UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Mirko Brkušanin

# IMPLEMENTACIJA PRAVILA IZ STANDARDA AUTOSAR C++14 U OKVIRU PROGRAMSKOG PREVODIOCA CLANG

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**: _____

*Ivančici i Pajčetu*

**Naslov master rada**: Implementacija pravila iz standarda AUTOSAR C++14 u okviru programskog prevodioca Clang

**Rezime**: Upotrebe smernica i standarda kodiranja su od posebne važnosti kada je reč o programiranju ugrađenih (eng. *embedded*) i sigurnosno kritičnih sistema. Jedan od standarda koji se izdvaja je standard AUTOSAR C++ 14 za kodiranje koji se koristi u automobilskoj industriji gde cene grešaka usled neispravnog softvera, u određenim situacijama, mogu biti čak i fatalne. Upotreba standarda pomaže u povećanju bezbednosti i ispravnosti samog softvera. U radu je predstavljen alat koji obavlja statičku analizu radi provere saglasnosti izvornog koda sa određenim podskupom pravila iz standarda. Implementacija je obavljena upotrebom biblioteka koje pruža potprojekat Clang kompilatorske infrastrukture LLVM. Upoređena su dva pristupa koja su dostupna u Clang-u za ovu vrstu analize: rekurzivni AST-posetioci i AST-uparivači. Izvršena je evaluacija radi poređenja i predstavljeni su eksperimentalni rezultati.

**Ključne reči**: verifikacija softvera, kompilatori, statička analiza, LLVM, Clang, standard MISRA C++, standard AUTOSAR C++, biblioteka LibTooling, apstraktna sintaksna stabla (AST), rekurzivni AST-posetioci, AST-uparivači

# Sadržaj

**1 Uvod** **1**

**2 Standardi MISRA i AUTOSAR** **3**
2.1 Motivacija . . . . . 3
2.2 Standardi kodiranja . . . . . 5
2.3 Standard MISRA . . . . . 7
2.4 Standard AUTOSAR . . . . . 11

**3 Kompilatorska infrastruktura LLVM** **14**
3.1 Istorija projekta LLVM . . . . . 14
3.2 Struktura projekta LLVM . . . . . 15
3.3 Potprojekat Clang . . . . . 17

**4 Implementacija alata *switch-check*** **27**
4.1 Osnovni elementi alata *switch-check* . . . . . 28
4.2 Pravilo M6-4-7 (*no-bool-condition*) . . . . . 30
4.3 Pravilo M6-4-6 (*default-final*) . . . . . 33
4.4 Pravilo A6-4-1 (*two-clause-minimum*) . . . . . 37
4.5 Pravilo M6-4-4 (*structured-label*) . . . . . 42
4.6 Pravilo M6-4-5 (*terminate-clause*) . . . . . 45
4.7 Pravilo M6-4-3 (*well-formed*) . . . . . 51
4.8 Evaluacija . . . . . 53

**5 Zaključak** **59**

**Literatura** **62**

# Glava 1

# Uvod

Upotreba softvera u svakodnevnom životu neprestano raste. Porast upotrebe računara je primetan u mnogim industrijama uključujući automobilsku, avionsku, medicinsku i nuklearnu [16, 17]. Posledice greške mogu se kretati od blage neugodnosti preko materijalne ili finansijke štete po do gubitaka ljudskih života [32]. Samim tim raste potreba za bezbednim i ispravnim softverom. Programski jezik C++ se danas široko koristi u razvoju softvera, kako u ugrađenim sistemima (eng. *embedded*), tako i u sigurnosno kritičnim sistemima [16]. On programerima daje veliku količinu slobode, koja nije uvek poželjna za sisteme gde je bezbednost od izuzetnog značaja.

Prilikom korišćenja jezika C++ potrebno je uložiti određen napor kako be se izbegli konstrukti jezika koji mogu potencijalno dovesti do nepoželjnog ponašanja. Uprkos svemu ovome, jezici C i C++ su ostali vrlo popularni [16]. Kako bi se umanjli razni rizici, često su definisane smernice ili podskupovi jezika koji su se koristili kao preporuke [26]. Mali broj njih je zaživeo. Jedan od izuetaka su smernice koje pruža MISRA (eng. *Motor Industry Software Reliability Association*) [38], među kojima se izdvaja standard kodiranja za jezik C++ [31]. Kao nadogradnja se pojavljuje standard kodiranja AUTOSAR C++ koji takođe potiče iz automobilske industrije i pokriva nova svojsva jezika za verzije C++11 i C++14 [5].

Kako ovi standardi predstavljaju veliki skup pravila koji nije lako upamtiti i držati na umu prilikom samog programiranja, javlja se potreba za alatom koji može pružiti pomoć ili u nekoj meri automatizovati ceo process. Sami kompilatori pored toga što moraju proveriti da li je programski kôd pravilno napisan i ispoštovao sve detalje sintakse, često obavljaju i semantičke provere. Neki kompilatori dolaze sa pomoćnim alatom koji pruža i složenije vrste statičke analize [7, 36].

U ovom radu je predstavljeno jedno rešenje za automatizaciju provera saglasnosti određenog skupa pravila standarda AUTOSAR C++ 14. Predstavlja se implementacija novog alata zasnovanog na bibliotekama koje pruža kompilator otvorenog koda Clang, inače deo kompilatorske infrastrukture LLVM.

U glavi 2 su predstavljeni detalji jezika C++ bitni za implementaciju kompilatora tog jezika kao i upotrebu jezika, detalji njegovog standarda, kao i problemi koji se mogu javiti sa programima napisanim u jeziku C++. Potom su dati opisi standarda kodiranja sa fokusom na standarde MISRA C++ 2008 i AUTOSAR C++ 14.

U glavi 3 je opisana kompilatorska infrastruktura LLVM. Dat je kratak pregled istorije projekta, njene strukture kao i strukture tipičnog kompilatora. Najveća pažnja je posvećena potprojektu Clang koji pored implementacije prednjeg dela kompilatora za jezike kao što je C++, sadrži i veliki broj biblioteka koje se mogu iskoristiti za pisanje raznih alata.

U glavi 4 je predstavljen opis novog alata *switch-check* za statičku analizu koda implementiranog pomoću biblioteka Clang-a. Predstavljeni su opisi implementacije pojedinačnih pravila pomoću dva različita pristupa. Naposletku je data evaluacija u vidu porđenja dva pristupa i predstavljeni su eksperimentalni rezultati.

# Glava 2

# Standardi MISRA i AUTOSAR

U ovoj glavi su opisani neki problemi koji se mogu javiti sa programima napisanim u jeziku C++. Zatim su definisani standardi kodiranja i prednosti njihove upotrebe. Potom su dati i opisi stadarda MISRA C++ 2008 kao i standarda koji ga proširuje, standard AUTOSAR C++14, koji su implementirani u radu.

## 2.1 Motivacija

Programeri prave greške. Mogu biti proste kao što je pogrešno naveden naziv promenljive ili složene kao što je nerazumevanje algoritma ili koda koji se piše. Programski jezik direktno ima uticaj na tip greške koja se može pojaviti. Izražajnost programskog jezika može pomoći ili naškoditi programeru. Greške prilikom programiranja mogu jednu validnu konstrukciju pretvoriti u drugu takođe validnu ali nepoželjnu. Kompilatori neće uvek detektovati greške kada se naprave, ali to ne znači da greške nije moguće otkriti i klasifikovati [31, 5, 3, 12].

Greške se mogu javiti i ukoliko programer ne razume dovoljno jezik koji koristi. Primeri su mnogobrojni: upotreba vrednosti promenljive koja nije inicijalizovana i nema podrazumevanu vrednost, pristup elementu van granica rezervisane memorije za niz, pokušaj upotrebe predefinisanog metoda koji u baznoj klasi nije označen kao virtuelan, pogrešno shvatanje prioriteta među mnogobrojnim operatorima.

Nijedan programski jezik ne može da garantuje da će se konačni izvršni kôd ponašati tačno kako je programer zamislio. Postoje brojni problemi koji se mogu pojaviti sa programima napisanim u bilo kom jeziku, ali u ovom radu najveći fokus je na jeziku C++. Značaj jezika C++ raste i u upotrebi u sigurnosno kritičnim

sistemima kako zbog svoje fleksibilnosti tako i zbog obimne podrške i potencijala prenosivosti na široki spektar hardvera [31].

Programski jezik C++ je vrlo izražajan i u njemu se može napisati uredno strukturiran i čitljiv kôd. Međutim, podjednako je lako napisati i vrlo nejasan kôd. Neke pojedinosti sintakse jezika C++ mogu učiniti da jednostavne greške prilikom programiranja i dalje daju validan kôd i da greške ostanu neprimetne. To mogu biti upotrebe promenljive pogrešnog tipa koji je implicitno konvertovan, sakrivanje promenljivih sa deklaracijom nove promeljive sa istim nazivom, upotreba neinicijalizovane promenljive, upotreba pogrešnog operatora (npr. '=' umesto '==').

U slučaju prioriteta operatora, programski jezik C++ ima precizno definisana pravila i uzorak greške je moguće jasno utvrditi. Međutim, jezik može imati svojstva koja nisu u potpunosti definisana ili su dvosmislena. Tako, na primer, u C++ standardu ISO/IEC 14882:2017 [24] se mogu naći različita objašnjenja koja promenljiva *i* se zapravo koristi u narednom kodu:

```
const int i = -1;
namespace T {
  namespace N { const int i = 1; }
  namespace M {
    using namespace N;
    int a[i];
  }
}
```

Postoje dve deklarisane promenljive sa istim identifikatorom *i*. Puna kvalifikovana imena ovih promenljivih su *::i* i *T::N::i*. Prema poglavlju 6.4.1 *[basic.lookup.unqual]*[1] pretraga za identifikator *i* pronalazi *T::N::i* i zaustavlja se. Međutim, prema poglavlju 6.3.10 *[basic.scope.hiding]* pojava *T::N::i* unutar *namespace T* ne sakriva *::i* pa su obe deklaracije promenljive *i* vidljive prilikom deklaracije za *a*. Ovo dalje može prouzrokovati situaciju gde programer ima jedno tumačenje, dok kompilator implementira drugačije shvatanje.

Ponekad implementacije svesno odstupaju od standarda ili ne implementiraju određena svojstva. Tako, na primer, kompilator GCC, do verzije 12.2 (koja je poslednja stabila verzija u trenutku pisanja rada), ne podržava predlog N3664 [25] koji je prihvaćen i predstavlja dopunu u poglavlju 5.3.4 *[expr.new]* za paragrafe 8,

[1]Oznaka *[basic.lookup.unqual]* predstavlja simbolično ime za poglavlje 6.4.1. Redosled poglavlja se može promeniti od jedne verzije standarda do druge, a simbolični nazivi se koriste kako ne bi došlo do zabune usled različitih brojeva poglavlja. Tako je ovo isto poglavlje blio pod brojem 3.4.1 u standardu ISO/IEC 14882:2014 [23].

10, 11, 12 standarda ISO/IEC 14882:2014 [23]. Izmena se, između ostalog, odnosi na mogućnost obavljanja optimizacije koja uklanja upotrebu operatora *new* (pod određenim uslovima) u narednom kodu:

```
int test() {
    int result = 0;
    for (auto i = 0; i < 1000000; ++i) {
        result += (new int() != nullptr);
    }
    return result;
}
```

Ova izmena zahteva od implementacije jezika, kada je u pitanju operator *new*, da dostavi upotrebljivu memoriju, a ne i da ima određen redosled izvršavanja poziva za alokaciju memorije. Clang je imao mogućnost da izvrši ovu opcionu optimizaciju i pre nego što je obavljena dopuna standarda, dok neki prevodioci kao što su GCC ili Intel C++ ovo smatraju takođe opcionim svojstvom koje ne implementiraju [8, 9].

Još jedan prihvaćen predlog koji je često izostavljen je N2670 [20] koji opisuje mehanizme za sakupljače otpadaka. Usled dostupnosti pametnih pokazivača u jeziku C++, ovo svojstvo retko ko zahteva i nijedna od poznatijih implementacija prevodilaca ga ne sadrži. Primer jedne retke implmentacije je *BoehmGC* [18] koja implementira raniji predlog N2310 [19].

Greške koje se događaju tokom izvršavanja koda je znatno teže uočiti. Pogotovo u jeziku kao što je C++ koji ima slabije mehanizme detekcije grešaka. Takvi mehanizmi obično se moraju ručno implementirati u kodu. To mogu biti greške koje se javljaju za određene ulazne podatke kao što je deljenje nulom ili prekoračenje prilikom sabiranja ili množenja.

## 2.2 Standardi kodiranja

Jedan od razloga zbog kojeg je i pored svih ranije spomenutih mana jezik C++ ostao popularan i za sigurnosno kritične sisteme je postajanje standarada kodiranja. Glavna ideja iza standarda kodiranja je da pruže skup pravila koja usmeravaju ka bezbednom, pouzdanom, proverljivom i lako održivom kodu [29, 6]. Ponekad se standardi pišu za određenu svrhu ili okruženje, što znači da ne može postojati jedan standard kodiranja za sve. Kompanija ili grana industrije može imati svoj standard koji odgovara njihovim potrebama.

Dobar standard kodiranja treba da oslikava isprobana i proverena iskustva zajednice. Treba da sadrži dokazane idiome zasnovane na iskustvu i temeljnom razumevanju jezika. Konkretno, standard za kodiranje trebalo bi da bude čvrsto zasnovan na bogatoj i opširnoj literaturi o razvoju softvera, koja spaja pravila, smernice i najbolje navike i prakse koje bi u suprotnom bile raštrkane po mnogim izvorima [21].

Iako je poželjno imati standard kodiranja, postoje i primeri standarda ili skupa smernica koji definišu pravila koja se u praksi pokazuju kao loše rešenje:

- Primena standarda kodiranja ili praksi iz jednog jezika u drugom kada to nije prikladno [15, 35]. Na primer, deklarisanje svih promenljivih na početku funkcije u jeziku C++ često nema smisla jer se time utiče na opseg i životni vek same promenljive.

- Pisanje komentara za svaku promenljivu, granu ili neki drugi odabrani konstrukt jezika [34, 15, 35]. Za jednostavan kôd ovo predstavlja smetnju, a usled čestih izmena može se napraviti propust gde komentari ostanu zastareli i netačni. Uvek je bolje napisati jasan i čitljiv kôd ukoliko je to moguće, a komentare koristiti kao poslednju meru.

To govori da je nekada bolje nemati standard nego imati loš standard [33, 21]. Loši standardi su napisani od strane ljudi koji ne poznaju jezik C++ dovoljno, nisu dovoljno stručni ili ne razumeju razvoj softvera. Loš standard brzo gubi na kredibilitetu i u najboljem slučaju će biti ignorisan od strane programera koji se ne slažu sa njegovim pravilima [21]. U najgorem slučaju će biti primenjen.

Ukoliko se ne sastavi skup razumnih pravila, programeri ih mogu odbiti i pokušati da uvedu svoja zapažanja i navike. Jedna česta greška je pravljenje strogih standarda. Preveliko sitničarenje i uverenje da je veliki broj zabrana bolji, ne dovode do kvalitetnih rešenja. Primer protiv zabrana je da poneka nepoželjna svojstva jezika postoje kako bi izbegli upotrebu još lošijih svojstava.

Takođe treba obratiti pažnju na zastarele standarde ili na pokušaje definisanja novih standarda od strane ljudi sa zastarelim znanjem jezika. Nešto što je bila dobra praksa godinama unazad ne mora nužno biti i danas. Ovo može izazvati i nepoverenje prema standardima kodiranja.

Primer standarda koji je zastareo može biti upravo standard MISRA C++ 2008. Iako zastareo, ovaj standard je važan jer on predstavlja osnovu za standard AUTOSAR C++ 14 koji ga ažurira i proširuje novim pravilima.

## 2.3 Standard MISRA

Konzorcijum MISRA (eng. *Motor Industry Software Reliability Association*) je 1994. godine objavio svoje „Razvojne smernice za softver za vozila" (eng. *Development Guidelines for Vehicle Based Software*) [38]. One opisuju pun skup mera koje treba koristiti prilikom razvoja ugrađenog softvera koji se odnosi na bezbednost za sisteme vozila (eng. *safety-related embedded software development for vehicle systems*.

Smernice traže od korisnika radno znanje relevantnih softverskih praksi kao i da budu svesni kako postojećih standarda tako i onih u nastajanju koji se odnose na razvoj softvera. Namera ovih smernica nije da budu strogo propisujuće niti zabranjujuće. Određena doza fleksibilnosti je dopuštena programerima i na njima je da odluče koji pristup najviše odgovara cilju koji žele da postignu. Zbog toga se u dokumentima dosta ćešće upotrebljava izraz „trebalo bi" umesto „mora". Samim tim je i definisano više nivoa integriteta gde svaki naredni zahteva strožije tehnike i aktivnosti, a sa njima raste i značaj uticaja relevantnih preporuka. Između ostalog, razmatraju se jezik i kompilator, tj. da li se koristi ceo jezik, ili neki njegov podskup, a za kompilator da li je proveren i testiran pa čak i da li je sa verifikovanom sintaksom i semantikom. Nivoi verifikacije i validacije mogu se zadržati na testovima, a može se zahtevati i da sav alat bude formalno verifikovan.

Smernice MISRA su zasnovane na osnovnim principima povezanim sa projektovanjem sistema vezanih za bezbednost (eng. *safety-related systems*[2]). [38]:

- Bezbednost mora biti primetna i prisutna.
- Za svaki faktor rizika potrebno je dostaviti informacije i dokaze o pouzdanosti sistema.
- Funkcionalnost, pouzdanost i bezbednost softvera moraju biti ugrađeni kao i kvalitet, a ne naknadno dodati.
- Zahtevi za bezbednost ljudi i sigurnost vlasništva nekada mogu biti u konfliktu. Bezbednost ljudi mora imati prioritet.
- Dizajn sistema trebalo bi da razmatra i nasumične (eng. *random*) i sistematske (eng. *systematic*) greške. Ukoliko govorimo o softverskom sistemu, svaka gre-

[2]Sistemi vezani za bezbednost su dizajnirani da implementiraju sigurnosne funkcije i upravljaju rizicima koji mogu nastati usled grešaka. Rizici mogu biti povrede ljudi, materijalna šteta ili šteta po okolinu [14].

ška je sistematska greška. Nasumične greške se javljaju usled fizičkih kvarova hardvera.

- Neophodno je demonstrirati pouzdanost umesto oslanjanje na odsustvo grešaka.
- Bezbednosna razmatranja trebalo bi primeniti pri dizajnu, proizvodnji, delovanju, opsluživanju i odstranjivanju proizvoda.

Cilj standarda MISRA nije promovisanje upotrebe programskog jezika C++ već njegove bezbedne upotrebe. Dosta je napisano o prednostima i manama raznih jezika, standard MISRA objedinjuje veliki broj tih pravila u jednom dokumentu. Takođe je stavljen akcenat na upotrebu statičkih alata za proveru saglasnosti sa standardom. Usled velikog broja pravila, ne može se od programera očekivati da zapamti i razmatra sve smernice istovremeno, pa je postojanje takvih alata poželjno i neophodno za bilo koje razumne upotrebe u praksi.

## Opseg standarda MISRA

Standard MISRA C++ 2008 [31] je napisan za C++ verziju 03, odnosno za standard ISO/IEC 14882:2003 [22] i odnosi se na izbor jezika iz skupa izbora koji je određen nivoom integriteta [38]. Vrlo je česta praksa da se izabere podskup nekog jezika za koji verujemo da je proveren i bezbedan, a pogotovo u automobilskoj, avionskoj, nuklearnoj ili odbrambenoj industriji. Takođe izbor potpuno verifikovanih kompilatora je vrlo ograničen zbog njihove velike složenosti pa čak i tad ne implmentiraju ceo jezik. Primer je kompilator CompCert [11] koji koristi podskup jezika ISO C99.

Propisi koji se odnose na stil kodiranja i metrike su poprilično subjektivni. U tim situacijama MISRA ne daje konkretan odgovor. Ono što ipak zahteva je da se napravi odluka koja će se konzistentno primenjivati. Pravila koje se odnose na pisanje dokumentacije, određivanje imena datoteka ili promenljivih na osnovu njihovog sadržaja ili svrhe su neki od primera koji sprečavaju pravljenje alata koji bi statičkom analizom u potpunosti implementirao provere saglasnosti sa traženim standardom.

Kada su u pitanju biblioteke, bilo sistemske bilo korisničke, standard MISRA preferira da je i u njima u potpunosti ispoštovan skup zadatih pravila. Ovo, međutim, nije uvek moguće ispoštovati. Ono što je poželjno pokazati je da odstupanje od pojedinih pravila ne predstavlja značajan sigurnosni rizik. Primeri pravila od kojih

standard MISRA u određenim slučajevima dopušta odstupanja su: svaka definisana funkcija mora biti upotrebljena barem jednom ili svaka šablonska funkcija mora biti instancirana barem jednom.

Svaki automatsko generisan C++ kôd od strane nekog alata takođe mora poštovati standard MISRA. Ukoliko alat ne poštuje neko pravilo potrebno je izmeniti ga da generiše kôd po tom pravilu. Ako se koristi takav alat neophodno je da on prođe kroz odgovarajuću validaciju.

## Upotreba standarda MISRA

Tokom faze kodiranja, upotreba podskupa nekog jezika kao i poštovanje izabranih pravila je jedno od svojstava dobre prakse. Ovaj izbor gubi na značaju ukoliko nije u skladu i sa drugim odlukama koje je potrebno doneti za fazu kodiranja, a to su: obuka, stil kodiranja, izbor kompilatora i njegova validacija, validacija alata za proveru saglasnosti sa standardom, metrike, pokrivenost testovima. Sve donete odluke kao i razlozi zbog kojih su donete moraju biti dokumentovane.

Pre nego što se može razviti kôd koji se pridržava standarda kodiranja potrebno je ispuniti i sledeće zahteve:

- *Potrebno je proizvesti matricu saglasnosti koja govori kako se svako pravilo proverava* (videti tabelu 2.1). Matrica služi da dâ pregledan opis ponuđenih alata i njihovih mogućnosti. Za svako pravilo se navodi koji alat i na koji način proverava saglasnost sa pravilom. Ukoliko ni jedan alat ne pruža mogućnost provere nekog pravila potrebno je obaviti ručni pregled.

- *Potrebno je proizvesti i proceduru odstupanja od pravila.* Nije uvek moguće pridržavati se standarda. Ukoliko se piše deo koda koji mora da komunicira sa nekim hardverom tada je neizbežno upotrebiti procedure i ekstenzije koje nisu uvek napisane po standardu. Kako programeri ne bi svojevoljno mogli da odstupaju od standarda kada to njima odgovara potrebno je uvesti formalnu proceduru po kojoj se dobija takvo dopuštenje. Svako odstupanje mora biti opravdano kako sa aspekta neophodnosti tako i sigurnosti. Odstupanje može biti za jedno pojavljivanje u jednoj datoteci ili za sistematsku upotrebu nekog konstrukta u odgovarajućim okolnostima. Samim tim mogu se podeliti u dve kategorije.

  - Projektna odstupanja koja se odnose na oslabljivanje ili uklanjanje uslova nekog pravila. Obično se ugovaraju na početku projekta, odnosno pre početka razvojne faze.
  - Specifična odstupanja koja se odnose na jedan slučaj odstupanja od pravila i obično se javljaju usled okolnosti na koje se naiđe tokom razvojne faze projekta.

- *Potrebno je formalizovati postupke upotrebe sistema za proveru kvaliteta.* Opis upotrebljenog podskupa jezika tj. skupa pravila, načini korišćenja obezbeđenih alata moraju biti formalno opisani kao deo sistema za upravljanje kvalitetom. Kao takvi su podložni i spoljašnjim i unutrašnjim revizijama.

Tabela 2.1: Primer matrice saglasnosti koja opisuje kako se proverava saglasnost sa svakim od pravila standarda za projekat. Na primer, pravilo 0-1-1 se proverava sa prvim kompilatorom koji proizvodi upozorenje *W1* ukoliko kôd nije saglasan sa tim pravilom.

| Broj pravila | Kompilator 1 | Kompilator 2 | Alat 1 | Alat 2 | Ručni pregled |
|---|---|---|---|---|---|
| 0-1-1 | upozorenje W1 | | | | |
| 0-1-2 | | | | | procedura P3 |
| 0-1-3 | | | poruka M1 | | |
| 0-1-4 | | greška E47 | | | |
| 0-1-5 | | | | izveštaj I6 | |
| ... | | | | | |

## Klasifikacija pravila standarda MISRA C++

Pravila u stadnardu MISRA se mogu klasifikovati u tri kategorije prema nivou važnosti: obavezna (eng. *required*), preporučena (eng. *advisory*) i dokumentaciona (eng. *document*). Unutar samih klasa ne postoji dalje klasifikovanje po važnosti i sva pravila koju su označena kao obavezna su među sobom podjednako važna. Isto se odnosi i na preporučena i dokumentaciona pravila.

- Kôd za koji se tvrdi da je napisan po standardu MISRA C++ mora poštovati sva obavezna pravila. Odustpanja su ipak moguća ali moraju biti formalno obrazložena.

- Za odstupanja od preporučenih pravila nije potrebno formalno obrazloženje ali to ne znači da se mogu u potpunosti ignorisati. Treba ih se pridržavati što je više moguće i dokle god je to praktično.
- Dokumentaciona pravila zahtevaju da budu ispoštovana kao i obavezna i ne dozvoljavaju odstupanja.

Pravila su organizovana u sekcije prema brojevima iz C++ standarda ISO/IEC 14882:2003 [22]. U slučaju da je pravilo prikladno za više sekcija odabrana je prva.

Pravila mogu imati redudantnosti, tj. mogu sadržati određena preklapanja pa čak i situacije kada je jedno pravilo podskup drugog. Razlog za ovo je da se nekada očekuje da kôd zahteva odstupanja od pravila i potrebno je pokriti određen podskup uslova umesto celog pravila. Tako na primer postoji pravilo koje opisuje šta je dobro formirana naredba *switch*, ali postoji i pravilo koje zahteva da klauza *default* uvek bude poslednja, što se takođe podrazumeva za prvo pravilo. Kada se odstupa od pravila za dobro formirane naredbe *switch*, onda pravilo za klauze *default* postaje relevantno jer tada njegovi uslovi više nisu redudantni.

Pravila su opisana u sledećem formatu:

```
Pravilo <broj> (<kategorija>) <naslovni tekst>
```

<***broj***> predstavlja jedinstveni identifikator koji se sastoji iz tri povezana broja u obliku *aa-bb-cc* gde prva dva ukazuju na sekciju iz standarda ISO/IEC 14882:2003.

<***kategorija***> predstavlja jedan od tri nivoa važnosti.

<***naslovni tekst***> predstavlja sam tekst pravila.

Uz svako pravilo je takođe dato kratko obrazloženje zašto je ono korisno. Još jedna bitna stavka koju neka pravila mogu imati je opis izuzetaka i on predstavlja dopunu ili modifikaciju teksta pravila tj. *naslovnog teksta*.

Standard MISRA C++ 2008 sadrži 228 pravila kodiranja. Od toga su 198 obavezna, 18 preporučena i 12 dokumentaciona pravila.

## 2.4 Standard AUTOSAR

Standard AUTOSAR C++ 14 [5] predstavlja dopunu i proširenje standarda MISRA C++ 2008. Nastao je iz zajednice AUTOSAR koja je formirana 2003. go-

dine kao udruženje raznih proizvođača iz automobilske industrije [4]. Sve smernice, principi i motivacije koje su se odnosile na formiranje standarda MISRA važe i za standard AUTOSAR.

Razlog za novi standard je taj što je standard MISRA zastareo jer je pisan za C++03. Namera AUTOSAR-a je da pokrije i nova svojstva iz verzija C++11 i C++14. Tako je u martu 2017. objavljena prva verzija standarda AUTOSAR C++ 14, verzija 17-03. Ona propisuje 342 pravila, od toga je:

- 154 pravila usvojeno iz MISRA C++ 2008 bez izmena (67% tog standarda).
- 131 pravilo zasnovano na postojećim C++ standardima.
- 57 pravila zasnovano na istraživanju ili drugoj literaturi.

U međuvremenu je i ovaj standard dopunjen. Prvo za verzijom 17-10, a potom i 18-03 i sada sadrži 402 pravila:

- 148 pravila je usvojeno iz MISRA C++ 2008 bez izmena (64% tog standarda).
- 195 pravila su zasnovana na postojećim C++ standardima.
- 59 pravila je zasnovano na istraživanju ili drugoj literaturi.

## Klasifikacija pravila standarda AUTOSAR C++

Klasifikacija po nivou važnosti se deli na obavezna i preporučena pravila. Grupa dokumentacionih pravila je izostavljena, a pravila iz te grupe su spojena sa grupom obaveznih. Jedina razlika između grupe dokumentacionih i obaveznih je ta što dokumentaciona ne dopuštaju odsutpanja ni u kom slučaju. Kako je za odstupanje potrebno dati formalno obrazloženje to pored ostalog iziskuje i neku vrstu dokumentacije, što je upravo ono što se i traži od pravila. Drugim rečima, ako se želi izbeći pisanje dokumentacije to se mora obrazložiti sa drugom vrstom dokumentacije.

AUTOSAR nudi dodatne kriterijume po kojem su klasifikovana pravila. Jedan od njih koji je od velikog značaja je klasifikacija po primenljivosti statičke analize:

- Automatizovana (eng. *automated*) pravila je moguće automatizovati u smislu statičke analize.
- Delimično automatizovana (eng. *partially automated*) pravila je takođe moguće automatizovati ali pomoću odgovarajućih heuristika, i obično mogu da

pokriju samo najčešće slučajeve upotrebe. Mogu poslužiti kao podrška ručnom pregledu koda.

- Neautomatizovana (eng. *non-automated*) pravila se ne mogu pouzdano detektovati statičkom analizom i zahtevaju ručne preglede koda.

Poslednji način klasifikovanja je prema oblasti primene odnosno cilju kome su namenjena:

- Implementaciona (eng. *implementation*) se odnose na samu implementaciju kao što je sam kôd, dizajn softvera, arhitektura.
- Verifikaciona (eng. *verification*) se odnose na verifikacione aktivnosti poput pregleda koda, analize i testiranja koda.
- Infrastrukturna (eng. *infrastructure*) se odnose na operativni sistem i hardver.
- Iskorišćeni alat (eng. *toolchain*) su pravila koja se odnose na pojedine specifičnosti pretprocesora, kompilatora, linkera i biblioteka.

Pravila su i dalje opisana u istom formatu kao za standard MISRA sa par dopuna:

***<broj>*** ima prefix *<M>* ili *<A>* koji označava da li je pravilo preuzeto iz standarda MISRA C++ 2008 ili je u pitanju novo pravilo koje je uveo standard AUTOSAR C++ 14 respektivno.

***<kategorija>*** pored osnovne klasifikacije po nivou važnosti, sadrži i kojem od dva nova načina klasifikovanja pripada određeno pravilo.

# Glava 3

# Kompilatorska infrastruktura LLVM

U ovom poglavlju je predstavljena istorija projekta LLVM kao i sastavni elementi ovog projekta sa kratkim pregledom strukture kompilatora. Potom su opisani detalji potprojekta Clang koji predstavlja prednji deo kompilatora za programske jezike C i C++. Posebna pažnja je posvećena elementima Clang-a koji su od značaja u ovom radu.

## 3.1 Istorija projekta LLVM

Projekat LLVM je započet 2000. godine na Univerzitetu u Ilinoisu [36]. Originalno je bio istraživački projekat za ispitivanje tehnika kompilacije nad kodom u SSA formi (eng. *Single Static Assigment*), što je i bilo okarakterisano nazivom *Virtuelna mašina niskog nivoa* (eng. *Low Level Virtual Machine*). Projekat je samostalno započeo Kris Latner. Kris ubrzo dobija podršku od svog mentora Vikrama Advea koji koristi LLVM na svojim kursevima o kompilatorima. Tako su studenti prvi doprineli razvoju projekta sa svojim izveštajima o greškama i bagovima [7].

Jasni i jednostavni koncepti u računarskoj teoriji nekada podrazumevaju veliki broj implementacionionih detalja što ih čini značajno težim za razumevanje. Pametan dizajn sa jakom apstrakcijom je ključni element koji omogućava da projekat bude jasan ljudima koji ga razvijaju od najvišeg nivoa pa sve do najsitnijih detalja [7]. Ovo važi i za kompilatore. Pre LLVM-a, GCC je bio jedan od retkih kompilatora otvorenog koda. Reč je o projektu koji je star više od 30 godina i na kojem je radilo više generacija programera koji nekada mogu imati različite poglede i mišljenja o istim pojmovima. Sve ovo otežava održavanje i razumevanje projekta novim programerima.

LLVM se javlja kako privlačnija alternativa mladim i radoznalim programerima u odnosu na GCC. Ovo je najbolje oslikano u velikom broju naučnih radova koji se oslanjaju na LLVM [27, 28]. Glavni razlog je modularni dizajn LLVM-a koji je napisan u jeziku C++ nasuprot GCC-a koji je pisan u C-u i ima monolitnu strukturu. LLVM takođe bolje oslikava teorijske koncepte o kompilatorima koji su prisutni u naučnoj literaturi.

Projekat je originalno bio pod licencom slobodnog koda *University of Illinois/NCSA Open Source License* koja za razliku od GCC-ove GLP licence nije kopileft (eng. *copyleft*) [37]. Ovo je dopuštalo da se projekti izvedeni iz LLVM-a mogu staviti pod drugu licencu bez ograničenja koje kopileft nosi sa sobom, zbog čega je LLVM privukao i interesovanje industrije. Sve ovo je doprinelo da 2012. godine prestižna nagrada *ACM Software System Award* bude dodeljena projektu LLVM kao softveru trajnog uticaja koji je doprineo razvoju nauke [1]. Od verzije 9.0.0 LLVM je pod licencom *Apache License, Version 2.0*, koja je vrlo slična prethodnoj, sa određenim izuzecima specifičnim za LLVM [37].

Sa porastom interesovanja mnogih kompanija sa različitim potrebama, brzo je porastao i opseg projekta. LLVM nije samo podržavao dodatne jezike sa kojima zna da rukuje ili dodatne arhitekture za koje je znao da generiše kôd, već je obuhvatao i dodatne biblioteke i alate. Samim tim je prerastao iz akademskog projekta u programski okvir (eng. *framework*) koji se koristi u komercijalnim proizvodima [7]. Takođe je i odbačen stari naziv projekta *Virtuelna mašina niskog nivoa* koji više nije bio prikladan i danas se koristi akronim LLVM kao pun naziv.

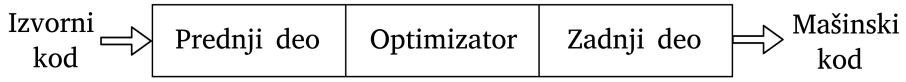
## 3.2 Struktura projekta LLVM

Projekat LLVM sastoji se iz više biblioteka i alata koji zajedno čine veliku kompilatorsku infrastrukturu. Osnovna filozofija LLVM-a je da je „svaki deo neka biblioteka" i veliki deo koda je ponovno upotrebljiv [7]. Projekat je u potpunosti napisan u jeziku C++, a od verzije LLVM-a 9.0.0 to podrazumeva verziju jezika C++14.

### Struktura tipičnog kompilatora

Tipičan kompilator se sastoji iz prednjeg dela (eng. *front end*), središnjeg dela (eng. *middle end*) koji se nekada zove i optimizator i zadnjeg dela (eng. *backend*) (slika 3.1). Prednji deo parsira izvorni kôd, proverava da li sadrži greške i izgrađuje

apstraktno sintaksno stablo (eng. *Abstract Syntax Tree*). Dobijeno stablo se dalje, po potrebi, može prevesti u neku novu reprezentaciju koja je pogodnija za obavljanje optimizacija.




Slika 3.1: Struktura tipičnog kompilatora

Središnji deo se sastoji od optimizatora čija je odgovornost da obavi široki spektar transformacija sa ciljem unapređenja vremena izvršavanja koda. To se obavlja uklanjanjem suvišnih izračunavanja. Optimizatori teže da budu nezavisni i od izvornog i od ciljnog jezika. Zadnji deo služi da optimizovani kôd zatim preslika u odgovarajuće instrukcije ciljne arhitekture. Potrebno je pravilno iskoristiti specifična svojstva izabrane arhitekture jer cilj nije samo generisati ispravan već i efikasan kôd. Zadnji deo čine procesi koji se bave izborom instrukcija, alokacijom registara i raspoređivanjem instrukcija [10].

## Potprojekti kompilatorske infrastrukture LLVM

LLVM je podeljen na više potprojekata i biblioteka koji implementiraju različite delove kompilatorske infrastrukture. Zvanični potprojekti LLVM-a su: jezgro LLVM, Clang, LLDB, libc++, libc++ ABI, OpenMP, polly, libclc, klee, LLD [37].

*Jezgro LLVM* sadrži biblioteke koje pružaju moderan optimizator programskog koda koji je nezavisan od izvornog jezika i ciljne arhitekture. Glavni razlog ove nezavisnosti je što biblioteke rade nad međureprezentacijom LLVM-a ili IR-om (eng. *intermediate representation*). LLVM IR je jezik u formi SSA koju odlikuju dve bitne karakteristike: kôd se sastoji od troadresnih instrukcija i ima beskonačan broj registara. Ovo nije jedina interna reprezentacija koju LLVM koristi ali jeste ključna i predstavlja osnovu od koje je projekat i započet. Postoje optimizacije koje se vrše i nad drugim internim međureprezentacijama ali su manje brojne i specifične za samu reprezentaciju. IR i njene optimizacije pripadaju **središnjem delu** kompilatora.

Drugi bitan deo jezgra LLVM se odnosi na generisanje izvršnog koda od zadatog IR-a. Podržan je veliki broj popularnih arhitektura kao što: X86, Mips, AMDGPU, ARM, PowerPC i AARCH64 između ostalih. Ove biblioteke predstavljaju **zadnji deo** kompilatora.

**Clang** je dugo bio jedini **prednji deo** u sastavu projekta LLVM. On, međutim, nije deo jezgra LLVM već se vodi kao zaseban potprojekat. Podržava jezike C, C++ i Objective-C. Clang se oslanja na jezgro LLVM i nije ga moguće koristiti samostalno.

Glavni fokus ovog rada je upravo na potprojektu Clang i o ostatku projekta se spominju osnovni detalji. Ostali potprojekti predstavljaju korisne alate, ali nisu od značaja u ovom radu. Među njima se, između ostalih, nalazi **LLD** koji se može iskoristiti kao zamena za sistemski linker i **LLDB** debager koji se oslanja na biblioteke jezgra LLVM i Clang-a. Potprojekte koje takođe treba istaći su i **libc++** i **libc++ABI** koji zajedno čine implementaciju standardne biblioteke za jezik C++.

## 3.3 Potprojekat Clang

Razvoj novog prednjeg dela je nastao iz potrebe za kompilatorom koji pruža bolju dijagnostiku, integraciju sa razvojnim okruženjima i licencom koja je saglasna sa komercijalnim proizvodima. Takođe je bilo poželjno da bude i lak za razvoj i održavanje.

Naziv Clang, slično kako i LLVM, može da izazove zabunu zbog svojih višestrukih značenja. Kada se govori o Clang-u, referiše se na jedan od narednih pojmova:

- **Prednji deo** koji je implementiran u Clang-ovim bibliotekama.
- **Rukovalac (*eng. driver*) kompilatora** koji se koristi kao naredba *clang*. Implementiran unutar Clang-ove biblioteke *Driver*.
- **Stvarni kompilator** koje se koristi uz pomoć naredbe `clang -cc1`[1]. Implementacija kompilatora koristi Clang-ove bibiloteke i jezgro LLVM kako bi implementirao srednji i zadnji deo kompalera. Takođe sadrži i integrisani asembler.

Tokom ranih faza razvoja Clang-a, linker LLD iz projekta LLVM nije bio stabilan [7]. Iz tog razloga se kompilacija uz pomoć `clang -cc1` zaustavlja kod objektnog koda. Kako bi se proizveo izvršni kôd, potrebno je pozvati linker. Neugodnost upotrebe dvostrukih naredbi prilikom svakog obavljanja potpune kompilacije se može izbeći upotrebom rukovaoca Clang tj. naredbom `clang`. Rukovalac Clang upravlja celim

[1]Naziv *cc1* potiče iz kompilatora GCC gde se odnosio na potprogram koji je vršio prevođenje počevši od pretprocesora i završno sa asemblerom.

procesom prevođenja i prvo poziva kompilator `clang -cc1` pa potom sistemski linker kao podprocese rukovaoca[2]. Precizni pozivi programa sa svim zadatim opcijama koje rukovalac koristi se mogu saznati upotrebom opcije `-###`. Primer upotrebe ove opcije daje sledeći ispis (lista dodatnih parametara je skraćena):

```
$ clang test.cpp -o test -###
clang version 10.0.0
Target: x86_64-w64-windows-gnu
Thread model: posix
 "clang" "-cc1" "-triple" "x86_64-unknown-linux-gnu" "-emit-obj"
                "-internal-isystem" "/usr/include/c++/9" "-x" "c++"
                (... preostali parametri ...)
                "-o" ".../test-25b959.o" "test.cpp"
 "ld" "-z" "relro" "--hash-style=gnu" "--eh-frame-hdr" "-m" "elf_x86_64"
      "-L/usr/lib/gcc/x86_64-linux-gnu/9" "-lstdc++"
      (... preostali parametri ...)
      "-o" "test" ".../test-25b959.o"
```

Iako se *cc1* koristi tako što se upotrebi dodatni argument prilikom poziva rukovaoca, on se smatra zasebanim alatom. Značajan broj argumenata koje rukovalac Clang prosledi kompilatoru i linkeru nisu podrazumevane vrednosti, već pokušava da ih odredi na osnovu okruženja i sistema na kojem se pokreće. Neki od tih argumenata se odnose na ciljnu arhitekturu ili na putanje do zaglavlja sistemskih biblioteka. Van zajednice LLVM, rukovalac Clang je taj koji je poznatiji i vrlo često upravo on dobija pohvale i kritike čak i za delove za koje nije direktno odgovoran. Na primer, komentari koji su upućeni na račun optimizacija koje se obavljaju u srednjem delu kada se upotrebi naredba `clang -O3` se često prisvajaju Clang-u iako je to deo jezgra LLVM.

Jedno od bitnih svojstava rukovaoca Clang je i da je kompatibilan sa kompilatorom GCC. Clang podržava sve dodatne zastavice (eng. *flag*) za kompilaciju koji su dostupni i u GCC-u pa se prelazak sa GCC-a na Clang može jednostavno obaviti. Glavni nedostaci u odnosu na GCC pored toga da ne podržava programske jezike van C porodice je i da podržava manji broj ciljnih arhitektura. Iako je ovo više odlika jezgra LLVM, ono je svakako neophodno za Clang i utiče na način na koji se ispituju mogućnosti Clang-a kao potpunog kompilatora.

[2]Od verzije 11.0.0 upotrebom opcija `-fintegrated-cc1` i `-fno-integrated-cc1` moguće je kontrolisati da li se *cc1* izvršava kao podproces ili kao sastavni deo rukovaoca.

## Mehanizmi dijagnostike

Značajna odlika Clang-a je i njegova izražajna dijagnostika. Kada je u pitanju alat komandne linije kao što je kompilator to se odnosi na to da dijagnostika, tj. poruke o greškama i upozorenja, budu što je moguće korisnije.

Na sledećem primeru je prikazano kako izgleda jedna dijagnostička poruka. Za naredni kôd:

```
int a
```

dobija se informacija o grešci u sledećem formatu:

```
..\test.cpp:1:6: error: expected ';' after top level declarator
int a
     ^
     ;
1 error generated.
```

Diajgnostika sa sobom nosi više informacija. Prva linija je u formatu **datoteka:lokacija:nivo:poruka** i njena polja imaju sledeća značenja:

- **datoteka** iz koje je potekla greška, predstavljena relativnom putanjom,
- **lokacija** u izvornom kodu u obliku *linija:kolona* koja ukazuje na tačno mesto gde je kompilator detektovao prekršaj,
- **nivo** dijagnostike koji može biti greška (*error*), upozorenje (*warning*), beleška (*note*),
- **tekst poruke** koji opisuje vrstu prekršaja.

Ispod toga stoji prepisana linija iz izvornog koda koja odgovara broju koji predstavlja red u lokaciji dijagnostike, dok u liniji ispod stoji oznaka (`^`) koja pokazuje na poziciju (kolonu) u liniji iznad za koju je prijavljena dijagnostička poruka. U nekim situacijama Clang može predložiti i rešenje za eliminisanje konkretne greške ili upozorenja u formi nagoveštaja za popravku (eng. *FixItHint*). U primeru je prikazano da je ta preporuka upravo karakter (`;`) koji nedostaje. U Clang-u klasa *DiagnosticsEngine* definiše interfejs za upravljanje sa dijagnostičkim porukama.

## Apstraktno sintaksno stablo

Apstraktno sintaksno stablo (eng. *Abstract syntax tree*, *AST*) predstavlja hijerarhijsku sintaksnu strukturu koja odgovara izvornom kodu [2]. Za razliku od izvornog

koda, oslobođeno je raznih detalja sintakse, kao što su zarezi, zagrade, nazubljivanja i formatiranje koda pa je samim tim pogodnije za automatsku obradu od strane optimizatora. Ova stabla su rezultat rada prednjeg dela kompilatora, konkretno leksera i parsera.

Zbog svoje strukture, apstraktno sintaksno stablo koje Clang generiše je pogodano za upotrebu primarno za dve sledeće svrhe. Prva je upotreba u semantičkoj analizi koja služi kao validacija stabla. Pravila standarda AUTOSAR koja se opisuju u ovom radu se mogu okarakterisati kao semantičke provere. Provere saglasnosti koda sa ovim pravilima su implementirane na sličan način koristeći slične metode i mehanizme koje se koriste u semantičkoj analizi. Druga primarna upotreba apstraktnog sintaksnog stabla je prevođenje u neki drugi jezik. Konkretno Clang prevodi apstraktno sintaksno stablo u međureprezentaciju LLVM-a.

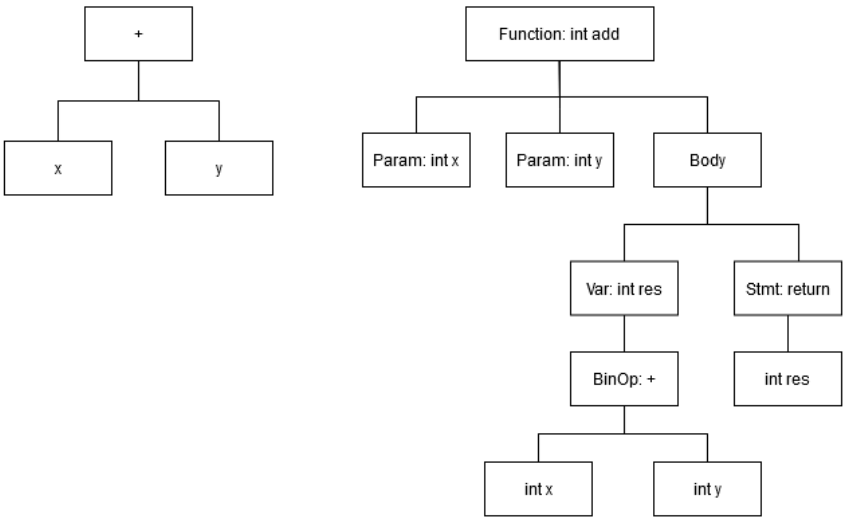
Elementi stabla odgovaraju konstruktima jezika iz izvornog koda. Svaka funkcija, naredba i promenljiva se transformiše u odgovarajući element. Ti elementi su povezani na logičan način koji odgovara izvornom kodu. Ovo se najlakše može videti na kratkom primeru:

```
int add(int x, int y) {
  int res = x + y;
  return res;
}
```

Apstraktno sintaskno stablo za navedeni kôd se može predstaviti na razne načine. Pošto je reč o stablu, vizuelno se može predstaviti kao graf. U zavisnosti od potrebe, mogu se apstrahovati različiti detalji iz izvornog koda.

Na slici 3.2 mogu se videti dva različita načina na koji se može predstaviti funkcija iz primera. Prvi način masksimalno eliminiše sve detalje sintakse i fokusira se na rezlutat koji ona prozvodi. Koren stabla je predstavljen čvorom koji odgovara operaciji sabiranja i ima dva potomka koji odgovaraju njegovim operandima. Drugi način zadržava više informacija kao što su nazivi funkcije, argumenata i promenljivih kao i njihove tipove. Takođe sadrži i broj i redosled svih naredbi u telu same funkcije.

Fokus ovog rada je na konkretnoj reprezentaciji koju Clang generiše. U nastavku kada se bude govorilo o apstraktnom sintaksnom stablu, reč je o Clang-ovom apstraktnom sintaksnom stablu. Ova reprezentacija se može videti pomoću opcije `-ast-dump`. Pošto je ovo opcija *cc1* kompilatora potrebno je iskoristiti `-Xclang` opciju koja govori rukovaocu *clang* da je prosledi. Takođe, ukoliko generisanje koda




Slika 3.2: Grafičke reprezentacije apstraktnog sintaksnog stabla za različitim predstavljenim detaljima

nije od interesa, može se upotrebiti opcija `-fsyntax-only` kako bi se kompilator zaustavio nakon što se obavi prednji deo:

```
$ clang -Xclang -ast-dump -fsyntax-only 1.cpp
```

U nastavku je prikazan tekstualni prikaz dobijen spomenutom opcijom `-ast-dump` za primer iznad:

```
1  TranslationUnitDecl 0x56406c1d8ba1 <<invalid sloc>>
2  ... uklonjene razne interne deklaracije ...
3  `-FunctionDecl 0x56406c1d8de8 <1.cpp:1:1, line:4:1> line:1:5 add 'int (int, int)'
4    |-ParmVarDecl 0x56406c1d8c98 <col:9, col:13> col:13 used x 'int'
5    |-ParmVarDecl 0x56406c1d8d10 <col:16, col:20> col:20 used y 'int'
6    `-CompoundStmt 0x56406c1d9068 <col:23, line:4:1>
7      |-DeclStmt 0x56406c1d8ff8 <line:2:3, col:18>
8      | `-VarDecl 0x56406c1d8ef0 <col:3, col:17> col:7 used res 'int' cinit
9      |   `-BinaryOperator 0x56406c1d8fd0 <col:13, col:17> 'int' '+'
10     |     |-ImplicitCastExpr 0x56406c1d8fa0 <col:13> 'int' <LValueToRValue>
11     |     | `-DeclRefExpr 0x56406c1d8f50 <col:13> 'int' lvalue ParmVar
       0x56406c1d8c98 'x' 'int'
12     |     `-ImplicitCastExpr 0x56406c1d8fb8 <col:17> 'int' <LValueToRValue>
13     |       `-DeclRefExpr 0x56406c1d8f78 <col:17> 'int' lvalue ParmVar
       0x56406c1d8d10 'y' 'int'
14     `-ReturnStmt 0x56406c1d9050 <line:3:3, col:10>
15       `-ImplicitCastExpr 0x56406c1d9038 <col:10> 'int' <LValueToRValue>
16         `-DeclRefExpr 0x56406c1d9010 <col:10> 'int' lvalue Var 0x56406c1d8ef0
             'res' 'int'
```

Koren ovog stabla predstavlja celu jedinicu prevođenja odnosno jednu datoteku izvornog koda (.cpp) i predstavlja se čvorom *TranslationUnitDecl*. Njegovi prvi

potomci su interne definicije tipova koje Clang koristi, ali pošto nisu od značaja u ovom radu, ignorišu se. Preostali čvorovi odgovaraju elementima C++-a koji se nalaze u izvornoj datoteci.

U datom primeru, posle zanemarenih internih deklaracija, postoji još jedan čvor koji je direktan potomak korena. On predstavlja celu funkciju *add*, opisanu čvorom *FunctionDecl* i sadrži tri potomka. Prva dva su čvorovi tipa *ParmVarDecl* koji odgovaraju parametrima funkcije $x$ i $y$. Treći potomak je naredba tipa *CompoundStmt* koja odgovara bloku u C++-u odnosno telu ove funkcije. Blok se sastoji od dve naredbe i upravo toliko potomaka sadrži čvor *CompoundStmt*, po jednog za svaku naredbu, *DeclStmt* i *ReturnStmt*.

Tri osnovne vrste klasa koje čine apstraktno sintaksno stablo su deklaracije, naredbe i tipovi. One su opisane baznim klasama *Decl*, *Stmt* i *Type*. Sve tri predstavljaju baznu klasu za veliki broj izvedenih klasa. Kao što se moglo videti na primeru gore, neke od izvedenih klasa od klase *Decl* su *FunctionDecl*, *VarDecl* i *ParmVarDecl* (koja je zapravo izvedena od klase *VarDecl*). Bazna klasa za sve izraze je *Expr* i je ona zapravo izvedena od klase *Stmt*.

Svaka klasa sadrži u sebi razne informacije o čvoru koji opisuje. Neke od njih se mogu videti u ispisu stabla:

```
|   `-BinaryOperator 0x56406c1d8fd0 <col:13, col:17> 'int' '+'
```

Binarni operator je jedna od specijalizacija izraza. Čvor je opisan imenom klase (*BinaryOperator*), jedinstvenom adresom (u heksadecimalnom zapisu), kao i lokacijom u izvornom fajlu (*<col:13, col:17>*). Ove informacije se ispisuju za svaki čvor. Osim njih, za binarni operator imamo ispisan i tip (*int*) kao i vrstu operatora (*+*).

Sve ove informacije se mogu dobiti nekom od metoda ove klase: *getBeginLoc()* i *getEndLoc()* za lokacije, *getOpcode()* za vrstu operatora, *getType()* za tip. Metodi *getLHS()* i *getRHS()* vraćaju čvor koji je potomak i koji odgovara levoj ili desnoj strani izraza.

Podatak o lokaciji govori da je binarni operator predstavljen delom teksta od 13. do 17. kolone. Kako broj reda nije naveden to znači da je isti kao za njegovog direktnog pretka u stablu, što je u ovom slučaju red 2. Konkretno to je sledeći tekst u izvornoj datoteci: `x + y`.

Lokacije se predstavljaju klasom *SourceLocation* koja pokazuje na određeni karakter u datoteci izvornog koda. Čvorovi obično sadrže dve lokacije koje označavaju početak i kraj odgovarajućeg teksta na osnovu kog su isparsirani i generisani. Pošto svaki čvor ima svoju lokaciju, poželjno je da veličina ovog objekata bude minimalna.

Zbog toga klasa *SourceLocation* sadrži samo udaljenost od početka jedinice prevođenja (eng. *offset*). Klasa *SourceManager* služi za učitavanje i keširanje izvornih datoteka i predstavlja interfejs za razne informacije o lokaciji. Pomoću nje i konkretne lokacije može se saznati da li čvor potiče iz datoteke koja je uključena u izvorni kôd pomoću direktive *#include* ili je možda nastao ekspanzijom iz makroa.

Sve informacije o apstraktnom sintaksnom stablu za jednu jedinicu prevođenja se mogu naći u objektu klase *ASTContext*. Ona sadrži pokazivač na koreni čvor, tj. *TransaltionUnitDecl*. Ranije spomenute klase *DiagnosticsEngine* i *SourceManager* se takođe vezuju za jednu jedinicu prevođenja i može im se pristupiti preko instance *ASTContext*-a. Kako bi se obavila provera da li je ispoštovano neko pravilo kodiranja, potrebno je obaviti obilazak celog stabla i obaviti određene provere. Za to su dostupna dva mehanizma: **AST-posetioci** i **AST-uparivači**.

## AST-posetioci

Rekurzivni AST-posetilac (eng. *Recursive AST Visitor*) pruža način da se obiđu svi čvorovi apstraktnog sintaksnog stabla. Podrazumevano se vrši obilazak sa ulaznom obradom (eng. *preorder*), ali moguće je promeniti ga u obilazak sa izlaznom obradom (eng. *postorder*).

Konkretan obilazak se obavlja tako što se nasledi bazna klasa *RecursiveASTVisitor*, a potom se implementiraju odgovarajuće metode koje vrše željene provere u stablu. Posetioci sadrže tri različite grupe metoda: metode obilaska (*Traverse**), metode prolaska kroz hijerarhiju klasa (*WalkUpFrom**) i metode posete (*Visit**).

Metode obilaska služe da započnu rekurzivni obilazak podstabla počevši od zadatog čvora. Ukoliko se pozove takav metod nad korenom celokupnog stabla, odnosno čvorom tipa *TranslationUnitDecl*, u tom slučaju se obavlja obilazak celog stabla. U većini slučajeva to je željeni cilj.

Tri osnovne vrste klasa koje predstavljaju čvorove stabla (*Decl*, *Stmt* i *Type*) nemaju zajedničku baznu klasu pa zbog toga ne postoji ni opšti metod obilaska. Kako bi se započeo obilazak potrebno je znati barem baznu klasu čvora i izabrati odgovarajući metod od tri osnovna: *TraverseDecl()*, *TraverseStmt()* i *TraverseType()*. Metode obilaska postoje i za mnoge druge izvedene klase tako da se mogu koristi i metode poput: *TraverseSwitchStmt()* ili *TraverseNamedDecl()*.

Sledeća grupa su metodi *WalkUpFrom** koji prolaze kroz hijerarhiju klasa počevši od specifične klase prema jednoj od baznih klasa: *Decl*, *Stmt* ili *Type*. Poslednja grupa su metodi posete ili *Visit** koji zapravo vrše obilazak konkretnog čvora.

Ovi metodi su uređeni po nivoima u sledećem redosledu: *Traverse**, *WalkUpFrom**, *Visit**. Metodi iz jednog nivoa mogu pozivati metode iz istog ili narednog nivoa, ali ne i iz prethodnog.

Kada se pozove neki od metoda obilaska npr. *TraverseDecl()* za neki čvor $x$ on zapravo pozva metod obilaska za najužu izvedenu klasu čvora $x$. Taj metod dalje prvo pozove odgovarajući metod *WalkUpFrom** a potom rekurzivno poziva metode *Traverse** za sve potomke čvora $x$.

Metod *WalkUpFrom** za čvor $x$ neke klase prvo pozove metod *WalkUpFrom** za prvu roditeljsku klasu čvora $x$, a potom pozove odgovarajući metod *Visit**. To znači da se metodi *Visit** pozvaju za svaku klasu čvora $x$ u redosledu od bazne klase prema specifičnoj. Na primer, klasa *NamespaceDecl* nasleđuje *NamedDecl* koja nasleđuje *Decl*. U skladu sa time za čvor tipa *NamespaceDecl* pozivi funkcija su u sledećem redosledu: *WalkUpFromNamespaceDecl()*, *WalkUpFromNamedDecl()*, *WalkUpFromDecl()* koji je za baznu klasu i poziva *VisitDecl()*, a potom po povratku u metode iznad pozvaju se *VisitNamedDecl()* i *VisitNamespaceDecl()*. Ovaj način garantuje da su svi pozivi metoda *Visit** za jedan čvor stabla uvek grupisani zajedno i nikada nisu isprepletani sa metodama *Visit** drugih čvorova.

Korisnici najčešće predefinišu metode *Visit**. Metodi *Traverse** i *WalkUpFrom** se obično predefinišu ukoliko je potreban neki složeniji način obilaska stabla od podrazumevanog. Svi metodi imaju povratnu vrednost tipa *bool*. Ukoliko bilo koji metod vrati netačnu vrednost, celokupan obilazak stabla se obustavlja.

## AST-uparivači

AST-uparivač (eng. *AST Matcher*) nam pruža alternativni metod u odnosu na posetioce za prikupljanje informacija iz apstraktnog sintaksnog stabla. Uparivači se konstruišu na osnovu generatorskih funkcija koje predstavljaju vrstu domenskog jezika funkcionalnog tipa sa ciljem definisanja upita nad apstraktnim sintaksnim stablom.

Ukoliko je, na primer, potrebno pronaći sve pozive funkcija može se upotrebiti uparivač *callExpr()* koji se uparuje sa svim čvorovima klase *CallExpr*. Ako je potrebno proveriti i neko dodatno svojstvo kao što je tip povratne vrednosti, može se konstruisati sledeći složeni uparivač:

```
callExpr(hasType(asString("int")))
```

Svaki uparivač ima klasu ili skup klasa čvorova nad kojima se može pozvati. Tako npr. uparivač *callExpr()* se može pozvati nad čvorovima tipa *Stmt* i pripada grupi **osnovnih uparivača** ili uparivača nad čvorovima (eng. *Node Matchers*) što znači da u izrazu za uparivanje može biti početni uparivač.

Grupa **uparivača za obilazak** (eng. *Traversal Matchers*) služi za proveru odnosa između trenutnog čvora i ostalih čvorava u stablu. Uparivač *hasType()* iz primera iznad je jedan takav. Postoji veća grupa klasa nad kojima se on može pozvati, među kojima je i klasa *Expr*, bazna klasa čvora *CallExpr*. U ovoj grupi se nalazi i neki posebni uparivači koji se mogu upariti sa svakim čvorom i mogu se koristiti radi pisanja opštih izraza. Među njima imamo uparivače kao što su *hasAncestor()*, *hasDescendant()*, *forEach()* i mnoge druge.

Poslednja grupa su **sužavajući uparivači** (eng. *Narrowing Matchers*) i služe da provere neko svojstvo trenutno uparenog čvora. Primeri takvih provera su uparivači *isPrivate()*, *isProtected()* ili *isPublic()* koji se mogu primeniti nad čvorovima koji predstavljaju klase u C++-u. I ovde postoji nekoliko posebnih uparivača kao što su *anyOf()*, *allOf()*, *anything()* i *unless()* koji služe za pisanje složenijih izraza.

Rezultat uspešnog uparivanja je čvor koji odgovara početnom uparivaču, odnosno korenu celog izraza. Ukoliko je potrebno sačuvati uparivanje nekog drugog čvora koji je posećen usput to se može učiniti *vezivanjem* (eng. *bounding*). Na sledećem primeru se može videti kako izgleda vezivanje leve strane izraza binarnog operatora:

```
binaryOperator(hasLHS(expr().bind("lhs")))
```

Vrednost *lhs* je identifikator koju korisnik dodeljuje tom čvoru. Taj identifikator se kasnije može iskoristiti da se pristupi čvoru koji je vezan upravo tim identifikatorom u rezultatu uparivanja.

Pošto su opisani uparivači, potrebno je videti kako se zapravo pokreću nad apstraktnim sintaksnim stablom. Uparivači su samo jedan deo ovog mehanizma za analizu stabla. Definisane uparivače je potrebno proslediti AST-pronalazaču uparivanja (eng. *AST Match Finder*). Jedan pronalazač može koristiti više uparivača. Uz svaki uparivač potrebno je proslediti i objekat klase *Callback* koji obrađuje svaki rezultat uparivanja. Ponašanje za pronađene rezultate se zadaje tako što se nasledi bazna klasa *MatchFinder::MatchCallback* i predefiniše metod *run()* koji se automatski poziva za svako uparivanje. Ovaj metod kao argument dobija skup čvorova gde je svakom dodeljena jedinstvena identifikacija i to upravo ona koja je zadata prilikom vezivanja čvorova, odnosno upotrebom posebnog uparivača *bind()*. Nakon što se pronalazaču proslede svi željeni uparivači, kako bi se otpočela pretraga apstraktnog

sintaksnog stabla, potrebno je proslediti mu i odgovarajuću instancu *ASTContext*-a.

Pronalazač svoj posao obavlja tako što konstruiše svoj interni rekurzivni AST-posetilac i pokuša da obavi uparivanje nad svakim čvorom. Ovo znači da svaka upotreba uparivača košta barem onoliko koliko i jedan obilazak celog stabla. Prednost upotrebe uparivača je što se može iskorititi više njih sa jednim pronalazačem.

## Biblioteka LibTooling

Pisanje novih alata koji koriste biblioteke Clang-a se može obaviti pomoću interfejsa *LibTooling*. Značajan broj alata je implementiran na ovaj način, mnogi od njih su uključeni u sam projekat LLVM. Jedan od njih je i alat *clang-tidy* koji vrši statičku analizu koda i može detektovati veliki broj prekršaja počevši od pogrešne upotrebe stilova i konvencija kodiranja LLVM-a pa do grešaka i bagova u kodu.

Ova biblioteka omogućava da se izvrši Clang-ov prednji deo nad nekom izvornom datotekom. To se može učitini upotrebom klase *FrontendAction* koja za jednu jedinicu prevođenja proizvodi jedno apstraktno sintaksno stablo.

Obradu stabla obavlja klasa *ASTConsumer*. Veliki broj akcija koje sam Clang obavlja su implementirane upravo na ovaj način. Primeri su klasa *ASTPrinter* koja generisano apstraktno sintaksno stablo konvertuje nazad u formatirani izvorni kôd ili klasa *CodeGen* koja generiše međureprezentaciju LLVM-a na osnovu stabla.

Sve ovo započinje instanca klase *ClangTool*. Ukoliko je potrebno da alat podrži različite opcije ili konfiguracije u tome može pomoći Clang-ov parser opcija komandne linije definisan klasom *CommonOptionsParser*. Detalji upotrebe svih ovih klasa su predstavljeni u sekciji za implementaciju (Glava 4).

# Glava 4

# Implementacija alata *switch-check*

U ovom poglavlju je opisan proces implementacije alata *switch-check* koji je dostupan na adresi `https://gitlab.com/Sandman1705/switch-check`. Pojašnjene su razne odluke i strategije koju su donete za implementaciju svakog pravila. Sama implementacija je podeljena u četiri biblioteke.

Prvo se opisuje biblioteka *Tool* koja predstavlja skelet alata. Zajedno sa ovom biblotekom se opisuje i jedini samostali izvorni fajl (*SwitchCheck.cpp*) koji definiše opcije komande linije i sadrži funkciju *main*.

Pravila koja su ovde prikazana se sva odnose na naredbu *switch*. Prema klasifikacijama pravila standarda AUTOSAR sva pravila ovde predstavljena su obavezna po nivou važnosti, automatizovana po primenljivosti statičke analize i implementaciona po oblasti primene.

Svako pravilo je implementirano jednim posetiocem i jednim uparivačem koji obavljaju identičnu vrstu posla drugim metodama. Implementacije oba metoda služe radi njihovog poređenja. Kada se kaže da je neko pravilo implementirano misli se na to da postoji statička analiza apstraktnog sintaksnog stabla koja detektuje odstupanje od pravila u zadatom kodu i prijavljuje odgovarajuću poruku. Sve implementacije pravila su sadržane u bibliotekama *Visitors* i *Matchers*.

Poslednja biblioteka koja je sastavni deo projekta je pomoćna biblioteka *Common*. Ovde se nalaze neke od pomoćnih funkcija koje se koriste i u biblioteci *Visitors* i u biblioteci *Matchers*. Funkcije ove biblioteke su opisane u sklopu opisa ostalih biblioteka.

## 4.1 Osnovni elementi alata *switch-check*

Upotreba alata *switch-check* je slična Clang-u i njegovim opcijama. Lista argumenata treba da sadrži u sebi listu datoteka koje je potrebno obraditi, kao i listu opcija koje govore koje provere je potrebno obaviti. Primer upotrebe ovog alata je sledeći:

```
$ switch-check test.cpp --v-structured-label --m-terminate-clause --
```

Za svako pravilo postoji opcija komandne linije koja govori alatu da li je potrebno izvršiti analizu koja proverava saglasnost koda sa tim pravilom. Sve opcije su definisane pomoću elemenata biblioteke *Tooling*. Komandne opcije alata koje su definisane uvek počinju sa dve crtice (--). Potom sledi naziv opcije koji počinje malim slovom *v* za posetioce (*Visitors*) ili slovom *m* za uparivače (*Matchers*), prećeni simboličnim imenom koje opisuje koja se provera vrši. Lista opcija se može videti u tabeli 4.1.

Tabela 4.1: Pregled opcija komande linije alata koje kontrolišu izvršavanje provera

| Broj pravila | Posetilac | Uparivač |
|---|---|---|
| M6-4-3 | --v-well-formed | --m-well-formed |
| M6-4-4 | --v-structured-label | --m-structured-label |
| M6-4-5 | --v-terminate-clause | --m-terminate-clause |
| M6-4-6 | --v-default-final | --m-default-final |
| M6-4-7 | --v-no-bool-condition | --m-no-bool-condition |
| A6-4-1 | --v-two-clause-minimum | --m-two-clause-minimum |
| * | --v-all-checks | --m-all-checks |

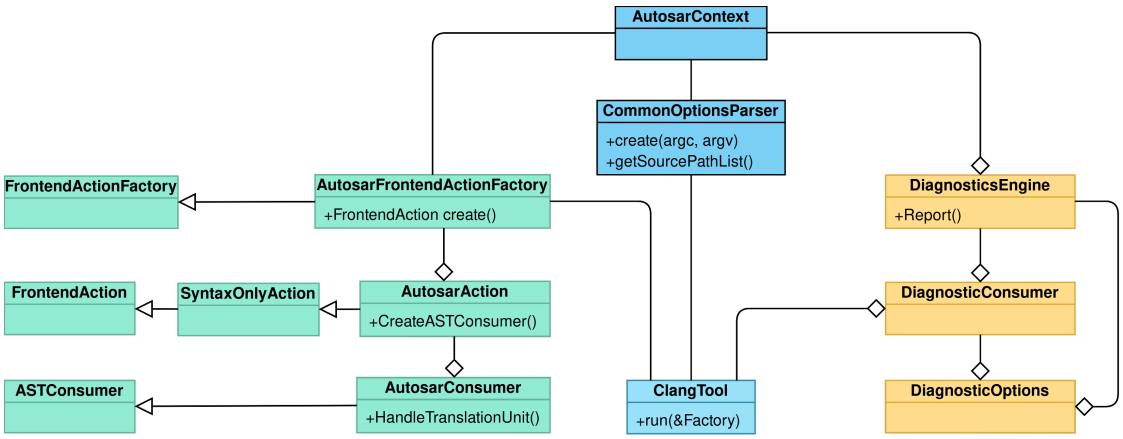
Klasa *CommonOptionsParser* pomaže u parsiranju argumenata komande linije. Informaciju o uključenim opcijama je potrebno nekako zapamtiti. Da bi se zapamtile uključene opcije u okviru alata je definisana klasa *AutosarContext* kojoj se prosleđuje rezultat parsera o uključenim opcijama. Lista ulaznih datoteka, koju je parser prikupio iz argumenata komandne linije, se prosleđuje novoj instanci klase *ClangTool*.

Alatu je potrebno zadati akciju koju treba da izvrši. U ovom slučaju to je akcija prednjeg dela, odnosno instanca klase *FrontendAction*. Međutim, ne može se definisati specijalizacija nove akcije. Klasa *ClangTool* traži objekat koji proizvodi akcije odnosno uzorak fabrike (eng. *Factory*). Fabrika za ovaj alat se zove *AutosarFrontendActionFactory* koja uzima objekat *AutosarContext* i sa njim iniciajlizuje novi objekat klase AutosarAction. Razlog zbog kojeg se zahteva fabrika je što alat

nekada mora obraditi više jedinica prevođenja i potreban mu je način inicijalizacije akcije za svaku od njih.

Nova akcija, *AutosarAction*, je specijalizacija sintaksne akcije, tj. klase *SyntaxOnlyAction* (koja je zauzvrat specijalizacija klasa *ASTFrontendAction* i *FrontendAction*). Ona obavlja parsiranje ulazne datoteke i generiše odgovarajuće apstraktno sintaksno stablo. Dodatno ponašanje koje je zadato ovoj akciji se sastoji iz konstrukcije novog AST-potrošača (*ASTConsumer*) koji zapravo vrši obradu stabla. Pre toga je naglašeno instanci kompilatora koja je deo akcije da ne prikazuje bilo kakvu dijagnostiku koja se može pojaviti prilikom parsiranja ulaznog koda. Sva upozorenja koja se prijavljuju korisniku su generisana od strane ovog alata.

Potrošač, čija klasa nosi naziv *AutosarConsumer*, nasleđuje bazni *ASTConsumer* i takođe je inicijalizovan instancom *AutosarContext*-a. Nakon što akcija prednjeg dela isparsira ulazni kôd i generiše apstraktno sintaksno stablo ono se prosleđuje potrošaču. Potošač potom inicijalizuje odgovarajuće posetioce i uparivače na osnovu opcija komandne linije, odnosno instance *AutosarContext*-a, a potom im prosleđuje apstraktno sintaksno stablo kako bi mogli da obave svoje analize. Dijagram koji opisuje strukturu alata se može videti na slici 4.1.




Slika 4.1: Struktura alata switch-check

## Dijagnostika

Postoji još jedan bitan element koji je potrebno obezbediti celom ovom mehanizmu, a to je način za obradu dijagnostike. Za generisanje upozorenja koja je potrebno prijaviti korisniku, koristi se isti mehanizam koji koristi i sam Clang. Instanciran je novi stroj za dijagnosiku (*DiagnosticsEngine*). Ovom objektu se prijavljuju svi

prekršaji koji se detektuju, zbog čega se čuva njegova referenca u klasi *AutosarContext* kako bi se dalje mogla proslediti posetiocima i uparivačima. Stroj takođe mora imati svog potrošača (*DiagnosticConsumer*) i u ovom slučaju to je klasa koja ispisuje upozorenja u tekstualnom formatu isto kao što to radi i Clang, tj. instanca klase *TextDiagnosticPrinter*.

Sa ovime je skelet alata upotpunjen. Instanci *ClangTool*-a se potom može proslediti tražena fabrika nakon čega alat može da otpočne svoj zadatak.

### Dijagnostičke poruke

Prekršaj pravila se registruje kod sistema za dijagnostiku pomoću metode *Report()* klase *DiagnosticsEngine*. Prlikom prijave može se zadati veći broj informacija, ali osnovni su lokacija gde treba prijaviti poruku koja je objekat klase *SourceLocation*, kao i identifikator koji je celobrojna vrednost.

Identifikatori za greške i upozorenja koje Clang može da ispiše su definisani statički, u jednoj od osnovnih biblioteka. Novi se mogu generisati pomoću metode *getCustomDiagID()* koja uzima dva argumenta. Prvi argument je nivo poruke. To može biti greška (eng. *error*), upozorenje (eng. *warning*) ili beleška (eng. *note*). Sve poruke koje alat *switch-check* generiše se smatraju upozorenjima. Drugi argument ove metode je niska koja predstavlja tekst koji se ispisuje u slučaju upotrebe ove dijagnostike. Poruke imaju sledeći oblik: tekst pravila praćen uglastim zagradama gde je naveden broj pravila i objekat koji je generisao upozorenje (posetilac ili uparivač). Primer poruke može izgledati ovako:

```
terminate-clause.cpp:20:3: warning: The condition of a switch statement shall
    not have bool type [M6-4-7 NoBoolConditionVisitor]
```

## 4.2 Pravilo M6-4-7 (*no-bool-condition*)

Za početak se prikazuje jedno kratko pravilo koje je definisano na sledeći način:

M6-4-7

(eng) *The condition of a switch statement shall not have bool type.*

(srp) Uslov naredbe *switch* ne bi trebalo da bude istinitosnog tipa.

Opravdanje za ovu vrstu preporuke je što se takve naredbe mogu zameniti naredbama *if*. Primer koda koji nije saglasan sa ovim pravilom je:

```
switch (a > 0) {
  ...
}
```

Apstraktno sintaksno stablo koje ovaj primer generiše je sledeće:

```
-SwitchStmt 0x562e612b9368 <line:2:3, line:11:3>
 |-ImplicitCastExpr 0x562e612b9350 <line:2:11, col:15> 'int' <IntegralCast>
 | `-BinaryOperator 0x562e612b9330 <col:11, col:15> 'bool' '>'
 |   |-ImplicitCastExpr 0x562e612b9318 <col:11> 'int' <LValueToRValue>
 |   | `-DeclRefExpr 0x562e612b92d8 <col:11> 'int' lvalue ParmVar 0x562e612b9128
     'a' 'int'
 |   `-IntegerLiteral 0x562e612b92f8 <col:15> 'int' 0
 `-CompoundStmt 0x562e612b9508 <col:18, line:11:3>
   ...
```

Naredba *switch* je predstavljena čvorom klase *SwitchStmt*. Iz stabla se vidi da ona ima dva potomka. Prvi je izraz koji odgovara uslovu u zagradi naredbe *switch* i tom čvoru se može pristupiti pomoću metode *getCond()*. Drugi potomak je naredba predstavljena klasom *CompoundStmt* i odgovara bloku koda, odnosno telu naredbe *switch*. Može joj se pristupiti preko metode *getBody()* ali u uvom pravilu za tim nema potrebe.

Metod *getCond()* vraća čvor klase *Expr*. Čvorovi *Expr* sadrže element *QualType* koji predstavlja tip odgovarajućeg izraza iz izvornog koda, a metod *getCond()* daje tačan izraz koji je potrebno proveriti. Međutim, može se primetiti u stablu za dati primer da je taj izraz čvor tipa *ImplicitCastExpr* i da je njegov tip zapravo *int* iako je u kodu naveden istinitosni izraz.

```
|-ImplicitCastExpr 0x562e612b9350 <line:2:11, col:15> 'int' <IntegralCast>
| `-BinaryOperator 0x562e612b9330 <col:11, col:15> 'bool' '>'
```

Čvor ispod jeste izraz tipa *bool*. Dakle, postoji implicitna konverzija između ova dva tipa. Prema C++ standardu sekciji 6.4.2 *[stmt.switch]* paragrafu 2, uslov naredbe *switch* mora biti celobrojnog ili prebrojivog tipa ili se konvertuje u neki od tih tipova [23]. To znači da u slučaju uslova koji je tipa *bool* uvek se može očekivati konverzija u tip *int*.

Klasa *Expr* nudi metode *IgnoreImpCasts()* i *IgnoreCasts()* koji se spuštaju niz stablo do prvog čvora koji nije konverzija. Nakon toga se nad dobijenim rezultatom može izvršiti provera tipa. Takođe je dostupan i metod *isKnownToHaveBooleanValue()* koji automatski obavlja ovu proveru konverzija i tipova. Dodatno proverava i da li je u pitanju celobrojni tip koji može imati samo vrednosti 0 ili 1 što može biti

korisno za jezik C, a u ovom slučaju je i više nego što je potrebno. Bilo koji od ova dva pristupa dovodi do traženog rešenja.

## Implementacija posetioca

Posetilac za ovo pravilo je vrlo kratak jer je potrebna jedna metoda *Visit** i to metoda *VisitSwitchStmt()*. Preko nje se može pristupiti odgovarajućem izrazu koji predstavlja uslov, obaviti gore spomenute provere tipa i po potrebi prijaviti upozorenju za lokaciju koja odgovara čvoru uslova, ukoliko je taj uslov tipa *bool*. Kôd ove metode je sledeći:

```
bool NoBoolConditionVisitor::VisitSwitchStmt(const SwitchStmt *SS) {
  const Expr *Cond = SS->getCond();

  if (Cond->isKnownToHaveBooleanValue()) {
    DE.Report(Cond->getBeginLoc(), DiagID) << Cond->getSourceRange();
  }

  return true;
}
```

## Implementacija uparivača

Ponovo se kreće od čvorova *SwitchStmt*. Pomoću uparivača za obilazak *hasCondition* i *ignoringImpCasts* može se obaviti neophodno spuštanje niz stablo. Interno oni pozivaju metode *getCond()* i *IgnoreImpCasts()*, iste one koje su korišćene u posetiocu. Uparivač *hasType* vraća čvor klase *QualType* nad kojim se može upotrebiti sužavajući uparivač *booleanType*.

```
switchStmt(hasCondition(
    expr(ignoringImpCasts(
        hasType(booleanType()))
    .bind("condition")));
```

Uparivanje je uspešno obavljeno ukoliko je uslov tipa *bool*. Potrebno je zapamtiti odgovarajući čvor za lokaciju. Iako se preko korenog čvora uparivača može pristupiti uslovu naredbe *switch* sa *getCond()*, poželjno je biti što precizniji sa uparivanjima. Zbog toga se obavlja vezivanje nad čvorom uslova. Kako se vezivanje ne može obaviti nad uparivačem obilaska *hasCondition*, ubačen je uparivač *expr* posle njega. On je uvek uspešan i tako se može obaviti vezivanje.

Odgovarajuća klasa *Callback* prijavljuje po jedno upozorenje za lokaciju svakog vezanog čvora.

## 4.3 Pravilo M6-4-6 (*default-final*)

Sledeće pravilo zahteva više metoda *Visit** ili više uparivača za pravilnu implementaciju.

**M6-4-6**

(eng) *The final clause of a switch statement shall be the default-clause.*

(srp) Poslednja klauza u narebi *switch* bi trebalo da bude klauza *default*.

Sledeći kôd ne poštuje pravilo:

```
void test(int x) {
    switch (x)
    {
    case 0:
        break;
    default:
        break;
    case 1:
    case 2:
        break;
    }
}
```

Apstraktno sintaksno stablo koje odgovara naredbi *switch* iz koda je sledeće:

```
1  `-SwitchStmt 0x5581ce0fe3d0 <line:2:3, line:11:3>
2    |-ImplicitCastExpr 0x5581ce0fe3b8 <line:2:11> 'int' <LValueToRValue>
3    | `-DeclRefExpr 0x5581ce0fe398 <col:11> 'int' lvalue ParmVar 0x5581ce0fe1e8 'x'
         'int'
4    `-CompoundStmt 0x5581ce0fe568 <line:3:3, line:11:3>
5      |-CaseStmt 0x5581ce0fe438 <line:4:3, line:5:5>
6      | |-ConstantExpr 0x5581ce0fe418 <line:4:8> 'int'
7      | | |-value: Int 0
8      | | `-IntegerLiteral 0x5581ce0fe3f8 <col:8> 'int' 0
9      | `-BreakStmt 0x5581ce0fe460 <line:5:5>
10     |-DefaultStmt 0x5581ce0fe470 <line:6:3, line:7:5>
11     | `-BreakStmt 0x5581ce0fe468 <col:5>
12     `-CaseStmt 0x5581ce0fe4d0 <line:8:3, line:10:5>
13       |-ConstantExpr 0x5581ce0fe4b0 <line:8:8> 'int'
14       | |-value: Int 1
```

```
15      | `-IntegerLiteral 0x5581ce0fe490 <col:8> 'int' 1
16      `-CaseStmt 0x5581ce0fe538 <line:9:3, line:10:5>
17        |-ConstantExpr 0x5581ce0fe518 <line:9:8> 'int'
18        | |-value: Int 2
19        | `-IntegerLiteral 0x5581ce0fe4f8 <col:8> 'int' 2
20        `-BreakStmt 0x5581ce0fe560 <line:10:5>
```

Čvorovi koji odgovaraju klauzama su tipa *CaseStmt* ili *DefaultStmt* i imaju zajedničku baznu klasu *SwitchCase*.

Može se reći da pravilo nije ispoštovano ukolilko tokom podrazumevanog *pre-order* obilaska stabla, čvor *CaseStmt* se poseti posle čvora *DefaultStmt*. To bi bilo dovoljno ukoliko postoji samo jedna naredba *switch* u stablu. Potrebno je znati kada je završeno sa jednom naredbom *switch* i kada je započeta obrada sledeće.

## Implementacija posetioca

Poseta čvoru *SwitchStmt* govori kada je započet obilazak nove naredbe *switch* u kodu i može pomoći u situacijama sa više uzastopnih naredbi *switch*. Ovo, međutim, ne pruža dovoljnu količinu informacija za prepoznavanje situacije sa ugnježdenim naredbama *switch* jer i dalje nije poznato kada je završena poseta celog podstabla čvora *SwitchStmt*.

Neophodan podatak o tome kada je završen obilazak čvora se može saznati uz pomoć metode *Traverse**. Kada jedan od ovih metoda završi svoj posao za neki čvor, posećeno je celo podstablo tog čvora. Obavlja se specijalizacija metode *TraverseSwitchStmt()*. Kako zapravo nije potrebno promeniti podrazumevani način obilaska stabla, koristi se poziv tog istog metoda bazne klase koji obavlja rekurzivni obilazak podstabla i time se zadržava originalno ponašanje. Sva logika koja se doda pre ovog poziva se obavlja pre obilaska naredbe *switch*, a isto tako logika koja se doda posle poziva se obavlja tek nakon što je celo podstablo naredbe *switch* posećeno. Opšti oblik ovog metoda je:

```
bool DefaultFinalVisitor::TraverseSwitchStmt(SwitchStmt *S) {
  // Kod koji se izvrsava pre obilaska podstabla cvora S.
  ...

  // Poseta cvora S i celog njegovog podstabla.
  RecursiveASTVisitor<DefaultFinalVisitor>::TraverseSwitchStmt(S);

  // Kod koji se izvrsava posle obilaska podstabla cvora S.
  ...
  return true;
```

```
}
```

Za svaku naredbu *switch* tokom njenog obilaska, prati se određena grupa podataka. Podaci se prate uz pomoć dodatne strukture podatka. Prvi podatak govori da li je već posećen čvor *DefaultStmt* za tu naredbu *switch*, kako bi bilo prijavljeno upozorenje prilikom naredne posete čvora klase *CaseStmt*, ukoliko postoji. Drugi podatak govori da li je već prijavljeno upozorenje za trenutnu naredbu *switch*, jer nije poželjno opteretiti korisnika sa velikim brojem upozorenja o istoj naredbi.

Kako bi se pravilno ispratile situacije sa ugnježdenim naredbama *switch*, koristi se stek ovih grupa podataka. Kada se započne obilazak čvora *SwitchStmt*, dodaje se novi element na vrh steka, a kada se završi obilzak celog podstabla, uklanja se element sa vrha steka. To znači da na vrhu steka tokom obilaska uvek stoji struktura koja odgovara obradi elemenata odgovarajuće naredbe *switch*. Na kraju naredbe *switch*, potrebno je proveriti da li je uopšte posećen čvor tipa *DefaultStmt*. Ukoliko nije postojala klauza *default*, kôd takođe nije saglasan sa pravilom M6-4-6 i u toj situaciji je takođe potrebno prijaviti upozorenje.

## Implementacija uparivača

Kao što je primećeno tokom implementacije posetioca, postoje dve situacije koje je potrebno prepoznati za svaku naredbu *switch*:

- Da li uopšte postoji čvor *DefaultStmt*?
- Da li postoji čvor *CaseStmt* posle čvora *DefaultStmt*?

Implementacija je podeljena na više manjih pomoćnih uparivača. Glavni ili početni uparivač se može definisati na sledeći način:

```
switchStmt(stmt().bind("switch"), anyOf(noDefault, defaultNotLast))
```

Čvor *SwitchStmt* se vezuje iz dva razloga. Prvi je jer je potrebno da se na njegovoj lokaciji prijavi upozorenje ukoliko ne postoji klauza *default*. Drugi je jer služi u poduparivačima *noDefault* i *defaultNotLast* koji predstavljaju po jedan od uslova koji je potrebno proveriti.

Uparivač koji traži klauzu *default* se može proveriti pomoću uparivača za obilazak *hasDescendant*. Međutim i ovde je potrebno obratiti pažnju na slučajeve sa ugnježdenim naredbama *switch*. Potrebna je garancija da čvor *DefaultStmt* sa kojim se vrši uparivanje zapravo pripada odgovarajućem čvoru *SwitchStmt*, a ne ugnježdenoj naredbi *switch*. Uparivač je definisan na sledeći način:

```
auto noDefault = unless(hasDescendant(defaultStmt(isInSameSwitch)));
```

Zbog redosleda naredbi nije moguće biti siguran kojoj naredbi *switch* pripada upareni čvor *DefaultStmt*. Međutim, obilazak u obrnutom smeru je nedvosmislen. Uparivač *hasAncestor(switchStmt())* pozvan nad čvorom *DefaultStmt* se uparuje sa prvim čvorom klase *SwitchStmt* odnosno onom naredbom *switch* kojoj pripada. Ovo se pamti sa pomoćnim uparivačem u kojem se obavlja vezivanje tog čvora:

```
auto matchParentSwitch = hasAncestor(switchStmt().bind("pswitch"));
```

Preostalo je proveriti da li su to isti čvorovi. Provera se obavlja pomoćnim uparivačem *equalsBoundNode*. Međutim, on ne može uzeti dva proizvoljna čvora. Jedan od argumenata za poređenje mora biti čvor nad kojim je pozvan, tako da uslov uparivača *IsInSameSwitch* mora imati sledeći oblik:

```
auto isInSameSwitch = allOf(
  matchParentSwitch, hasAncestor(switchStmt(equalsBoundNode("switch"),
                                            equalsBoundNode("pswitch"))));
```

Ovo je dovoljno da se pokrije prva situaciju kada ne postoji čvor tipa *DefaultStmt*. Kako bi bila detektovana druga situacija potrebno je upariti par čvorova *CaseStmt* i *DefaultStmt* gde je zapravo klauza *case* posle klauze *defualt*:

```
auto defaultNotLast = allOf(matchDefault, caseAfterDefault);
```

Uparivanje za čvor tipa *DefaultStmt* je već prikazano ranije:

```
auto matchDefault =
  hasDescendant(defaultStmt(isInSameSwitch).bind("default"));
```

Uparivanje za čvor tipa *CaseStmt* se obavlja slično, ali sa jednom vrlo bitnom dopunom. Neophodno je da se ova naredba nalazi posle klauze *default*:

```
auto caseAfterDefault =
  hasDescendant(caseStmt(isInSameSwitch, afterBoundNode("default")));
```

Navedeni element *afterBoundNode* nije uparivač koji postoji u Clang-u. To je potpuno novi samostalni uparivač implementiran u okviru alata. Pod tim se ne misli na složene uparivače koji se izgrađuju od već postojećih elemenata i uparivača kao u bilo kojem primeru do sada. Novi uparivač *afterBoundNode* je jednak sa svim predefinisanim uparivačima u Clang-u i definisan je kao novi sužavajući uparivač sa globalnim opsegom.

Definicija je vrlo slična već postojećem uparivaču *equalsBoundNode*, s tim što predikat poređenja zapravo poredi lokacije čvorova u izvornom kodu umesto njihovih referenci. Uparivači ovog tipa se definišu pomoću makroa

*AST_POLYMORPHIC_MATCHER_P*. Iz naziva se može zaključiti da je ovo polimorfni uparivač, a razlog tome je što je neophodno da pokriva sve osnovne vrste čvorova: *Decl*, *Stmt* i *Type*, kao i pomoćni tip *QualType*. Kako bi se obavilo ispravno poređenje, uparivaču se daje pristup instanci *SourceManager*-a (koja se može dobiti od instance *ASTContext*-a) radi pristupa metodi *isBeforeInTranslationUnit()* za poređenje objekata *SourceLocation*.

U klasi *Callback* za celokupni uparivač potrebno je fokusirati se na dva identifikatora vezivanja. Ukoliko je identifikator vezivanja *switch* prisutan u skupu uparivanja to znači da je to čvor tipa *SwitchStmt* i da nema klauzu *default*. Ukoliko postoji identifikator *default* to je čvor *DefaultStmt* koji nije poslednja klauza unutar svoje naredbe *switch*.

## 4.4 Pravilo A6-4-1 (*two-clause-minimum*)

Ponekad uzastopne klauze *case* mogu da zakomplikuju proveru saglasnosti. To se može videti u narednom pravilu koje kaže:

A6-4-1

(eng) *A switch statement shall have at least two case-clauses, distinct from the default label.*

(srp) Naredba *switch* bi trebalo da ima barem dve klauze *case*, različite od labele *default*.

Motivacija za ovo pravilo je slična kao za pravilo M6-4-7. Naredbe *switch* koje imaju manje od dve klauze se mogu opisati sa naredbama *if*.

U nastavku je dat jedan primer sa uzastopnim klauzama *case*:

```
switch (x) {
case 0:
case 1:
case 2:
  // ...
  break;
case 3:
case 4:
  // ...
  break;
```

```
case 5:
default:
case 6:
  // ...
  break;
}
```

Sledi apstraktno sintaksno stablo koji se generiše od primera iznad. Eliminisan je veći broj čvorova iz stabla radi kraćeg zapisa:

```
1  `-SwitchStmt 0x55d4fa0f54a0 <line:2:3, line:14:3>
2    |-ImplicitCastExpr ...
3    `-CompoundStmt 0x55d4fa0f57d8 <col:14, line:14:3>
4      |-CaseStmt 0x55d4fa0f5508 <line:3:3, line:6:5>
5      | |-ConstantExpr ... Int 0
6      | `-CaseStmt 0x55d4fa0f5570 <line:4:3, line:6:5>
7      |   |-ConstantExpr ... Int 1
8      |   `-CaseStmt 0x55d4fa0f55d8 <line:5:3, line:6:5>
9      |     |-ConstantExpr ... Int 2
10     |     `-BreakStmt 0x55d4fa0f5600 <line:6:5>
11     |-CaseStmt 0x55d4fa0f5648 <line:7:3, line:9:5>
12     | |-ConstantExpr ... Int 3
13     | `-CaseStmt 0x55d4fa0f56b0 <line:8:3, line:9:5>
14     |   |-ConstantExpr ... Int 4
15     |   `-BreakStmt 0x55d4fa0f56d8 <line:9:5>
16     `-CaseStmt 0x55d4fa0f5720 <line:10:3, line:13:5>
17       |-ConstantExpr ... Int 5
18       `-DefaultStmt 0x55d4fa0f57b8 <line:11:3, line:13:5>
19         `-CaseStmt 0x55d4fa0f5788 <line:12:3, line:13:5>
20           |-ConstantExpr ... Int 6
21               `-BreakStmt 0x55d4fa0f57b0 <line:13:5>
```

Primećuje se da ukoliko su klauze grupisane u izvornom kodu, u stablu su deo zajedničkog podstabla koje počinje za čvorom prve klauze te grupe. Odnosno, ukoliko u kodu jedna klauza propada (eng. *fallthrough*) u drugu, ta druga je čvor koji predstavlja telo prve klazue tj. potomak koji se dobija metodom *getBody()*.

Kada je reč o grupama klauza, potrebno je obratiti pažnju na dve sledeće situacije. U prvoj postoji jedna grupa klauza koja ne sadrži klauzu *default* i takav kôd je saglasan sa pravilom:

```
switch (x) {
case 0:
case 1:
  break;
default:
  break;
}
```

Razlog je taj što bi naredba *if* koja bi trebala da zameni takav kôd morala da proveri da li je izraz iz uslova naredbe *switch* jednak bilo kojoj vrednosti neke od klauza *case*. Za male grupe klauza ili određene vrednosti taj izraz ne mora nužno biti složen, ali to ne važi u opštem slučaju.

Nasuprot ovome, sledeći primer nije saglasan sa pravilom:

```
switch (x) {
case 0:
  break;
case 1:
case 2:
case 3:
default:
  break;
}
```

Ponovo postoje dve grupe klauza, ali van grupe koja sadrži klauzu *default* postoji jedna, (*case 0*). Pošto klauza *default* pokriva sve slučajeve koji nisu pokriveni nekom od ostalih klauza, sve klauze *case* u grupi sa klauzom *default* se mogu eliminisati. To dovodi do slučaja gde efektivno postoje dve klauze: *case 0* i *default*.

## Implementacija posetioca

Posetilac za ovo pravilo sadrži jedan metod *Visit** i to za klasu *SwitchStmt*. Iteracijom kroz listu koja se dobija sa metodom *getSwitchCaseList()* mogu se posetiti sve klauze i nije potrebno brinuti o slučajevima ugnježdenih naredbi *switch*. Razlog zbog kojeg se ovaj način nije mogao iskoristiti u prethodnom pravilu, M6-4-6, je što ne postoji garancija da iteracija kroz listu daje klauze u redosledu kojim se pojavljuju u stablu.

Za bilo koji čvor se može videti koji su njegovi direktni potomci pomoću neke od metoda odgovarajuće klase. Tako za klasu *SwitchCase* postoji metod *getSubStmt()* koji vraća čvor klase *Stmt*. U ponudi je i šablonska funkcija *isa<>()* pomoću koje se može proveriti dinamička klasa ovog čvora. Ukoliko je u pitanju još jedan čvor *SwitchCase* onda je trenutni čvor deo grupe klauza.

Nasuprot tome, provera roditelja se može obaviti uz pomoć metoda *getParents()* klase *ASTContext*. Ovaj metod, međutim, nije bez svoje cene, jer može izazvati obilazak većeg broja čvorova. Zbog toga se izbegava njegova upotreba osim ukoliko nije neophodna. Za ovo pravilo su dovoljne provere koje vrše pretragu niz stablo.

Implementirane su dve pomoćne funkcije. Prva za čvor klase *SwitchCase* vrši pretragu niz stablo da prebroji broj klauza u koje se ta klauza preliva:

```
static unsigned countFallthroughGroup(const SwitchCase *SC) {
  unsigned Count = 0;
  while (const SwitchCase *Fallthrough =
             dyn_cast_or_null<SwitchCase>(SC->getSubStmt())) {
    SC = Fallthrough;
    ++Count;
  }
  return Count;
}
```

Druga proverava je da li se klauza predstavljena čvorom *SwitchCase* preliva u čvor *DefaultStmt*:

```
static bool hasFallthroughToDefault(const SwitchCase *SC) {
  while (const SwitchCase *Fallthrough =
             dyn_cast_or_null<SwitchCase>(SC->getSubStmt())) {
    if (isa<DefaultStmt>(Fallthrough))
      return true;
    SC = Fallthrough;
  }
  return false;
}
```

Tokom iteracije kroz sve klauze naredbe *switch* obavlja se brojanje klauza *case* koje se ne prelivaju u klauzu *default*. Za sledeći primer to znači da se ubrajaju i klauze koje su ispod klauze *default*:

```
case 1:
case 2:
default:
case 3:
case 4:
```

Kako bi se rešio ovaj problem, oduzima se broj klauza u koje se preliva čvor *DefaultStmt*. Time se dobija tačan broj klaza i ukoliko je na kraju iteracije manji od dva, naredba *switch* nije saglasna sa pravilom. Sledeći kod predstavlja način brojanja zadovoljavajućih klauza bez ranijeg zaustavljanja:

```
  const SwitchCase *Curr = SS->getSwitchCaseList();
  while (Curr) {
    if (isa<DefaultStmt>(Curr)) {
      SeenDefault = true;
```

```
      // ukloni pogresno uracunate klauze
      DistinctCases -= countFallthroughGroup(Curr);
    } else if (!hasFallthroughToDefault(Curr))
      ++DistinctCases;

    Curr = Curr->getNextSwitchCase();
  }
```

Opciono se može obaviti raniji prekid iteracije ukolko je brojač veći ili jednak dva i već je obrađen čvor *DefaultStmt* što znači da neće biti daljeg umanjenja.

## Implementacija uparivača

Implementacija uparivača može započeti šablonom koji je sličan prethodnom pravilu:

```
switchStmt(stmt().bind("switch"), unless(twoCasesDistinctFromDefault))
```

Koristi se uparivač *unless* kako bi se izvršila inverzija uslova. Potrebno je pronaći dve klauze *case* koje ispunjavaju sledeće uslove:

- Moraju biti deo iste naredbe *switch*, tj. potrebno je obratiti pažnju na slučajeve ugnježdenih naredbi *switch*.
- Ne smeju biti deo grupe klauza koja sadrži čvor *DefaultStmt*.
- Ove dve klauze *case* ne smeju biti ista klauza.

Uparivanje sa jednom takvom klauzom *case* se može opisati na sledeći način gde se proveravaju prva dva uslova:

```
hasDescendant(caseStmt(isInSameSwitch, noDefaultInFallthroughGroup)
```

Ukoliko bi ovaj uparivač bio upotrebljen više puta nad istim čvorom *SwitchStmt*, svaki put bi proizveo uparivanje sa istom klauzom *case*. Razlog je što se uparivači zaustavljaju kod prvog uparivanja i uvek prate isti redosled obilaska. Zbog toga je treći uslov navedan na sledeći način i odgovarajuće uparivanje glasi:

```
auto twoCasesDistinctFromDefault = allOf(
  hasDescendant(caseStmt(
    isInSameSwitch, noDefaultInFallthroughGroup).bind("case1")),
  hasDescendant(caseStmt(
    unless(equalsBoundNode("case1")), isInSameSwitch,
           noDefaultInFallthroughGroup)));
```

Uparivanje sa prvom klauzom *case* se uvezuje kako bi se kasnije za uparivanje sa drugom klauzom moglo zahtevati da ne bude jednako vezanom čvoru. Ova provera se navodi kao prvi uslov zato što je manje zahtevna od ostalih.

Preostalo je definisati uparivče za prva dva uslova. Prvi od njih koji je nazvan *isInSameSwitch* je zapravo identičan uparivaču koji je upotrebljen u prethodnom pravilu. Drugi uslov se može podeliti na dva poduslova:

```
auto noDefaultInFallthroughGroup =
    unless(anyOf(hasDefaultAbove(), hasDefaultBelow()));
```

Uslovi *hasDefaultAbove* i *hasDefaultBelow* se mogu definisati pomoću obilaska predaka *hasAncestor* i obilaska potomaka *hasDescendant*, ali bi ponovo bilo potrebno obartiti pažnju na slučajeve ugnježdenih naredbi *switch*. Ukoliko bi bio upotrebljen uparivač *isInSameSwitch*, došlo bi do konflka prilikom vezivanja čvorova jer su ti identifikatori već upotrebnjeni. Kako bi se izbeglo razrešavanje ovog problema i cene obilaska koje se javljaju sa uparivačima *hasAncestor* i *hasDescendant*, ovi uslovi se definišu kao novi samostalni uparivači.

Slično kao uparivač *afterBoundNode* iz prethodnog pravila, ovi uparivači se definišu na globalnom nivou. Nasuprot uparivaču *afterBoundNode*, ovo nisu polimorfni uparivači i ograničavaju se na jednu klasu čvorova i to klasu *SwitchCase*.

Uparivač *hasDefaultBelow* se iterativno spušta niz stablo sve dok je podizraz takođe klauza i proverava da li je zapravo klauza *default* ili klauza *case*. Ovo je sužavajući uparivač čije je uparivanje uspešno ako je početni čvor *SwitchCase* deo grupe koja sadrži čvor *DefaultStmt*. Uparivač *hasDefaultAbove* funkcioniše na sličan način, ali u suprotnom smeru, tj. vrši pretragu uz stablo.

Ovime je celokupni uparivač upotpunjen, a klasa *Callback* prijavljuje upozorenje za svako uspešno uparivanje.

## 4.5 Pravilo M6-4-4 (*structured-label*)

Labele u narebama *switch* se mogu javiti na bilo kom mestu unutar tela naredbe *switch*. Kako bi se izbeglo pojavljivanje nestrukturiranog koda definisano je sledeće pravilo:

M6-4-4

(eng) *A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.*

(srp) Labele u naredbama *switch* bi trebalo da se koriste samo kada je prva okružujuća naredba bloka zapravo telo naredbe *switch*.

Primer koda koji nije saglasan sa ovim pravilom je sledeći:

```
switch (x) {
case 1:
  if (cond) {
  case 2:
    f();
  }
  break;
default:
  break;
}
```

Lokacija labele *case 2* je ta koja krši pravilo. Apstraktno sintaksno stablo ovog primera, u skraćenom obliku, je:

```
1  `-SwitchStmt 0x55de7d7f66d0 <line:2:3, line:11:3>
2    |-ImplicitCastExpr ...
3    `-CompoundStmt 0x55de7d7f6938 <col:14, line:11:3>
4      |-CaseStmt 0x55de7d7f6738 <line:3:3, line:7:5>
5      | |-ConstantExpr ... Int 1
6      | `-IfStmt 0x55de7d7f68e8 <line:4:5, line:7:5>
7      |   |-ImplicitCastExpr ...
8      |   `-CompoundStmt 0x55de7d7f68d0 <col:15, line:7:5>
9      |     `-CaseStmt 0x55de7d7f67d8 <line:5:5, line:6:9>
10     |       |-ConstantExpr ... Int 2
11     |       `-CallExpr 0x55de7d7f68b0 <line:6:7, col:9> ... Function 'f' ...
12     |-BreakStmt 0x55de7d7f6908 <line:8:5>
13     `-DefaultStmt 0x55de7d7f6918 <line:9:3, line:10:5>
14       `-BreakStmt 0x55de7d7f6910 <col:5>
```

Čvorovi koje čine prekšaj se mogu prepoznati preko njihova prva dva pretka kao što je slučaj za čvor *CaseStmt* u liniju 9 u stablu u ispisu iznad.

## Implementacija posetioca

Potrebna je specijalizacija jednog metoda i to *VisitSwitchCase*. Sve provere se obavljaju ručnim kretanjem uz stablo. Kao što je ranije spomenuto, prvi direkt-

ni potomak nekog čvora se može dobiti pomoću instance *ASTContext*-a i njenog metoda *getParents()*.

Kao što ime metoda *getParents()* sugeriše, moguće je da metod vrati više od jednog čvora koji predstavlja roditelja. Situacije u kojima je ovo moguće je kada je čvor deo neke šablonske funkcije ili klase. Šabloni ne utiču na slučajeve koji se ovde obrađuju tako da je za potrebe ovog posetioca dovoljno koristiti prvi element iz liste dobijenih rezultata.

Pošto ne postoji zajednička bazna klasa za sve čvorove, elementi liste su tipa *DynTypedNode*. Ovo je pomoćna klasa koja sadrži refrencu na odgovarajući čvor. Takođe sadrži i konverter u vidu šablonskog metoda *get<>()*.

Provera da li je klauza saglasna sa pravilom se može obaviti tako što se proveri da li je roditelj klauze naredba bloka, odnosno čvor *CompoundStmt*. Potom se za taj čvor proverava da li je njegov roditelj naredba *switch*, odnosno čvor tipa *SwitchStmt*. Ukoliko bilo koja provera ne uspe, potrebno je prijaviti upozorenje za tu klauzu.

U slučaju grupisanih klauza, koje su postojale u primerima za prethodna pravila, poželjno je izbeći prijavljivanje upozorenja za svaku od klauza iz te grupe osim jedne. U suprotnom bi korisnik bio opterećen sa previše suštniski istih prekršaja. Svaka klauza osim prve iz jedne takve grupe ima zapravo čvor *SwitchCase* kao roditelja. Kako se ne bi obavilo uparivanje sa svakom klauzom grupe, dodaje se uslov da prvi roditelj može da bude i klasa *SwitchCase*. Sledeći kod predstavlja proveru da li je labela struktuirana:

```
bool StructuredLabelVisitor::isStructuredLabel(const SwitchCase *SC,
                                               ASTContext &AC) {
  const DynTypedNodeList &Parents = AC.getParents(*SC);

  // Zanemari slucajeve gde klauza nije na vrhu grupe.
  if (Parents[0].get<SwitchCase>())
    return true;

  // Proveri da li je prvi roditelj blok.
  if (const CompoundStmt *CS = Parents[0].get<CompoundStmt>()) {
    const DynTypedNodeList &CSParents = AC.getParents(*CS);
    // Proveri da li je roditelj roditelja naredba switch
    if (CSParents[0].get<SwitchStmt>())
      return true;
  }

  return false; // Klauza nije saglasa sa pravilom
```

```
}
```

### Implementacija uparivača

Uparivač prati istu logiku kao i posetilac. Pomoću uparivača obilaska *hasParent* moguće je proveriti prvog, a potom i drugog roditelja svakog čvora. Takođe, ponovo se uvodi isti izuzetak za klauze koje nisu prve u grupi uzastopnih klauza. Potpuni uparivač se može definisati na sledeći način:

```
switchCase(unless(hasParent(stmt(anyOf(
  switchCase(),
  compoundStmt(hasParent(switchStmt()))))))).bind("unstructured");
```

I u slučaju uparivača i u slučaju posetioca, kreće se ograničen broj koraka uz stablo i to tačno dva, tako da nije potrebno brinuti o ugnježdenim naredbama *switch*.

## 4.6 Pravilo M6-4-5 (*terminate-clause*)

Ukoliko na kraju klauze *switch* ne postoji naredba prekida (*break*) tada kontrola toka prelazi na sledeću klauzu *switch*. Iako postoje situacije u kojima je ovo urađeno sa namerom, često je učinjeno greškom [31]. Iz ovog razloga standard uvodi sledeće pravilo:

M6-4-5

(eng) *An unconditional throw or break statement shall terminate every non-empty switch-clause.*

(srp) Bezuslovna naredba *break* ili *throw* bi trebalo da bude završna naredba svake neprazne klauze *switch*-a.

Prazne klauze *case*, odnosno grupe uzastopnih klauza koje su postojale u prethodnim primerima su takođe dozvoljene. Glavni problem koji je potrebno razrešiti tokom implementacije ovog pravila je prepoznati završnu naredbu svake klauze. Naredni kôd ilustruje klauze koje su u skladu sa pravilom:

```
switch (x) {
case 11:
case 12:
  n = 1;
  ++n;
```

```
  break;
case 21:
case 22: {
  n = 2;
  break;
}
default:
  break;
}
```

Obe grupe klauza u primeru se završavaju naredbom *break*. Zbog upotrebe bloka u jednom od slučajeva i njegovog nedostatka u drugom, čvor *BreakStmt* se nalazi na drugačijoj lokaciji u stablu u odnosu na klauzu koju okončava:

```
1  `-SwitchStmt 0x55e57a7df488 <line:3:3, line:16:3>
2    |-ImplicitCastExpr ...
3    `-CompoundStmt 0x55e57a7df7a0 <col:14, line:16:3>
4      |-CaseStmt 0x55e57a7df4f0 <line:4:3, line:6:9>
5      | |-ConstantExpr ... Int 11
6      | `-CaseStmt 0x55e57a7df558 <line:5:3, line:6:9>
7      |   |-ConstantExpr ... Int 12
8      |   `-BinaryOperator ... '='
9      |-UnaryOperator ... prefix '++' ... Var ... 'n' ...
10     |-BreakStmt 0x55e57a7df618 <line:8:5>
11     |-CaseStmt 0x55e57a7df660 <line:9:3, line:13:3>
12     | |-ConstantExpr ... Int 21
13     | `-CaseStmt 0x55e57a7df6c8 <line:10:3, line:13:3>
14     |   |-ConstantExpr ... Int 22
15     |   `-CompoundStmt 0x55e57a7df758 <col:12, line:13:3>
16     |     |-BinaryOperator ... '=' ... Var ... 'n' ...  2
17     |     `-BreakStmt 0x55e57a7df750 <line:12:5>
18     `-DefaultStmt 0x55e57a7df780 <line:14:3, line:15:5>
19       `-BreakStmt 0x55e57a7df778 <col:5>
```

U prvom slučaju, čvor *BreakStmt* je na istom nivou stabla (linija 10) kao i početna klauza grupe koju okončava (linija 4), a na nivou iznad preostalih klauza te iste grupe (linija 6). U drugom slučaju je na najnižem nivou (linija 17) u odnosu na bilo koju od klauza grupe koje okončava (linije 11 i 13).

Prema tome, čak iako postoji slučaj gde je telo klauze blok naredba koja je okončana sa naredbom *break* i dalje može postojati situacija gde stvarna završna naredba nije terminirajuća, tj. naredba *break* ili *throw* (bez obzira da li je nedostižna ili ne):

```
case 3: {
  n = 2;
  break;
}
```

```
++n; // nedostizna linija koda i nedostizna zavrsna naredba za case 3:
case 4:
  // ...
```

A potom postoji i sledeća situacija sa ugnježdenim klauzama:

```
case 0: {
   n = 1;
case 1:
  break;
}
case 2:
```

Bez obzira što je tehnički telo klauze *case 0* blok čija je poslednja naredba terminirajuća to ne znači da ne postoji nedozvoljeno propadanje u drugu klauzu.

## Implementacija posetioca

Postoji potreba za proverama čvorova koji su susedni trenutnom i na istom nivou u stablu. Metodi *Traverse** mogu dati informaciju o tome da li je završena obrada celog podstabla. Međutim, metodi *Traverse** ne govore gde se nalazi sledeći čvor koji se posećuje nakon što metod završi sa radom. Nakon što metod *Traverse** obavi svoj obilazak, sledeći čvor koji se posećuje može biti sledeći sused na istom nivou ili čvor u daljem delu stabla na nekom od nivoa iznad. Jedan način provere da li su dva čvora na istom nivou i u istom podstablu je da li imaju istog roditelja. Ukoliko su dva čvora na istom nivou oni nisu nužno i susedi, a prilikom takve provere poželjno je izbeći i veliki broj poziva metode *getParents()*.

Ono što je ovde potrebno je vrsta obilaska u širinu. Kako postoje naredbe poput *CompoundStmt* koje mogu imati proizvoljan broj direktnih potomaka, takve klase moraju čuvati i listu svih direktnih potomaka. Takođe sadrže i iterator preko kojeg se može proći kroz listu.

Za posetilac se od baznih metoda koristi metod *VisitSwitchStmt*. Od naredbe *switch* se uzima čvor koji predstavlja telo naredbe tipa *CompoundStmt*, a potom se obavlja iteracija kroz listu njegovih direktnih potomaka. Za svaki čvor tipa *SwitchCase* se izvršavaju dodatne provere.

```
bool TerminateClauseVisitor::VisitSwitchStmt(const SwitchStmt *SS) {
  const CompoundStmt *Body
    = dyn_cast_or_null<CompoundStmt>(SS->getBody());
  ...
```

```
  for (auto Curr = Body->body_begin(), End = Body->body_end();
       Curr != End; ++Curr) {
    const SwitchCase *SC = dyn_cast_or_null<SwitchCase>(*Curr);

    if (!SC) continue; // Preskoci svaki cvor koji nije SwitchCase

    if (!isSwitchClauseTerminated(SC, Curr, End)) {
      // Prijavi upozorenje
      ...
    }
  }
  return true;
}
```

Kao što je primećeno na prva dva primera završna naredba može biti jedna od dva potencijalna čvora. Ukoliko sledeći sused u istom nivou nije čvor *SwitchCase* onda je završna naredba u tom istom nivou. Konkretno to je poslednja naredba pre sledeće naredbe *SwitchCase*. Za ovaj posao je implmentirana pomoćna funkcija koja kopira trenutni iterator, nastavlja iteriranje do sledeće klauze (ili kraja iteratora) i vraća čvor neposredno pre nje:

```
<SwitchCase>: <-- Klauza za koju se trazi zavrsna naredba
   <...>        <-- Podstablo klauze
     <...>
<Stmt>        <-- Prvi sledeci sused za pocetni cvor
<Stmt>
<Stmt>        <-- Zavrsna naredba
<SwitchCase>: <-- Sledeca klauza
```

Potom se proverava da li je to terminirajuća naredba, tj. čvor klase *CXXThrowExpr* ili *BreakStmt*. Ovde se može pojaviti i blok naredba, odnosno čvor tipa *CompoundStmt* u kom slučaju je potrebno rekurzivno izvršiti istu proveru za poslednju naredbu tog bloka. Za netermninirajuće naredbe se prijavljuje upozorenje.

U slučaju da je prvi sledeći sused trenutne naredbe *SwitchCase* takođe čvor *SwitchCase*, tada je završna naredba potomak trenutnog čvora:

```
<SwitchCase>: <-- Klauza za koju se trazi zavrsna naredba
   <...>        <-- Podstablo klauze
     <...>
<SwitchCase>: <-- Prvi sledeci sused za pocetni cvor je takodje klauza
                  prema tome, zavrsna naredba je u podstablu
```

Ukoliko je klauza deo grupe uzastopnih klauza, prvo je potrebno spustiti se niz stablo do poslednje klauze. Time se garantuje jedno upozorenje po grupi uzastop-

nih klauza. Potom se kao u prethodnom slučaju obavlja provera da li je naredba terminirajuća i po potrebi prijavljuje upozorenje:

```
bool TerminateClauseVisitor::isSwitchClauseTerminated(
  const SwitchCase *SC, ... Curr, ... End) {

  // Pronadji zavrsnu naredbu u istom nivou
  const Stmt *LastStmt = getLastStmtBeforeNextCase(Curr, End);

  if (LastStmt != SC) {
    // Proveri zavrsnu naredbu u istom nivou
    if (!isLastStmtTerminator(LastStmt))
      return false; // nije saglasna sa pravilom
  } else {
    // Proveri zavrsnu naredbu u podstablu
    const SwitchCase *FC = getFallthroughCase(SC);
    if (!isLastStmtTerminator(FC->getSubStmt()))
      return false; // nije saglasna sa pravilom
  }
  return true; // klauza je saglasna sa pravilom
}
```

## Implementacija uparivača

Kao što je primećeno ranije, završna naredba je ili završni čvor podstabla ili poslednji čvor pre sledeće klauze u istom nivou stabla. Zbog toga je potrebno obaviti opciono uparivanje sa oba čvora, u čemu pomaže uparivač *optionally*. Kao poseban slučaj se izdvaja klauza koja je poslednja u naredbi *switch*. Uparivanje koje se obavlja u tom slučaju izgleda nešto drugačije.

Kao početni uparivač se uzima *switchCase*, a uparivanje *getFallthroughCase* osigurava da se proverava po jedna klauza iz grupe uzastopnih klauza:

```
auto lastCaseInGroup =
  allOf(getFallthroughCase(switchCase().bind("case")),
        unless(hasParent(switchCase())));
```

Potrebno je uvezati poslednju klauzu iz grupe radi kasnijih provera. Za klauzu od koje se počinje uparivanje se zahteva da je prva u grupi klauza, pošto je jedna od potencijalnih završnih naredbi na istom nivou stabla kao ta klauza.

Dve potencijalno završne naredbe se pronalaze sa uparivačima: *lastStmtbefore-NextCase* i *lastSubStmt*. Prvi od njih se uparuje sa završnom klauzom u istom nivou (ukoliko postoji). Pretraga počinje prvo pronalaskom sledeće klauze u istom nivou:

```
auto lastStmtbeforeNextCase = hasParent(compoundStmt(has(nextCase)));
```

Tražena klauza je prva klauza u istom bloku čija je lokacija posle trenutne. Od tog čvora se obavlja uparivanje sa prvim prethodnim susedom u istom nivou:

```
auto nextCase =
  switchCase(afterBoundNode("case"), bindPreviousSibiling);
```

Nakon toga se obavlja dvostruko vezivanje čvora. Prvo je obavezno jer taj čvor predstavlja završnu naredbu. Drugo je opciono koje govori da li je čvor naredba koja je terminirajuća. Opciono uparivanje ne zahteva da uslov u njemu bude ispunjen da bi se smatralo uspešnim. Ova uparivanja se kasnije proveravaju u klasi *Callback* radi prijave upozorenja.

```
auto bindPreviousSibiling = hasPreviousSibiling(
  stmt(optionally(stmt(terminatingStmt()).bind("terminating")))
    .bind("laststmt"));
```

Pomoćni uparivač *bindPreviousSibiling* koristi dva nova samostalana uparivača: *hasPreviousSibiling* i *terminatingStmt*.

Prvi je novi samostalni uparivač grupe obilazaka koji se definiše za klasu *Stmt* i uparuje se sa čvorovima takođe klase *Stmt*. On traži suseda prethodnika tako što uzima roditelja trenutne naredbe, a potom iterira kroz listu čvorova dece dok ne pronađe čvor koje je neposredo pre trenutnog.

Drugi je novi sužavajući uparivač koji se takođe definiše za klasu *Stmt* i vrši proveru da li je trenutna naredba terminirajuća. Provera je analogna onoj iz posetioca.

Druga naredba koja je potencijalno završna se vezuje pomoću uparivača *lastSubStmt*. On prati istu logiku koja je korišćena za posetioca. Prvo se spušta niz grupu uzastopnih klauza, a potom se proveri da li je naredba *getSubStmt()* najniže klauze terminirajuća. Provera da li je terminirajuća je opciona kao i njeno vezivanje.

```
auto lastSubStmt = getFallthroughCase(switchCase(
  hasSubStmt(stmt(
    optionally(stmt(
      terminatingStmt()).bind("terminatingSub"))))));
```

Uparivač *getFallthroughCase* je još jedan samostalni uparivač koji pripada grupi uparivača za obilazak. Definisan je nad čvorovima klase *SwitchCase* i uparuje se sa tom istom klasom čvorova.

Slučaj poslednje klauze u naredbi *switch* se obavlja uparivanjem sa klauzom koja je dete čvor tela naredbe *switch* i ne postoji ni jedna druga klauza posle nje:

```
auto lastCase =
  hasParent(compoundStmt(
```

```
    hasParent(switchStmt()),
    unless(has(switchCase(afterBoundNode("case"))))));
```

Potencijalna završna naredba koja je u istom nivou je ujedno i poslednja naredba čvora *CompoundStmt* koji je sadrži. Obavlja se obavezno vezivanje, kao i opciono ukoliko je terminirajuća, isto kao i u ostalim slučajevima:

```
auto lastSwitchStmt =
  hasParent(compoundStmt(
    hasLastStmt(stmt(
      optionally(stmt(terminatingStmt()).bind("terminating")))
                .bind("laststmt"))));
```

Potencijalna završna naredba podstabla se pronalazi istim uparivačem kao i za ostale situacije. Preostalo je spojiti ove uslove u jedan:

```
 switchCase(lastCaseInGroup,
            anyOf(allOf(lastStmtbeforeNextCase, lastSubStmt),
                  allOf(lastCase, lastSwitchStmt, lastSubStmt)));
```

Potrebno je još pravilno prijaviti upozorenja u klasi *Callback*. Ukoliko je obavljeno opciono uparivanje na završnoj naredbi koja je u istom nivou to znači da je klauza terminirajuća i upozorenje se ne prijavljuje. Slično ako je obavljeno opciono uparivanje na završnoj naredbi u podstablu, a nije obavljeno obavezno uparivanje za završnu naredbu u istom nivou, takođe se ne prijavljue upozorenje. U svim ostalim slučajevima se prijavljuje.

## 4.7 Pravilo M6-4-3 (*well-formed*)

Poslednje pravilo koje se predstavlja je najsloženije do sada ali se zapravo sastoji od prethodnih:

M6-4-3

(eng) *A switch statement shall be a well-formed switch statement.*

(srp) Naredba *switch* bi trebalo da bude dobro formirana naredba *switch*.

Standard C++ daje definiciju sintakse koja je dozvoljena za naredbu *switch*. Standard kodiranja MISRA tu definiciju vidi kao slabu jer dozvoljava dosta složenog i nestruktuiranog ponašanja [31]. Umesto toga nudi strožiju definiciju sintakse. Kako je ova definicja poduža, neće biti prepisana ovde već se navode razlike, odnosno

dodatna ograničenja koja standard zahteva da bi naredbu *switch* smatrao dobro formiranom:

- Naredba *switch* bi trebalo da počne sa klauzom naredbe *switch*.
- Klauza *default* bi trebalo da bude poslednja.
- Klauze bi trebalo da se završe sa naredbama *break*, *throw* ili blokom koji se završava sa naredbama *break* ili *throw*.
- Unutar bloka klauze ne bi trebalo da se nalazi nova klauza.

Poslednje tri uslova su već obrađena tokom implementacije pravila M6-4-6, M6-4-5 i M6-4-4. Sve te provere se ovde mogu ponoviti. Jedino je potrebno dodatno obraditi prvi uslov. Primer koji nije saglasan sa prvim uslovom je:

```
switch (x) {
 int n;
 case 0: {
   n = 0;
   break;
 }
 case 1:
 // ...
```

Kako bi korisniku bile pružene korisne dijagnostičke poruke, definišu se četiri različita dijagnostička identifikatora, po jedan za svaki od četiri uslova. Svaki od njih sadrži drugačiju poruku koja opisuje koji od uslova je prekršen.

## Implementacija posetioca

Kako bi se lakše iskoristile već postojeće provere iz prethodnih posetilaca, svaka složena provera koju je potrebno obaviti za ovo pravilo je izdvojena u odgovarajući statički metod ili pomoćnu funkciju u unatar biblioteke *Common*. Pored toga je ipak potrebno implementirati četiri metode *Visit** kao i jednu *Traverse**. Takođe je potrebno i ponoviti izvesan deo koda koji je pratio ugnježdene naredbe *switch* kako bi u potpunosti ponovili provere za tri već obrađena uslova (M6-4-4, M6-4-5 i M6-4-6).

Preostao je prvi uslov iz liste, a to je da naredba *switch* bi trebalo da počne sa nekom od klauza. Uslov se može proveriti upitom da li je prva naredba u telu naredbe *switch* zapravo čvor klase *SwitchCase*.

### Implementacija uparivača

Provere za uparivač se mogu jednostavno spojiti, tj. spajanje provera ne zahteva dosta ponovljenog koda. Iskorišćavaju se funkcije koje generišu uparivače za svako od pravila:

```
auto StructuredLabel = StructuredLabelMatcher::makeMatcher();
auto TerminateClause = TerminateClauseMatcher::makeMatcher();
auto DefaultFinal = DefaultFinalMatcher::makeMatcher();
```

Potom se spajaju sa uparivačem *eachOf* koji potencijalno proizvodi po jedan rezultat za svako od uparivanja.

```
 stmt(eachOf(StructuredLabel, TerminateClause, DefaultFinal,
             StartWithCase));
```

Uslov koji ranije nije bio pokriven se može opisati sa pomoćnim uparivačem *StartWithCase*:

```
auto StartWithCase =
  switchStmt(has(
    compoundStmt(hasFirstStmt(
      stmt(unless(
        switchCase()))).bind("firstStmt")))));
```

Ovde uparivač *hasFristStmt* je novi samostalni uparivač obilaska koji se definiše nad klasom *CompoundStmt* i proizvodi rezultat klase *Stmt*. Implementacija vraća prvi element iz liste čvorova dece kojoj se pristupa.

U klasi *Callback* potrebno je proveriti svako moguće od četiri različite grupe vezivanja da li je došlo do nekog od prekršaja.

## 4.8 Evaluacija

Za predstavljeni skup pravila standarda, date su implementacije svake provere i preko posetilaca i preko uparivača. To govori da se razne provere koje se pišu za posetioce mogu na jedan ili drugi način pretvoriti u provere za uparivače i obratno. Primećuje se da se pojedine provere prirodnije implementintiraju preko jednog od interfejsa, a neke preko drugog. Pojedini konstrukti su često nezgodi za pisanje jednom od metoda.

Kako su uparivači zapravo implementirani pomoću posetilaca, tj. prilikom uparivanja koriste interni posetilac pomoću kojeg se zaustave na svakom čvoru i pokušaju da obave uparivanje, može se očekivati da se izvršavaju sporije. Prednosti koje uparivači imaju je ta što se više njih može dodeliti jednom pronalazaču i tako nekim

delom uštedeti na vremenu izvršavanja. Takođe, provere koje se definišu se bez izmena ili dopuna mogu ponovo upotrebiti u drugim uparivačima, kao što se vidi u implementaciji pravila M6-4-3 za dobro formirane naredbe *switch*.

Primeri pravila koja su poželjna za implementaciju preko uparivača su pravilo M6-4-7 za proveru da li je uslov tipa *bool* ili M6-4-4 za pravilo struktuirane labele. Ovo se primećuje i iz njhove kratke implementacije. Zahvaljujući tome što ne koriste specijalne uparivače obilaska kao što su *hasAncestor* ili *hasDescendant*, svaka provera obilazi ograničen broj čvorova. Nasuprot tome, za uparivače koji koriste ove elemente može se očekivati lošije vreme izvršavanja.

Iako je uparivač dosta složen, neke od njegovih provera se neće nužno obaviti ukoliko uparivanje ne uspe na ranijoj proveri. Zbog toga treba brinuti i o redosledu provera koje se navode, jer pronalazač koristi lenjo izračunavanje. U situacijama kao što je ova, u zavisnosti od toga da li traženi element postoji u strukturi i to sa većim brojem ponavljanja, celokupno vreme analize će biti veće.

Sa druge strane, ukoliko tražimo određeni element u nekom podstablu, kao što je za pravilo M6-4-6 potrebno proveriti postojanje i poziciju klauze *default*, uparivanje se ne zaustavlja dok ne pronađe tu klauzu ili ne obiđe celo podstablo. Ovde postoji obrnuta situacija, da nepostojanje traženog elementa iziskuje da veći broj čvorova bude posećen.

Pravila tipa poput A6-4-1, za barem dve klauze (koje nisu klauze *default*), zahtevaju neku vrstu brojanja koja je strana za deklarativni zapis uparivača. Ova situacija se može prevazići trikom koji je upotrebljen u implmentaciji, gde se zahteva da druga uparena klauza ne bude jednaka prvoj. Međutim, tu se javlja problem u efikasnosti izvršavanja. Prilikom uparivanja druge klauze ponovo se započinje obilazak od istog korenog čvora. To znači da se ponovno obilazi deo stabla koji je posećen prilikom uparivanja prve klauze. Kada pretraga stigne do prve klauze, proverava se uslov da ona ne bude jednaka prvoj koji nikada ne može biti ispunjen i služi da se preskoči taj čvor. Sve ovo se efikasnije može obaviti posetiocem, gde se usput, tokom jedinog obilaska, izbroje takve klauze.

Složene i nezgodne provere za uparivače se uvek mogu izdvojiti u nove samostale uparivače definisane preko nekog od makroa koje Clang nudi. To je učinjeno u nekoliko situacija prilikom implementacije pravila M6-4-5. Na primer, kada je bio potreban pristup metodi *hasSubStmt()* klase *SwitchCase*, a nije postojao odgovarajuću ugrađeni uparivač. Drugi primer je pomoćni uparivač *afterBoundNode* koje je korišćen za poređenje redosleda čvorova koji nisu nužno jedan drugome predak ili

potomak.

Kako su uparivači nadograđeni na posetioce, sledi da se uparivač može prevesti u posetilac. Slično tome, sa novim samostalnim uparivačima mogu se nadoknaditi nedostaci ugrađenih. Takođe se mogu obaviti i ranija uparivanja i vezivanja, kojima se kasnije i složenije provere odlažu za obradu klase *Callback*, pa može se i posetilac do određene mere prevesti u uparivač. U tom slučaju se postavlja sledeće pitanje: koji pristup je efikasniji u smislu vremena izvršavanja?

## Raniji rezultati

Ranija istraživanja su pokazala da veliki broj dodatnih dijagnostičkih provera, od oko 160 pravila standarda, može da zahteva i preko 30% dodatnog vremena izvršavanja u odnosu na vreme potrebno da se izgradi apstraktno sintaksno stablo (odnosno da se izvrši leksička, sintaksička i semantička analiza, bez daljeg generisanja međureprezentacije) [13]. Takođe je pokazano i da je značajan deo vremena utrošen na ispis samih dijagnostičkih poruka, koje se za kôd koji nije pisan po standardu, mogu javiti u velikom broju. Ukoliko se eliminše ovaj parametar, vreme utrošeno na samu analizu iznosi oko 19% [13].

Takođe su vršena i ranija poređenja efikasnosti između posetilaca i uparivača. U jednom od njih, napisani su identični posetioci i uparivači i pokrenuti na kodu koji u različitim merama sadrži tražene strukture [30]. U slučaju nedostatka tražene strukture, nije primećena razlika u efikasnosti, dok je ona rasla sa procentom učestalosti strukture u kodu. Izvršavanje je bilo sporije od 1.2 do 1.5 puta za kôd koji je sadržao 5% odgovarajuće strukture, a za primere koji su sadržali samo traženu strukturu u kodu i od 3.1 do 5.1 puta sporije.

## Eksperimentalni rezultati

Pravila koja su implementirana su sva vezana za naredbe *switch*, pa je poželjno izvršiti i testiranje nad kodom ili projektom koji sadrži veliki broj takvih naredbi. Kao prirodan izbor možemo uzeti sam projekat Clang, čije biblioteke *Lex*, *Parse* i *Sema* sadrže ove naredbe u značajnom broju.

Alat je pokrenut nad svakom datotekom *.cpp* i rezultati su grupisani po bibliotekama. Izvršeno je pokretanje sa opcijom za ignorisanje zaglavlja kao i sa proverom svih korisničkih zaglavlja. Sistemska zaglavlja su uvek ignorisana, jer obično ne postoje namere da se tamo vrše izmene i takva upozorenja ne bi bila od koristi. Iz

rezultata u tabeli 4.2 se može videti da ja detektovan veliki broj prekršaja implementiranih pravila. Treba naglasiti da dosta veći broj upozorenja prilikom pokretanja sa proverama zaglavlja proističe iz toga da su ona ponovljena za svaki fajl *.cpp* koji ih uključuje.

Tabela 4.2: Broj naredbi *switch*, klauza i prijavljenih upozorenja po biblioteci

| | libLex | libParse | libSema | ukupno |
|---|---|---|---|---|
| Ne uključujući zaglavlja | | | | |
| Broj naredbi *switch* | 59 | 115 | 724 | 898 |
| Broj klauza | 614 | 2231 | 11736 | 14581 |
| Broj prijavljenih upozorenja | 417 | 1300 | 5503 | 7220 |
| Uključujući zaglavlja | | | | |
| Broj naredbi *switch* | 822 | 1405 | 5399 | 7626 |
| Broj klauza | 5526 | 14751 | 75176 | 95453 |
| Broj prijavljenih upozorenja | 7856 | 16929 | 89940 | 114725 |

Najveći broj upozorenja je za pravilo M6-4-3, oko 50% u proseku, što nije iznenađujuće, jer ono zapravo sadrži provere koje su identične sa tri druga pravila. Pravila M6-4-4 i M6-4-7 ne prijavlju ni jedno upozorenje. Procenti pojavljivanja za biblioteke *Lex*, *Parse* i *Sema* se mogu videti u tabeli 4.3.

Tabela 4.3: Procenat prijavljenih upozorenja po pravilu za biblioteke *Lex*, *Parse* i *Sema*

| Broj pravila | Procenat pojavljivanja |
|---|---|
| M6-4-3 | 50% |
| M6-4-5 | 45% |
| M6-4-6 | 4,6% |
| A6-4-1 | <1% |
| M6-4-4 | 0% |
| M6-4-7 | 0% |

Provere su obavljne i uz pomoć posetilaca i uz pomoć uparivača na velikom broju biblioteka projekta i upozorenja koje prozvode su identična i po broju i po lokacijama nad kojima su prijavljena. Ova vrsta provere je korišćena tokom implementacije kako bi se uskladila upozornja i sa velikom količinom pouzdanja može se tvrditi da su analize koje posetioci i uparivači vrše identične.

Usled malog broja podržanih pravila standarda, ukupan uticaj dela analize na celokupan alat neće biti značajan. Ovo je bolje oslikano ranijim radovima koji su

opisivali veći broj pravila [13, 30]. Radi potpunosti, ova evaluacija je ipak obavljena tako što je izvršeno 100 iteracija upotrebe implementiranih analiza i upoređeno vreme koja alat provede izvršavajući samo posetioce ili samo uparivače nasuprot celukupnog vremena izvršavanja. U situacijama kada se ne proveravaju zaglavlja, celokupno vreme utrošeno na analizu sa posetiocima je $< 1\%$, dok za je za uparivače to vreme između 1% i 2%. Kada se uključe provere korisničkih zaglavlja to vreme raste na 6% u proseku za posetioce i 12% u proseku za uparivače. Vreme predstavljeno je relativno vreme u odnosu na izvršavanje celokupnog alata koji podrazumeva parsiranje izvornog koda i izgradnju apstraktnog sintaksnog stabla.

Korisniji rezultati od prethodnih se mogu dobiti poređenjem vremena koje je potrebno da se izvrši jedan konkretan posetilac nasuprot njemu ekvivalentnog uparivača. Time se bolje oslikava razlika u performansama između posetilaca i uparivača. Ponovo je obaljveno po 100 iteracija za svako pravilo i upoređeno je prosečno vreme između posetioca i uparivača. U svim situacijama posetioci su bili brži u proseku. Većina uparivača je zahtevala između 1,8 i 2,7 puta više vremena. Međutim, uparivač M6-4-5 za terminirajuće klauze (kao i M6-4-3 koji ga sadrži u sebi) odskače od ostalih rezultata time što zahteva čak i do 7,5 puta više vremena nasuprot posetiocu. Puni rezlati za biblioteku *Sema* su prikazani u tabeli 4.4.

Tabela 4.4: Umnožak vremena potrebnog da se izvrši analiza uparivačem nasuprot posetiocu

| Broj pravila | Faktor usporenja |
|---|---|
| M6-4-3 | 17,1 |
| M6-4-4 | 2,1 |
| M6-4-5 | 7,5 |
| M6-4-6 | 2,7 |
| M6-4-7 | 1,8 |
| A6-4-1 | 2,3 |

Osim toga što je uparivač M6-4-5 najsloženiji (ne računajući M6-4-3), njegova velika cena se može objasniti upotrebom opcionih uparivanja. Zbog prirode problema koji zahteva da se provere dve potencijalne lokacije, kako bi se pojednostavio ceo uparivač, upotrebljen je specijalni uparivač *optionally*. Zbog toga je uparivanje svake klauze tehnički uspešno, a rezultat se kasnije obrađuje u klasi *Callback*. To znači da se prilikom uparivanja svake klauze, određeni deo stabla u okolini obavezno ponovo posećuje i to u više navrata. Iako bi se ovaj problem mogao prevazići upotrebom novih samostalnih uparivača, i složenijim izrazom uparivanja, ovo dovodi u pitanje

upotrebu uparivača kao alternative za posetioce. Ukoliko postoji česta potreba za definisanjem novih uparivača, onda se gubi prednost kraćeg zapisa.

Zbog toga što se uparivač M6-4-5 sadrži u M6-4-3, tj. pravilu za dobro formirane naredbe *switch*, kao i još neke druge provere, to pravilo takođe ima značajno veću cenu izvršavanja. Cene se akumuliraju i za pojedine primere imamo i do 17,1 puta sporije izvršavanje. Sve ovo ukazuje da za složenije provere, uparivači su dosta neefikasnije rešenje.

Uparivači se mogu upotrebiti za neke manje složene provere ili alate koji ne sadrže ili barem ne obavljaju veliki broj provera istovremeno. U situacijama gde se zahteva interfejs za analizu koda koji nije samo izražajan i pouzdan, već i daje dobre performanse, Clang-ovi AST-posetioci se pokazauju kao bolje rešenje.

# Glava 5

# Zaključak

U ovom radu su prikazani razni problemi koji se mogu javiti prilikom pisanja programa u jeziku C++. Zbog svoje sintakse, relativno je lako napraviti greške prilikom programiranja koje i dalje proizvode validan kôd. Zbog svoje izražajnosti, često se može napisati kôd u jeziku C++ koji je čudan ili težak za razumevanje. Opisani su primeri koji doprinose situacijama gde se mogu javiti greške u izvršnom programu. Greške mogu nastati od programera koji pišu kôd koristeći konstrukte jezika koje ne razumeju u potpunosti. Takođe se greške mogu javiti i zbog upotrebe različitih kompilatora koji pojedine delove standarda koji nisu precizno definisani implementiraju na različite načine.

Kao jedno rešenje predstavljeni su standardi kodiranja, njihove prednosti i mane, kada su poželjni i šta ih čini dobrim. Kao primer su izdvojeni standardi kodiranja MISRA C++ 2008 i AUTOSAR C++14. Oni čine skup pravila koja programer mora ispoštovati sa ciljem da se dobije pouzdan i stabilan kôd. Bezbednost se naglašava kao jedan od bitnih razloga postojanja samog standarda.

Kroz opise projekta LLVM kao i potprojekta Clang opisana je kompilatorska infrastruktura koju čine više bilbioteka i alata. Oni se mogu iskoristiti zarad dizajniranja i implementiranja daljih alata, a kroz opis potprojekta Clang opisane su razne mogućnosti koje njegove biblioteke pružaju. Poseban značaj je dat prednjem delu koji se bavi generisanjem apstraktnog sintaksnog stabla od izvornog koda, kao i mehanizmima koji pomažu u analizi takvog stabla.

Prikazano je kako se pomoću biblioteke *LibTooling* može napraviti novi alat koji vrši takve analize. Opisane su razne mogućnosti koje nude AST-uparivači i AST-posetioci. Kroz impelementaciju je prikazano da se pouzdano mogu dodati razne provere saglasnosi sa standardom kodiranja. Analizirana su oba pristupa, imple-

mentacija posetioca i implementacija uparivača i predstavljene mogućnosti koje oba nude, kao i strategije koje je najbolje primeniti u zavisnosti od prirode problema. Ukoliko se proveravaju čvorovi stabla koji su pretci ili potomci jedni drugima, tada se uparivači mogu relativno jednostavno definisati. Međutim, ukoliko to nije slučaj, čvorovi su susedi ili delovi različitih podstabala, ili je možda potrebna neka vrsta prebrojavanja čvorova, tada su posetioci značajno jednostavniji za upotrebu. Utvrđeno je da su oba pristupa dovoljno izražajna da pokriju podskup pravila standarda AUTOSAR koji se odnosi na naredbu *switch* i to na efikasan način.

Analizirana je efikasnost provera saglasnosti koda sa pravilima standarda koje su implementirane preko posetilaca i uparivača. Eksperimentalnom evaluacijom je utvđrno da se alat implementiran upotrebom Clang-ovih biblioteka pouzdano može iskoristiti i za veće projekte sa zadovoljavajućim performansama. Utvrđeno je da je vreme izvršavanja u odnosu na vreme parsiranja koda i izgradnju apstraktnog sintaksnog stabla, prihvatljivo i obećavajuće u smislu implementacije većeg broja provera saglasnosti sa pravilima standarda. Vreme koje alat utroši za proizvodnju stabla od izvornog koda je ekvivalentno vremenu koje sam kompilator Clang utroši u istu svrhu jer je stablo izgrađeno pomoću istih biblioteka.

Predstavljeno je i poređenje između posetioca i uparivača. Dok uparivači mogu biti prirodnije rešenje u određenom domenu problema, kada su u pitanju performanse, pokazali su se u najboljem slučaju kao skoro duplo sporiji od posetilaca i ne preporučuju se za složenije analize.

## Moguća unapređenja

Očigledno unapređenje alata je podrška za provere saglastnosti sa dodatnim pravilima. Određen skup pravila standarda AUTOSAR se odnosi na pretprocesorske direktive, a one se obrade i uklone iz koda tokom ranih faza kompilacije i apstraktno sintaksno stablo neće imati informacije o njima. Clang pruža dodatne mehanizme za obradu ovih elemenata koji se mogu upotrebiti u alatu izgrađenom pomoću biblioteke *libTooling*, zajedno sa posetiocima i uparivačima. Pored toga alat može koristiti i Clang-ov statički analizator radi vršenja složenih analiza.

Dijagnostika, koju alat *switch-check* generiše, se može unaprediti na više načina. Jedan od njih je u ispisu samih poruka koje alat prijavljuje. U slučajevima gde imamo značajan broj prekršaja istog pravila, može se ograničiti ispis na određen broj kako se korisnik ne bi opteretio velikim brojem sličnih upozorenja. Ovo se

takođe može efikasnije obaviti ranijim zaustavljanjem obilaska stabla radi uštede na vremenu izvršavanja.

Sama klasa za obradu dijagnostike, tj. potrošač dijagnostike se može dopuniti na razne načine. Umesto ukupnog broja upozorenja može se prijaviti i broj prekršaja po svakom pravilu. Može se promeniti sam format ispisa, na primer, može se generisati izveštaj sa listom prekršaja u formatu *.json* i sačuvati u nekoj novoj datoteci.

Prilikom prijavljivanja dijagnostičkih poruka, moguće je generisanje i nagoveštaja za popravku (eng. *FixItHint*). Clang sadrži mehanizme koje mogu uzeti ove vrste nagoveštaja i u skladu sa njima obaviti transformaciju izvornog koda (*source-to-source transformation*). Isti mehanizam se može ugraditi i u alat izgrađen sa bibliotekom *libTooling*. Za neka od pravila kod kojih se jasno može utvrditi potrebna izmena koda koja bi ga učinila saglasnim, moguće je automatski uskladiti kôd sa standardom prilikom pokretanja alata.

# Literatura


[1] *ACM Awards - Celebrating Innovation and Recognizing Achievements and Lasting Contributions*. 2012. URL: https://awards.acm.org/.

[2] Alfred V. Aho, Ravi Sethi i Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0-201-10088-6.

[3] Andrew Ettles, Andrew Luxton-Reilly, Paul Denny. „Common logic errors made by novice programmers". U: *the 20th Australasian Computing Education Conference* (2018). URL: https://www.researchgate.net/publication/322352077_Common_logic_errors_made_by_novice_programmers.

[4] AUTOSAR. *autosar.org*. URL: https://www.autosar.org/about/history/.

[5] "Autosar, Guidelines for the use of the C++14 language in critical and safety-related systems". English. In: (2017).

[6] Bjarne Stroustrup, Herb Sutter. „C++ Core Guidelines". U: (). URL: http://stroustrup.com/JSF-AV-rules.pdf.

[7] Bruno Cardoso Lopes, Rafael Auler. *Getting started with LLVM core libraries*. Packt Publishing, 2014.

[8] *C++ Standards Support in GCC*. 2022. URL: https://gcc.gnu.org/projects/cxx-status.html.

[9] *C++14 Features Supported by Intel® C++ Compiler Classic*. 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/c14-features-supported-by-intel-c-compiler.html.

[10] Chris Lattner. *The Architecture of Open Source Applications*. Jan. 2011, str. 155–170.

[11] CompCert project. *The CompCert C compiler*. URL: http://compcert.inria.fr/compcert-C.html.

[12] PhD David B. Stewart. „Twenty-Five Most Common Mistakes with Real-Time Software Development". U: *Embedded Systems Conference* (2006). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.113.6245&rep=rep1&type=pdf.

[13] Đorđe Milićević, Mirko Brkušanin, Milena Vujošević Janičić, Teodora Novković, Petar Jovanović. „Improving Clang compiler with MISRA/AUTOSAR coding standard support". U: *63. Konferencija za elektroniku, telekomunikacije, računarstvo, automatiku i nuklearnu tehniku (ETRAN)* (Jun 2019).

[14] Florent Brissaud, Didier Turcinovic. „Functional Safety for Safety-Related Systems: 10 Common Mistakes". U: *25th European Safety and Reliability Conference* (2015).

[15] Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, David Svoboda. *Java Coding Guidelines, 75 Recommendations for Reliable and Secure Programs*. 2013.

[16] Günter Obiltschnig. „C++ for Safety-Critical Systems". U: (2012). URL: https://www.researchgate.net/publication/228737242_C_for_Safety-Critical_Systems.

[17] Wolfgang A. Halang i Janusz Zalewski. „Programming languages for use in safety-related applications". U: *Annual Reviews in Control* 27.1 (2003), str. 39–45. ISSN: 1367-5788. DOI: https://doi.org/10.1016/S1367-5788(03)00005-1. URL: https://www.sciencedirect.com/science/article/pii/S1367578803000051.

[18] Hans-J. Boehm, Alan Demers, Mark Weiser. *https://hboehm.info/gc/*. URL: https://hboehm.info/gc/.

[19] Hans-J. Boehm, Mike Spertus. „Transparent Programmer-Directed Garbage-Collection for C++". U: (2007). URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2310.pdf.

[20] Hans-J. Boehm, Mike Spertus, Clark Nelson. „Minimal Support for Garbage Collection and Reachability-Based Leak Detection (revised) - ISO/IEC JTC1 SC22 WG21 N2670". U: (2008). URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2670.htm.

[21] Herb Sutter, Andrei Alexandrescu. *C++ Coding Standards. 101 Rules, Guidelines, and Best Practices (1st edition)*. Addison-Wesley Professional, 2004. URL: https://doc.lagout.org/programmation/C/CPP101.pdf.

[22] „ISO/IEC 14882:2003, Programming Languages — C++, International Organization for Standardization". U: (2003). URL: https://www.iso.org/standard/38110.html.

[23] „ISO/IEC 14882:2014, Programming Languages — C++, International Organization for Standardization". U: (2014). URL: https://www.iso.org/standard/64029.html.

[24] „ISO/IEC 14882:2017, Programming Languages — C++, International Organization for Standardization". U: (2017). URL: https://www.iso.org/standard/68564.html.

[25] Lawrence Crowl, Chandler Carruth, Richard Smith. „Clarifying Memory Allocation - ISO/IEC JTC1 SC22 WG21 N3664". U: (2013). URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3664.html.

[26] Les Hatton, Computing Laboratory, University of Kent at Canterbury. „Safer Language Subsets: an overview and a case history, MISRA C". U: (2003).

[27] *List of LLVM Related Publications*. 2018. URL: https://llvm.org/pubs/.

[28] *List of Talks and Related Publications*. 2022. URL: https://circt.llvm.org/talks/.

[29] Lockheed Martin Corporation. „Joint Strike Fighter. Air Vehicle. C++ Coding Standards For The System Development and Demonstration Program". U: (). URL: http://stroustrup.com/JSF-AV-rules.pdf.

[30] Milena Vujošević Janičić, Ognjen Plavšić, Mirko Brkušanin, Petar Jovanović. „AUTOCHECK: A Tool For Checking Compliance With Automotive Coding Standards". U: *2021 Zooming Innovation in Consumer Technologies Conference (ZINC)*. 2021, str. 150–155. DOI: 10.1109/ZINC52049.2021.9499304. URL: https://ieeexplore.ieee.org/document/9499304.

[31] "MISRA, MISRA – The Motor Indistry Software Reliability Association". English. In: (2008). URL: https://www.misra.org.uk/.

[32] National Institute of Standards and Technology. „Strategic Planning, "The economic impacts of inadequate infrastructure for software testing,"" u: (2002).

[33] *Standard C++ Foundation, Coding Standards*. URL: `https://isocpp.org/wiki/faq/coding-standards`.

[34] Steve McConnell. *CodeComplete, A Practical handbook of software construction*. Microsoft, 2004.

[35] Steve Yohanan. „Code Conventions for Java". U: (2000).

[36] Suyog Sarda, Mayur Pandey. *LLVM essential*. Packt Publishing, 2018.

[37] *The LLVM Compiler Infrastructure*. 2019. URL: `https://llvm.org/`.

[38] The Motor Industry Research Association. „MISRA. Development Guidelines for Vehicle Based Software". U: (1994). URL: `https://pdfs.semanticscholar.org/f7ab/6319c0ab1fb546e406a3f65463cd0e5d2f0c.pdf`.

# Biografija autora

**Mirko Brkušanin** rođen je 17.05.1994 u Kraljevu. Završio je osnovnu školu kao nosilac Vukove dilpome, a potom upisuje Matematičku gimnaziju u Kraljevu. Smer informatika na Matematičkom fakultetu Univerziteta u Beograd upisao je 2013. godine, a diplomirao 2016. godine sa prosečnom ocenom 9,00. Odmah nakon toga upisuje master studije na istom fakultetu. Od 2018. godine radi kao softverski inženjer u kompilatorskom timu za firmu *RT-RK*, prvo na internim projektima za Clang, a potom na kompilatoru LLVM za arhitekturu MIPS. Od 2020. godine je zaposlen u firmi *Syrmia* gde sarađuje sa zaposlenima iz firme *AMD Inc.* na podršci kao i implementaciji novih generacija arhitekture za AMD grafičke kartice unutar *shader* kompilatora.

Učestvovao je u izradi dva rada na temu unapređenja kompilatora Clang i podrške za standard AUTOSAR.

1. Prvi je „*Unapređenje programskog prevodioca Clang sa podrškom za standard MISRA/AUTOSAR*" koji je prezentovan na konferenciji ETRAN 2019. godine i koji je nagrađen kao najbolji rad u oblasti računarstva.

2. Drugi rad je „*AUTOCHECK : A Tool For Checking Compliance With Automotive Coding Standards*" koji je predstavljen na konferenciji *Zooming Innovation in Consumer Technologies Conference (ZINC)* 2021. godine.