UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Ognjen Ž. Plavšić

ALAT ZA STATIČKU ANALIZU I PREDLAGANJE IZMENA U C++ KODU

master rad

Beograd, 2022.

Mentor:

dr Milena Vujošević Janičić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Filip Marić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Jelena Graovac, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Porodici

Naslov master rada: Alat za statičku analizu i predlaganje izmena u C++ kodu

Rezime: Standardi za pravilno pisanje C++ koda sve su zastupljeniji u industrijama koje razvijaju sisteme sa ugrađenim računarom (eng. *embedded systems*) i druge sisteme sa kritičnom bezbednošću. Standard AUTOSAR C++14 jedan je od vodećih standarda ovog tipa. Primarno se koristi u automobilskoj industriji, odnosno za razvoj softvera za automobile. Široka upotreba ovih standarda izrodila je potrebu za alatkama za statičku analizu koji bi automatizovali proces provere da li je kôd napisan u skladu sa standardom. Kompilatorska infrastruktura LLVM pruža podršku za jednostavno razvijanje kvalitetnih alatki ovog tipa. Cilj ovog rada je implementacija alata za statičku analizu *AutoFix* koji proverava da li je kôd napisan u skladu sa podskupom pravila koja se odnose na deklaracije u okviru standarda AUTOSAR C++14. Alat ispisuje upozorenja za delove koda koji nisu u skladu sa nekim od pravila i predlaže izmene tog koda.

Ključne reči: verifikacija softvera, statička analiza programa, programski jezik C++, standardi kodiranja, AUTOSAR, kompilatori, LLVM, Clang

Sadržaj

Glava 1

Uvod

Softver je neizostavni deo modernog sveta. U zavisnosti od primene softvera i konteksta u kome se koristi, kvalitet softvera može da igra manju ili veću ulogu. Na primer, greške u video igri imaće za posledicu samo nezadovoljstvo korisnika, dok greške u softveru za kontrolisanje kočnica automobila mogu imati fatalne posledice. Iz ovog razloga određene industrije ulažu dodatni napor kako bi se uverile u kvalitet softvera i kako bi smanjili mogućnost pojave greške.

Način da se smanji mogućnost pojave greške u kodu jeste definisanje strogog pravila kodiranja u izabranom programskom jeziku. Jedan takav standard za programski jezik C++14 (jezik C++ iz verzije standarda za 2014. godinu) predstavlja standard kodiranja AUTOSAR C++14. Ovaj standard primarno se primenjuje u automobilskoj industriji. S obzirom da industrije koje primenjuju ovaj standard često koriste jako kompleksne softvere, potrebno je automatizovati proces provere da li je kôd napisan u skladu sa standardom. U ovu svrhu koriste se alatke za statičku analizu koda.

Alatke za statičku analizu proveravaju ispravnost programa bez njegovog izvršavanja. Ovakve alatke se mogu implementirati na više načina. Jednostavnije alatke koriste informacije dobijene tokom kompilacije programa da utvrde da li kôd sadrži grešku. Naprednije alatke za statičku analizu vrše i simboličko izvršavanje programa, odnosno koriste različite tehnike kojima simuliraju izvršavanje programa, bez njegovog pokretanja [23, 20].

Kompilatorska infrastruktura LLVM omogućava izradu alatki za statičku analizu. Ova infrastruktura sadrži niz biblioteka koje omogućavaju analizu informacija dobijenih tokom kompilacije, ali sadrži i biblioteke koje omogućavaju izradu samostalnih alatki. Cilj ovog rada je implementacija alata za statičku analizu *Au-*

toFix. *AutoFix* proverava da li je kôd napisan u skladu sa podskupom pravila iz standarda AUTOSAR C++14 koja se odnose na deklaracije u programskom jeziku C++14. Ukoliko kôd nije napisan u skladu sa nekim od tih pravila, alat prijavljuje upozorenje zajedno sa predlogom za ispravljanje koda.

U glavi 2 opisan je programski jezik C++ i standard kodiranja AUTOSAR C++14. Opisana je klasifikacija pravila u okviru standarda i navedeni su primeri pravila koja pripadaju svakoj od grupa u okviru klasifikacije. U glavi 3 opisani su delovi kompilatorske strukture LLVM koji su korišćeni za izradu alata *AutoFix*. U glavi 4 opisana je implementacija alata *AutoFix* i način upotrebe alata. Takođe, opisano je i svako od pravila iz standarda AUTOSAR C++14 koje alat podržava. U zaključku iznet je osvrt na ceo rad i predložen je dalji tok razvoja alata *AutoFix*.

Glava 2

Standard kodiranja AUTOSAR C++14

*AUTomotive Open System ARchitecture (*AUTOSAR*)* je međunarodna organizacija proizvođača vozila, dobavljača, pružaoca usluga i kompanija iz automobilske industrije i industrija elektronike, poluprovodnika i softvera [19]. Cilj organizacije je da stvori i uspostavi otvorenu i standardizovanu softversku arhitekturu za automobilske elektronske upravljačke jedinice (*eng. Electronic Control Units, skraćeno ECU*). Radi ostvarenja pomenutih ciljeva AUTOSAR definiše, između ostalog, pravila kodiranja u programskom jeziku C++14 za sisteme sa kritičnom bezbednošću. Glavni sektor primene standarda kodiranja AUTOSAR C++14 je automobilska industrija, međutim ovaj standard može biti primenjen i na druge aplikacije za sisteme sa ugrađenim računarom. Ovaj standard predstavlja nadogradnju standarda MISRA C++:2008 [18].

2.1 Programski jezik C++

C++ je programski jezik opšte namene. Kreirao ga je danski softverski inženjer Bjarne Stroustrup kao ekstenziju programskog jezika C. U trenutku kreiranja, osnovno proširenje u odnosu na programski jezik C bile su klase. C++ pripada grupi objektno orijentisanih jezika.

Dizajn programskog jezika C++

Programski jezik C++ zadržava osnovne ideje i koncepte jezika C. Takođe, jezik pruža sintaksu koja omogućava direktan i koncizan pristup problemu koji rešava. U svrhu toga, C++ pruža:

- Direktna preslikavanja ugrađenih operacija i tipova na hardver kako bi obezbedio efikasno korišćenje memorije i efikasne operacije niskog nivoa (eng. *low-level operations*).
- Priuštive (u smislu računarskih resursa) i fleksibilne mehanizme apstrakcija za podršku korisnički definisanih tipova koji se mogu koristiti sa istom sintaksom, u istom obimu i sa istim performansama kao ugrađeni tipovi.

Dizajn jezika C++ je fokusiran na tehnike programiranja koje se bave osnovnim pojmovima računarstva kao što su memorija, mutabilnost, apstrakcija, upravljanje računarskim resursima, izražavanje algoritama, rukovanje greškama i modularnost. Jezik je dizajniran sa ciljem da što više olakša sistemsko programiranje, odnosno pisanje programa koji direktno koriste hardverske resurse i kod kojih su ovi resursi u velikoj meri ograničeni [22].

Standard C++14

Programski jezik C++ je standardizovan. U okviru međunarodne organizacije za standardizaciju (eng. *International Standard Organization*, skraćeno ISO), standard za programski jezik C++ propisuje radna grupa poznata kao JTC1/SC22/WG21 [13]. Do sada je objavljeno šest revizija C++ standarda i trenutno se radi na reviziji C++23.

Standard C++14 predstavlja proširenje standarda C++11 uglavnom manjim poboljšanjima i ispravljanjem grešaka iz standarda C++11. Standard C++11 sa druge strane uveo je velike izmene u odnosu na prethodnu reviziju standarda, C++03.

Standardi C++11/14 uveli su većinu fundamentalnih koncepta onog što se danas smatra modernim jezikom C++. Ovde spadaju desne reference, „move" semantika i savršeno prosleđivanje, pametni pokazivači, lambda funkcije, dedukcija tipova ali i mnogi drugi koncepti.

2.2 Klasifikacija pravila

Standard AUTOSAR C++14 definiše 342 pravila kodiranja u programskom jeziku C++14. Od toga je:

- 154 pravila prisvojeno bez modifikacija iz standarda MISRA C++:2008,
- 131 pravila prisvojeno iz drugih C++ standarda,
- 57 pravila je zasnovano na istraživanju, literaturi ili je preuzeto iz drugih resursa.

U nastavku su prikazana tri pravila iz standarda AUTOSAR C++14. Pravila su izabrana nasumično i prikazana su kako bi čitalac stekao inicijalnu ideju o pravilima iz standarda AUTOSAR C++14.

A5-0-1
Vrednost izraza treba biti ista u bilo kom redosledu evaluacije koji standard jezika C++ dozvoljava.

A5-5-1
Desni operand celobrojnog deljenja ili operatora ostatka pri deljenju ne sme biti jednak nuli.

A23-0-1
Iterator ne sme biti implicitno konvertovan u `const_iterator`.

Pravila su klasifikovana po nivou obaveze, mogućnosti ispitivanja saglasnosti koda sa pravilom korišćenjem algoritama statičke analize i cilju korišćenja.

Klasifikacija po nivou obaveze

Klasifikacija po nivou obaveze deli pravila na obavezna i preporučena. Obavezna pravila predstavljaju neophodne zahteve koje C++ kôd mora ispuniti kako bi bio u saglasnosti sa standardom. U slučaju kada ovo nije moguće, formalna odstupanja moraju biti prijavljena. Preporučena pravila predstavljaju zahteve koje

C++ kôd treba da ispuni kad god je to moguće. Međutim, ovi zahtevi nisu obavezni. Pravila sa ovim nivoom obaveze ne treba smatrati savetom ili sugestijom koja može biti ignorisana već ih treba pratiti uvek kada je to praktično izvodljivo. Za ova pravila ne moraju biti prijavljena formalna odstupanja.

Klasifikacija po primenjivosti statičke analize

Klasifikacija po primenjivosti statičke analize deli pravila na:

1. automatizovana
2. delimično automatizovana
3. neautomatizovana

Automatizovana su ona pravila kod kojih se ispitivanje saglasnosti koda može u potpunosti automatizovati algoritmima statičke analize. Kod delimično automatizovanih pravila se ispitivanje saglasnosti koda može samo delimilčno automatizovati, na primer, korišćenjem neke heuristike ili pokrivanjem određenog broja slučajeva upotrebe i služi kao dopuna pregledu koda od strane programera. Za neautomatizovana pravila statička analiza ne pruža razumnu podršku. Za ispitivanje saglasnosti koda sa neautomatizovanim pravilima koriste se druga sredstva, kao što je recimo pregled koda od strane programera.

Većina pravila iz standarda AUTOSAR C++14 spadaju u automatizovana pravila. Alatke za statičku analizu koda koje tvrde da podržavaju standard AUTOSAR C++14 treba da u potpunosti obezbede podršku za sva automatizovana pravila i delimičnu podršku, u meri u kojoj je to moguće, za pravila koja se ne mogu u potpunosti ispitati algoritmima statičke analize [18].

Primenjivost statičke analize na proveru saglasnosti koda sa određenim pravilom u velikoj meri zasniva se na teorijskoj klasifikaciji problema na odlučive i neodlučive probleme. Ukoliko se pravilo zasniva na neodlučivom problemu možemo sa sigurnošću reći da alatke za statičku analizu nisu u mogućnosti da u potpunosti ispitaju saglasnost koda sa ovim pravilom. Pravilo će biti klasifikovano kao parcijalno automatizovano ili neautomatizovano ukoliko detektovanje kršenja pravila obuhvata određivanje vrednosti koju promenljiva sadrži u fazi izvršavanja ili da li izvršavanje doseže određeni deo programa.

Primer parcijalno automatizovanog pravila je:

M5-8-1 (obavezno, parcijalno automatizovano)
Desni operand šift operacije treba biti manji za broj između nula i jedan od bitske širine tipa levog operanda.

Pravilo nije moguće u potpunosti automatizovati jer je potrebno poznavati vrednost desnog operanda, što u opštem slučaju nije moguće precizno zaključiti. Primer ovakvog koda prikazan je na listingu 2.1.

Listing 2.1: Kôd za koji statička analiza u opštem slučaju ne može da dâ precizne rezultate.

```
#include <iostream>
#include <cstdint>
#include <cstdlib>

int main(){
  int8_t u8a = rand() % 100;
  u8a = (uint8_t) (u8a << rand() % 10);
}
```

Međutim, ukoliko je desni operand konstanta ili promenljiva konstantnog izraza (ključna reč *constexpr*), alat za statičku analizu može da proveri vrednost ove promenljive (s obzirom da su ove vrednosti poznate tokom kompilacije), a samim tim i ispitati saglasnost koda sa ovim pravilom. Primer ovakvog koda prikazan je na listingu 2.2.

Listing 2.2: Kôd čija se ispravnost jednostavno može utvrditi statičkom analizom.

```
#include <iostream>
#include <cstdint>
#include <cstdlib>

int main(){
  int8_t u8a = rand() % 100;
  u8a = (uint8_t) (u8a << 7);
}
```

Naprednije alatke za statičku analizu koje podržavaju simboličko izvršavanje programa (npr. *Clang Static Analyzer* [5]) mogu pokriti i znatno kompleksnije slučajeve od slučaja prikazanog na listingu 2.2.

Ukoliko su pravila koja se odnose na implementaciju C++ projekta, odnosno na C++ konstrukte i semantiku programa, dovoljno kompleksna, može se desiti da u potpunosti nije moguće koristiti alatke za statičku analizu. Ovo uglavnom znači da je broj slučajeva upotrebe koji algoritmi iz alatki za statičku analizu mogu pokriti, zanemarljiv. Međutim, određeni broj pravila koja su klasifikovana kao neautomatizovana odnose se na aspekte koda koji zavise od samog projekta u okviru kog je kôd napisan, stoga je nemoguće koristiti algoritme statičke analize. Primer ovakvog pravila je:

> Pravilo A1-4-2 (obavezno, neautomatizovano)
> Kôd treba da poštuje zadate granice metrika koda.

Kako bi se odredilo da li je kôd napisan u skladu sa ovim pravilom potrebno je poznavati koje metrike koda se koriste u okviru projekta i granice definisane za te metrike. S obzirom da je ovo specifično za sam projekat, mogu se koristiti interne alatke za statičku analizu koda u kombinaciji sa pregledom koda od strane programera.

Klasifikacija pravila prema cilju primene

Klasifikacija pravila prema cilju primene (slučaju upotrebe) deli pravila na:

1. implementaciona,
2. verifikaciona,
3. pravila za alatke,
4. infrastrukturna.

Implementaciona pravila se odnose na implementaciju projekta odnosno na kôd, arhitekturu i dizajn. Primer implementacionog pravila:

Pravilo A2-9-1 (obavezno, implementaciono, automatizovano)
Ime zaglavlja mora biti identično imenu tipa deklarisanog u njemu ukoliko deklariše tip.

Verifikaciona pravila odnose se na proces provere koji uključuje pregled koda, analizu i testiranje. Primer verifikacionog pravila:

Pravilo A15-0-6 (obavezno, verifikaciono, neautomatizovano)
Analiza treba biti izvršena kako bi se detektovalo loše rukovanje izuzecima. Treba analizirati sledeće slučajeve lošeg rukovanja izuzecima:
(a) Najgore vreme izvršavanja ne postoji ili se ne može utvrditi,
(b) Stek nije korektno raspakovan,
(c) Izuzetak nije bačen, drugačiji izuzetak je bačen, aktivirana je pogrešna „catch" naredba,
(d) Memorija nije dostupna tokom rukovanja izuzecima.

Pravila za alatke odnose se na softverske alatke kao što su pretprocesor, kompilator, linker i biblioteke kompilatora. Infrastrukturna pravila odnose se na operativni sistem i hardver [18]. Primer pravila za alatke koje je ujedno i infrastrukturno pravilo:

Pravilo A0-4-1 (obavezno, pravilo za infrastrukturu/alatke, neautomatizovano)
Implementacija brojeva u pokretnom zarezu treba da bude u skladu sa standardom IEEE 754.

Glava 3

Statička analiza i LLVM

U ovom poglavlju opisane su biblioteke i klase kompilatorske infrastrukture LLVM koje su korišćene za implementaciju alata za statičku analizu *AutoFix*. Biblioteke su opisivane ukoliko su u celosti bitne za implementaciju. Ukoliko nisu bitne u celosti, opisivane su samo klase tih biblioteka koje implementiraju funkcionalnosti koje alat koristi.

S obzirom na to da se alat *AutoFix* zasniva na analizi apstraktnog sintaksičkog stabla, u ovom poglavlju opisana je biblioteka `clangAST` koja implementira osnovne strukture i algoritme za konstrukciju stabla i njegov obilazak. U okviru ove biblioteke posebno je objašnjena klasa `RecursiveASTVisitor` koja omogućava obilazak stabla. Opisana je i biblioteka `LibASTMatchers` koja implementira jezik specijalne namene (eng. *domain specific language*) kojim se mogu pronaći i obraditi specifične sintaksne strukture iz apstraktnog sintaksičkog stabla.

Apstraktne klase `ASTConsumer` i `FrontendAction` omogućavaju interakciju alata sa prednjim delom kompilatora. Alat *AutoFix* ih koristi u kontekstu kreiranja i izvršavanja akcija nad apstraktnim sintaksnim stablom.

Za kreiranje alata *AutoFix* korišćena je i biblioteka `LibTooling` koja u okviru infrastrukture LLVM omogućava kreiranje samostalnih alatki. Pored korišćenja ove biblioteke, alatke se mogu kreirati i upotrebom dodataka kompilatora *Clang* (eng. *Clang Plugins*) ili upotrebom biblioteke `LibClang`. U okviru ovog poglavlja diskutovane su prednosti i mane upotrebe ovih metoda u svrhu kreiranja alata kao i razlozi zbog kojih je biblioteka `LibTooling` izabrana za implementaciju alata *AutoFix*.

3.1 LLVM i Clang

Kompilatorska infrastruktura LLVM predstavlja kolekciju modularnih i ponovo iskoristivih kompilatorskih tehnologija i alatki. Ova kompilatorska infrastruktura započeta je kao instraživački projekat Krisa Latnera (eng. *Chris Lattner*) i Vikrama Advea (eng. *Vikram Adve*) na Univerzitetu Ilinois 2000. godine. Dizajn LLVM-a omogućava jednostavno dodavanje podrške za kompilaciju za specifičnu arhitekturu hardvera. Kompilatorska infrastruktura ugrubo je podeljena na tri dela: prednji (eng. *frontend*), srednji (eng. *middle-end*) i zadnji (eng. *backend*).

1. Prednji deo LLVM-a prevodi izvorni kôd podržanih jezika u LLVM međukod. U ovu fazu spadaju leksička, sintaksna i semantička analiza izvornog koda, kreiranje apstraktnog sintaksičkog stabla (eng. *abstract syntax tree* (AST)) i generisanje LLVM međukoda (eng. *intermediate representation (IR))* koristeći informacije iz apstraktnog sintaksičkog stabla.

2. Srednji deo kompilatora vrši niz optimizacija nad instrukcijama LLVM međukoda. LLVM međukod predstavlja apstrakciju asemblera koja je nezavnisna od arhitekture hardvera. LLVM međukod zasnovan je na svojstvu jedinstvenog statičkog dodeljivanja vrednosti (eng. *static single assignment*, skraćeno *ssa*), strogo je tipiziran, fleksibilan i omogućava jednostavnu reprezentaciju svih jezika visokog nivoa (eng. *high-level languages*).

3. Zadnji deo kompilatora vrši mašinski zavisne optimizacije koda i generiše mašinski kôd za ciljnu arhitekturu.

Clang predstavlja prednji deo (*eng. frontend*) kompilatorske infrastrukture LLVM za familiju jezika u čijoj se osnovi nalazi programski jezik C (C, C++, Objective C/C++, OpenCL ...). Pored optimizacija i efikasnog generisanja LLVM međukoda, *Clang* odlikuje i ekspresivnost dijagnostike odnosno kvalitet poruka upozorenja i grešaka prijavljenih za izvorni kôd. *Clang* se sastoji od više biblioteka od kojih su najznačajnije nabrojane u nastavku.

Biblioteka `clangLex` sadrži nekoliko usko povezanih klasa koje implementiraju pretprocesiranje i leksičku analizu izvornog koda. Najvažnije klase u okviru ove biblioteke su `Lexer` i `Preprocessor`. `Preprocessor` pruža mogućnost uslovne kompilacije, uključivanja datoteka zaglavlja i proširenja makroa. `Lexer` kreira niz tokena od izvornog koda.

Biblioteka `clangParse` obrađuje niz tokena dobijenih leksičkom analizom i od njih kreira čvorove apstraktnog sintaksičkog stabla. Ova biblioteka koristi funkcionalnosti biblioteke `clangSema` kako bi ispitala semantičku validnost sintaksnih konstrukta (niza tokena) od kojih kreira čvorove apstraktnog sintaksičkog stabla. Parser kompilatora *Clang* je implementiran kao parser rekurzivnog spuštanja (eng. *recursive-descent parser*), odnosno analizira izvorni kôd od vrha ka dnu nizom rekurzivnih funkcija [21].

Biblioteka `clangAST` implementira algoritme i strukture podataka koje parser koristi za izgradnju apstraktnog sintaksičkog stabla. Specifična je po strukturi čvorova koji podsećaju na izvorni C++ kôd što je čini pogodnom za kreiranje alatki za refaktorisanje koda i statičku analizu. S obzirom da se ova biblioteka koristi u okviru alata *AutoFix*, opisana je detaljnije u poglavlju 3.2.

Biblioteka `clangSema` vrši semantičku analizu programa tokom parsiranja. Ova biblioteka proverava da li je kôd napisan u skladu sa sistemom tipova koji standard jezika propisuje. Za razliku od uobičajenog načina implementacije provere tipova, obilaskom apstraktnog sintaksičkog stabla nakon parsiranja, biblioteka `clangSema` implementira proveru tipova zajedno sa generisanjem čvorova apastraktong sintaksičkog stabla [21]. Ova biblioteka usko je povezana sa bibliotekama `clangParse` i `clangAST`.

Biblioteka `clangCodeGen` generiše LLVM međukod. Ova biblioteka obilazi apstraksno sintaksičko stablo i na osnovu njegovog sadržaja generiše instrukcije LLVM međukoda koje implementiraju ponašanje opisano u stablu [21].

3.2 Biblioteka `clangAST`

U računarstvu, **apstraktno sintaksičko stablo**, ili samo **sintaksičko stablo**, je drvoidna reprezentacija apstraktne sintaksne strukture izvornog koda napisanog u programskom jeziku. Svaki čvor stabla predstavlja konstrukt koji se pojavljuje u izvornom kodu. Sintaksa je apstraktna u smislu da ne sadrži svaki detalj koji se pojavljuje u sintaksi, ali sadrži sve detalje neophodne za nedvosmislen prikaz izvornog koda.

Ekspresivnost dijagnostike kompilatora *Clang* i jednostavnost kreiranja moćnih alatki za statičku analizu u velikoj meri oslanja se na dizajn biblioteke `clangAST`.

Struktura apstraktnog sintaksičkog stabla može se jednostavno ispisati na standardni izlaz opcijom komandne linije `-ast-dump`. Slika 3.1 predstavlja tekstualnu reprezentaciju apstraktnog sintaksičkog stabla generisanog za kôd iz fajla `hello.cpp` prikazanog na listingu 3.1.

Listing 3.1: Kôd čije je apstraktno sintaksičko stablo prikazano na slici 3.1.

```
1 int main(){
2   int a = 4;
3   int b = 5;
4   int result = a * b + 8;
5 }
```

```
TranslationUnitDecl 0x5572e9d09c28 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x5572e9d0a490 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x5572e9d0a1f0 '__int128'
|-TypedefDecl 0x5572e9d0a500 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x5572e9d0a210 'unsigned __int128'
|-TypedefDecl 0x5572e9d0a878 <<invalid sloc>> <invalid sloc> implicit __NSConstantString '__NSConstantString_tag'
| `-RecordType 0x5572e9d0a5f0 '__NSConstantString_tag'
|   `-CXXRecord 0x5572e9d0a558 '__NSConstantString_tag'
|-TypedefDecl 0x5572e9d0a910 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x5572e9d0a8d0 'char *'
|   `-BuiltinType 0x5572e9d09cd0 'char'
|-TypedefDecl 0x5572e9d4ffa8 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list '__va_list_tag[1]'
| `-ConstantArrayType 0x5572e9d4ff50 '__va_list_tag[1]' 1
|   `-RecordType 0x5572e9d0aa00 '__va_list_tag'
|     `-CXXRecord 0x5572e9d0a968 '__va_list_tag'
`-FunctionDecl 0x5572e9d50050 </home/ognjen/AutoFixTest.cpp:1:1, line:5:1> line:1:5 main 'int ()'
  `-CompoundStmt 0x5572e9d50438 <col:12, line:5:1>
    |-DeclStmt 0x5572e9d50200 <line:2:1, col:10>
    | `-VarDecl 0x5572e9d50178 <col:1, col:9> col:5 used a 'int' cinit
    |   `-IntegerLiteral 0x5572e9d501e0 <col:9> 'int' 4
    |-DeclStmt 0x5572e9d502b8 <line:3:1, col:10>
    | `-VarDecl 0x5572e9d50230 <col:1, col:9> col:5 used b 'int' cinit
    |   `-IntegerLiteral 0x5572e9d50298 <col:9> 'int' 5
    `-DeclStmt 0x5572e9d50420 <line:4:1, col:23>
      `-VarDecl 0x5572e9d502e8 <col:1, col:22> col:5 result 'int' cinit
        `-BinaryOperator 0x5572e9d50400 <col:14, col:22> 'int' '+'
          |-BinaryOperator 0x5572e9d503c0 <col:14, col:18> 'int' '*'
          | |-ImplicitCastExpr 0x5572e9d50390 <col:14> 'int' <LValueToRValue>
          | | `-DeclRefExpr 0x5572e9d50350 <col:14> 'int' lvalue Var 0x5572e9d50178 'a' 'int'
          | `-ImplicitCastExpr 0x5572e9d503a8 <col:18> 'int' <LValueToRValue>
          |   `-DeclRefExpr 0x5572e9d50370 <col:18> 'int' lvalue Var 0x5572e9d50230 'b' 'int'
          `-IntegerLiteral 0x5572e9d503e0 <col:22> 'int' 8
```

Slika 3.1: Apstraktno sintaksičko stablo za kôd iz listinga 3.1 koje je generisano komandom: `clang -Xclang -ast-dump hello.c`.

Čvorovi od kojih je izgrađeno apstraktno sintaksičko stablo predstavljaju apstrakciju sintaksnih struktura iz samog jezika. Svi čvorovi apstraktnog sintaksičkog stabla kompilatora *Clang* nasleđuju jednu od tri osnovne (bazne) klase:

- `Decl`

- `Stmt`
- `Type`

Ove klase redom opisuju deklaracije, naredbe i tipove iz familije jezika u čijoj se osnovi nalazi jezik C. Na primer, klasa `IfStmt` opisuje naredbe `if` u jeziku i direktno nasleđuje klasu `Stmt`. Sa druge strane, klase `FunctionDecl` i `VarDecl`, koje se koriste za opisivanje deklaracija i definicija funkcija i varijabli, ne nasleđuju dikertno klasu `Decl` već nasleđuju više njenih podklasa.

Čvorovi apstraktnog sintaksičkog stabla dugog životnog veka (eng. *long-lived*), kao što su tipovi i deklaracije, čuvaju se u klasi `ASTContext`. Ova klasa omogućava upotrebu tih čvorova tokom semantičke analize programa. `ASTContext` takođe čuva referencu na objekat klase `SourceManager`. Ovo je čini pogodnom i za prikupljanje informacija o lokacijama iz izvornog fajla koje odgovaraju čvorovima iz apstraktnog sintaksičkog stabla. Ovakve informacije posebno su korisne za kreiranje preciznih poruka dijagnostike koda.

Klasa `Type`

Klasa `Type` igra važnu ulogu u ekspresivnosti dijagnostike kompilatora *Clang*, a samim tim i u kvalitetu alatki za statičku analizu. Ova klasa omgućava da poruke upozorenja sadrže precizne informacije o tipovima. Na primer, upozorenja vezana za kôd koji koristi tip `std::string`, ispisaće baš taj tip u svojim porukama umesto tipa koji `std::string` predefiniše, a to je `std::basic_string<char, ... >`. Iza ove funkcionalnosti stoji ideja kanonskih tipova.

Svaka instanca klase `Type` sadrži pokazivač na svoj kanonski tip. Za jednostavne tipove koji nisu definisani korišćenjem naredbe `typedef`, pokazivač na kanonski tip će zapravo pokazivati na sebe. Za tipove čija struktura uključuje naredbu `typedef`, kanonski pokazivač pokazivaće na strukturno ekvivalentan tip bez naredbe `typedef`. Na primer, kanonski tip tipa `int *` sa listinga 3.2 biće sam taj tip, dok će kanonski tip za `foo *` biti `int *`.

Listing 3.2: Primer kanonskog tipa (`int *`) i tipa koji nije kanonski (`foo *`).

```
1  int *a;
2  typedef int foo;
3  foo *b;
```

Ovakav dizajn omogućava semantičkim proverama da donose zaključke direktno o pravom tipu ignorišući naredbe `typedef`, kao i efikasno poređenje strukturne identičnosti tipova.

Klasa `Type` ne sadrži informacije o kvalifikatorima tipova kao što su `const`, `volatile`, `restrict` itd. Ove informacije enkapsulirane su u klasi `QualType` koja predstavlja par pokazivača na tip (objekat klase `Type`) i bitova koji predstavljaju kvalifikatore. Čuvanje kvalifikatora u vidu bitova omogućava veoma efikasno dohvatanje, dodavanje i brisanje kvalifikatora za tip. Postojanje ove klase smanjuje upotrebu hip memorije time što se ne moraju kreirati duplikati tipova sa različitim kvalifikatorima. Na hipu se alocira jedan tip, a zatim svi kvalifikovani tipovi pokazuju na alocirani tip na hipu sa dodatim kvalifikatorima [17].

AST-posetioci

AST-posetioci (eng. *AST-visitors*) implementiraju mehanizam obilaska apstraktnog sintaksičkog stabla kompilatora *Clang*, odnosno pružaju interfejs [1] za posetu svakog čvora u apstraktnom sintaksičkom stablu. Funkcionalnost AST-posetioca implementirana je u okviru šablonske klase `RecursiveASTVisitor<Derived>`. Objekat ove klase posećuje svaki čvor apstraktnog sintaksičkog stabla obilaskom u dubinu. AST-posetilac je svaka potklasa klase `RecursiveASTVisitor<Derived>`. Klasa `RecursiveASTVisitor<Derived>` omogućava obavljanje tri odvojena zadatka:

1. Obilazak apstraktnog sintaksičkog stabla, odnosno posećivanje svakog čvora.
2. Obilazak klasne hijerarhije za čvor, počevši od dinamičkog tipa čvora do klase na vrhu hijerarhije (npr. `Stmt`, `Decl` ili `Type`).
3. Za datu kombinaciju *(čvor, klasa)* omogućava pozivanje funkcije koje korisnik može predefinisati kako bi izvršio analizu čvora.

Ova tri zadatka obavljaju tri grupe metoda, redom:

1. Metode `TraverseDecl(Decl *x)`, `TraverseStmt(Stmt *x)` i `TraverseType-(QualType x)` implementiraju obilazak, redom, deklaracija, izraza i tipova u okviru apstraktnog sintaksičkog stabla. Ovo su ulazne tačke za obilazak

[1] U ovom radu pod terminom *interfejs* smatra se skup metoda ili funkcija koje pružaju pristup određenim funkcionalnostima i omogućavaju njihovu upotrebu.

apstraktnog sintaksičkog stabla sa korenom u čvoru `x`. Ove metode pozivaju metod
`TraverseFoo(Foo *x)`, gde je `Foo` dinamički tip od `*x`, koji poziva metod `WalkUpFromFoo(x)`, a zatim rekurzivno posećuje decu čvora `x`.
Na primer, ukoliko je dinamički tip od `*x` tip `CXXRecordDecl`, metod `TraverseDecl(Decl *x)`
će pozvati metod
`TraverseCXXRecordDecl(CXXRecordDecl *x)`
koji će pozvati metod
`WalkUpFromCXXRecordDecl(CXXRecordDecl *x)`.

2. Metod `WalkUpFromFoo(Foo *x)` obilazi klasnu hijerarhiju za čvor `x`. Ovaj metod ne pokušava odmah da poseti decu čvora `x`, umesto toga prvo zove `WalkUpFromBar(x)`, gde je `Bar` direktna nadklasa klase `Foo`, i tek onda zove `VisitFoo(x)`.
   Na primer, `WalkUpFromCXXRecordDecl(CXXRecordDecl *x)` poziva `WalkUpFromRecordDecl(x)` i `VisitCXXRecordDecl(x)`.

3. Metod `VisitFoo(Foo *x)` analizira čvor `x` tipa `Foo`.
   Na primer, metod `VisitCXXRecordDecl(CXXRecordDecl *x)` može biti predefinisan kako bi se analizirao čvor `x` tipa `CXXRecordDecl`.

Za ove tri grupe metoda definiše se naredni poredak: `Traverse > WalkUpFrom > Visit`. Ovaj poredak označava da metod može pozvati samo metode iz svoje grupe metoda ili iz grupe metoda direktno ispod nje. Metod ne može pozvati metode iz grupe iznad [8]. Na primer, metod `WalkUpFrom` može pozvati metode iz svoje grupe i grupe `Visit` ali ne može pozvati metode iz grupe `Traverse`. Metode iz grupe `Traverse` ne mogu pozvati metode iz grupe `Visit`, jer iako je grupa metoda `Visit` u definisanom poretku ispod grupe metoda `Traverse`, između te dve grupe metoda nalazi se grupa `WalkUpFrom`. Drugim rečima, u definisanom poretku grupa metoda `Visit` nije direktno ispod grupe metoda `Traverse`.

Primer implementacije AST-posetioca

Da bi se izvršila analiza izvornog koda pomoću AST-posetioca potrebno je naslediti klasu `RecursiveASTVisitor<Derived>` i predefinisati željene metode `Visit` u okviru nje. Ukoliko je metodama `Visit` pronađen nepravilan konstrukt izvornog

koda može se prijaviti upozorenje. Na listingu 3.3 prikazan je primer posetioca. Metod `VisitEnumDecl` (linija 6) će biti pozvan nad svim objektima klase `EnumDecl` u apstraktnom sintaksičkom stablu. U okviru nje, proverava se da li deklaracija koristi sintaksu `enum class` [2], odnosno da li su konstante u okviru nabrojivih tipova definisane u svom unutrašnjem opsegu. Ukoliko sintaksa `enum class` nije korišćena pri deklaraciji, lokacija ove deklaracije, ukoliko je validna, ispisuje se na standardni izlaz (linija 13).

Listing 3.3: Primer posetioca koji posećuje sve deklaracije nabrojivih tipova i ispisuje lokaciju onih koji nisu deklarisani sintaksom `enum class`.

```
1  class FindUnscopedEnumVisitor
2      : public RecursiveASTVisitor<FindUnscopedEnumVisitor> {
3  public:
4    explicit FindUnscopedEnumVisitor(ASTContext *Context) : Context(
       Context) {}
5
6    bool VisitEnumDecl(EnumDecl *ED) {
7      if (!ED->isScopedUsingClassTag()) {
8        // Get declaration location.
9        FullSourceLoc FullLocation =
10           Context->getFullLoc(ED->getBeginLoc());
11       // Check if location is valid.
12       if (FullLocation.isValid())
13         llvm::outs() << "Found declaration at "
14                      << FullLocation.getSpellingLineNumber() << ":"
15                      << FullLocation.getSpellingColumnNumber() << "
                           \n";
16     }
17     return true;
18   }
19
20 private:
21   ASTContext *Context;
22 };
```

[2]Razlika izmedju deklaracija nabrojivog tipa sa sintaksom `enum` i `enum class` objašnjena u okviru opisa pravila **A7-2-3** u sekciji 4.2.

3.3 Dijagnostika u okviru kompilatora *Clang*

Sistem dijagnostike u kompilatoru *Clang* igra važnu ulogu u tome kako kompilator komunicira sa korisnikom. Pod dijagnostikom smatraju se upozorenja i greške koje kompilator prijavljuje za kôd koji je neispravan ili čija je ispravnost sumnjiva.

U kompilatoru *Clang* svaka dijagnostika sadrži jedinstveni identifikator, poruku na Engleskom jeziku koja opisuje problem u kodu, lokaciju u izvornom kodu (objekat klase `SourceLocation`) na koju se dijagnostika odnosi i nivo ozbiljnosti (eng. *severity*) dijagnostike (na primer, upozorenje ili greška).

Objekat klase `SourceLocation` omogućava ispisivanje putanje do izvornog fajla u kome se nalazi kôd, linije koda i kolone u okviru linije. Ove informacije ispisuju se u obliku `putanja_do_fajla:broj_linije:broj_kolone`. Na lokaciju u izvornom fajlu na koju se dijagnostika odnosi, u okviru ispisane poruke, pokazivaće karakter `^`. Opciono, pored lokacije na koju se odnosi, dijagnostika može sadržati i opseg lokacija na koji se odnosi (objekat klase `SourceRange`). Opseg će u poruci biti podvučen karakterima `~` [17].

Na slici 3.2 prikazan je primer ispisa upozorenja na standardni izlaz. Upozorenje sadrži informacije o lokaciji, nivo upozorenja, opseg i poruku upozorenja.

```
/home/ognjen/Desktop/AutoFixTest.cpp:1:6: warning: Enumerations shall be
declared as scoped enum classes.
enum E2 : int {E20, E21};
~~~~~^~~~~~~~~~~~~~~~~~~
1 warning generated.
```

Slika 3.2: Primer ispisa upozorenja na standardni izlaz.

Predlozi za ispravljanje koda

Kompilator *Clang* podržava mehanizam kojim se za neispravan kôd, na koji se odnosi dijagnostika, mogu ispisati predlozi za ispravljanje tog koda (eng. *fixit hint*). Ovaj mehanizam podržan je kroz klasu `FixItHint`.

Instance klase `FixItHint` dodaju se već kreiranoj dijagnostici. Objekti ove klase mogu biti kreirani jednim od sledeća tri konstruktora:

- `FixItHint::CreateInsertion(Loc, Code)` — Kreira objekat koji predlaže da se kôd (predstavljen stringom) `Code` umetne ispred lokacije `Loc`.

- `FixItHint::CreateRemoval(Range)` — Kreira objekat koji predlaže da se izbriše kôd iz opsega `Range`.
- `FixItHint::CreateReplacement(Range, Code)` — Kreira objekat koji predlaže da se izbriše kôd iz opsega `Range` i da se zameni sa kodom `Code`.

Na slici 3.3 prikazan je ispis upozorenja sa predlogom za ispravljanje koda na standardni izlaz. Na slici je prikazana lokacija ispred koje treba umetnuti ključnu reč `class` kako bi kôd bio ispravan.

```
/home/ognjen/Desktop/AutoFixTest.cpp:1:6: warning: Enumerations shall be
declared as scoped enum classes.
enum E2 : int {E20, E21};
     ^
     class
1 warning generated.
```

Slika 3.3: Primer ispisa upozorenja sa predlogom za ispravljanje koda na standardni izlaz.

Klasa `DiagnosticConsumer`

Klasa `DiagnosticConsumer` u okviru kompilatora *Clang* ima ulogu da obradi (konzumira) dijagnostiku prijavljenu za izvorni kôd. Ovo je apstraktna klasa i svaka klasa koje je nasleđuje treba da implementira konkretan vid obrade dijagnostike.

Na primer, jedan vid obrade dijagnostike jeste formatiranje poruka dijagnostike i ispisivanje poruka zajedno sa informacijama o lokaciji i opsegu. Ovakvu obradu vrši klasa `TextDiagnosticPrinter`, potklasa klase `DiagnosticConsumer`. Ispisivanje dijagnostike nije jedini vid obrade dijagnostike, niti je neophodan. Na primer, dijagnostika se može obraditi tako što će za svaku kreiranu dijagnostiku biti provereno da li je očekivana i ukoliko nije, sprovešće se određene akcije. Ovakav vid obrade dijagnostike vrši klasa `VerifyDiagnosticConsumer`, potklasa klase `DiagnosticConsumer` [17].

Klasa `DiagnosticConsumer` definiše nekoliko metoda u svrhu obrade dijagnostike. U ovom radu korišćene su metode:

- `HandleDiagnostic` — poziva se nakon kreiranja svake dijagnostike u okviru kompilatora i služi za obradu te dijagnostike.

- `finish` — poziva se nakon što su kreirane i obrađene sve dijagnostike u okviru kompilatora i služi za dodatnu obradu celokupne dijagnostike.

Klasa `DiagnosticsEngine`

U okviru kompilatora *Clang* dijagnostika se prijavljuje upotrebom klase `DiagnosticsEngine`. Ova klasa služi za kreiranje dijagnostike i prosleđivanje dijagnostike objektu klase `DiagnosticConsumer`. Dijagnostika se kreira pozivanjem metode `Report`. Ovaj metod kao argumente dobija informacije potrebne za kreiranje dijagnostike, kao što su lokacija i poruka i koristi ih da kreira objekat klase `DiagnosticBuilder`. Kreirani objekat predstavlja povratnu vrednost metode `Report`.

`DiagnosticBuilder` predstavlja malu, pomoćnu klasu za prijavljivanje dijagnostike. Ova klasa olakšava prosleđivanje dodatnih informacija prilikom kreiranja dijagnostike, kao što su opsezi ili predlozi izmene koda. Prosleđivanje dodatnih informacija se vrši upotrebom operatora `<<` koji klasa `DiagnosticBuilder` definiše. Operator `<<` kao argument prima dodatne informacije, kao što su objekti klase `SourceRange` ili `FixItHint`. Dijagnostika kreirana metodom `Report` se prosleđuje objektu klase `DiagnosticConsumer` prlikom uništavanja, odnosno u destruktoru kreiranog objekta klase `DiagnosticBuilder`.

Primer prijavljivanja dijagnostike u kompilatoru *Clang*

Na listingu 3.4 prikazana je funkcija `emitWarningWithHintInsertion` kojom se prijavljuje upozorenje zajedno sa predlogom za ispravljanje koda. Upozorenje se prijavljuje pozivom metode `Report` (linija 10). Ovaj metod kao argumente dobija lokaciju za koju treba prijaviti dijagnostiku (`diagLoc`) i jedinstveni identifikator (`ID`). Identifikator je kreiran, pozivom metode `getCustomDiagID(DiagnosticIDs::Warning, msg)`, za dijagnostiku sa nivoom ozbiljnosti upozorenja i porukom `msg` (linija 7). Metod `Report` na osnovu ovih informacija kreira i vraća objekat klase `DiagnosticBuilder` kom se prosleđuje kreirani objekat klase `FixItHint` (linija 10). Upozorenje će biti prosleđeno objektu klase `DiagnosticConsumer` odmah nakon dodavanja predloga za ispravljanje koda s obzirom da objekat klase `DiagnosticBuilder` nije vezan ni za jednu promenljivu i odmah će biti uništen.

Listing 3.4: Primer prijavljivanja dijagnostike u kompilatoru *Clang*.

```
void emitWarningWithHintInsertion(DiagnosticsEngine &DE,
                                  std::string &msg,
                                  std::string &str,
                                  SourceLocation insertLoc,
                                  SourceLocation diagLoc) {
  unsigned ID =
      DE.getDiagnosticIDs()->getCustomDiagID(DiagnosticIDs::Warning
                                             ,msg);
  FixItHint hint = FixItHint::CreateInsertion(insertLoc, str);
  DE.Report(diagLoc, ID) << hint;
}
```

3.4 AST-uparivači

AST-uparivač (ili samo uparivač) je objekat šablonske klase `Matcher`. Uparivači služe za jednostavno pronalaženje čvorova apstraktnog sintaksičkog stabla koji imaju određene željene karakteristike. Pronađeni čvorovi se čuvaju u odgovarajućim strukturama podataka kako bi se kasnije mogli analizirati. Željene karakteristike čvorova koje uparivač treba da pronađe se zadaju prilikom kreiranja uparivača korišćenjem izraza za uparivanje (eng. *match expressions*). Ovo su izrazi jezika specijalne namene implementiranog u okviru biblioteke AST-uparivača (eng. *LibASTMatchers*). Ovaj jezik pruža mogućnost kreiranja predikata nad apstraktnim sintaksnim stablom.

Na primer, za kreiranje uparivača koji izdvaja sve deklaracije nabrojivog tipa (ključna reč `enum`) iz apstraktnog sintaksičkog stabla može se koristiti izraz za uparivanje `enumDecl()`. Ukoliko prilikom uparivanja treba ignorisati deklaracije iz zaglavlja, izraz za uparivanje može se proširiti izrazom `isExpansionInMainFile()`. Izraz

`enumDecl(isExpansionInMainFile())`

kreiraće uparivač koji služi za izdvajanje deklaracija nabrojivog tipa iz glavnog (`.cpp`) fajla [1, 16].

Nakon uparivanja, nad izdvojenim konstruktom može se vršiti dodatna analiza, na primer ispitivanje saglasnosti konstrukta sa pravilom standarda za pravilno pisanje C++ koda. S obzirom da uparivači često predstavljaju kompoziciju više uparivača, zgodno je imati mogućnost adresiranja svakog podrezultata (re-

zultata svakog od uparivača u kompoziciji) zasebno. Na primer, uparivač može izdvajati deklaracije nabrojivog tipa, ali samo one koje definišu i neku konstantu u okviru nabrajanja, odnosno deklaracije koje imaju potomka u stablu tipa `EnumConstantDecl`. Na listingu 3.5 prikazan je primer deklaracije nabrojivog tipa `E1` koja definiše konstante `E10` i `E11`.

Listing 3.5: Primer deklaracije nabrojivog tipa (`E1`) koja definiše konstante u okviru nabrajanja (`E10` i `E11`).

```
1 enum E1 : int {E10, E11};
```

U ovom slučaju zgodno je da se u okviru analize izdvojenog čvora, odnosno deklaracije nabrojivog tipa, može direktno analizirati njegov potomak, konstanta u okviru nabrajanja, bez potrebe da se ovaj potomak ponovo traži u apstraktnom sintaksičkom stablu.

U svrhu adresiranja rezultata pretrage, uparivači se mogu „vezati” (eng. *binding*) za određeni string. Na primer, izraz

```
enumDecl(hasDescendant(enumConstantDecl().bind(„EnumConstNode")))
                                                .bind(„EnumNode")
```

će vezati izdvojene deklaracije nabrojivog tipa za string `„EnumNode"`, dok će konstante vezati za string `EnumConstNode`. Rezultati uparivanja predstavljeni su kao objekti klase `MatchResult`. Iz promenljive `Result` koja predstavlja objekat klase `MatchResult`, čvor koji predstavlja deklaraciju nabrojivog tipa može se dobiti izrazom

```
auto ED = Result.Nodes.getNodeAs<EnumDecl>(„EnumNode")
```

dok se čvor koji predstavlja deklaraciju konstante u okviru nabrajanja može dobiti izrazom

```
auto ECD = Result.Nodes.getNodeAs<EnumConstantDecl>(„EnumConstNode").
```

Dobijeni čvorovi se onda mogu koristiti u svrhu analiziranja koda koji predstavljaju.

Nakon formulisanja izraza za uparivanje kreirani uparivač se pokreće nad apstraktnim sintaksnim stablom. Ovo se postiže pozivanjem metode `matchAST()` klase `MatchFinder`. Za obilazak koji će izvršiti objekat klase `MatchFinder` uparivači se registruju zajedno sa objektima koji implementiraju povratni poziv uparivača (eng. *match callback*). Ovo su objekti klase `MatchCallback` čiji metod `run(const MatchResult &)` se poziva nakon svakog uspešnog uparivanja uparivača sa kojim je ovaj povratni poziv registrovan. Za implementaciju specifičnog povratnog pozi-

va treba implementirati klasu koja nasleđuje klasu `MatchCallback` i predefinisati metod `run`. U okviru metode `run` može se vršiti dodatna analiza izdvojenih čvorova i po potrebi prijavljivati dijagnostika vezana za kôd koji taj rezultat predstavlja.

Primer implementacije AST-uparivača

Na listingu 3.6 prikazan je primer uparivača koji pronalazi sve deklaracije nabrojivog tipa koje ne koriste sintaksu `enum class`. Za ove deklaracije nabrojivog tipa prijavljuje se upozorenje zajedno sa predlogom za ispravljanje koda u okviru funkcije `emitWarningWithHintInsertion` (linija 22). U funkciji `matchASTExample` kreiraju se uparivač (linija 33) i objekat klase povratnog poziva (linija 35). Nakon toga, uparivač se registruje za obilazak (linija 37) i pokreće nad apstraktnim sintaksnim stablom pomoću objekta klase `MatchFinder` (linija 39).

Listing 3.6: Primer uparivača koji pronalazi sve deklaracije nabrojivog tipa koje ne koriste sintaksu `enum class`. Ovaj primer demonstrira i upotrebu klasa `MatchFinder`, `MatchCallback` i `MatchResult`. Funkcija `emitWarningWithHintInsertion` implementirana je na listingu 3.4 i dostupna je kroz zaglavlje `EmitWarning.h`

```
1  #include "EmitWarning.h"
2
3  // Callback class.
4  class A7_2_3 : public MatchFinder::MatchCallback {
5  public:
6    A7_2_3(ASTContext &ASTCtx) : ASTCtx(ASTCtx) {}
7    virtual void run(const MatchFinder::MatchResult &Result);
8
9  private:
10   ASTContext &ASTCtx;
11 };
12
13 void A7_2_3::run(const MatchFinder::MatchResult &Result) {
14   if (auto ED = Result.Nodes.getNodeAs<clang::EnumDecl>("
       A7_2_3_Matcher")) {
15     // Check if declaration contains 'class' tag.
16     if (!ED->isScopedUsingClassTag()) {
17       // Create warning string.
18       std::string msg =
19           "Enumerations shall be declared as scoped enum classes.";
```

```
20        std::string insStr = "class ";
21        // Function for emitting warnings with fixit hints using
              diagnostics engine.
22        emitWarningWithHintInsertion(
23            ASTCtx.getDiagnostics(), msg, insStr,
24            ED->getSourceRange().getBegin().getLocWithOffset(5),
25            ED->getLocation());
26      }
27    }
28  }
29
30  void matchASTExample(ASTContext *Context){
31    MatchFinder Finder;
32    // Create matcher.
33    Matcher<Decl> Matcher = enumDecl(isExpansionInMainFile()).bind("
        A7_2_3_Matcher");
34    // Create callback class.
35    MatchCallback *Callback = new A7_2_3(Context);
36    // Register matcher.
37    Finder.addMatcher(Matcher, Callback);
38    // Run matcher over AST.
39    Finder.matchAST(Context);
40  }
```

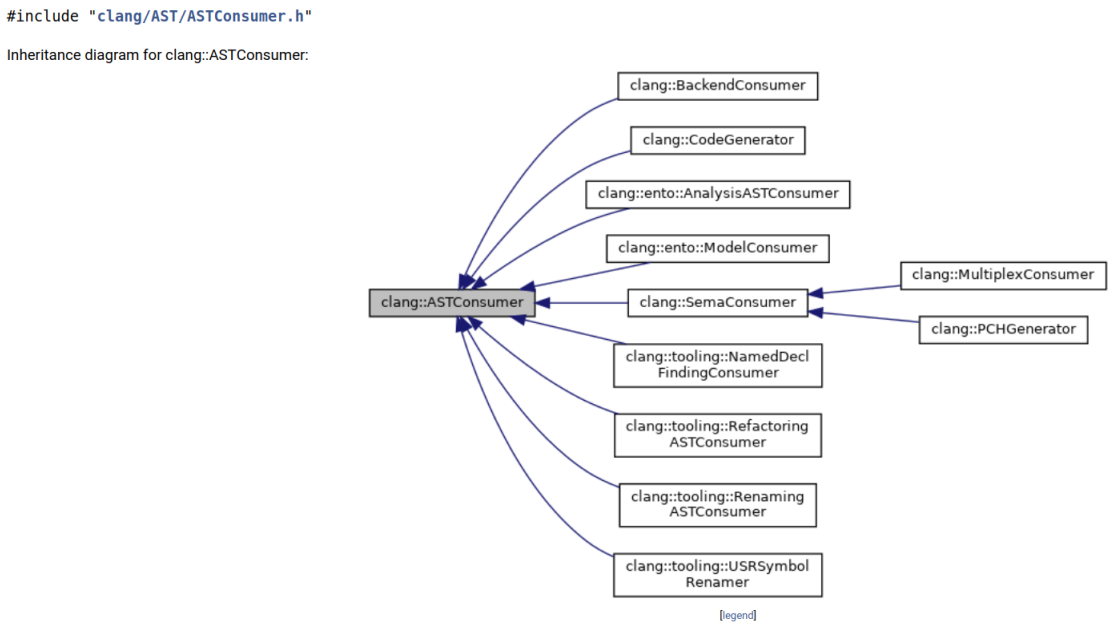
3.5 Interfejsi za akcije nad prednjim delom kompilatora

Akcije nad prednjim delom kompilatora omogućavaju analizu i upotrebu rezultata i informacija koje pruža prednji deo kompilatora. Ove informacije mogu biti korisne za kreiranje alatki za refaktorisanje koda, statičku analizu, prikupljanje statistike i grafičko prezentovanje rezultata kompilatora. Takođe, igraju i ključnu ulogu u samoj kompilaciji koda i deo su glavne protočne obrade (eng. *pipeline*) u kompilatorskoj infrastrukturi LLVM. Ova funkcionalnost je efikasno i sistematično implementirana u okviru klasa `ASTConsumer`, `ASTFrontendAction` i njihovih potklasa.

Klasa ASTConsumer

`ASTConsumer` je apstraktna klasa koja omogućava izvršavanje različitih akcija nad apstraktnim sintaksnim stablom nezavisno od toga kako je apstraktno sintaksičko stablo kreirano. Akcije se mogu izvršavati u različitim fazama tokom kreiranja apstraktnog sintaksičkog stabla [12]. Na primer, metod
`virtual void HandleInlineFunctionDefinition (FunctionDecl *D)`
biće pozvan svaki put kada se završi kreiranje umetnutih (eng. *inline*) funkcija prilikom kreiranja apstraktnog sintaksičkog stabla. `ASTConsumer` definiše niz sličnih virtuelnih metoda koje predefinišu klase koje je nasleđuju.

Na slici 3.4 prikazane su klase u okviru kompilatora *Clang* koje nasleđuju klasu `ASTConsumer`. Klasa `CodeGenerator`, prikazana na slici, generiše LLVM međukod od apstraktnog sintaksičkog stabla i predstavlja jedan od osnovnih delova u kompilatorskoj infrastrukturi LLVM, što demonstrira značaj klase `ASTConsumer`.



Slika 3.4: Klase koje nasleđuju klasu `ASTConsumer` [6].

Klasa `ASTConsumer` korisna je i za kreiranje samostalnih alatki za statičku analizu koji se zasnivaju na analizi apstraktnog sintaksičkog stabla. U svrhu kreiranja alatki može se koristiti kombinacija upotrebe klase `ASTConsumer` sa mehanizmima za obilazak i obradu apstraktnog sintaksičkog stabla kao što su AST-posetioci i

AST-uparivači.

Za implementaciju specifične akcije nad apstraktnim sintaksnim stablom potrebno je implementirati potkasu klase `ASTConsumer` i u okviru nje predefinisati metod
`virtual void HandleTranslationUnit(ASTContext &Ctx)`.
Ovaj metod biće pozvan nakon što je kreirano apstraktno sintaksičko stablo za jedinicu prevođenja, odnosno u trenutku kada je celokupno apstraktno sintaksičko stablo za jedinicu prevođenja dostupno. U okviru njega, nad asptraktnim sintaksnim stablom, može se pozvati AST-uparivač koji će izvršiti obilazak i analizu apstraktnog sintaksičkog stabla [6].

Primer upotrebe klase `ASTConsumer`

Na listingu 3.7 prikazana je implementacija klase `AutoFixConsumer`. U okviru metode `HandleTranslationUnit` poziva se funkcija za kreiranje i pokretanje uparivača `matchASTExample` (linija 9).

Listing 3.7: Implementacija klase `AutoFixConsumer`. Funkcija `matchASTExample` prikazana je na listingu 3.6 i dostupna je kroz zaglavlje `AutoFixMatchers.h`.

```
1  #include "clang/AST/ASTConsumer.h"
2  #include "AutoFixMatchers.h"
3
4  class AutoFixConsumer : public clang::ASTConsumer {
5  public:
6    explicit AutoFixConsumer(ASTContext *Context) : Context(Context)
       {}
7
8    virtual void HandleTranslationUnit(clang::ASTContext &Context) {
9      matchASTExample(Context);
10   }
11
12   ASTContext *Context;
13 };
```

Klasa ASTFrontendAction

`FrontendAction` je apstraktna klasa za akcije koje izvršava prednji deo kompilatora (eng. *frontend*). Klasa `FrontendAction` ima raznolike upotrebe, odnosno

specijalizacije. Primer specijalizacija ove klase su `DumpCompilerOptionsAction` koja omogućava ispisivanje opcija koje se mogu zadati kompilatoru i `PreprocessorFrontendAction` koja omogućava izvršavanje akcija vezanih za pretprocesiranje izvornog koda. Međutim, najčesća upotreba ove klase vezana je za akcije koje se izvršavaju nad apstraktnim sintaksnim stablom. U ovu svrhu koristi se apstraktna klasa `ASTFrontendAction` koja je direktna potklasa klase `FrontendAction`.

`ASTFrontendAction` predefiniše metod `executeAction` klase `FrontendAction`. U okviru ove metode pozivaju se funkcije za semantičku analizu i kreiranje apstraktnog sintaksičkog stabla. Nad ovim apstraktnim sintaksnim stablom izvršiće se akcije implementirane u objektu klase `ASTConsumer` pridruženom ovoj klasi. Na slici 3.5 prikazane su klase u okviru kompilatora *Clang* koje nasleđuju klasu `ASTFrontendAction`. Slika demonstrira raznolikost upotrebe ove klase.

Da bi se implementirala specifična akcija nad apstraktnim sintaksnim stablom, potrebno je implementirati klasu koja nasleđuje klasu `ASTFronendAction` i dodeliti joj objekat klase `ASTConsumer` koji implementira željenu akciju. Objekat se kreira i dodeljuje predefinisanjem metode

```
unique_ptr<ASTConsumer> CreateASTConsumer(CompilerInstance Compiler,
                                          StringRef InFile)
```

Ova metoda kao argumente dobiija instancu kompilatora *Clang* i ime fajla za koji se kreira apstraktno sintaksičko stablo. Povratna vrednost metode je pokazivač na kreirani objekat klase `ASTConsumer`.

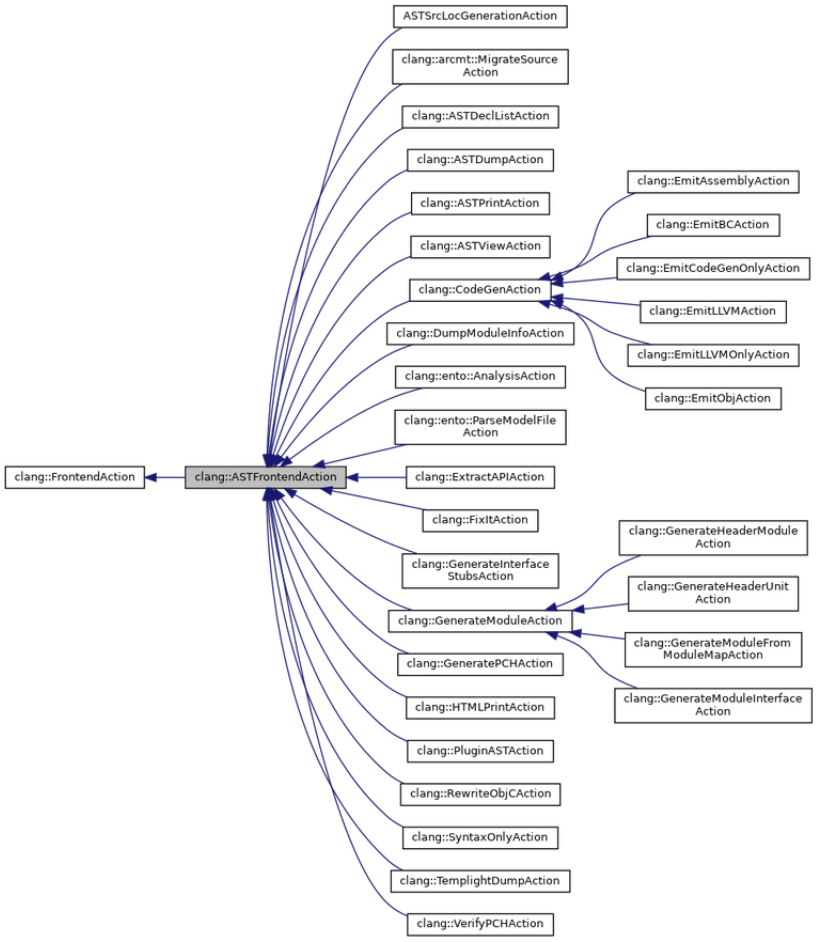
Primer upotrebe klase `ASTFrontendAction`

Na listingu 3.8 prikazana je implementacija klase `AutoFixAction` koja izvršava akcije nad apstraktnim sintaksnim stablom. Klasa `AutoFixAction` u okviru metode `CreateASTConsumer` (linija 6) kreira pokazivač na objekat klase `AutoFixConsumer`.

Listing 3.8: Implementacija klase `AutoFixAction`. Klasa `AutoFixConstumer` prikazana je na listingu 3.7 i dostupna je kroz zaglavlje `AutoFixConsumer.h`.

```
1 #include "AutoFixConsumer.h"
2
3 class AutoFixAction : public clang::ASTFrontendAction {
4 public:
5   virtual std::unique_ptr<clang::ASTConsumer>
6   CreateASTConsumer(clang::CompilerInstance &Compiler,
```

```
7                        llvm::StringRef InFile) {
8      return std::make_unique<AutoFixConsumer>(
9                    &Compiler.getASTContext(),
10                   Compiler.getSourceManager());
11   }
12 };
```



Slika 3.5: Klase koje nasleđuju klasu `ASTFrontendAction` [7].

3.6 Interfejsi za kreiranje alatki

Kompilatorska infrastruktura LLVM pruža podršku za jednostavno kreiranje kvalitetnih alatki za statičku analizu izvornog koda. Ove alatke zasnivaju se na

upotrebi interfejsa ka apstraktnom sintaksičkom stablu kompilatora *Clang* ili korišćenjem statičkog analizatora kompilatora *Clang* (eng. *Clang Static Analyzer*) za potrebe simboličkog izvršavanja programa.

Alatke za statičku analizu mogu koristiti kombinaciju tehnika obrade apstraktnog sintaksičkog stabla i simboličkog izvršavanja programa u zavisnosti od kompleksnosti analize koja je potrebna. Implementacija statičke analize obradom apstraktnog sintaksičkog stabla je jeftinija po pitanju računarskih resursa ali je ograničena informacijama dostupnim tokom kompilacije programa.

Kompilator *Clang* pruža više infrastruktura za pisanje različitih vrsta softverskih alatki koji koriste sintaksne i semantičke informacije o programu. U nastavku će biti opisano nekoliko interfejsa koji se mogu koristiti u ovu svrhu zajedno sa njihovim prednostima i manama.

`LibClang` je stabilni C interfejs visokog nivoa (eng. *high level*) ka kompilatoru *Clang*. Ovaj interfejs pruža parsiranje izvornog koda i izagradnju apstraktnog sintaksičkog stabla, učitavanje i obilazak već kreiranog apstraktnog sintaksičkog stabla i dohvatanje određenih informacija o izgrađenom apstraktnom sintaksičkom stablu kao što su lokacije iz izvornog koda elemenata iz stabla. Ovaj interfejs ne pruža sve informacije i detalje iz izgrađenog apstraktnog sintaksičkog stabla [14]. Ovo ga čini nepogodnim za implementaciju alatki za statičku analizu ali omogućava stabilnost pri promeni verzija kompilatora *Clang*. Treba ga koristiti u slučajevima kada:

- je potreban interfejs ka kompilatoru *Clang* iz jezika koji nije C++.
- je potreban stabilni inferfejs koji je kompatibilan sa starijim verzijama kompilatora *Clang*.
- su potrebne apstrakcije visokog nivoa kao što je iteriranje kroz apstraktno sintaksičko stablo sa kursorima [3] ili drugi detalji vezani za AST.

`LibClang` ne treba koristiti kada je potrebna puna kontrola nad apstraktnim sintaksnim stablom [3].

Dodaci kompilatora *Clang* omogućavaju izvršavanje dodatnih akcija nad apstraktnim sintaksnim stablom tokom kompilacije programa. Ovo su dinamič-

[3] Kursori su instance strukture `CXCursor` i služe da ujedine različite entitete u programu, na primer deklaracije i izraze, u apstakcije sa istim skupom operacija [9].

ke biblioteke koje kompilator učitava tokom izvršavanja i lako ih je integrisati u okruženje za prevođenje programa (eng. *build enviroment*) [4].

Dodatke kompilatora *Clang* treba koristiti kada:

- je potrebno ponovno izvršavanje alata uvek kada se zavisnosti potrebne za prevođenje programa izmene.
- je potrebno da alat omogući ili neomogući prevođenje programa.
- je potrebna potpuna kontrola nad apstraktnim sintaksnim stablom.

Dodatke kompilatora *Clang* ne treba koristiti kada:

- je potrebno kreirati alat koji se ne koristi u okviru sistema za prevođenje programa.
- su alatu potrebne informacije o tome kako je *Clang* podešen uključujući mapiranje virtuelnih fajlova u memoriji.
- je potrebno koristiti alat nad podskupom fajlova u projektu koji nisu povezani sa izmenama koje bi zahtevale ponovno prevođenje programa [3].

`LibTooling` je C++ interfejs koji služi za pisanje samostalnih alatki. Ova biblioteka omogućava jednostavnu upotrebu opisanih akcija prednjeg dela kompilatora (eng. *frontend actions*), ali i jednostavno dodavanje opcija komandne linije i pokretanje nad fajlovima nezavisnim od sistema za prevođenje. Uopšteno, `LibTooling` treba koristiti kada:

- je potrebno pokretati alat nad jednim fajlom ili specifičnim podskupom fajlova nezavisnim od sistema za prevođenje.
- je potrebno imati punu kontrolu nad apstraktnim sintaksnim stablom kompilatora *Clang*.
- je potrebno deliti kôd sa dodacima kompilatora *Clang*.

`LibTooling` nije najbolji izbor u slučajevima kada:

- je potrebno pokretati alat nakon promena u zavisnostima u sistemu za prevođenje.
- je potreban stabilan interfejs tako da se kôd alata ne mora menjati kada se interfejs apstraktnog sintaksičkog stabla promeni.

- su potrebne apstrakcije visokog nivoa kao što su kursori.
- alat neće biti napisan u jeziku C++ [3].

Da bi se implementirao kvalitetan alat za statičku analizu neophodna je puna kontrola nad apstraktnim sintaksnim stablom kako bi se omogućila što preciznija analiza izvornog koda. Fleksibilne alatke za statičku analizu se ne moraju nužno pokretati u okviru sistema za prevođenje i podržavaju mogućnost analize fajlova nezavnisnih od sistema za prevođenje. Takođe, mogućnost dodavanja opcija komandne linije olakšava korisniku upotrebu alata i omogućava korisniku veću kontrolu nad radom alata. Na osnovu ovoga je zaključeno da je biblioteka `LibTooling` najbolji izbor za izradu kvalitetnog alata za statičku analizu u okviru kompilatorske infrastrukture LLVM.

Primer kreiranja alata upotrebom biblioteke LibTooling

Na listingu 3.9 prikazana je implementacija jednostavnog alata korišćenjem biblioteke `LibTooling`. Ovaj alat pokreće definisanu akciju `AutoFixAction` nad izvornim kodom koji je prosleđen kao argument komandne linije. U ovu svrhu koristi se funkcija `runToolOnCode` (linija 6) biblioteke `LibTooling`. Alat pronalazi sve deklaracije nabrojivog tipa koje ne koriste sintaksu `enum class` i za ove deklaracije se prijavljuje upozorenje zajedno sa predlogom za ispravljanje koda.

Listing 3.9: Primer implementacije jednostavnog alata upotrebom biblioteke `LibTooling`. Alat koristi klasu `AutoFixAction` sa listinga 3.8 dostupnom kroz zaglavlje `ASTFrontendAction.h`.

```
1  #include "clang/Tooling/Tooling.h"
2  #include "ASTFrontendAction.h"
3
4  int main(int argc, char **argv) {
5    if (argc > 1) {
6      clang::tooling::runToolOnCode(
7                      std::make_unique<AutoFixAction>(),
8                      argv[1]);
9    }
10 }
```

Biblioteka `LibTooling` i kompilacione baze podataka

Samostalne alatke koje su razvijene bibliotekom `LibTooling` zahtevaju kompilacionu bazu podataka (eng. *compilation database*) kako bi zaključili koje opcije treba koristiti prilikom prevođenja fajla nad kojim se pokreće alat. Informacije o opcijama koje se koriste prilikom proveđenja fajla mogu biti neophodne za pokretanje alata nad tim fajlom. Na primer, ukoliko fajl nad kojim je pokrenut alat uključuje zaglavlja koja nisu sistemska zaglavlja, neophodno je navesti putanju do tih zaglavlja inače kompilator *Clang* neće moći da izgradi apstraktno sintaksičko stablo, a samim tim će se prekinuti i izvršavanje alata.

Kompilaciona baza podataka kreira se na osnovu fajla `compile_commands.json` koji se generiše alatom `CMake` [11]. Putanja do fajla `compile_commands.json` može se proslediti alatu opcijom komandne linije `-p=<string>`. U suprotnom, alat će sam pokušati da nađe fajl `compile_commands.json` u okviru repozitorijuma.

Ukoliko korisnik nije u mogućnosti da kreira kompilacionu bazu podataka za fajl nad kojim želi da pokrene alat, nakon komande za pokretanje alata može navesti dvostruku crtu `--` u kom slučaju alat neće pokušati da nađe kompilacionu bazu podataka. U ovom slučaju podrazumeva se da su opcije neophodne za prevođenje fajla navedene prilikom pokretanja alata.

3.7 Alatke za testiranje

Kompilatorska infrastruktura LLVM sadrži alatke koji se mogu koristiti u svrhu pisanja i pokretanja testova. U svrhu testiranja alata *AutoFix* korišćene su alatke *lit* i *FileCheck*. Ovi alatke imaju raznoliku upotrebu i širok spektar opcija. U nastavku će biti opisana samo svojstva ovih alatki relevantna za testiranje alata *AutoFix*.

Alat *lit* služi za izvršavanje testova i testnih paketa (eng. *test suites*) u okviru kompilatorske infrastrukture LLVM. Alat takođe sumira rezultate i generiše informacije o greškama u okviru testova. Testovi se pokreću komandom
`llvm-lit PUTANJA`
gde `PUTANJA` može biti do direktorijuma sa testovima, u kom slučaju će se pokrenuti svi testovi u okviru direktorijuma, ili do testa, u kom slučaju će se izvršiti pokretanje pojedinačnog testa. Svaki test koji se pokreće upotrebom alata *lit* mora sadržati `RUN` liniju. `RUN` linije su linije formata `RUN: KOMANDA`.

Ove linije treba koristiti u okviru komentara u testu. Na primer, za C++ testove, `RUN` linija može izgledati ovako: `// RUN: KOMANDA`.
`KOMANDA` će biti izvršena alatom *lit*. Na primer, ukoliko je navedena `RUN` linija
`// RUN: echo „Hello World!"`
alat *lit* će pokrenuti program `echo` sa argumentom `„Hello World!"`

Ukoliko u okviru testa ne postoji `RUN` linija, *lit* će prijaviti grešku prilikom pokretanja testa [15].

Alat *FileCheck* služi za poređenje sadržaja fajlova. Kao ulaz dobija dva fajla, jedan sa standardnog ulaza i jedan naveden kao argument komandne linije i zatim koristi jedan da proveri ispravnost sadržaja drugog. Ovaj alat je koristan za kreiranje testova u okviru kojih je potrebno proveriti da li izlaz nekog alata sadrži očekivane informacije [10]. Ukoliko je u fajlu prosleđenom putem komandne linije navedena direktiva `CHECK: TEKST`, alat *FileCheck* će proveriti da li se `TEKST` nalazi u fajlu koji mu je prosleđen putem standardnog ulaza i prijaviti grešku u slučaju da `TEKST` nije nađen. Direktiva `CHECK-NEXT: TEKST` proverava da li je `TEKST` pronađen na prvoj liniji nakon teksta koji je uparen poslednjom direktivom `CHECK`. U zavisnosti od toga da li je postavljena pre prve, između dve ili nakon poslednje direktive, direktivom `CHECK-NOT: TEKST` utvrđuje se da `TEKST` nije nađen pre prvog uparivanja, između dva uparivanja ili nakon poslednjeg uparivanja direktiva `CHECK` ili `CHECK-NEXT`. U okviru teksta mogu se koristiti i regularni izrazi. Regularnim izrazom se smatra sve što se nalazi u okviru dvostrukih vitičastih zagrada `{{}}`.

Glava 4

Alat AutoFix

Alat *AutoFix* predstavlja alat za statičku analizu izvornog koda napisanog u jeziku C++14. Alat prijavljuje upozorenja za kôd koji nije napisan u skladu sa odabranim podskupom pravila iz standarda kodiranja AUTOSAR C++14 koja se odnose na deklaracije. Zajedno sa upozorenjima alat ispisuje i predlog koda kojim se početni kôd može zameniti kako bi bio u skladu sa standardom. Alat je implementiran u programskom jeziku C++ korišćenjem biblioteka za razvoj alatki dostupnim u okviru kompilatorske infrastrukture LLVM. Osnovna svrha alata je demonstracija kreiranja alatki u okviru kompilatorske infrastrukture LLVM i predstavljanje tehnika obilaska i analize apstraktnog sintaksičkog stabla kompilatora *Clang*. Alat je dostupan i nalazi se na linku `https://github.com/ognjen-plavsic/master/tree/main/code`. Na pomenutom linku se nalazi kôd alata, skup testova i uputstvo za instalaciju alata.

4.1 Korišćenje alata

Alat *AutoFix* se pokreće komandom:

```
auto-fix [options] <source0> [... <sourceN>]
```

Argument `options` označava opcije koje se mogu proslediti alatu *AutoFix*, dok `<source0> [... <sourceN>]` predstavlja listu fajlova, razdvojenih razmakom, nad kojima će se pokrenuti alat. Moguće opcije su:

`--apply-fix`: Ovom opcijom se predložene izmene mogu primeniti na kôd, me-

njajući izvorni fajl nad kojim je pokrenuta analiza. Predložene izmene biće primenjene na kôd ukoliko među njima ne postoji konflikt, odnosno ukoliko se različiti predlozi ne odnose na isti deo koda.

`--exclude-headers`: Ova opcija omogućava da se upozorenja ne prijavljuju za kôd iz zaglavlja. Sistemska zagavlja alat *AutoFix* ignoriše i bez navođenja ove opcije.

`--list-rules`: Ovom opcijom se ispisuju sva podržana pravila u okviru alata u formatu `oznaka - tekst_pravila` gde je `oznaka` jedinstvena oznaka pravila iz AUTOSAR dokumenta, a `tekst_pravila` prestavlja kratak opis pravila iz AUTOSAR dokumenta koji se ujedno ispisuje prilikom prijavljivanja upozorenja vezanih za to pravilo. Na slici 4.1 prikazan je rezultat rada alata *AutoFix* prilikom pokretanja sa opcijom `--list-rules`.

```
ognjen@ognjen-Precision-7710:~$ auto-fix --list-rules
A7-1-6 - The typedef specifier shall not be used.
A7-2-3 - Enumerations shall be declared as scoped enum classes.
A8-5-2 - Braced-initialization {}, without equals sign, shall be used for variable
initialization.
A8-5-3 - A variable of type auto shall not be initialized using {} or ={} braced in
itialization.
```

Slika 4.1: Ispis alata *AutoFix* prilikom korišćenja opcije `--list-rules`.

`--rules=<string>`: Ova opcija omogućava navođenje podskupa implementiranih pravila za koje će alat izvršiti analizu. Pravila u okviru ove opcije se navode po svojoj oznaci iz AUTOSAR dokumenta i treba ih razdvojiti zarezom. Ukoliko se umesto opcije pravila prosledi string „`all`" alat će pokrenuti analizu sa svim implementiranim pravilima u okviru alata. Ukoliko se ova opcija ne navede prilikom pokretanja, *AutoFix* će ovo protumačiti kao da je navedena opcija `--rules=„"`, odnosno neće se izvršiti analiza ni za jedno pravilo. Primer korišćenja ove opcije:

```
auto-fix ./AutoFixTest.cpp --rules=„A7_2_3, A7_1_6"
```

`--help`: Ovom opcijom se ispisuje uputstvo za upotrebu alata.

Pored opcija koje su definisane u okviru alata *AutoFix*, prilikom pokretanja alata mogu se dodati i opcije koje se prosleđuju kompilatoru *Clang*. Na primer,

može da bude korisno da se navede opcija `-Wno-everything` kako bi se prijavljivala isključivo upozorenja generisana alatom *AutoFix* i ignorisala sva upozorenja koja generiše kompilator *Clang* tokom kompilacije. Ovo se može postići komandom:

```
auto-fix --rules=„all" test.cpp --extra-arg=„-Wno-everything" --
```

4.2 Opis implementiranih pravila

Pored formalne klasifikacije opisane u sekciji 2.2, pravila u okviru dokumenta koji opisuje standard kodiranja AUTOSAR C++14 [18] strukturirana su po poglavljima. Struktura poglavlja ovog dokumenta slična je strukturi iz samog C++ standarda ISO/IEC 14882:2014. Svako poglavlje odgovara jednoj komponenti (svojstvu) jezika C++14, to jest sadrži pravila koja se odnose na tu komponentu.

Pravila razmatrana u ovom radu predstavljaju podskup pravila koja se odnose na deklaracije. Deklaracije predstavljaju jedan od osnovnih i najvažnijih koncepta u programskom jeziku C++ i programiranju generalno.

Sva implementirana pravila u okviru alata *AutoFix* spadaju, prema klasifikaciji iz sekcije 2.2, u sledeće kategorije:

1. Obavezna, prema klasifikaciji po obavezi.
2. Automatizovana, prema klasifikaciji po primenjivosti statičke analize.
3. Implementaciona, prema klasifikaciji po cilju primene.

Pravila razmatrana u okviru ovog rada birana su tako da se kôd koji nije u saglasnosti sa pravilom može detektovati analizom apstraktnog sintaksičkog stabla kompilatora *Clang* i da se za taj kôd mogu kreirati razumne alternative koje su u skladu sa standardom AUTOSAR C++14. Implementirana su pravila **A8-5-3**, **A8-5-2**, **A7-1-6**, **A7-2-3**.

Pravilo A8-5-3

> Promenljiva tipa `auto` ne sme biti inicijalizovana korišćenjem vitičastih zagrada tipa {} ili ={}.

Po standardu C++14, kompilator će promenljivu deklarisanu specifikatorom `auto` koja je inicijalizovana sintaksom vitičastih zagrada (`{}` ili `={}`) tretirati kao objekat klase `std::initializer_list<type>`. Ukoliko programer nije svestan ove činjenice, može pomisliti da će zaključeni tip zapravo biti `type`. Da bi se izbegla konfuzija oko zaključivanja tipova, AUTOSAR standard nalaže da se ne koristi nijedna od navedene dve vrste inicijalizacije. Na listingu 4.1 prikazan je kôd nad kojim će biti ilustrovana podrška pravilu **A8-5-3** u okviru alata *AutoFix*. Zaključen tip za promenljive `x2` (linija 7) i `x4` (linija 11) biće `std::initializer_list<int>`, dok će za promenljive `x1` (linija 5) i `x3` (linija 9) biti zaključen tip `int`. S obzirom da deklaracije promenljivih `x2` i `x4` koriste sintaksu vitičastih zagrada, ove deklaracije nisu napisane u skladu sa pravilom **A8-5-3**. Na slici 4.2 prikazan je ispis alata *AutoFix* za kôd sa listinga 4.1. Alat u oba slučaja predlaže da promenljiva bude deklarisana koristeći simbol `=`.

Listing 4.1: Kôd nad kojim je demonstrirana podrška pravilu **A8-5-3** u okviru alata *AutoFix*. Ispis alata *AutoFix* nakon pokretanja nad ovim kodom prikazan je na slici 4.2.

```
1  #include <initializer_list>
2
3  void fn() {
4    // Compliant with rule A8-5-3.
5    auto x1(10);
6    // Not compliant with rule A8-5-3.
7    auto x2{10};
8    // Compliant with rule A8-5-3.
9    auto x3 = 10;
10   // Not compliant with rule A8-5-3.
11   auto x4 = {10};
12 }
```

Pravilo A8-5-2

> Inicijalizacija vitičastim zagradama bez simbola jednako (=) treba biti korišćena za inicijalizaciju promenljive.

Po standardu C++14, prilikom upotrebe inicijalizacije vitičastih zagrada bez znaka = neće doći do konverzija tipova iz tipa veće bitske širine u tip manje bitske

```
ognjen@ognjen-Precision-7710:~$ auto-fix --rules="A8_5_3" ~/AutoFixTest.cpp --
/home/ognjen/AutoFixTest.cpp:7:10: warning: A variable of type auto shall not be in
itialized using {} or ={} braced initialization
  auto x2{10};
         ^~~~
          = 10
/home/ognjen/AutoFixTest.cpp:11:13: warning: A variable of type auto shall not be i
nitialized using {} or ={} braced initialization
  auto x4 = {10};
            ^~~~
            10
2 warnings generated.
```

Slika 4.2: Ispis alata za pravilo **A8-5-3** za kôd sa listinga 4.1.

širine (eng. *narrowing conversions*) što se moze dogoditi prilikom upotrebe ostalih vrsta inicijalizacija. Upotreba simbola = pri inicijalizaciji može izazvati konfuziju kod programera i navesti ga na pomisao da se nad objektom poziva operator dodele iako se zapravo poziva konstruktor. Na listingu 4.2 prikazan je kôd nad kojim je demonstrirana podrška pravilu **A8-5-2** u okviru alata *AutoFix*. Deklaracije promenljivih x1 (linija 6), x2 (linija 8) i x3 (linija 10) nisu u skadu sa pravilom **A8-5-2** s obzirom da ne koriste inicijalizaciju vitičastih zagrada bez simbola =. Ispis alata *AutoFix* kada se pokrene nad fajlom sa kodom iz listinga 4.2 prikazan je na slici 4.3.

Listing 4.2: Kôd nad kojim je demonstrirana podrška pravilu **A8-5-2** u okviru alata *AutoFix*.

```
1  #include <cstdint>
2  #include <initializer_list>
3
4  void f1() {
5    // Not compliant with rule A8-5-2.
6    std::int32_t x1 = 8;
7    // Not compliant with rule A8-5-2.
8    std::int8_t x2(x1);
9    // Not compliant with rule A8-5-2.
10   std::int8_t x3 = {50};
11   // Compliant with rule A8-5-2.
12   std::int8_t x4{50};
13 }
```

```
ognjen@ognjen-Precision-7710:~$ auto-fix --rules="A8_5_2" ~/AutoFixTest.cpp --
/home/ognjen/AutoFixTest.cpp:6:18: warning: Braced-initialization {}, without equal
s sign, shall be used for variable initialization
  std::int32_t x1 = 8;
                ^~~~
                {8}
/home/ognjen/AutoFixTest.cpp:8:17: warning: Braced-initialization {}, without equal
s sign, shall be used for variable initialization
  std::int8_t x2(x1);
               ^~~~
               {x1}
/home/ognjen/AutoFixTest.cpp:10:17: warning: Braced-initialization {}, without equa
ls sign, shall be used for variable initialization
  std::int8_t x3 = {50};
               ^~~~~~~
               {50}
3 warnings generated.
```

Slika 4.3: Ispis alata za pravilo **A8-5-2** za kôd sa listinga 4.2.

Pravilo A7-1-6

> Ne treba koristiti specifikator `typedef`.

Specifikator `typedef` nije pogodan za kreiranje pseudonima (eng. *alias*) za šablonske tipove i čini kôd manje čitljivim. Oba nedostatka mogu se zaobići korišćenjem specifikatora `using`. Ispis alata *AutoFix* kada se pokrene nad fajlom sa kodom iz listinga 4.3 prikazan je na slici 4.4. Alat *AutoFix* od izraza za kreiranje pseudonima za tip korišćenjem specifikatora `typedef` kreira i ispisuje analogni izraz koji koristi sintaksu sa specifikatorom `using`.

Listing 4.3: Primer koda koji nije napisan u skladu sa pravilom **A7-1-6**, odnosno koristi specifikator `typedef`.

```
1 #include <cstdint>
2
3 // Not compliant with rule A7-1-6.
4 typedef unsigned long ulong;
5 // Not compliant with rule A7-1-6.
6 typedef std::int32_t (*fPointer1)(std::int32_t);
7 // Not compliant with rule A7-1-6.
8 typedef int int_t, *intp_t;
```

```
ognjen@ognjen-Precision-7710:~$ auto-fix --rules="A7_1_6" ~/AutoFixTest.cpp --
/home/ognjen/AutoFixTest.cpp:4:23: warning: The typedef specifier shall not be used
.
typedef unsigned long ulong;
~~~~~~~~~~~~~~~~~~~~~~^~~~~
using ulong = unsigned long
/home/ognjen/AutoFixTest.cpp:6:24: warning: The typedef specifier shall not be used
.
typedef std::int32_t (*fPointer1)(std::int32_t);
~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~
using fPointer1 = std::int32_t (*)(std::int32_t)
/home/ognjen/AutoFixTest.cpp:8:13: warning: The typedef specifier shall not be used
.
typedef int int_t, *intp_t;
~~~~~~~~~~~~^~~~~
using int_t = int
/home/ognjen/AutoFixTest.cpp:8:21: warning: The typedef specifier shall not be used
.
typedef int int_t, *intp_t;
~~~~~~~~~~~~~~~~~~~~^~~~~~
using intp_t = int *
4 warnings generated.
```

Slika 4.4: Ispis alata za pravilo A7-1-6 za kôd sa listinga 4.3.

Pravilo A7-2-3

> Nabrojive tipove (eng. *enumerators*) treba deklarisati kao nabrojive tipove sa opsegom odnosno treba koristiti sintaksu `enum class`.

Ukoliko se prilikom deklaracije nabrojivog tipva ne koristi sintaksa `enum class` može doći do ponovnog deklarisanja konstanti iz globalnog opsega. Na listingu 4.4 deklarisan je nabrojiv tip `E1` (linija 3) koje definiše promenljive `E10`, `E11` i `E12` u globalnom opsegu. Ukoliko programer nije svestan činjenice da su promenljive `E10`, `E11` i `E12` definisane u globalnom opsegu može pokušati da deklariše globalnu promenljivu sa identifikatorom koji je korišćen u okviru nabrojivog tipa (linija 6). Ovo će rezultovati greškom prilikom kompilacije zbog dvostruke deklaracije promenljive sa istim identifikatorom.

Listing 4.4: Primer koda u okviru kog dolazi do dvostruke deklaracije promenljive sa istim identifikatorom.

```
1 #include <cstdint>
2
3 enum E1 : std::int32_t { E10, E11, E12 };
4
```

```
5 // Compilation error. Redeclaration of E10.
6 static std::int32_t E10;
```

Korišćenjem nabrojivog tipa sa opsegom, odnosno upotrebom sintakse `enum class`, identifikatori korišćeni prilikom nabrajanja biće deklarisani u svom unutrašnjem opsegu i time sprečiti mogućnost dvostruke deklaracije identifikatora u globalnom opsegu. Primer deklaracije nabrojivog tipa koja koristi sintaksu `enum class` prikazana je na listingu 4.5. Identifikator `E10` (linija 3) u okviru deklaracije nabrojivog tipa `E1` je definisan u unutrašnjem opsegu. Iz ovog razloga neće doći do ponovne deklaracije identifikatora `E10` prilikom deklaracije ovog identifikatora u globolnom opsegu (linija 7) i kôd će se uspešno prevesti.

Listing 4.5: Primer upotrebe sintakse `enum class`.

```
1 #include <cstdint>
2
3 enum class E1 : std::int32_t { E10, E11, E12 };
4
5 // No compilation error.
6 // Identifier E10 inside enum E1 is defined in inner scope.
7 static std::int32_t E10;
```

Ispis alata *AutoFix* kada se pokrene nad fajlom sa kodom iz listinga 4.6 prikazan je na slici 4.5. Alat pokazuje na koju lokaciju treba umetnuti specifikator `class`.

Listing 4.6: Primer koda koji nije napisan u skladu sa pravilom **A7-2-3**, odnosno ne koristi sintaksu `enum class`.

```
1 #include <cstdint>
2
3 // Not compliant with rule A7-2-3.
4 enum E1 : std::int32_t { E10, E11, E12 };
```

4.3 Opis implementacije alata

Alat *AutoFix* implementiran je u okviru projekta `clang-tools-extra`, potprojekta kompilatorske infrastrukture LLVM. Projekat `clang-tools-extra` sadrži

```
ognjen@ognjen-Precision-7710:~$ auto-fix --rules="A7_2_3" ~/AutoFixTest.cpp --
/home/ognjen/AutoFixTest.cpp:4:6: warning: Enumerations shall be declared as scoped
 enum classes.
enum E1 : std::int32_t { E10, E11, E12 };
     ^
     class
1 warning generated.
```

Slika 4.5: Ispis alata za pravilo **A7-2-3** za kôd sa listinga 4.6.

alatke implementirane interfejsima za alatke kompilatora *Clang* (eng. *Clang's tooling APIs*). Alat *AutoFix* je podeljen na četiri jedinice prevođenja: `AutoFix.cpp`, `AutoFixMatchers.cpp`, `AutoFixDiagnosticConsumer.cpp` i `AutoFixHelper.cpp`.

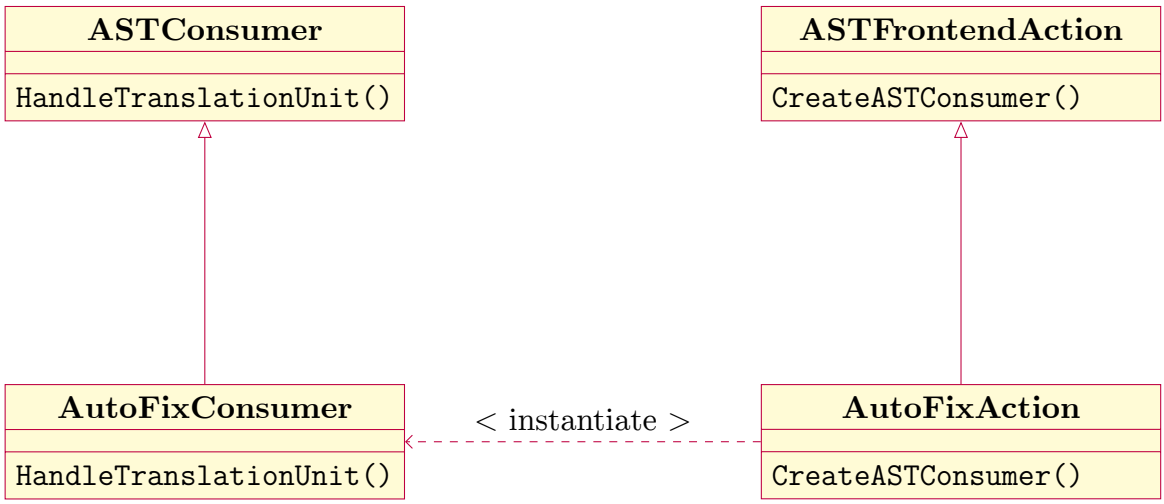
AutoFixMatchers.cpp

Alat *AutoFix* koristi biblioteku `LibAstMatchers` za analizu i obradu apstraktnog sintaksičkog stabla kompilatora *Clang*. *AutoFix* definiše uparivače za uparivanje osnovnih konstrukta na koje se pravilo odnosi i za sužavanje pretrage apstraktnog sintaksičkog stabla. Na primer, ukoliko je pravilo vezano za nabrojive tipove, uparivač koji odgovara ovom pravilu će upariti sve deklaracije nabrojivih tipova iz apstraktnog sintaksičkog stabla ali će i suziti pretragu tako što će uparivati samo deklaracije koje nisu implicitne. Ovo se može postići izrazom za uparivanje `enumDecl(unless(isImplicit()))`. Svakom pravilu koje alat *AutoFix* podržava odgovara jedan uparivač. Uparivači su nazvani po broju pravila na koje se odnose i imaju imena `A7_1_6_Matcher`, `A7_2_3_Matcher`, `A8_5_2_Matcher` i `A8_5_3_Matcher`.

Za svaki od uparivača implementirana je i klasa povratnog poziva u okviru koje se vrši analiza uparenih konstrukta, konstrukcija predloga za ispravljanje koda i prijavljivanje upozorenja ukoliko je analizom utvrđeno da kôd nije napisan u skladu sa pravilom na koje se uparivač odnosi. Klase povratnih poziva su takođe nazvane po pravilima na koje se odnose i nose imena `A7_1_6`, `A7_2_3`, `A8_5_2` i `A8_5_3`.

AutoFix.cpp

Ova jedica prevođenja predstavlja ulaznu tačku alata *AutoFix*. U okviru nje, implementirane su klase `AutoFixConsumer` i `AutoFixAction` koje nasleđuju redom klase `ASTConsumer` i `ASTFrontendAction` (opisane u sekciji 3.5). Osnovna uloga



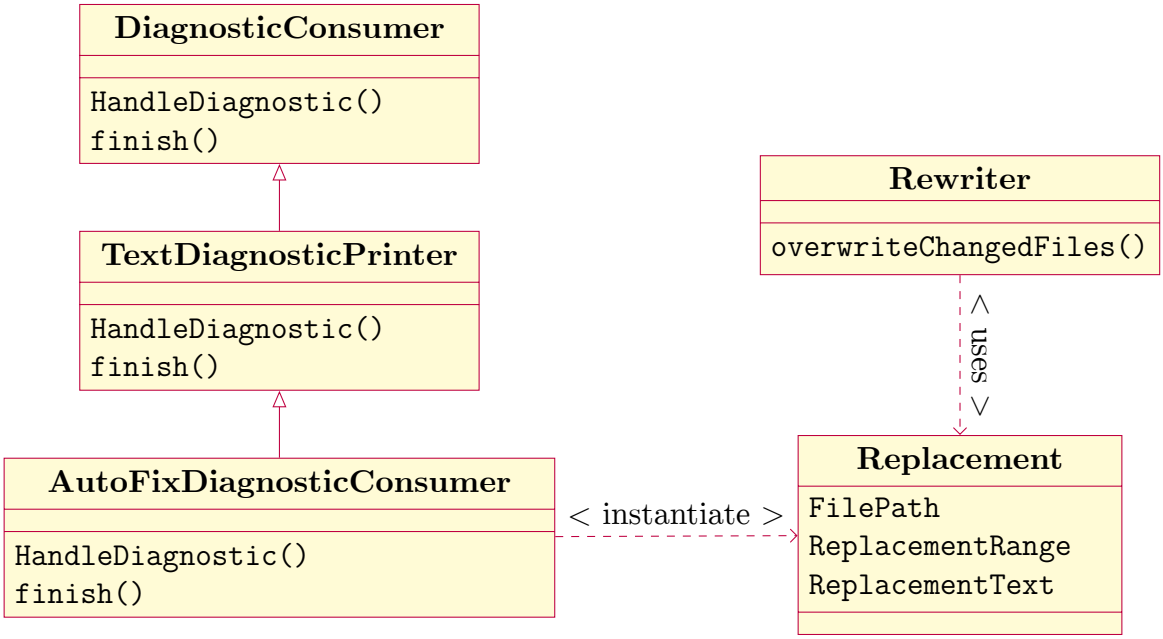
Slika 4.6: Odnos klasa `ASTConsumer`, `AutoFixConsumer`, `ASTFrontendAction` i `AutoFixAction`.

klase `AutoFixConsumer` jeste da obezbedi da se nad jedinicom prevođenja pokrenu odgovarajući uparivači. To su uparivači koji odgovaraju pravilima koje je korisnik zadao u okviru opcije komandne linije `--rules`. Parsiranje ove opcije i pokretanje uparivača nad apstraktnim sintaksnim stablom implementirano je u okviru metode `HandleTranslationUnit` klase `AutoFixConsumer`. Ova metoda se poziva tokom parsiranja na kraju izgradnje apstraktnog sintaksičkog stabla za svaku jedinicu prevođenja nad kojom je pokrenut alat. Klasa `AutoFixAction` instancira objekat klase `AutoFixConsumer` u okviru metode `CreateASTConsumer`. Na slici 4.6 prikazan je odnos klasa `AutoFixConsumer`, `AutoFixAction`, `ASTConsumer` i `ASTFrontendAction`.

U okviru jedinice prevođenja `Autofix.cpp` takođe je implementirana funkcija `main` alata *AutoFix*. U okviru nje vrši se parsiranje opcija komandne linije, kreira se instanca alata, kreiranoj instanci se pridružuje objekat klase `AutoFixDiagnosticConsumer` i alat se pokreće nad zadatom jedinicom prevođenja.

AutoFixDiagnosticConsumer.cpp

Vid obrade dijagnostike najrelevantniji za alat *AutoFix* jeste njeno ispisivanje na standardni izlaz. U ovu svrhu *AutoFix* koristi funkcionalnost postojeće klase `TextDiagnosticPrinter` (opisane u sekciji 3.3). Pored ispisivanja poruka upozorenja i predloga za ispravljanje koda na standardni izlaz, alat *AutoFix* konzumira



Slika 4.7: Odnos klasa `DiagnosticConsumer`, `TextDiagnosticPrinter`, `AutoFixDiagnosticConsumer`, `Rewriter` i `Replacement`.

dijagnostiku tako što predložene izmene koda primenjuje na izvorni fajl ukoliko je prosleđena opcija komandne linije `--apply-fix`. Da bi se ovo postiglo u okviru alata *AutoFix* implementirana je klasa `AutoFixDiagnosticConsumer`. Ova klasa nasleđuje klasu `TextDiagnosticPrinter` i time zadržava funkcionalnost ispisivanja dijagnostike na standardni izlaz koja je implementirana u okviru nje.

Klasa `AutoFixDiagnosticConsumer` predefiniše metod `HandleDiagnostic` i u okviru njega pored ispisivanja dijagnostike kreira objekat klase `Replacement` za predloženu izmenu koda. Objekat klase `Replacement` čuva putanju do fajla (`std::string FilePath`), informacije o tome koje delove izvornog koda treba zameniti sa predlogom za ispravljanje koda i sam predlog (`Range ReplacementRange`, `std::string ReplacementText`). U okviru metode `finish` svi kreirani objekti klase `Replacement` se pridružuju objektu klase `Rewriter`. Ova klasa omogućava primenjivanje predloženih izmena koda na izvorni fajl. Predložene izmene koda primenjuju se pozivom metode `overwriteChangedFiles` klase `Rewriter`. Na slici 4.7 prikazan je odnos klasa `DiagnosticConsumer`, `TextDiagnosticPrinter`, `AutoFixDiagnosticConsumer`, `Rewriter` i `Replacement`.

AutoFixHelper.cpp

U okviru jedinice prevođenja `AutoFixHelper.cpp` implementirane su pomoćne funkcije korišćene u okviru alata *AutoFix*. U nastavku su ukratko opisane funkcije iz ove jedinice prevođenja.

- `getWordsFromString` — kreira niz reči od stringa prosleđenog u okviru opcija komandne linije.
- `getExprStr` — kreira string od čvorova apstraktnog sintaksičkog stabla tipa `Expr`.
- `getChildOfType` - pronalazi dete čvora iz apstraktnog sintaksičkog stabla koje ima određeni tip.
- `trimString` — Izbacuje prazne karaktere (eng. *whitespace characters*) sa početka i sa kraja stringa.
- `trimBraces` — Izbacuje karakter `{` sa početka i karakter `}` sa kraja stringa.

4.4 Opis testiranja alata

U okviru alata *AutoFix* testirana je implementacija svakog od podržanih pravila. Svaki test proverava implementaciju jednog pravila. Testovi `auto-fix-A7-1-6-.cpp`, `auto-fix-A7-2-3.cpp`, `auto-fix-A8-5-2.cpp` i `auto-fix-A8-5-3.cpp` nalaze se u okviru direktorijuma `autofix-test` i redom testiraju implementaciju pravila **A7-1-6**, **A7-2-3**, **A8-5-2** i **A8-5-3**. C++ kôd u okviru testova, nad kojim je testirana ispravnost rada alata, većinski je preuzet iz primera u okviru dokumenta u kom je opisan standard AUTOSAR C++14 [18]. Testovi su dopunjeni kodom za koji je autor ovog rada smatrao da ilustruje bitne slučajeve upotrebe a ne nalazi se u okviru primera iz dokumenta.

Na listingu 4.7 prikazan je pojednostavljeni primer testa za pravilo **A7-1-6** u okviru alata *AutoFix*. Test je pojednostavljen tako što se u okviru njega nalazi samo jedan konstrukt koji nije u skladu sa pravilom **A7-1-6**. Pojednostavljeni test je korišćen kako bi se demonstrirali svi bitni aspekata testiranja alata *AutoFix* sa što manje koda.

Test proverava da se pokretanjem alata `AutoFix` ispisuje adekvatno upozorenje zajedno sa predlogom za ispravljanje koda. U okviru `RUN` linije (linija 1) pokreće se alat *AutoFix* komandom `auto-fix --rules=„A7_1_6" %s 2>&1 --`. Simbol

`%s` će prilikom pokretanja biti zamenjen putanjom do testa u kome se ova komanda nalazi (testa koji *lit* pokreće). `2>&1` preusmerava standardni izlaz za greške (STDERR) na standardni izlaz (STDOUT). Drugi deo komande `| FileCheck %s` prosleđuje izlaz iz alata *AutoFix* na standardni ulaz alata *FileCheck* i pokreće alat *FileCheck* nad testom. Putanja do testa prosleđena je simbolom `%s` [15].

U okviru listinga 4.7 koriste se tri direktive alata *FileCheck*, `CHECK: TEKST` (linija 6), `CHECK-NEXT: TEKST` (linije 8-11) i `CHECK-NOT: TEKST` (linije 5 i 12). Ove direktive služe da se izvrši provera da li je alat ispisao upozorenje vezano za pravilo **A7-1-6**, predlog za ispravljanje koda i da se u okviru ispisa nije našlo nijedno drugo upozorenje. U okviru direktive `CHECK` koriste se regularni izrazi za putanju na operativnom sistemu Linuks (eng. *Linux*), za početak i kraj linije. Redom, ovo su regularni izrazi `(/|/[a-zA-Z0-9_-]+)+`, `^` i `$`. Upotrebom regularnih izraza za početak i kraj linije test proverava da alat *AutoFix* nije ispisao ništa nepredviđeno, odnosno da svaka linija počinje i završava se tekstom navedenim između ta dva regularna izraza. Regualrni izraz za putanju na operativnom sistemu Linux koristi se kako test ne bi očekivao specifičnu apsolutnu putanju u okviru ispisa alata *AutoFix*.

Listing 4.7: Pojednostavljeni primer testa za pravilo **A7-1-6** u okviru alata *AutoFix*.

```
1  // RUN: auto-fix --rules="A7_1_6" %s 2>&1 -- | FileCheck %s
2
3  typedef unsigned long ulong;
4
5  // CHECK-NOT: {{.+}}
6  // CHECK: {{^((/|/[a-zA-Z0-9_-]+)+)}}/auto-fix-A7-1-6.cpp:3:23:
       warning: The typedef specifier shall not be used.{{$}}
7  // CHECK-NEXT: {{^}}typedef unsigned long ulong;{{$}}
8  // CHECK-NEXT: {{^}}~~~~~~~~~~~~~~~~~~~~~~~^~~~{{$}}
9  // CHECK-NEXT: {{^}}using ulong = unsigned long{{$}}
10 // CHECK-NEXT: {{^}}1 warning generated.{{$}}
11 // CHECK-NOT: {{.+}}
```

4.5 Analiza rezultata rada alata

U svrhu provere robusnosti i kvaliteta alata *AutoFix*, izvršena je analiza nad delovima projekta *Automotive Grade Linux (AGL)* [2]. Projekat *AGL* je izabran

na osnovu svoje relevantnosti u automobilskoj industriji.

AGL je zajednički projekat otvorenog koda koji okuplja proizvođače automobila, dobavljače i tehnološke kompanije kako bi ubrzao razvoj i usvajanje potpuno otvorenog softverskog paketa (eng. *software stack*) za povezane automobile (eng. *connected cars*). Sa operativnim sistemom Linuks u svojoj osnovi, AGL razvija platformu koja može služiti kao *de facto* standard u industriji čime bi se omogućio brz razvoj novih funkcija i tehnologija [2].

Alat *AutoFix* pokretan je nad dva podprojekta projekta *AGL*. To su projekti *re2c* i *ninja*. Projekti su izabrani nasumično osim kriterijuma da opseg projekta bude relativno mali. Analiza rezultata rada alata, odnosno upozorenja koje je alat *AutoFix* prijavio, izvršena je ručno od strane autora ovog rada. Kriterijum da projekat bude manjeg obima korišćen je kako bi se smanjila verovatnoća greške prilikom pomenute analize rezultata. Alat je pokretan skriptom koji rekurzivno obilazi direktorijume u okviru projekta i pokreće alat *AutoFix* nad svim fajlovima sa ekstenzijom `.cpp` i `.cc`. Nad svakim fajlom, alat je pokrenut komandom
`auto-fix --rules=„all" putanja_do_fajla`

Projekat *re2c*

Projekat *re2c* nad kojim je pokretan alat *AutoFix* nalazi se u okviru projekta *AGL* na lokaciji `build/tmp/work/x86_64-linux/re2c-native`. Projekat sadrži 79 fajlova sa ekstenzom `.cc` čiji ukupan broj linija koda iznosi 21021. U okviru projekta nalazi se 77 zaglavlja čiji ukupan broj linija koda iznosi 3910 [1].

Tabela 4.1 prikazuje broj prijavljenih upozorenja za svako od pravila prilikom pokretanja alata *AutoFix*. Za svako od upozorenja ispisan je i odgovarajući predlog za ispravljanje koda. Upozorenje za pravilo **A8-5-3** nije prijavljeno nijednom. Pretragom alatom *grep* kroz svaki od fajlova ustanovljeno je da konstrukti na koje se odnosi pravilo **A8-5-3** zaista nisu korišćeni u okviru koda i da se ne radi o grešci u alatu *AutoFix*. Ovaj rezultat slaže se sa činjenicom da je najveći broj upozorenja prijavljen za pravilo **A8-5-2**. Pravilo **A8-5-2** predlaže upotrebu inicijalizacije vitičastim zagradama pri inicijalizaciji promenljive dok pravilo **A8-5-3** zabranjuje upotrebu inicijalizacije vitičastim zagradama prilikom deklaracije promenljivih sa tipom `auto`. S obzirom na veliki broj upozorenja za pravilo **A8-5-2** i činjenice da za pravilo **A8-5-3** nije prijavljeno nijedno upozorenje, zaključeno je da prilikom

[1] Broj linija koda je prebrojan alatom *cloc*. Alat je dostupan na linku `http://cloc.sourceforge.net/`.

inicijalizacije promenljivih nije korišćena sintaksa vitičastih zagrada ne vezano da li je promenljiva tipa `auto` ili ne. U kombinaciji sa činjenicom da je prijavljen veliki broj upozorenja za pravila **A7-1-6** i **A8-5-2** zaključeno je da kôd u okviru projekta *re2c* nije pisan po standardu AUTOSAR C++14.

Skript za pokretanje alata *AutoFix* nad projektom *re2c* pokrenut je deset puta kako bi se izračunalo prosečno vreme izvršavanja alata, koje iznosi 39.602 sekunde. Vreme je izmereno programom `time` u okviru operativnog sistema Linux. Izračunato prosečno vreme se odnosi na komponentu `real` u okviru rezultata programa `time`.

Tabela 4.1: Broj prijavljenih upozorenja po pravilu za projekat *re2c*.

Broj prijavljenih upozorenja po pravilu				
Ime projekta	Pravilo A7-1-6	Pravilo A7-2-3	Pravilo A8-5-2	Pravilo A8-5-3
re2c	1329	714	2561	0

Projekat *ninja*

Projekat *ninja* nad kojim je pokretan alat *AutoFix* nalazi se u okviru projekta *AGL* na lokaciji `build/tmp/work/x86_64-linux/ninja-native`. Projekat sadrži 55 fajlova sa ekstenzom `.cc` čiji ukupan broj linija koda iznosi 12587. U okviru projekta nalazi se 43 zaglavlja čiji ukupan broj linija koda iznosi 2291.

Tabela 4.2 prikazuje broj prijavljenih upozorenja za svako od pravila prilikom pokretanja alata *AutoFix*. Na osnovu rezultata sa slike primećeno je da je odnos broja prijavljenih upozorenja po pravilu sličan rezultatima prikazanim na tabeli 4.1 dobijenim pri pokretanju alata *AutoFix* nad projektom *re2c*. Na osnovu ovoga, zaključci izvedeni tokom diskusije rezultata nad projektom *re2c* važe i za rezultate projekta *ninja*. Skript za pokretanje alata *AutoFix* nad projektom *ninja* pokrenut je deset puta kako bi se izračunalo prosečno vreme izvršavanja alata, koje iznosi 18.002 sekunde.

Rezime analize rezultata rada alata

Alat se uspešno izvršio prilikom pokretanja nad svakim fajlom sa ekstenzijom `.cc` u okviru projekata *re2c* i *ninja*. Prilikom ručne analize upozorenja koje je alat *AutoFix* prijavio nisu uočene greške u radu alata. Na osnovu dobijenih rezultata

Tabela 4.2: Broj prijavljenih upozorenja po pravilu za projekat *ninja*.

Broj prijavljenih upozorenja po pravilu				
Ime projekta	Pravilo A7-1-6	Pravilo A7-2-3	Pravilo A8-5-2	Pravilo A8-5-3
ninja	334	188	1007	0

može se zaključiti da je alat *AutoFix* robustan i da se može efikasno koristiti nad realnim industrijskim projektima.

Glava 5

Zaključak

Standardi za pravilno pisanje koda u programskom jeziku definišu niz pravila koje programer treba da sledi tokom razvoja softvera. Primena ovakvih standarda tokom razvoja softvera povećava kvalitet softvera time što smanjuje verovatnoću pojavljivanja greške u kodu. U automobilskoj industriji, najzastupljeniji standard za pravilno pisanje koda u jeziku C++14 je standard AUTOSAR C++14.

Ručno proveravanje da li je kôd napisan u skladu sa standardom predstavlja mukotrpan i neefikasan proces. U svrhu automatizovanja ovog procesa koriste se alatke za statičku analizu koda, koji bez pokretanja programa detektuju kôd koji nije napisan u skladu sa standardom. Alat *AutoFix*, koji je razvijen u ovom radu, ispisuje upozorenja vezana za kôd koji nije napisan u skladu sa podskupom pravila iz standarda AUTOSAR C++14 koja se odnose na deklaracije u programskom jeziku C++14 i predlaže kako izmeniti kôd da bi bio u skladu sa standardom. *AutoFix* podržava i opciju komandne linije kojom se predložene izmene mogu primeniti na izvorni kôd. Alat je razvijen korišćenjem biblioteka koje pruža kompilatorska infrastruktura LLVM. Pravila standarda AUTOSAR C++14 podržana u okviru alata *AutoFix* su **A8-5-3**, **A8-5-2**, **A7-1-6**, **A7-2-3**. Detaljnim testiranjem upotrebom alatki *lit* i *FileCheck* iz kompilatorske infrastrukture LLVM obezbeđen je i visok kvalitet razvijenog alata. Analizom rezultata dobijenim pokretanjem alata *AutoFix* nad podprojektima projekta *AGL* zaključeno je da u okviru podprojekta *re2c* i *ninja* kôd nije napisan po standardu AUTOSAR C++14.

U daljem razvoju alat se može unaprediti na nekoliko načina. Statička analiza u okviru alata mogla bi se unaprediti upotrebom naprednijih tehnika kao što je simboličko izvršavanje programa. Ovakva analiza omogućila bi i podršku značajno šireg skupa pravila. U ovu svrhu u okviru alata *AutoFix* mogao bi se integrisati

statički analizator kompilatora *Clang* koji omogućava ovakav tip analize. Alat bi se mogao unaprediti i implementiranjem dodatnih opcija komandne linije koje bi omogućile korisniku veću kontrolu nad samim alatom. Na primer, mogla bi se dodati opcija koja omogućava korisniku da isključi analizu u zadatim delovima koda.

Literatura

[1] AST Matcher Reference. https://clang.llvm.org/docs/LibASTMatchersReference.html.

[2] Automotive Grade Linux. https://www.automotivelinux.org/.

[3] Choosing the Right Interface for Your Application. https://clang.llvm.org/docs/Tooling.html.

[4] Clang Plugins. https://clang.llvm.org/docs/ClangPlugins.html.

[5] Clang Static Analyzer website. https://clang-analyzer.llvm.org/.

[6] clang::ASTConsumer Class Reference. https://clang.llvm.org/doxygen/classclang_1_1ASTConsumer.html.

[7] clang::ASTFrontendAction Class Reference. https://clang.llvm.org/doxygen/classclang_1_1ASTFrontendAction.html.

[8] clang::RecursiveASTVisitor<Derived> Class Template Reference. https://clang.llvm.org/doxygen/classclang_1_1RecursiveASTVisitor.html.

[9] CXCursor Struct Reference. https://clang.llvm.org/doxygen/structCXCursor.html.

[10] FileCheck - Flexible pattern matching file verifier. https://llvm.org/docs/CommandGuide/FileCheck.html.

[11] How To Setup Clang Tooling For LLVM. https://clang.llvm.org/docs/HowToSetupToolingForLLVM.html.

[12] How to write RecursiveASTVisitor based ASTFrontendActions. https://clang.llvm.org/docs/RAVFrontendAction.html.

[13] ISO official website. `https://www.iso.org/committee/45202.html`.

[14] libclang: C Interface to Clang. `https://clang.llvm.org/doxygen/group__CINDEX.html`.

[15] lit - LLVM Integrated Tester. `https://llvm.org/docs/CommandGuide/lit.html`.

[16] Matching the Clang AST. `https://clang.llvm.org/docs/LibASTMatchers.html`.

[17] "Clang" CFE Internals Manual. `https://clang.llvm.org/docs/InternalsManual.html`.

[18] AUTOSAR. Guidelines for the use of the C++14 language in critical and safety-related systems, 2017.

[19] AUTOSAR. AUTOSAR official website, 2018.

[20] Milena Vujošević Janičić, Ognjen Plavšić, Mirko Brkušanin, and Petar Jovanović. AUTOCHECK: A Tool For Checking Compliance With Automotive Coding Standards. *Zooming Innovation in Consumer Electronics International Conference (ZINC)*, 2021.

[21] Bruno Cardoso Lopes. *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014.

[22] Bjarne Stroustrup. *The C++ Programming Language*. Addison–Wesley, fourth edition, 2013.

[23] Đorđe Milićević, Mirko Brkušanin, Milena Vujošević Janičić, Teodora Novković, and Petar Jovanović. Unapređenje programskog prevodioca Clang sa podrškom za standard MISRA/AUTOSAR. *Etran*, 2019.

Biografija autora

Ognjen Plavšić rođen je 14.06.1995. u Leskovcu. Završio je Gimnaziju u Leskovcu, Matematički smer, 2014. godine i iste godine upisao Matematički fakultet u Beogradu. 2019. godine je završio osnovne studije Matematičkog fakulteta sa prosečnom ocenom 9.14 i iste godine upisao master studije. Marta 2019. godine kreće na praksu u Naučno-istraživačkom centru RT-RK (kasnije Syrmia), gde se oktobra iste godine zapošljava na poziciji softverskog inženjera. Radio je na projektu čiji je cilj bio kreiranje alata za statičku analizu u okviru kompilatorske infrastrukture LLVM. Jula 2021. godine prelazi u kompaniju HTEC Group gde i danas radi kao softverski inženjer. Trenutno se bavi kompilatorima za mašinsko učenje (eng. *machine learning (ML) compilers*). U okviru ovih kompilatora radi na generisanju i optimizaciji koda od modela mašinskog učenja predstavljenim u nekom od formata poznatih okruženja mašinskog učenja (eng. *machine learning frameworks*). Vezano za temu master teze, ima objavljen rad na međunarodnoj konferenciji *ZINC* [20]:

1. *Milena Vujošević Janičić, Ognjen Plavšić, Mirko Brkušanin, Petar Jovanović: AUTOCHECK: A Tool For Checking Compliance With Automotive Coding Standards, 2021 Zooming Innovation in Consumer Electronics International Conference (ZINC), (Novi Sad, Serbia)*