

Analiza projekta Alat za analitiku

Projekat iz Verifikacije softvera

Jelisaveta Gavrilović 1028/2024

9. oktobar 2025.

Sadržaj

1	Uvod	2
2	Valgrind	2
2.1	Instalacija	3
2.2	Memcheck	3
2.2.1	Pokretanje alata	4
2.2.2	Rezultati analize	5
2.3	Massif	6
2.3.1	Pokretanje alata	6
2.3.2	Rezultati analize	7
3	Clang-Format	9
3.1	Instalacija	9
3.2	Pokretanje alata	9
3.3	Analiza rezultata	10
4	Cppcheck	11
4.1	Instalacija	11
4.2	Pokretanje alata	12
4.3	Analiza rezultata	12
4.3.1	Pregled izveštaja <code>stats.html</code>	12
4.3.2	Pregled izveštaja <code>index.html</code>	13
5	Zaključak	15

1 Uvod

Projekat koji je predmet ove analize je [Alat za analitiku](#). Analiza je rađena nad granom `main` i commit-om [2f803aed](#).

Alat za analitiku je softverski alat koji omogućava generisanje izveštaja, statistika i vizualizacija nad podacima. Podaci se učitavaju iz datoteka i, po potrebi, vrši se konverzija tipova. Korisnik može da prikazuje različite grafike, uključujući raspodele, bar plot-ove i analize zavisnosti podataka, kao i da štampa ili čuva izveštaje. Ovaj alat se može posmatrati kao mini verzija alata poput PowerBI-ja, namenjena analizi i vizualizaciji podataka.

Cilj ove analize je da se kroz korišćenje alata za verifikaciju softvera identifikuju potencijalni problemi u kodu i pruže smernice za optimizaciju, poboljšanje stila i standardizaciju koda. Za ovu svrhu su korišćeni sledeći alati:

- **Valgrind – Memcheck** za detekciju curenja memorije,
- **Valgrind – Massif** za praćenje potrošnje memorije,
- **Clang-Format** za proveru i automatsko formatiranje koda,
- **Cppcheck** za statičku analizu koda.

2 Valgrind

Valgrind je okvir (*framework*) za dinamičku analizu programa koji omogućava otkrivanje širokog spektra problema u radu sa memorijom, optimizacijom performansi i praćenje resursa tokom izvršavanja aplikacije. U osnovi, Valgrind radi tako što izvršava program unutar virtuelne mašine koja nadgleda svaku instrukciju i pristup memoriji, čime omogućava detaljno praćenje ponašanja programa na nivou bajta.

Valgrind ne zahteva izmene u izvornom kodu — pokreće se nad već kompajliranim izvršnim fajlom. Zbog toga je posebno koristan u fazi verifikacije i testiranja softvera, jer može da detektuje:

- curenja memorije (*memory leaks*),
- pristup oslobođenoj ili neinicijalizovanoj memoriji,
- nevažeće pokazivače i prepisivanje memorijskih granica,
- nepotpuno oslobađanje resursa (npr. otvoreni fajlovi ili deskriptori),
- neefikasnu ili prekomernu upotrebu memorije.

Valgrind se sastoji od više specijalizovanih alata, koji se biraju parametrom **-tool=** prilikom pokretanja. Najznačajniji među njima su:

- **Memcheck** – detektuje greške u radu sa memorijom (neinicijalizovane promenljive, curenja memorije, nevažeći pokazivači i slično)
- **Massif** – meri i prikazuje potrošnju memorije tokom izvršavanja programa, omogućavajući profilisanje i analizu performansi
- **Cachegrind** – analizira keš memoriju i procese grananja radi optimizacije performansi procesora
- **Callgrind** – prati tok poziva funkcija i meri vreme izvršavanja svake funkcije.
- **Helgrind** i **DRD** – detektuju probleme u višenitnim (multithreaded) aplikacijama

2.1 Instalacija

Na Linux sistemima, Valgrind se može jednostavno instalirati pokretanjem komande u terminalu:

```
sudo apt install valgrind
```

Nakon instalacije, dostupna je osnovna komanda:

```
valgrind --tool=<alat> <izvršni_fajl>
```

gde se, u zavisnosti od izabranog alata, analiziraju različiti aspekti rada programa.

U ovom projektu korišćena su dva Valgrind alata:

- **Memcheck** – za detekciju grešaka i curenja memorije.
- **Massif** – za praćenje i vizuelizaciju potrošnje memorije tokom izvršavanja.

Njihovi rezultati i analiza predstavljeni su u narednim odeljcima.

2.2 Memcheck

Memcheck je alat za dinamičku analizu memorije koji omogućava detekciju grešaka u radu sa memorijom u C i C++ programima. Koristi se za pronađenje:

- curenja memorije (memory leaks),

- neinicijalizovanih promenljivih,
- nevalidnih pristupa memoriji (segmentation faults),
- pogrešnih oslobođanja memorije.

Memcheck funkcioniše tako što izvršava program u posebnom okruženju koje prati svaku operaciju nad memorijom, beleži potencijalne probleme i prikazuje detaljne izveštaje o njima. Na ovaj način se mogu uočiti problemi koji nisu vidljivi prilikom standardnog testiranja.

2.2.1 Pokretanje alata

Za pokretanje alata kreirana je skripta `run_memcheck.sh`, koja automatski pokreće proces izvršavanja Memcheck-a nad glavnim izvršnim fajlom projekta. Da bismo pokrenuli skriptu, potrebno je da se pozicioniramo u direktorijum `valgrind/memcheck/` i izvršimo sledeće komande:

```
chmod +x run_memcheck.sh
./run_memcheck.sh
```

Komanda kojom pokrećemo Memcheck u skripti je sledeća:

```
valgrind --tool=memcheck \
    --leak-check=summary \
    --track-origins=yes \
    --suppressions="suppressions.sup" \
    --log-file="results.txt" \
    ../../alat-za-analitiku/alatZaAnalitiku/build/
Desktop_Qt_6_9_3-Debug/alatZaAnalitiku
```

Objašnjenje parametara:

- `-tool=memcheck` – pokreće Memcheck alat
- `-leak-check=summary` – prikazuje sažetak curenja memorije
- `-track-origins=yes` – pokušava da prati "poreklo" neinicijalizovanih vrednosti
- `-suppressions=suppressions.sup` – koristi fajl sa izuzecima za poznate, ignorisane greške - greške koje dolaze iz sistemskih, Qt biblioteka i na taj način nam omogućava da pratimo samo naš kod
- `-log-file="results.txt"` – zapisuje izlaz Memcheck-a u fajl
- poslednji argument je putanja do izvršnog fajla aplikacije

Rezultati analize se čuvaju u fajlu `results.txt`, koji se nalazi u istom direktorijumu. Ovaj fajl sadrži detaljan izveštaj o svim detektovanim curenjima memorije, neinicijalizovanim vrednostima i drugim problemima koji su uočeni tokom izvršavanja programa.

2.2.2 Rezultati analize

Analiza je pokazala da projekat Alat za analitiku sadrži određene probleme u radu sa memorijom. Ključni delovi iz izveštaja su sledeći:

- **Use of uninitialised value** – u funkciji `DynamicUI::openPlotWindow()` (linija 74, fajl `dynamicui.cpp`) koristi se neinicijalizovana vrednost, čije poreklo vodi do alokacije u funkciji `FileImportWindow::dropEvent()` (linija 49, `fileimportwindow.cpp`).
- **Curenja memorije:**
 - `definitely lost: 20,016 bytes in 16 blocks` – predstavlja stvarno curenje memorije, gde je memorija alocirana, ali je pokazivač izgubljen i više joj se ne može pristupiti. Ovakvi gubici obično nastaju kada se dinamički alocirani objekti ne oslobođe nakon upotrebe.
 - `indirectly lost: 492,354 bytes in 3,072 blocks` – odnosi se na objekte koji su zavisni od drugih alokacija. Kada glavni pokazivač bude izgubljen, svi pokazivači koji ukazuju na njegove delove postaju indirektno izgubljeni.
 - `possibly lost: 32,947 bytes in 233 blocks` – potencijalni gubici memorije koji se najčešće javljaju kod složenih struktura (npr. pokazivači unutar objekata), gde Valgrind ne može sa sigurnošću da utvrdi pristupnost memoriji.
 - `still reachable: 15,593,825 bytes in 59,018 blocks` – memorija koja nije oslobođena pre završetka programa, ali je i dalje dostupna. U kontekstu Qt aplikacija ovo je uobičajeno ponašanje i ne mora nužno predstavljati grešku, jer Qt često zadržava određene resurse do potpunog gašenja aplikacije.

Statistika potrošnje memorije:

- **Ukupan broj alokacija:** 1,699,269
- **Ukupan broj oslobođenih blokova:** 1,633,089
- **Ukupno alocirano:** 426,406,236 bajtova (\approx 426 MB)

- **Memorija u upotrebi pri izlasku programa:** 16,590,246 bajtova u 66,180 blokova

Na kraju izveštaja zabeleženo je: **ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 1013 from 732)**, što znači da je Memcheck detektovao jednu stvarnu grešku koja potiče iz korisničkog koda, dok su ostale greške detektovane, ali potisnute (“suppressed”) jer potiču iz eksternih biblioteka, pretežno iz samog Qt okvira.

Važno je napomenuti da, iako je Memcheck u ovom izvršavanju uspešno završio analizu, primećeno je da program prilikom ponovljenih pokretanja ponekad "puca" usled pristupa neinicijalizovanoj memoriji (**SIGSEGV**).

2.3 Massif

Massif je alat za profilisanje memorije u C/C++ aplikacijama. Njegova primarna funkcija je praćenje upotrebe heap memorije tokom izvršavanja programa. Massif beleži koliko memorije aplikacija koristi u različitim trenucima i omogućava identifikaciju mesta u kodu gde dolazi do prekomerne alokacije ili curenja memorije.

Alat je posebno koristan za:

- otkrivanje potencijalnih problema sa alokacijom memorije,
- analizu performansi i optimizaciju memorijskih resursa,
- detekciju skrivenih grešaka u upravljanju memorijom, koje se ponekad ne manifestuju pri standardnom izvršavanju programa.

2.3.1 Pokretanje alata

U ovom projektu, Massif se pokreće kroz skriptu **run_massif.sh** koja se nađe u direktorijumu **valgrind/massif/**. Skripta obezbeđuje pravilnu konfiguraciju i parametre i pokreće se komandama:

```
chmod +x run_massif.sh  
./run_massif.sh
```

Komanda koja se koristi u skripti za pokretanje alata:

```
valgrind --tool=massif --stacks=no --time-unit=ms --threshold=5.0 \  
--detailed-freq=200 --massif-out-file=massif.out ./alatZaAnalitiku
```

Parametri znače:

- **-tool=massif** – pokreće Massif modul Valgrind-a
- **-stacks=no** – isključuje praćenje stack memorije
- **-time-unit=ms** – koristi milisekunde kao jedinicu vremena
- **-threshold=5.0** – beleži promene memorije veće od 5%
- **-detailed-freq=200** – detaljna frekvencija prikupljanja podataka
- **-massif-out-file=massif.out.<PID>** – ime izlaznog fajla sa profilom memorije

Rezultat izvršavanja skripte je profil memorije koji nam može otkriti funkcije ili objekte koji najviše troše memoriju. Izlazni fajl se nalazi u direktorijumu **massif_out** i može se pregledati kroz terminal komandnom linijom:

```
ms_print maassif_out/massif.out.<PID>
```

ili grafičkim prikazom pomoću Massif Visualizer alata:

```
massif-visualizer maassif_out/massif.out.<PID>
```

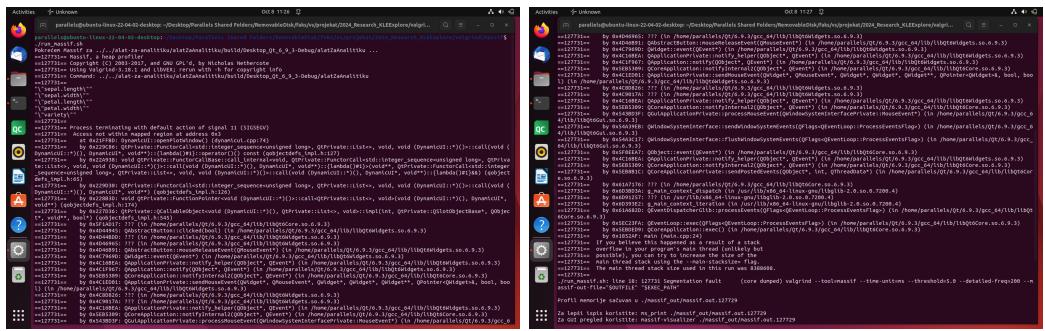
2.3.2 Rezultati analize

Tokom pokretanja aplikacije pod Massif-om, program je prekidal izvršavanje (pucao) sa signalom **SIGSEGV** zbog pokušaja pristupa neinicijalizovanom pokazivaču.

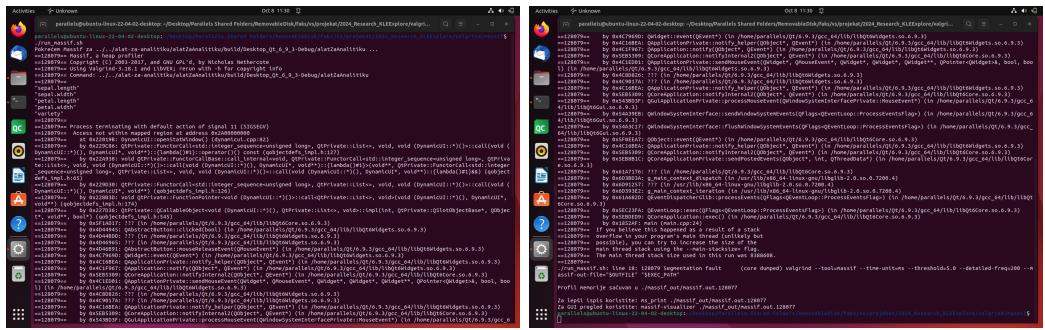
Konkretno, u funkciji **DynamicUI::openPlotWindow()** linija 74:

```
if(p) delete p;
```

pokazivač **p** nije inicijalizovan na **nullptr**, što dovodi do nedefinisanog ponašanja i padu programa. Sličan problem se javlja u funkciji **openStatWindow()** sa pokazivačem **s**.



Slika 1: Pokretanje aplikacije i SIGSEGV greška u funkciji `openPlotWindow()`.



Slika 2: Pokretanje aplikacije i SIGSEGV greška u funkciji `openStatWindow()`.

Zbog toga što program puca pre nego što normalno završi izvršavanje, Massif beleži nepotpun profil memorije. Izlaz alata stoga nije pouzdan za analizu stvarnog zauzeća memorije ili optimizaciju.

Ovaj problem ne mora da se manifestuje pri standardnom izvođenju programa jer:

- raspored memorije u normalnom izvršavanju ponekad slučajno stavlja neinicijalizovane pokazivače na adresu koja ne izaziva segfault,
- brzina izvršavanja i stanje heap memorije u realnom runtime okruženju utiču na to da se greška ne vidi uvek.

Međutim, ovaj alat strogo proverava sve operacije sa memorijom i odmah detektuje pristup neinicijalizovanim ili već obrisanim pokazivačima, zbog čega se crash dosledno javlja prilikom analize.

3 Clang-Format

Clang-Format je alat koji automatski formatira izvorni kod napisan u jezicima C, C++, Java, JavaScript i drugim, prema definisanim pravilima stila. Njegov glavni cilj je da obezbedi konzistentan izgled koda nezavisno od programera, čime se povećava čitljivost, održavanje i preglednost projekata.

Formatiranje se vrši prema pravilima definisanim u konfiguracionom fajlu `.clang-format`, koji može da nasleđuje neki od postojećih stilova (*LLVM, Google, Mozilla, Microsoft*, itd.), a zatim se po potrebi dalje prilagođava.

3.1 Instalacija

Na Linux sistemima alat se može instalirati komandom:

```
sudo apt install clang-format
```

3.2 Pokretanje alata

Pokretanjem komande:

```
clang-format -style=LLVM -dump-config > styleFile
```

generišemo fajl `styleFile` koji sadrži podrazumevana pravila LLVM stila, što nam omogućava pregled i selekciju parametara koje možemo da promenimo. Na sličan način je moguće generisati fajl i za druge stilove.

Za potrebe projekta kreirana je skripta `run_clangFormat.sh` koja automatskuje proces formatiranja svih `.cpp` i `.h` fajlova u okviru projekta. Skripta pretražuje sve relevantne fajlove (isključujući `tests`, `build` i `CMakeFiles` direktorijume) i za svaki od njih primenjuje `clang-format` prema unapred definisanom stilu.

Za formatiranje koda korišćen je stil zasnovan na LLVM šablonu, uz nekoliko prilagođenih parametara definisanih direktno u skripti.

```
clang-format -style="{ \
    BasedOnStyle: LLVM, \
    IndentWidth: 4, \
    TabWidth: 4, \
    AlignConsecutiveAssignments: true, \
    AlignEscapedNewlines: Left, \
    AllowShortBlocksOnASingleLine: true, \
    AllowShortCaseLabelsOnASingleLine: true, \
}
```

```
AllowShortFunctionsOnASingleLine: Inline,
AllowShortLambdasOnASingleLine: All, \
BreakBeforeBraces: Attach, \
MaxEmptyLinesToKeep: 2, \
ColumnLimit: 150}"
```

Ovi parametri nam omogućavaju:

- `IndentWidth: 4` i `TabWidth: 4` – uvlačenje od četiri razmaka radi bolje čitljivosti koda,
- `AlignConsecutiveAssignments: true` – poravnavanje uzastopne dodele radi preglednosti,
- `AllowShortFunctionsOnASingleLine: Inline` i `AllowShortLambdasOnASingleLine: All` – dozvoljavanje kraćih funkcija i lambda izraza u jednom redu,
- `BreakBeforeBraces: Attach` – otvarajuća vitičasta zagrada se nalazi u istom redu kao i naredba,
- `ColumnLimit: 150` – povećavanje maksimalne dužine reda radi smanjenja nepotrebnih preloma u kompleksnim izrazima.

Ovakav pristup je bio neophodan jer u okviru projekta već postoji `.clang-format` fajl zasnovan na *Microsoft* stilu. Da bi se izbeglo njegovo automatsko nasleđivanje, željeni stil je definisan direktno unutar skripte (`inline`), čime je osigurano da se uvek koristi *LLVM*-bazirani stil sa prilagođenim parametrima.

Pokretanje skripte se vrši sledećim komandama iz direktorijuma `clang-format`:

```
chmod +x run_clangFormat.sh
./run_clangFormat.sh
```

3.3 Analiza rezultata

Skripta kreira direktorijum `formattedFiles` u koji se smeštaju svi fajlovi čiji format nije bio u skladu sa zadatim pravilima.

```

plotwindow.cpp (Original)
plotwindow.cpp (Formatted)

```

The image shows a code editor with two tabs open. The left tab is labeled 'plotwindow.cpp' and contains the original unformatted code. The right tab is also labeled 'plotwindow.cpp' and contains the same code but with horizontal lines and spaces added to make it more readable. Both tabs show the same code, which includes functions like 'setupPlot()', 'setupPiePlot()', 'setupBarPlot()', and 'setupHorizontalBarPlot()'. The code is primarily in C++ with some C-style comments and includes several header files like 'QChart', 'QLineSeries', and 'QColor'.

Slika 3: Originalni kod (levo) i formatirani kod (desno)

4 Cppcheck

Nakon dinamičke analize korišćenjem alata Valgrind i provere stila pomoću Clang-Format-a, sprovedena je i statička analiza koda pomoću alata Cppcheck, sa ciljem otkrivanja logičkih i potencijalno kritičnih grešaka u projektu.

Cppcheck je statički analizator koda za programske jezike C i C++. Njegova osnovna funkcija je da analizira izvorni kod bez njegovog izvršavanja i identificuje potencijalne greške i rizične konstrukcije. Alat može detektovati probleme kao što su logičke greške, curenje memorije, neoptimalni izrazi i druge potencijalne izvore bug-ova.

Za razliku od kompjajlera, koji prijavljuje samo sintaksne greške i konstrukcije koje sprečavaju kompilaciju, Cppcheck upozorava na probleme koji možda mogu dovesti do neočekivanog ponašanja programa tokom izvršavanja. Korišćenjem Cppcheck-a, programeri mogu unaprediti kvalitet, pouzdanost i sigurnost svog koda, smanjujući rizik od kritičnih grešaka u produkcionom okruženju.

4.1 Instalacija

Na Linux sistemima, Cppcheck se može instalirati komandom:

```
sudo apt install cppcheck
```

4.2 Pokretanje alata

U ovom projektu, Cppcheck je pokrenut kroz skriptu `run_cppcheck.sh` koja se nalazi u direktorijumu `cppcheck`. Skripta automatski generiše izveštaje u tekstualnom i HTML formatu.

Pokretanje iz odgovarajućeg direktorijuma:

```
chmod +x run_cppcheck.sh  
./run_cppcheck.sh
```

Skripta uključuje sledeće korake:

- Definisanje putanje do projekta i direktorijuma za izveštaje.
- Pokretanje `cppcheck` sa parametrima:
 - `-enable=all` — omogućava sve vrste provera,
 - `-inconclusive` — uključuje i moguće neodređene greške,
 - `-std=c++17` — koristi C++17 standard,
 - `-I` — dodaje include direktorijume (Qt biblioteke),
 - `-D` — definiše makro (QT_CORE_LIB),
 - `-suppress=missingIncludeSystem` — ignoriše greške vezane za sistemske include fajlove.
- Generisanje XML fajla za kasniju konverziju u HTML izveštaj.
- Kreiranje HTML izveštaja pomoću `cppcheck-htmlreport`.

4.3 Analiza rezultata

Svi rezultati analize se nalaze u direktorijumu `cppcheck-report`. Mogu se pregledati u dva formata:

- **Tekstualni izveštaj:** `cppcheck-output.txt` sadrži listu svih detektovanih problema i njihov opis.
- **HTML izveštaj:** omogućava interaktivni pregled sa laksim filtriranjem i smernicama po fajlovima.

4.3.1 Pregled izveštaja `stats.html`

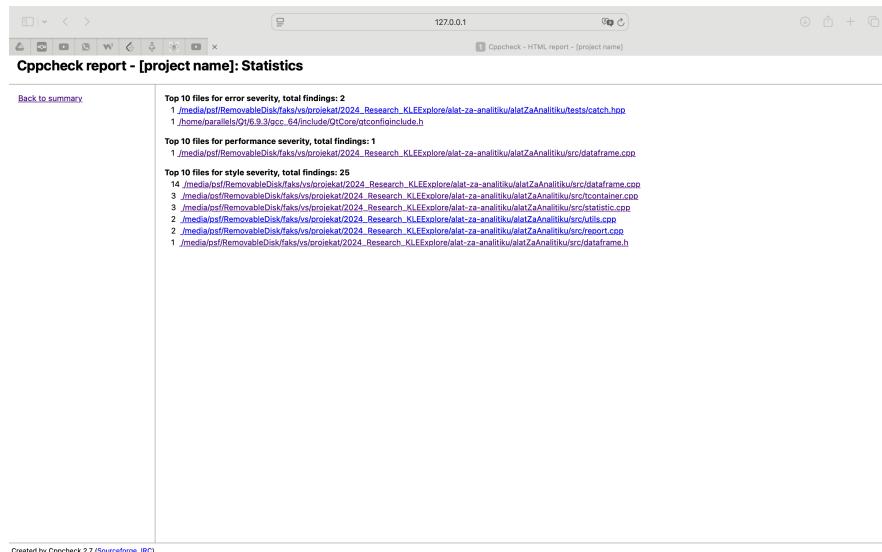
`stats.html` je deo HTML izveštaja koji Cppcheck automatski generiše i predstavlja sumarni pregled rezultata analize (statistički prikaz otkrivenih

problema).

U ovom izveštaju su prikazane sledeće informacije:

- broj grešaka po kategorijama (npr. *error*, *warning*, *style*, *performance*, *portability*),
- distribucija grešaka po fajlovima.

Ove informacije omogućavaju da se dobije kvantitativni uvid u stanje projekta i da se identifikuju delovi koda koji zahtevaju najviše pažnje. Na osnovu prikazanih statistika moguće je proceniti kvalitet koda, kao i napredak u njegovom unapređenju tokom razvoja.



Slika 4: Prikaz pregleda grešaka u izveštaju `stats.html`

4.3.2 Pregled izveštaja `index.html`

`index.html` predstavlja glavni deo HTML izveštaja koji generiše Cppcheck i sadrži detaljan prikaz svih pronađenih problema u kodu. Svaka detektovana greška, upozorenje ili preporuka za poboljšanje prikazana je zajedno sa informacijama o fajlu, liniji koda i opisom problema.

Izveštaj omogućava jednostavno "kretanje" kroz projekat, jer su rezultati sortirani po fajlovima, a za svaku stavku je moguće direktno videti relevantan deo koda u kojem je problem otkriven. Na ovaj način programer može brzo da locira i razume kontekst potencijalne greške.

`index.html` prikazuje sledeće informacije:

- listu svih analiziranih fajlova,
- tip i opis svake detektovane greške,
- ozbiljnost problema (*error, warning, style*),
- tačnu liniju koda gde je greška pronađena,
- preporuku za ispravku.

Ovaj izveštaj omogućava pregledan i interaktivn način analize rezultata, gde se problemi mogu lako filtrirati i pretraživati po kategorijama. Na taj način `index.html` olakšava proces analize koda i ubrzava identifikaciju najkritičnijih delova projekta.

The screenshot shows the Cppcheck report interface with the following details:

- Defect summary:**
 - missinginclude: 14 unusedFunction, 6 constParameter, 4 useStlAlgorithm, 3 functionConst, 2 preprocessErrorDirective, 1 missingInclude, 1 noExplicitConstructor, 1 useInitializationList (32 total).
- Statistics:**
 - 14 unusedFunction, 6 constParameter, 4 useStlAlgorithm, 3 functionConst, 2 preprocessErrorDirective, 1 missingInclude, 1 noExplicitConstructor, 1 useInitializationList (32 total).
- Message table:**

Line	ID	CWE	Severity	Message
10	561		style	The function 'TYPEToString' is never used.
24	561		style	The function 'isDateType' is never used.
50	398		performance	Variable '_vars' is assigned in constructor body. Consider performing initialization in initialization list.
81	561		style	The function 'unique' is never used.
149	561		style	Unused function 'unique' is never used.
185	398		style	Parameter 'column' can be declared with const
186	398		style	Parameter 'item' can be declared with const
188	561		style	The function 'count' is never used.
172	398		style	Consider using std::count_if algorithm instead of a raw loop.
178	398		style	Parameter 'col' can be declared with const
183	398		style	Parameter 'col' can be declared with const
188	398		style	Parameter 'column' can be declared with const
188	398		style	Parameter 'item' can be declared with const
188	561		style	The function 'filter' is never used.
200	398		style	Consider using std::transform algorithm instead of a raw loop.
20	398		style	Class 'DataFrame' has a constructor with 1 argument that is not explicit.
21	398		style, iconcl.	Technically the member function 'VarVector<type>' can be const.
22	398		style, iconcl.	Technically the member function 'VarVector<type>' can be const.
38	398		style	Class 'DataFrame' has a constructor with 1 argument that is not explicit.
12	561		style	The function 'setTargetWidget' is never used.
34	561		style	The function 'saveToPif' is never used.
106	398		style	Consider using std::accumulate algorithm instead of a raw loop.
111	561		style	The function 'value_counts' is never used.
129	398		style	Consider using std::count_if algorithm instead of a raw loop.
51	561		style	The function 'focusInEvent' is never used.

Slika 5: Prikaz pregleda grešaka u izveštaju `index.html`

```

Defects: dataframe.h
functionConst_20
functionConst_21
functionConst_22
noExplicitConstructor_38

3 #include <QString>
4 #include <QVector>
5 #include <unordered_map>
6
7 enum TYPE
8 {
9     NUMERIC,
10    STRING
11 };
12
13
14 class VarVector
15 {
16 public:
17     VarVector();
18     VarVector(TYPE t, QVector<QString> vars);
19
20     QString operator[](int x); ===== Technically the member function 'VarVector::operator[]' can be const. [=]
21     TYPE type(); ===== Technically the member function 'VarVector::type' can be const. [=]
22     QVector<QString> vars(); ===== Technically the member function 'VarVector::vars' can be const. [=]
23
24     void push_back(QString s);
25     void set_type(TYPE t);
26     std::unordered_map<QString, VarVector> _df;
27
28 private:
29     TYPE _t;
30     QVector<QString> _vars;
31 };
32
33
34 class DataFrame
35 {
36 public:
37     std::unordered_map<QString, VarVector> _df;
38
39     DataFrame();
40
41     DataFrame(const QString &path); ===== Class 'DataFrame' has a constructor with 1 argument that is not explicit. [=]
42
43     VarVector operator[](const QString &header) const; ===== Class 'DataFrame' has a constructor with 1 argument that is not explicit. Such constructors should in general be explicit for type safety reasons. The explicit keyword in the constructor means some mistakes when using the class can be avoided.
44     void push_back(QString &col, const TYPE colType); ===== Class 'DataFrame' has a constructor with 1 argument that is not explicit. Such constructors should in general be explicit for type safety reasons. The explicit keyword in the constructor means some mistakes when using the class can be avoided.
45     void setColumnType(QString &col, TYPE colType);
46     int count(QString column, QString header);

```

Created by Cppcheck 2.7 (Sourceforce, RC)

Slika 6: Detaljan prikaz grešaka i predlog načina popravke u `dataframe.h` fajlu

Kombinovanjem `index.html` i `stats.html`, programeri dobijaju potpun pregled kvaliteta koda i mogu efikasno pratiti napredak u otklanjanju grešaka.

Na osnovu rezultata analize može se zaključiti da alat Cppcheck predstavlja efikasno rešenje za rano otkrivanje potencijalnih problema u kodu.

5 Zaključak

Primena različitih alata za verifikaciju i analizu koda omogućila je sveobuhvatan uvid u kvalitet i (ne)pouzdanost projekta Alat za analitiku. Korišćenjem dinamičkih alata poput Valgrind-a detektovani su problemi u radu sa memorijom, uključujući curenje i neinicijalizovane pokazivače. Alat Clang-Format obezbedio je konzistentnost u stilu i formatiranju koda, što doprinosi većoj čitljivosti i održivosti projekta. Na kraju, statička analiza korišćenjem Cppcheck-a omogućila je prepoznavanje logičkih grešaka, potencijalno rizičnih konstrukcija i problema u strukturi koda, čak i pre faze kompajliranja.

Kombinovanjem ovih pristupa ostvarena je dublja verifikacija softverskog sistema — od dinamičkog ponašanja i efikasnosti memorije, preko standardizacije stila, do statičke provere logike koda. Ovaj proces ne samo da doprinosi većem kvalitetu i stabilnosti aplikacije, već postavlja i osnovu za dalje unapređenje procesa razvoja i testiranja softvera kroz automatizaciju i kontinuiranu integraciju.