

Analiza i Verifikacija Softvera

Mini Student Management System

Nikola Stojanović 1024/2024
Matematički fakultet, Univerzitet u Beogradu

21. mart 2025.

Sadržaj

1	Uvod	3
2	Jedinični testovi i analiza pokrivenosti koda	3
2.1	Analiza rezultata testiranja i pokrivenosti koda	3
3	Statička i stilska analiza koda	6
3.1	Korišćenje Clang-Tidy alata	6
3.2	Analiza rezultata	6
3.2.1	Curenje memorije (Memory Leaks)	6
3.2.2	Neinicijalizovane promenljive (Uninitialized Variables) .	7
3.2.3	Nepravilno oslobađanje memorije (Mismatched Deal- locator)	7
3.2.4	Problemi sa <code>nullptr</code> (modernize-use-nullptr)	7
3.2.5	Korišćenje <code>auto</code> (modernize-use-auto)	8
4	Dinamička analiza koda	8
4.1	Valgrind	8
4.2	Memcheck	9
4.2.1	Analiza rezultata	9
4.2.2	Tipovi curenja memorije	10
4.2.3	Ostali problemi	11
4.2.4	Preporuke za optimizaciju	11
4.3	Callgrind	11
4.3.1	Analiza rezultata	11
4.3.2	Preporuke za optimizaciju	13

1 Uvod

Za potrebe kursa Verifikacija Softvera, analiziran je projekat *Mini Student Management System*. Projekat je razvijen u *C++* jeziku. Ova aplikacija pruža funkcionalnosti neophodne za upravljanje akademskim podacima, kao što su kreiranje i uređivanje predmeta, vođenje evidencije prisustva studenata, upravljanje profesorima i studentima, kao i izvoz podataka u CSV format.

Analiza kvaliteta koda i ispitivanje njegovih karakteristika postignuta je korišćenjem raznih alata za testiranje, dinamičku i statičku analizu softvera. Napisani su jedinični testovi i primenjeni *Lcov*, *Valgrind* i *Clang* alati.

Glavni cilj ove analize je da se proceni pouzdanost i efikasnost ovog softvera, kao i da se identifikuju mogući nedostaci u njegovoj implementaciji. Na osnovu dobijenih rezultata biće predložene potencijalne optimizacije i poboljšanja, sa ciljem da se unapredi kvalitet i stabilnost koda.

Pošto je aplikacija inicijalno razvijena za Windows operativni sistem, a kako je podrška za alate za verifikaciju softvera na Windowsu ograničena, bilo je neophodno prilagoditi kod za pokretanje na Linux sistemu. U tu svrhu, izvršene su minimalne izmene u kodu, u vidu zamene Windows sistemskih poziva odgovarajućim sistemskim pozivima za Linux, kako bi aplikacija mogla biti uspešno kompajlirana i analizirana na Linux okruženju.

2 Jedinični testovi i analiza pokrivenosti koda

Jedinično testiranje (*Unit Testing*) je tehnika u softverskom inženjerstvu koja podrazumeva testiranje pojedinačnih komponenti programa, obično na nivou funkcija ili klasa. Cilj jediničnih testova je da se proverí ispravnost svake jedinice koda izolovano od ostatka sistema, čime se omogućava rano otkrivanje grešaka i pojednostavljuje proces ispravljanja.

Za potrebe ovog rada, jedinični testovi su implementirani korišćenjem biblioteke *Catch2*, koja predstavlja moderni C++ framework za testiranje sa jednostavnim sintaksom i automatskim otkrivanjem testova. *Catch2* se pokazao kao pogodan alat za pisanje i organizaciju testova, bez potrebe za posebnom konfiguracijom ili dodatnim zavisnostima.

2.1 Analiza rezultata testiranja i pokrivenosti koda

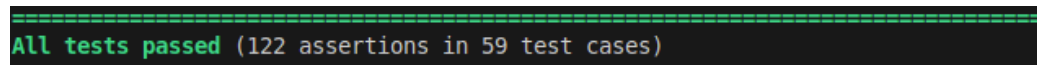
S obzirom na to da je aplikacija koju analiziramo implementirana za Windows operativni sistem i koristi njegove sistemske pozive koji blokiraju ulazno/izlazne operacije, za potrebe automatizacije izvršavanja jediničnih testova bilo je neophodno dodati opciono isključenje ovih poziva uz

pomoć preprocesorskih direktiva koje ih isključuju korišćenjem opcije **-D UNIT_TESTING** prilikom prevodjenja.

```
1  #ifndef UNIT_TESTING
2      #ifdef _WIN32
3          system("pause");
4      #else
5          system("read -p \"Press Enter to continue
6              ...\" dummy");
7      #endif
8  #endif
```

Izvorni kod za jedinične testove smešten je u *src* direktorijumu, dok su propratni header fajlovi u *include* direktorijumu. Ovakva modularna struktura projekta olakšava izolovano testiranje pojedinačnih komponenti i omogućava jednostavno dodavanje novih testova po potrebi.

Rezultati testiranja pokazuju da su svi test slučajevi uspešno prošli, što ukazuje na stabilnost implementiranih funkcija u trenutnoj verziji koda. Kolekcija testova obuhvata ukupno 59 test slučajeva, sa 122 tvrdnji, što je prikazano na slici 1.



Slika 1: Prolaznost jediničnih testova

Analiza pokrivenosti koda primenom *LCOV*-a dala je sledeće rezultate:

- **Ukupna pokrivenost linija koda:** 90.9% (2105 linija od ukupno 2317)
- **Ukupna pokrivenost funkcija:** 98.8% (164 funkcije od ukupno 166)

Na slici 2 možemo videti da je ukupna pokrivenost koda preko 90%, što se može smatrati visokim nivoom testiranja u odnosu na veličinu i složenost projekta. Preostalih 9.1% nepokrivenog koda može biti dodatno analizirano kako bi se osigurala potpuna validacija implementacije.

LCOV - code coverage report

Current view: top level		Coverage		Total	Hit
Test: filtered_coverage.info		Lines:	90.9 %	2317	2105
Test Date: 2025-03-19 22:27:14		Functions:	98.8 %	166	164

Directory	Line Coverage ↕			Function Coverage ↕		
	Rate	Total	Hit	Rate	Total	Hit
Mini-Student-Management-System/src	<div><div></div></div> 87.6 %	1715	1503	<div><div></div></div> 98.0 %	100	98
unit_tests/tests	<div><div></div></div> 100.0 %	602	602	<div><div></div></div> 100.0 %	66	66

Slika 2: Ukupna pokrivenost linija i funkcija

U nastavku su prikazani detalji o pokrivenosti koda u glavnim direktorijumima projekta:

- **unit_tests** direktorijum: 100% (602 od 602 linije) i (66 od 66 funkcija)
- **src** direktorijum: 87.6% (1503 od 1715 linija) i 98.0% (98 od 100 funkcija)

Filename	Line Coverage ↕			Function Coverage ↕		
	Rate	Total	Hit	Rate	Total	Hit
attendancelist.cpp	<div><div></div></div> 85.5 %	214	183	<div><div></div></div> 92.3 %	13	12
class.cpp	<div><div></div></div> 82.6 %	201	166	<div><div></div></div> 90.9 %	11	10
courses.cpp	<div><div></div></div> 87.2 %	635	554	<div><div></div></div> 100.0 %	30	30
csv_import.cpp	<div><div></div></div> 100.0 %	21	21	<div><div></div></div> 100.0 %	2	2
date.cpp	<div><div></div></div> 88.4 %	172	152	<div><div></div></div> 100.0 %	12	12
interface.cpp	<div><div></div></div> 88.4 %	198	175	<div><div></div></div> 100.0 %	17	17
lecturer.cpp	<div><div></div></div> 88.5 %	122	108	<div><div></div></div> 100.0 %	7	7
scoreboard.cpp	<div><div></div></div> 94.7 %	152	144	<div><div></div></div> 100.0 %	8	8

Slika 3: Pokrivenost *src* direktorijuma

Što se tiče preostalih 2% pokrivenosti funkcija, na slici 3, se može primetiti da su fajlovi `attendancelist.cpp` i `class.cpp` jedini sa manje od 100% pokrivenosti funkcija.

U fajlu `attendancelist.cpp`, funkcija `edit_attendance()` služi kao *wrapper funkcija* za dve specijalizovane funkcije: `edit_attendance_by_course()` i `edit_attendance_by_student()`. Ove funkcije su već detaljno testirane, pa bi dodavanje posebnog testa za `edit_attendance` bilo suvišno.

Sa druge strane, u fajlu `class.cpp`, funkcija `update_users()` nije pokri-vena testovima jer nigde nije korišćena u ostatku projekta. S obzirom na to da funkcija nije deo aktivne logike aplikacije, nije imalo smisla uključivati je u testiranje.

Ovi podaci pokazuju da je kod visoko pokriven testovima i da su ključne funkcionalnosti aplikacije detaljno ispitane. Nepokrivene funkcije ne predstavljaju značajan rizik po stabilnost sistema, jer ili nisu deo aktivnog koda ili su već implicitno testirane kroz druge funkcije.

3 Statička i stilaska analiza koda

Statička analiza je tehnika za automatsko ispitivanje koda bez njegovog izvršavanja, sa ciljem detekcije grešaka, potencijalnih problema i sigurnosnih propusta u implementaciji. Pored toga, stilaska analiza se koristi za procenu čitljivosti i usaglašenosti koda sa preporučenim standardima.

U ovom radu, za statičku i stilsku analizu korišćen je alat *Clang-Tidy*. Ovaj alat omogućava analizu C++ koda na osnovu različitih pravila i preporuka, pružajući uvid u potencijalne greške u implementaciji.

3.1 Korišćenje Clang-Tidy alata

Alat *Clang-Tidy* je pokrenut nad svim .cpp fajlovima projekta koristeći `run_clang_tidy.sh` skriptu za automatizaciju, sa sledećim proveravačima:

Listing 1: Pokretanje Clang-Tidy alata

```
1 --checks=clang-diagnostic-*,clang-analyzer-*,modernize-  
  use-auto,modernize-use-nullptr,modernize-use-noexcept,  
  modernize-use-emplace,modernize-use-emplace-back,  
  modernize-loop-convert,modernize-use-using
```

Ova komanda koristi `compile_commands.json` generisan uz pomoć alata `bear`, koji omogućava precizno praćenje svih kompilacionih opcija.

3.2 Analiza rezultata

U ovom poglavlju se prikazuju različiti problemi detektovani u kodu korišćenjem alata *Clang-Tidy*. Za svaki problem se daje opis trenutnog stanja, preporučena izmena i primer koda nakon ispravke.

3.2.1 Curenje memorije (Memory Leaks)

Opis problema: Potencijalno curenje memorije se javlja kada se dinamički alocirana memorija ne oslobađa pravilno. Ovi problemi su detektovani u fajlovima: `attendancelist.cpp` i `csv.h`.

Problematičan kod (iz `csv.h`):

Listing 2: Nepravilno oslobađanje memorije

```
1 Pchar buf = new Char[2];
```

Preporučeno rešenje: Zameniti manuelnu alokaciju memorije sa pametnim pokazivačima.

Izmenjeni kod:

```
1 auto buf = std::make_unique<Char[]>(2);
```

3.2.2 Neinicijalizovane promenljive (Uninitialized Variables)

Opis problema: Neinicijalizovane promenljive mogu izazvati nepredvidivo ponašanje programa.

Problematičan kod (iz attendancelist.cpp):

Listing 3: Neinicijalizovana promenljiva

```
1 int chosen;  
2 std::cin >> chosen;
```

Preporučeno rešenje: Uvek inicijalizovati promenljive pri deklaraciji.

Izmenjeni kod:

```
1 int chosen = 0;  
2 std::cin >> chosen;
```

3.2.3 Nepravilno oslobađanje memorije (Mismatched Deallocator)

Opis problema: Korišćenje `delete[]` za memoriju alociranu sa `new` ili obrnuto. Ovo je problematično jer uzrokuje rušenje programa ili curenje memorije.

Problematičan kod (iz csv.h):

Listing 4: Nepravilno oslobađanje memorije

```
1 delete[] root->next;
```

Preporučeno rešenje: Korišćenje odgovarajuće sintakse za oslobađanje memorije.

Izmenjeni kod:

```
1 delete root->next;
```

3.2.4 Problemi sa nullptr (modernize-use-nullptr)

Opis problema: Korišćenje 0 ili NULL umesto nullptr smanjuje čitljivost i može izazvati probleme u radu sa pokazivačima.

Problematičan kod (iz date.cpp):

Listing 5: Korišćenje 0 umesto nullptr

```
1 std::time_t t = std::time(0);
```

Preporučeno rešenje:

```
1 std::time_t t = std::time(nullptr);
```

3.2.5 Korišćenje auto (modernize-use-auto)

Opis problema: U kodu se koriste eksplicitno deklarirani tipovi koji se mogu zameniti sa `auto` za bolje čitljivost i smanjenje šanse za grešku.

Problematičan kod (iz `interface.cpp`):

Listing 6: Korišćenje eksplicitnog tipa umesto auto

```
1 std::ctype<char> const& ctype(std::use_facet<std::ctype<char>>(in.getloc()));
```

Preporučeno rešenje: Korišćenjem `auto` smanjuje se količina koda i poboljšava čitljivost.

```
1 auto const& ctype = std::use_facet<std::ctype<char>>(in.getloc());
```

4 Dinamička analiza koda

Dinamička analiza koda predstavlja tehniku testiranja i analize softverskih aplikacija pri njihovom izvršavanju. Za razliku od statičke analize koja analizira kod bez njegovog izvršavanja, dinamička analiza omogućava detekciju grešaka koje se javljaju tokom samog rada programa. Ova metoda omogućava identifikaciju problema kao što su curenje memorije, pristup neinicijalizovanim promenljivima, nedozvoljen pristup memoriji (*buffer overflow*), neoptimalno korišćenje memorije, preklapanje memorijskih resursa itd.

Dinamička analiza se koristi za unapređivanje pouzdanosti i efikasnosti aplikacija, posebno onih razvijenih u jezicima kao što su *C* i *C++* koji omogućavaju manuelno upravljanje memorijom.

4.1 Valgrind

Jedan od najkorišćenijih alata za dinamičku analizu *C/C++* programa je *Valgrind*. Valgrind je platforma za instrumentaciju softvera koja omogućava detaljno praćenje izvršavanja programa i detekciju različitih vrsta grešaka.

Prednosti Valgrind-a:

- Omogućava detekciju curenja memorije u realnom vremenu.
- Pronalazi neinicijalizovane ili nepropisno korišćene promenljive.
- Otkriva nepravilno korišćenje funkcija za oslobađanje memorije (`delete` umesto `delete[]`, ili obrnuto).
- Prati neoptimalno korišćenje memorije i procesorskih resursa.

Mane Valgrind-a:

- Značajno usporava izvršavanje programa (često i do 10 puta).
- Ne može da detektuje logičke greške u kodu, već samo greške vezane za memoriju i performanse.
- Može prijaviti lažno pozitivne greške ukoliko kod nije pravilno kompajliran sa potrebnim zastavicama za debugovanje.

4.2 Memcheck

Memcheck je osnovni alat unutar *Valgrind* platforme koji omogućava detekciju problema vezanih za rad sa memorijom tokom izvršavanja programa. Njegova glavna namena je detekcija curenja memorije, pristupa neinicijalizovanoj memoriji i nepravilnog oslobađanja memorijskih resursa.

Analiza je izvršena pokretanjem *run_memcheck.sh* skripte sa opcijama:

```
1 --leak-check=full --show-leak-kinds=all --track-origins=
  yes
```

4.2.1 Analiza rezultata

Na osnovu **HEAP Summary** sekcije možemo uočiti ukupnu količinu memorije koja je alocirana i oslobođena tokom izvršavanja programa. Prema generisanom izveštaju, ukupno je alocirano:

- **Alocirano:** 6,440,108 bajtova u 530,341 blokova.
- **Oslobođeno:** 3,956 blokova (što predstavlja svega 0.75% svih alociranih blokova).

- **Preostalo zauzeto nakon izvršavanja programa:** 6,022,418 bajtova u 526,385 blokova.

Ovaj podatak ukazuje na ozbiljno curenje memorije, jer veliki deo memorije ostaje zauzet nakon završetka programa. Šta više, u slučaju dugotrajnog rada programa ili učestalih pokretanja, ovo može značajno povećati potrošnju memorije i znatno usporiti rad samog programa.

4.2.2 Tipovi curenja memorije

U delu **LEAK Summary** sekcije su prikazani različiti tipovi curenja memorije koje alat detektuje:

Definitely Lost Memory Ovo je najkritičniji tip curenja memorije. Prijavljeno je ukupno:

- **Definitely lost:** 3,459,762 bajtova u 308,048 blokova.

Ova memorija je alocirana, ali se više ne može dohvatiti nijednim pokazivačem. Ovakvo curenje najčešće se javlja u funkcijama koje koriste dinamički alocirane nizove ili složene strukture.

Indirectly Lost Memory Ovaj tip curenja se odnosi na memoriju koja se ne može direktno dohvatiti, ali je povezana sa drugim blokovima memorije koji su definitivno izgubljeni. Prijavljeno je:

- **Indirectly lost:** 2,537,232 bajtova u 216,362 blokova.

Ovakva curenja su uglavnom rezultat složenih struktura u kojima se glavna struktura oslobađa bez oslobađanja svih njenih delova. U ‘csv.h’ biblioteci, mnoge funkcije alociraju memoriju koju kasnije ne oslobađaju pravilno.

Still Reachable Memory Memorija koja je i dalje dostupna, ali nije oslobođena pre završetka programa. Prijavljeno je:

- **Still reachable:** 25,424 bajtova u 1,975 blokova.

Ovaj tip curenja memorije najčešće potiče iz globalnih promenljivih ili dinamički alocirane memorije koja se zadržava do završetka programa. Iako nije kritično, može predstavljati problem ako se program koristi duže vreme.

4.2.3 Ostali problemi

Memcheck je takođe prijavio nekoliko slučajeva **Mismatched free/delete/delete[]** grešaka. Najčešći uzrok ovih grešaka su:

- Korišćenje `delete` umesto `delete[]` za oslobađanje memorije alocirane sa `new[]`.
- Korišćenje `free()` umesto `delete[]` ili `delete` za oslobađanje dinamički alocirane memorije.

4.2.4 Preporuke za optimizaciju

Na osnovu dobijenih rezultata, identifikovani su sledeći problemi i predložena su sledeća rešenja:

- **Ispravno oslobađanje dinamički alocirane memorije:** Zamena svih `delete` instrukcija sa `delete[]` gde je to neophodno.
- **Korišćenje pametnih pokazivača:** Upotreba `std::unique_ptr` i `std::shared_ptr` omogućava automatsko upravljanje memorijom i značajno smanjuje verovatnoću curenja memorije.
- **Oslobađanje memorije pre završetka programa:** Preporučuje se da se svi resursi pravilno oslobode pre izlaska iz programa.

4.3 Callgrind

Callgrind je alat iz skupa *Valgrind* alata koji omogućava profilisanje programa sa ciljem analize performansi. Omogućava praćenje broja instrukcija koje se izvršavaju, memorijskih operacija i grananja u programu. Ova analiza omogućava identifikaciju kritičnih delova koda koji najviše utiču na performanse.

4.3.1 Analiza rezultata

Rezultati dobijeni pomoću *Callgrind* alata ukazuju na to da najveći deo izvršenih instrukcija pripada funkcijama iz standardnih biblioteka ili indirektno pozvanim funkcijama iz `csv.h`. Međutim, identifikovane su i funkcije koje su direktno implementirane u okviru projekta, a koje značajno utiču na performanse.

Ukupni broj instrukcija (Ir)

- **Ukupno izvršenih instrukcija:** 246,158,275

Najintenzivniji delovi koda (Hotspots - Sistemske funkcije) Najveći broj instrukcija se izvršava u sledećim funkcijama:

- **malloc.c: `_int_malloc`** - 64,097,518 instrukcija (26.04%)
- **csv.h: `Csv::string_copy`** - 37,472,595 instrukcija (15.22%)
- **strcmp-avx2.S: `__strcmp_avx2`** - 32,157,602 instrukcija (13.06%)
- **csv.h: `Csv::multitype::is_type`** - 24,767,424 instrukcija (10.06%)
- **malloc.c: `malloc`** - 20,293,471 instrukcija (8.24%)

Analiza sistemskih funkcija Veliki procenat izvršenih instrukcija odnosi se na funkcije koje se koriste za alokaciju memorije (`malloc()` i `_int_malloc()`) kao i na poređenja (`strcmp-avx2.S: __strcmp_avx2`). To sugeriše da bi optimizacija koda mogla da se postigne smanjenjem učestalih poziva na dinamičku alokaciju memorije, kao i korišćenjem efikasnijih metoda za poređenje stringova. Funkcija `Csv::string_copy` koristi značajan deo memorijskih resursa zbog čestih kopiranja stringova, što ukazuje na potencijal za optimizaciju.

Najčešće pozivane funkcije u projektu (Call Count) Pored funkcija sa najvećim brojem instrukcija, analizirane su i funkcije koje su najčešće pozivane u okviru projekta. Ove funkcije mogu predstavljati usko grlo performansi ako se pozivaju previše puta unutar različitih operacija.

Funkcija	Datoteka	Broj poziva
<code>Csv::multitype::is_type()</code>	<code>csv.h</code>	1,547,964
<code>Csv::string_copy()</code>	<code>csv.h</code>	387,587
<code>Csv::multitype::to_str()</code>	<code>csv.h</code>	176,082
<code>Csv::multitype::type()</code>	<code>csv.h</code>	174,079
<code>Csv::multitype::assign()</code>	<code>csv.h</code>	145,352
<code>Csv::list<Csv::multitype>::at()</code>	<code>csv.h</code>	75,326
<code>Csv::multitype::init()</code>	<code>csv.h</code>	73,702
<code>Csv::multitype()</code>	<code>csv.h</code>	66,562

Tabela 1: Najčešće pozivane funkcije unutar projekta

4.3.2 Preporuke za optimizaciju

Na osnovu analiza i dobijenih rezultata neki od predloga za optimizaciju su:

- **Smanjenje dinamičkih alokacija memorije:** Veliki deo izvršenih instrukcija dolazi iz funkcija koje koriste `malloc()` i `new[]`, posebno unutar funkcija iz `csv.h` kao što su `Csv::string_copy()`. Preporučuje se:
 - Smanjenje učestalih poziva na `malloc()` kroz unapred alocirane bafer memorije.
 - Korišćenje ‘move’ semantike i ‘`std::unique_ptr`’ za bolju kontrolu nad dinamičkom memorijom.
 - Optimizacija funkcija koje koriste stringove ili strukture kao što su `Csv::multitype::assign()` i `Csv::string_copy()`.
- **Poboljšanje algoritama za učitavanje CSV fajlova:** Funkcije `init_all_csvs()` i `csv_read_util()` dominiraju u broju izvršenih instrukcija. Preporučuje se:
 - Korišćenje više niti za paralelno učitavanje velikih skupova podataka.
 - Implementacija bržih algoritama za parsiranje i sortiranje CSV fajlova.
 - Korišćenje struktura podataka koje omogućavaju brži pristup i obradu.
- **Optimizacija funkcija za manipulaciju podacima:** Najčešće pozivane funkcije (`Csv::multitype::is_type()`, `Csv::string_copy()`, itd.) često se koriste prilikom manipulacije podacima u `csv.h`. Preporučuje se:
 - Keširanje rezultata funkcija koje se često pozivaju sa istim ulaznim vrednostima.
 - Smanjenje broja poziva na funkcije koje često alociraju memoriju.
- **Unapređenje algoritama za prikaz rezultata:** Funkcije `view_scoreboard()` i `edit_scoreboard()` generišu veliki broj instrukcija zbog čestog sortiranja i filtriranja. Preporučuje se:
 - Optimizacija algoritama za sortiranje.

- Unapređenje rada sa listama kroz korišćenje efikasnijih struktura podataka.
- Korišćenje indeksa za brži pristup podacima.

5 Zaključak

Analiza projekta *Mini Student Management System* sprovedena kroz različite alate za verifikaciju i profilisanje ukazala je na nekoliko ključnih aspekata vezanih za kvalitet i stabilnost samog projekta. Korišćenjem jediničnih testova, statičke i dinamičke analize, kao i profilisanja performansi, dobijen je uvid u osnovne prednosti i slabosti implementacije.

Projekat poseduje solidnu osnovu kada je u pitanju funkcionalnost i modularnost, što se može zaključiti na osnovu visokog stepena pokrivenosti koda kroz jedinične testove. Ovaj nivo testiranja jasno pokazuje da su osnovne funkcionalnosti implementirane ispravno i da sistem, u većini slučajeva, adekvatno reaguje na očekivane ulaze. Ipak, postoji određeni procenat koda koji nije pokriven testovima, što ostavlja prostor za potencijalne propuste u budućim verzijama aplikacije.

S druge strane, statička i dinamička analiza pokazale su da projekat ima određenih problema vezanih za efikasno upravljanje memorijom i konzistentno korišćenje pokazivača. Iako se ovi problemi mogu rešiti refaktorisanjem koda i prelaskom na modernije C++ pristupe, primećeno je da značajan deo problema proizilazi iz korišćenja spoljne biblioteke `csv.h`, koja nije razvijena u okviru ovog projekta. To ukazuje na potencijalni rizik koji dolazi sa oslanjanjem na biblioteke koje nisu dovoljno robusne ili prilagođene specifičnim potrebama aplikacije.

Generalno posmatrano, projekat *Mini Student Management System* je funkcionalan i stabilan u većini aspekata, ali zahteva dodatne optimizacije kako bi se postigao željeni nivo efikasnosti i pouzdanosti. Primena modernijih programskih konstrukcija, poboljšanje metoda za upravljanje memorijom i unapređenje algoritama za obradu podataka mogli bi značajno poboljšati performanse i kvalitet aplikacije.