TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatika

Timo Loomets 185994IAIB

ROS-IL PÕHINEVATE PROJEKTIDE AUTOMAATTESTIMISSÜSTEEMI ARENDUS JA INTEGRATSIOON ARETE TESTIMISSÜSTEEMIGA

bakalaureusetöö

Juhendaja
Gert Kanter
PhD

Tallinn 2021

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Timo Loomets

25.05.2021

Annotatsioon

Tallinna Tehnikaülikoolis on mitu ainet, mille raames puututakse kokku robotite programmeerimisega. Üks peamisi tööstuses kasutatavaid tehnoloogiaid on ROS (*Robot Operating System*), mida kasutatakse ka antud kursustel. Kuigi eksisteerivad ülesanded, puudus enne seda lõputööd võimalus lahendusi automaatselt simuleerida. Samas olid olemas automaattestimise lahendused, millega sai testida *Python* ja *Java* lahendusi.

Selle lõputöö eesmärk oli luua võimekus ROS lahenduste automaattestimiseks Tallinna Tehnikaülikoolis. Peamiseks kriteeriumiks oli võimaluse loomine võimalikult suure ühilduvusega olemasolevate süsteemidega. Lõputöös käsitletakse testide defineerimise võimalusi õppejõu vaatepunktist ja süsteemi integreerimiseks *GitLab*, *Arete*, *Charon* ja AI Lab süsteemidega lahendatud probleeme. Töö käigus selgus, et lahendus kiirendab testimist ja võimaldab paremat objektiivsust võrreldes käsitsi testimisega.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 27 leheküljel, 7 peatükki, 16 joonist, 1 tabel.

Abstract

DEVELOPMENT OF AUTOMATED TESTING SYSTEM FOR ROS-BASED PROJECTS AND INTEGRATION WITH TESTING SYSTEM ARETE

There are multiple courses in Tallinn University of Technology which include programming robots. One of the main technologies used in the industry is ROS (Robot Operating System) which is also used in some courses. Although there exist exercises for the given courses there was no solution for automated testing of them. At the same time there existed automated testing solutions for Python and Java.

The goal of this thesis was to create automated testing support for ROS-based projects for Tallinn University of Technology. The main criteria was to create the option with maximal integration with existing systems. Thesis covers test definition options from the viewpoint of a teacher and the problems encountered during integration with GitLab, Arete, Charon and AI Lab systems. The developed solution has been proven to speed up the testing process. The solution also improves marking neutrality by removing humans from the marking process.

Thesis describes the usage flow from the viewpoints of a developer, a teacher and a student. Main user stories explored in this work are: updating of testing environment, updating tests and running tests. The thesis also includes description for two different ways for defining tests and some recommendations for test structure. Finally, it explains how automated testing can save time for all the participants.

This thesis is written in Estonian and is 27 pages long, including 7 chapters, 16 figures, and 1 table.

Lühendite ja mõistete sõnastik

AI Lab	*Artificial Intelligence laboratory*
CPU	*Central Processing Unit*
GTest	*Google Test*
JSON	*JavaScript Object Notation*
ROS	*Robot Operating System*
SLURM	*Simple Linux Utility for Resource Management*
URDF	*Unified Robotic Description Format*
URL	*Uniform Resource Locator*
XML	*Extensible Markup Language*

Sisukord

Jooniste loetelu

Tabelite loetelu

1. Sissejuhatus

Koos tarkvaraarendusega kaasneb vajadus tulemusi valideerida. Üheks võimalikuks lahenduseks on automaattestimine, mis võimaldab hinnata tulemuse vastavust konkreetsetele nõuetele. Selline võimalus on vajalik akadeemilises keskkonnas, kus tudengeid on õppejõududest märkimisväärselt rohkem. Automaattestri ülesanne on antud olukorras automatiseerida lihtsalt kontrollitavate nõuete täitmist, et õppejõudude aega efektiivsemalt kasutada ja võimaldada keskendumist personaliseeritumale tagasisidele.

Praegu on olemas Tallinna Tehnikaülikoolis juba mitmed sarnased lahendused spetsiifilistes keeltes testimiseks. Näiteks on testrid *Python*'i, *Prolog*'i ja *C* keelte jaoks. Antud testrid toimivad vaid kindla keelega ja võimaldavad vaid üksustestimist. Hetkel on kasutuses järgnevad testrid:

- kattis-tester;
- uva-tester;
- java-tester;
- java11-tester;
- java13-tester;
- java15-tester;
- c-tester;
- python-tester;
- hackerrank-tester;
- prolog-tester;
- fsharp-tester.

Seega puudub hetkel võimalus automaatselt testida robootika lahendusi. Näited kursustest, milles õpetatakse robootikat ja milles lahendatakse ülesandeid kasutades ROS (*Robot Operating System*) raamistikku:

- ITI0201 - Robotite programmeerimine;
- IAS0220 - Robotite juhtimine ja tarkvara.

2. Hetkeseis

2.1 Olemasolevad lahendused

ROS raamistikus robootikalahenduste testimiseks on olemas juba lahendusi. Osasid neist lahendustest rakendatakse ka Tallinna Tehnikaülikoolis. Antud peatükis tuuakse välja, millised on alternatiivid lõputöös loodud lahendusele ja mis on nende eelised ja puudused.

2.1.1 Python tester

Ülikoolis on kasutusel automaattestimise süsteemi Arete integreeritud tester *Python* lahenduste jaoks. Seda rakendatakse näiteks Programmeerimise algkursusel, et testida Python lahendusi. Selle kasutamise peamisteks eelisteks on järgnevad omadused:

- integreeritus eksisteerivate süsteemidega;
- süsteem on testitud ja olnud kasutuses;
- ülesehitus on piisavalt kompaktne, et võimaldada laiendamist.

Samas kaasneksid selle kasutamisega ROS lahenduste testimiseks märkimisväärsed probleemid:

- testrisse peaks lisama ROS teegid, mis lisab tavalistele testidele *overhead*'i;
- testide vormistus peab sisaldama ROS süsteemi käivitamist;
- puudub tugi C++ keeles kirjutatud ROS lahendustele;
- vajaks märkimisväärset osa olemasolevast ressursist ja segaks takistaks testimisi.

2.1.2 Bag failidega testimine

ROS toetab saadetud sõnumite salvestamist ja taasesitamist kasutades *rosbag*[1] tööriista, salvestades informatsiooni *bag* laiendiga faili. Selle jaoks salvestatakse saadetud sõnumid koos ajatemplitega ja täpse struktuuriga. Sedasi on võimalik taas luua varasemaid situatsioone ning jälgida programmi koodi väljundit. Lahenduse puudus on tagasisidetsükli puudumine. Järelikult pole sedasi võimalik kontrollida, kas robot sõidaks mööda teistsugust

trajektoori. Samuti kaasneb *bag* failidega andmemahu probleem. Sõltuvalt sõnumite suurusest ja *bag*'i kestusest võivad minna *bag*'ide suurused gigabaitidesse.

2.1.3 Käsitsi testimine

Robotite simulaatoris testimiseks robotite programmeerimise aines kasutatakse hetkel Gazebo simulaatorit manuaalselt. Praegu peab jooksutama koodi käsitsi ja siis visuaalselt ning subjektiivselt hindama roboti eesmärgi täitmist. Kaitsmisel kulub palju aega, et sedasi kontrollida põhjalikult kõiki keskkondi ja simulatsiooni detailsustasemeid. Põhjalik ja metoodiline kõikide variatsioonide manuaalne läbivaatus on aeganõudev protsess, mida saab automatiseerida. Veel üheks oluliseks automatiseerimise eeliseks on tagatud objektiivsus. Tulemusi kontrollitakse arvutuslikult mitte kaitsmise vastuvõtja subjektiivse hinnangu alusel.

2.2 Olemasolev infrastruktuur

Lõputöö raames laiendati Tallinna Tehinkaülikoolis olemasolevaid automaattestimise võimalusi. See tähendas mitmete komponentide integreerimist omavahel ja uute lahendustega. Olemas olevatest süsteemidest oli vaja siduda:

- Arete testimiste haldamine;
- AI Lab tööde SLURM järjekord;
- AI Lab *Singularity* keskkond.

Olemas olev riistvara, mille peal need süsteemid töötavad:

- Testimise server
  - Arete testimise ootejärjekord
  - Praegused testimise keskkonnad
  - Selle serveri koormuse haldamine
- AI Lab serverite kobar
  - SLURM järjekord
  - Singularity konteinerid

3. Tehnoloogiad

Lõputöös kasutati erinevaid varasemalt teiste arendajate poolt loodud komponente neljal peamisel eesmärgil:

- testimistaotluse edastamiseks;
- testimiskeskkonna hoidmiseks;
- testide jooksutamiseks;
- testide tulemuste kokkukogumiseks.

Testimise taotlused tulevad läbi *Arete* süsteemi. Testimiskeskkonda hoitakse *Docker* tehnoloogia abil kujutise formaadis, millele lisatakse konkreetne testitav kood ja testid konteineri moodustamiseks. Testid tuvastatakse *ROS* raamistiku abil ning füüsika simuleerimiseks on keskkonnas *Gazebo* simulaator. Testide väljundit loetakse *XML* (Extensible Markup Language) failist. Toetatud on *GTest* ja *unittest* teekide formaadid. Saadud tulemus pakitakse *JSON* formaati ja edastatakse uuesti *Arete* süsteemi.

3.1 Arete

Arete on *Spring* raamistikku kasutav automaattestimise süsteem, mis sisaldab suhtlust testi tulemusi kuvavate süsteemidega. Näiteks tulemuste saatmist *Moodle* keskkonda läbi *Charon* süsteemi ja omab vastavalt koormusele tööde jagamise tehnoloogiat. *Charon* on ülesannete ja testide haldamise süsteem, mõeldud jagama informatsiooni *GitLab*, Arete ja *Moodle* vahel ning kuvama seda kasutajale. Tegu on keskse testimissüsteemi töid jagava komponendiga, mille ülesannete hulka kuuluvad:

- *GitLab* lehelt, *Charon*'ist või *Codera*'st tellimuste vastu võtmine;
- *Git* repositooriumitest testitavate koodide ja testide allalaadimine;
- testimiste tellimuste autentimine;
- *Docker Hub* süsteemist testri kujutise valimine;
- testimiste järjekorra hoidmine;
- testri koormuse piiramine;
- päringu alusel testimise tulemuste tagastamine;
- tööde agregeerimine ja kasutajaliideses statistika kuvamine.

Arete süsteemi laiendati *Service* tüüpi klassiga, mille nimi on *LoadBalancerMasterService*. *Service* märge tähistab, et tegu on äriloogikat sisaldava klassiga. Sisemiselt kasutatakse *HashMap* tüüpi konteinerit, et hoida seoseid *LoadBalancerClient* objektide ja nende poolt toetatud testimiskeskkondade vahel. Koormuse jaotajate omavaheliseks võrdluseks loodi liides, mis laiendab Java *Comparator* liidest *LoadBalancerClient* klassi jaoks.

3.2 Docker

Docker[2] on virtualiseerimistehnoloogia operatsioonisüsteemi tasandil. See võimaldab defineerida üldistatud keskkonna, mille saab salvestada ja seejärel kasutada konkreetse keskkonna instantsi loomiseks. Docker'i kasutamine tagab süsteemi ja keskkonna eraldatuse, mis võimaldab samal ajal jooksutada mitut sõltumatut testimist.

Docker'iga koos kasutatakse *Docker Compose* laiendust, mis võimaldab määrata konteinerite süsteemi. Antud süsteem koosneb individuaalsetest konteineritest ja need pannakse jooksma samal ajal ning võimaldab konfigureerida ka konteinerite vahelist suhtlust. Samuti määrab antud konfiguratsioon ka konteineri ja operatsioonisüsteemi vahelise failide jagamise.

3.3 Singularity

Vajadus toetada Singularity[3] konteinereid lõputöös tuleneb ülikooli piiratud arvutusvõimsusest. Füüsika simuleerimine kolmemõõtmelises ruumis on arvutuslikult ja mälu poolest ressursimahukas. Samuti kaasneb pilditöötlusega vajadus graafikakaardi järele. Nendele nõuetele vastav riistvara asub Tallinna Tehnikaülikoolis AI Lab serveris. Turvalisuse kaalutlustel ei toeta antud süsteem *Docker*'i kasutamist, aga on toetatud Singularity. Sellest tuleneb ka vajadus võimaldada testide jooksutamist Singularity konteineris.

Singularity on konteineriseerimistehnoloogia, mis on välja töötatud teadusarvutuse jaoks. Singularity kasutamise eelis võrreldes alternatiividega seisneb *Docker* kujutiste toes. Peamised erinevused seisnevad:

- piiratud kasutaja õigustes, kus Docker võimaldab juurkasutaja privileege, siis Singularity's on õigused sarnasemad tavakasutajale
- erinev välise keskkonnaga sidumise süsteem, Singularity võimaldab ühendada väliseid kaustu ilma lisa seadistamata vaid lugemisõigustega

3.4 Simple Linux Utility for Resource Management

SLURM (*Simple Linux Utility for Resource Management*) on ressursijagamise süsteem, mis võimaldab jagada töid mitme serveri vahel. See on mõeldud kasutamiseks *Linux* operatsioonisüsteemis ja on kasutusel Tallinna Tehnikaülikoolis AI Lab'is. Antud lõputöös arendati integratsioon Arete testimise tellimuste ja AI Lab SLURM tööde jooksutamise vahel. SLURM pakub asünkroonset tööde käivitamist määratletud parameetritega riistvara peal. See hoiab endas järjekorda tellitud töödest. Järjekorrast võetakse tellitud töid ning käivitatakse neid vastavalt nõutud riistvarale ja hetke koormusele.

Teine tähtis osa SLURM võimalustest on süsteemi monitoorimine. Seda võimaldatakse erinevatel tasanditel. Esiteks on võimalik saada informatsiooni partitsioonide ja sõlmede tasemel (Joonis 1). Teisena on võimalik saada teada konkreetse töö prioriteedi komponente (Joonis 2). Järgmisena on võimalik näha hetke tööde järjekorda prioriteetsuse järjekorras (Joonis 3).

```
autotester@ai-lab:~$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
common*      up 14-00:00:0      4    mix ai-lab-[01,04-05,07]
common*      up 14-00:00:0      1  alloc ai-lab-06
common*      up 14-00:00:0      1   idle ai-lab-02
```

Joonis 1. SLURM partitsioonide ja sõlmede monitoorimine.

```
autotester@ai-lab:~$ sprio
          JOBID PARTITION   PRIORITY       SITE        AGE  FAIRSHARE    JOBSIZE  PARTITION
           1266 common         11914          0       1000          0        915      10000
```

Joonis 2. SLURM prioriteedi monitoorimine.

```
autotester@ai-lab:~$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
              1440    common script.s   wadjed  R    3:18:08      1 ai-lab-05
              1412    common cnn_all.   ivuvar  R 1-07:08:25      1 ai-lab-01
              1429    common loop_fs4   alegue  R    7:15:06      1 ai-lab-07
              1428    common kd2.bash   alegue  R    7:16:45      1 ai-lab-06
              1427    common kd3.bash   alegue  R    7:17:57      1 ai-lab-06
              1426    common kd4.bash   alegue  R    7:26:05      1 ai-lab-04
              1425    common kd5.bash   alegue  R    7:44:59      1 ai-lab-04
              1424    common  kd.bash   alegue  R    7:46:10      1 ai-lab-01
```

Joonis 3. SLURM järjekorra monitoorimine.

3.5 Robot Operating System

ROS (*Robot Operating System*) on robootikatööstuses ja arenduses standardne raamistik komponentide vaheliseks suhtluseks. Aasta 2020 seisuga oli ROS-ile viidatud 7410-s teadustöös ja aasta jooksul oli ROS *wiki* lehte külastatud 2 miljonit korda [4]. Samuti

sisaldab see integratsiooni tööriistadega programmikoodi kompileerimiseks ning sõltuvuste lahendamiseks. Antud lõputöös kasutati nendest *catkin_make* programmi.

3.5.1 Roslaunch

ROS süsteem sisaldab *roslaunch* tööriista, mis on võimeline leidma ROS pakettides *launch* tüüpi faile. Antud failid on *XML* formaadis ja määravad ära, milliseid programme ja mis parameetritega korraga käivitada. Samuti võivad *launch* failid sisaldada viiteid teistele *launch* failidele.

Lõputöös kasutatakse *launch* faile, et defineerida testidega samal ajal jooksvad programmid nagu näiteks *Gazebo* simulaator ja tudengi testitav kood. *Launch* failides on peamiselt kolme tüüpi elemente:

- *include*, mis viitab teisele *launch* failile ja sisestatakse käivitamisel;
- *node*, mis sisaldab paketti ja programmi nime käivitamiseks;
- *param*, mis võimaldab parameetrite edastamist *rosnode* programmidele.

Launch failides, mis defineerivad teste on ka *test* tüüpi elemendid. Need elemendid määravad teste sisaldava programmi ning võimaldavad lisaparameetrite konfigureerimist nagu näiteks ajalimiit.

3.5.2 Rostest

Rostest on alamsüsteem ROS raamistikus, mis lihtsustab testide jooksutamist. See süsteem võimaldab kompileerimise faasis defineerida testide programmid. Antud programmid võivad olla viidad käivitatavatele failidele või *launch* konfiguratsioonidele. Teine kasulik komponent rostest süsteemis on testide leidmine. Kõiki kompileerimise käigus defineeritud teste on võimalik käivitada korraga, ilma nimesid või asukohti teadmata.

Esimene toetatud teekidest testide kirjutamiseks on *GTest*. See on *Google*'i poolt arendatava testimise teegiga, mis on mõeldud C++ keele jaoks. Teek sisaldab järgnevaid võimalusi:

- luua testi mall;
- võrrelda tulemust oodatava vastusega;
- jagada teste alamüksustesse.

Samuti toetab see *XML* formaadis tulemuste väljastamist, mida kasutab ka *rostest*. ROS süsteemist sõnumite saamiseks on antud testides vaja kasutada *roscpp* teeki.

Teine toetatud teek on *Unittest*. Tegu on *Python* keeles kasutatava testide kirjutamise teegiga. See võimaldab mitmeid *GTest*'iga sarnaseid funktsionaalsuseid: tulemuste valideerimine, testide gruppeerimine, *XML* failina tulemuste vormistamine. Tasub mainida, et kuigi *GTest* ja *Unittest* väljastavad mõlemad *XML* faili, siis kasutavad nad erinevaid vormistusi, mis erinevad elementide ja väljade nimede poolest kui ka elementide struktuuri osas.

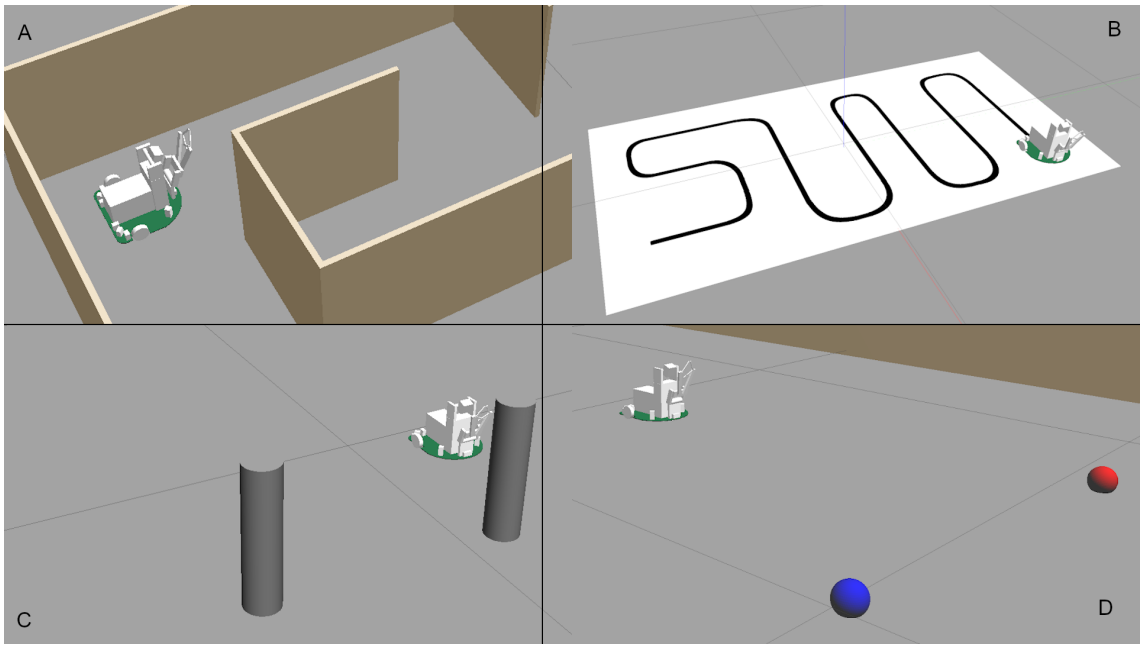
3.6 Gazebo

Gazebo on robotite füüsika simuleerimiseks kasutatav tarkvara. Antud tehnoloogia valiti, kuna see toetab integratsiooni ROS süsteemiga ja on kasutusel juba mitmes Tallinna Tehnikaülikooli õppeaines. Gazebo süsteem koosneb kahest peamisest komponendist: klient ja server. Server simuleerib füüsikat ja loeb konfiguratsiooni faile. Klient visualiseerib kasutajale nähtaval kujul simulatsiooni. Selles lõputöös on vajalik vaid serveri osa.

Selleks, et võimaldada suhtlust Gazebo süsteemide ja ROS süsteemide vahel kasutatakse *gazebo_ros* kimp (*package*). Gazebo simulatsioon pannakse jooksma koos teiste ROS sõlmedega. Selleks kasutatakse *launch* failides Gazebo süsteemide käivitamist. Käivitus failides saab konfigureerida simulatsiooni maailma, kasutades *world* faili. Roboteid saab simulatsioonis defineerida *URDF* failide abil.

3.6.1 Keskkonna defineerimine

Maailma, milles robotit simuleeritakse, saab defineerida kasutades *world* tüüpi faile. Tegu on *XML* formaadis definitsioonidega, milles saab määratleda valgust, füüsilisi objekte ja visuaalseid objekte. Füüsilisteks objektideks on näiteks seinad (Joonis 4 A), mis piiravad roboti liikumist, pallid (Joonis 4 D), mida robot pead tuvastama ning liigutama või postid (Joonis 4 C), mille robot peab üles leidma. Visuaalne objekt on näiteks joonejärgimise väljak (Joonis 4 B), mille peal robot tuvastab valgel taustal musta joont. Sisestatavad objektid võivad olla defineeritud läbi geomeetriliste kujundite või viitadena välistele mudelitele.



Joonis 4. A) Labürint Gazebo süsteemis B) Joonejärgimise rada Gazebo süsteemis C) Posti kujulised objektid Gazebo süsteemis D) Punane ja sinine pall Gazebo süsteemis.

4. Arhitektuur

Lõputööna valmis laiendus Arete süsteemile ja konteinerina vormistatud ROS lahenduste testimise keskkond. Lahenduse peamised komponendid asuvad järgnevates repositooriumites:

- ros-tester(TalTech GitLab) - ROS testimiskeskkonna definitsioon [5];
- Authentication service(TalTech GitLab) - LoadBalancerMasterService koormuse jagamiseks [6];
- robotics-gtest-extension(GitHub) - GTest teegile robootika testimise abi funktsioonide teek [7];

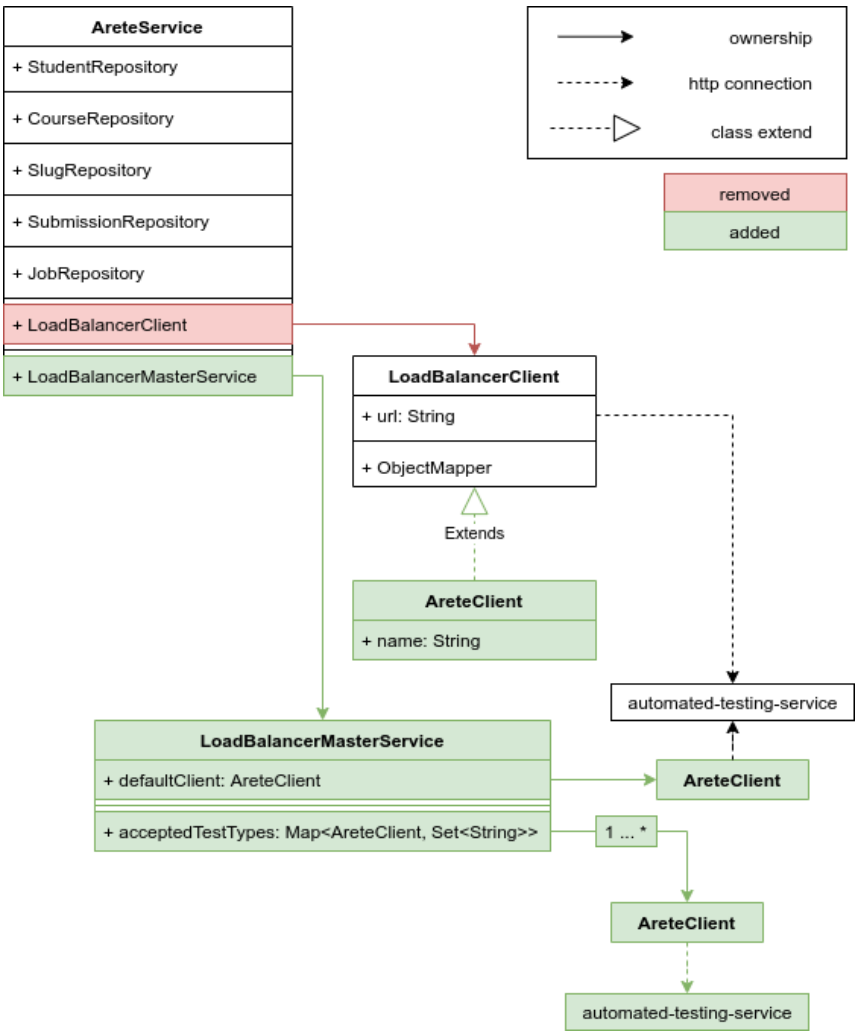
Automaattesitimissüsteemile Arete lisati tugi mitme tööjaotuse instantsi tugi. Tööjaotus instants eksisteeris varasemast ja oli vormistatud *Docker* konteinerina. Lahendus võimaldas vaid ühe masina sisest koormuse juhtimist. Antud lõputöö raames lisati funktsionaalsus toetamaks mitut konteinerit. See võimaldab jagada koormust juhtivaid sõlmi erinevate masinate vahel. Võimalus rakendada rohkem kui ühte masinat aitab rakendada suuremas koguses aruvutusressurssi paralleelselt testimiseks.

Selle näitlikustamiseks loodi töö raames ka ROS testimiskeskkond. ROS keskkond on realiseeritud nii *Docker* kui ka *Singularity* kujutisena. Uues arhitektuuris saabub testimise soov Arete süsteemi autentimise komponenti. Autentimise teenus valib vastavalt soovis defineeritud keskkonnale testimisseadme. Soov edastatakse seadmesse ja seal luuakse vastavalt sovile kujutise alusel keskkond ning selle sees jooksutatakse teste.

Peale testide jooksutamist tagastatakse tulemus Arete süsteemi. Edasi saadab Arete tulemuse korrektsetesse sihtkohtadesse. Võimalikud lõpp sihtpunktid on näiteks Moodle süsteem või meili aadress. Samuti saab edastada tulemuse *Charon* süsteemi, mis toimib kui vahelüli Moodle'sse testimise tulemuste edastamiseks ja õppejõududele nende haldamiseks ning ülevaate saamiseks. Peamine väljund fail on *output.json*

4.1 Arete autentimise teenus

Arete süsteem koosneb autentimis ja automaattestimise teenusest. Lõputöö osana laiendati autentimise teenust. Autentimise teenus autentiseerib tellimusi ja salvestab vahemällu Aretega seotud väärtuseid. Autentimise süsteemi algne arhitektuur sisaldas viita koormusejagajale, mis juhtis tööde jooksutamise järjekorda ühe masina siseselt (Joonis 5). Autentimise teenusele lisati komponent, mis jagab töid koormusejagajate vahel võimaldades rakendada rohkem kui ühte masinat.



Joonis 5. Autentimise teenuse arhitektuur.

Selle funktsionaalsuse lisamiseks laiendati eksisteerivat klienti nimesildiga, mis lihtsustab klientide üksteisest eristamist. Seejärel loodi ülemteenus, mis võtab sisse testimise soovid ja valib kliendi, millele see edastada. Selle teostamiseks loodi lõputöö raames *LoadBalancerMasterService* klass, mida tüüpi objekt on osa *AreteService* klassist. See toimib kui vastuvõtja testimise platvormidega seotud tegevustele.

LoadBalancerMasterService objekti üks peamiseid ülesandeid on testimiseks kliendi valimine. Valimise esimeseks kriteeriumiks on kliendi vastuvõetavate testide tüüp. Testimiste tüübid on vastavalt testimiskeskkondadele ehk näiteks: *Python*, *C* või ROS. See tähendab, et kitsendatud testrid toimivad *whitelist* põhimõttel. Erandiks on *default* klient, mis võtab vastu kõiki töid. Alles jäänud testritest valitakse sobivuse skoori alusel parim. Sobivuse leidmiseks kasutatakse loodud *LoadBalancerClientEvaluator* liidest implementeerivat klassi objetki. See liides nõuab, et klass implementeeriks *LoadBalancerClient* tüüpi objektide omavahelise võrdluse. Sobivuse arvutamiseks kasutatakse näiteks hetke CPU kasutust.

Kliendist edastatakse tellimus sihtmasina koormusejaotajale. Tellimuste edastamine võib toimuda sünkroonselt, mida kasutatakse näiteks kattis-tester puhul või asünkroonselt, mida kasutatakse ülejäänud testrite korral. See kasutab sisemist järjekorda ja maksimaalse koormuse piiranguid, et otsustada, millal käivitada töö. Sealt edasi luuakse testimise konteiner ja käivitub testimine. Peale töö lõppemist võetakse väljundid ja tagastatakse autentimise teenusesse.

4.2 ROS keskkond

ROS keskkond defineerib operatsioonisüsteemi tasandil keskkonna, kus jooksutatakse teste. Keskkond on määratletud *Dockerfile* formaadis. See sisaldab programmide installeerimist, keskkonna kaustade loomist ja keskkonna muutujate määratlemist. Selle kujutise baasil on loodud ka *Singularity* versioon ROS testimise keskkonnast. Kujutise kokkupanemise eest vastutab *GitLab*'i pidevvalmiduse protsess.

ROS testimise kujutise baasiks on võetud avalik ROS kujutis, mis sisaldab endas juba ROS raamistiku kompileerimise põhikomponente. See omakorda kasutab baasina avalikku *Ubuntu* kujutist. Seega sarnaneb kasutatav operatsioonisüsteemi keskkond *Ubuntu* keskkonnaga. Seejärel on keskkonda installeeritud *catkin-tools* komponendid, et lihtsustada *catkin* kompileerimistööriista kasutamist. Samuti lisatakse *git*, et võimaldada repositooriumite formaadis failide allalaadimist.

Testimise lihtsustamiseks installeeritakse keskkonda *robotics-gtest-extension* teek , mis valmis samuti selle lõputöö osana. Antud teegi eesmärk on pakkuda tööriistu *gtest* teegi abil testide loomiseks. Järgmise testimise tööriistana lisatakse keskkonda *Gazebo*. See võimaldab testides kasutada *Gazebo* simulatsioone, et simuleerida robotit ja maailma. Peale seda lisatakse keskkonda *X Virtual framebuffer*. See aitab tekitada virtuaalse ekraani, et toetada *Gazebo* süsteemides piltide genereerimist. Viimase sammuna lisatakse keskkonda testide jooksutamise ja tulemuse lugemise koodid.

Testimise keskkonna juhtimiseks loodi *ros_tester* pakett. Paketi esimesena käivituv osa on *ros_builder* selle ülesanne on kompileerida testid ja testitav kood. Samuti loob see *robot.py* failide alusel uued ROS paketid, et võimaldada ITI0201 lahenduste testimist ilma, et tudengid peaksid oma lahendusi vormistama ROS pakettideks. See kood kasutab *catkin_make* tööriista, et kompileerida kogu *catkin* töökeskkond.

Järgmise osana käivitub testide jooksutamine. Testide käivitamiseks kasutatakse *catkin_make run_tests* käsku, mis kasutab *catkin* süsteemi, et leida ja käivitada testide failid. See käivitab kõik pakettides testidena defineeritud failid. Testid genereerivad väljundina *XML* faile, mis kirjeldavad testide tulemusi. Need tulemuste failid otsitakse üles *ros_tester* koodi poolt. Failide sisude lugemiseks kasutatakse *xml.dom* teeki, mis teostab *XML* failidest Python objektide loomist.

Need väljundid on vaja muuta *XML* formaadist *Arete* süsteemile vastavaks *output.json* failiks. See eeldab *XML* objektidest õigete väljade ja elementide leidmist. Ülesande teeb keerulisemaks asjaolu, et erinevad testimise teegid struktureerivad *XML* faile erinevalt. Lõputöös arendati funktsionaalsus toetamaks *C++ GTest* teeki ja *Python unittest* teeki. Samuti kogutakse kokku */host/bags* kaustas olevad *bag* failid. Nendest failidest kogutakse kokku salvestatud piltide kanalid ja nendest koostatakse *AVI* tüüpi video failid. Koostatud *JSON* fail koos silumiseks vajalike failidega väljastatakse konteinerist.

4.3 Singularity keskkond

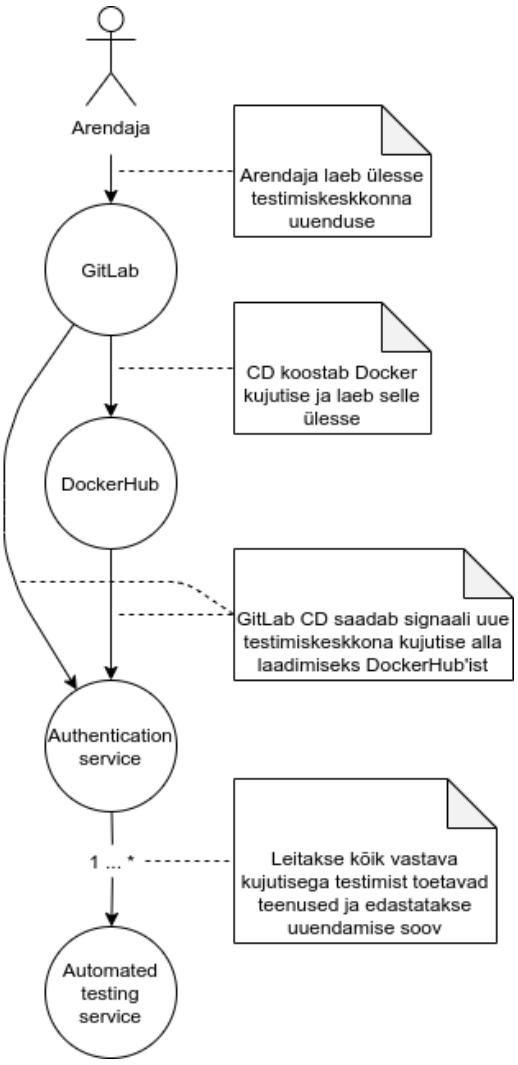
Selleks, et võimaldada testimist AI Lab keskkonnas, on vaja kujutise definitsiooni Singularity tehnoloogia abil. See tuleneb asjaolust, et turvakaalutlustel ei ole seal serverite kobaras toetatud *Docker* tehnoloogia. Kuna Singularity disaini toetab kujutise loomist ka Docker kujutise baasil, siis on aluseks võetud *Docker* ros-tester kujutis *automatedtestingservice* kogumikust[8].

Kujutise defineerimiseks Singularity süsteemis kasutatakse *def* laiendiga faili. Lisaks viidale baas kujutisele sisaldab see ka seotavate kaustade päritolu ja sihtpunkte. Erinevalt *Docker*'ist käsitakse Singularity'l luua ka kirjutamist võimaldav kiht konteinerisse, sest vastasel juhul on kaustad vaid lugemisõigustega. Selle definitsiooni alusel luuakse kujutis samuti osana pidevvalmidussüsteemist. Peale kujutise loomist laetakse see üles Singularity standardsesse kujutiste hoidlasse[9], kust saab neid kätte viidates *library*'le kui kujutise päritolu allikale.

5. Kasutamine

5.1 Testimiskeskkonna muutmine

Testimiskeskkond on defineeritud kui *Docker* kujutis. Sellega kaasnevad ka *Python* skriptid, mis haldavad testimiskeskkonna kasutamist. Selleks, et lisada keskkonda sõltuvusi või muuta käivitavaid faile muuta peab uuendama kogu kujutist. Selle protsessi kiirendamiseks ja lihtsustamiseks on ülesse seatud pidevvalmidussüsteem GitLab'is, mis koostab kujutise, laeb selle Docker Hub'i ja annab testimissüsteemile märku, et kohalikku kujutist peab uuendama (Joonis 6).



Joonis 6. Testimiskeskkonna uuendamine.

5.2 Testide defineerimine

Antud süsteemis toimivad testid peavad olema koostatud vastavalt *rostest* standarditele. Selle pärast peavad testid olema ROS kimbu (*package*) sees. Lisaks testidel võib samas kimbus olla ka teisi faile. See piirang on tingitud asjaolust, et testimiskeskkonnas otsitakse *rostest* tööriista abil teste. Kaustade struktuuri osas on kasutajal vaba voli, sest *catkin* süsteem tuvastab *build* operatsiooni käigus paketid *src* kaustas sõltumata kaustade struktuurist. Tavaliselt koondatakse testimisega seotud failid kimbu siseselt *test* nimelisse kausta.

Testide märkimine toimub *CMakeLists.txt* failis. Selleks saab kasutada käske *add_rostest* ja *add_rostest_gtest*, mis on vastavalt *Python* ning *unittest* ja C++ ning *GTest* jaoks. Samuti on vaja nende funktsioonide kasutamiseks lisada *rostest* teek *CMake* sõltuvuste hulka. Hea tava on testide kompileerimine ümbritseda ka kontrolliga *CATKIN_ENABLE_TESTING* parameetri kohta. See võimaldab kimpu kompileerida ka ilma teste käivitamata (Joonis 7).

```
find_package(rostest REQUIRED)

if (CATKIN_ENABLE_TESTING)
  # cpp tests
  add_rostest_gtest(cpp_test
    test/my_cpp_test.test
    test/my_cpp_test.cpp)
  target_link_libraries(cpp_test ${catkin_LIBRARIES})

  # py tests
  add_rostest(test/py_basic.test)
endif()
```

Joonis 7. *CMakeLists.txt* failis *rostest* teegi abil testide lisamine.

Testide defineerimiseks on vaja kasutada kahte tüüpi faile. Esiteks on vaja määratleda käivitatavad sõlmed. Selle jaoks tuleb kasutada *launch* formaadis faile, mille jaoks kasutatakse tavaliselt laiendit *test*. Tegelik faililaiend ei ole oluline, sest *CMakeLists* failis on määratletud otsitava faili täielik nimi. Teste sisaldav sõlm määratletakse *XML* failis *test* tüüpi elemendis. Teiseks on vaja faili, mis sisaldab endas sooritatavaid kontrolle. See võib olla C++ või *Python* keeles.

Lihtsam ja kiirem lahendus on testide kirjutamine *Python*'is. Esiteks on kirjutamine lihtsam selle pärast kuna keel on lihtsam. Teiseks ei vaja *Python* kompileerimist ega linkimist. Seega piisab *unittest* testimise teegi kasutamiseks sellele viitamisest testide failis. Samuti

sisaldab *Python* väga head tuge ROS funktsionaalsustele läbi *rospy* teegi. Hea tava on koondada teste kokku vastavalt teemadele. Selle jaoks saab luua klasse, mis laiendavad *TestSuite* klassi. Sinna sisse saab paigutada üksikuid teste. Testide kirjeldamine käib sama moodi läbi uute klasside loomise. Need uued klassid peavad laiendama *TestCase* klassi. Kombineerides neid komponente saab teste luua vaid loetud ridadega (Joonis 8).

```
#!/usr/bin/env python
import unittest
import rostest
import rospy

class TestBasic(unittest.TestCase):
  def runTest(self):
    pass

class SuiteTest(unittest.TestSuite):
  def __init__(self):
    super(SuiteTest, self).__init__()
    self.addTest(TestBasic())

if __name__ == '__main__':
  rostest.rosrun('tests', 'PY_Test', 'py_test.SuiteTest')
```

Joonis 8. Näidis *Python* abil testide defineerimisest.

Teine peamine ja rohkemate võimalustega testide defineerimise keel on C++. Selle keelega kaasneb lisa keerukus, sest *CMakeLists.txt* failis peab *link*'ima *catkin* teeke ja teisi, mida soovitakse kasutada. Testimise teekidest on soovitatav kasutada *Google Test* teeki, sest see on põhjalikult dokumenteeritud ja toetatud. See teek sisaldab mitmeid kasulikke võimalusi nagu näiteks ühise algseadistusfunktsiooni loomine. Testide defineerimiseks on antud teegis *macro* nimega TEST, millel on parameeter testide komplekti nime jaoks. Selleks, et kasutada ROS raamistiku võimalusi saab kasutada *roscpp* teeki. Kasutades neid võimalusi tuleb testid luua sõlm, mis käivitab testid (Joonis 9).

```
#include <gtest/gtest.h>
#include <ros/ros.h>

TEST(CPP_Test, testEmpty)
{
}

int main(int argc, char** argv)
{
  testing::InitGoogleTest(&argc, argv);
  ros::init(argc, argv, "cpp_test");

  ros::AsyncSpinner spinner(1);
  spinner.start();
  int ret = RUN_ALL_TESTS();
  spinner.stop();
  ros::shutdown();
  return ret;
}
```

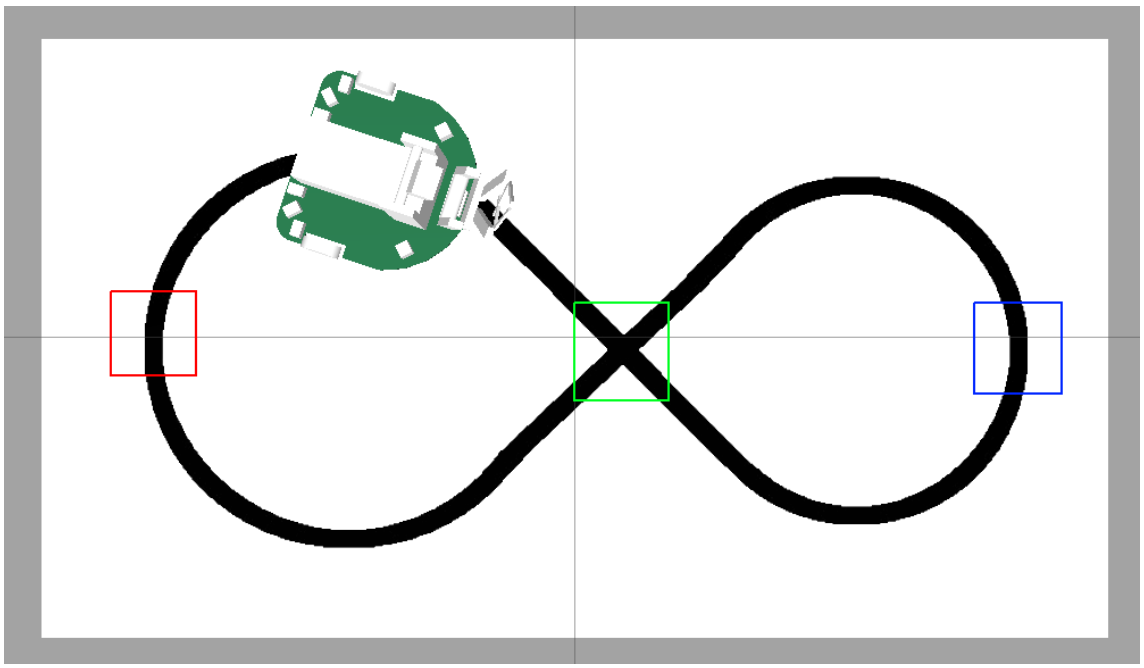
Joonis 9. Näidis C++ abil testide defineerimisest.

Selleks, et lisada robootika jaoks kasulikke funktsioone loodi lõputöö raames *robotics-gtest-extension* teek. See on implementeeritud kasutades C++ keelt ja mõeldud laiendama *Google Test* teegi funktsionaalsuseid. Tegu on *header only* teegiga ja seega piisab selle kasutamiseks sellele viitamisest ning seda ei pea eraldi kompileerima. Üheks kasulikuks lisaks antud teegis on funktsioon kontrollimaks, kas punkt asetseb hulknurgas. See aitab sooritada roboti peal kontrolle, kas see asub mingis piirkonnas või vastupidi ei asu. See toimib nii kumerate kui ka mittekumerate hulknurkadega. Korrektsuse tagamiseks sisaldab see teek ka üksusteste. Kasutamise lihtsustamiseks sisaldab see ka installeerimise konfiguratsiooni *CMake* tööriista jaoks.

5.3 Testide näited

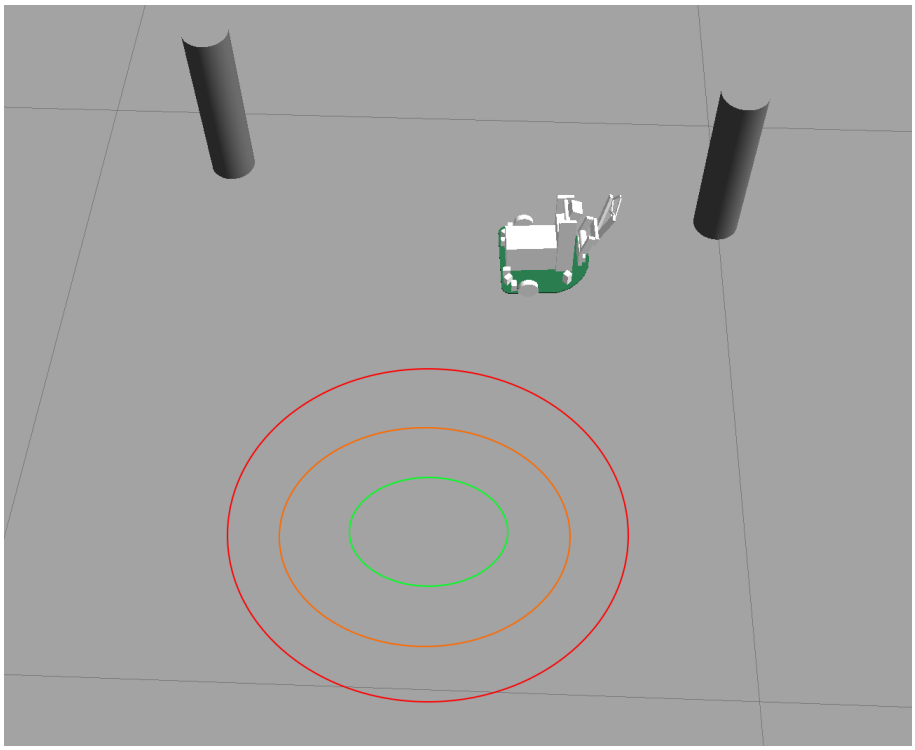
Näiteks saab testida roboti käitumist joonejärgimise ülesandes. Antud ülesandes peab robot sõitma mööda musta 8-kujulist joont sooritades ristmikul korrektse pöörde vastavalt sellele mitmes kord ristmikku ületatakse. Selleks tuleb esiteks defineerida maailmas raja kujutis ja selle peal robot (Joonis 10). Seejärel on vaja määratleda testid. Üks võimalikest kõrgema keerukusega loogika kontrollidest on läbitavate lõikude järjekorra kontrollimine. Sellega saab valideerida, et robot pöörab õiges järjekorras ja õiges suunas ristmikult. Luues testis kolm kontrollala, millesse sisenemise järjekorda peab kontrollima saab teada, mis

järjekorras robot silmuseid (Joonis 10 punane ja sinine) sõidab ja millal robot siseneb ristmikule (Joonis 10 roheline). Näiteks kui robot peab esimene kord pöörama vasakule, teine kord sõitma otse ja seejärel pöörama paremale ja sedasi korduvas tsüklis. Siis oleks õige alade läbimise järjekord: roheline, sinine, roheline, punane, roheline, sinine, roheline sinine.



Joonis 10. Joonejärgimise testpiirkondade näidis.

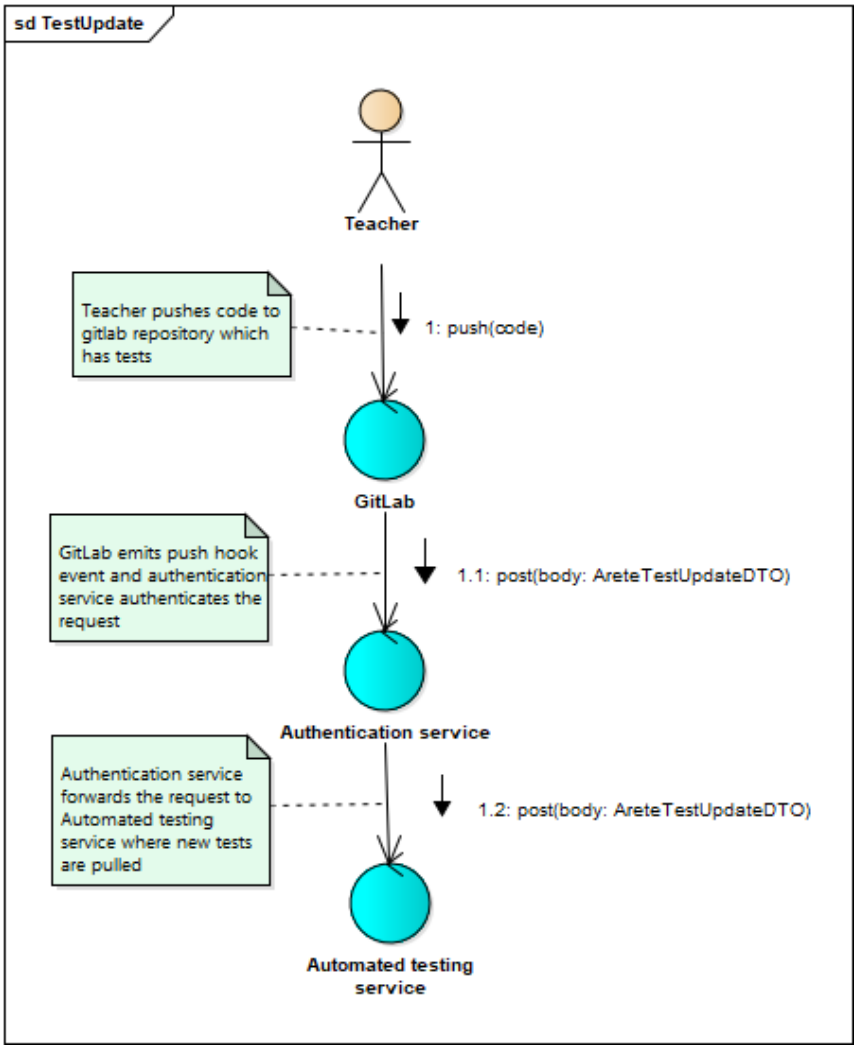
Üks robotite programmeerimise aines eksisteerivaid ülesandeid on objektide ja robotiga kolmnurga moodustamine. Antud ülesandes peab robot tuvastama staatiliste objektide asukohad ja paigutama ennast sedasi, et moodustuks võrdkülgne kolmnurk roboti ja kahe objekti vahel. Perfektse lahenduse saamiseks peaks robot olema täpselt õiges punktis, aga piirkondade abil on võimalik anda osalisi punkte osalise lahenduse eest. Näiteks saab anda osad punktid punasesse alasse jõudmise eest, veidi rohkem punkte oranži alasse jõudmise eest ja maksimaalsed punktid rohelisse alasse jõudmise eest (Joonis 11). Sedasi on võimalik luua lõputöös loodud vahenditega astmelise hindamisega test.



Joonis 11. Kolmnurga moodustamise ülesande võimalike testpiirkondade näidis.

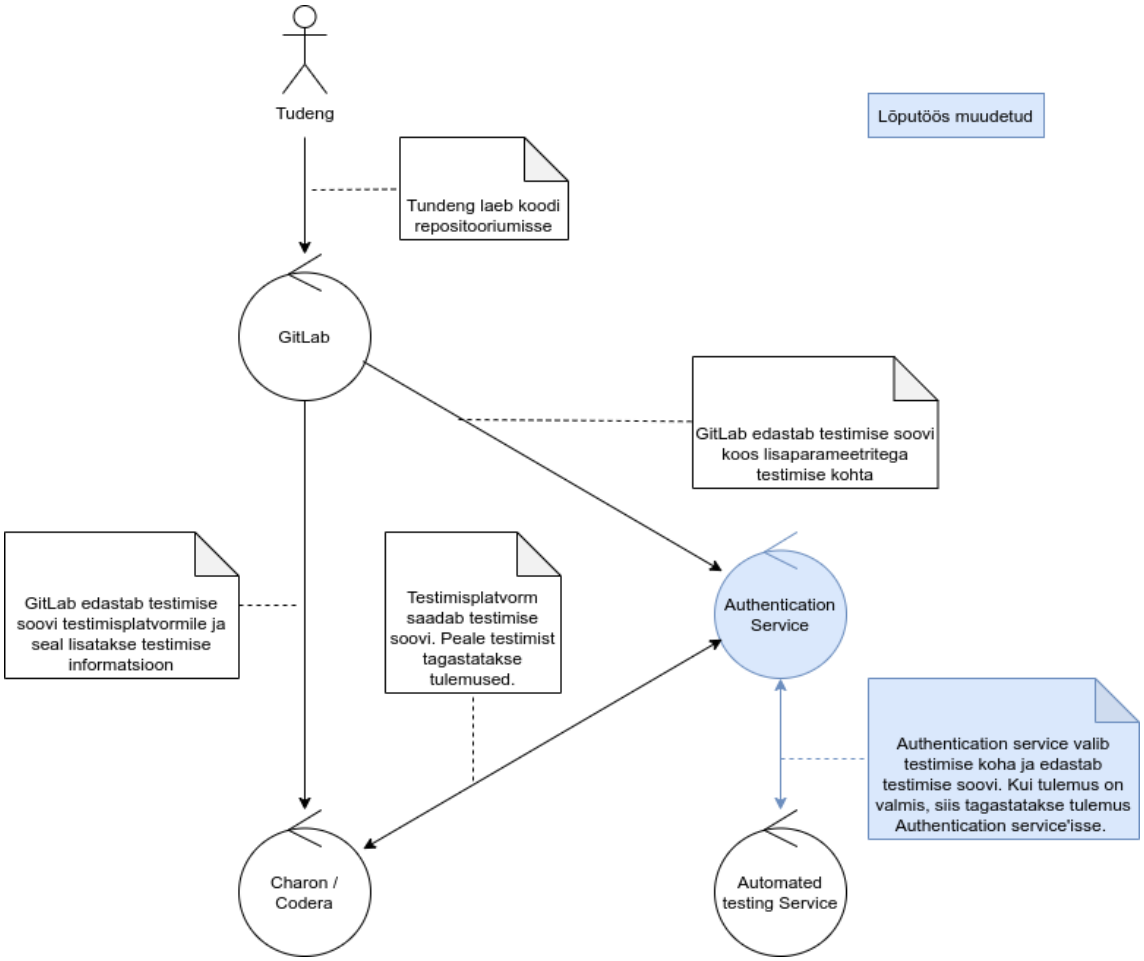
5.4 Testide jooksutamine

Testide uuendamine toimub vastavalt Arete süsteemi mudelile (Joonis 12). See tähendab, et protsessi käivitab õppejõud tehes muudatuse ja laadides selle testide repositooriumisse. Sealt edasi toimuvad muudatused automaatselt ja testid jõuavad testimise teenustesse, kus neid vastava tellimuse korral kasutatakse.



Joonis 12. Testide uuendamine [10].

Testimise käivitumise esimene samm on tudengi poolne koodi ülesse laadimine repositooriumisse, kus eksisteerib vastav GitLab käivitav päästik (Joonis 13). Sealt liigub testimise soov edasi läbi testimisplatvormi ja autentimisteenuse vastavalt Arete loogikale. Lõputöö raames täiendati autentimisteenuse loogikat ja üks ühele seose asemel valib autentimisteenus nüüd mitme võimaliku testimisplatvormi seast parima.



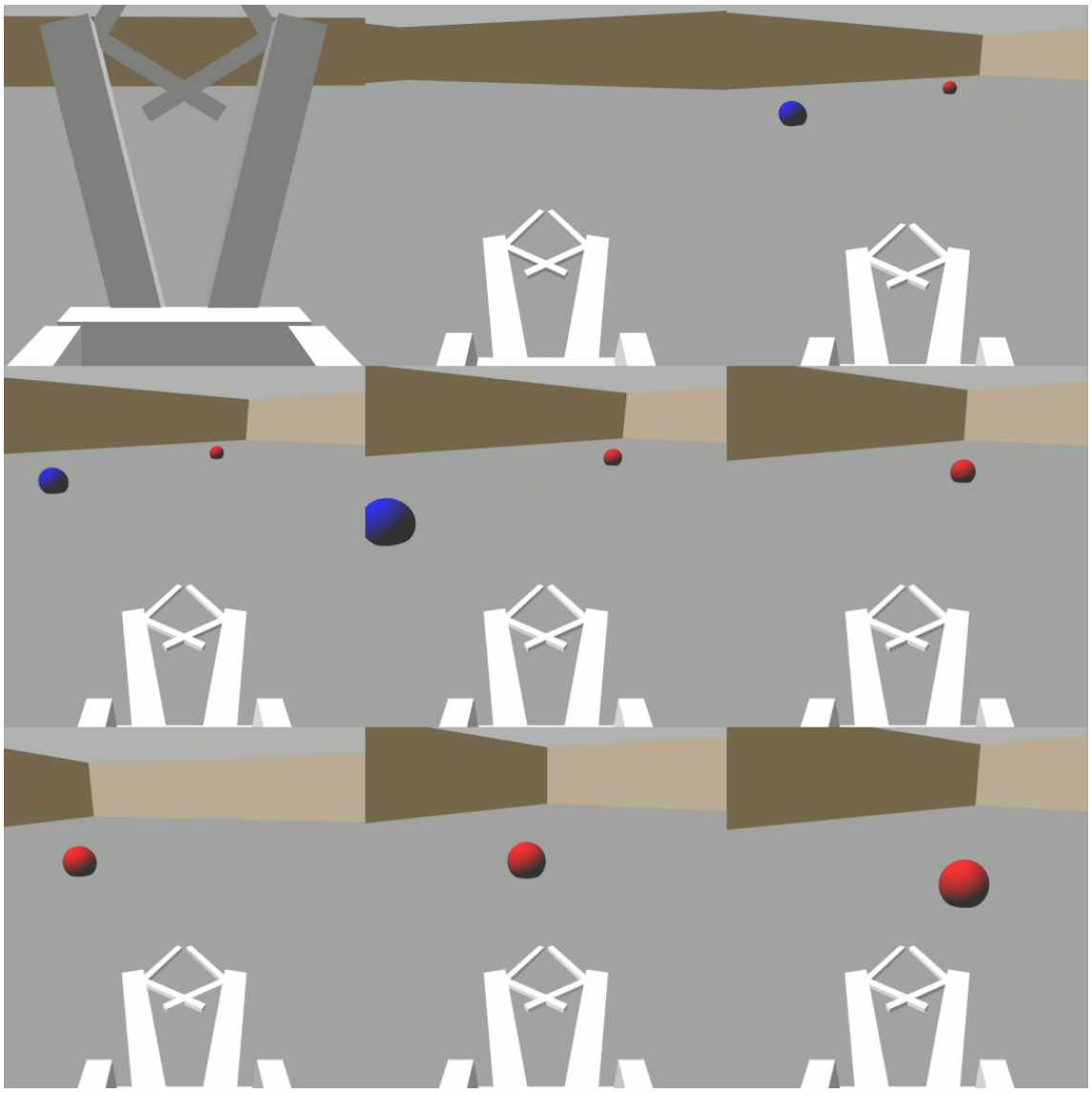
Joonis 13. Testide jooksutamine.

5.5 Tudengi töövoog

Tudengi poolt on esimene samm lahenduse loomine lokaalselt. See sisaldab endas ROS kimbu loomist lahenduse jaoks ja programmi lahenduse loomist selle sees. Olles veendunud ise lahenduse sobivuses laeb tudeng enda töö *GitLab* repositooriumisse. Selle peale asub server testima lahendust ja tudeng peab ootama tulemust. Tulemuse saabumist saab tudeng näha kahest kohast.

Esiteks saabub tulemus tudengile tema ülikooli meilile (Lisa 3). Sinna saabub ülevaade konkreetsete testide tulemustest ja kokku saadud punktidest. Meilile saadetakse lisaks punktidele ka kõik teised testimise poolt genereeritud väljund failid. Nende hulka kuuluvad:

- ROS bag failidest genereeritud videod roboti sooritusest (Joonis 14);
- testimise ja kompileerimise *stderr* (standard vigade) ning *stdout* (standard väljundi) voog.



Joonis 14. Kaadrid vahega 0,4 sekundit testrist tagastatud videost roboti kaamerast palli tuvastamisest ja selle suunas sõites.

Teiseks on näha punktide ja testide informatsiooni *Moodle* süsteemis (Lisa 2). See kuvatakse tudengile valitud aine antud ülesandepüstituse juures. Samuti on seal näha ka varasemate esituste tulemusi. *Moodle* vaates kuvatakse tudengile läbitud, vahele jäetud ja ebaõnnestunud testid. Vahele võidakse näitaks jätta teste kui õppejõud on seadnud mingi testi teistele testidele eelduseks ja eeldus ei ole läbi läinud. Näha on igaks testiks kulunud aeg ja antud testi kaal. Lõpuks on ka üldine arvestus, kus loetakse kokku igasse kategooriasse kuulunud testide arv.

6. Tulemuste analüüs

Antud lõputöö üks eesmärk oli säästa kasutajate aega. Aega säästetakse peamiselt automatiseerides korduvaid ülesandeid. Antud peatükis võrreldakse ja analüüsitakse erinevate kasutajate ajakulu manuaalse ja automatiseeritud süsteemi puhul. Peamisteks kasutajate rollideks on:

- õppejõud, kes defineerib testid ja seab nõuded tulemusele;
- tudeng, kes loob lahenduse ja soovib tagasisidet.

6.1 Õppejõudude ajavõit

Õppejõupoolne tulemuse kontrollimise protseduur on manuaalsel ja automatiseeritud kontrollimisel erinev. Peamised erinevused on korduvus ja keerukus. Manuaalselt peab sooritama lihtsamat kontrolli, aga igakord kui soovitakse kontrollida. Automaatselt peab määratlema testide parameetrid ja tingimused, mis eeldab hoolikat nõuete läbi mõtlemist, kuid seda peab tegema iga ülesande kohta vaid korra. Seega sõltub lõplik õppejõupoolne ajavõit kontrollitavate tudengite arvust.

Esimene samm ülesande määratlemiseks on selle keskkonna kirjeldamine. Automaatne ROS tester arendati arhitektuuriga, et see toetaks samasugust formaati nagu käsitsi testimine ja seega ei lisandu ega vähene sellele kulunud aeg. Kontrollides tulemust vaadatakse alguses peamisi ja lihtsamaid funktsionaalsuseid. Nende hulka kuulub näiteks liikumine ja lubatud piiridesse jäämine. Selliseid kriteeriume on lihtne kirjeldada testides, aga käsitsi kontrollimisel peab ootama roboti ja simulaatori järgi.

Kui minimaalsed nõuded on täidetud, siis järgmisena peab kontrollima keerulisemaid osasid ning eri osade koosmõju. Selliste testide nõuete määratlemine võib olla keerulisem, sest peab arvestama rohkemate asjadega. Samas võtab ka käsitsi kontrollimine rohkem aega, sest robot peab teostama mitmeid järjestikuseid tegevusi ning lahenduse rõhk on kvaliteedil mitte kiirusel. Seda tüüpi kontrolli puhul tasub automatiseerimine ära juba mõne tudengi puhul, sest on tõenäolisem, et on vaja korduvalt kontrollida enne kui lahendus töötab.

Sageli on vaja kontrollida, et lahendatud oleks ette antud probleem ja mitte konkreetne olukord. Selle jaoks on kasulik kontrollida tulemust erinevate algseisude puhul. Kontrollides manuaalselt peab õppejõud vaatama tulemust mitmes maailmas ning kordama oma tööd. Automaattestimise korral on uute testide loomine eelmiste baasil lihtsam, sest põhimõtted on juba paigas. Seega, on seda tüüpi kontrolli puhul käsitsi tegemine kogu ajakulu kordistav, aga automatiseerimisel vaid võrdlemisi kiire ja ühekordne tegevus.

Viimasena peab veenduma, et tulemus saavutati tänu heale lahendusele ja mitte tänu heale õnnele. Käsitsi kontrollides saab õppejõud kasutada oma intuitsiooni ning korduvat testimist. Esimesel juhul võib tekkida probleem objektiivsusega ning teisel puhul ajakulu mitmekordistub. Testide kirjutamisel on sellist probleemi kõige lihtsam lahendada, sest ei pea looma uusi teste, vaid käivitama olemasolevaid korduvalt ja agregeerima tulemust.

Näiteks oli aastal 2021 kevadsemestril aines Robotite Programmeerimine deklaratsioone 65. Samal ajal oli selles õppeaines 14 erinevat simulaatoriga kontrollitavat ülesannet. Sõltuvalt ülesandest oli saadaval erinev arv maailmaid vahemikus 1 kuni 7. Eeldades, et ühe ülesande visuaalseks kontrollimiseks kulub 15 minutit ja testide testide kirjutamiseks kulub 45 minutit saame arvutada hüpoteetilise säästetava aja (Tabel 1). Testide kirjutamise aja sisse ei arvestata maailma ja roboti defineerimise aega, sest see on ühine osa mõlema meetodi puhul.

Table 1. Ajakulu Robotite Programmeerimise õppeaine ülesannetele manuaalselt ja hüpoteetiline võit automatiseerimisel.

Ülesanne	**Maailmaid**	**Ajakulu manuaalselt (tundi)**	**Ajakulu automatiseerimisele (tundi)**	**Hüpoteetiline võit (tundi)**
Joonejärgija - Pronks	7	113.75	5.25	108.5
Joonejärgija - Hõbe	5	81.25	3.75	77.5
Joonejärgija - Kuld	4	65.00	3.00	62.0
Objektid - Pronks	5	81.25	3.75	77.5
Objektid - Hõbe	4	65.00	3.00	62.0
Objektid - Kuld	3	48.75	2.25	46.5
Objektid - Eliit	3	48.75	2.25	46.5
Pallid - Pronks	5	81.25	3.75	77.5
Pallid - Hõbe	3	48.75	2.25	46.5

Jätkub...

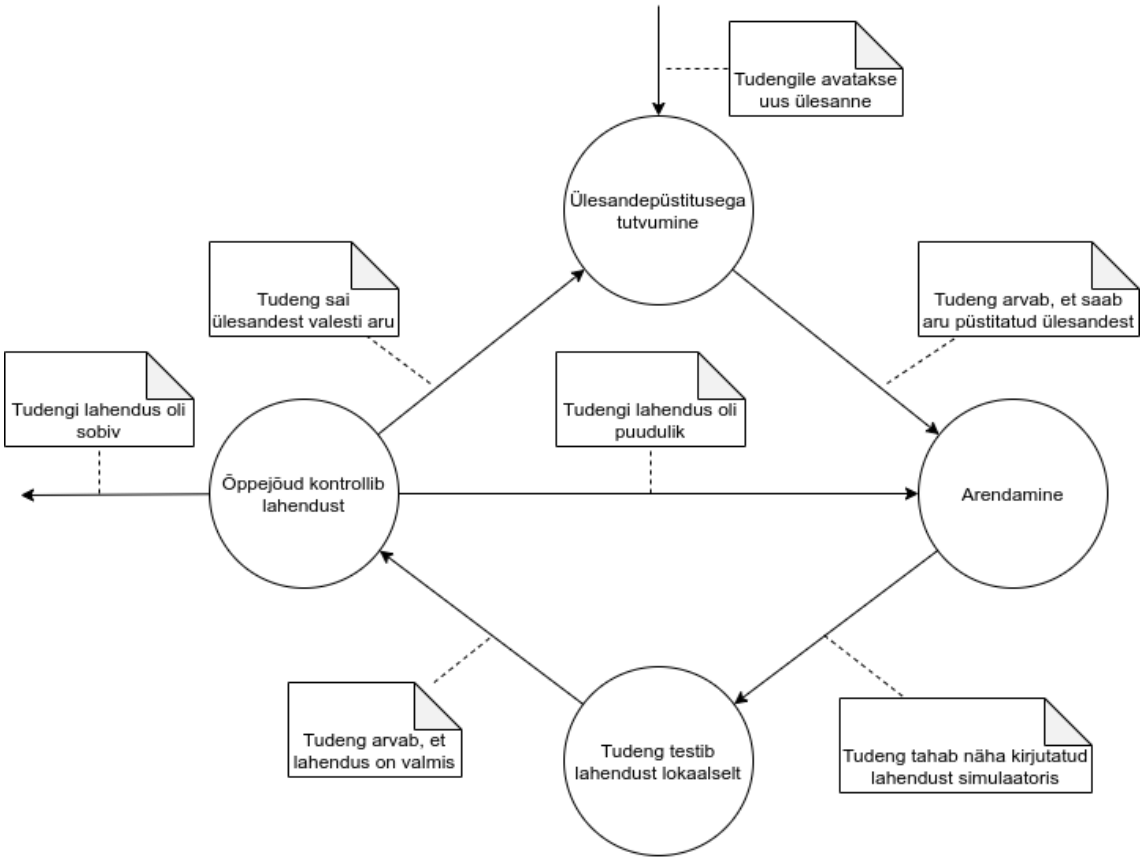
Table 1 – *Jätkub...*

Ülesanne	**Maailmaid**	**Ajakulu manuaalselt (tundi)**	**Ajakulu automatiseerimisele (tundi)**	**Hüpoteetiline võit (tundi)**
Pallid - Kuld	3	48.75	2.25	46.5
Pallid - Eliit	2	32.50	1.50	31.0
Labürint - Pronks	1	16.25	0.75	15.5
Labürint - Hõbe	4	65.00	3.00	62.0
Labürint - Kuld	4	65.00	3.00	62.0
Labürint - Eliit	4	65.00	3.00	62.0
Kokku	57	926.25	42.75	883.5

Saamaks ülevaadet robotite toimimise kontrollimiseks kuluvast ajast, sooritati küsitlus Robotite Programmeerimise (ITI0201) aine abiõppejõudude seas. Üks nende ülesannetest on tudengite lahenduste kontrollimine ja antud tööriist mõjutaks otseselt nende tööprotsessi. Abiõppejõududel paluti hinnata ülevaatuse protsessi, mille alusel koostati eelnevalt käsitletud tabel.

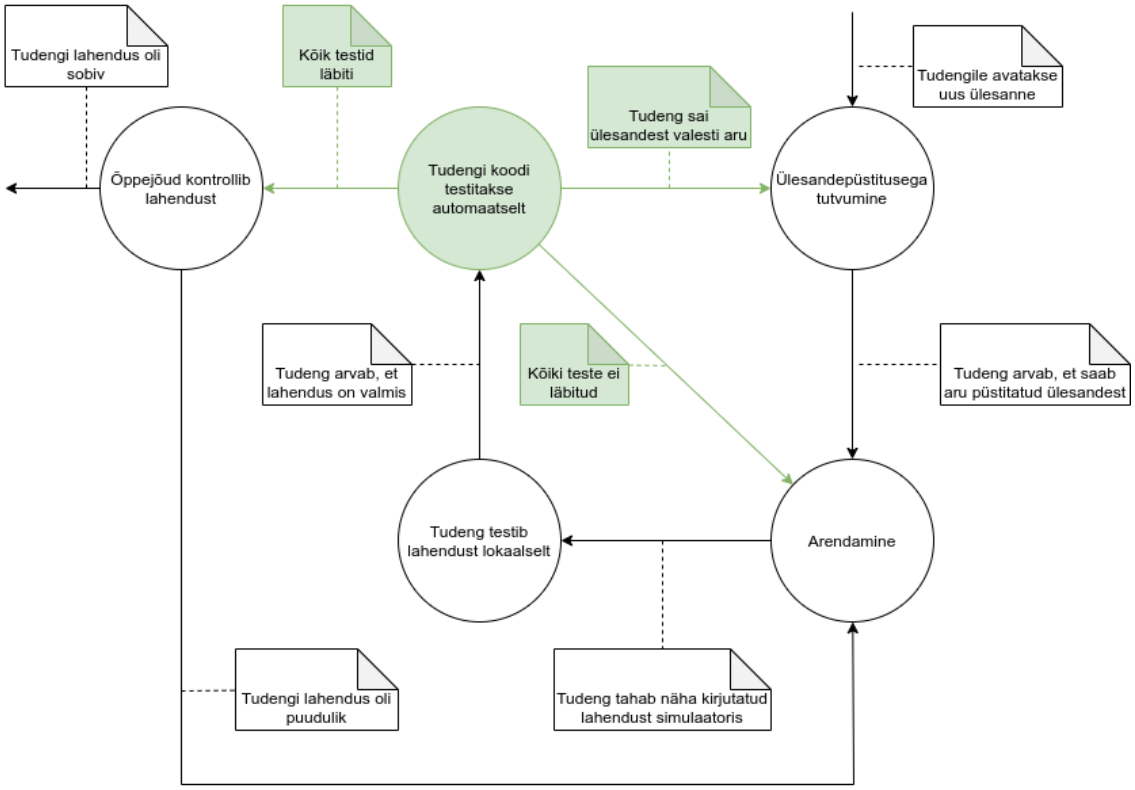
6.2 Eelised tudengile

Tudengite ehk lahendusi loovate kasutajate jaoks on peamised kasutegurid uued võimalused, mis kaasnevad automaattestimisega. Need lisavad mugavust ja suurendavad tarkvara arendamise efektiivsust. Mugavus tuleneb konkreetses tagasisides, mida on võimalik saada mõne minutiga peale lahenduse esitamist. Kui peab ootama praktikumi, et saada teada, kas lahendus vastab peamistele nõuetele on arendustsükkel aeglasem ja tagasisidet saab maksimaalselt paar korda nädalas. Sedasi saab parandada lihtsaid probleeme kiiremini (Joonis 15).



Joonis 15. Vana arendustsükkel.

Teine tähtis kasu automaattestidest on objektiivne ja konkreetne tagasiside ülesandest aru saamisele. Kui tudeng kahtleb, kas ta lahendab probleemi õigesti, siis testide läbimine või mitte läbimine annab selge vastuse. Samuti on head testide nimed, mis võivad anda vihjeid ülesande osadeks jagamiseks. Saades aru, millised on alamülesanded, mida kontrollitakse, on tudengil parem tulla ideede peale, kuidas lahendada kogu ülesannet ja kuidas struktureerida lahendust (Joonis 16).



Joonis 16. Uus arendustsükkel.

7. Kokkuvõte

Töö esmane eesmärk oli luua testimise keskkond ROS lahendustele, mis oleks pakendatud operatsioonisüsteemist sõltumatusse kujutisse. Teine eesmärk oli integreerida see olemasolevatesse süsteemidesse, et võimaldada selle kasutamist Tallinna Tehnikaülikooli kursustel.

Esimene eesmärk täideti luues *Docker* kujutise definitsioon koos teste kompileerivate, jooksutavate ja analüüsivate skriptidega. Samuti arendati *GTest* teegi jaoks lisa, mis lihtsustab testide kirjutamist. Testkeskonna definitsiooni lisati levinumad sõltuvused ROS lahendustes nagu näiteks *Gazebo* simulaator. Kujutis on avalikult saadaval *Docker Hub*'is leheküljel [1] ja selle automaatne uuendamine on osa projekti pidevvalmidussüsteemist.

Teise eesmärgi täitmiseks laiendati *Arete* projekti, lisades sinna toe mitme masina kasutamiseks, mis oli vajalik suurema võimsusega ressursside rakendamiseks füüsika simuleerimiseks ja video genereerimiseks. Selle jaoks loodi ka tugi AI Lab serverites tööde jooksutamiseks ning *Docker* kujutise baasil *Singularity* kujutise loomiseks.

[1] `https://hub.docker.com/r/automatedtestingservice/ros-tester`

Kasutatud kirjandus

[1] Gabriel Staples. *rosbag*. 2020. URL: http://wiki.ros.org/rosbag.

[2] *Docker*. 2021. URL: https://www.docker.com.

[3] *Singularity*. 2021. URL: https://sylabs.io/singularity/.

[4] Katherine Scott Tully Foote. *ROS Community Metrics Report*. 2020. URL: http://download.ros.org/downloads/metrics/metrics-report-2020-07.pdf.

[5] *ros-tester*. 2021. URL: https://gitlab.cs.ttu.ee/ained/codetest/ros-tester.

[6] *Authentication service*. 2021. URL: https://gitlab.cs.ttu.ee/testing/authentication-service.

[7] *robotics-gtest-extension*. 2021. URL: https://github.com/TimoLoomets/robotics-gtest-extension.

[8] *Dockerhub automatedtestingservice/ros-tester*. 2021. URL: https://hub.docker.com/r/automatedtestingservice/ros-tester.

[9] *Sylabs.io Cloud Library*. 2021. URL: https://cloud.sylabs.io/library.

[10] Enrico Vompa Ago Luberg. *Arete - automated testing service*. 2020. URL: https://ained.pages.taltech.ee/it-doc/arete/index.html#flows.

Lisad

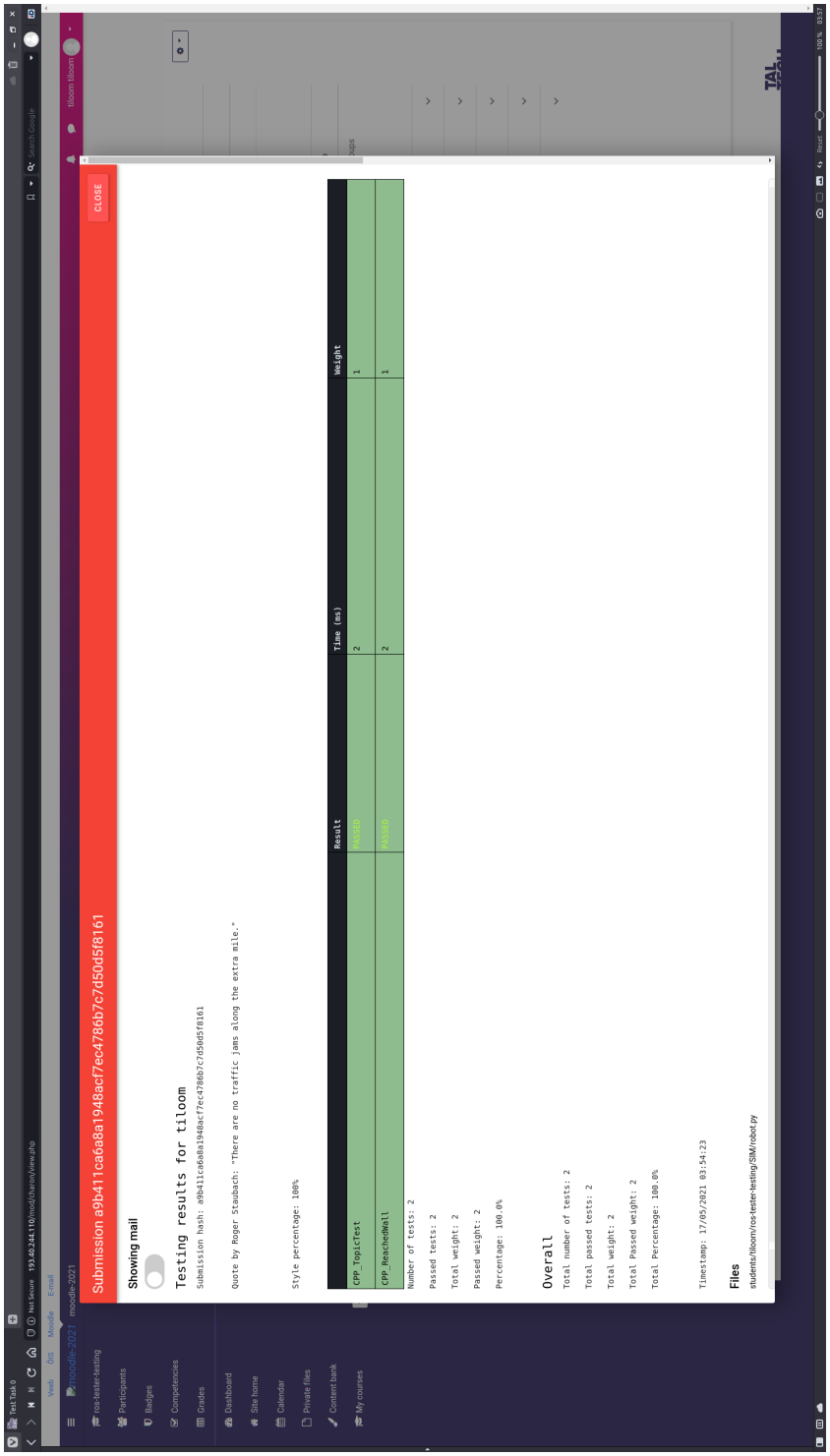
Lisa 1 - Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, Timo Loomets

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "ROS-il põhinevate projektide automaattestimissüsteemi arendus ja integratsioon Arete testimissüsteemiga", mille juhendaja on Gert Kanter
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadust ning muudest õigusaktidest tulenevaid õigusi.

25.05.2021

Lisa 2 - Tudengi vaade esitatud tööst *Moodle* keskkonnas



Lisa 3 - Tudengi vaade esitatud töö tagasiside meilist