

UNIVERZITET U BEOGRADU  
MATEMATIČKI FAKULTET



Ognjen Ž. Plavšić

## **RADNI NASLOV**

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:** RADNI NASLOV

**Rezime:**

**Ključne reči:** računarstvo, autosar, clang, llvm, c++

# Sadržaj

<b>1</b>	<b>Uvod</b>	<b>1</b>
<b>2</b>	<b>Autosar C++14 standard kodiranja</b>	<b>2</b>
2.1	Klasifikacija pravila . . . . .	2
2.2	Opis implementiranih pravila . . . . .	5
<b>3</b>	<b>LLVM (nije spremno za reivew)</b>	<b>7</b>
3.1	Clang . . . . .	7
3.2	AST biblioteka . . . . .	8
<b>4</b>	<b>Zaključak</b>	<b>9</b>
	<b>Literatura</b>	<b>10</b>

# Glava 1

## Uvod

## Glava 2

# Autosar C++14 standard kodiranja

AUTomotive Open System ARchitecture (AUTOSAR) je razvojno partnerstvo proizvođača vozila, dobavljača, pružaoca usluga i kompanija iz automobilske industrije i industrija elektronike, poluprovodnika i softvera na globalnom nivou [2]. Cilj Autosara je da stvori i uspostavi otvorenu i standardizovanu softversku arhitekturu za automobilske elektronske upravljačke jedinice (ECU). Radi ostvarenja pomenutih ciljeva AUTOSAR definiše, između ostalog, pravila kodiranja u programskom jeziku C++14 za sigurnosna i kritična okruženja. Glavni sektor primene AUTOSAR C++14 standarda kodiranja je automobilska industrija, međutim ovaj standard može biti primenjen i na druge ugrađene (*eng. embedded*) aplikacije. Pomenuti standard predstavlja nadogradnju postojećeg MISRA C++:2008 standarda [1].

### 2.1 Klasifikacija pravila

AUTOSAR C++14 standard definiše 342 pravila od kojih je 154 prisvojeno bez modifikacija od MISRA C++:2008 standarda, 131 su prisvojeni iz drugih C++ standarda i 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++ kod mora ispuniti kako bi bio u saglasnosti sa standardom. U slučaju kada ovo nije moguće, formalna od-

stupanja moraju biti prijavljena. Preporučena pravila predstavljaju zahteve koje C++ kod treba ispuniti 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 i treba ih pratiti kad god je to izvodljivo u praksi. Za ova pravila ne moraju biti prijavljena formalna odstupanja.

Klasifikacija po primenjivosti statičke analize deli pravila na automatizovana, delimično automatizovana i 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 manuelnog pregleda koda. Za neautomatizovana pravila statička analiza ne pruža razumnu podršku. Za ispitivanje saglasnosti koda sa neautomatizovanim pravilima koriste se druga sredstva, kao što su manualni pregled koda ili drugi alati.

Većina pravila iz Autosar C++14 standarda spadaju u Automatizovana pravila. Alati za statičku analizu koda koji tvrde da podržavaju Autosar C++14 standard 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 [1].

Primenjivost statičke analize na proveru saglasnosti koda sa određenim pravilom u velikoj meri zasniva se na teorijskoj klasifikaciji problema na odlučive i neodlučive. Ukoliko se pravilo zasniva na neodlučivom problemu, odnosno dokazano je da ne postoji algoritam koji bi u konačnom broju koraka odgovorio sa DA ili NE na pomenuti problem, možemo sa sigurnošću reći da alati za statičku analizu nisu u mogućnosti da u potpunosti ispituju saglasnost koda sa ovim pravilom. Pravilo će verovatno biti klasifikovano kao parcijalno automatizovano ili neautomatizovano ukoliko detektovanje kršenja pravila obuhvata određivanje vrednosti koju promenljiva sadrži ili da li program doseže određeni deo programa.

Primer parcijalno automatizovanog pravila je:

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

Pravilo nije moguće u potpunosti automatizovati jer je očigledno potrebno po-



znovati vrednost desnog operanda, što u opštem slučaju nije moguće 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: Kod koji ilustruje nemogućnost primene statičke analize

Medjitim, ukoliko je desni operand konstanta ili `constexpr` promenljiva, 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 <stdlib.h>
4
5 int main(){
6     int8_t u8a = rand() % 100;
7     u8a = (uint8_t) ( u8a << 7);
8 }
```

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

Napredniji alati za statičku analizu koji podržavaju simboličko izvršavanje programa (npr. Clang Static Analyzer) 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 kod napisan, stoga je nemoguće koristiti algoritme statičke analize. Primer ovakvog pravila je:

Pravilo A1-4-2 (obvezno, implementaciono, neautomatizovano)

Sav kod treba poštovati definisane granice metrika koda.

Kako bi se odredilo da li je kod napisan u skladu sa ovim pravilom, očigledno je potrebno 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 implementaciona, verifikaciona, pravila za alate i infrastrukturna. Implementaciona su ona pravila koja se odnose na samu implementaciju projekta odnosno na kod, arhitekturu i dizajn. Verifikaciona pravila odnose se na proces verifikacije koji uključuje pregled koda, analizu i testiranje. Pravila za alate odnose se na softverske alate kao što su preprocesor, kompajler, linker i biblioteke kompajlera. Infrastrukturna su ona pravila koja se odnose na operativni sistem i hardver [1].

## 2.2 Opis implementiranih pravila

Pored formalne klasifikacije opisane u prethodnom poglavlju, pravila u okviru samog dokumenta AUTOSAR C++14 standarda 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.

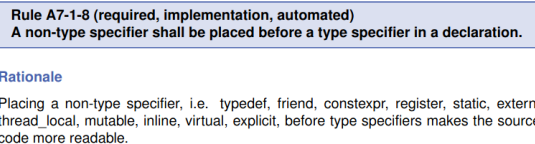
Pravila razmatrana u ovom radu predstavljaju podskup pravila koja se odnose na deklaracije. Razlog za ovo je dvojak. Deklaracije predstavljaju jedan od osnovnih i najvažnijih koncepta u C++-u i programiranju generalno. U C++-u deklaracije čine samu srž ekspresivne moći jezika i u direktnoj su vezi sa 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 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:

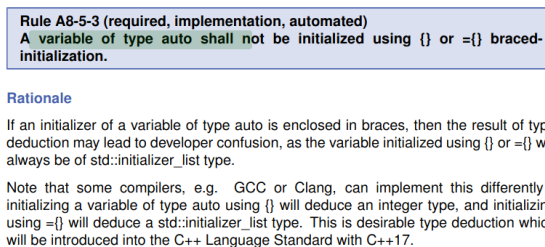
1. Obavezna, prema klasifikaciji po obavezi.
2. Automatizovana, prema klasifikaciji po primenjivosti statičke analize.

### 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*) 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 Autosar C++14 standardom. Primeri pravila koje podržava Autofix alat prikazani su na slikama 2.1 i 2.2.



Slika 2.1: Pravilo A7-1-8



Slika 2.2: Pravilo A8-5-3

## Glava 3

# 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 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. Izvorni kod podržanih jezika prevodi se u LLVM medjukod bibliotekama koji predstavljaju prednji deo kompajlera (*eng. frontend*). Zatim se nad medjukodom 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 kod za ciljnu arhitekturu od LLVM medjukoda.

### 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 medjukoda 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

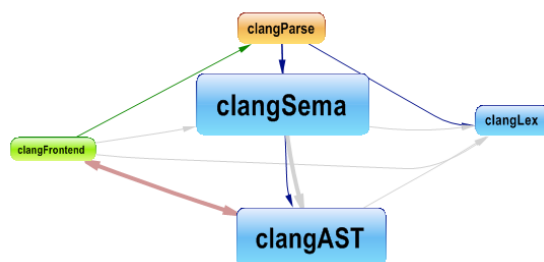
Ova biblioteka implementira algoritme i strukture podataka koje parser koristi za izgradnju AST-a. Specifična je po strukturi čvorova koji podsećaju na pisani C++ kod što je čini pogodnim za kreiranje alata za refaktorisanje koda i statičku analizu.

### 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 medjukod.



Slika 3.1: Odnos osnovnih biblioteka Clang-a

## 3.2 AST biblioteka

**Glava 4**

**Zaključak**

# Literatura

- [1] AUTOSAR. Guidelines for the use of the C++14 language in critical and safety-related systems, 2017.
- [2] AUTOSAR. AUTOSAR official website, 2018.

## Biografija autora