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

Glava 1

Uvod

Glava 2

Autosar C++14 standard kodiranja

AUTomotive Open System ARchitecture (AUTOSAR) je razvojno partnerstvo proizvodjača 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, izmedju ostalog, pravila kodiranja u programskom jeziku C++14 za sigurnosna i kritična okruženja. Glavni sektor primene AUTOSAR C++14 standarda kodiranja je automobilska industrija, medjutim ovaj standard može biti primenjen i na druge ugradjene (*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 moguce. Medjutim, 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 delimilčno 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 manuelni 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 odredjenim 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 ispitaju saglasnost koda sa ovim pravilom. Pravilo će verovatno biti klasifikovano kao parcijalno automatizovano ili neautomatizovano ukoliko detektovanje kršenja pravila obuhvata odredjivanje vrednosti koju promenljiva sadrži ili da li program doseže odredjeni deo programa.

Primer parcijalno automatizovanog pravila je:

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

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

znavati vrednost desnog operanda, što u opštem slučaju nije moguće 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 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 <cstdint>
3 #include <cstdlib>
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. Medjutim, odredjeni 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 pretprocesor, 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 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*) 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.

Rule A7-1-8 (required, implementation, automated)
A non-type specifier shall be placed before a type specifier in a declaration.

Rationale

Placing a non-type specifier, i.e. typedef, friend, constexpr, register, static, extern, thread_local, mutable, inline, virtual, explicit, before type specifiers makes the source code more readable.

Slika 2.1: Pravilo A7-1-8

Rule A8-5-3 (required, implementation, automated)
A variable of type auto shall not be initialized using {} or ={} braced-initialization.

Rationale

If an initializer of a variable of type auto is enclosed in braces, then the result of type deduction may lead to developer confusion, as the variable initialized using {} or ={} will always be of std::initializer_list type.

Note that some compilers, e.g. GCC or Clang, can implement this differently - initializing a variable of type auto using {} will deduce an integer type, and initializing using ={} will deduce a std::initializer_list type. This is desirable type deduction which will be introduced into the C++ Language Standard with C++17.

Slika 2.2: Pravilo A8-5-3

Glava 3

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