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

Glava 1

Uvod

Glava 2

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 [5]. Kreirao ga je Danski softverski inženjer Bjarne Stroustrup kao ekstenziju programskog jezika *C*. Osnovno proširenje u odnosu na programski jezik *C* jeste mogućnost kreiranja korisnički definisanih tipova, odnosno klasa. *C++* pripada grupi objektno orijentisanih jezika.

2.1 Dizajn programskog jezika C++

Programski jezik *C* dizajniran je sa ciljem da programer može sto jednostavnije zadavati akcije koje mašina treba da izvrši. 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 mapiranja 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, upravljanje 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 [5].

2.2 Standard C++14

Programski jezik *C++* je standardizovan od strane ISO (*International Standard Organization*) radne grupe poznate kao JTC1/SC22/WG21 [2]. 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 da 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.

Glava 3

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 [4]. 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 [3].

3.1 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 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 deli pravila na:

  1. automatizovana
  2. delimično automatizovana
  3. neautomatizovana

  Automatizovana su ona pravila kod kojih se ispitivanje saglasnosti koda može u potpunosti automatizovati algoritmima statičke analize. Kod delimično automatizovanih pravila se ispitivanje saglasnosti koda može samo delimilčno automatizovati, na primer, korišćenjem neke heuristike ili pokrivanjem određenog broja slučajeva upotrebe i služi kao dopuna 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 [3].

  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 <cstdint>
3 #include <cstdlib>
4
5 int main(){
6 int8_t u8a = rand() % 100;
7 u8a = (uint8_t) ( u8a << rand() % 10);
8 }
```

Listing 3.1: Kôd koji ilustruje nemogućnost primene statičke analize

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

Listing 3.2: Kôd čija se ispravnost jednostavno može utvrditi statičkom analizom

Napredniji alati za statičku analizu koji podržavaju simboličko izvršavanje programa (npr. Clang Static Analyzer [1]) 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 (slučaju upotrebe) deli pravila na:

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

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

  > Pravilo A2-9-1 (obvezno, implementaciono, automatizovano)
  > Ime heder fajla mora biti identično imenu tipa deklarisanog u njemu ukoliko 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 pogresna „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 [3]. 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.

3.2 Opis implementiranih pravila

Pored formalne klasifikacije opisane u prethodnom poglavlju, pravila u okviru samog dokumenta standarda AUTOSAR C++14 kodiranja struktuirana 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 računarastva, 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 Clang kompajlera 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 Autofix alata, 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. Clang Statički analizator (*eng. Clang Static Analyzer* [1]) nije korišćen u okviru ovog alata, tako da su pravila izabrana tako da što manji broj slučajeva upotrebe zahteva simboličko izvršavanje programa. Drugo ograničenje potiče iz činjenice da u nekim slučajevima nije moguće ili je znatno komplikovanije kreirati predlog ispravke koda (*eng. fixit hint*). Pravila razmatrana u okviru ovog rada birana su tako da se većina konstrukta koji nisu u saglasnosti sa pravilom mogu detektovati analizom Clang-ovog AST-a i da se za njih mogu kreirati razumne alternative koje su u skladu sa standardom Autosar C++14. Primeri pravila koje podržava Autofix alat:

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 4

LLVM (nije spremno za reivew)

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

LLVM podržava kompilaciju različitih programskih programiskih 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. Izvorni kôd podržanih jezika prevodi se u LLVM međukod bibliotekama koji predstavljaju prednji deo kompajlera (*eng. frontend*). Zatim se nad međukodom vrši niz optimizacija koje su nezavisne od izvornog koda ali i ciljne arhitekture. Biblioteke koje implementiraju pomenute optimizacije čine srednji deo (*eng. middleend*) infrastrukture LLVM-a. Poslednji deo dizajna čini zadnji deo (*eng. backend*) kompajlera prilikom kog se generiše izvršni kôd za ciljnu arhitekturu od LLVM međukoda.

4.1 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:

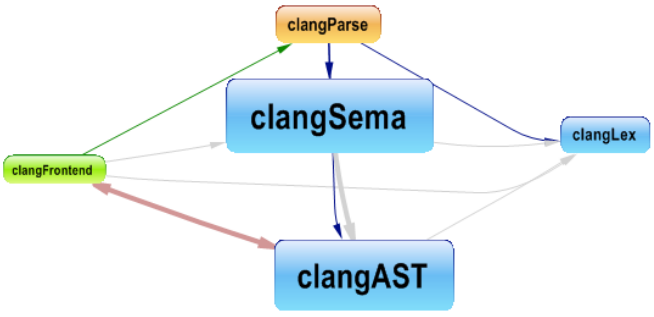
1. **Lekser i Predprocesor**
   Implementira leksičku analizu i predprocesiranje izvornog koda. Pruža mogućnost uključivanja datoteka zaglavlja, proširenja makroa, uslovne kompilacije i kontrole linija. Kreira niz tokena od sintakse izvornog koda.

2. **Parser**
   Kreira sintaksne strukture C++-a od niza tokena dobijenih leksičkom analizom. Clang-ov parser implementiran je kao parser rekurzivnog spuštanja, odnosno analizira izvorni kod od vrha ka dnu nizom rekurzivnih funkcija.

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++ kod 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 4.1: Odnos osnovnih biblioteka Clang-a

4.2 AST biblioteka

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 kod iz fajla hello.cpp prikazanog na listingu 3.2. Slika 3.1 predstavlja AST ispis dobijen komandom iz listinga 3.1.

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

Listing 4.1: 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 4.2: Kod čiji je AST prikazan na slici 3.1

Čvorovi od kojih je izgrađen AST predstavljaju apstrakciju sintaksnih struktura iz samog jezka. 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 (promenljivih) ne nasleđuju dikertno klasu Decl već nasleđuju više njenih podklasa.

- ***Klasa Type***
  Tipovi igraju važnu ulogu u ekspresivnosti Clang-ove dijagnostike. Dizajn ove klase ključan je za preciznost emitovanih poruka upozorenja i grešaka u kodu. Na primer, upozorenja vezana za kod 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 korićenjem *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 4.3: 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 alocira jedan tip, a zatim svi kvalifikovani tipovi pokazuju na alocirani tip na hipu sa dodatim kvalifikatorima.

- *Stmt*
- *Type*

Glava 5

Zaključak

Literatura

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

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

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

[4] AUTOSAR. AUTOSAR official website, 2018.

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

Biografija autora