

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 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	Statička analiza u okviru kompajlerske infrastrukture LLVM	10
3.1	LLVM i Clang	11
3.2	Biblioteka clangAST	12
3.3	AST-uparivači	18
3.4	Interfejsi za akcije nad prednjim delom kompajlera	21
3.5	Interfejsi za kreiranje alata	25
3.6	Alati za testiranje	29
4	Alat Autofix	31
4.1	Korišćenje alata	31
4.2	Opis implementiranih pravila	33
4.3	Opis implementacije alata	38
4.4	Opis testiranja alata	42
4.5	Analiza rezultata rada alata	43
5	Zaključak	46
	Literatura	48

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 [21, 18].

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 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 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*. 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 [17]. 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 [16].

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 [20].

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 [11]. 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 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 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 [16].

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 dà 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* [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 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 zaglavlja mora biti identično imenu tipa deklarisanog u njemu ukoliko deklariše 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 [16]. 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

Statička analiza u okviru kompajlerske infrastrukture 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.

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 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 sintaksnog stabla.

Apstraktne klase `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 korišćenja ove biblioteke, alati se mogu kreirati i upotrebom dodataka kompajlera *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

Kompajlerska infrastruktura LLVM predstavlja kolekciju modularnih i ponovo iskoristivih kompajlerskih tehnologija i alata. Ova kompajlerska infrastruktura započeta je kao instraživački 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 assemblera 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 spuštanja (eng. *recursive-descent parser*), odnosno analizira izvorni kôd od vrha ka dnu nizom rekurzivnih funkcija [19].

Biblioteka `clangAST` implementira algoritme i strukture podataka koje parser koristi za izgradnju apstraktnog sintaksnog stabla. Specifična je po strukturi čvorova koji podsećaju na izvorni C++ kôd što je čini pogodnom za kreiranje alata za refaktorisane 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 sintaksnog stabla nakon parsiranja, biblioteka `clangSema` implementira proveru tipova zajedno sa generisanjem čvorova apstraktnog sintaksnog stabla [19]. Ova biblioteka usko je povezana sa bibliotekama `clangParse` i `clangAST`.

Biblioteka `clangCodeGen` generiše LLVM međukod. Ova biblioteka obilazi apstraktno sintakšno stablo i na osnovu njegovog sadržaja generiše instrukcije LLVM međukoda koje implementiraju ponašanje opisano u stablu [19].

3.2 Biblioteka `clangAST`

U računarstvu, **apstraktno sintakšno stablo**, ili samo **sintakšno 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

GLAVA 3. STATIČKA ANALIZA U OKVIRU KOMPAJLERSKE INFRASTRUKTURE LLVM

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

Listing 3.1: Kôd čije je apstraktno sintakšno 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 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'
- CXRecord 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'
- CXRecord 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: Apstraktno sintakšno stablo za kôd iz listinga 3.1 koje je generisano komandom: `clang -Xclang -ast-dump hello.c`.

Čvorovi od kojih je izgrađeno apstraktno sintakšno stablo predstavljaju apstrakciju sintaksnih struktura iz samog jezika. Svi čvorovi apstraktnog sintaksnog stabla kompajlera *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, `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.

Čvorovi apstraktnog sintaksnog 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 sintaksnog stabla. 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, ... >`. 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 3.2 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 `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 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 [15].

AST-posetioci

AST-posetioci (eng. *AST-visitors*) implementiraju mehanizam obilaska apstraktnog sintaksnog stabla kompajlera *Clang*, odnosno pružaju interfejs¹ 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. 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

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

u okviru apstraktnog sintaksnog stabla. Ovo su ulazne tačke za obilazak apstraktnog sintaksnog 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 predefinisano 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 [7]. 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 `Visit` me-

GLAVA 3. STATIČKA ANALIZA U OKVIRU KOMPAJLERSKE INFRASTRUKTURE LLVM

tode u okviru nje. Ukoliko je `Visit` metodama 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 sintaksnom stablu. U okviru nje, proverava se da li deklaracija koristi `enum class` sintaksu, odnosno da li su konstante u okviru nabiranja 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 nabiranja i ispisuje lokaciju onih koji nisu deklarirani sintaksom `enum class`.

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

3.3 AST-uparivači

Biblioteka AST-uprarivača (eng. *LibASTMatchers*) implementira jezik specijalne namene za kreiranje predikata nad apstraktnim sintaksnim stablom 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 nivou apstraktnog sintaksnog stabla.

Uparivač je objekat šablonske klase `template <typename T> class Matcher`. Svaki uparivač obilazi stablo na specifičan način koji je opisan prilikom kreiranja uparivača. Uparivači se kreiraju izrazima za uparivanje (eng. *match expressions*). Ovi izrazi sastoje se od niza poziva funkcija koje su deo jezika specijalne namene implementiranog u okviru biblioteke *LibASTMatchers*. Rezultat izaraza za uparivanje je uparivač, odnosno objekat klase `template <typename T> class Matcher`.

Uopšteno, strategija kreiranja uparivača, odnosno pisanja izraza za uparivanje, svodi se na sledeće korake:

1. Naći baznu klasu čvora apstraktnog sintaksnog stabla 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 [14].

Na primer, za kreiranje uparivača koji uparuje sve deklaracije enumeratora iz apstraktnog sintaksnog stabla 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.

GLAVA 3. STATIČKA ANALIZA U OKVIRU KOMPAJLERSKE INFRASTRUKTURE LLVM

Nakon uparivanja, nad uparenim 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 (rezultata svakog od uparivača u kompoziciji) zasebno. Na primer, uparivač može uparivati deklaracije enumeratora, ali samo one koje definišu i neku konstantu u okviru nabiranja, odnosno deklaracije koje imaju potomka u stablu tipa `EnumConstantDecl`. U ovom slučaju zgodno je da se u okviru analize uparenog čvora, odnosno deklaracije enumeratora, može direktno analizirati njegov potomak, konstanta u okviru nabiranja, bez potrebe da se ovaj potomak ponovo traži u apstraktnom sintaksnom stablu.

Zbog toga, uparivači se mogu „vezati” (eng. *binding*) za određeni string. Na primer, izraz

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

će vezati uparene deklaracije enumeratora za string „EnumNode”, dok će konstante vezati za string `EnumConstNode`. Rezultati uparivanja predstavljeni su kao objekti klase `MatchResult`. Čvor koji predstavlja deklaraciju enumeratora može se dobiti izrazom

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

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

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

Nakon formulisanja izraza za uparivanje kreirani uparivač se pokreće nad apstraktnim sintaksnom stablom. 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 klasu `MatchCallback` 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.

Primer implementacije AST-uparivača

Na listingu 3.4 prikazan je primer uparivača koji pronalazi sve deklaracije enumeratora koje ne koriste sintaksu `enum 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 3.4: Primer uparivača koji pronalazi sve deklaracije enumeratora koje ne koriste sintaksu `enum class`. Ovaj primer demonstrira i upotrebu klasa `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>("
13         A7_2_3_Matcher")) {
14         // Check if declaration contains 'class' tag.
15         if (!ED->isScopedUsingClassTag()) {
16             // Create warning string.
17             std::string msg =
18                 "Enumerations shall be declared as scoped enum classes.";
19             std::string insStr = "class ";
20             // Function for emitting warnings with fixit hints using
21                 diagnostics engine.
22             emitWarningWithHintInsertion(
23                 ASTCtx.getDiagnostics(), msg, insStr,
24                 ED->getSourceRange().getBegin().getLocWithOffset(5),
25                 ED->getLocation());

```

```
24     }
25 }
26 }
27
28 void matchASTExample(ASTContext *Context){
29     MatchFinder Finder;
30     // Create matcher.
31     Matcher<Decl> Matcher = enumDecl(isExpansionInMainFile()).bind("
32         A7_2_3_Matcher");
33     // Create callback class.
34     MatchCallback *Callback = new A7_2_3(Context);
35     // Register matcher.
36     Finder.addMatcher(Matcher, Callback);
37     // Run matcher over AST.
38     Finder.matchAST(Context);
39 }
```

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 i grafičko prezentovanje rezultata kompajlera. Takodje, 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 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 sintakšno stablo kreirano. Akcije se mogu izvršavati u različitim fazama tokom kreiranja apstraktnog sintaksnog stabla [10]. Na primer, metod

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

biće pozvan svaki put kada se završi kreiranje umetnutih (eng. *inline*) funkcija

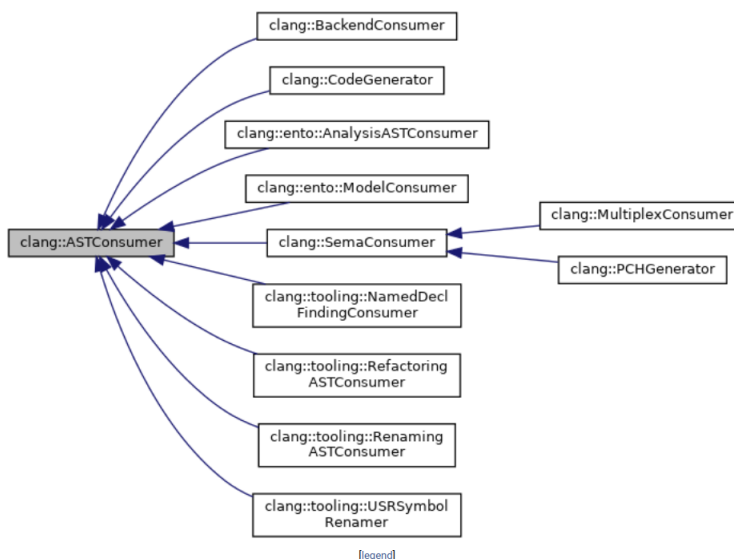
GLAVA 3. STATIČKA ANALIZA U OKVIRU KOMPAJLERSKE INFRASTRUKTURE LLVM

prilikom kreiranja apstraktnog sintaksnog stabla. `ASTConsumer` definiše niz sličnih virtuelnih metoda koje mogu biti predefinisane od strane klasa koje je nasleđuju.

Na slici 3.2 prikazane su klase u okviru kompajlera *Clang* 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 klase `ASTConsumer`.

```
#include "clang/AST/ASTConsumer.h"
```

Inheritance diagram for clang::ASTConsumer:



Slika 3.2: Klase koje nasleđuju klasu `ASTConsumer`.

Ova klasa korisna je i za kreiranje samostalnih alata za statičku analizu koji se zasnivaju na analizi apstraktnog sintaksnog stabla. U ovu svrhu može se koristiti kombinacija upotrebe klase `ASTConsumer` 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 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 sintakšno stablo za jedinicu prevođenja, odnosno u trenutku kada je celokupno apstraktno sintakšno stablo za jedinicu prevođenja dostupano. U okviru njega, nad apstraktnim sin-

taksnim stablom, može se pozvati AST-uparivač koji će izvršiti obilazak i analizu apstraktnog sintaksnog stabla [6].

Primer upotrebe klase `ASTConsumer`

Na listingu 3.5 prikazana je implementacija klase `AutoFixConsumer`. U okviru metode `HandleTranslationUnit` poziva se funkcija za kreiranje i pokretanje uparivača `matchASTExample` (linija 9), prikazana na listingu 3.4. Ova funkcija dostupna je kroz zaglavlje `AutoFixMatchers.h`.

Listing 3.5: Implementacija klase `AutoFixConsumer`.

```
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 |
9 |     virtual void HandleTranslationUnit(clang::ASTContext &Context) {
10 |         matchASTExample(Context);
11 |     }
12 |
13 |     ASTContext *Context;
```

Klasa `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 ove klase vezana je za akcije koje se izvršavaju nad apstraktnim sintaksnom 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 sintaksnog stabla. Nad ovim apstraktnim sintaksnim stablom izvršiće se akcije implementirane u objektu `ASTConsumer` pridruženom ovoj klasi. Na slici 3.3 prikazane su klase u okviru kompajlera *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 `ASTFrontendAction` i dodeliti joj `ASTConsumer` objekat koji implementira željenu akciju. Objekat se kreira i dodeljuje predefinisanjem metode

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

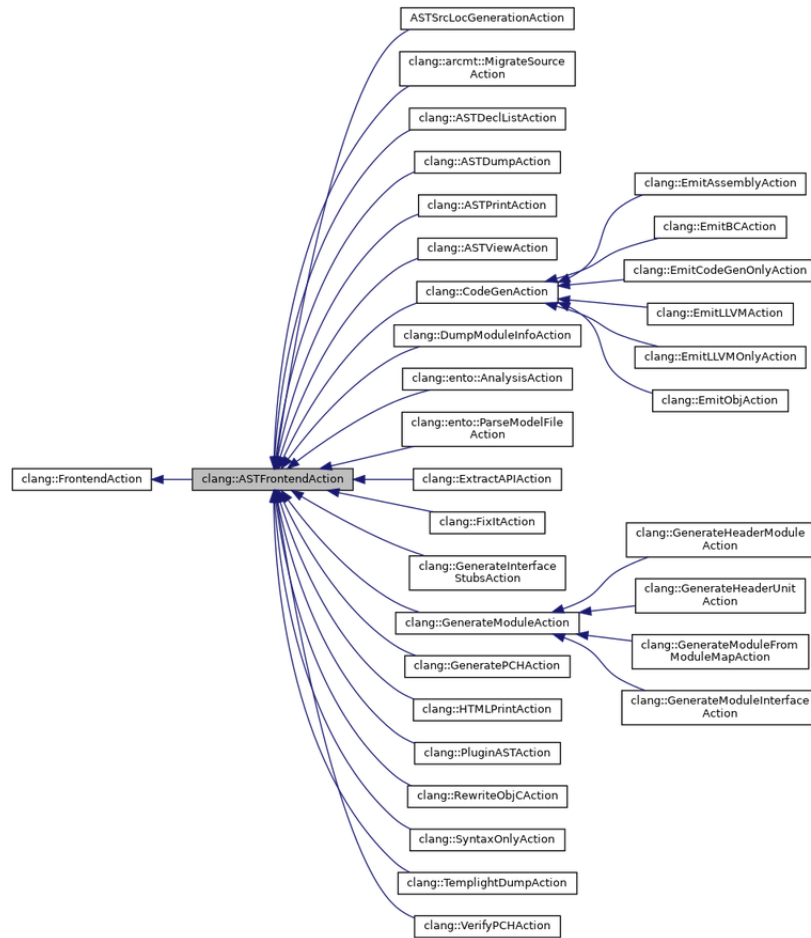
Ova metoda kao argumente dobija instancu kompajlera *Clang* i ime fajla za koji se kreira apstraktno sintakšno stablo. Povratna vrednost metode je pokazivač na kreirani objekat klase `ASTConsumer`.

Primer upotrebe klase `ASTFrontendAction`

Na listingu 3.6 prikazana je implementacija klase `AutoFixAction` koja izvršava akcije nad apstraktnim sintaksnom stablom. Klasa `AutoFixAction` u okviru metode `CreateASTConsumer` (linija 6) kreira pokazivač na objekat klase `AutoFixConsumer`. Klasa `AutoFixConsumer` prikazana je na listingu 3.5 i dostupna je kroz zaglavlje `AutoFixConsumer.h`

Listing 3.6: Implementacija klase `AutoFixAction`.

```
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 3.3: Klase koje nasleđuju klasu ASTFrontendAction.

3.5 Interfejsi 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 apstraktnom sintaksnom stablu 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 apstraktnog sintaksnog stabla i simboličkog izvršavanja programa u zavisnosti od kompleksnosti analize koja je potrebna. Implementacija statičke analize obradom apstraktnog sintaksnog stabla je jeftinija po pitanju računarskih resursa ali je ograničena informacijama dostupnim tokom kompilacije programa.

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 interfejsa 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 izagradnju apstraktnog sintaksnog stabla, učitavanje i obilazak već kreiranog apstraktnog sintaksnog stabla i dohvaćanje određenih informacija o izgrađenom apstraktnom sintaksnom stablu kao što su lokacije iz izvornog koda elemenata iz stabla. Ovaj interfejs ne pruža sve informacije i detalje iz izgrađenog apstraktnog sintaksnog stabla [12]. 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 apstraktno sintakšno stablo sa kursorima ili drugi detalji vezani za AST.

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

Dodaci kompajlera *Clang* omogućavaju izvršavanje dodatnih akcija nad apstraktnim sintaksnom stablom tokom kompilacije programa. Ovo su dinamičke biblioteke koje kompajler učitava tokom izvršavanja i lako ih je integrisati u okruženje za prevođenje programa (eng. *build enviroment*) [4].

Dodatke kompajlera *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 sintaksnom stablom.

Dodatke kompajlera *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 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. 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 sintaksnom stablom 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 interfejs apstraktnog sintaksnog 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. Fleksibilni alati za statičku analizu se ne moraju nužno pokretati u okviru sistema za prevođenje i podržavaju mogućnost analize fajlova nezavisnih 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 kompajlerske infrastrukture LLVM.

Primer kreiranja alata upotrebom biblioteke LibTooling

Na listingu 3.7 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 enumeratore koji nisu deklarirani sintaksom `enum class` i za ove deklaracije se prijavljuje upozorenje zajedno sa predlogom izmene koda. Alat koristi klasu `AutoFixAction` sa listinga 3.6 dostupnom kroz zaglavlje `ASTFrontendAction.h`.

Listing 3.7: Primer implementacije jednostavnog alata upotrebom biblioteke LibTooling.

```
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

Samostalni alati koji su razvijeni 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 provedenja 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 kompajler *Clang* neće moći da izgradi apstraktno sintaksno stablo, a samim tim će se prekinuti i izvršavanje alata.

Kompilaciona baza podataka kreira se na osnovu `compile_commands.json` fajla koji se generiše alatom CMake [9]. 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.

3.6 Alati za testiranje

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

Alat *lit* služi za izvršavanje testova i testnih paketa (eng. *test suites*) u okviru kompajlerske 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 [13].

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 proverí ispravnost sadržaja drugog. Ovaj alat je

koristan za kreiranje testova u okviru kojih je potrebno proveriti da li izlaz nekog alata sadrži očekivane informacije [8]. 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 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 apstraktnog sintaksnog stabla kompajlera *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:

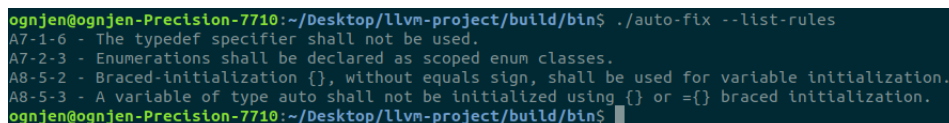
```
auto-fix [options] <source0> [... <sourceN>]
```

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 predstavlja 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:~/Desktop/llvm-project/build/bin$ ./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 initialization.
ognjen@ognjen-Precision-7710:~/Desktop/llvm-project/build/bin$
```

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 kompajleru *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 kompajler *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 [16] 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 sintaksnog stabla kompajlera *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**.

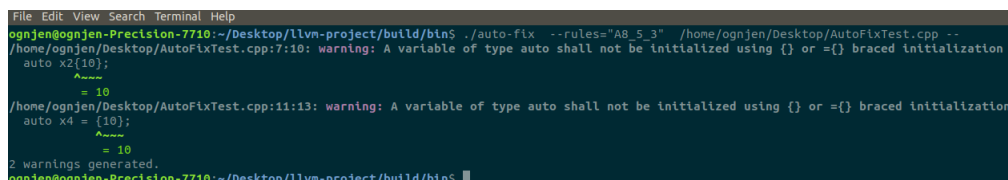
Pravilo A8-5-3

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

Po standardu C++14, kompajler 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 deklarisanu 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 }
```



The screenshot shows the output of the AutoFix tool. It displays two warnings: one for line 7 where `auto x2{10};` is used, and another for line 11 where `auto x4 = {10};` is used. Both warnings state: "A variable of type auto shall not be initialized using {} or ={} braced initialization". The output also shows the corrected code for these lines: `auto x2 = 10;` and `auto x4 = 10;`. At the bottom, it says "2 warnings generated."

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

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 širine (eng. *narrowing conversions*) što se može 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 skladu 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 <cstdlib>
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 | }
```

Pravilo A7-1-6

Ne treba koristiti specifikator typedef.


```

File Edit View Search Terminal Help
ognjen@ognjen-Precision-7710:~/Desktop/llvm-project/build/bin$ ./auto-fix --rules="A8_5_2" /home/ognjen/Desktop/AutoFixTest.cpp --
/home/ognjen/Desktop/AutoFixTest.cpp:6:18: warning: Braced-initialization {}, without equals sign, shall be used for variable initialization
std::int32_t x1 = 8;
               ^
               {8}
/home/ognjen/Desktop/AutoFixTest.cpp:8:17: warning: Braced-initialization {}, without equals sign, shall be used for variable initialization
std::int8_t x2{x1};
               ^
               {x1}
/home/ognjen/Desktop/AutoFixTest.cpp:10:17: warning: Braced-initialization {}, without equals sign, shall be used for variable initialization
std::int8_t x3 = {50};
               ^
               {50}
3 warnings generated.
ognjen@ognjen-Precision-7710:~/Desktop/llvm-project/build/bin$

```

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

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;

```

```

File Edit View Search Terminal Help
ognjen@ognjen-Precision-7710:~/Desktop/llvm-project/build/bin$ ./auto-fix --rules="A7_1_6" /home/ognjen/Desktop/AutoFixTest.cpp --
/home/ognjen/Desktop/AutoFixTest.cpp:4:23: warning: The typedef specifier shall not be used.
typedef unsigned long ulong;
                        ^
using ulong = unsigned long
/home/ognjen/Desktop/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/Desktop/AutoFixTest.cpp:8:13: warning: The typedef specifier shall not be used.
typedef int int_t, *intp_t;
            ^
using int_t = int
/home/ognjen/Desktop/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.
ognjen@ognjen-Precision-7710:~/Desktop/llvm-project/build/bin$

```

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

Pravilo A7-2-3

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

Ukoliko se prilikom deklaracije nabiranja ne koristi sintaksa `enum class` može doći do ponovnog deklariranja konstanti iz globalnog opsega. Na listingu 4.4 deklarirano je nabiranje 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 deklarise globalnu promenljivu sa identifikatorom koji je korišćen u okviru nabiranja (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. Redclaration of E10.
6 | static std::int32_t E10;
```

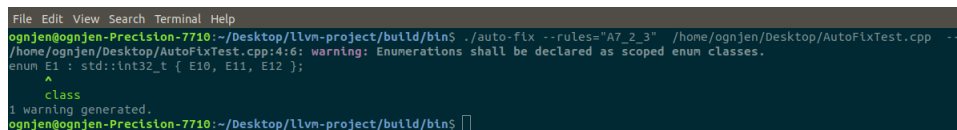
Korišćenjem nabiranja sa opsegom, odnosno upotrebom sintakse `enum class`, identifikatori korišćeni prilikom nabiranja biće deklarirani u svom unutrašnjem opsegu i time sprečiti mogućnost dvostruke deklaracije identifikatora u globalnom opsegu.

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

Listing 4.5: 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 };
```



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

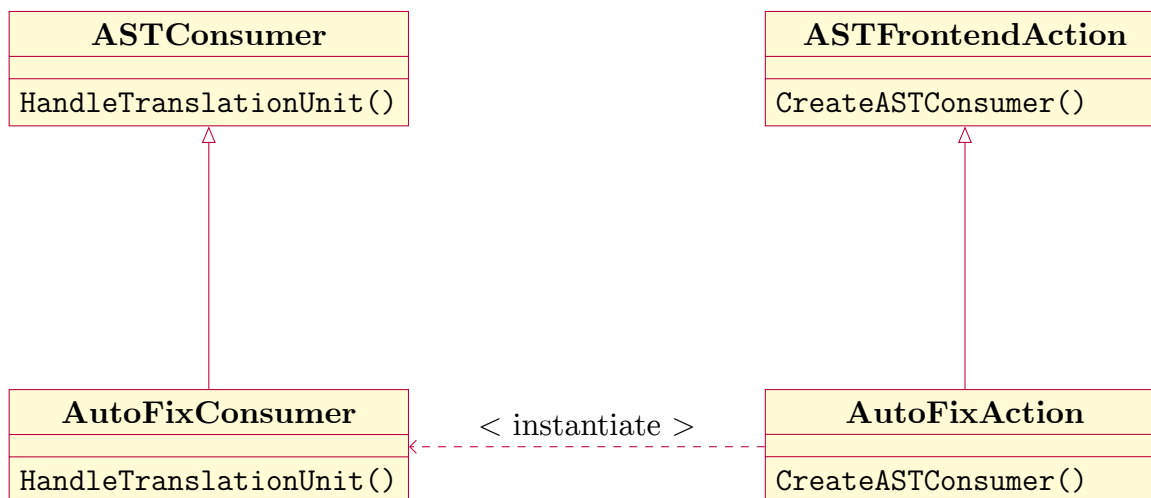
4.3 Opis implementacije alata

Alat *AutoFix* implementiran je u okviru projekta `clang-tools-extra`, potprojekta kompajlerske infrastrukture LLVM. Projekat `clang-tools-extra` sadrži alate implementirane interfejsima za alate kompajlera *Clang* (eng. *Clang's tooling APIs*). Alat *AutoFix* je podjeljen 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 sintaksnog stabla kompajlera *Clang*. *AutoFix* definiše uparivače za uparivanje osnovnih konstrukta na koje se pravilo odnosi i za sužavanje pretrage apstraktnog sintaksnog stabla. Na primer, ukoliko je pravilo vezano za enumeratore (nabrajanja), uparivač koji odgovara ovom pravilu će upariti sve deklaracije enumeratora iz apstraktnog sintaksnog 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 izmena 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`.



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

AutoFix.cpp

Ova jedinica 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 3.4). Osnovna uloga 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 sintaksnom stablom implementirano je u okviru metode `HandleTranslationUnit` klase `AutoFixConsumer`. Ova metoda se poziva tokom parsiranja na kraju izgradnje apstraktnog sintaksnog 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`.

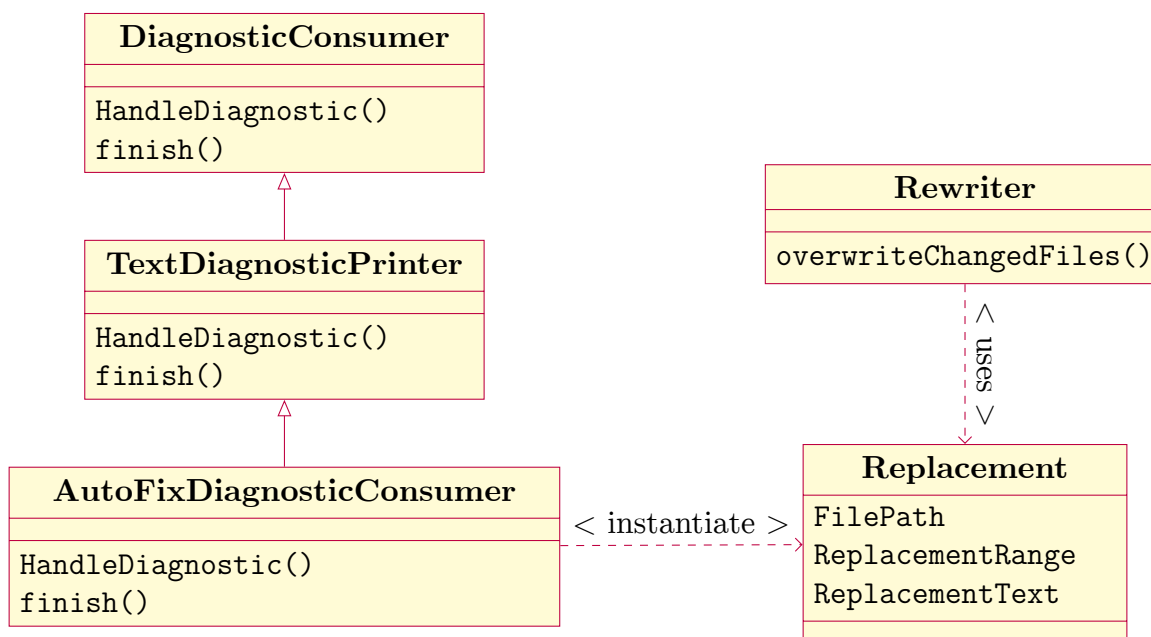
U okviru jedinice prevođenja `Autofix.cpp` takođe je implementirana funkcija `main` alata *AutoFix*. U okviru nje vrši se parsiranje opcija komandne linije, kreira se instanca alata, kreiranoj instanci se pridružuje objekat klase `AutoFixDiagnosticConsumer` i alat se pokreće nad zadatom jedinicom prevođenja.

AutoFixDiagnosticConsumer.cpp

Klasa `DiagnosticConsumer` u okviru kompajlera *Clang* ima ulogu da obradi (konzumira) dijagnostiku prijavljenu za izvorni kôd. Za alat *AutoFix* najbitnije metode ove klase su `HandleDiagnostic` i `finish`. Metod `HandleDiagnostic` poziva se nakon prijavljivanja svakog upozorenja u okviru alata i služi za obradu tog upozorenja. Metod `finish` poziva se nakon što su prijavljena i obrađena sva upozorenja u okviru alata i služi za dodatnu obradu celokupne dijagnostike. Vid obrade dijagnostike najrelevantniji za alat *AutoFix* jeste njeno ispisivanje na standardni izlaz. U ovu svrhu *AutoFix* koristi funkcionalnost postojeće klase `TextDiagnosticPrinter`, potklase klase `DiagnosticConsumer`. Ispisivanje dijagnostike u okviru klase `TextDiagnosticPrinter` implementirano je u okviru predefinisane metode `HandleDiagnostic`.

Pored ispisivanja poruka upozorenja i predloga izmena koda na standardni izlaz, alat *AutoFix* konzumira 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.

Dodatno, 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 izmene koda i sam predlog izmene (`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 metoda `overwriteChangedFiles` klase `Rewriter`. Na slici 4.7 prikazan je odnos klase `DiagnosticConsumer`, `TextDiagnosticPrinter`, `AutoFixDiagnosticConsumer`, `Rewriter` i `Replacement`.



Slika 4.7: 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 sintaksnog stabla tipa `Expr`.
- `getChildOfType` - pronalazi dete čvora iz apstraktnog sintaksnog 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 [16]. 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 4.6 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 aspekti testiranja alata *AutoFix* sa što manje koda.

Test proverava da se pokretanjem alata *AutoFix* ispisuje adekvatno upozorenje zajedno sa predlogom izmene 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` [13].

U okviru listinga 4.6 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 izmene 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 Linux (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 tekстом navedenim između ta dva regularna izraza. Regularni izraz za putanju na operativnom sistemu Linux koristi se kako test ne bi očekivao specifičnu apsolutnu putanju u okviru ispisa alata

AutoFix.

Listing 4.6: 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:
7     warning: The typedef specifier shall not be used.{{$}}
8 // CHECK-NEXT: {{~}}typedef unsigned long ulong;{{$}}
9 // CHECK-NEXT: {{~}}~~~~~^~~~~{{$}}
10 // CHECK-NEXT: {{~}}using ulong = unsigned long{{$}}
11 // CHECK-NEXT: {{~}}1 warning generated.{{$}}
12 // 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 automobilske 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 su u okviru *AGL* projekta na lokaciji `build/tmp/work/x86_64-linux/re2c-native`. Projekat sadrži 84 fajlova sa ekstenzom `.cc`.

Tabela 4.1 prikazuje broj prijavljenih upozorenja za svako od pravila prilikom pokretanja alata *AutoFix*. Za svako od upozorenja ispisan je i odgovarajući predlog izmene 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 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`.

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

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

Projekat *ninja*

Projekat *ninja* nad kojim je pokretan alat *AutoFix* nalazi su u okviru *AGL* projekta na lokaciji `build/tmp/work/x86_64-linux/ninja-native`. U okviru ovog

direktorijuma, alat *AutoFix* je pokrenut rekurzivno nad 55 fajlova sa ekstenzijom *.cc*.

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 u 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.

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

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

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 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 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 **A8-5-3**, **A8-5-2**, **A7-1-6**, **A7-2-3**. Detaljnim testiranjem upotrebom alata *lit* i *FileCheck* iz kompajlerske 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 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 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::RecursiveASTVisitor<Derived> Class Template Reference. https://clang.llvm.org/doxygen/classclang_1_1RecursiveASTVisitor.html.
- [8] FileCheck - Flexible pattern matching file verifier. <https://llvm.org/docs/CommandGuide/FileCheck.html>.
- [9] How To Setup Clang Tooling For LLVM. <https://clang.llvm.org/docs/HowToSetupToolingForLLVM.html>.
- [10] How to write RecursiveASTVisitor based ASTFrontendActions. <https://clang.llvm.org/docs/RAVFrontendAction.html>.
- [11] ISO official website. <https://www.iso.org/committee/45202.html>.
- [12] libclang: C Interface to Clang. https://clang.llvm.org/doxygen/group__CINDEX.html.

- [13] lit - LLVM Integrated Tester. <https://llvm.org/docs/CommandGuide/lit.html>.
- [14] Matching the Clang AST. <https://clang.llvm.org/docs/LibASTMatchers.html>.
- [15] “Clang” CFE Internals Manual. <https://clang.llvm.org/docs/InternalsManual.html>.
- [16] AUTOSAR. Guidelines for the use of the C++14 language in critical and safety-related systems, 2017.
- [17] AUTOSAR. AUTOSAR official website, 2018.
- [18] 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.
- [19] Bruno Cardoso Lopes. *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014.
- [20] Bjarne Stroustrup. *The C++ Programming Language*. Addison–Wesley, fourth edition, 2013.
- [21] 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 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 međunarodnoj konferenciji *ZINC* [18]:

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)*