

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:

Ključne reči: računarstvo, AUTOSAR, clang, llvm, c++

Sadržaj

1	Uvod	1
2	Standard kodiranja AUTOSAR C++14	2
2.1	Programski jezik C++	2
2.2	Klasifikacija pravila	3
2.3	Opis implementiranih pravila	7
3	Kompajlerska infrastruktura LLVM	10
3.1	Clang	11
3.2	AST biblioteka	12
3.3	Biblioteke za obilazak AST-a	15
3.4	Interfejsi za akcije nad prednjim delom kompajlera	19
3.5	Biblioteke za kreiranje alata	22
4	Zaključak	26
	Literatura	27

Glava 1

Uvod

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 [1]. Cilj AUTOSARA je da stvori i uspostavi otvorenu i standardizovanu softversku arhitekturu za automobilske elektronske upravljačke jedinice (*eng. Electronic Control Units (ECU)*). Radi ostvarenja pomenutih ciljeva AUTOSAR definiše, između ostalog, pravila kodiranja u programskom jeziku C++14 za sigurnosno kritične sisteme. Glavni sektor primene standarda kodiranja AUTOSAR C++14 je automobilska industrija, međutim ovaj standard može biti primenjen i na druge aplikacije za uređaje sa ugrađenim računarom (*eng. embedded systems*). Ovaj standard predstavlja nadogradnju MISRA C++:2008 standarda [11].

2.1 Programski jezik C++

C++ je programski jezik opšte namene koji pruža direktan i efikasan model hardvera u kombinaciji sa strukturama za definisanje lakih (*eng. lightweight*) apstrakcija [13]. 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++

Osnovna ideja iza dizajna programskog jezika C++ jeste da zadrži dizajn jezika C ali ga i proširi tako da jezik, odnosno njegova sintaksa, bude bliska problemu koji rešava. Tako implementiranim programskim jezikom se koncepti rešenja problema mogu izraziti direktno i koncizno. U svrhu toga, C++ pruža:

- Direktna preslikavanja ugrađenih operacija i tipova na hardver kako bi obezbedio efikasno korišćenje memorije i efikasne niske (eng. *low-level*) operacije.
- 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 C++-a 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 [13].

Standard C++14

Programski jezik C++ je standardizovan od strane ISO (*International Standard Organization*) radne grupe poznate kao JTC1/SC22/WG21 [7]. 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 C++-om. Ovde pre svega 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 od kojih je:

- 154 prisvojeno bez modifikacija iz MISRA C++:2008 standarda.
- 131 prisvojeno iz drugih C++ standarda
- 57 pravila je zasnovano na istraživanju, literaturi ili 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 [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 program 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 očigledno potrebno poznavati vrednost desnog operanda, što u opštem slučaju nije moguće precizno zaključiti. Primer ovakvog koda prikazan je na listingu 2.1.

```
1 | #include <iostream>
2 | #include <cstdlib>
3 | #include <stdlib.h>
4 |
5 | int main(){
6 |     int8_t u8a = rand() % 100;
7 |     u8a = (uint8_t) ( u8a << rand() % 10);
8 | }
```

Listing 2.1: Kôd koji ilustruje nemogućnost primene statičke analize

Međutim, ukoliko je desni operand konstanta ili promenljiva konstantnog izraza (ključna reč *constexpr*), vrlo je verovatno da će alat za statičku analizu biti u stanju da zaključi 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.

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

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

Napredniji alati za statičku analizu koji podržavaju simboličko izvršavanje programa (npr. *Clang Static Analyzer* [2]) mogu pokriti i znatno kompleksnije slučajeve od slučaja prikazanog u 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)
Kod 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. infrastruktura

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 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. Infrastruktura pravila odnose se na operativni sistem i hardver [1]. Primer pravila za alate koje je ujedno i infrastruktorno 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.

2.3 Opis implementiranih pravila

Pored formalne klasifikacije opisane u prethodnom poglavlju, 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. Svako 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 odnose 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 [2] (*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 detektovati analizom Clang-ovog tako da se većina konstrukta koji nisu u saglasnosti sa pravilom moguAST-a i da se za njih mogu kreirati razumne alternative koje su u skladu sa standardom AUTOSAR C++14. Primeri pravila koje podržava alat Autofix:

Pravilo A7-1-8 (obvezno, implementaciono, automatizovano)

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

Pravilo A8-5-3 (obvezno, implementaciono, automatizovano)

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

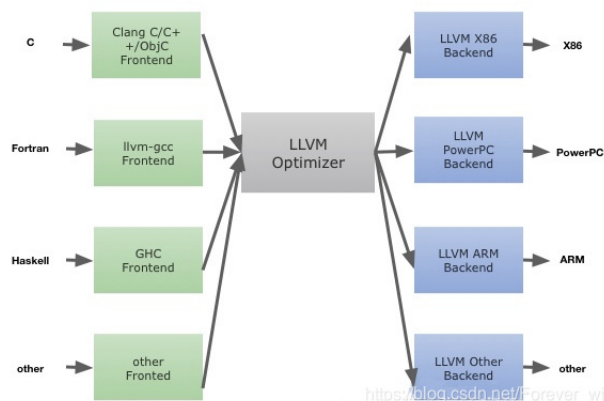
Glava 3

Kompajlerska infrastruktura LLVM

Kompajlerska infrastruktura LLVM predstavlja kolekciju modularnih i ponovo iskoristivih kompajlerskih tehnologija i alata. Započet je 2000. godine kao istraživački projekat Krisa Latnera (eng. *Chris Lattner*) i Vikrama Advea (eng. *Vikram Adve*) na Univerzitetu Illinois.

LLVM podržava kompilaciju različitih programskih programskih jezika na mnoštvo različitih arhitektura hardvera. Jednostavnost dodavanja podrške kompilacije programskog jezika za hardversku arhitekturu omogućeno je fleksibilnim dizajnom kod kog je infrastruktura kompajlera ugrubo podeljena na 3 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, sintakсна i semantička analiza izvornog koda, kreiranje apstraktnog sintaksnog stabla (eng. *abstract syntax tree* (AST)) i generisanje LLVM međukoda koristeći informacije iz AST-a
2. Srednji deo kompajlera vrši niz optimizacija nad instrukcijama LLVM međukoda (eng. *intermediate representation (IR)*). LLVM međukod predstavlja apstrakciju asemblera koja je nezavisna od arhitekture hardvera. LLVM međukod zasnovan je na statičnom pojedinačnom dodeljivanju (eng. *Static Single Assignment* (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.



Slika 3.1: Dizajn kompajlerske infrastrukture LLVM

3.1 Clang

Clang projekat predstavlja prednji deo (*eng. frontend*) LLVM kompajlerske infrastrukture za C familiju jezika (C, C++, Objective C/C++, OpenCL ...). Pored optimizacija i efikasnog generisanja LLVM međukoda karakterističan je po ekspresivnosti dijagnostike odnosno kvalitetu poruka upozorenja i grešaka prijavljenih za izvorni kod. Dizajn Clang-a se sastoji od mnoštva biblioteka od kojih su najznačajnije za dizajn sledeće:

1. Lekser i Pretprocesor

Implementiraju leksičku analizu i pretprocesiranje izvornog koda. Pretprocesor pruža mogućnost uslovne kompilacije, uključivanja datoteka zaglavlja i proširenja makroa. Lekser kreira niz tokena od sintakse izvornog koda.

2. Parser

Kreira sintaksne strukture C++-a od niza tokena dobijenih leksičkom analizom. Clang-ov parser je implementiran kao parser rekurzivnog spuštanja (*eng. recursive-descent parser*), odnosno analizira izvorni kod od vrha ka dnu nizom rekurzivnih funkcija [12].

3. AST biblioteka

Ova biblioteka implementira algoritme i strukture podataka koje parser koristi za izgradnju AST-a. Specifična je po strukturi čvorova koji podsećaju

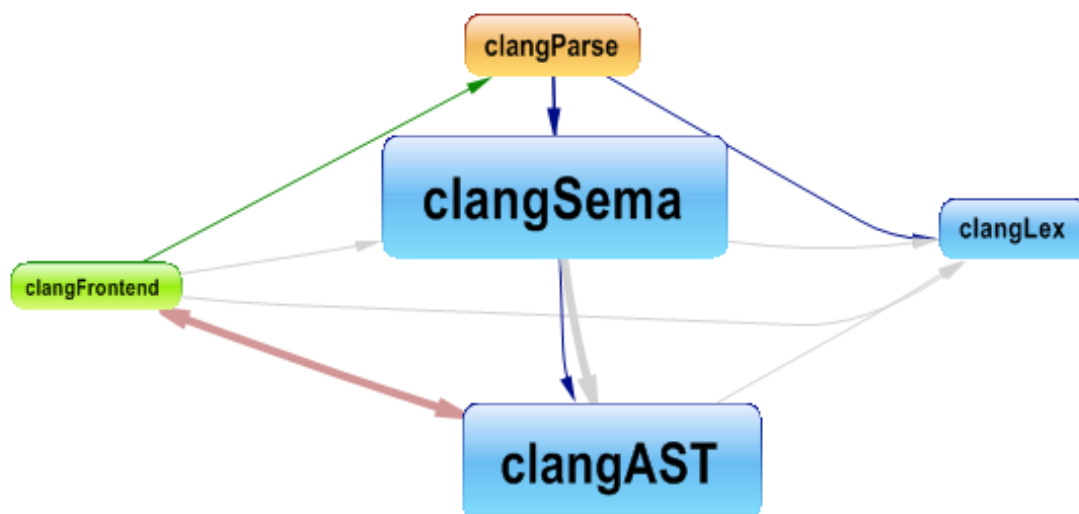
na izvorni C++ kôd što je čini pogodnim za kreiranje alata za refaktorisanje koda i statičku analizu.

4. Sema

Vrši semantičku analizu programa tokom parsiranja i od semantički validnih konstrukta kreira AST. Usko je povezana sa parserom i AST bibliotekom.

5. Biblioteka za generisanje koda (eng. CodeGen Library)

Dobija AST kao ulaz i od njega generiše LLVM međukod.



Slika 3.2: Odnos osnovnih biblioteka Clang-a.

3.2 AST biblioteka

U računarstvu, **apstraktno sintaksno stablo** (eng. abstract syntax tree (AST)), ili samo **sintaksno 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 Clang-ove dijagnostike i jednostavnost kreiranja moćnih alata za statičku analizu u velikoj meri oslanja se na dizajn Clang-ove AST biblioteke.

Struktura AST-a može se jednostavno ispisati na standardni izlaz Clang-ovom `-ast-dump` opcijom. Komanda na listingu 3.1 ispisuje na standardni izlaz AST za kôd iz fajla `hello.cpp` prikazanog na listingu 3.2. Slika 3.3 predstavlja tekstualnu reprezentaciju AST-a ispisanu komandom iz listinga 3.1.

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

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

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

Listing 3.2: Kod čiji je AST prikazan na slici 4.1

```
TranslationUnitDecl 0x55d2e32c1448 <<invalid sloc>> <invalid sloc>
- TypedefDecl 0x55d2e32c1a00 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
- BuiltinType 0x55d2e32c10e0 '__int128'
- TypedefDecl 0x55d2e32c1a70 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
- BuiltinType 0x55d2e32c1700 'unsigned __int128'
- TypedefDecl 0x55d2e32c1db8 <<invalid sloc>> <invalid sloc> implicit __NSConstantString_tag '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' '+'
- BinaryOperator 0x55d2e32f9060 <col:14, col:18> '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 3.3: Primer AST-a

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 C familije jezika. 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.

- **Klasa *Type***

Klasa `Type` biće posebno opisana s obizrom da igra važnu ulogu u ekspresivnosti Clang-ove dijagnostike a samim tim i u kvalitetu alata za statičku analizu. Dizajn ove klase ključan je za preciznost emitovanih poruka upozorenja i grešaka u kodu. 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 *`.

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

Listing 3.3: Demonstracija kanonskih tipova

Ovakvim dizajnom omogućno je 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 suštinski 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 alokira jedan tip, a zatim svi kvalifikovani tipovi pokazuju na alocirani tip na hipu sa dodatim kvalifikatorima [10].

3.3 Biblioteke za obilazak AST-a

Kompajlerska infrastruktura LLVM pruža podršku za jednostavno kreiranje kvalitetnih alata za statičku analizu izvornog koda. Ovi alati baziraju se na korišćenju interfejsa ka Clang-ovom AST-u ili korišćenjem statičkog analizatora kompajlera Clang (eng. *Clang Static Analyzer*) za potrebe simboličkog izvršavanja programa. Alati za statičku analizu mogu koristiti kombinaciju tehnika obilaska AST-a i simboličkog izvršavanja programa u zavisnosti od kompleksnosti analize koja je potrebna. Biblioteke za obilazak AST-a su jeftinije za korišćenje po pitanju računarskih resursa ali su ograničene informacijama dostupnim tokom kompilacije programa. Alati koji se implementiraju kao deo sistema za prevodjenje programa omogućavaju dodatnu optimizaciju procesa pronalaženja nepravilnih konstrukta direktnim proverama tokom kompilacije. Ovo se može postići nadogradnjom osnovnih delova kompajlera kao što su Lexer, Parser ili Sema. Osnovne biblioteke za obilazak AST-a u okviru kompajlera Clang su AST posetioci (eng. *ASTVisitors*) i AST uparivači (eng. *ASTMatchers*).

AST posetioci

AST posetioci implementiraju mehanizam obilaska Clang-ovog AST stabla, odnosno pružaju interfejs za posećivanje svakog čvora u AST stablu. Logika AST posetioca sadržana je u šablonskoj klasi `RecursiveASTVisitor<Derived>`. Ovo je klasa koja posećuje svaki čvor Clang-ovog AST stabla obilaskom u dubinu. AST posetioc je svaka potklasa klase `RecursiveASTVisitor<Derived>`. Klasa `RecursiveASTVisitor<Derived>` obavlja tri odvojena zadatka:

1. Obilazi AST tj. posećuje svaki AST čvor.
2. Za dati čvor ide uz klasnu hijerarhiju počevši od dinamičkog tipa čvora do klase na vrhu hijerarhije (npr. `Stmt`, `Decl` ili `Type`).

3. Za datu kombinaciju (čvor, klasa), gde je klasa neka od baznih klasa dinamičkog tipa čvora, zove funkcije koje korisnik može predefinisati kako bi posetio čvor.

Ova tri zadatka obavljaju tri grupe metoda, redom:

1. `TraverseDecl(Decl *x)` obavlja zadatak 1. 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`. `TraverseStmt(Stmt *x)` i `TraverseType(QualType x)` rade na sličan način.
2. `WalkUpFromFoo(Foo *x)` izvršava zadatak 2. 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)`.
3. `VisitFoo(Foo *x)` izvršava zadatak 3.

Ove tri grupe metoda slede naredni poredak: `Traverse` > `WalkUpFrom` > `Visit`. Metoda (npr. `Traverse`) može pozvati samo metode iz svoje grupe metoda ili iz grupe metoda direktno ispod nje (u smislu predstavljenog poretka). Metoda ne može pozvati metode iz grupe iznad [5]. Primer posetioca prikazan je na listingu 3.4:

```
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() == "n::m::C") {
9 |
10 |             FullSourceLoc FullLocation = Context->getFullLoc(Declaration->getBeginLoc());
11 |
12 |             if (FullLocation.isValid())
13 |                 llvm::outs() << "Found_declaration_at_"
14 |                     << FullLocation.getSpellingLineNumber() << ":"
15 |                     << FullLocation.getSpellingColumnNumber() << "\n";
16 |         }
17 |
18 |         return true;
19 |     }
20 |
21 | private:
22 |     ASTContext *Context;
```

23 || };

Listing 3.4: Primer posetioaca koji posećuje sve strukture, unije i klase i ispisuje lokaciju onih koji se zovu `n::m::C` [6]

Da bi se izvršila neka analiza izvornog koda pomoću AST posetioca dovoljno je naslediti klasu `RecursiveASTVisitor<Derived>` i predefinisati željene Visit metode u okviru nje. Ukoliko je Visit metodama pronađena konstrukcija izvornog koda koja se iz nekog razloga smatra nepravilnom, može se prijaviti upozorenje.

AST uparivači

Biblioteka AST uparivača (eng. *LibASTMatcher*) pruža oblasno specifičan jezik (eng. domain specific language) za kreiranje predikata nad Clang-ovim AST stablom. Ovaj oblasno specifičan jezik je napisan i može se koristiti u C++-u omogućavajući 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.

Na primer, za kreiranje uparivača koji uparuje sve deklaracije klasa ili unija u AST stablu neke jedinice prevođenja, može se koristiti poziv `recordDecl()`. Za sužavanje pretrage, na primer za nalaženje deklaracija svih klasa ili unija sa imenom `Foo`, treba iskoristiti `hasName` matcher: Poziv `recordDecl(hasName("Foo"))` vraća uparivač koji uparuje klase i unije sa imenom **"Foo"** u bilo kom prostoru imena (eng. *namespace*). Podrazumevano, uparivači koji prihvataju više drugih uparivača koriste implicitno `allOf()` metod koji će upariti sve delove stabla na koje referišu njegovi argumenti koji su takođe uparivači. Ovo omogućava dalje sužavanje pretrage. Na primer za uparivanje klasa koje nasleđuju **"Bar"**, uparivač bi izgledao ovako: `recordDecl(hasName("Foo"), isDerivedFrom("Bar"))`.

Uopšteno, strategija kreiranja uparivača se svodi da sledeće korake:

1. Naći baznu klasu u Clang-ovom AST-u koju je potrebno upariti.
2. Naći u AST Matcher Reference dokumentu 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.

Izrazi za uparivanje (eng. *match expressions*) omogućuju uparivanje delova AST stabla. Nakon uparivanja, nad uparenim konstruktom uglavnom se vrši određena analiza, na primer ispitivanje saglasnosti konstrukta sa pravilom standarda za pravilno pisanje C++ koda.

Zbog toga, uparivači koji uparuju specifične čvorove AST stabla se mogu „vezati” (eng. *binding*). Na primer, `recordDecl(hasName("MyClass")).bind("id")` će vezati upareni `recordDecl` čvor za string `id` kako bi se kasnije mogao koristiti u povratnom pozivu uparivača (eng. *match callback*) [9]. Na listingu 3.6 prikazan je primer AST uparivača.

```
1 DeclarationMatcher NonFinalClassWithNonVirtualDestructor::makeMatcher() {
2     return cxxRecordDecl(isClass(), unless(isFinal()),
3         anyOf(hasMethod(cxxDestructorDecl(isPublic(),
4             unless(isVirtual()))),
5             unless(hasMethod(cxxDestructorDecl()))))
6         .bind("nonFinalClassWithNonVirtualDestructor");
7 }
8
9 void NonFinalClassWithNonVirtualDestructor::
10     NonFinalClassWithNonVirtualDestructorCallback::run(
11         const MatchFinder::MatchResult &Result) {
12
13     const BoundNodes &BN = Result.Nodes;
14
15     if (const clang::CXXRecordDecl *CRD = BN.getNodeAs<clang::CXXRecordDecl>(
16         "nonFinalClassWithNonVirtualDestructor"))
17         reportWarning(
18             SM.getDiagnostics(), CRD->getBeginLoc(),
19             diag::warn_non_final_class_with_non_virtual_destructor);
20 }
21
22 void NonFinalClassWithNonVirtualDestructor::runMatcher(
23     clang::ASTContext &AC) {
24     MatchFinder NonFinalClassWithNonVirtualDestructor;
25     NonFinalClassWithNonVirtualDestructor.addMatcher(makeMatcher(), &CB);
26     NonFinalClassWithNonVirtualDestructor.matchAST(AC);
27 }
```

Listing 3.5: Primer uparivča koji pronalazi sve klase koje nisu obeležene atributom `final` i čiji destruktor nije virtuelan, `Callback` klase i poziva uparivača

3.4 Interfejsi za akcije nad prednjim delom kompajlera

Akcije nad prednjim delom kompajlera predstavljaju 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`, `FrontendAction` i njihovih potklasa.

Interfejs `ASTConsumer`

`ASTConsumer` je apstraktni interfejs koji omogućava izvršavanje različitih akcija nad AST-om nezavisno od toga kako je AST kreiran. Akcije se mogu izvršavati u različitim fazama tokom kreiranja AST-a [6]. Na primer, metod `virtual void HandleInlineFunctionDefinition (FunctionDecl *D)` biće pozvan svaki put kada se završi kreiranje umetnutih (eng. *inline*) funkcija prilikom kreiranja AST-a. `ASTConsumer` definiše niz sličnih virtuelnih metoda koje mogu biti predefinisane od strane klasa koje nasleđuju ovaj interfejs. Jedna od najznačajnijih upotreba ovog interfejsa jeste generisanje LLVM međukoda implementacijom koju pruža klasa `CodeGenerator`.

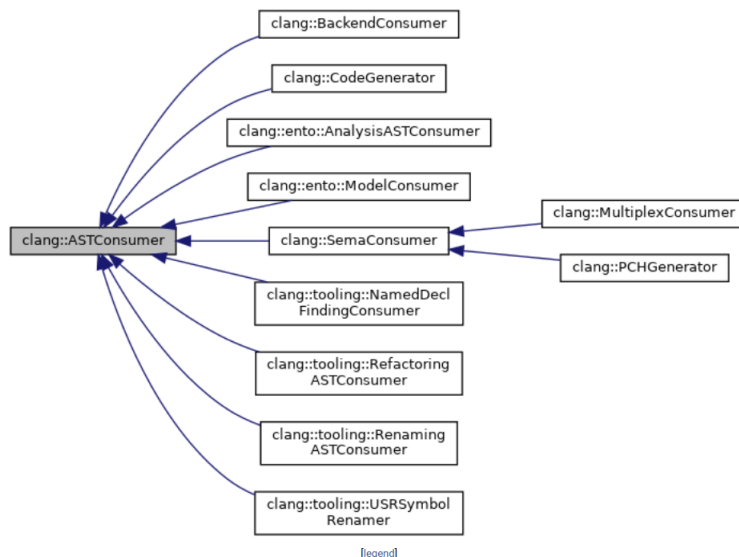
Ovaj interfejs koristan je i za kreiranje samostalnih alata za statičku analizu koji se baziraju na analizi AST-a. U ovu svrhu može se koristiti kombinacija ovog interfejsa sa interfejsima za obilazak AST-a kao što su AST posetioci i AST uparivači. Klasa koja implementira neku analizu nad AST-om treba sadržati jedan ili više objekata klase za obilazak AST-a i zatim izvršiti taj obilazak predefinisanjem metoda `virtual void HandleTranslationUnit (ASTContext &Ctx)` koji se poziva nakon što je kreiran AST za jedinicu prevođenja. Logika same analize AST-a tokom obilaska treba biti implementirana u okviru klase koja implementira obilazak, odnosno u AST uparivačima ili posetiocima [3].

```
1 | class FindNamedClassVisitor
2 | : public RecursiveASTVisitor<FindNamedClassVisitor> {
3 | public:
4 |     bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
5 |         Declaration->dump();
6 |         return true;
    | }
```



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

Inheritance diagram for clang::ASTConsumer:



Slika 3.4: Klase koje implementiraju *ASTConsumer* interfejs

```

7 | }
8 | };
9 |
10 | class FindNamedClassConsumer : public clang::ASTConsumer {
11 | public:
12 |     virtual void HandleTranslationUnit(clang::ASTContext &Context) {
13 |         Visitor.TraverseDecl(Context.getTranslationUnitDecl());
14 |     }
15 | private:
16 |     FindNamedClassVisitor Visitor;
17 | };

```

Listing 3.6: Primer upotrebe klase *ASTConsumer* [6]

Intefejs FrontendAction

FrontendAction je apstraktna klasa za akcije koje mogu biti izvršene od strane prednjeg dela kompajlera (eng. *frontend*). Ovu klasu karakteriše jednostavan javni interfejs koji se sastoji od sledećih metoda:

- `bool PrepareToExecute (CompilerInstance &CI)`

Ova metoda služi za pripremanje primarne akcije koja će biti izvršena nad

objektom klase `CompilerInstance`. Priprema uključuje izmene početne konfiguracije kompajlera kako bi olakšala izvršavanje akcije.

- `bool` `BeginSourceFile` (`CompilerInstance &CI`, `const` `FrontendInputFile &Input`)
Priprema akciju za procesiranje ulaznog fajla. Ukoliko ova metoda vrati vrednost `false`, kompilacija fajla `Input` će biti prekinuta i primarna akcija se neće izvršiti.
- `llvm::Error` `Execute` ()
Metoda odgovorna za izvršavanje akcije. Ova metoda interno poziva čistu virtuelnu metodu `virtual void` `ExecuteAction()`=0 koju svaka podklasa klase `FrontendAction` predefiniše implementirajući u okviru nje akciju koja će biti izvršena.
- `virtual void` `EndSourceFile`()
Izvršava post-procesiranje nakon svakog fajla, dealocira objekte, obrađuje statistiku i sređuje kôd u izlaznom fajlu.

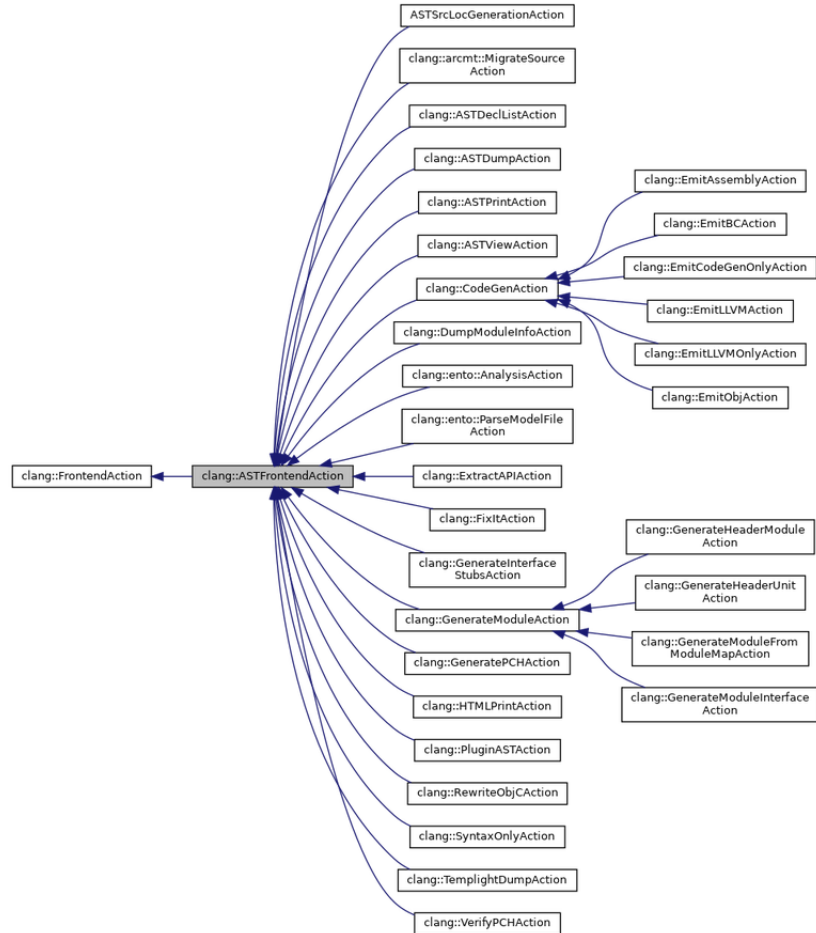
Klasa `FrontendAction` ima raznolike upotrebe, odnosno specijalizacije. Primer specijalizacija ove klase su `DumpCompilerOptionsAction` koja omogućava ispisivanje opcija koje se mogu zadati kompajleru i `PreprocessorFrontendAction` koja omogućava izvršavanje akcija vezanih za pretprocesiranje izvornog 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`.

Ova klasa oslanja se na korišćenje interfejsa `ASTConsumer`. Dovoljno je inicijalizovati klasu `ASTFrontendAction` objektom implementiranog `ASTConsumer`-a, a zatim u metodi `ExecuteAction()` kreirati parserom AST, tokom čega će se izvršavati akcije opisane u `ASTConsumer`-u. Neke od bitnih implementacija ovog interfejsa predstavljaju klase `CodeGenAction`, `ASTDumpAction`, `FixitAction` [4]... Na listingu 3.7 prikazan je primer upotrebe ovog interfejsa.

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

7 || };

Listing 3.7: Primer upotrebe klase `ASTFrontendAction` interfejsa [6]



Slika 3.5: Klase koje implementiraju `ASTFrontendAction` interfejs

3.5 Biblioteke za kreiranje alata

Kompajler Clang pruža infrastrukturu za pisanje različitih 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 ka Clang-u visokog nivoa (eng. *high level*). 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 [8]. Ovo ga čini nepogodnim za implementaciju alata za statičku analizu ali omogućava stabilnost pri promeni verzija Clang-a. Treba ga koristiti u slučajevima kada:

- je potreban interfejs ka Clang-u iz jezika koji nije C++.
- je potreban stabilni interfejs koji je kompatibilan sa starijim verzijama Clang-a.
- su potrebne apstrakcije visokog nivoa kao što je iteriranje kroz AST sa kursorima i drugi detaljni vezani za Clang-ov AST.

LibClang ne treba koristiti kada je potrebna puna kontrola nad AST-om [1].

- 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*).

Treba ih 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 Clang-ovim AST-om.

Ne treba ih 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 [1].

- 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 Clang-ovim AST-om.
 - je potrebno deliti kôd sa Clang-ovim Plugin-ovima.

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 API (eng. *Application Programming Interface*) promeni.
- su potrebne apstrakcije visokog nivoa kao što su kursori.
- alat neće biti napisan u C++-u [1].

Na listingu 3.8 prikazana je implementacija jednostavnog alata korišćenjem opisanih interfejsa ASTVisitor, ASTConsumer i FrontendAction. Ovaj alat koristi libtooling biblioteku za pokretanje definisane FrontendAction akcije nad izvornim kodom koji je prosleđen kao argument komandne linije. Alat ispisuje lokacije svih struktura, unija i klasa sa imenom `n::m::C`.

```
1 | #include "clang/AST/ASTConsumer.h"
2 | #include "clang/AST/RecursiveASTVisitor.h"
3 | #include "clang/Frontend/CompilerInstance.h"
4 | #include "clang/Frontend/FrontendAction.h"
5 | #include "clang/Tooling/Tooling.h"
6 |
7 | using namespace clang;
8 |
9 | class FindNamedClassVisitor
10 | : public RecursiveASTVisitor<FindNamedClassVisitor> {
11 | public:
```

```
12 | explicit FindNamedClassVisitor(ASTContext *Context)
13 | : Context(Context) {}
14 |
15 | bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
16 |     if (Declaration->getQualifiedNameAsString() == "n::m::C") {
17 |         FullSourceLoc FullLocation = Context->getFullLoc(Declaration->getBeginLoc());
18 |         if (FullLocation.isValid())
19 |             llvm::outs() << "Found_declaration_at_"
20 |                 << FullLocation.getSpellingLineNumber() << ":"
21 |                 << FullLocation.getSpellingColumnNumber() << "\n";
22 |     }
23 |     return true;
24 | }
25 |
26 | private:
27 |     ASTContext *Context;
28 | };
29 |
30 | class FindNamedClassConsumer : public clang::ASTConsumer {
31 | public:
32 |     explicit FindNamedClassConsumer(ASTContext *Context)
33 |         : Visitor(Context) {}
34 |
35 |     virtual void HandleTranslationUnit(clang::ASTContext &Context) {
36 |         Visitor.TraverseDecl(Context.getTranslationUnitDecl());
37 |     }
38 | private:
39 |     FindNamedClassVisitor Visitor;
40 | };
41 |
42 | class FindNamedClassAction : public clang::ASTFrontendAction {
43 | public:
44 |     virtual std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
45 |         clang::CompilerInstance &Compiler, llvm::StringRef InFile) {
46 |         return std::make_unique<FindNamedClassConsumer>(&Compiler.getASTContext());
47 |     }
48 | };
49 |
50 | int main(int argc, char **argv) {
51 |     if (argc > 1) {
52 |         clang::tooling::runToolOnCode(std::make_unique<FindNamedClassAction>(), argv[1]);
53 |     }
54 | }
```

Listing 3.8: Primer implementacije jednostavnog alata upotrebom interfejsa ASTVisitor, ASTConsumer, FrontendAction i bibliotekom libtooling [6]

Glava 4

Zaključak

Literatura

- [1] Choosing the Right Interface for Your Application. <https://clang.llvm.org/docs/Tooling.html>.
- [2] Clang Static Analyzer website. <https://clang-analyzer.llvm.org/>.
- [3] clang::ASTConsumer Class Reference. https://clang.llvm.org/doxygen/classclang_1_1ASTConsumer.html.
- [4] clang::FrontendAction Class Reference. https://clang.llvm.org/doxygen/classclang_1_1FrontendAction.html.
- [5] clang::RecursiveASTVisitor<Derived> Class Template Reference. https://clang.llvm.org/doxygen/classclang_1_1RecursiveASTVisitor.html.
- [6] How to write RecursiveASTVisitor based ASTFrontendActions. <https://clang.llvm.org/docs/RAVFrontendAction.html>.
- [7] ISO official website. <https://www.iso.org/committee/45202.html>.
- [8] libclang: C Interface to Clang. https://clang.llvm.org/doxygen/group__CINDEX.html.
- [9] Matching the Clang AST. <https://clang.llvm.org/docs/LibASTMatchers.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] Bruno Cardoso Lopes. *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014.

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

Biografija autora