

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 sigurnošću. 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 alatima za statičku analizu koji bi automatizovali proces provere da li je kôd napisan u skladu sa standardom. Kompajlerska infrastruktura LLVM pruža podršku za jednostavno razvijanje kvalitetnih alata 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, kompajleri, LLVM, Clang

Sadržaj

1	Uvod	1
2	Standard kodiranja AUTOSAR C++14	3
2.1	Programski jezik C++	3
2.2	Klasifikacija pravila	5
3	Kompajlerska infrastruktura LLVM	10
3.1	LLVM i Clang	11
3.2	Biblioteka clangAST	12
3.3	AST uparivači	17
3.4	Interfejsi za akcije nad prednjim delom kompajlera	19
3.5	Biblioteke za kreiranje alata	23
4	Alat Autofix	29
4.1	Korišćenje alata	29
4.2	Opis implementiranih pravila	30
4.3	Primeri rada alata	31
4.4	Opis implementacije	36
5	Zaključak	38
	Literatura	40

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 automobilske 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 alati za statičku analizu koda. Alati za statičku analizu proveravaju ispravnost programa bez njegovog izvršavanja. Ovakvi alati se mogu implementirati na više načina. Jednostavniji alati koriste informacije dobijene tokom kompilacije programa da utvrde da li kôd sadrži grešku. Napredniji alati za statičku analizu vrše i simboličko izvršavanje programa, odnosno koriste različite tehnike kojima simuliraju izvršavanje programa, bez njegovog pokretanja.

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

AutoFix proverava da li je kôd napisan u skladu sa podskupom pravila iz standarda AUTOSAR C++14 koji 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 izmene 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 kompajlerske strukture LLVM koji su korišćeni za izradu alata *AutoFix*, koji predstavlja rezultat ovog rada. 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 [12]. 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 sigurnošću. 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 [11].

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 bilo je mogućnost kreiranja korisnički definisanih tipova, odnosno klasa. 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 [14].

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 [8]. 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.

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 primenljivosti statičke analize

Klasifikacija po primenljivosti 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 delimično automatizovati, na primer, korišćenjem neke heuristike ili pokrivanjem određenog broja

slučajeva upotrebe i služi kao dopuna pregleda koda. 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.

Većina pravila iz standarda AUTOSAR C++14 spadaju u automatizovana pravila. Alati za statičku analizu koda koji tvrde da podržavaju standard AUTOSAR C++14 moraju u potpunosti obezbediti 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 [11].

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 alati 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 (obvezno, implementaciono, 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 da precizne rezultate.

```
1 | #include <iostream>
2 | #include <cstdint>
3 | #include <cstdlib>
4 |
5 | int main() {
```

```
6 | int8_t u8a = rand() % 100;  
7 | u8a = (uint8_t) ( u8a << rand() % 10 );  
8 | }
```

Međitim, ukoliko je desni operand konstanta ili promenljiva konstantnog izraza (ključna reč *constexpr*), alat za statičku analizu može da proveriti 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.

```
1 | #include <iostream>  
2 | #include <cstdint>  
3 | #include <cstdlib>  
4 |  
5 | int main(){  
6 |     int8_t u8a = rand() % 100;  
7 |     u8a = (uint8_t) ( u8a << 7 );  
8 | }
```

Napredniji alati za statičku analizu koji podržavaju simboličko izvršavanje programa (npr. *Clang Static Analyzer* [3]) 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 alate za statičku analizu. Ovo uglavnom znači da je broj slučajeva upotrebe koji algoritmi iz statičkih alata 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 (obvezno, implementaciono, 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 interni alati za statičku analizu koda u kombinaciji sa manuelnim pregledom koda.

Klasifikacija pravila prema cilju primene

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

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

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

Pravilo A2-9-1 (obvezno, implementaciono, automatizovano)
Ime heder fajla mora biti identično imenu tipa deklarisanog u njemu ukoliko deklarise tip.

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

Pravilo A15-0-6 (obvezno, 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 alate odnose se na softverske alate kao što su pretprocesor, kompajler, linker i biblioteke kompajlera. Infrastrukturna pravila odnose se na operativni sistem i hardver [11]. Primer pravila za alate koje je ujedno i infrastrukturno pravilo:

Pravilo A0-4-1 (obvezno, pravilo za infrastrukturu/alate, neautomatizovano)

Implementacija brojeva u pokretnom zarezu treba da bude u skladu sa standardom IEEE 754.

Glava 3

Kompajlerska infrastruktura LLVM

U ovom poglavlju opisane su biblioteke i klase kompajlerske 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. Za bazne apstraktne klase u daljem tekstu koristi se termin *interfejs*.

S obzirom da se alat *AutoFix* zasniva na analizi apstraktnog sintaksnog 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 objašnjen interfejs `RecursiveASTVisitor` koji 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 sintaksnog stabla.

Interfejsi `ASTConsumer` i `FrontendAction` omogućavaju interakciju alata sa prednjim delom kompajlera. 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 alata. Pored ove biblioteke podršku za kreiranje alata pružaju i biblioteke `LibClang` i `ClangPlugins`. U okviru ovog poglavlja diskutovane su prednosti i mane upotrebe ovih biblioteka kao i razlozi zbog kojih je biblioteka `LibTooling` izabrana za implementaciju alata *AutoFix*.

3.1 LLVM i Clang

Kompajlerska infrastruktura LLVM predstavlja kolekciju modularnih i ponovo iskoristivih kompajlerskih tehnologija i alata. Ova kompajlerska infrastruktura započeta je kao inistraživački projekat Krisa Latnera (*eng. Chris Lattner*) i Vikrama Advea (*eng. Vikram Adve*) na Univerzitetu Illinois 2000. godine. Dizajn LLVM-a omogućava jednostavno dodavanje podrške za kompilaciju za specifičnu arhitekturu hardvera. Kompajlerska 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 sintaksnog stabla (*eng. abstract syntax tree (AST)*) i generisanje LLVM međukoda (*eng. intermediate representation (IR)*) koristeći informacije iz apstraktnog sintaksnog stabla.
2. Srednji deo kompajlera vrši niz optimizacija nad instrukcijama LLVM međukoda. LLVM međukod predstavlja apstrakciju asemblera koja je nezavisna 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 kompajlera vrši mašinski zavisne optimizacije koda i generiše mašinski kôd za ciljnu arhitekturu.

Clang predstavlja prednji deo (*eng. frontend*) kompajlerske 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 sintaksnog stabla. Ova biblioteka koristi funkcionalnosti biblioteke `clangSema` kako bi ispitala semantičku validnost sintaksnih konstrukta (niza tokena) od kojih kreira čvorove apstraktnog sintaksnog stabla. Parser kompajlera *Clang* je implementiran kao parser rekurzivnog spuštavanja (eng. *recursive-descent parser*), odnosno analizira izvorni kôd od vrha ka dnu nizom rekurzivnih funkcija [13].

Biblioteka `clangAST` implementira algoritme i strukture podataka koje parser koristi za izgradnju AST-a. Specifična je po strukturi čvorova koji podsećaju na izvorni C++ kôd što je čini pogodnom za kreiranje alata 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. Usko je povezana sa bibliotekama `clangParse` i `clangAST`.

Biblioteka `clangCodeGen` dobija AST kao ulaz i od njega generiše LLVM međukod.

3.2 Biblioteka `clangAST`

U računarstvu, **apstraktno sintakšno stablo**, ili samo **sintakšno stablo**, je drvoidna reprezentacija apstraktne sintaktičke 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 kompajlera *Clang* i jednostavnost kreiranja moćnih alata za statičku analizu u velikoj meri oslanja se na dizajn biblioteke `clangAST`. Struktura AST-a može se jednostavno ispisati na standardni izlaz opcijom komandne linije `-ast-dump`. Komanda na listingu 3.1 ispisuje na standardni izlaz AST za kôd iz fajla `hello.cpp` prikazanog na listingu 3.2. Slika ?? predstavlja tekstualnu reprezentaciju AST-a ispisanu komandom iz listinga 3.1.

Listing 3.1: Komanda za ispisivanje Clang-ovog AST-a.

```
1 || $ clang -Xclang -ast-dump hello.c
```

Listing 3.2: Kôd čiji je AST prikazan na slici 3.1.

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

```
TranslationUnitDecl 0x55d2e32c1448 <<invalid sloc>> <invalid sloc>
- TypedefDecl 0x55d2e32c1a00 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
- BuiltinType 0x55d2e32c16e0 '__int128'
- TypedefDecl 0x55d2e32c1a70 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
- BuiltinType 0x55d2e32c1700 'unsigned __int128'
- TypedefDecl 0x55d2e32c1db8 <<invalid sloc>> <invalid sloc> implicit __NSConstantString '__NSConstantString_tag'
- RecordType 0x55d2e32c1b60 '__NSConstantString_tag'
- CXXRecord 0x55d2e32c1ac8 '__NSConstantString_tag'
- TypedefDecl 0x55d2e32c1e50 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
- PointerType 0x55d2e32c1e10 'char *'
- BuiltinType 0x55d2e32c14e0 'char'
- TypedefDecl 0x55d2e32f8c48 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list '__va_list_tag [1]'
- ConstantArrayType 0x55d2e32f8bf0 '__va_list_tag [1]' 1
- RecordType 0x55d2e32c1f40 '__va_list_tag'
- CXXRecord 0x55d2e32c1ea8 '__va_list_tag'
- FunctionDecl 0x55d2e32f8cf8 <vezba.cpp:1:1, line:5:1> line:1:5 main 'int ()'
- CompoundStmt 0x55d2e32f90e8 <col:11, line:5:1>
- DeclStmt 0x55d2e32f8ea0 <line:2:1, col:10>
- VarDecl 0x55d2e32f8e20 <col:1, col:9> col:5 used a 'int' cinit
- IntegerLiteral 0x55d2e32f8e80 <col:9> 'int' 4
- DeclStmt 0x55d2e32f8f50 <line:3:1, col:10>
- VarDecl 0x55d2e32f8ed0 <col:1, col:9> col:5 used b 'int' cinit
- IntegerLiteral 0x55d2e32f8f30 <col:9> 'int' 5
- DeclStmt 0x55d2e32f90d0 <line:4:1, col:23>
- VarDecl 0x55d2e32f8f80 <col:1, col:22> col:5 result 'int' cinit
- BinaryOperator 0x55d2e32f90a8 <col:14, col:22> 'int' '+'
- ImplicitCastExpr 0x55d2e32f9030 <col:14> 'int' <LValueToRValue>
- DeclRefExpr 0x55d2e32f8fe0 <col:14> 'int' lvalue Var 0x55d2e32f8e20 'a' 'int'
- ImplicitCastExpr 0x55d2e32f9048 <col:18> 'int' <LValueToRValue>
- DeclRefExpr 0x55d2e32f9008 <col:18> 'int' lvalue Var 0x55d2e32f8ed0 'b' 'int'
- IntegerLiteral 0x55d2e32f9088 <col:22> 'int' 8
```

Slika 3.1: Primer reprezentacije AST-a generisanog opcijom `-ast-dump`. AST prikazan na slici prezentuje kôd iz listinga 3.1.

Čvorovi od kojih je izgrađen AST predstavljaju apstrakciju sintaksnih struktura iz samog jezika. Svi čvorovi Clang-ovog AST-a 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, `IfStmt` klasa opisuje `if` naredbe jezika i direktno nasleđuje `Stmt` klasu. Sa druge strane, `FunctionDecl` i `VarDecl` klase, koje se koriste za opisivanje deklaracija i definicija funkcija i varijabli, ne nasleđuju direktno klasu `Decl` već nasleđuju više njenih podklasa.

AST čvorovi 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 AST-a. Ovakve informacije posebno su korisne za kreiranje preciznih poruka dijagnostike koda.

Klasa `Type`

Klasa `Type` igra važnu ulogu u ekspresivnosti dijagnostike kompajlera *Clang*, a samim tim i u kvalitetu alata za statičku analizu. Ova klasa omogućava 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, std::... >`. Iza ove funkcionalnosti stoji ideja kanonskih tipova.

Svaka instanca klase `Type` sadrži pokazivač na svoj kanonski tip. Za jednostavne tipove koji nisu definisani korišćenjem `typedef` naredbe pokazivač na kanonski tip će zapravo pokazivati na sebe. Za tipove čija struktura uključuje `typedef` naredbu kanonski pokazivač pokazivaće na strukturno ekvivalentan tip bez `typedef` naredbi. Na primer, kanonski tip tipa `int *` sa listinga 3.3 biće sam taj tip, dok će kanonski tip za `foo *` biti `int *`.

Listing 3.3: 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 `typedef` naredbe 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 omogućuje 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 [10].

AST posetioci

AST posetioci implementiraju mehanizam obilaska apstraktnog sintaksnog stabla kompajlera *Clang*, odnosno pružaju interfejs za posećivanje svakog čvora u apstraktnom sintaksnom stablu. Funkcionalnost AST posetioca implementirana je u okviru šablonske klase `RecursiveASTVisitor<Derived>`. Objekat ove klase posećuje svaki čvor apstraktnog sintaksnog stabla obilaskom u dubinu. AST posetilac je svaka potklasa klase `RecursiveASTVisitor<Derived>`. Klasa `RecursiveASTVisitor<Derived>` omogućava obavljanje tri odvojena zadatka:

1. Obilazak apstraktnog sintaksnog stabla, odnosno posećivanje svakog čvora.
2. Kreiranje kombinacija oblika (*čvor*, *klasa*). Ove kombinacije sadrže sve klase 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. Metod `TraverseDecl(Decl *x)` obavlja prvi zadatak. Ovo je ulazna tačka za obilazak AST-a sa korenom u čvoru `x`. Ovaj metod poziva 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)`.

Metode `TraverseStmt(Stmt *x)` i `TraverseType(QualType x)` implementirane su na sličan način.

2. Metod `WalkUpFromFoo(Foo *x)` izvršava drugi zadatak. 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)` izvršava treći zadatak.

Na primer, metod `VisitCXXRecordDecl(CXXRecordDecl *x)` može biti predefinisano kako bi se analizirao čvor `x`.

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. Metoda ne može pozvati metode iz grupe iznad [6]. Na primer, metod `WalkUpFrom` može pozvati metode iz svoje grupe i grupe `Visit` ali ne može pozvati metode iz grupe `Traverse`.

Da bi se izvršila analiza izvornog koda pomoću AST posetioca dovoljno je naslediti klasu `RecursiveASTVisitor<Derived>` i predefinisati željene `Visit` metode u okviru nje. Ukoliko je `Visit` metodama pronađen nepravilan konstrukt izvornog koda može se prijaviti upozorenje. Primer posetioca prikazan je na listingu 3.4.

Listing 3.4: Primer posetioaca koji posećuje sve strukture, unije i klase i ispisuje lokaciju onih čiji naziv je `matematika::geometrija::Trogao`.

```
1 | class FindNamedClassVisitor : public RecursiveASTVisitor<FindNamedClassVisitor> {
2 | public:
3 |     explicit FindNamedClassVisitor(ASTContext *Context)
4 |         : Context(Context) {}
5 |
6 |     bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
7 |
8 |         if (Declaration->getQualifiedNameAsString() == "matematika::geometrija::Trogao"
9 |             ) {
10 |
11 |             FullSourceLoc FullLocation = Context->getFullLoc(Declaration->getBeginLoc());
12 |
13 |             if (FullLocation.isValid())
14 |                 llvm::outs() << "Nadjena deklaracija na "
15 |                             << FullLocation.getSpellingLineNumber() << ":"
16 |                             << FullLocation.getSpellingColumnNumber() << "\n";
17 |         }
18 |     }
```

```
17 |  
18 |     return true;  
19 | }  
20 |  
21 | private:  
22 |     ASTContext *Context;  
23 | };
```

3.3 AST uparivači

Biblioteka AST uprarivača (eng. *LibASTMatcher*) implementira jezik specijalne namene za kreiranje predikata nad AST-om kompajlera *Clang*. Ovaj jezik specijalne namene napisan je i može se koristiti u jeziku C++. Ovo omogućava korisnicima da u istom programu pristupe željenom delu stabla i da nad tim čvorovima koriste C++ interfejs za analiziranje raznih atributa, lokacija, i ostalih informacija dostupnih na AST nivou. Izrazi za uparivanje (eng. *match expressions*) omogućuju uparivanje delova apstraktnog sintaksnog stabla tako što kreiraju i vraćaju uparivače.

Uopšteno, strategija kreiranja uparivača se svodi na sledeće korake:

1. Naći baznu klasu AST čvora kog je potrebno upariti.
2. Naći u *AST Matcher Reference* dokumentu [1] uparivač koji ili uparuje željeni čvor ili sužava pretragu.
3. Kreirati spoljašnji izraz za uparivanje i proveriti da li radi očekivano.
4. Pronaći uparivače koji bi mogli upariti neki unutrašnji čvor iz željenog dela stabla.
5. Ponavljati postupak dok uparivanje željenog dela stabla nije završeno.

Na primer, za kreiranje uparivača koji uparuje sve deklaracije enumeratora iz AST-a jedinice prevođenja može se koristiti izraz za uparivanje `enumDecl()`. Ukoliko ne treba analizirati deklaracije iz fajlova zaglavlja (eng. *header files*), izraz za uparivanje može se proširiti izrazom `isExpansionInMainFile()`. Izraz `enumDecl(isExpansionInMainFile())` kreiraće uparivač koji će upariti samo deklaracije enumeratora iz glavnog (.cpp) fajla.

Nakon uparivanja, nad uparenim konstruktom uglavnom se vrši 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 (rezultata svakog od uparivača u kompoziciji) zasebno. Zbog toga, uparivači se mogu „vezati” (eng. *binding*) za određeni string. Na primer,

```
enumDecl(isExpansionInMainFile()).bind(„A7_2_3_Matcher”)
```

će vezati uparene deklaracije enumeratora za string „A7_2_3_Matcher”. Rezultati uparivanja predstavljeni su kao objekti klase `MatchResult`. Konkretni čvor koji predstavlja rezultat uparivanja za uparivač vezan za string „A7_2_3_Matcher”, može se dobiti izrazom

```
auto ED = Result.Nodes.getNodeAs<clang::EnumDecl>(„A7_2_3_Matcher”).
```

Nakon formulisanja izraza za uparivanje kreirani uparivač se treba pokrenuti nad AST-om. Ovo se postiže pozivanjem metoda `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 poziva treba implementirati klasu koja nasleđuje `MatchCallback` klasu i predefinisati metod `run`. U okviru metode `run` može se vršiti dodatna analiza uparenih čvorova i po potrebi prijavljivati dijagnostika vezana za kôd koji taj rezultat predstavlja.

Na listingu 3.5 prikazan je primer uparivača koji pronalazi sve enumeratore koji nisu deklarirani koristeći specifikator `class`. Za ove enumeratore prijavljuje se upozorenje zajedno sa predlogom ispravke koda (eng. *fixit hint*) u okviru funkcije `emitWarningWithHintInsertion` (linija 20). U svrhu samog prijavljivanja upozorenja funkciji se prosleđuje objekat klase `DiagnosticsEngine`. Ova klasa zadužena je prijavljivanje dijagnostike u okviru kompajlera *Clang*. U funkciji `matchASTExample` kreiraju se uparivač (linija 31) i objekat klase povratnog poziva (linija 33). Nakon toga, uparivač se registruje za obilazak (linija 35) i pokreće nad apstraktnim sintaksnim stablom pomoću objekta klase `MatchFinder` (linija 37).

Listing 3.5: Primer uparivača koji pronalazi sve deklaracije enumeratora koji nisu deklarirani koristeći specifikator `class`. Ovaj primer demonstrira i upotrebu klase

MatchFinder, MatchCallback i MatchResult.

```
1 // Callback class.
2 class A7_2_3 : public MatchFinder::MatchCallback {
3 public:
4     A7_2_3(ASTContext &ASTCtx) : ASTCtx(ASTCtx) {}
5     virtual void run(const MatchFinder::MatchResult &Result);
6
7 private:
8     ASTContext &ASTCtx;
9 };
10
11 void A7_2_3::run(const MatchFinder::MatchResult &Result) {
12     if (auto ED = Result.Nodes.getNodeAs<clang::EnumDecl>("A7_2_3_Matcher")) {
13         // Check if declaration contains 'class' tag.
14         if (!ED->isScopedUsingClassTag()) {
15             // Create warning string.
16             std::string msg =
17                 "Enumerations shall be declared as scoped enum classes.";
18             std::string insStr = "class ";
19             // Function for emitting warnings with fixit hints using diagnostics engine.
20             emitWarningWithHintInsertion(
21                 ASTCtx.getDiagnostics(), msg, insStr,
22                 ED->getSourceRange().getBegin().getLocWithOffset(5),
23                 ED->getLocation());
24         }
25     }
26 }
27
28 void matchASTExample(ASTContext *Context){
29     MatchFinder Finder;
30     // Create matcher.
31     Matcher<Decl> Matcher = enumDecl(isExpansionInMainFile()).bind("A7_2_3_Matcher");
32     // Create callback class.
33     MatchCallback *Callback = new A7_2_3(Context);
34     // Register matcher.
35     Finder.addMatcher(Matcher, Callback);
36     // Run matcher over AST.
37     Finder.matchAST(Context);
38 }
```

3.4 Interfejsi za akcije nad prednjim delom kompajlera

Akcije nad prednjim delom kompajlera omogućavaju analizu i upotrebu rezultata i informacija koje pruža prednji deo kompajlera. Ove informacije mogu biti korisne za kreiranje alata za refaktorisanje koda, statičku analizu, prikupljanje

statistike, grafičko prezentovanje rezultata kompajlera ali igraju i ključnu ulogu u samoj kompilaciji koda i deo su osnovnog sistema (eng. *pipeline*) u infrastrukturi LLVM-a. Ova funkcionalnost je efikasno i sistematično implementirana u okviru klase `ASTConsumer`, `ASTFrontendAction` i njihovih potklasa.

Interfejs `ASTConsumer`

`ASTConsumer` je apstraktni interfejs koji omogućava izvršavanje različitih akcija nad apstraktnim sintaksnim stablom nezavisno od toga kako je AST kreiran. Akcije se mogu izvršavati u različitim fazama tokom kreiranja apstraktnog sintaksnog stabla [7]. Na primer, metod

```
virtual void HandleInlineFunctionDefinition (FunctionDecl *D)
```

biće pozvan svaki put kada se završi kreiranje umetnutih (eng. *inline*) funkcija prilikom kreiranja apstraktnog sintaksnog stabla. `ASTConsumer` definiše niz sličnih virtuelnih metoda koje mogu biti predefinisane od strane klase koje nasleđuju ovaj interfejs. Na slici 3.2 prikazane su klase u okviru *Clang* kompajlera koje nasleđuju klasu `ASTConsumer`. Klasa `CodeGenerator`, prikazana na slici, generiše LLVM međukod od apstraktnog sintaksnog stabla i predstavlja jedan od osnovnih delova u infrastrukturi LLVM-a, što demonstrira značaj interfejsa `ASTConsumer`.

Ovaj interfejs koristan je i za kreiranje samostalnih alata za statičku analizu koji se baziraju na analizi apstraktnog sintaksnog stabla. U ovu svrhu može se koristiti kombinacija ovog interfejsa sa mehanizmima za obilazak i obradu apstraktnog sintaksnog stabla kao što su AST posetioci i AST uparivači.

Za implementaciju specifične akcije nad apstraktnim sintaksnim stablom dovoljno je implementirati potkasu klase `ASTConsumer` i u okviru nje predefinisati metod

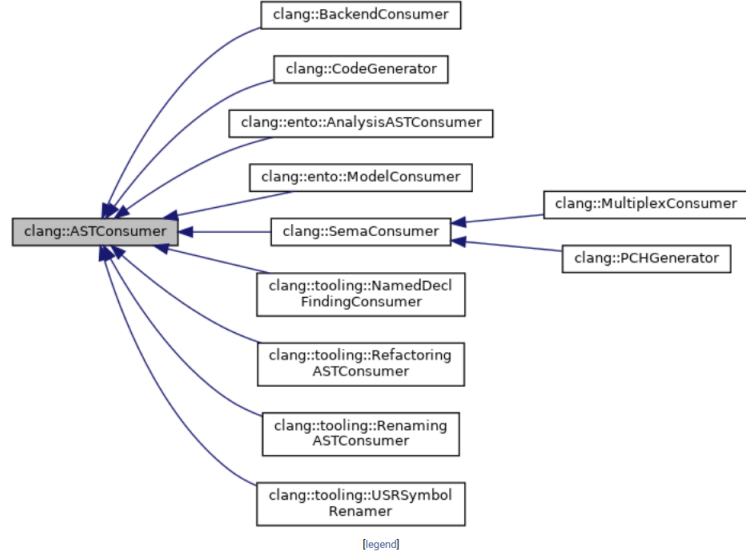
```
virtual void HandleTranslationUnit(ASTContext &Ctx).
```

Ovaj metod biće pozvan nakon što je kreiran AST za jedinicu prevođenja, odnosno u trenutku kada je celokupan AST za jedinicu prevođenja dostupan. U okviru njega, nad apstraktnim sintaksnim stablom, može se pozvati AST uparivač koji će izvršiti obilazak i analizu apstraktnog sintaksnog stabla [4].

Na listingu 3.6 prikazana je implementacije klase `AutoFixConsumer`. Ova klasa, u okviru alata *AutoFix*, služi za pokretanje uparivača nad apstraktnim sintaksnim stablom. U okviru opisanog metoda `HandleTranslationUnit` parsiraju se pravila koje je korisnik zadao kao argumente komandne linije (linija 8). Za svako od pravila u petlji (linija 20) se kreira po jedan uparivač (u okviru

#include "clang/AST/ASTConsumer.h"

Inheritance diagram for clang::ASTConsumer:



Slika 3.2: Klase koje implementiraju ASTConsumer interfejs.

funkcije `matcherFactory`) i jedan objekat povratnog poziva (u okviru funkcije `printerFactory`). Ukoliko je prosleđen string „all“, kreiraju se svi uparivači koje alat *AutoFix* podržava (linija 10). Uparivači i povratni pozivi se zatim registruju za obilazak apstraktnog sintaksnog stabla koji izvršava objekat klase `MatchFinder` (linija 30). Nakon registrovanja poziva se metod `matchAST` kojim se registrovani uparivači pokreću nad apstraktnim sintaksnim stablom (linija 37).

Listing 3.6: Implementacija klase `AutoFixConsumer` u okviru alata *AutoFix*.

```

1 | class AutoFixConsumer : public clang::ASTConsumer {
2 | public:
3 |     explicit AutoFixConsumer(ASTContext *Context, SourceManager &SM) : SM(SM) {}
4 |
5 |     virtual void HandleTranslationUnit(clang::ASTContext &Context) {
6 |         auto &DE = Context.getDiagnostics();
7 |
8 |         // Parse rules added as a part of command line option -rules.
9 |         SmallSet<std::string, 20> RulesMap = parseComaSeparatedWords(Rules);
10 |
11 |         // If string "all" is passed to -rules option, add rule strings
12 |         // for all supported rules within Autofix tool.
13 |         if (RulesMap.count("all")) {
14 |             RulesMap.erase("all");
15 |             for (const auto &Rule : SupportedRulesVec) {

```

```
16         RulesMap.insert(Rule);
17     }
18 }
19
20 MatchFinder Finder;
21 std::vector<internal::Matcher<Decl>*> MatcherVec;
22 std::vector<MatchFinder::MatchCallback*> PrinterVec;
23 for (const auto &MatcherName : RulesMap) {
24     // Create matcher from rule string.
25     internal::Matcher<Decl> *Matcher = matcherFactory(MatcherName);
26     // Create printer (MatcherCallBack object) from rule string.
27     MatchFinder::MatchCallback *Printer =
28         printerFactory(MatcherName, Context, SM);
29
30     MatcherVec.push_back(Matcher);
31     PrinterVec.push_back(Printer);
32     // Register matchers.
33     Finder.addMatcher(*Matcher, Printer);
34 }
35
36 // Run matchers over AST for translation unit.
37 Finder.matchAST(Context);
38
39 for (auto *Matcher : MatcherVec) {
40     delete Matcher;
41 }
42 for (auto *Printer : PrinterVec) {
43     delete Printer;
44 }
45 }
46 SourceManager &SM;
47 };
```

Intefejs ASTFrontendAction

FrontendAction je apstraktna klasa za akcije koje mogu biti izvršene od strane prednjeg dela kompajlera (eng. *frontend*). Klasa FrontendAction ima raznolike upotrebe, odnosno specijalizacije. Primer specijalizacija ove klase su DumpCompilerOptionsAction koja omogućava ispisivanje opcija koje se mogu zadati kompajleru i PreprocessorFrontendAction koja omogućava izvršavanje akcija vezanih za pretprocesiranje izvornog koda. Međutim, najčešća upotreba ovog interfejsa vezana je za akcije koje se izvršavaju nad AST-om. U ovu svrhu koristi se apstarktna 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 sintaksnog stabla. Nad ovim apstraktnim sintaksnim stablom izvr-

šiće se akcije implementirane u `ASTConsumer` objektu pridruženom ovoj klasi. Na slici 3.3 prikazane su klase u okviru *Clang* kompajlera koje nasleđuju klasu `ASTFrontendAction`. Slika demonstrira raznolikost upotrebe ovog interfejsa.

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

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

Ova metoda za argumente dobija instancu kompajlera *Clang* i ime fajla za koji se kreira AST. Povratna vrednost metode je pokazivač na kreirani objekat klase `ASTConsumer`. Na listingu 3.7 prikazana je implementacija klase `AutoFixAction` koja izvršava akcije nad AST-om implementirane u okviru alata *AutoFix*.

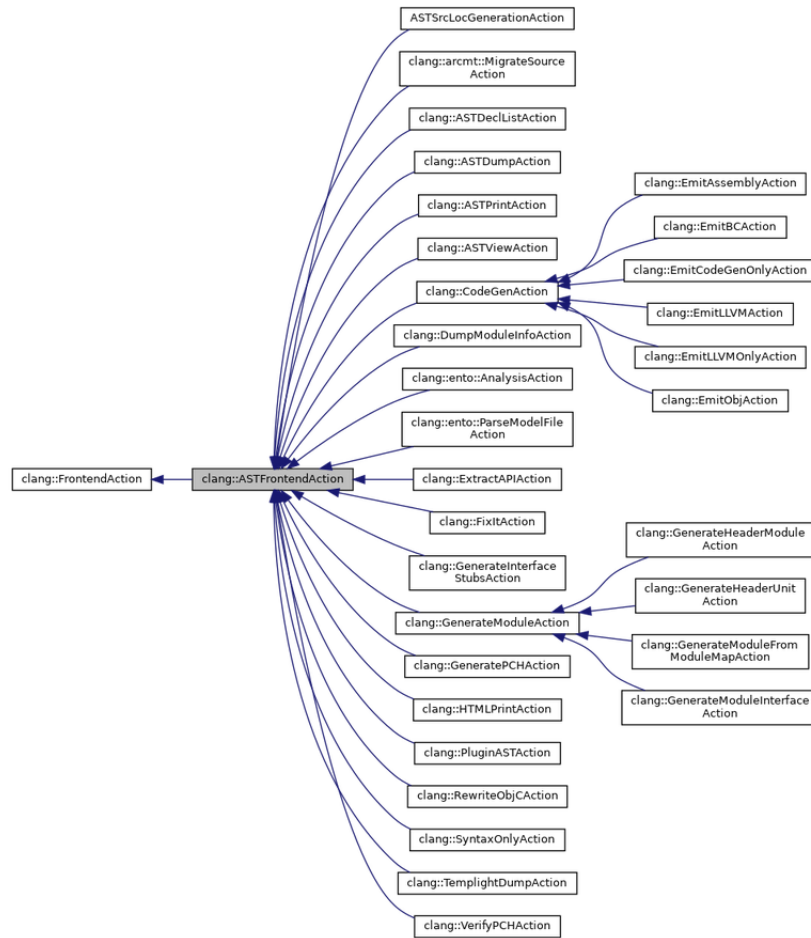
Listing 3.7: Implementacija klase `AutoFixAction` korišćene u okviru alata *AutoFix*.

```
1 | class AutoFixAction : public clang::ASTFrontendAction {
2 | public:
3 |     virtual std::unique_ptr<clang::ASTConsumer>
4 |     CreateASTConsumer(clang::CompilerInstance &Compiler, llvm::StringRef InFile) {
5 |         return std::make_unique<AutoFixConsumer>(&Compiler.getASTContext(),
6 |                                                  Compiler.getSourceManager());
7 |     }
8 | };
```

3.5 Biblioteke za kreiranje alata

Kompajlerska infrastruktura LLVM pruža podršku za jednostavno kreiranje kvalitetnih alata za statičku analizu izvornog koda. Ovi alati zasnivaju se na upotrebi interfejsa ka AST-u kompajlera *Clang* ili korišćenjem statičkog analizatora kompajlera *Clang* (eng. *Clang Static Analyzer*) za potrebe simboličkog izvršavanja programa.

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

Slika 3.3: Klase koje implementiraju `ASTFrontendAction` interfejs.

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

- **LibClang** je stabilni C interfejs visokog nivoa (eng. *high level*) ka kompajleru *Clang*. Ovaj interfejs pruža parsiranje izvornog koda i izgradnju AST-a, učitavanje već kreiranog AST-a, obilazak AST-a i dohvaćanje određenih informacija o izgrađenom AST-u kao što su lokacije iz izvornog koda elemenata iz stabla. Ovaj interfejs ne pruža sve informacije i detalje iz izgrađenog AST-a [9]. Ovo ga čini nepogodnim za implementaciju alata za statičku analizu ali omogućava stabilnost pri promeni verzija kompajlera *Clang*. Treba ga

koristiti u slučajevima kada:

- je potreban interfejs ka kompajleru *Clang* iz jezika koji nije C++.
- je potreban stabilni interfejs koji je kompatibilan sa starijim verzijama kompajlera *Clang*.
- su potrebne apstrakcije visokog nivoa kao što je iteriranje kroz AST sa kursorima ili drugi detalji vezani za AST.

LibClang ne treba koristiti kada je potrebna puna kontrola nad AST-om [2].

- Clang Plugins biblioteka omogućava izvršavanje dodatnih akcija nad AST-om tokom kompilacije programa. Ovo su dinamičke biblioteke koje kompajler učitava tokom izvršavanja i lako ih je integrisati u okruženje za prevođenje programa (eng. *build environment*).

Biblioteku Clang Plugins 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 AST-om.

Biblioteku Clang Plugins 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 [2].
- LibTooling je C++ interfejs koji služi za pisanje samostalnih alata. Ova biblioteka omogućava jednostavnu upotrebu opisanih akcija prednjeg dela kompajlera (eng. *frontend actions*), ali i jednostavno dodavanje opcija komandne linije i pokretanje nad fajlovima nezavisnim od sistema za prevođenje. Ova svojstva čine LibTooling biblioteku najkorisnijom od prethodno opisanih biblioteka u svrhu kreiranja alata za statičku analizu. 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 AST-om kompajlera *Clang*.
- je potrebno deliti kôd sa dodacima (eng. *plugins*) kompajlera *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 AST interfejs promeni.
- su potrebne apstrakcije visokog nivoa kao što su kursori.
- alat neće biti napisan u jeziku C++ [2].

Na listingu 3.8 prikazana je implementacija jednostavnog alata korišćenjem opisanih interfejsa `ASTConsumer` i `ASTFrontendAction`. Ovaj alat koristi `LibTooling` biblioteku za pokretanje definisane `ASTFrontendAction` akcije nad izvornim kodom koji je prosleđen kao argument komandne linije. U ovu svrhu koristi se funkcija `runToolOnCode` (linija 67) biblioteke `LibTooling`. Alat pronalazi sve enumeratore koji nisu deklarirani koristeći specifikator `class` i za ove deklaracije se prijavljuje upozorenje zajedno sa predlogom izmene koda. Alat koristi uparivač sa listinga 3.5.

Listing 3.8: Primer implementacije jednostavnog alata upotrebom interfejsa `ASTConsumer`, `ASTFrontendAction` i bibliotekom `libtooling`.

```
1 | #include "clang/AST/ASTConsumer.h"
2 | #include "clang/ASTMatchers/ASTMatchFinder.h"
3 | #include "clang/ASTMatchers/ASTMatchers.h"
4 | #include "clang/Frontend/CompilerInstance.h"
5 | #include "clang/Frontend/FrontendAction.h"
6 | #include "clang/Tooling/Tooling.h"
7 |
8 | using namespace clang;
9 |
10 | // Callback class for matcher.
11 | class A7_2_3 : public MatchFinder::MatchCallback {
12 | public:
13 |     A7_2_3(ASTContext &ASTCtx) : ASTCtx(ASTCtx) {}
14 |     virtual void run(const MatchFinder::MatchResult &Result) {
```

```
15     if (auto ED = Result.Nodes.getNodeAs<clang::EnumDecl>("A7_2_3_Matcher")) {
16         // Check if enum is defined using 'class' tag.
17         if (!ED->isScopedUsingClassTag()) {
18             // Create warning message.
19             std::string msg =
20                 "Enumerations shall be declared as scoped enum classes.";
21             std::string insStr = "class ";
22             // Call function for emitting warnings with fixit hints using diagnostics
23             // engine.
24             emitWarningWithHintInsertion(
25                 ASTCtx.getDiagnostics(), msg, insStr,
26                 ED->getSourceRange().getBegin().getLocWithOffset(5),
27                 ED->getLocation());
28         }
29     }
30
31 private:
32     ASTContext &ASTCtx;
33 };
34
35 class AutoFixConsumer : public clang::ASTConsumer {
36 public:
37     explicit AutoFixConsumer(ASTContext *Context) : Context(Context) {}
38
39     virtual void HandleTranslationUnit(clang::ASTContext &Context) {
40         MatchFinder Finder;
41         // Create matcher.
42         Matcher<Decl> Matcher =
43             enumDecl(isExpansionInMainFile()).bind("A7_2_3_Matcher");
44
45         // Create CallBack class object.
46         MatchCallback *Callback = new A7_2_3(Context);
47
48         // Register matcher for running over AST.
49         Finder.addMatcher(Matcher, Callback);
50
51         // Run matcher over AST for translation unit.
52         Finder.matchAST(Context);
53     }
54 };
55
56 class AutoFixAction : public clang::ASTFrontendAction {
57 public:
58     virtual std::unique_ptr<clang::ASTConsumer>
59     CreateASTConsumer(clang::CompilerInstance &Compiler, llvm::StringRef InFile) {
60         return std::make_unique<AutoFixConsumer>(&Compiler.getASTContext(),
61             Compiler.getSourceManager());
62     }
63 };
64
65 int main(int argc, char **argv) {
66     if (argc > 1) {
```



```
67 | clang::tooling::runToolOnCode(std::make_unique<FindNamedClassAction>(),
68 |                               argv[1]);
69 | }
70 | }
```

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 i zajedno sa upozorenjima ispisuje predlog koda kojim se početni kôd može zaminiti kako bi bio u skladu sa standardom. Alat je implementiran u programskom jeziku C++ korišćenjem biblioteka za razvoj alata dostupnim u okviru kompajlerske infrastrukture LLVM. Osnovna svrha alata je demonstracija kreiranja alata u okviru kompajlerske infrastrukture LLVM i predstavljanje tehnika obilaska i analize Clang-ovog AST-a. Alat je dostupan i nalazi se na linku <https://github.com/ognjen-plavsic/master/tree/main/code>. Na pomenutom linku se nalaze neophodne datoteke alata, opis sistema, kao i skup test primera i njihova pokretanja.

4.1 Korišćenje alata

Alat *AutoFix* se pokreće komandom:

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

Navedeni arumenti podrazumevaju sledeće:

- **options** - Ovde spadaju opcije koje se mogu proslediti alatu *AutoFix*. Podržane su opcije:

1. **-apply-fix**: Ovom opcijom se predložene izmene mogu primeniti na kod, menjajući 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.
2. **-list-rules**: Ispisuje sva podržana pravila u okviru alata u formatu `oznaka: tekst-pravila` gde je `oznaka` jedinstvena oznaka pravila iz AUTOSAR dokumenta, a `tekst-pravila` predstavlja kratak opis pravila iz AUTOSAR dokumenta koji se ujedno ispisuje prilikom prijavljivanja upozorenja vezanih za to pravilo. Primer:

```
A7-1-8 - A non-type specifier shall be placed before a type specifier
in a declaration.
```

3. **-rules=<string>**: 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. Primer korišćenja ove opcije:

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

4. **-help**: Ispisuje uputstvo za upotrebu alata.
- **<source0> [... <sourceN>]** - Predstavlja listu fajlova, razdvojenih razmakom, nad kojima će se pokrenuti alat.

4.2 Opis implementiranih pravila

Pored formalne klasifikacije, pravila u okviru samog dokumenta standarda AUTOSAR C++14 kodiranja strukturirana su po poglavljima. Struktura poglavlja ovog dokumenta slična je strukturi iz samog C++ standarda ISO/IEC 14882:2014. Sva-ko poglavlje odgovara jednoj komponenti (svojstvu) C++14 jezika, to jest, sadrži pravila koja se odnose na tu komponentu.

Pravila razmatrana u ovom radu predstavljaju podskup pravila koja se od-

nose na deklaracije. Razlog za ovo je dvojak. Deklaracije predstavljaju jedan od osnovnih i najvažnijih koncepta u C++-u i programiranju generalno. U C++-u deklaracije čine samu srž ekspresivne moći jezika i u direktnoj su vezi sa naprednijim konceptima jezika i računarstva, kao što je, na primer, šablonsko metaprogramiranje (*eng. template metaprogramming*). Sa druge strane jednostavnost sintakse deklaracija u C++-u čini pogodno tlo za korišćenje kompajlerskih tehnika i struktura u okviru kompajlera *Clang* kojim se mogu analizirati konstrukti jezika koji nisu u skladu sa pravilima i predlagati prikladne alternative.

Sva implementirana pravila u okviru projekta spadaju, prema klasifikaciji iz prethodnog poglavlja, u sledeće kategorije:

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

Razmatrana pravila nisu nužno implementirana u potpunosti u okviru alata Autofix, iako činjenica da pravila spadaju u kategorije obaveznih i automatizovanih implicira da je to teorijski moguće uraditi. Pravila koje Autofix podržava birana su tako da se ograničenja koja potiču iz same prirode projekta minimalno manifestuju. Ograničenja potiču od primarnih tehnologija i biblioteka kojima je alat implementiran ali i činjenice da se alat zasniva na predlogu izmena koda. statički analizator kompajlera *Clang* [3] (*eng. Clang Static Analyzer*) nije korišćen u okviru ovog alata, tako da su pravila izabrana tako da što manji broj slučajeva upotrebe zahteva simboličko izvršavanje programa. Drugo ograničenje potiče iz činjenice da u nekim slučajevima nije moguće ili je znatno komplikovanije kreirati predlog ispravke koda (*eng. fixit hint*). Pravila razmatrana u okviru ovog rada birana su tako da se većina konstrukta koji nisu u saglasnosti sa pravilom mogu detektovati analizom Clang-ovog AST-a i da se za njih mogu kreirati razumne alternative koje su u skladu sa standardom AUTOSAR C++14.

4.3 Primeri rada alata

U ovoj sekciji biće navedena sva pravila iz AUTOSAR dokumenta koje alat *AutoFix* podržava i biće prikazani primeri rada alata za svako od tih pravila. Primeri će sadržati izvorni kôd programa nad kojim se alat pokreće i ispis na standardnom

izlazu koji predstavlja rezultat izvršavanja alata. U primeru rada alata pri zadanju opcije `--apply-fix` biće prikazan i kôd nakon završetka rada alata, odnosno kôd za primenjim predlozima izmena.

Primer 4.3.1 *Pravilo A7-1-8 (obvezno, implementaciono, automatizovano)*

U deklaracijama specifikatori koji nisu vezani za tipove treba da stoje ispred tipskih specifikatora.

Razmotrimo kôd sa listinga 4.1. Deklaracija metoda `f3` sadrži tri specifikatora: `void`, `virtual` i `inline`. Specifikator `void` označava „prazan” tip, odnosno da metoda `f3` nema povratnu vrednost. Stoga, `void` spada u tipske specifikatore. Specifikator `virtual` omogućava dinamički polimorfizam dok `inline` predlaže kompajleru da kôd ove funkcije umetne u funkciju iz koje je pozvana kako bi se izbeglo dodatno vreme potrebno za pozivanje funkcije. Dakle `virtual` i `inline` se ne odnose na tipove i nisu tipski specifikatori. Pravilo A7-1-8 nalaže da se specifikator `void` u deklaraciji nađe nakon specifikatora `virtual` i `inline`, odnosno da metod `f3` bude deklarisan kao metod `f1` ili metod `f2`. slično važi i za deklaraciju promenljive `x` i specifikatore `std::int32_t` i `mutable`.

Listing 4.1: Primer koda koji nije napisan u skladu sa pravilom A7-1-8.

```
1 | #include <cstdint>
2 |
3 | class C {
4 | public:
5 |     virtual inline void f1();
6 |     inline virtual void f2();
7 |     void virtual inline f3();
8 |
9 | private:
10 |     std::int32_t mutable x;
11 |     mutable std::int32_t y;
12 | };
```

Primer 4.3.2 *Pravilo A8-5-3 (obvezno, implementaciono, automatizovano)*

Varijabla tipa `auto` ne sme biti inicijalizovana korišćenjem inicijalizacijom vitičastih zagrada tipa `{}` ili `={}.`

```
ognjen@ognjen-Precision-7710:~/Desktop/llvm-project/build$ bin/auto-fix /home/ognjen/Desktop/AutoFixTest.cpp -rules="A7_1_8" --
/home/ognjen/Desktop/AutoFixTest.cpp:7:23: warning: A non-type specifier shall be placed before a type specifier in a declaration
void virtual inline f3();
~~~~~~^~~~~~
/home/ognjen/Desktop/AutoFixTest.cpp:10:24: warning: A non-type specifier shall be placed before a type specifier in a declaration
std::int32_t mutable x;
~~~~~~^~~~~~
mutable std::int32_t x
2 warnings generated.
```

Slika 4.1: Ispis alata za pravilo A7-1-8

Po standardu C++14 kompajler će auto promenljivu inicijalizovanu sinaksom vitičastih zagrada tretirati kao objekat klase `std::initializer_list`. Međutim neki kompajleri mogu implementirati ovo drugačije i zaključiti tip `int` za objekat inicijalizovan sintaksom `{}`, dok će za sintaksu `={}` zaključiti tip `std::initializer_list`. Da bi se izbegla konfuzija oko zaključivanja tipova, AUTOSAR standard nalaže da se ne koristi nijedna od navede dve vrste inicijalizacije. Za kôd sa listinga 4.2 *AutoFix* će predložiti inicijalizaciju simbolom `=`. Ispis alata prikazan je na slici 4.2.

Listing 4.2: Primer koda koji nije napisan u skladu sa pravilom A8-5-3.

```
1 | #include <initializer_list>
2 |
3 | void fn() noexcept {
4 |     auto x1(10);
5 |     auto x2{10};
6 |     auto x3 = 10;
7 |     auto x4 = {10};
8 | }
```

```
ognjen@ognjen-Precision-7710:~/Desktop/llvm-project/build$ bin/auto-fix /home/ognjen/Desktop/AutoFixTest.cpp -rules="A8_5_3" --
/home/ognjen/Desktop/AutoFixTest.cpp:5:8: warning: A variable of type auto shall not be initialized using {} or ={} braced initialization
auto x2{10};
~~~~~~^~~~~~
auto x2 = 10
/home/ognjen/Desktop/AutoFixTest.cpp:7:8: warning: A variable of type auto shall not be initialized using {} or ={} braced initialization
auto x4 = {10};
~~~~~~^~~~~~
auto x4 = 10
2 warnings generated.
```

Slika 4.2: Ispis alata za pravilo A8-5-3

Primer 4.3.3 Pravilo A8-5-2 (obvezno, implementaciono, automatizovano)

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

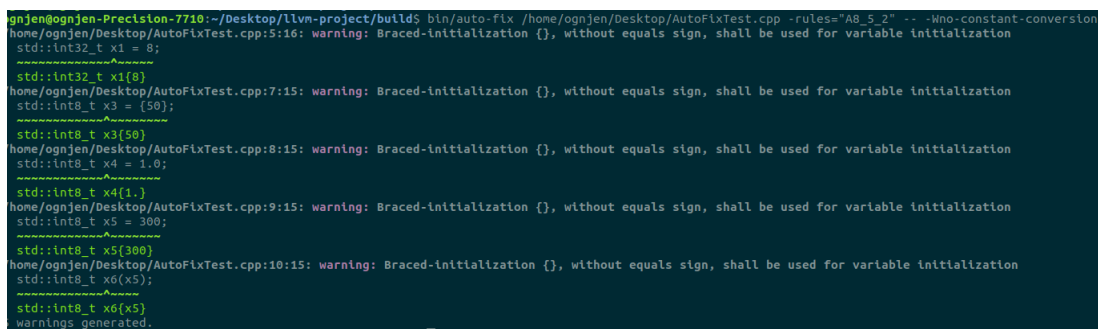
AUTOSAR zahteva upotrebu ove inicijalizacije kako bise izbegla dvosmislenost u kodu. Na primer, upotreba znaka jednako (=) pri inicijalizaciji navodi programere na pomisao da dolazi do dodeljivanja vrednosti iako se zapravo vrši kreiranje i inicijalizacija objekta. Takođe ukoliko se koristi predožena sintaksa neće doći do konverzija tipova iz tipa veće bitske širine u tip manje bitske širine (eng. *narrowing conversions*), što je ujedno čini i sigurnijom od ostalih vrsta inicijalizacija. Ispis alata pokrenutim nad fajlom sa kodom iz listinga 4.3 prikazan je na slici 4.3.

Listing 4.3: Primer koda koji nije napisan u skladu sa pravilom A8-5-2.

```

1  #include <cstdint>
2  #include <initializer_list>
3
4  void f1() noexcept {
5      std::int32_t x1 = 8;
6      std::int8_t x2{50};
7      std::int8_t x3 = {50};
8      std::int8_t x4 = 1.0;
9      std::int8_t x5 = 300;
10     std::int8_t x6(x5);
11 }

```



```

ognjen@ognjen-Precision-7710:~/Desktop/llvm-project/build$ bin/auto-flx /home/ognjen/Desktop/AutoFlxTest.cpp -rules="A8_5_2" -- -Wno-constant-conversion
/home/ognjen/Desktop/AutoFlxTest.cpp:5:16: warning: Braced-initialization {}, without equals sign, shall be used for variable initialization
std::int32_t x1 = 8;
               ^
/home/ognjen/Desktop/AutoFlxTest.cpp:6:15: warning: Braced-initialization {}, without equals sign, shall be used for variable initialization
std::int8_t x2{50};
               ^
/home/ognjen/Desktop/AutoFlxTest.cpp:7:15: warning: Braced-initialization {}, without equals sign, shall be used for variable initialization
std::int8_t x3{50};
               ^
/home/ognjen/Desktop/AutoFlxTest.cpp:8:15: warning: Braced-initialization {}, without equals sign, shall be used for variable initialization
std::int8_t x4 = 1.0;
               ^
/home/ognjen/Desktop/AutoFlxTest.cpp:9:15: warning: Braced-initialization {}, without equals sign, shall be used for variable initialization
std::int8_t x5 = 300;
               ^
/home/ognjen/Desktop/AutoFlxTest.cpp:10:15: warning: Braced-initialization {}, without equals sign, shall be used for variable initialization
std::int8_t x6(x5);
               ^
std::int8_t x6(x5)
warnings generated.

```

Slika 4.3: Ispis alata za pravilo A8-5-2

Primer 4.3.4 Pravilo A7-1-6 (obvezno, implementaciono, automatizovano)

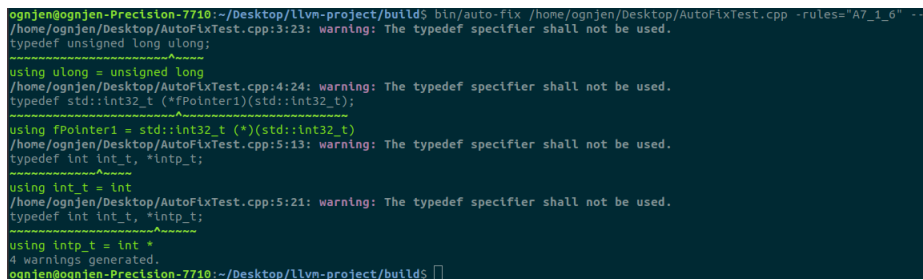
Ne treba koristiti specifikator *typedef*.

Specifikator *typedef* nije pogodan za kreiranje pseudonima (eng. *alias*) za šablonske tipove ali i čini kôd manje čitljivim. Oba nedostatka mogu se zaobići

korišćenjem specifikatora `using`. Alat *AutoFix* od izraza za kreiranje pseudonima za tip korišćenjem specifikatora `typedef` kreira i ispisuje analogni izraz koji koristi `using` sintaksu. Ispis alata pokrenutim nad fajlom sa kodom iz listinga 4.4 prikazan je na slici 4.4.

Listing 4.4: Primer koda koji nije napisan u skladu sa pravilom A7-1-6.

```
1 | #include <cstdint>
2 |
3 | typedef unsigned long ulong;
4 | typedef std::int32_t (*fPointer1)(std::int32_t);
5 | typedef int int_t, *intp_t;
```



```
ognjen@ognjen-Precision-7710:~/Desktop/llvn-project/builds$ bin/auto-fix /home/ognjen/Desktop/AutoFixTest.cpp -rules="A7_1_6" --
/home/ognjen/Desktop/AutoFixTest.cpp:3:23: warning: The typedef specifier shall not be used.
typedef unsigned long ulong;
~~~~~
using ulong = unsigned long
/home/ognjen/Desktop/AutoFixTest.cpp:4:24: 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/Desktop/AutoFixTest.cpp:5:13: warning: The typedef specifier shall not be used.
typedef int int_t, *intp_t;
~~~~~
using int_t = int
/home/ognjen/Desktop/AutoFixTest.cpp:5:21: warning: The typedef specifier shall not be used.
typedef int int_t, *intp_t;
~~~~~
using intp_t = int *
4 warnings generated.
ognjen@ognjen-Precision-7710:~/Desktop/llvn-project/builds$
```

Slika 4.4: Ispis alata za pravilo A7-1-6

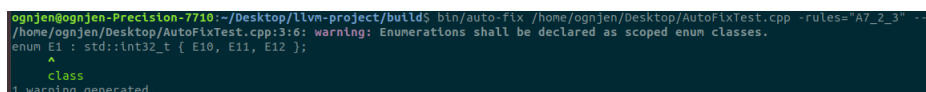
Primer 4.3.5 *Pravilo A7-2-3 (obvezno, implementaciono, automatizovano)*

Nabrajanja (eng. enumerators) treba deklarirati kao nabranja sa opsegom odnosno treba koristiti `scoped enum class` sintaksu.

Ukoliko je nabranje bez opsega deklarirano u globalnom opsegu, onda njegove vrednosti mogu ponovo deklarirati konstante koje su deklarirane sa istim identifikatorom u globalnom opsegu. Korišćenjem nabranja sa opsegom, odnosno upotrebom `scoped enum class` sintakse, identifikatori korišćeni prilikom nabranja biće deklarirani u svom unutrašnjem opsegu i time sprečiti ponovno deklarisanje identifikatora iz spoljašnjeg opsega. Ispis alata *AutoFix* pokrenutim nad fajlom sa kodom iz listinga 4.5 prikazan je na slici 4.5.

Listing 4.5: Primer koda koji nije napisan u skladu sa pravilom A7-2-3.

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



```
ognjen@ognjen-Precision-7710:~/Desktop/llvm-project/build$ bin/auto-fix /home/ognjen/Desktop/AutoFixTest.cpp -rules="A7_2_3" --
/home/ognjen/Desktop/AutoFixTest.cpp:3:6: 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

4.4 Opis implementacije

AutoFix implementiran je u okviru `clang-tools-extra` projekta koji sadrži alate implementirane interfejsima za alate kompajlera *Clang* (eng. *Clang's tooling APIs*). Alat 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 AST-a kompajlera *Clang*. Svakom pravilu koje podržava alat *AutoFix* odgovara jedan uparivač (instanca klase `Matcher`). *AutoFix* koristi ovu biblioteku na sledeći način. U prvom delu definišu se jednostavni uparivači koji uparuju osnovne strukture iz AST-a na koje se pravilo odnosi. Na primer, ukoliko je pravilo vezano za enumeratore, ovaj uparivač će upariti sve deklaracije enumeratora iz AST-a. Drugi deo implementiran je upotrebom klase `MatchCallback`. Svaka potklasa ove klase implementirana u okviru alata *AutoFix* odgovara jednom uparivaču, a samim tim i jednom pravilu. U okviru `run` metode ove klase vrše se dodatne analize nad konstruktima AST-a koji su dobijeni rezultatom rada uparivača i prijavljuje odgovarajuća dijagnostika za izvorni kod. Zajedno sa dodatnom analizom u okviru `run` metode vrši se i kreiranje stringova koji predstavljaju predloge izmena izvornog koda (eng. *fixit hints*) kako bi kôd bio u skladu sa pravilom.

AutoFix.cpp

Ova jedinica prevođenja predstavlja ulaznu tačku samog alata. U okviru nje,

implementirane su klase `AutoFixConsumer` i `AutoFixAction` koje nasleđuju redom klase `ASTConsumer` i `ASTFrontendAction`. Uloga pomenutih baznih klasa pri kreiranju alata opisana je u sekciji ***. Osnovna uloga klase `AutoFixConsumer` jeste da obezbedi da se nad jedinicom prevođenja pokrenu odgovarajući uparivači. To su uparivači koji odgovaraju pravilima za koje korisnik želi da proveri da li je izvorni kôd napisan u skladu sa njima. Korisnik može da zada podskup ovih pravila u okviru argumenta komandne linije `-rules`. Parsiranje ove opcije i samo pokretanje uparivača nad AST-om implementirano je u okviru metode `HandleTranslationUnit` klase `AutoFixConsumer`. Ova metoda biće pozvana od strane parsera na kraju izgradnje AST-a za svaku jedinicu prevođenja nad kojom je pokrenut alat.

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

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 automobilske 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 alati 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 kompajlerska infrastruktura LLVM. Pravila standarda AUTOSAR C++14 podržana u okviru alata *AutoFix* su **A7-1-8**, **A8-5-3**, **A8-5-2**, **A7-1-6**, **A7-2-3**.

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 kompajlera *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 delovi-

ma koda. Alat *AutoFix* je testiran upotrebom `FileCheck` alata iz kompajlerske infrastrukture LLVM.

Literatura

- [1] AST Matcher Reference. <https://clang.llvm.org/docs/LibASTMatchersReference.html>.
- [2] Choosing the Right Interface for Your Application. <https://clang.llvm.org/docs/Tooling.html>.
- [3] Clang Static Analyzer website. <https://clang-analyzer.llvm.org/>.
- [4] clang::ASTConsumer Class Reference. https://clang.llvm.org/doxygen/classclang_1_1ASTConsumer.html.
- [5] clang::FrontendAction Class Reference. https://clang.llvm.org/doxygen/classclang_1_1FrontendAction.html.
- [6] clang::RecursiveASTVisitor<Derived> Class Template Reference. https://clang.llvm.org/doxygen/classclang_1_1RecursiveASTVisitor.html.
- [7] How to write RecursiveASTVisitor based ASTFrontendActions. <https://clang.llvm.org/docs/RAVFrontendAction.html>.
- [8] ISO official website. <https://www.iso.org/committee/45202.html>.
- [9] libclang: C Interface to Clang. https://clang.llvm.org/doxygen/group__CINDEX.html.
- [10] “Clang” CFE Internals Manual. <https://clang.llvm.org/docs/InternalsManual.html>.
- [11] AUTOSAR. Guidelines for the use of the C++14 language in critical and safety-related systems, 2017.
- [12] AUTOSAR. AUTOSAR official website, 2018.

- [13] Bruno Cardoso Lopes. *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014.
- [14] Bjarne Stroustrup. *The C++ Programming Language*. Addison–Wesley, fourth edition, 2013.

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 prosekom 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 kompajlerske infrastrukture LLVM. Jula 2021. godine prelazi u kompaniju HTEC Group gde i danas radi kao softverski inženjer. Trenutno se bavi kompajlerima za mašinsko učenje (eng. *machine learning (ML) compilers*). U okviru ovih kompajlera 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 konferenciji *ZINC*.

Radovi:

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)*