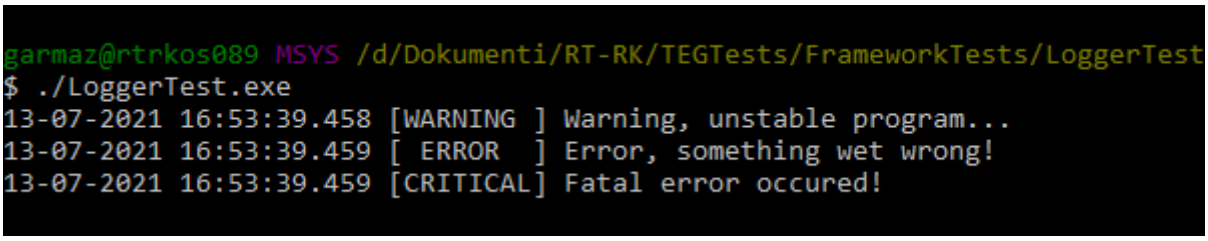
4.1 Ispitivanje rada bilježnika

Ispitivanje ispravnosti rada bilježnika vrši se ispisivanjem poruka koje su namijenjene za različite razine ozbiljnosti. Također, potrebno je navesti datoteku u koju se želi pohraniti bilješke, razina ozbiljnosti za konzolu i razina ozbiljnosti za datoteku. U ovome ispitivanju, za ime datoteke odabran je `TestLog.txt`, bilježenje razina ozbiljnosti za konzolu je postavljeno na `WARNING`, dok je bilježenje razina ozbiljnosti za datoteku postavljeno na `DEBUG`. U programu za ispitivanje priloženi su nizovi znakova, svaki predstavljajući jednu razinu ozbiljnosti, sveukupno njih pet. Potrebno je pomoću bilježnika ispisati ove nizove znakova od najmanje razine ozbiljnosti do najveće. Prema postavkama bilježnika, za njegov ispravan rad očekuje se da se na ekranu ispišu zadnja tri ispitna niza znakova, dok se u datoteku `TestLog.txt` ispišu svi ispitni znakovni nizovi. Osim samih poruka, bilježnik bi također trebao ispisati točno vrijeme (uz milisekunde) i razinu ozbiljnosti uz koju je poruka zabilježena. Isječak koda 4.1 prikazuje implementaciju algoritma za ispitivanje ispravnosti rada bilježnika.

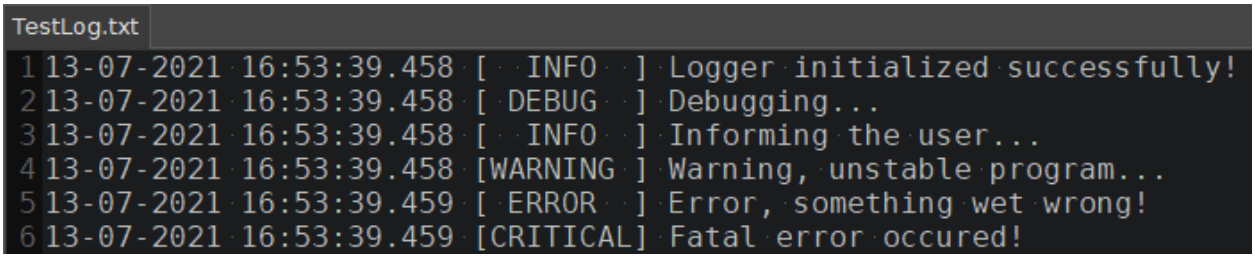
```
Bilježnik::Inicijaliziraj("TestLog.txt", Debug, Warning);

string TestStr1("Debugging...");
string TestStr2("Informing the user...");
string TestStr3("Warning, unstable program...");
string TestStr4("Error, something wet wrong!");
string TestStr5("Fatal error occured!");

Bilježnik::Zabilježi(TestStr1, Debug);
Bilježnik::Zabilježi(TestStr2, Info);
Bilježnik::Zabilježi(TestStr3, Warning);
Bilježnik::Zabilježi(TestStr4, Error);
Bilježnik::Zabilježi(TestStr5, Critical);
```

Isječak Koda 4.1: *Programski kod za ispitivanje rada bilježnika.*

Rezultati zasebnog rada bilježnika su zadovoljili očekivanja. Na slikama 4.1 i 4.2 vrijeme je točno očitano i u pravilnom formatu je ispisano na ekran i u datoteku zajedno s ozbiljnosti i porukom. Prema tome bilježnik radi ispravno.



Slika 4.1: *Rezultati ispitivanja rada bilježnika – ispis tri znakovnih nizova manjih ozbiljnosti na ekran.*



Slika 4.2: *Rezultati ispitivanja rada bilježnika – ispis svih pet znakovnih nizova u datoteku.*

4.2 Ispitivanje rada rukovatelja datotekama

Ispitivanje rada rukovatelja datotekama provodi se pomoću dva programa. Prvi program ispituje ispravnost parsiranja različitih nizova znakova. Drugi program ispituje ispravnost pretrage stvarnih datoteka na datotečnom sustavu, korištenjem djelomičnih naziva.

Da bi se ispitala ispravnost parsiranja, u prvome programu koriste se izrazi koji predstavljaju djelomična imena datoteka. Za djelomična imena odabrani su primjeri izraza iz tablice 3.1 (potpoglavlje 3.3). Za svaki izraz definiraju se nizovi znakova koji oponašaju putanje i imena datoteka. Svaki izraz ima tri nizova znakova koji zadovoljavaju taj izraz, te tri nizova znakova koji ne zadovoljavaju izraz. Na izlazu iz prvog programa očekuje se za svaki izraz tri istinitih vrijednosti, a nakon toga tri neistinitih vrijednosti ispisanih na ekran. U isječku koda 4.2 dan je algoritam za ispitivanje provjere valjanosti imena. Može se primijetiti da konkretni nazivi nisu dani nego je samo dan algoritam, budući da ima velik broj primjera na kojima se ispitivao rad validacije imena prema djelomičnom nazivu. Na slici 4.3 prikazani su konkretni testni primjeri djelomičnih putanji i putanja do datoteka, te njihovi rezultati. Sa slike 4.3 također se vidi da za svaki test prva tri primjera su validna imena (rezultat je 1), a ostala tri primjera nisu (rezultat je 0).

```
int main(int argc, char **argv) {
        RukovateljDatotekama Rukovatelj;
        string Izraz = // ...

        string DobarNaziv1 = // ...
        string DobarNaziv2 = // ...
        string DobarNaziv3 = // ...
        string LosNaziv1 = // ...
        string LosNaziv2 = // ...
        string LosNaziv3 = // ...

        bool Rez1 = Rukovatelj.ProvjeriIme(DobarNaziv1, Izraz, false);
        bool Rez2 = Rukovatelj.ProvjeriIme(DobarNaziv2, Izraz, false);
        bool Rez3 = Rukovatelj.ProvjeriIme(DobarNaziv3, Izraz, false);
        bool Rez4 = Rukovatelj.ProvjeriIme(LosNaziv1, Izraz, false);
        bool Rez5 = Rukovatelj.ProvjeriIme(LosNaziv2, Izraz, false);
        bool Rez6 = Rukovatelj.ProvjeriIme(LosNaziv3, Izraz, false);

        cout << Izraz << "\t" << DobarNaziv1 << "\t" << Rez1 << endl;
        cout << Izraz << "\t" << DobarNaziv2 << "\t" << Rez2 << endl;
        cout << Izraz << "\t" << DobarNaziv3 << "\t" << Rez3 << endl;
        cout << Izraz << "\t" << LosNaziv1 << "\t" << Rez4 << endl;
        cout << Izraz << "\t" << LosNaziv2 << "\t" << Rez5 << endl;
        cout << Izraz << "\t" << LosNaziv3 << "\t" << Rez6 << endl;

        return 0;}
```

Isječak Koda 4.2: *Algoritam za ispitivanje rada validacije imena za dano djelomično ime, odnosno izraz.*

```
TEST 1
*.xml file1.xml       1
*.xml dir3/filexml.xml        1
*.xml xml.xml.xml.xml 1
*.xml file1.txt       0
*.xml dir3/fakefilexml.xml/dir4/invalid.txt   0
*.xml xml.xml.xml.ini 0

TEST 2
file*.txt     file1.txt       1
file*.txt     filetxt.txt     1
file*.txt     file.txt.file.txt       1
file*.txt     file1.exe       0
file*.txt     txt/file/txt.file       0
file*.txt     file.txt.file   0

TEST 3
file* file.txt        1
file* file123.ini     1
file* file    1
file* fil3.exe        0
file* txt/file/txt.file       0
file* dir/file.txt    0

TEST 4
dir*subdir*file*.ini  dir/subdir/file.ini     1
dir*subdir*file*.ini  dirx/x/subdiry/y/filez.ini      1
dir*subdir*file*.ini  dirfakesubdirfakefile123.ini    1
dir*subdir*file*.ini  dir/subdir/file.exe     0
dir*subdir*file*.ini  dir123/subdir456/file.ini.file  0
dir*subdir*file*.ini  /dir/subdir/file.ini    0
```

Slika 4.3: *Rezultati validacije imena rukovatelja datotekama. U prvome stupcu je željeno djelomično ime. U drugome stupcu je primjer putanje do neke datoteke. U trećem stupcu je ispis rezultata.*

Ispitivanje drugog programa provodi se nad mapom u kojoj se nalazi kompletno postojeće rješenje TEG-a, uz sve potrebne datoteke za generiranje konačnog izlaza. Ispituje se rad pretrage bez korištenja djelomičnih naziva, korištenje djelomičnih naziva sa jednim specijalnim znakom, te korištenje djelomičnih naziva uz korištenje dva specijalna znaka. Očekivani rezultat je da rukovatelj datotekama pronađe sve datoteke, definirane u programu. Kao potvrda da su sve željene datoteke pronađene, rezultati rukovatelja datotekama uspoređuju se sa rezultatima dobro poznate i provjerene Linux naredbe *find*. Isječak koda 4.3 prikazuje pretragu pomoću funkcije rukovatelja datotekama `Trazi`, koji najprije pretražuje datoteke sa punim nazivom, nakon toga pretražuje datoteke korištenjem djelomičnog naziva sa jednim specijalnim znakom i na kraju sa dva specijalna znaka.

```
#define TEG_DIR "D:\\Dokumenti\\RT-RK\\iecu_spielwiese"

RukovateljDatotekama Rukovatelj(TEG_DIR);

// Pronađi __init__.py files
Rukovatelj.Trazi("__init__.py", true);

// Pronađi sve .c datoteke
Rukovatelj.Trazi("*.c", false);

// Pronađi datoteke koje u svome imenu imaju niz znakova "test"
Rukovatelj.Trazi("*test*", false);
```

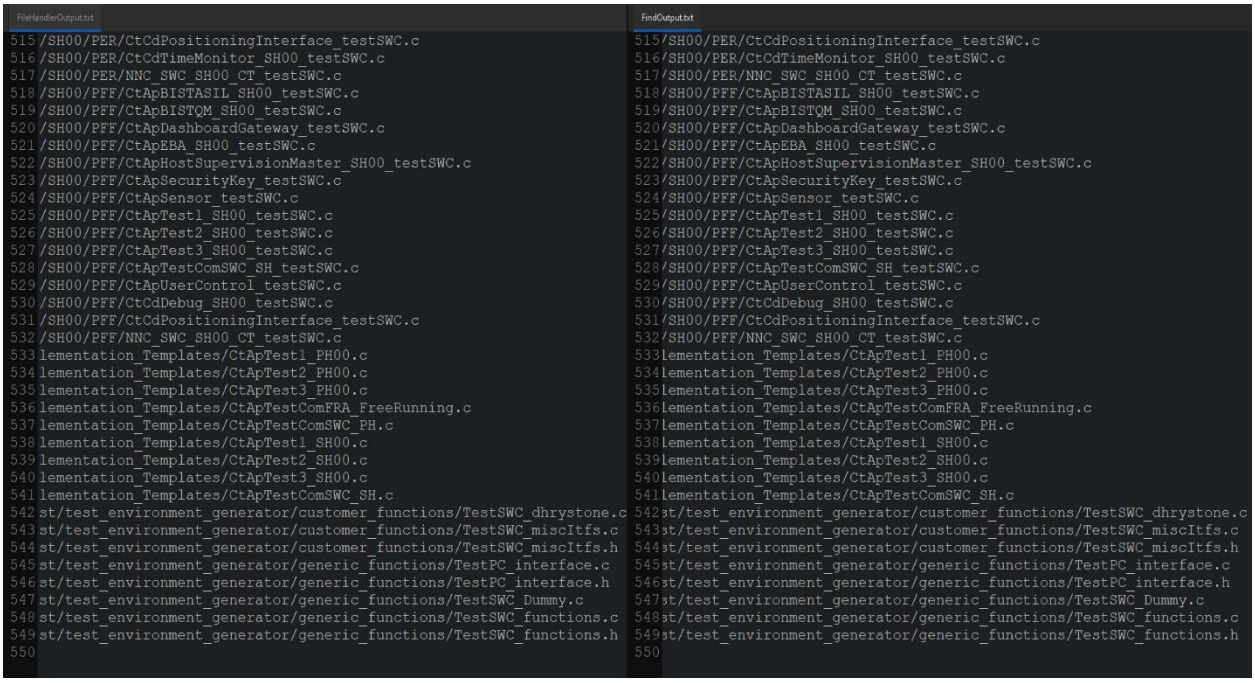
Isječak Koda 4.3: *Ispitivanje pretrage datoteka pomoću različitih kriterija.*

Nakon pretrage datoteka slijedećim naredbama u *Windows PowerShell-u* ili u Linux terminalu ostvaruju se rezultati kao i pomoću rukovatelja datotekama. U isječku koda 4.4 dane su tri naredbe *find* koje su ekvivalentne funkciji `Trazi` od rukovatelja datotekama.

```
find . -type f -name __init__.py
find . -type f -iname *.c
find . -type f -iname *test*
```

Isječak Koda 4.4: *Zadavanje naredbe find za pretragu datoteka u terminalu.*

Na slici 4.4 može se primijetiti da su rezultati, koje je rukovatelj datotekama ispisao, identični onima koje je ispisala naredba *find*, kao i broj pronađenih stavki od ukupno 549 datoteka. Ovim rezultatom pokazana je ispravnost rada rukovatelja datotekama.



Slika 4.4: *Usporedba datoteka, koje je pronašao rukovatelj datotekama nasuprot naredbi find.*

Osim ispravnosti rada rukovatelja datotekama, također se provodi ispitivanje performansi rada u odnosu na naredbu *find*. Za mjerenje brzine pronalaska datoteka, koristi se C++ biblioteka *chrono*. Za mjerenje vremena pretrage rukovatelja datotekama, dodavaju se vremenski brojači biblioteke *chrono* između linija za pretragu datoteka (`high_resolution_clock::now()`). Međutim, naredbu *find* nije moguće izravno mjeriti, jer nije dio programskog jezika C++, već zaseban program. Ipak, moguće je izraditi C++ program, koji poziva naredbu *find* i čeka da ta naredba završi sa radom, mjereći pri tome vrijeme pomoću biblioteke *chrono*. Naime, C++ posjeduje funkciju `system()`, koja pokreće programe u zasebnom procesu. Funkcija `system()` je po svome načinu rada bliska upravitelju procesima, međutim ne podržava pokretanje programa u više od jednog procesa istovremeno, što nije paralelno višeprocesno upravljanje. Upravo zbog nedostatka paralelnog pokretanja procesa omogućava da se pokrene jedna naredba i svakom pozivu naredbi *find* može se zasebno izmjeriti vrijeme, jer se naredbe izvršavaju sekvencijalno. U isječku koda 4.5 nadograđen je algoritam isječka koda 4.3 sa vremenskim brojačima. Isječak koda 4.6 demonstrira korištenje funkcije `system()` kako bi bilo moguće izmjeriti trajanje naredbe *find*. Ispitivanje se ponavlja 10 puta kako bi se moglo precizno odrediti prosječno vrijeme pretrage za oba programa.

```
#define TEG_DIR "D:\\Dokumenti\\RT-RK\\iecu_spielwiese"

RukovateljDatotekama Rukovatelj(TEG_DIR);

// Pronađi __init__.py files
auto Start = high_resolution_clock::now();
Rukovatelj.Trazi("__init__.py", true);
auto Stop = high_resolution_clock::now();
auto Vrijeme = duration<float, seconds::period>( Stop - Start);

// Pronađi sve .c datoteke
Start = high_resolution_clock::now();
Rukovatelj.Trazi("*.c", false);
Stop = high_resolution_clock::now();
Vrijeme = duration<float, seconds::period>(Stop - Start);

// Pronađi datoteke koje u svome imenu imaju niz znakova "test"
Start = high_resolution_clock::now();
Rukovatelj.Trazi("*test*", false);
Stop = high_resolution_clock::now();
Vrijeme = duration<float, seconds::period>(Stop - Start);
```

Isječak Koda 4.5: *Algoritam pretrage datoteka korištenjem rukovatelja datotekama, nadograđen vremenskim brojačima.*

```
auto Start = high_resolution_clock::now();
cout << system("find . -type f -name __init__.py") << endl;
auto Stop = high_resolution_clock::now();
auto Vrijeme = duration<float, seconds::period>(Stop - Start);

Start = high_resolution_clock::now();
cout << system("find . -type f -name *.c") << endl;
Stop = high_resolution_clock::now();
Vrijeme = duration<float, seconds::period>(Stop - Start);

Start = high_resolution_clock::now();
cout << system("find . -type f -name *test*") << endl;
Stop = high_resolution_clock::now();
Vrijeme = duration<float, seconds::period>(Stop - Start);
```

Isječak Koda 4.6: *Algoritam pretrage naredbom find, nadograđen vremenskim brojačima.*

Nakon provođenja ispitivanja, dobiveni su rezultati prikazani u tablici 4.1.

Prosječno vrijeme pretrage [s]	__init__.py	*.c	*test*
Rukovatelj datotekama	0.0806145	0.154722	0.155973
find	0.127611	0.124288	0.156078

Tablica 4.1: *Rezultati prosječnih brzina rukovatelja datotekama u odnosu na naredbu find.*

Što se tiče performansi pronalaska datoteka, brzine pretrage ovise o brzinama jedinica za pohranu (*Western Digital WD10EZEX Blue*). Također, zbog činjenice da datotečnim sustavom također upravlja sam operacijski sustav, to ima utjecaj i na brzinu pretrage. U nekim slučajevima je rukovatelj datotekama je bio brži od naredbe *find*, a nerijetko je i naredba *find* nadmašila brzine rukovatelja datotekama. Ono što se može reći pomoću podataka iz tablice 4.1 jest da je u prosjeku rukovatelj datotekama brži za manji broj datoteka (npr. za datoteke __init__.py, ima ih 11). Oba programa postižu približno iste brzine za veći broj datoteka (npr. za .c datoteke, kojih je 276 i datoteke koje imaju dio imena „test“, a ima ih 262).

4.3 Ispitivanje rada rukovatelja pogreškama

Kako bi se na pravilan način ispitao rad rukovatelja pogreškama, implementiran je program koji koristi rekurzivnu funkciju. Rekurzivna funkcija poziva samu sebe do nekog određenog broja N, tvoreći tako stog pozivanja na procesoru. Zadnja pozvana funkcija na procesorskom stogu podiže pogrešku pomoću rukovatelja pogreški, koju rukovatelj pogrešaka treba propagirati sve do glavnog djela koda (funkcija `main()`). Nakon toga, potrebno je pomoću bilježnika ispisati stog svih pogrešaka (engl. *stack trace*). Očekivani ispis je stog svih pogrešaka, kojih ima N. Prilikom ispisa pogreške očekuje se ispis opisa, izvorne datoteke u kojima su nastale, kao i linije unutar izvornog koda i naziv funkcije. Ispisati bi se trebali pomoću bilježnika u predviđenom formatu (sa vremenom ispisa i ozbiljnosti poruke). Isječak koda 4.4 ispituje ispravnost rukovatelja pogreškama.

```
const unsigned int MaxPoziva = // N...
unsigned int BrPoziva = 0u;

void Rekurzija() {
        if(BrPoziva == MaxPoziva)
                RukovateljPogreskama::Podigni("Error, max call stack re-
ached...");

        BrPoziva += 1u;
```

```
        unsigned int Dubina = BrPoziva;

        try {
                Recursion();
        }
        catch (TEGPogreska &e) {
                string ErrorStr = "Error at depth: ";
                ErrorStr += to_string(Dubina);
                PukovateljPogreskama::Propagiraj(ErrorStr);
        }
}

int main(int argc, char **argv) {
        ErrorHandler::Rukuj(Rekurzija);

        return 0;
}
```

Isječak Koda 4.7: *Ispitivanje ispravnosti rada rukovatelja pogreškama, korištenjem četiri ugniježđene funkcije.*

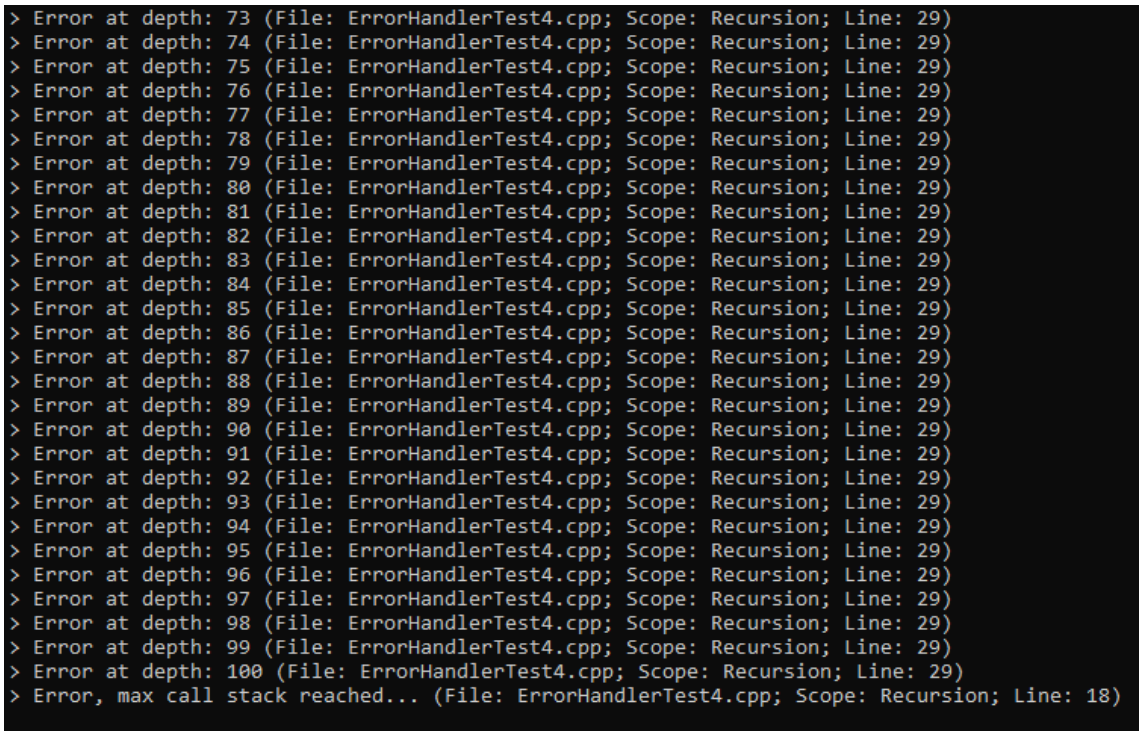
Na slikama 4.5, 4.6 i 4.7 mogu se vidjeti sve ispisane pogreške te ispis njihovog stoga u formatu bilježnika. Prema tome, ovim testom zadovoljena je funkcionalnost rukovatelja pogreškama.

```
2021-09-02 01:34:00.406 [ ERROR ] Following errors were handled successfully:
                                  > Error at depth: 1 (File: ErrorHandlerTest4.cpp; Scope: Recursion; Line: 29)
                                  > Error, max call stack reached... (File: ErrorHandlerTest4.cpp; Scope: Recursion; Line: 18)
```

Slika 4.5: *Ispis stoga pogrešaka na ekran za N = 1.*

```
2021-09-02 01:35:06.457 [ ERROR ] Following errors were handled successfully:
                                  > Error at depth: 1 (File: ErrorHandlerTest4.cpp; Scope: Recursion; Line: 29)
                                  > Error at depth: 2 (File: ErrorHandlerTest4.cpp; Scope: Recursion; Line: 29)
                                  > Error at depth: 3 (File: ErrorHandlerTest4.cpp; Scope: Recursion; Line: 29)
                                  > Error at depth: 4 (File: ErrorHandlerTest4.cpp; Scope: Recursion; Line: 29)
                                  > Error at depth: 5 (File: ErrorHandlerTest4.cpp; Scope: Recursion; Line: 29)
                                  > Error at depth: 6 (File: ErrorHandlerTest4.cpp; Scope: Recursion; Line: 29)
                                  > Error at depth: 7 (File: ErrorHandlerTest4.cpp; Scope: Recursion; Line: 29)
                                  > Error at depth: 8 (File: ErrorHandlerTest4.cpp; Scope: Recursion; Line: 29)
                                  > Error at depth: 9 (File: ErrorHandlerTest4.cpp; Scope: Recursion; Line: 29)
                                  > Error at depth: 10 (File: ErrorHandlerTest4.cpp; Scope: Recursion; Line: 29)
                                  > Error, max call stack reached... (File: ErrorHandlerTest4.cpp; Scope: Recursion; Line: 18)
```

Slika 4.6: *Ispis stoga pogrešaka na ekran za N = 10.*



Slika 4.7: *Ispis stoga pogrešaka na ekran za N = 100.*

4.4 Ispitivanje paralelnog rada programa

Za ispitivanje paralelnog rada uspoređivana su vremena sekvencijalnog izvođenja sa paralelnim izvođenjem pomoću više niti, te pomoću više procesa. Za ispitivanje rada u više niti ili više procesa, potreban je vremenski zahtijevan program koji se može izvršavati paralelno, tj. imati mogućnost istovremenog obavljanja zadaća. Kako bi se to postiglo, odabrano je onoliko zadataka koliko ima logičkih jezgara na procesoru, za potpunu iskoristivost procesora. Procesor, koji se koristi prilikom testiranja (Intel i7) posjeduje osam logičkih jezgri, što znači da bi se paralelizam najbolje mogao ostvariti izradom osam zadataka. Stoga, kao primjer odabran je zadatak u kojem je potrebno sortirati osam polja od *N* elemenata od najmanjeg prema najvećem. Ovaj zadatak izvodi se na tri načina: na sekvencijalni način, gdje se obavlja jedan po jedan zadatak, način u višestrukim nitima i način u višestrukim procesima, gdje se pomoću upravitelja nitima, odnosno procesima svih osam zadataka obavlja istovremeno. Za pravilan rad u više niti ili procesa očekuje se oko 8 puta brže izvođenje programa, te približno 100% iskoristivosti procesora. Za mjerenje vremena izvođenja programa u sva tri načina, koristi se standardna biblioteka *chrono*. U isječku koda 4.8 nalazi se algoritam koji radi sortiranje jednodimenzionalnog polja. Algoritam radi u dvije petlje i na taj način prolazi kroz polje i uspoređuje svaki element iz prve petlje sa svakim elementom iz druge. Ako je element iz druge petlje manji od elementa iz prve petlje, algoritam im

zamjeni mjesta.

```
void Zamijeni(int *Num1, int *Num2) {
	int Temp = *Num1;
	*Num1 = *Num2;
	*Num2 = Temp;
}

void Sortiraj(vector<int> &Arr) {
	unsigned int ArrSize = (unsigned int)Arr.size();

	unsigned i, j;
	for(i = 0u; i < ArrSize - 1u; i++)
		for(j = i + 1u; j < ArrSize; j++)
			if(j < i)
				Zamijeni(&Arr[i], &Arr[j]);

}
```

Isječak Koda 4.8: *Funkcija koja obavlja zadatak sortiranja.*

Osim samog algoritma potrebno izraditi spomenutih 8 polja na kojima bi algoritam mogao izvršavati sortiranje. U isječku koda 4.9 dan je algoritam u kojem su definirane veličine polja i sama polja.

```
#define N // Proizvoljan broj elemenata u polju, koji se odabire tijekom tesrtiranja

const unsigned int ArrsCount = 8u;
vector<unsigned int> ArrSizes{N, N, N, N, N, N, N, N};

vector<vector<int>> Arrs;

unsigned int i;
for(i = 0u; i < ArrsCount; i++)
	Arrs.push_back(InitArr(ArrSizes[i]));
```

Isječak Koda 4.9: *Inicijalizacija 8 polja nasumičnim brojevima koje je potrebno sortirati.*

Osim same deklaracije polja, također ih je potrebno popuniti neodređenim brojevima. U isječku koda 4.10 dan je algoritam popunjavanja polja sa neodređenim cijelim brojevima u rasponu od $-\frac{N}{2}$ do $\frac{N}{2}$.

```
vector<int> InitArr(const unsigned int ArrSize) {
        vector<int> NewArr(ArrSize, 0);
        unsigned int i;
        for(i = 0u; i < ArrSize; i++) {
                int RandNum = rand() % ArrSize;
                NewArr[i] = RandNum / 2 - (int)i;
        }
        return NewArr;
}
```

Isječak Koda 4.10: *Funkcija za inicijalizaciju polja nasumičnim brojevima.*

Nakon inicijalizacije provodi se sortiranje i pri tome mjeri se vrijeme potrebno da se izvrši algoritam sortiranja te ukupno vrijeme do završetka programa. Mjerenje vremena izvršavanja pojedinog načina rada mjeri se osam puta, na različitim veličinama polja. Za veličine polja N odabrane su slijedeće vrijednosti: $N = 10, 50, 100, 500, 1\,000, 5\,000, 10\,000, 50\,000$.

4.4.1 Rezultati algoritma sekvencijalnog sortiranja

Isječak koda 4.11 prikazuje klasičan sekvencijalni način izvođenja sortiranja. Nakon izvršavanja sortiranja računa se vrijeme proteklo od početka sortiranja.

```
auto Start = high_resolution_clock::now();

for(i = 0u; i < ArrsCount; i++)
        Sortiraj(Arrs[i]);

auto Stop = high_resolution_clock::now();
auto Vrijeme = duration<float, seconds::period>(Stop - Start);

cout << "All tasks done! Total time elapsed: " << Vrijeme.count() << "s" << endl;
```

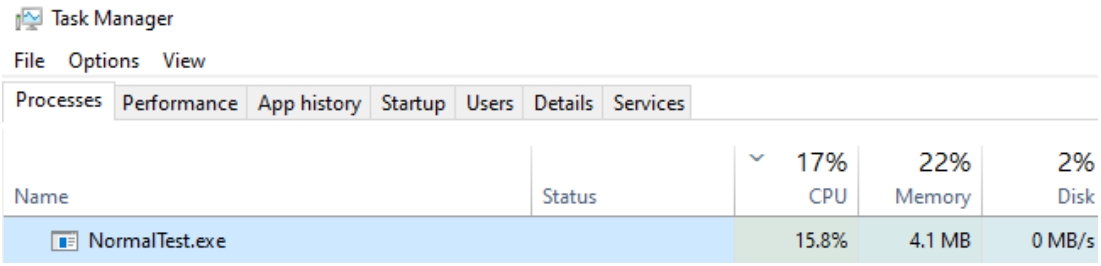
Isječak Koda 4.11: *Sekvencijalni način izvođenja sortiranja uz mjerenje vremena izvršavanja pojedinog zadatka, kao i cijelog programa.*

U tablici 4.2 dobiveni su vremenski rezultati izvođenja algoritma sortiranja za sve odabrane vrijednosti N.

N	*Vrijeme trajanja programa [s]*
10	0.000001
50	0.000023
100	0.000089
500	0.002394
1000	0.009431
5 000	0.236643
10 000	0.958583
50 000	24.442707

Tablica 4.2: *Rezultati dobiveni ispitivanjem sekvencijalnog izvršavanja algoritma sortiranja.*

Prema podacima iz tablice 4.2 i na slici 4.8 može se primijetiti da je sekvencijalno izvođenje zadataka vrlo sporo. Također u tom slučaju iskoristivost procesora je manja od 20%. Zadacima treba sveukupno oko 24 sekunde da se izvrše, za veličinu polja od samo 50 000 elemenata.



Slika 4.8: *Popis procesa i iskoristivost procesora tijekom sekvencijalnog izvođenja programa.*

4.4.2 Rezultati algoritma višenitnog sortiranja

Nakon sekvencijalnog izvođenja programa, u algoritam se uključuje upravitelj nitima. Nakon inicijalizacije upravitelja nitima, zadaci se pokreću paralelno na način kako je objašnjeno u potpoglavlju 3.5. i kako je prikazano u isječku koda 4.12.

```
auto Start = high_resolution_clock::now();

UpraviteljNitima Upravitelj;

for(i = 0u; i < Upravitelj.BrojNiti; i++) {
        vector<int> &ArrRef = Arrs[i];

        unsigned int NitIndx = i + 1u;
```

```
        Upravitelj.DodajZadatak([&ArrRef, NitIndx]() {
                auto Start2 = high_resolution_clock::now();
                Sortiraj(ArrRef);
                auto Stop2 = high_resolution_clock::now();
                auto Vrijeme2 = duration<float, seconds::period>(Stop2 -
Start2);
                cout << "Thread " << NitIndx << " finished sorting. Elapsed
time is: " << Vrijeme2.count() << "s" << endl;
        });
}

cout << "Threads have started!" << endl;
Upravitelj.Cekaj();

auto Stop = high_resolution_clock::now();
auto Vrijeme = duration<float, seconds::period>(Stop - Start);

cout << "All tasks done! Elapsed time is: " << Vrijeme.count() << "s" <<
endl;
```

Isječak Koda 4.12: *Realizacija višenitnog obavljanja zadataka, korištenjem upravitelja nitima.*

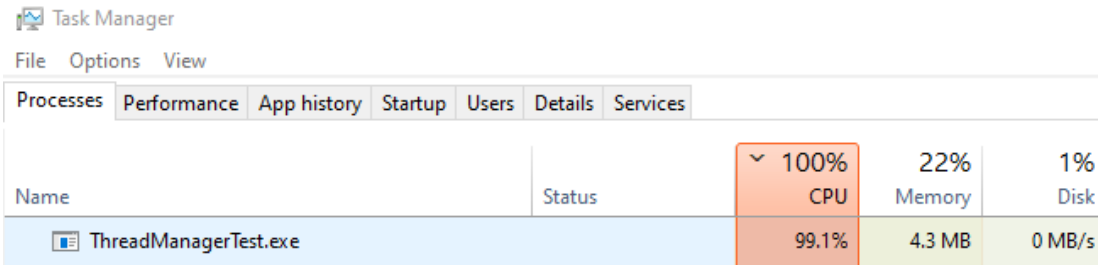
U tablici 4.3 dobiveni su vremenski rezultati izvođenja algoritma sortiranja u više niti za sve odabrane vrijednosti N.

N	*Prosječno trajanje niti [s]*	*Ukupno trajanje programa [s]*
10	0	0.000103
50	0.0000035	0.000096
100	0.0000203	0.102504
500	0.0003157	0.113711
1 000	0.0012597	0.100799
5 000	0.0310077	0.109183
10 000	0.1242035	0.230849
50 000	3.1021018	3.175554

Tablica 4.3: *Rezultati dobiveni ispitivanjem višenitnog izvršavanja algoritma sortiranja.*

U tablici 4.3 i na slici 4.9 vidi se da je se niti izvršavaju puno brže od sekvencijalnog izvođenja, što je i očekivano na poljima sa većim brojem elemenata. Izvršavanjem zadataka pomoću 8 niti vrijeme izvršavanja smanjeno je sa 24 s na prosječno vrijeme od 3.17 s za veličinu polja od 50 000. Time program postiže prosječno do 7.7 puta brže izvođenje. Iskorištenost

procesora je 100%, a od toga sam program koristi 99.1% procesora što je dokaz da procesor obrađuje zadatke na svih 8 jezgri, pomoću niti. Također valja napomenuti da to vrijedi za veličine polja koje su veće od 1 000 elemenata. Budući da vrijeme uloženo u stvaranje niti doprinosi sporijem izvođenju programa za veličine polja manje od 1 000, često se za takve jednostavnije zadatke ne isplati koristiti višenitnost.



Slika 4.9: *Popis procesa i iskoristivost procesora tijekom višenitnog izvođenja programa.*

4.4.3 Rezultati algoritma višeprocesnog sortiranja

Prilikom testiranja upravitelja procesima, koristio se isti zadatak, kao prilikom testiranja upravitelja nitima. Međutim, iako upravitelj procesima postiže paralelizam, kao i upravitelj nitima, ipak implementacija upravitelja procesima je u potpunosti različita u odnosu na upravitelj nitima. Program koji ispituje rad upravitelja procesima implementira se na način koji je objašnjen na kraju potpoglavlja 3.6. Konkretna implementacija sortiranja u više procesa prikazana je u isječku koda 4.13.

```
#define BR_PROCESA 8u

PID = UpraviteljProcesima::ProcitajPID(argc, argv);
UpraviteljProcesima Upravitelj(PID, BR_PROCESA);

// Gazda
if(Upravitelj.LokalniPID == 0u) {
        auto Start = high_resolution_clock::now();
        Upravitelj.PokreniProces(argv, argc);

        auto Stop = high_resolution_clock::now();
        auto Vrijeme = duration<float, seconds::period>(Stop - Start);

        cout << "All tasks done! Total time elapsed: " << Vrijeme << "s" <<
endl;
}
```

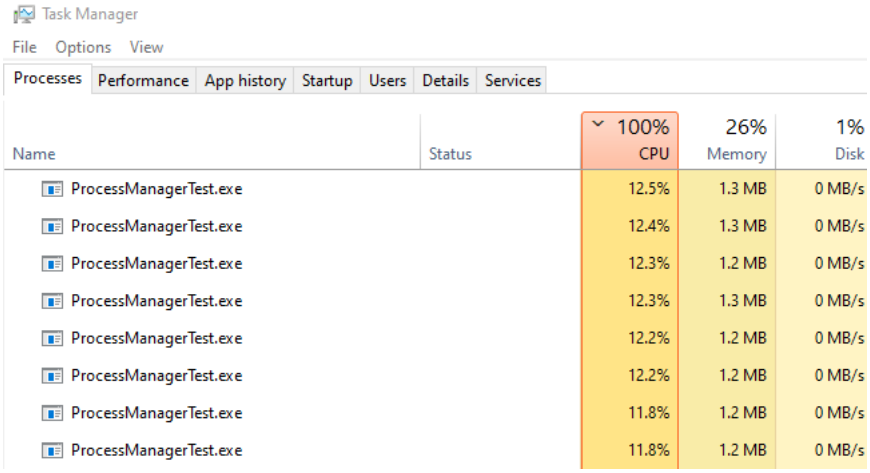
```
// Sluge
else {
        unsigned int ArrSizeIndex = Upravitelj.LokalniPID - 1u;
        unsigned int ArrSize = TaskSizes[ArrSizeIndex];

        vector<int> Arr = InitArr(ArrSize);

        cout << "Slave process initialized, with PID: " << Upravitelj.Lo-
kalniPID << endl;

        auto StartSluga = high_resolution_clock::now();
        Sortiraj(Arr);

        auto StopSluga = high_resolution_clock::now();
        auto VrijemeSluga = duration<float, seconds::period>(StopSluga -
StartSluga);

        cout << "Slave process with PID  " << Upravitelj.LokalniPID << "
finished sorting. Elapsed time is: " << VrijemeSluga.count() << "s" << endl;
}
```

Isječak Koda 4.13: *Realizacija obavljanja zadataka u više procesa, korištenjem upravitelja procesima.*

U tablici 4.4 dobiveni su vremenski rezultati izvođenja algoritma sortiranja u više procesa za sve odabrane vrijednosti N.

N	*Prosječno trajanje procesa [s]*	*Ukupno trajanje programa [s]*
10	0	0.112237
50	0.0000031	0.120298
100	0.0000118	0.116160
500	0.0003036	0.111227
1 000	0.0011857	0.122102
5 000	0.0287400	0.131699
10 000	0.1243081	0.245062
50 000	3.1214936	3.260105

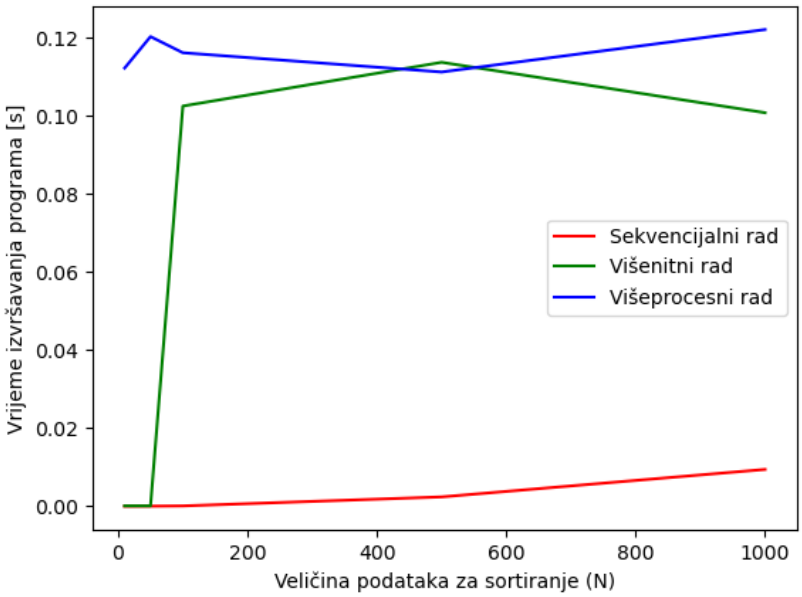
Tablica 4.4: *Rezultati dobiveni ispitivanjem višeprocesnog izvršavanja algoritma sortiranja.*



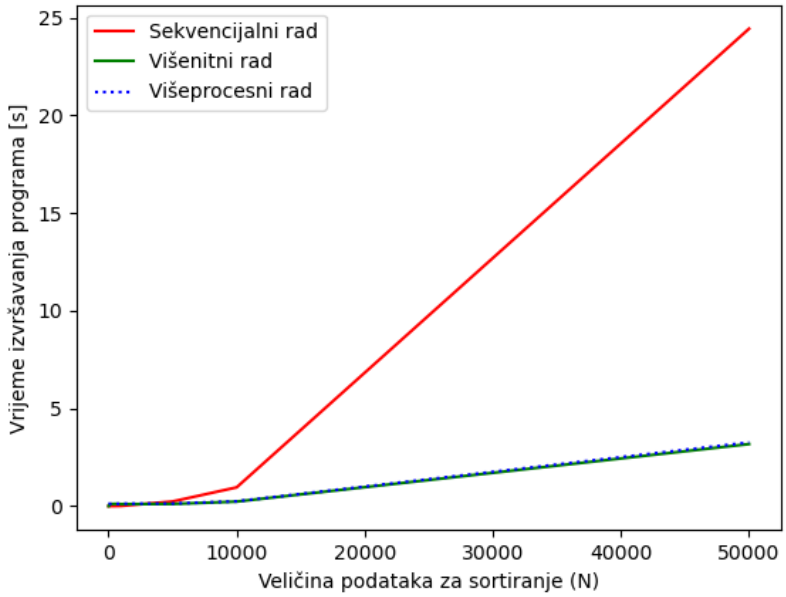
Slika 4.10: *Popis procesa i iskoristivost procesora tijekom višeprocesnog izvođenja programa.*

U tablici 4.4 i na slici 4.10 vidi se da i upravitelj procesima, kao upravitelj nitima postiže velike brzine i iskoristivosti procesora, međutim na drugačiji način. Umjesto jednog procesa koji koristi sve procesorske jezgre, rad programa raspodijeljen je u 8 zasebnih instanci procesa, te svaki od njih koristi jednu jezgru procesora. Prema tome, postojanje više instanci programa za testiranje rada upravitelja procesima (`ProcessManagerTest.exe`) dokaz je višeprocesnog rada. Paralelan rad programa u više procesa postiže prosječnu brzinu izvršavanja od $3.26\ s$ na veličini polja od $50\ 000$ elemenata. Time je od sekvencijalnog izvođenja programa brži čak do 7.5 puta. Isto kao za slučaj upravitelja nitima, upravitelj procesima nije isplativ za jednostavnije zadatke, jer također vrijeme inicijalizacije ima utjecaj na vrijeme izvršavanja procesa, čak i sporije od inicijalizacije niti.

Slika 4.11 daje grafički prikaz performansi svih programa za veličine polja do $N = 1\ 000$. Može se primijetiti da ovdje crvena linija (sekvencijalni način izvođenja) ostvaruje najbolje performanse. Međutim, na većim poljima za sortiranje vrijeme rada sekvencijalnog programa naglo raste i daleko premašuje vremena za slučaj izvođenja programa paralelno. Izvođenje programa pomoću više niti i više procesa gotovo je isto te se njihove linije gotovo poklapaju, kako je prikazano na slici 4.12.



Slika 4.11: *Grafički prikaz načina sortiranja za veličinu polja do 1000 elemenata.*



Slika 4.12: *Grafički prikaz načina sortiranja do 50 000 elemenata.*

5. ZAKLJUČAK

Moderna vozila sadrže velik broj elektroničkih komponenata, koje je potrebno je testirati kako bi se utvrdio njihov ispravan rad. Ručno testiranje ECU-ova je prilično složen posao pa se koriste generatori testnog okruženja kako bi se automatizirao postupak ispitivanja. Prije samog korištenja ECU-ova u automobilu, potrebno je najprije odraditi testiranje njihovog rada koristeći razne simulacije, odnosno testna okruženja, kako bi se provjerilo radi li komunikacija između kanala ECU-ova kako je očekivano. Upravo za tu svrhu koristi se generator testnog okruženja, odnosno TEG. Korištenjem TEG-a moguće je izrađivati različita testna okruženja i simulacije i na taj način učinkovito provoditi veliki broj testova, te kao posljedica toga povećati pouzdanost rada sustava. Temelj TEG-a predstavlja TEG okvir (*engl. Framework*), koji nudi set alata za siguran i efikasan rad ostalih dijelova TEG-a, te omogućuje optimalno korištenje resursa operacijskog sustava, kao i oporavak od mogućih pogrešaka. Predloženi TEG okvir sadrži pet modula, koji obavljaju različite zadaće. To su: bilježnik, rukovatelj datotekama, rukovatelj pogreškama, upravitelj nitima i upravitelj procesima. Pomoću bilježnika postignuto je zapisivanje poruka raznih razina ozbiljnosti na ekran ili u datoteku proizvoljnog imena čime je omogućen visok stupanj informatiziranosti programa te obavještavanja programera za važne događaje u stvarnome vremenu. Uz pomoć rukovatelja datotekama omogućena je naprednija, dinamičnija i efikasnija pretraga datoteka u odnosu na prijašnje verzije TEG-a, napisanih u Pythonu. Uvođenjem djelomičnih imena riješen je problem dinamičke pretrage. Implementacijom rukovatelja pogreškama postignuta je detekcija mjesta nastanka prilagođene pogreške, kao i standardnih pogrešaka, te je omogućeno njihovo rukovanje i otklanjanje, bez intervencije programera. Ipak, ovdje nije u potpunosti postignuto dinamičko izvršavanje djela koda koji je specifičan za rješavanje pojedine pogreške, zbog same arhitekture programskog jezika C++. Sav kod za rješavanje pogreški je unaprijed napisan unutar blokova za hvatanje pogrešaka. Implementacijom upravitelja nitima riješeni su nedostaci rada u više niti, koje prethodne verzije u Pythonu nisu posjedovale zbog ograničenja jezika. Također, pomoću upravitelja procesima postignuta je paralelizacija sa približno jednakim ubrzanjem kao što je postignuto upraviteljem nitima. Međutim upravitelj procesima ima puno veći potencijal za još veća ubrzanja na jačim i složenijim računalima. Upravitelj procesima oslanja se na biblioteku treće strane, koja se svaki puta mora prevoditi sa samim projektom, stvarajući tako zavisnost projekta o toj biblioteci. Za izbjegavanje zavisnosti o drugim bibliotekama, upravitelj procesima bilo bi poželjno nadograditi implementirajući vlastitu biblioteku, koja je kompatibilna na različitim platformama, a specijalizirana je za rad sa upraviteljem procesima.

LITERATURA

[1] A. Mihalj, »Generator softverskog koda namijenjenog za testiranje ADAS sustava,« Digitalni akademski arhivi i repozitoriji, Osijek, 2019.

[2] AUTOSAR GbR, »AUTOSAR Techincal Overwiev,« 2008.

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

[4] K. Vlašiček, »Optimizacija potrebne programske memorije kod automatski generiranih testova za AUTOSAR RTE,« Digitalni akademski arhiv i repozitorij, Osijek, 2020.

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

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

[7] L. Williams, "Multithreading vs Multiprocessing: What's the difference?," Guru99, 21 8 2021. [Online]. Available: https://www.guru99.com/difference-between-multiprocessing-and-multithreading.html.

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

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

[10] Refactoring.Guru, »Singleton in C++,« Refactoring.Guru, 2021. [Mrežno]. Available: https://refactoring.guru/design-patterns/singleton/cpp/example.

[11] cppreference.com, »Filesystem library,« cppreference.com, 24 June 2021. [Mrežno]. Available: https://en.cppreference.com/w/cpp/filesystem.

[12] cppreference.com, »std::filesystem::recursive_directory_iterator,« cppreference.com, 16 November 2020. [Mrežno]. Available: https://en.cppreference.com/w/cpp/filesystem/recursive_directory_iterator.

[13] M. Kerrisk, »find(1) - Linux manual page,« [Mrežno]. Available: https://man7.org/linux/man-pages/man1/find.1.html.

[14] GeeksForGeeks, »Difference between Process and Thread,« GeeksForGeeks, 1 April 2021. [Mrežno]. Available: https://www.geeksforgeeks.org/difference-between-process-and-thread.

[15] B. Shoshany, »A C++17 Thread Pool for High-Performance ScientificComputing,« Arxiv, 1812 Sir Isaac Brock Way, St. Catharines, Ontario, L2S 3A1, Canada, 2021.

[16] cplusplus, »std::function,« cplusplus, 2021. [Mrežno]. Available: https://www.cplusplus.com/reference/functional/function.

[17] C. Robertson, K. Sharkey, M. Jones, M. Blome, G. Hogenson i S. Cai, »Lambda expressions in C++,« Microsoft, 13 July 2021. [Mrežno]. Available: https://docs.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=msvc-160.

[18] J. M. M. Vidal, I. Sokolov, F. Tanus, J. Flinn, B. Schaeling i K. D. Morgenstern, »Boost.Process,« Boost C++ Libraries, 30 August 2017. [Mrežno]. Available: https://www.boost.org/doc/libs/1_65_1/doc/html/process.html.

[19] UserBenchmark, »WD10EZEX Benchmark,« UserBenchmark, [Mrežno]. Available: https://hdd.userbenchmark.com/WD-Blue-1TB-2012/Rating/1779.

[20] AUTOSAR Tutorials, »AUTOSAR Tutorials,« AUTOSAR Tutorials, [Mrežno]. Available: https://i0.wp.com/autosartutorials.com/wp-content/uploads/2019/10/Simplified-2BAUTOSAR-2BLayered-2BArchi.jpg.

SAŽETAK

Moderna vozila sadrže velik broj računalnih sustava ili ECU-a koji omogućuju udobniju, kvalitetniju, sigurniju i ekonomičniju vožnju. Smanjenjem dimenzija računala današnji automobili raspolažu sa više stotina ECU sustava, koji međusobno razmjenjuju informacije u stvarnome vremenu. Komunikacija među ECU-ovima propisana je AUTOSAR standardom. Sve te komponente u sustavu potrebno je testirati i utvrditi njihov rad na očekivani način prije same uporabe komponenti na samom vozilu. U tu svrhu koristi se generator testnog okruženja, odnosno TEG, kako bi omogućio učinkovito generiranje testnog okruženja i simulaciju za velik broj slučajeva. Temelj generatora testnog okruženja predstavlja TEG okvir (engl. *Framework*), koji nudi alate drugim dijelovima TEG-a za učinkovit i siguran tijek programa, kao i rukovanje resursima operacijskog sustava. TEG okvir sastoji se od pet modula koji su predviđeni za različite zadaće. To su: bilježnik (engl. *Logger*), rukovatelj datotekama (engl. *Error Handler*), rukovatelj pogreškama (engl. *Error Handler*), upravitelj nitima (engl. *Thread Manager*) i upravitelj procesima (engl. *Process Manager*). Spomenuti moduli implementirani su korištenjem C++ programskog jezika sa ciljem da unaprijede performanse TEG programa. Nakon implementacije provodi se ispitivanje i evaluacija svakog modula uz pomoć različitih zadataka i pomoćnih programa da bi se utvrdila cjelokupna funkcionalnost okvira.

ABSTRACT

Test Environment Generator development in C++ programming language

Modern vehicles have great number of computer systems or ECUs which enable more comfortable, safer and more economic driving experience. By reducing computers dimensions, today's cars possess hundreds of ECU systems, which mutually exchange information in real time. Communication among ECUs is prescribed by AUTOSAR standard. All these components in system need to be tested to establish whether they work as expected before the very use of those components within a vehicle. For that purpose test environment generator, or TEG for short, is used in order to enable efficient test environment generation and simulation for great number of cases. TEG foundation is presented by TEG Framework, which provides tools for other parts of TEG for more efficient and safer program flow, as well as operating system's resource handling. TEG Framework is made of five modules, which are designed for different tasks. These are: Logger, File Handler, Error Handler, Thread Manager and Process Manager. Aforementioned modules are implemented using C++ programming language with a goal to improve TEG program performances. After implementation, the module needs to be tested and evaluated with help of different tasks and test programs in order to establish entire TEG Framework functionality.

ŽIVOTOPIS

Filip Garmaz rođen je 26. lipnja 1996. godine u Osijeku, Hrvatska. Pohađa Osnovnu školu Bratoljuba Klaića u Bizovcu do 2011. godine te iste godine upisuje III. Gimnaziju u Osijeku. Srednju školu završava s vrlo dobrim uspjehom. Po završetku srednje škole, 2015. godine, upisuje tadašnji Elektrotehnički fakultet Osijek, danas Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek, smjer računarstvo. Preddiplomski studij završava 2019. godine te na istom fakultetu upisuje diplomski studij računarstva, smjer Robotika i umjetna inteligencija. Vrlo dobro se služi engleskim jezikom te ima srednju razinu znanja s Microsoft Office alatima. Ima visoku razinu znanja s programskim jezicima C, C++ i C#, srednju razinu znanja s programskim jezicima Python, Matlab, Pascal te opisnim jezikom HTML i stilskim jezikom CSS. Također je poznavatelj ostalih jezika kao što su PHP, SQL, LISP (ELISP) i asembler. Godine 2019./2020. dobitnik je stipendije RT-RK iz Osijeka te se trenutno u istoj tvrtci obučava za daljnji rad nakon diplome.