Analiza projekta korišćenjem alata za verifikaciju softvera

Samostalni seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Dunja Čitlučanin, 1024/2022
dunja.citlucanin1@gmail.com

30. avgust 2023.

Sažetak

U ovom radu biće prikazana analiza projekta `TankAttack`, dobijena primenom alata za verifikaciju softvera, i ukratko će biti opisani alati i naredbe koji su korišćeni u tu svrhu. Analizirani projekat nalazi se na sledećoj adresi:https://gitlab.com/matf-bg-ac-rs/course-rs/projects-2020-2021/17-tankattack, a autori su Nikola Mićić, Luka Miletić, Nikola Lazarević, Slobodan Jovanović i Mihailo Trišović. Projekat je nastao u okviru kursa Razvoj softvera.

Sadržaj

1 Uvod

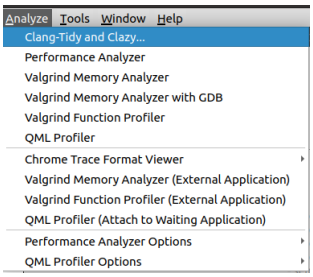
`TankAttack` je igrica,namenjena za dva igrača, koja simulira bitku između tenkova koji se nalaze u lavirintu. Cilj je eliminsati protivnika-tenkovi inicijalno ispaljuju tenkovsko đule, a tokom bitke moguće je i skupljanje supermoćnog oružja koje olakšava eliminaciju protivnika. Takođe, tokom igrice, moguće je i skupljanje srca koja omogućavaju punjenje helti na maksimum. Tenk koji prvi osvoji 3 pobede je pobednik. Za testiranje projekta korišćeni su alati: `Clang-Tidy i Clazy, Cppcheck, Clangd i Callgrind`.

2 Clang-Tidy i Clazy

Prvi alat koji ćemo primeniti je alat za statičku analizu. U pitanju je `Clang-Tidy`. On se koristi za detekciju grešaka i upotorenja, kao i stilskih problema u `C++` kodu. Veoma je koristan zato što omogućava otkrivanje potencijalnih problema u kodu pre kompilacije i njihovo ispravljanje u fazi razvoja.

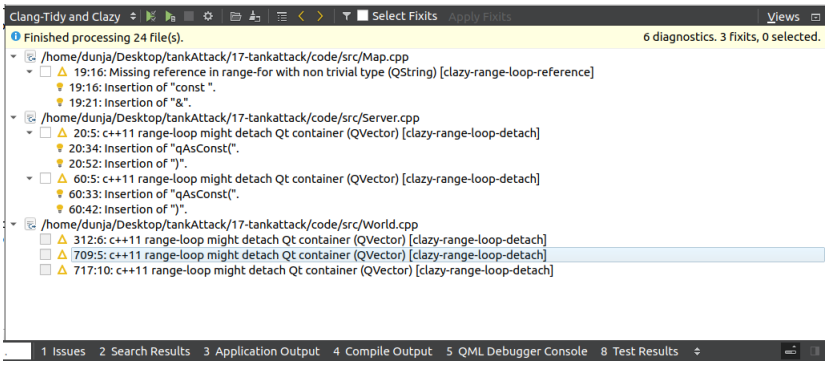
Dodatak ovom alatu je alat `Clazy`, koji je razvijen posebno za Qt aplikacije. Ovaj alat može da detektuje potencijalno curenje memorije, nepravilne konverzije tipova podataka, neefikasne upotrebe Qt API-ja, često se koristi u Qt projektima i omogućava otkrivanje problema specifičnih za ovu tehnologiju. Kada se koriste zajedno, `Clang-Tidy` i `Clazy`, pružaju dublju analizu Qt koda, poboljšanje kvaliteta i bezbednosti.

Pokazaćemo upotrebu ovih alata pomoću QtCreator-a. Potrebno je da klikom na *Analyze* izaberemo opciju *Clang-Tidy and Clazy*, a potom treba da izaberemo fajlove na koje želimo da primenimo analizu i pokrenemo je klikom na dugme *Analyze*.



Slika 1: Clang

Rezultat ovih alata prikazan je na slici 2.

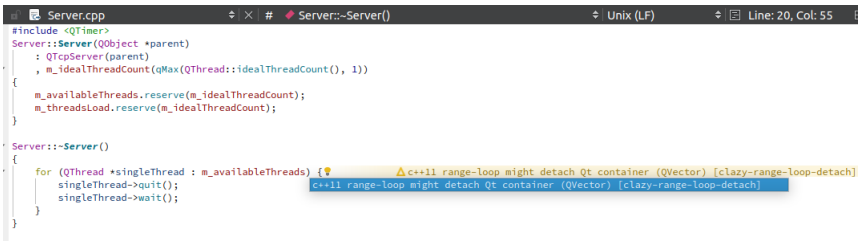


Slika 2: Rezultat primene alata Clang-Tidy i Clazy

Prva poruka, odnosno poruka prikazana na slici 3, koja se odnosi na fajl `Map.cpp`, ukazuje na to da nije korišćena referenca u `for` petlji, što znači da će se kopirati elementi iz opsega. Takođe, poruka ukazuje na problem zbog toga što se for petlja koristi za prolazak kroz ne-trivijalni tip podataka koji obično imaju složene operacije prilikom kopiranja (konstruktore, destruktore...), u ovom slučaju QString. Ovu poruku možemo rešiti tako što dodamo referencu u for pelji čime bismo izbegli kopiranje elemenata.

Slika 3: Poruka koja se odnosi na fajl Map.cpp

Sledeća poruka koju ćemo komenarisati, pojavljuje u analizi fajlova `Server.cpp` i `World.cpp`, prikazana je na slici 4. Ovo upozorenje ukazuje na potencijalni problem prilikom korišćenja `for` petlje sa opsegom za iteriranje kroz Qt kontejnere, kao što je QVector. Postoji opasnost od odvajanja kontejnera, tj. situacije u kojoj se napravi kopija kontejnera kako bi se izvršila iteracija, što može da dovede do promena u originalnom kontejneru ako se on menja u telu petlje. Jedan od načina da se reši ovaj problem je upotrebom const reference u `for` petlji.



Slika 4: Poruka koja se pojavljuje za fajl Server.cpp (kao i za World.cpp

3 Cppcheck

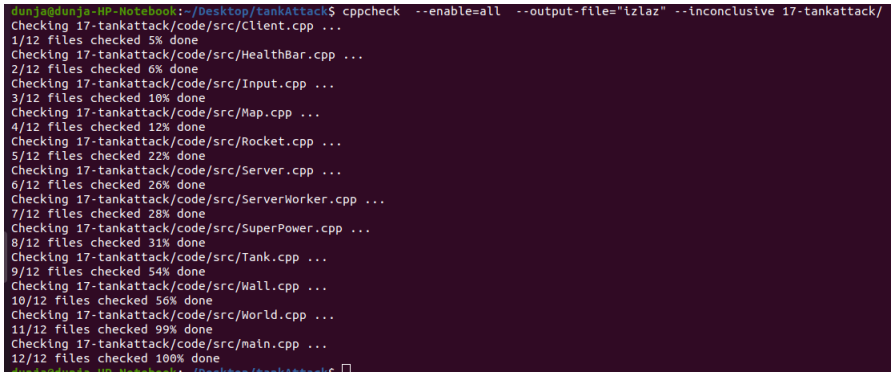
Analiziraćemo ovaj projekat još jednim moćnim alatom za statičku analizu `C` i `C++` koda. Sada je reč o alatu `Cppcheck`. I ovaj alat pomaže u otkrivanju potencijalnih grešaka, upozorenja i stilskih problema pre kompilacije i izvršavanja. Vrši veliki broj provera kao što su detektovanje neiskorišćenih funkcija, neinicijalizovanih promenljivih, loše alokacije memorije...Upotrebom ovog alata može se značajno poboljšati kvalitet koda i povećati pouzdanost softvera.

Za instalaciju ovog alata potrebno je pokrenuti sledeću komandu u terminalu:

```
sudo apt-get install cppcheck
```

Pokrenućemo ovaj alat i uključiti neke dodatne opcije koje će doprineti našoj analizi.Prva opcija, '–enable=all', će uključiti sve dostupne provere, tj. alat će proveriti sve moguće vrste grešaka i upozorenja. Sa '–inonclusive' uključujemo i provere koje se svrstavaju u "neodlučne", tj. one provere koje cppcheck nije mogao da potvrdi kao greške ili upozorenja ali ih ipak prijavljuje. Sa '–output-file' preusmeravamo izlaz u dati fajl.

```
cppcheck  --enable=all  --output-file="izlaz" --inconclusive 17-tankattack/
```



Slika 5: Pokretanje alata cppcheck

Alat možemo pokrenuti i pomoću skripte cppcheck_script.sh koja se nalazi u okviru foldera `Cppcheck`. Ceo izlaz ovog alata može se naći u

okviru foldera `Cppcheck`. Alat daje napomenu o velikom broju funkcija koje se nikada ne koriste, što može ukazivati na nepotreban kod i time narušavanje čitljivosti koda.

```
Checking 17-tankattack/code/src/main.cpp ...
12/12 files checked 100% done
17-tankattack/code/src/Tank.cpp:487:0: style: The function 'GetX' is never used. [unusedFunction]

^
17-tankattack/code/src/Tank.cpp:491:0: style: The function 'GetY' is never used. [unusedFunction]

^
17-tankattack/code/src/Wall.cpp:62:0: style: The function 'getCoordinates' is never used. [unusedFunction]

^
17-tankattack/code/src/Map.cpp:37:0: style: The function 'getNumOfWalls' is never used. [unusedFunction]

^
17-tankattack/code/src/SuperPower.cpp:20:0: style: The function 'getSize' is never used. [unusedFunction]

^
17-tankattack/code/src/Client.cpp:154:0: style: The function 'getTankX' is never used. [unusedFunction]

^
17-tankattack/code/src/Client.cpp:158:0: style: The function 'getTankY' is never used. [unusedFunction]

^
17-tankattack/code/src/Tank.cpp:479:0: style: The function 'getXposition' is never used. [unusedFunction]

^
17-tankattack/code/src/Tank.cpp:483:0: style: The function 'getYposition' is never used. [unusedFunction]

^
17-tankattack/code/src/Tank.cpp:541:0: style: The function 'get_score' is never used. [unusedFunction]
```

Slika 6: Deo izlaza alata cppcheck (neiskorišćene funkcije)

Zatim, pojavljuju se i različiti nazivi argumenata u deklaracijama i definicijama funkcija, što ne dovodi do greške u kompilaciji, ali takođe, narušava čitljivost i održivost koda. Još jedna infomacija koju alat daje je i postojanje promenljivih koje nisu inicijalizovane. Takođe, pojavljuje se i stilski problem hvatanja izuzetaka po vrednosti, što bi trebalo raditi po referenci ne bismo kopirali izuzetke.

```
Checking 17-tankattack/code/src/Client.cpp ...
17-tankattack/code/include/Client.hpp:10:1: style: The class 'Client' does not have a constructor although it has private member variables. [n
oConstructor]
class Client : public QObject
^
17-tankattack/code/src/Client.cpp:71:18: style: Condition 'text=="Space"' is always true [knownConditionTrueFalse]
    else if(text == "Space")
                 ^
17-tankattack/code/src/Client.cpp:85:46: style:inconclusive: Function 'jsonReceived' argument 1 names different: declaration 'doc' definition
'docObj'. [funcArgNamesDifferent]
void Client::jsonReceived(const QJsonObject &docObj)
                                             ^
17-tankattack/code/include/Client.hpp:47:42: note: Function 'jsonReceived' argument 1 names different: declaration 'doc' definition 'docObj'.
    void jsonReceived(const QJsonObject &doc);
                                        ^
17-tankattack/code/src/Client.cpp:85:46: note: Function 'jsonReceived' argument 1 names different: declaration 'doc' definition 'docObj'.
void Client::jsonReceived(const QJsonObject &docObj)
                                             ^
17-tankattack/code/src/Client.cpp:145:30: style:inconclusive: Function 'setTanksX' argument 1 names different: declaration 'pozicija_tenka_x'
definition 'TankX'. [funcArgNamesDifferent]
void Client::setTanksX(float TankX)
```

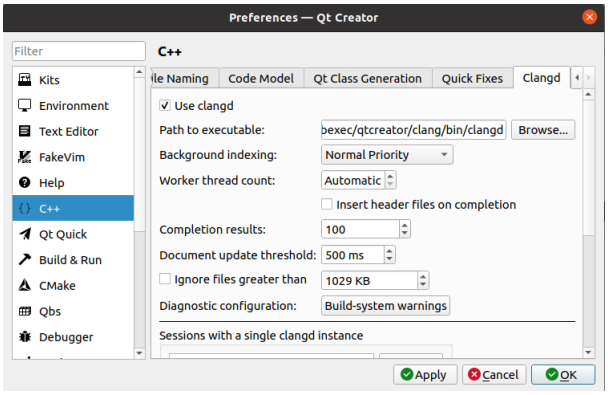
Slika 7: Deo izlaza alata Cppcheck

Poruke na slici 7 ukazuju na problem klase *Server* koja ima privatne promenljive a nema definisan konstruktor, koji bi bio potreban za njihovu inicijalizaciju. Zatim, za datoteku `Client.cpp` pojavljuje se poruka da je uslov u `else if` naredbi uvek tačan i treba proveriti da li postoji greška u logici, kao i mogućnost da je ovo potencijalno nepotreban kod.

4 Clangd

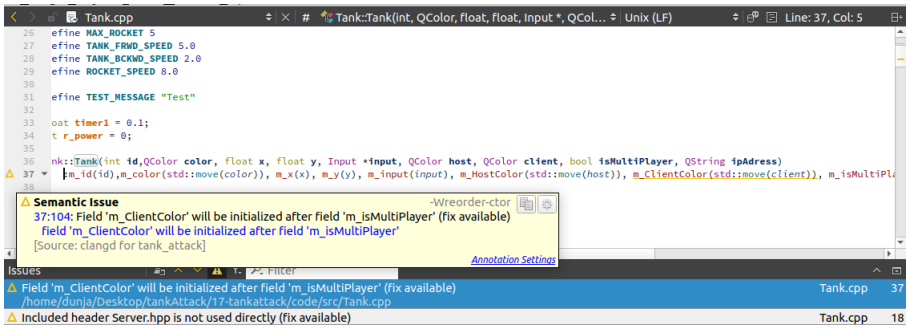
Sledeći alat koji ćemo primeniti je `Clangd`. Detektuje različite vrste grešaka i upozorenja u `C++` kodu, poput sintaksnih i semantičkih grešaka, upozorenja o kodu koji se ne koristi i slično. On omogućava brzu analizu te je zato pogodan za primenu i na velikim i kompleksnim projektima. Pruža podršku za refaktorisanje koda u cilju poboljšanja strukture i čitljivosti

koda.
Pokazaćemo upotrebu ovog alata pomoću QtCreator-a. Potrebno je da klikom na *Edit* izaberemo opciju *Preferences* i odaberemo opciju *C++*.Potom, kao što je prikazano na sledećoj slici, treba da izaberemo karticu *Clangd*, čekiramo polje *"Use clangd"* i kliknemo na dugme *Apply*.



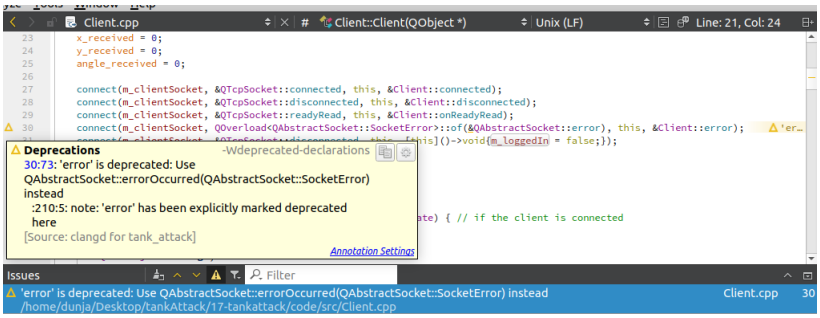
Slika 8: Podešavanja za clangd

Alat nam daje poruke o više pronađenih grešaka u različitim fajlovima. Na slici 9 vidimo poruke za fajl `Tank.cpp`. Prva poruka je upozorenje o potencijanom problemu u inicijalizaciji članova klase Tank. Član klase 'm_ClientColor' će se inicijalizovati posle 'm_isMultiPlayer' što može biti problem ukoliko redosled inicijalizacije članova utiče na ispravno funkcionisanje klase. Podatak `'(fix aviable)'` nam govori da postoji automatsko popravljanje ovog problema, tj. zamenom redosleda inicijalizacije članova problem bi bio rešen. Druga poruka odnosi se na zaglavlje koje je uključeno, a ne koristi se direktno u kodu. Takođe, postoji automatsko popravljanje ovog problema, tj. clangd je prepoznao da se zaglavlje `Server.hpp` može isključiti iz fajla `Tank.cpp`.



Slika 9: Poruke za falj Tank.cpp

Sve informacije dobijene ovim alatom date su u folderu `Clangd`. Prokomentarisaćemo još jedan tip poruke koju smo dobili za fajlove `Client.cpp` i `main.cpp`. Prikazana je na slici 10. Ukazuje na korišćenje zastarele funkcije `'error'` čije se dalje korišćenje ne preporučuje. Umesto nje preporučuje se korišćenje druge funkcije koja ima sličnu funkcionalnost, a koja neće generisati upozorenja.



Slika 10: Poruka za falj Client.cpp

5 Callgrind

Sledeći alat koji ćemo primeniti je Valgrind alat- `Callgrind`. Ovaj alat generiše listu poziva funkcija korisničkog programa u vidu grafa. Zahvaljujući grafu poziva, može da se odredi, počevši od main funkcije, koja funkcija ima najveću cenu poziva. Callgrind prikuplja podatke o tome koliko vremena program provodi u svakoj funkciji, što omogućava precizno merenje performansi, a takođe, broji i koliko se instrukcija izvršava u svakoj funkciji, što može pomoći u identifikaciji delova koda koji troše najviše procesorskog vremena. U kombinaciji sa `Kcachegrind` grafičkim korisničkim interfejsom, Callgrind omogućava programerima dublju analizu performansi njihovih programa i efikasno otkrivanje problema u kodu.

Prvo je potrebno da prevedemo program u debug režimu, na sledeći način: pozicioniramo se u direktorijum projekta i prevedemo program sledećim komandama:

```
qmake CONFIG+=DEBUG
make
```

Ili u `.pro` fajlu dodamo:

```
CONFIG+=DEBUG
```

Pozicioniramo se u direktorijum gde se nalazi izvršni fajl i alat pokrećemo sledećom komandom:

```
valgrind --tool=callgrind --callgrind-out-file="izlaz.txt" ./tank_attack
```

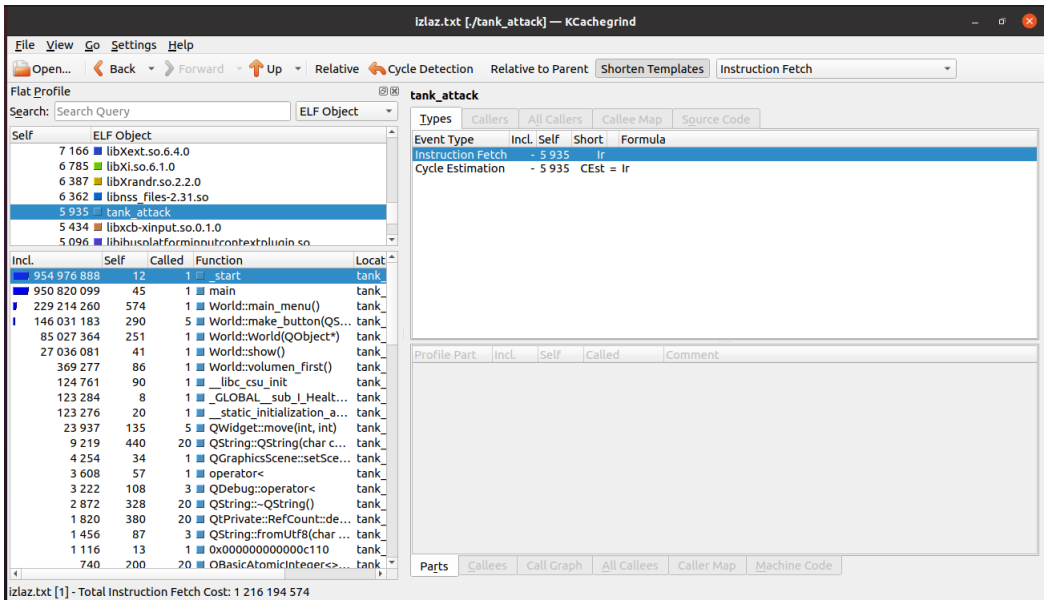
Kako bismo imali lep prikaz ovog izveštaja, otvorićemo ga uz pomoć Kcachegrind-a. Ukoliko je potrebno, prethodno ga treba instalirati:

```
sudo apt install kcachegrind
```

Pokrećemo ga komandom:

```
kcachegrind izlaz.txt
```

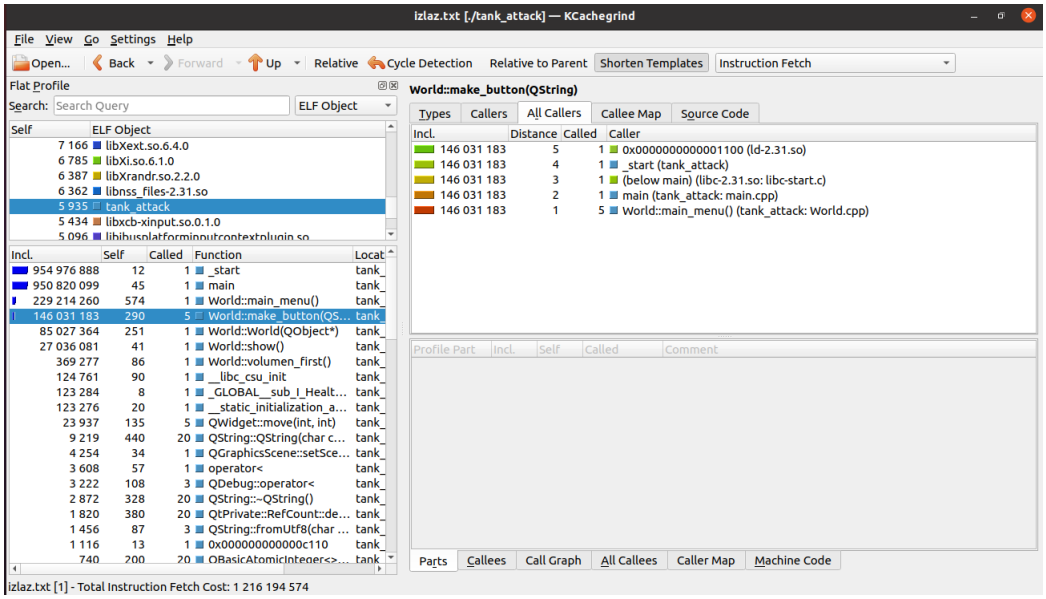
Alat možemo pokrenuti i pomoću skripte run_callgrind.sh koja se nalazi u okviru foldera `Callgrind`.



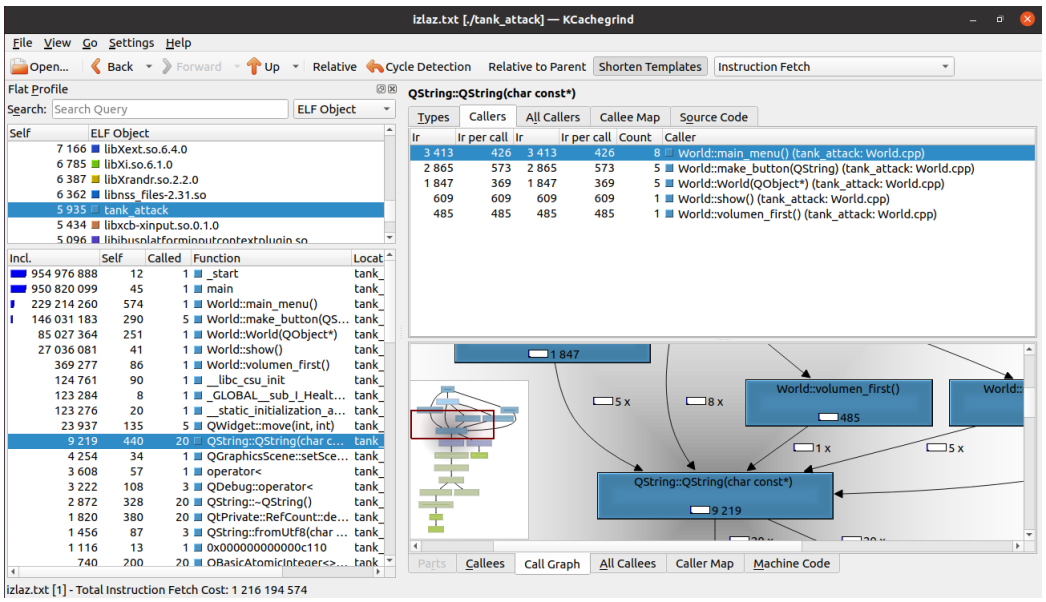
Slika 11: Vizuelan prikaz izvestaja preko Kcachegrind

Zanimaju nas funkcije koje se najviše puta pozivaju. Na levoj strani se nalaze informacije o broju pozivanja svake funkcije i broju instrukcija koji je zahtevalo njeno izvršavanje, samostalno i uključu- jući izvršavanja drugih funkcija koje je pozivala. Na desnoj strani možemo izabrati opciju *All Callers* i videćemo koje sve funkcije su pozivale funkciju koja nas zanima.

Na slici 12 vidimo da je funkcija World::make_button pozivana 5 puta, a sa desne strane izborom opcije *All Callers* vidimo spisak funkcija koje su je pozivale.



Slika 12: Vizuelan prikaz izvestaja preko Kcachegrind



Slika 13: Vizuelan prikaz izvestaja preko Kcachegrind

Na slici 13 vidimo detalje o funkciji koja je pozivana 20 puta. Sada je generisan i prikaz grafa poziva, što je urađeno opcijom *Call Graph*. Zaključak je da nema velikih broja poziva funkcija u delu koji je implementiran od strane programera ovog projekta.

6 Zaključak

Analizom ovog projekta došli smo do zaključka da postoje greške koje bi trebalo ispraviti. Ukazano je na postojanje propusta koji narušavaju čitljivost i održivost koda. Za većinu poruka, koje smo dobili primenom ovih alata, dato je detaljnije objašnjenje, kao i na koji način ispraviti greške i upozorenja na koje se poruke odnose.