Analiza projekta koriscenjem alata za verifikaciju softvera

Seminarski rad u okviru kursa
Verifikacija softvera

Matematički fakultet sara.kapetinic.sk@gmail.com

Sara Kapetinić

Avgust 2023.

Sažetak

Verifikacija softvera obuhvata proces evaluacije sistema ili komponente kako bi se utvrdilo da li proizvodi trenutne faze razvoja zadovoljavaju uslove postavljene na pocetku date faze. Ova analiza softvera obuhvata skup metoda, alata i procesa za utvrdjivanje ispravnosti softvera. Tehnike verifikacije se dele na dinamicke i staticke. Dinamicke obuhvataju ispitivanje koda tokom njegovog izvrsavanja, dok su staticke osvrnute na izvorni kod.

Sadržaj

1 Uvod

Projekat *Tudu* predtavlja jednostavni projekat u svrhu organizacije vremena. *Tudu* je nedeljni drag & drop planer, sa integrisanom listom obaveza (todo listom). Uz pomoć ove aplikacije možete organizovati svoje planove na pregledan i sitematizovan način. U listu obaveza se dodaju sve obaveze koje planirate da obavite u narednom periodu i zatim im se dodeljuje prioritet, tako da imate sistematizovan spisak obaveza koji omogućava da nikad ne zaborative na bitan sastanak, porodični ručak ili neku drugu obavezu. Iz liste obaveze možete premestiti u nedeljni kalendar i tako na pregledan način u svakom trenutku videti šta vas očekuje ove, ili bilo koje naredne nedelje.

2 Valgrind

Valgrind je GPL sistem za debagovanje i profajliranje Linux programa. Sa Valgrindovim paetom alata mozete detektovati mnoge probleme kod upravljanja memorije ili rada sa nitima. Takodje mozete izvrsiti detaljno profilisanje da biste ubrzali svoje programe. Prednosti Valgrinda:

- Valgrind je besplatan.
- Mozete se koristiti na razlicitim Linux platormama kao sto su: x86/Linux, AMD64/Linux PPC32/Linux.
- Radi na svim glavnim Linux distribucijama.
- Moze pomoci u ubrzanju programa.
- Znacajno skracuje vreme debagovanja.
- Lak je zakoriscnenje. Nije protrebno rekompajliranje programa vec samo pokretanje sa prefiksom valgrind.

Neki od Valgrindovih alata koji su korisceni na *Tudu* projektu su Callgrind, Memcheck i Massif. U narednom delu reci cemo nesto vise o njima i deonstrirati njihovu upotrebu.

Pre koriscenja Valgrind alata izmeni smo main, kako bi koriscenje alata bilo korisnije i kako bismo izbegli Qt-ove pozive. Main koji smo generisali izgleda kao na slici 1.

```
#include "headers/mainwindow.h"                    Included header mair
#include "headers/task.h"
#include "headers/tudulist.h"                      Included header tu
#include <iostream>
#include <QApplication>

int main(int argc, char *argv[])                                  Unuse
{
    int i = 0;

    while(i != 100){
        Task * task = new Task("Task"+QString::number(i) ,"desc",i%3);
        task->setName("New name"+QString::number(i));
        task->setStartTime(QDateTime::currentDateTime());
        qDebug()<<task->getName();
        qDebug()<<task->getTaskRow();
        task->save("tudu.json");
        i++;
        delete task;
    }

}
```

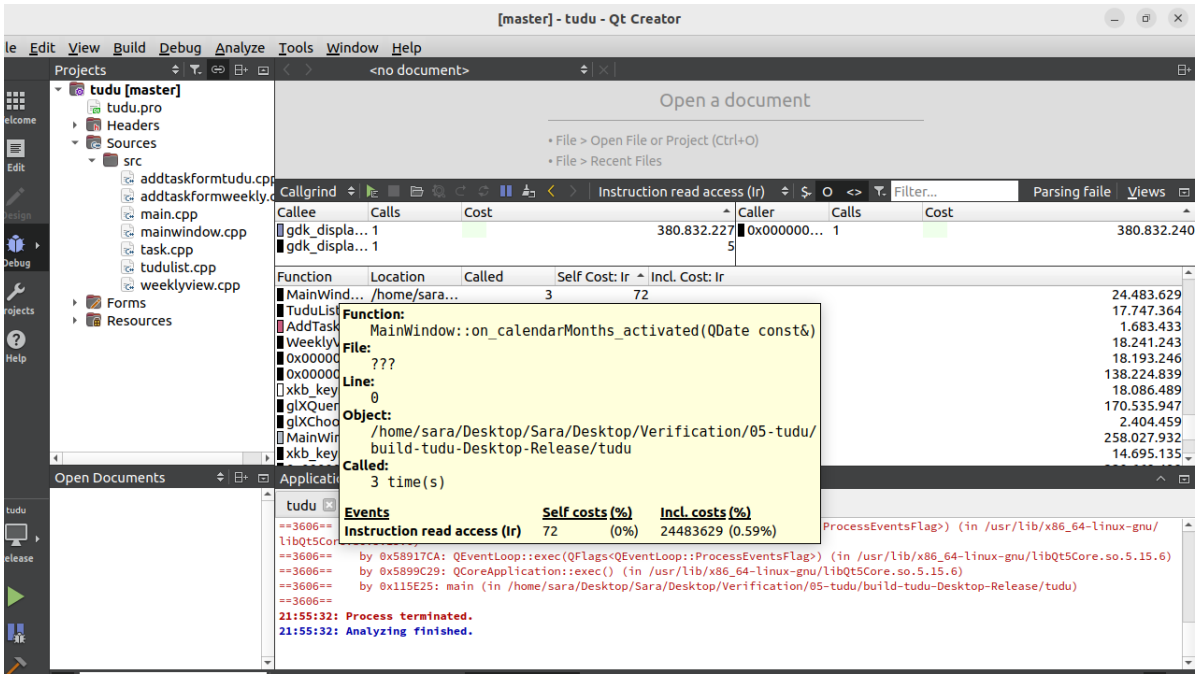
Slika 1: Novi main za testiranje pomocu Valgrind-a

2.1 Callgrind

Callgrind je alat za profilisanje koji belezi istoriju poziva medju funkcijama u programu koji se izvrsava kao graf poziva. Prikupljeni podaci se sastoje od: broja izvrsenih instrukcija, njihovog odnosa sa izvornim linijama koda, odnosa pozivalac/pozvani medju funkcijama i broja takvih poziva.

Posto je *Tudu* projekat radjen u Qt-u koji ima dosta ugradjenih Valgrind alata, simuliracemo prvo koriscenje Callgrind-a kroz Qt.

Postrebno je u Qt Creator-u uci u Debug mode i namestiti Callgrind. Zatim cemo pokrenuti projekat sa Callgrindom i testirati sto vise funkcionalnosti. Kao sto mozemo videti na slici 2 dobicemo izlistane sve funkcije kroz koje smo prosli i koje su se pozivale tokom naseg testiranje programa. Ukoliko stanemo na neku funkciju, mozemo detajnije pogledati informacije o njoj. Koriscenjem Callgrind alata na ovaj nacin, nismo dobili previse informacija o pozivima funkcija jer Tudu projekat se sastoji od dosta UI Qt poziva.



Slika 2: Callgrind Qt Creator

Drugi nacin pokretanja sa Callgrindom je iz terminala. Na ovaj nacin mozemo generisati callgrind izlazni fajl. Ovaj fajl nije preterano citljiv, ali se koristi kao ulaz za neke druge alate.
Komanda pomocu koje dobijamo callgrind izlazni fajl:

```
$ valgrind --tool=callgrind --callgrind-out-file=callgrind.out ./tudu
```

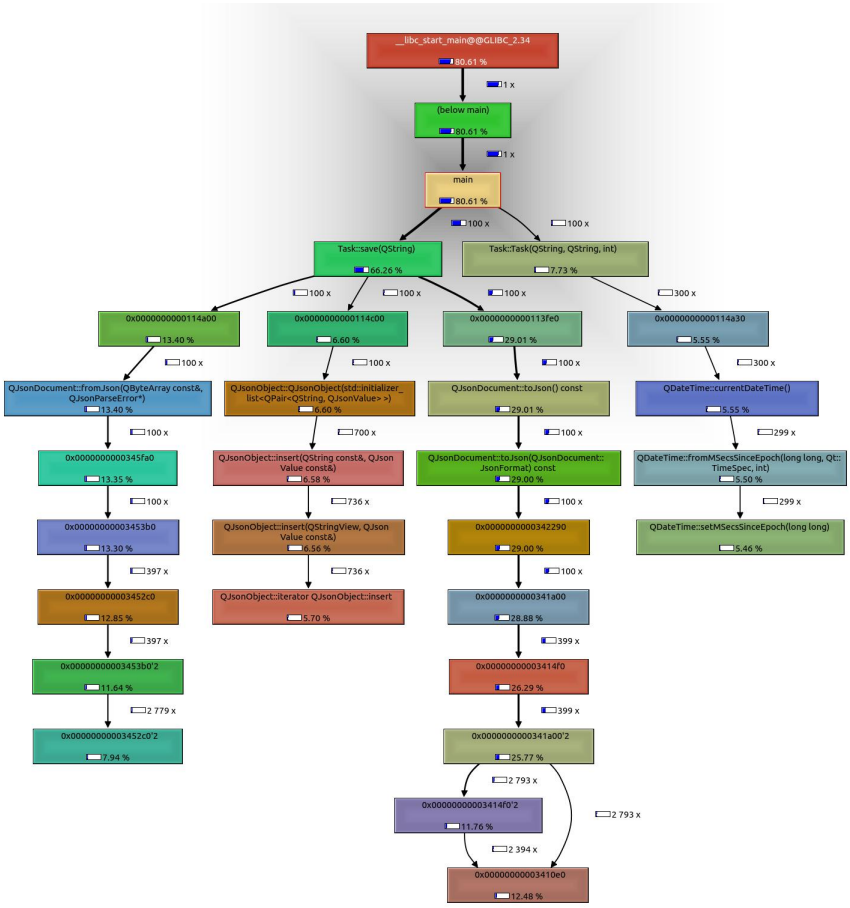
Pomocu ovog alata mozemo videti i broj pozivanja sistemskih funkcija, cime dobijamo sliku i o njihovom koriscenju. Ovaj alat dozvoljava pametnu manipulaciju kodom jer nam daje informacije o broj poziva metoda. Prema tome cesto pozivane metode treba da budu sto efikasnije. Izvrsavanje programa zajedno sa callgrind-om je znatno sporije nego bez njega.

2.1.1 KCacheGrind

Kcachegrind je alat za vizuelizaciju podataka profila, koja se koristi za odredjivanje delova koji oduzimaju najvise vremena pri izvrsavanju programa.

Komanda za pokretanje kcachegrind alata, kao ulaz se prosledjuje izlazni fajl dobijen koriscenjem callgrind-a:

```
$ kcachegrind callgrind.out
```

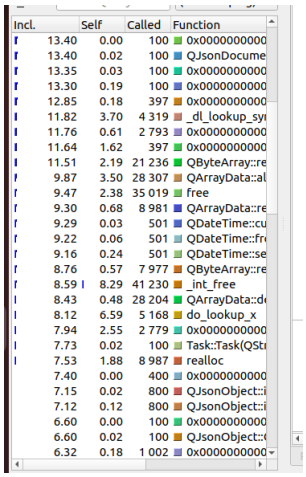


Slika 3: KCachegrind

Na slici 3 mozemo videti graf poziva funkcija na nasem *Tudu* projektu. Procenti na grafu poziva oznacavaju procenat od ukupnog utroska resursa(u vecini slucajeva procesorsko vreme), dok brojevi na strelicama se odnose na broj puta koliko metoda pozivalac poziva drugu metodu. Predstavljen je graf poziva iz main funkcije, gde se kroz 100 iteracija pravi novi task i cuva, sto mozemo videti i na grafiku.

Na slici 4 je prikazan KCachegrind-ov stack sa svim pozivima funkcija kroz koji se mozemo kretati. Mozemo videti da funkcije koje se najvise puta pozivaju su funkcij za alokaciju i dealokaciju memorije. Preko Grafika sa slike 3 mozemo videti i da se prilikom cuvanja task-a, pozivaju tri razlicite metode koje radi sa json objektima.

Ovaj alat se koristi i u kombinaciji sa nekim drugim valgrind alatima sto cemo videti u nastavku.



Slika 4: KCachegrind with Callgrind

2.2 Memcheck

Meecheck je Valgrind-ov alat koji sluzi za detektovanje gresaka vezanih za memoriju. Koristi se za programe pisane u C i C++ jezicima. Neka od svojstava koje Memcheck moze detektovati:

- Prisupanje nedozovvljenoj memoriji
- Koriscenje nedefinisanih vrednosti
- Nekorektno oslobadjanje hip memorije
- Curenje memorije

Kao i Callgrind, Memcheck nije tezak za koriscenje. U okviru Qt Creator-a se nalazi podrska za Memcheck. Na slici 5 mozemo videti koriscenje ovog alata kroz Qt na projektu Tudu. Pri prvom pokretanju memcheck-a iz Qt Creatora nisu se pojavile nikakve greske sa memorijom. Projekat je pkretan sa novim main-om sa slke 1, s obzirom da sva alocirana memorija se oslobadja i da je ovo mali projekt nismo dobili nikakvo curenje memorije.

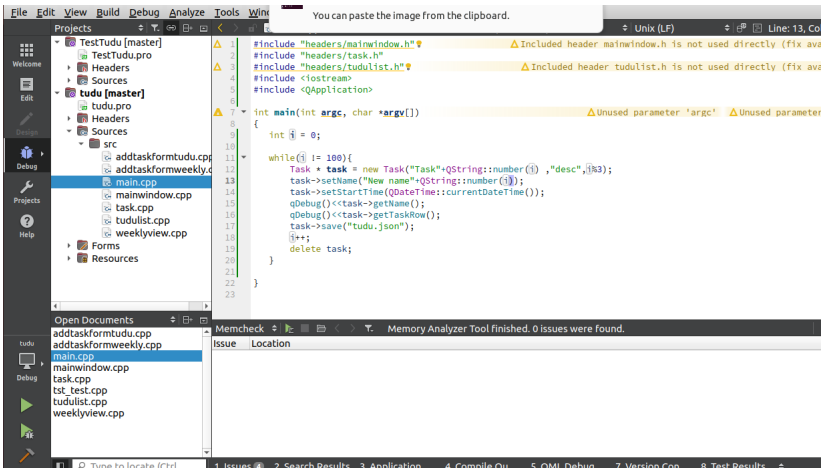
Memcheck se moze koristiti i iz terminala, pokretanjem komande:

```
$ valgrind --leak-check=full --leak-resolution=high --show-leak-kinds=all
--xtree-leak=yes ./tudu
```

Flag-ovi koji su korisceni u prethodnoj komandi oznacavaju:

- –leak-check=full svako curenje memorije ce biti zabelezeno i bice uracunato u greske
- –leak-resolution svi unosi moraju da se poklapaju
- –show-leak-kinds=all i –xtree-leak=yes za dobijanje izlaznog fajla u callgrind formatu

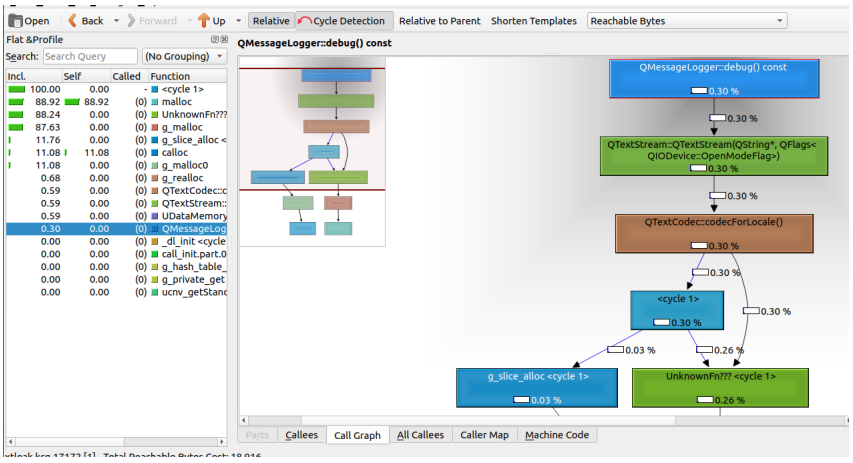
Pokretanjem ove komande dobili smo izlazni fajl koji sadrzi izvestaj o memoriji, kao i drugi izvrsni fajl koji se moze otvoriti pomocu kcachegrind-a. Kao sto mozemo videti na slici 6 i ovde nemamo nikakvo curenje memorije, sto je bilo i ocekivano, testiramo cpp klasu odvojeno od Qt-a, i dealociramo svu memoriju koju smo alocirali, odmah.



Slika 5: Memcheck in Qt

```
49
"New name98"
49
"New name99"
49
==17172==
==17172== HEAP SUMMARY:
==17172==     in use at exit: 18,916 bytes in 11 blocks
==17172==   total heap usage: 85,917 allocs, 85,906 frees, 9,538,229 bytes allocated
==17172==
==17172== xtree leak report: /home/sara/Desktop/Sara/Desktop/odbrana/05-tudu/tudu/xtleak.kcg.17172
==17172== LEAK SUMMARY:
==17172==    definitely lost: 0 bytes in 0 blocks
==17172==    indirectly lost: 0 bytes in 0 blocks
==17172==      possibly lost: 0 bytes in 0 blocks
==17172==    still reachable: 18,916 bytes in 11 blocks
==17172==         suppressed: 0 bytes in 0 blocks
==17172==
==17172== For lists of detected and suppressed errors, rerun with: -s
==17172== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Slika 6: Memcheck leak summary



Slika 7: Memcheck in KCachegrind

Na slici 6 nalazi se izvestaj o koriscenju hip memorije, dok na slici 7 mozemo videti izgenerisan kcg fajl u kcachegrind-u. Najveci postotak memorije odlazi na nase alociranje, sve ostalo je zanemarljivo.

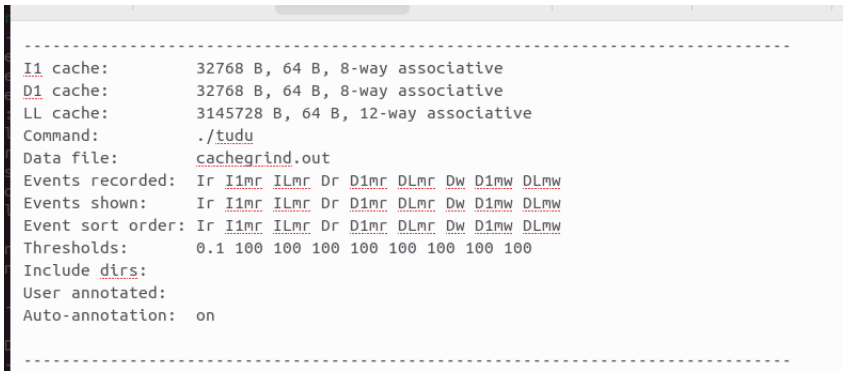
2.3 Cachegrind

Cachegrind je alat Valgrind-a koji omogucava softversko profajliranje kes memorije tako sto simulira i prati pristup kes memoriji masine na kojoj se program, koji se analizira, izvrsava. Takode, moze se koristiti i za profajliranje izvrsavanja grana. Cachegrind simulira memoriju masine, koja ima prvi nivo kes L1 memorije podeljene u dve odvojene nezavisne sekcije: I1 - sekcija kes memorije u koju se smestaju instrukcije D1 - sekcija kes memorije u koju se smestaju podaci. Drugi nivo kes memorije koju Cachegrind simulira je objedinjen - LL. Ovaj nacin konfiguracije odgovara mnogim modernim masinama. Vazno je da u okviru makefile-a ne posledimo kompajleru g++ -O0 jer zelimo da testiramo program u njegovom normalnom izvrsavanju.

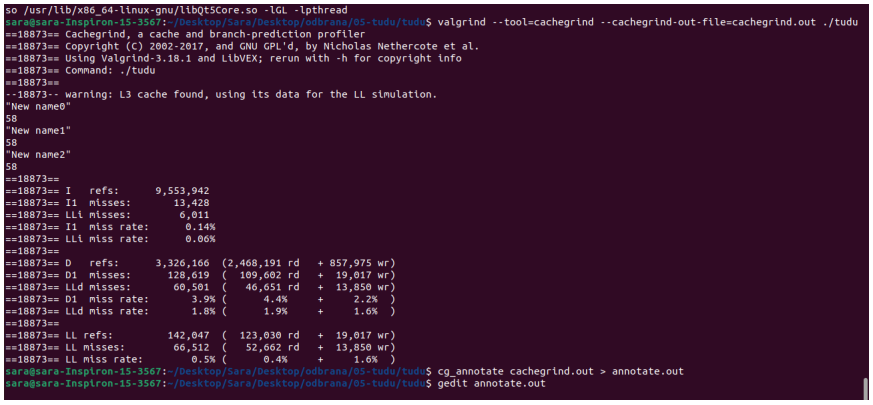
Komanda za pokretanje ovog alata:

```
$ valgrind --tool=cachegrind --cachegrind-out-file=cachegrind.out
./tudu
```

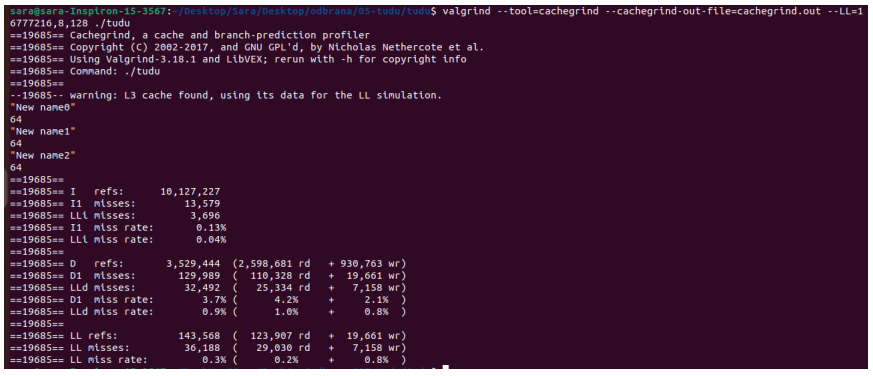
Ovom komandom smo generisali detiljniji izlazni fajl koji mozemo pomocu cg_annotate ispisati. Na slici 8 mozemo videti osobine kesa koje se nalaze na pocetku generisanog fajla. Prvi broj predstavlja velicinu kesa, drugi broj predstavlja velicinu linije dok treci predstavlja asocijativnost kesa. Prilikom pozivanja alata Cachegrind moguce je da promenimo velicinu LL kesa. Na slici 9 nalazi se stanje kesa tokom izvrsavanja Tudu projekta.



Slika 8: Cachegrind information about cache

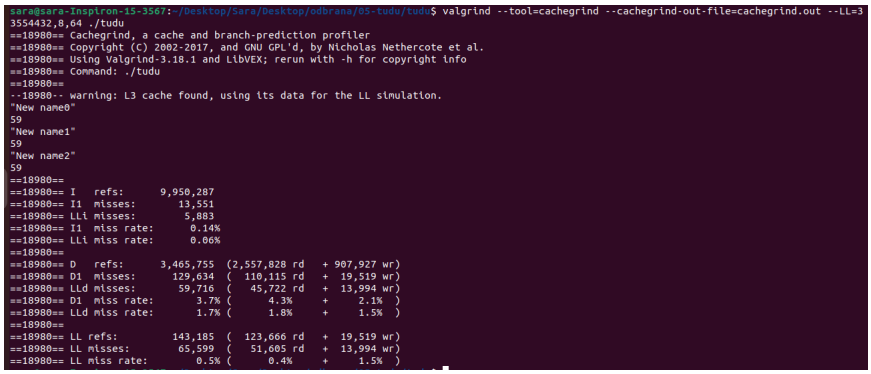


Slika 9: Cachegrind on Tudu with usual options



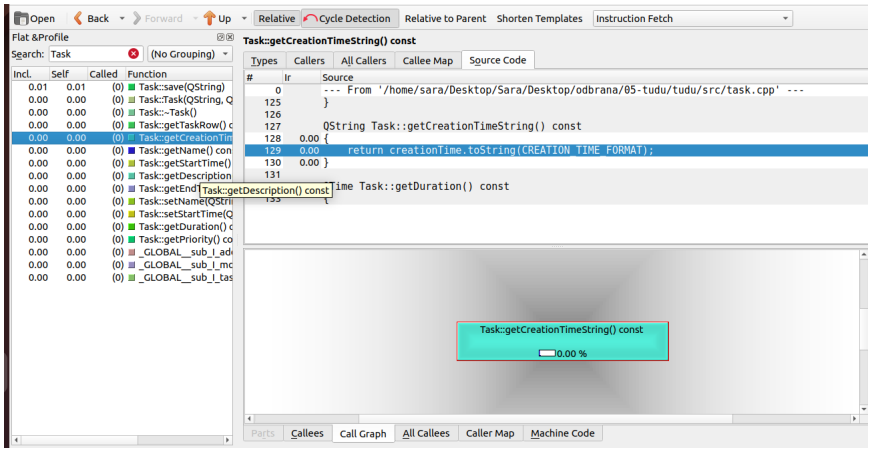
Slika 10: Cachegrind on Tudu with changed options

Na slikama 10 i 11 mozemo videti pokretanje cachegrinda sa drugacijim opcijama, gde manipulisemo LL kes-om. Mozemo primetiti da ukoliko samo povecamo velicinu kesa na 32MB, necemo dobiti previse znacajne



Slika 11: Cachegrind on Tudu with changed options

promene sto se tice promasaja kesa. Ali ukoliko proenimo elicinu linije kao na 10, vidimo da se stopa promasaja smanjila za 0,2%.



Slika 12: Cachegrind in KCacheGrind

Na slici 12 prikazan je alat cachegrind u okviru KCacheGrind-a, gde mozemo izvuci sledeca zapazanja. Iako smo modifikovali main, i dalje kes memoriji najvise pristupaju Qt objekti za alokacija u dealokaciju. Jedina metoda koje ima neki procenat udela u promajajima kes-a je funkcija save, iako je i njen procenat zanemarljiv.

2.4 Massif

Massif je hip profajler. Meri koliko hip memorije program koristi. Ovo podraumeva i koriscenu memoriju i onu ekstra alociranu u svrhe poravnjanja. Takodje moze da meri velicinu steka programa. Profiliranje hipa moze da pomogne u redukciji memorije koju program koristi. Na modernim masinama sa virtuelnom memorijom, ovo moze doprineti:

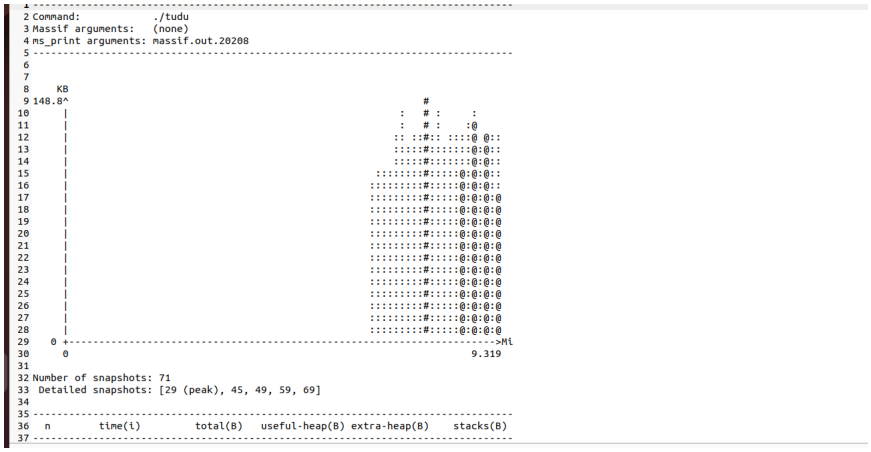
- Vecoj brzini izvrsavanja programa.
- Ako program koristi dosta memorije, smanjuje se sansa da se iscrpi sva memorija masine.

Massif se koristi lako kroz terminal, pokretanjem komande:

```
$valgrind --tool=massif ./tudu
```

Ovako dobijamo fajl massif.out<*PID*>, koji se moze citati pomocu ms_print komande. U ovom fajlu se nalaze detaljne informacije o koriscenju hip-a, kao sto je prikazano na slici 14. U sklopu massif fajla se nalazi i graf koji predstvalja memorijsku potrosnju. Tabela koja je prekizana na slici sastoji se od broja snepshota, vremena koje je potroseno, totalne memorijske potrosnje, broja koriscenih i dodatnih hip bajtova kao i velicinu steka.

Na slici 13 mozemo videt graf upotrebe heap memorije. Vidimo da se nova memorija prilikom prvih snepsotova nije alocirala, vec tek od sredine. Takodje mozemo videti koje snepshot-ovi su detaljnije opisani u massif.out fajlu(29,45,59...). Peak se dostize u 29 preseku i iznosi 148.8Kb, a nakon drugog preseka pocinje da se alocira veca memorija.



Slika 13: Massif graf

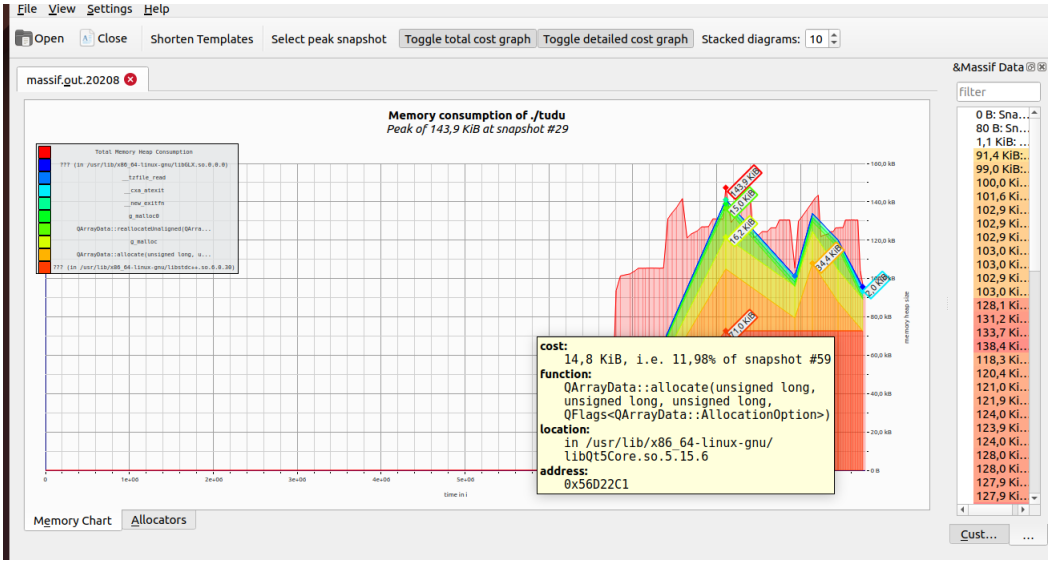
```
->21.18% (32,273B) 0x56D22C1: QArrayData::allocate(unsigned long,
unsigned long, unsigned long, QFlags<QArrayData::AllocationOption>)
(in /usr/lib/x86_64-linux-gnu/libQt5Core.so.5.15.6)
| ->10.86% (16,548B) 0x571FDF3: QByteArray::QByteArray(int, Qt::Initialization)
(in /usr/lib/x86_64-linux-gnu/libQt5Core.so.5.15.6)
| | ->10.77% (16,409B) 0x5709CB6: QRingBuffer::reserve(long long)
(in /usr/lib/x86_64-linux-gnu/libQt5Core.so.5.15.6)
| | | ->10.77% (16,409B) 0x5709F63: QRingBuffer::append(char const*, long long)
(in /usr/lib/x86_64-linux-gnu/libQt5Core.so.5.15.6)
| | | | ->10.77% (16,409B) 0x57D9016: QFileDevice::writeData(char const*, long long)
(in /usr/lib/x86_64-linux-gnu/libQt5Core.so.5.15.6)
| | | |   ->10.77% (16,409B) 0x57E323E: QIODevice::write(char const*, long long)
(in /usr/lib/x86_64-linux-gnu/libQt5Core.so.5.15.6)
| | | |     ->10.77% (16,409B) 0x117D9A: QIODevice::write(QByteArray const&) (qiodevice.h:137)
| | | |       ->10.77% (16,409B) 0x11DBBC: Task::save(QString) (task.cpp:183)
| | | |         ->10.77% (16,409B) 0x115269: main (main.cpp:17)
```

Deo izvestaja massif.out fajla je prikazan iznad, gde mozemo videti alociranje memorije od main funkcije, preko save metode klase Task, pa sve do allocate. Na slici 14 su prikazani podaci o odredjenim snepshot-ima, gde se vidi utrosak korisno alocirane memorije, kao i extra memorije koja je morala biti alocirana.

```
169
170 --------------------------------------------------------------------------------
171  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
172 --------------------------------------------------------------------------------
173 30      8,172,908          133,104          129,186         3,918            0
174 31      8,235,617          137,080          132,882         4,198            0
175 32      8,297,010          141,056          136,603         4,453            0
176 33      8,358,797          145,640          140,938         4,702            0
177 34      8,399,938          147,976          143,114         4,862            0
178 35      8,440,720          126,368          121,286         5,082            0
179 36      8,501,805          127,832          122,823         5,009            0
180 37      8,542,671          129,456          124,443         5,013            0
181 38      8,604,136          129,424          124,400         5,024            0
182 39      8,645,355          131,432          126,417         5,015            0
183 40      8,706,807          131,416          126,407         5,009            0
184 41      8,747,742          135,496          130,491         5,005            0
185 42      8,788,784          135,512          130,499         5,013            0
186 43      8,829,490          135,528          130,513         5,015            0
187 44      8,870,382          135,552          130,526         5,026            0
188 45      8,932,675          109,136          105,393         3,743            0
```

Slika 14: Massif snapshot

Radi lepseg prikaza mozemo instalirati massif visualizer, pomocu koga sva prethodna zapazanje mozemo lakse uociti. Ovaj alat je prikazan na slici 15



Slika 15: Massif visualizer

3 Coverage tools

GCov je alat koji se koristi zajedno sa GCC-om da bi se dobila pokrivenost koda u programu. Ovaj alat pomaze u pisanju efikasnije i brzeg koda, kao i u pronalazenju nedostiznih delova programa. Moze da se koristi kao alat za profilisanje, cime dobijamo ideju gde ce optimizacije koda biti ucinkovitije. Osnovne informacije koje dobijamo koriscenjem GCov alata:

- Koliko cesto se svaka linija koda izvrsava
- Koje linije se zapravo izvrsavaju
- Koiko racunarskog vremena koristi koji deo koda

3.1 Unit Testovi

Kako bi testirali Tudu projekat sa GCov alatom, potrebno je dodati unit testove. Pomocu Qt-a moguce je napraviti test projekat, koji se pokrece kao obican Qt projekat pri cemu ukljucuje QtTest biblioteku. Jedna od najpoznatijih biblioteka za testove je Catch2, ali o njoj nekom drugom prilikom. Neki od primera testova na Tudu projektu su navedeni u daljem tekstu.

```
    void test::test_case1()
{
    qDebug()<<"task1";

    TuduList *tudulist = new TuduList();
    tudulist->show();
    tudulist->addTask("Task1","desc1",0);
    QStandardItemModel* model = static_cast<QStandardItemModel*>(tudulist->model());
        bool taskFound = false;
        for (int row = 0; row < model->rowCount(); ++row)
```

```
        {
            QModelIndex index = model->index(row, 0);
            QString taskName = model->data(index).toString();
            if (taskName == "Task1")
            {
                taskFound = true;
                break;
            }
        }

        QVERIFY(taskFound);

}
```

Test1 proverava dodavanje novog task-a u tudu listu, koriscenjem QVERIFY-a.

```
    void test::test_case3()
{
    qDebug()<<"task3";
    Task task("Sample Task", "Sample Description", QDateTime::currentDateTime(), QDateTime::cur

    QString fileName = "sample_task";
    task.save(fileName);
    QString saveLocation = QString("/home/sara/Desktop/Sara/Desktop/test.json");
    QFile file(saveLocation);
    file.open(QIODevice::ReadOnly);
    QString val = file.readAll();
    QJsonDocument d = QJsonDocument::fromJson(val.toUtf8());

    QVERIFY(!d.isNull());
}
```

Prethodni test verifikuje cuvanje taska u json fajlu. S obzirom da je postojalo par problema u kodu, putanja na kojoj se cuva fajl je hardkodirana, ali se test izvrsava uspesno.

```
    void test::test_case4()
{
    qDebug()<<"task4 --- taskk";
    Task task("Sample Task", "Sample Description", QDateTime::currentDateTime(), QDateTime::cur
    task.setPriority(0);
    QIcon highPriorityIcon = task.fetchIcon(task.getPriority());
    task.setPriority(1);
    QIcon midPriorityIcon = task.fetchIcon(task.getPriority());
    task.setPriority(2);
    QIcon lowPriorityIcon = task.fetchIcon(task.getPriority());

    QVERIFY(!highPriorityIcon.isNull());
    QVERIFY(!midPriorityIcon.isNull());
    QVERIFY(!lowPriorityIcon.isNull());
}
```

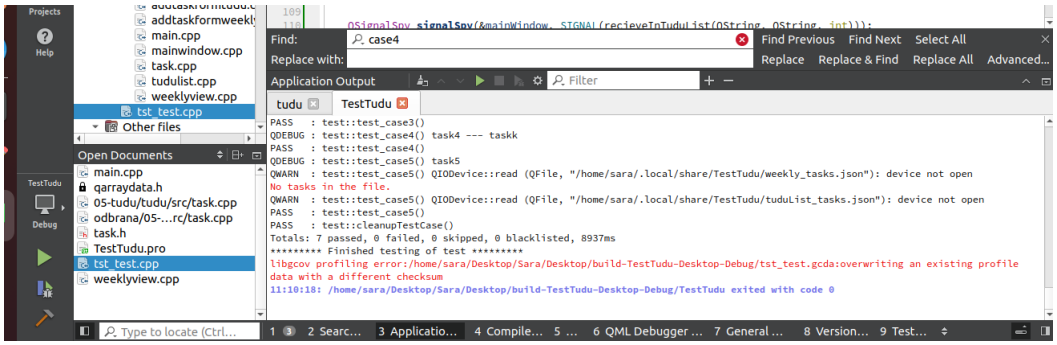
Jedan od dodataka Tudu projektu jesu ikonice za prioritet taskova. Ovaj test ce nam pokazati da li ikonice dobro dohvatamo.

```
void test::test_case5(){
qDebug()<<"task5";

MainWindow mainWindow;

QSignalSpy signalSpy(&mainWindow, SIGNAL(recieveInTuduList(QString, QString, int)));

mainWindow.on_addTaskButtonClicked();

QList<QWidget *> openDialogs = QApplication::topLevelWidgets();
QVERIFY(!openDialogs.isEmpty());


QVERIFY(signalSpy.isValid());

}
```

Poslednji test koji cemo pominjati je vezan za trigerovanje addTask dugmeta i provere signala.

Kao sto mozemo primetiti vecina testova je vezana za sam Task, jer je to cista cpp klasa. Ostale klase sadrze Qt GUI objekte sto dodatno otezava testiranje. Ostali testovi su fokusirani na pravljenje task-ova, setovanje vremena i ostalih parametara, pa su od manje vaznosti.

Za aplikaciju kao sto je Tudu korisni testovi bi bil GUI testovi ili integracioni. Preporuka je koristiti Selenium.



Slika 16: Izvrsavanje Unit Testova u Qt-u

3.2 Koriscenje GCov alata

Alat GCov primenicemo na Tudu projektu. Koristicemo alat iz terminala pomocu naredne komande

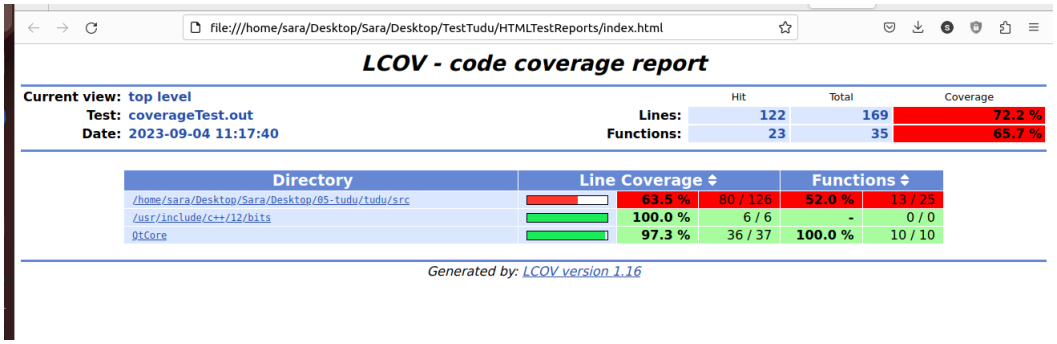
```
$lcov --rc lcov-branch-coverage=1 -c -d . -o coverage.out
```

Generisali smo .gcda fajlove kao i coverage.out fajl. Da bi imali lepsi vizuelni prikaz podataka koristicemo genhtml komandu, kojom cemo generisati html stranicu sa svim podacima koje smo dobili GCov-om.

Komanda za generisanje html stranice na osnovu .out fajla:

```
$ genhtml --rc lcov-branch-coverage=1 -o HTML_Reports coverage.out
```

Svi potrebni podaci smesteni su u direktorijum HTML_Reports, odakle mozemo otvoriti html stranicu i analizirati dobijene informacije.

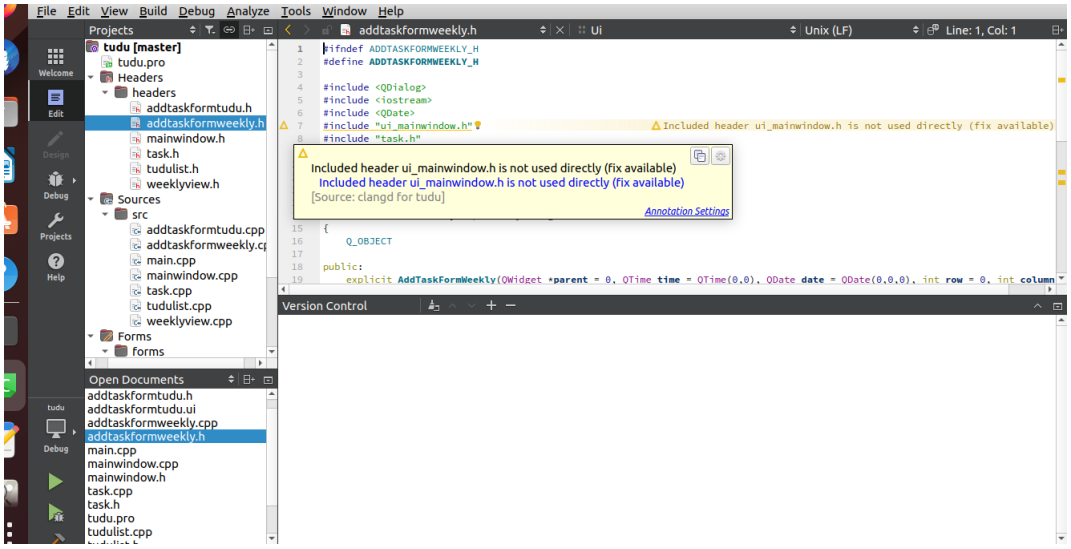


Slika 17: GCov alat primenjen na Tudu projektu

Na slici 17 mozemo videti neke dodatne informacije. Posto je Tudu jednostavan projekat, unit testovima uspeli smo da predjemo dve trecine linija koda i funkcija. U tabeli nam je oznaceno u kojim delovima se nalaze funkcije koje nisu pokrivene, ili linije koda koje nisu dostignute. Za testiranje programa koji se najvise oslanja na GUI, korisniji su GUI testovi kako bi se pokrilo sto vise slucajeva.

4 Clandg i CppCheck

Clangd je jezicki server koji postoji ugradjen u mnogim razvojnim okruzenjima. Zasnovan je na Clang C ++ kompilatoru i deo je LLVM projekta. Koristi se za provjeru ispravnosti koda: ispituje kompletnost, nalazi kompilacione greske.. Clangd mozemo koristiti kroz Qt uz pomoc sledecih koraka: Tools -> Options -> C++ -> Clangd. Koriscenjem clangd-a na projektu Tudu, dobili upozorenje prikazano na slici 18. Linije koja predtsavlja upozorenje je zuta, sa zutim trouglom. Kada stanemo na trougao mozemo videti detaljniji opis problema, kao i to da nam je clangd prijavio upozorenje.

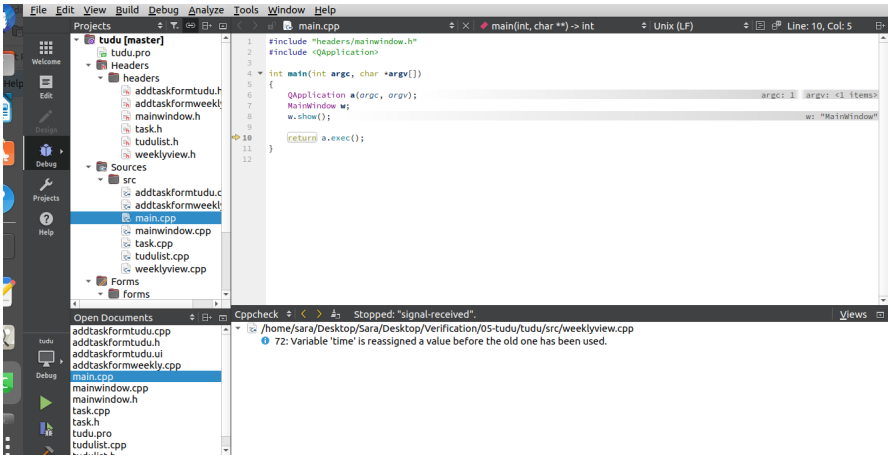


Slika 18: Clangd primenjen na Tudu projektu

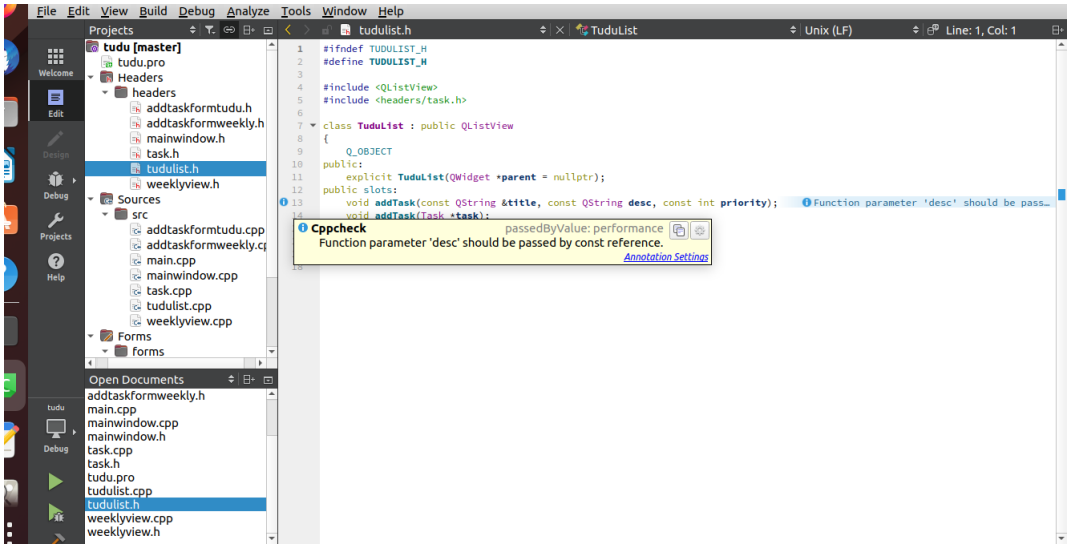
CppCheck se koristi za staticku analizu C/C++ programa. Daje nam jedinstvenu analziu koda za detektovanje bagova i fokusira se na pronalazenje nedefinisanog ponasanja i opasnih struktura. Primeri nedefinisanog ponasanja:

- Deljenje nulom.
- Neinicijalizovane varijable.
- Nevalidne konverzije.
- Nekorisceni pokazivaci.

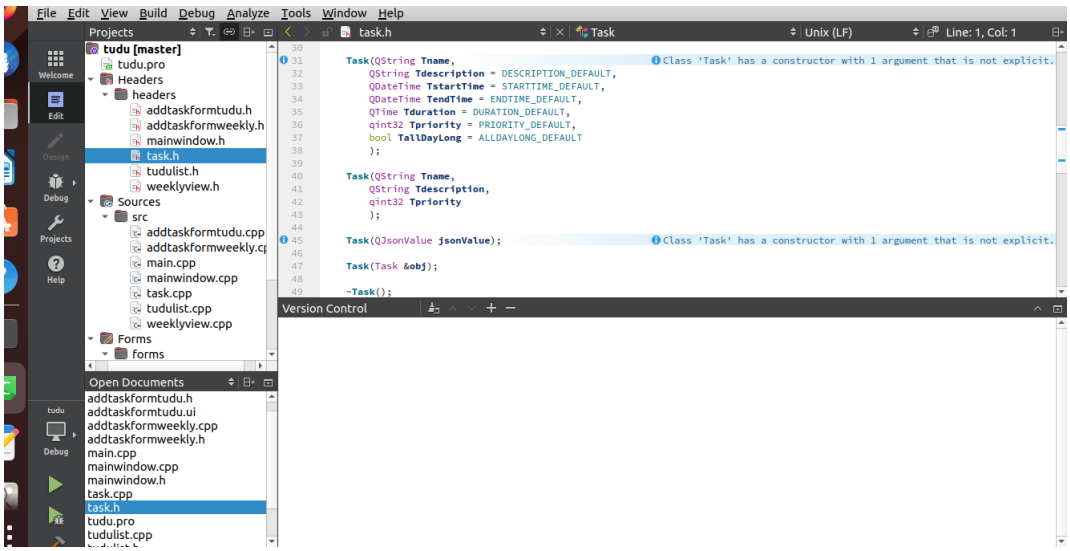
Podrska za koriscenje CppCheck alata postoji u okiru Qt-a, a sad cemo kroz Tudu projekat prikazati neke od navedenih gresaka koje je CppCheck nasao.



Slika 19: CppCheck i koriscenje promenljivih



Slika 20: CppCheck i const parametri



Slika 21: CppCheck i konstruktori

Na prethodnim slikama mogli smo videti neke od predloga izmena koje nam je CppCheck predlozio. Deo koda na kome je predstavljen predlog za promenljivu *time* se nalazi ispod. Promenljiva je inicijalizovana, ali joj se vrednost odmah menja, i cppcheck predlaze da inicijalizacija nije neophodna.

```
 QString time;
for (int i=0; i<HOURS_IN_DAY; i++) {
    for (int j=0; j<MINUTES_IN_HOUR; j+=MINUTE_INCREMENTS) {
```

```
            time = "";
//            time.sprintf("%02d:%02d", i, j);
            time = QString("%1:%2")
                    .arg(i, 2, 10, QLatin1Char('0'))
                    .arg(j, 2, 10, QLatin1Char('0'));
            m_verticalHeaders << time;
        }
    }
```

5 Zakljucak

U slucaju razvijanja velikih i kompleksnih programa prethodni alati nisu samo preporucljivi, vec u vecini slucajeva mogu biti i nephodni. Svaki alat ima svoje prednosti i mane, i vecina programera nije navikla da ih svakodnevno koristi, ali bi to trebalo promeniti. Kao sto smo mogli da primetimo, cak i na malom projektu kao sto je Tudu, uocili smo neke greske i propuste. Koliko bi ih tek bilo da je projekat dva, tri ili deset puta veci? Analiziranje svih tipova memorije, procesorskog vremena, efikasnosti programa, pokrivesnost koda pomazu u izgradnji softvera koji je od vece pouzdanosti i boljih performansi. Nekad je potrebno razvijati softver u ogranicenim uslovima, gde su resursi mali. Tada bi posebno bilo potrebo obratiti paznju na neke od prethodnih alata.