Izveštaj primene alata za verifikaciju u okviru samostalnog praktičnog projekta na kursu Verifikacija Softvera

Matematički fakultet

Nikola Mićić, 1086/2022
nikolamicic065@gmail.com
Mentor: Ivan Ristović

Decembar 2022.

Sažetak

Ovaj rad će sadržati detaljan opis analize projekta sa spiskom naredbi i alata koji su korišćene i zaključcima koji su napravljeni.

Projekat nad kojim će biti primenjeni alati se nalazi na github adresi: https://github.com/eminfedar/widgetci

Autor ovog projekta je Emin Fedar (github username: eminfedar).

Primena alata će biti izvršena na master grani, nad komitom čiji je hash code sledeći: 05588c036cd4c6d6eddd5b595d027968ac484140

Sadržaj

1 Spisak primenjenih alata

Spisak alata za verifikaciju koji su primenjeni nad projektom su:

1. Clang-tidy i Clazy

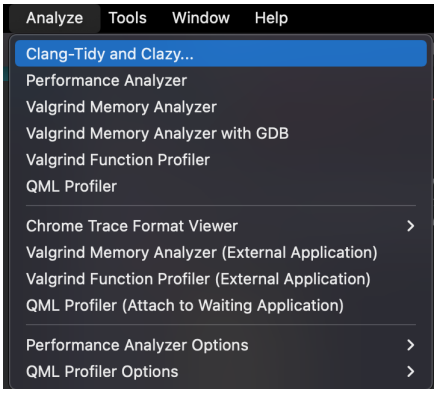
1.1 Clang-tidy i Clazy

Clang-tidy pruža dijagnostiku i ispravke za tipične programske greške, kao što su kršenje stila ili zloupotreba interfejsa.

Clazy pomaže Clang-u da razume Qt semantiku. Prikazuje upozorenja kompajlera vezana za Qt, u rasponu od nepotrebne alokacije memorije do zloupotrebe API-ja i pruža akcije refaktorisanja za rešavanje nekih problema.

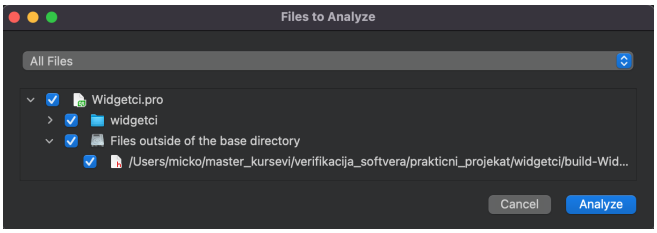
Clang-tidy i Clazy su alati koji su pokrenuti preko QtCreator-a. Alat je pokrenut preko Debug moda i sastoji se iz narednih koraka koji se mogu videti na slikama ispod.

Prvi korak je biranje alata Clang-Tidy i Clazy u okviru padajućeg menija Analyze.



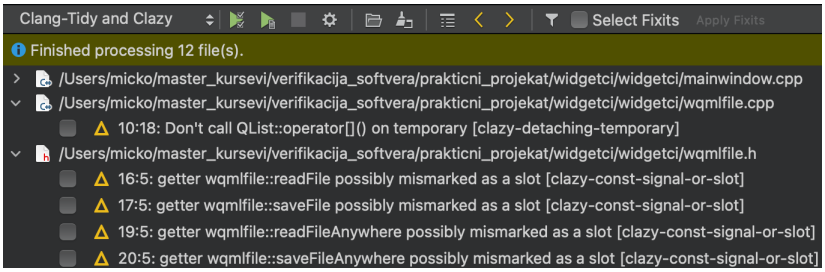
Slika 1: Prvi korak

Drugi korak se sastoji iz biranja fajlova koji će biti analizirani. U ovom slučaju sam birao sve fajlove uključujući glavni folder *code* u okviru kojeg se nalaze podfolderi *headers* i *src*. Nakon toga kliknuti dugme *Analyze*.



Slika 2: Drugi korak

Deo rezultata ovih alata je predstavljen na slici 3 ispod. Može se videti upozorenje da postoji neinicijalizovano polje na kraju poziva konstruktora, sa objašnjenjima. Ceo izlaz pozvanih alata se može naći u okviru repozitorijuma za analizu datog projekta, u okviru foldera Clang-tidy i Clazy.



Slika 3: Deo rezultata pozvanih alata

1.2 Cppcheck

Cppcheck je alat za statičku analizu za C/C++ kod. Pruža jedinstvenu analizu koda za otkrivanje grešaka i fokusira se na otkrivanje nedefinisanog ponašanja i opasnih konstrukcija kodiranja. Cilj je imati vrlo malo lažnih pozitivnih rezultata. Cppcheck je dizajniran da može da analizira vaš C/C++ kod čak i ako ima nestandardnu sintaksu (uobičajeno u ugrađenim projektima).

Cppcheck je alat koji sam pokretao preko terminala. Za instalaciju alata na macOS-u sam koristio komandu *brew install cppcheck*.

Alat proverava sve .cpp fajlove na prosleđenoj putanji i vraća uočene greške i upozorenja.

Pozivanje komande i deo rezultata je izgledao kao na slici ispod.

Greška koja je otkrivena na slici 4 je pronađena u fajlu widgetci/wqmlsystem.cpp na liniji 78 i predstavlja pronađenu funkciju koja je non-void tipa, a ipak ne sadrži povratnu vrednost.

Celokupan izlaz pozivanja alata se može naći u okviru repozitorijuma za analizu datog projekta, u okviru foldera Cppcheck.

Slika 4: Način pokretanja i rezultati alata cppcheck

1.3 Memcheck

Valgrind Memcheck je alat za otkrivanje problema sa korišćenjem memorije kao što je curenje memorije, nevažeći pristup memoriji, nepravilno oslobađanje memorije ili referenciranje nedefinisanih vrednosti.

Koristi se za detekciju navedenih problema koji su karakteristični za programe pisane u C i C++ programskim jezicima.

Da bismo pokrenuli valgrind memcheck alat potrebno je kreirati izvršni fajl. Izvšrni fajl će biti kreiran kroz build fazu projekta u QtCreator-u, nakon čega je potrebno pozicionirati se u okviru build foldera gde se nalazi izvršni fajl.

Nakon pozicioniranja vrši se pozivanje sledeće komande:

valgrind –tool=memcheck ./Widgetci

Preko opcije **–tool=memcheck** se bira alat koji se pokreće (memcheck je podrazumevana vrednost), a **./Widgetci** je izvršni fajl.

Dodatne opcije koje su korišćene su:

–track-origins=yes

–show-leak-kinds=all

–leak-check=full

–log-file=../../../../memcheck/valgrind_memcheck_output.txt

Rezultujući fajl **valgrind_memcheck_output.txt** se nalazi u okviru projekta u folderu memcheck.

```
==17195== HEAP SUMMARY:
==17195==     in use at exit: 0 bytes in 0 blocks
==17195==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==17195==
==17195== All heap blocks were freed -- no leaks are possible
==17195==
==17195== For lists of detected and suppressed errors, rerun with: -s
==17195== ERROR SUMMARY: 170706 errors from 313 contexts (suppressed: 0 from 0)
```

Slika 5: Deo rezultata memcheck alata - heap summary

Na osnovu dela izlaza na slici 5 možemo videti da nema curenja memorije za heap blokove memorije.

U izlaznom fajlu se nalazi veliki broj nevalidnih čitanja koji su zajednički za macOS biblioteku dyld, čiji primer se može videti na slici 6.

```
==17195== Invalid read of size 1
==17195==    at 0x100109B89: __chkstk_darwin_probe (in /usr/lib/dyld)
==17195==    by 0x1000B5BE6: dyld4::prepare(dyld4::APIs&, dyld3::MachOAnalyzer const*) (in /usr/lib/dyld)
==17195==    by 0x1000B54E3: (below main) (in /usr/lib/dyld)
==17195==  Address 0x104964d10 is on thread 1's stack
==17195==  6320 bytes below stack pointer
```

Slika 6: Deo rezultata memcheck alata - invalid read

Pored toga može se videti i problem sa **Conditional jump or move depends on uninitialised value(s)** koja ukazuje na korišćenje neinicijalizovanih vrednosti, primer se može videti na slici 7.

```
==17195== Conditional jump or move depends on uninitialised value(s)
==17195==    at 0x7FF816CADD07: ??? (in /dev/ttys000)
==17195==    by 0x7FF816CE496E: ??? (in /dev/ttys000)
==17195==    by 0x7FF816CE42D5: ??? (in /dev/ttys000)
==17195==    by 0x7FF816CF0A7C: ??? (in /dev/ttys000)
==17195==    by 0x7FF816CF09CA: ??? (in /dev/ttys000)
==17195==    by 0x7FF816CF0222: ??? (in /dev/ttys000)
==17195==    by 0x7FF816DFDE88: ??? (in /dev/ttys000)
==17195==    by 0x7FF816D4B23E: ??? (in /dev/ttys000)
==17195==    by 0x7FF816DBD3D5: ??? (in /dev/ttys000)
==17195==    by 0x7FF816AE3316: ??? (in /dev/ttys000)
==17195==    by 0x7FF816AE44F9: ??? (in /dev/ttys000)
==17195==    by 0x7FF816DBD3AB: ??? (in /dev/ttys000)
==17195==  Uninitialised value was created by a stack allocation
==17195==    at 0x7FF816CF0745: ??? (in /dev/ttys000)
```

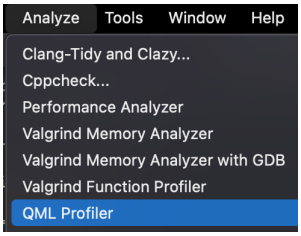
Slika 7: Deo rezultata memcheck alata - Conditional jump

1.4 QML Profiler

QML Profiler je alat koji omogućava da se dobiju neophodne dijagnostičke informacije, omogućujuću da se analizira kod aplikacije kako bi se otkrili problemi sa performansama. Primer problema sa performansama predstavljaju previše JavaScript-a u određenim frejmovima, C++ funkcija koje traju predugo ili troše previše memorije. QML Profiler predstalvja deo i Qt Creator-a i Qt Design Studio-a.

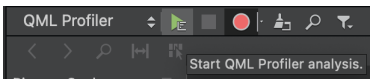
Kroz korake će biti predstavljen način rada ovog alata i rezultati prikazani.

Pokretanje alata preko Qt Creator-a se vrši preko padajućeg menija Analyze i biranje alata QML Profiler, kao na slici 8 ispod.



Slika 8: Biranje alata QML Profiler-a

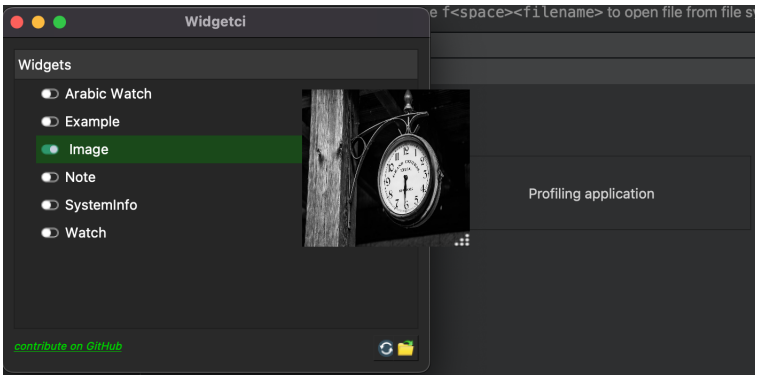
Zatim se na zeleno dugme pokreće alat, kao što se može videti na slici 9 ispod.



Slika 9: Pokretanje QML Profiler-a

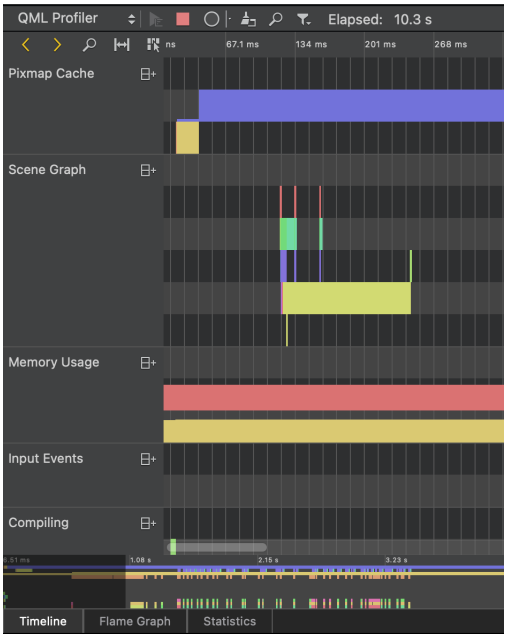
Analiza QML Profiler-a je odrađena pri pokretanju widget-a aplikacije koja predstavlja prikazivanje slike sata koja može da se povećava i smanjuje pomeranjem donje desne ivice kao što se može videti na slici 10 ispod.

Rezultati QML Profiler-a predstavljaju analizu nakon 10 sekundi prikazivanja sata i povećavanja i smanjivanja njegovih dimenzija pomeranjem donje desne ivice.

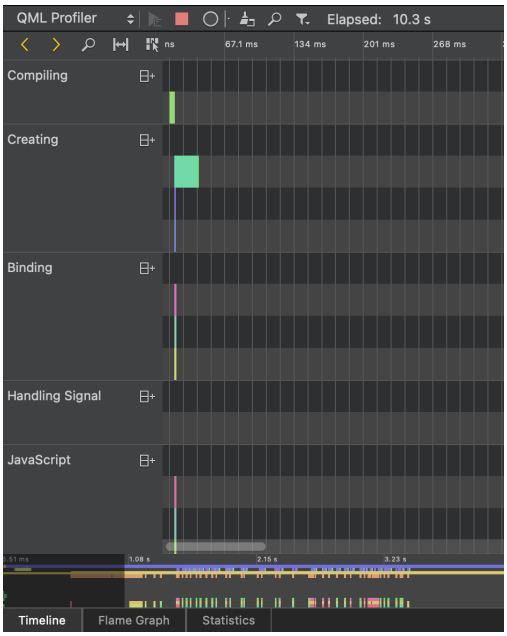


Slika 10: Interakcija sa aplikacijom tokom analiziranja QML Profiler-om

Timeline QML Profiler-a izdeljen po sekcijama u prvoj sekundi izvršavanja izgleda kao na slikama 11,12 (duže od jedne sekunde nije moglo biti prikazano na screenshot-u)

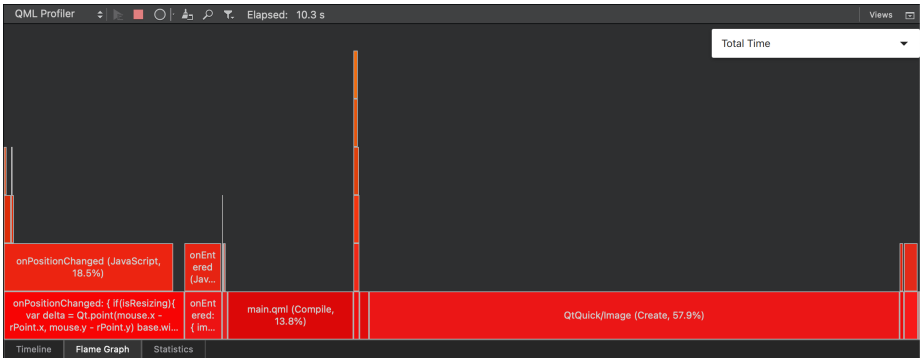


Slika 11: Prvi deo Timeline-a QML Profiler-a



Slika 12: Drugi deo Timeline-a QML Profiler-a

Rezultat analize prikazan kroz Flame Graph, za sekciju Total Time izgleda ovako 13.



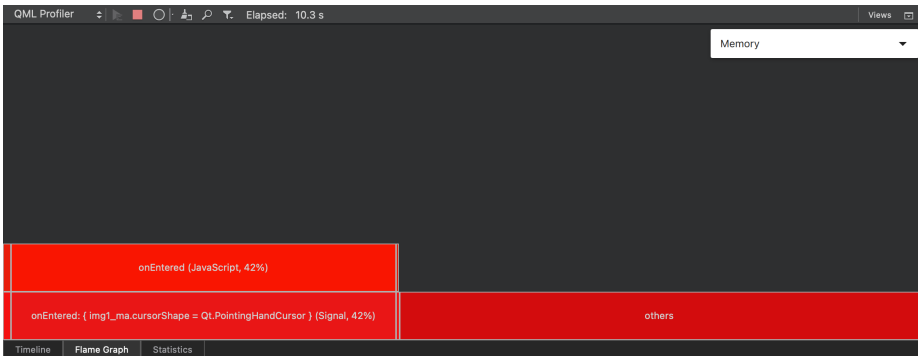
Slika 13: Flame Graph - Total Time

Rezultat analize prikazan kroz Flame Graph, za sekciju memorijskog utroška izgleda ovako 14.

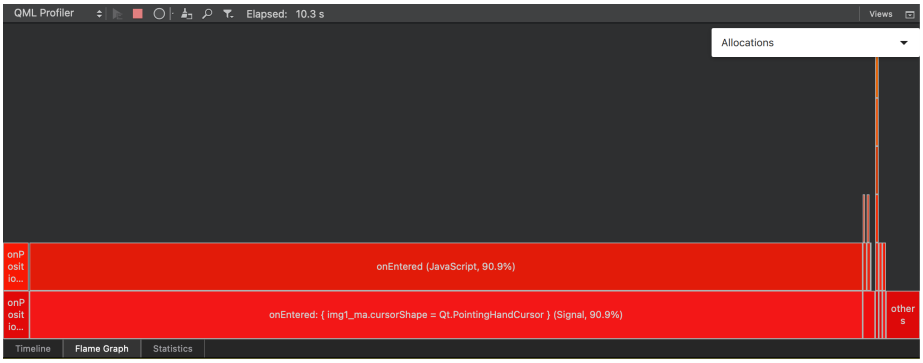
Rezultat analize prikazan kroz Flame Graph, za sekciju alokacija memorije izgleda ovako 15.

Na kraju se može videti statistički prikaz analize nad svim komponentama ove aplikacije 16.

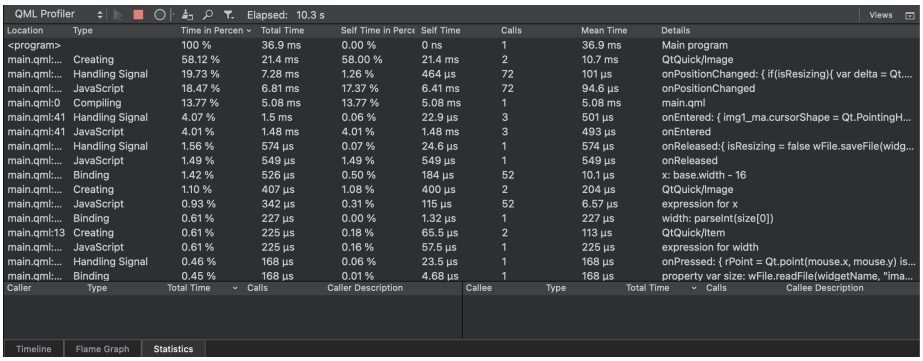
Na osnovu ovih dijagrama možemo videti performanse aplikacije tokom izvršavanja i zahvaljujući njima odrediti koji delovi koda troše najviše vremena i memorije i na osnovu toga ih optimizovati.



Slika 14: Flame Graph - Memory



Slika 15: Flame Graph - Allocations



Slika 16: Statistics

U okviru rezultata ovog alata nad analiziranim projektom nisam primetio problem sa performansama tokom izvršavanja aplikacije.