TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutiteaduse instituut

Lauri Roomere 163177IAPM

Isejuhtiva auto tarkvara mudelipõhine integratsioonitestimine Autoware näitel

Magistritöö

Juhendaja: Juhan-Peep Ernits
Doktorikraad

Tallinn 2018

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: Lauri Roomere
14.05.2018

Annotatsioon

Käesoleva magistritöö eesmärgiks on mudelipõhise integratsioonitestimise rakendamine Autoware tarkvaraga loodava süsteemi näitel. Mudelipõhiste testide loomiseks kasutati raamvara NModel ja programmeerimiskeeli F# ja C#. Töös loodi mudelprogramm ja töövoog mille abil on võimalik viia läbi mudelipõhiseid integratsiooni teste Autoware'iga loodavatel süsteemidel auto juhtimisega seotud komponentide korrektse käitumise testimiseks.

Töö käigus on loodud nõue stsenaariumi kujul, mis on tuletatud testimise käigus loodud mudelist, kasutades käitumiskeskse arenduse metoodikast tulenevat *given-when-then* malli. Magistritöös on rakendatud ka koodikatvuse mõõtmist kasutades vahendeid gcov ja gcovr'i. Magistritöö käigus leiti viga F# programmeerimiskeelt toetavast NModel'i versioonist, kuid loodud mudelprogrammiga testimisel ei leitud vigu Autoware'iga loodavast süsteemist. Mudelipõhine test suutis sihilikult loodud vead tuvastada ja korrektsed vastused anda.

Lõputöö on kirjutatud Eesti keeles ning sisaldab teksti 50 leheküljel, 4 peatükki, 23 joonist ja 7 tabelit.

Abstract

The goal of this thesis is to apply model-based testing technique to Autoware autonomous driving software as a case study. The model-based tests were made by using a tool called NModel. The version of NModel that was used in thesis supported F# and C# programming languages.

With this tool a model and adapter where built to test the Autoware components that deal with moving the car. The purpose of the test was to check if the car can travel along a certain path by given waypoints within certain time. For checking the car's movement correctness, a radius was set (3.2 meters) for the waypoint where the car had to travel. The radius and many other parameters can be changed in the model and adapter.

For testing the movement related components, requirements where made with using given-when-then template from BDD (behaviour driven development). Later the requirement or as it is called in BDD scenario was tested in two environments and two modes that where offered by Autoware for configuring pure pursuit algorithm. The testing results turned out as expected. Most of the tests failed, because of a problem that might have been related to adapter or configuration of the pure pursuit algorithm in Autoware. The problem occurred while using the "Waypoint" mode for pure pursuit. The car tended to cut very sharp corners. To solve that problem in the future from the adapter side a solution was also proposed in this thesis. As a result, it was found that the created model programm can be also used to check how well the given configuration for pure pursuit works. During the creation of tests two bugs were found in the used NModel version, that where not show stoppers in this case.

In the thesis an example of code coverage measuring was also shown for measuring the nodes in Autoware that were written with c++ programming language. As an example, for the case study, the pure_pursuit node was measured. Gcov and gcovr were used to measure and display the code coverage results of the measured node.

As a result, potential workflow for model-based integration testing was presented and documented with Autoware case study that could be used in other Autoware related projects to conduct model-based testing.

The thesis is in Estonian and contains 50 pages of text, 4 chapters and 23 figures, 7 tables.

Sisukord

Sissejuhatus 8
1 Taust ja kirjanduse ülevaade 10
1.1 Metoodika 10
1.2 Mudelipõhine testimine 12
1.3 NModel 14
1.4 ROS, Gazebo ja Autoware 14
1.5 RosBridgeClient ja ROS# 15
1.6 RViz 16
1.7 Mobiilsete robotite ja autonoomsete sõidukite testimise alased uurimustööd 16
2 Eksperimendi keskkond ja nõuded testitavale süsteemile 19
2.1 Keskkond 19
2.2 ROS# ja RosBridgeClient'i installeerimine 20
2.3 Testitavad komponendid 21
2.4 Nõuded 22
2.5 Näide mudelprogrammi loomisest kasutades „turtlesim" programmi ja ROS 23
3 Autoware'iga loodava tarkvara mudelipõhine testimine 29
3.1 Iseliikuva auto juhtimise mudel 29
3.2 Testi adapter 32
3.3 Testide käivitamine 36
3.4 Stsenaariumi nr 1 testimine 40
3.5 Koodikatvuse mõõtmine 42
4 Tulemused 46
4.1 Testimise tulemuste analüüs 46
4.2 Hinnang mudelipõhise testimise rakendamisele 47
Kokkuvõte 49
Kasutatud kirjandus 51
Lisa 1 – Autoware'i süsteemiga auto juhtimiseks kasutavate sõlmede ning kanalite graaf 53
Lisa 2 – Koodikatvuse raport pure pursuit sõlmest 54
Lisa 3 – Faili pure_pursuit.cpp koodikatvuse näide 55
Lisa 4 – Kilpkonna argumendifaili näide 56
Lisa 5 – Loodud mudelprogrammid, logifailid ja lähtekood 57

Jooniste loetelu

Joonis 1. V-diagramm mis näitab traditsioonilise tarkvaraarenduse projekti tegevusi ja protsesside järjestust [6]. 13
Joonis 2. V-diagramm mis näitab mudelipõhise testimise ja analüüsimise võimalusi [6]. 13
Joonis 3 . Autoware'i platvormi struktuur [12]. 15
Joonis 4 . Testimisel kasutatav gazebo maailm, TTÜ Iseauto mudel ja CAT sõiduki mudel [20]. 20
Joonis 5. Näide rosbridge'iga kanali kuulamise protokollist [21]. 21
Joonis 6. Pure pursuit algoritmi geomeetria [23]. 22
Joonis 7. Olekugraaf kilpkonna näitel 25
Joonis 8. Auto liikumist kirjeldava mudeli olekugraaf. 30
Joonis 9. Auto raadiuses olemise valideerimine. 34
Joonis 10. Auto raadiusest väljas olemise valideerimine. 34
Joonis 11. Gazebo maailmas auto teekonnale ette seatud takistus. 35
Joonis 12. Testi tulemus simulatsiooni keskkonda lisatud takistuse korral. 35
Joonis 13. RosBridgeClient'i käivitamise näide. 38
Joonis 14. Auto liikuma panemise näide. 38
Joonis 15. Testide käivitamine vahendi CT abil. 39
Joonis 16. Argumendifaili näide. 40
Joonis 17. Näide pure pursuit algoritmi konfiguratsioonist stsenaariumi testimisel. 41
Joonis 18. Autoware'i stabiilses harus läbiviidud testi tulemuste statistika. 41
Joonis 19. TTÜ iseauto projektis läbiviidud testi tulemuste statistika. 41
Joonis 20. Waypoint režiimi testi tulemuste statistika. 42
Joonis 21. GCC valikute ja silumis võtmega kompileerimine caktin_make_debug faili abil. 43
Joonis 22. Pure_pursuit sõlme käivitamine. 44
Joonis 23. Koodikatvuse kuvamine gcovr'i abil. 45

Tabelite loetelu

Tabel 1. Disainiteaduse metoodika rakendamist kirjeldav tabel [5]. 10
Tabel 2. Autoware'i poolt soovitatud ja test masina spetsifikatsioonid. 19
Tabel 3. Käsud vajalike komponentide käivitamiseks kilpkonna näites 28
Tabel 4. Teekonnapunktide parameetrid. 31
Tabel 5. Toimingute üleminekutingimusi kirjeldav tabel 32
Tabel 6. Autoware ja Gazebo sõnumi tüübid ning väljad. 33
Tabel 7. Auto liigutamiseks vajalike sõlmede ja andmete kirjeldus. 36

Sissejuhatus

Isesõitev auto peaks suutma iseseisvalt läbida teatud teekonda nii, et see oleks võimalikult ohutu autos reisijatele ning kaasliiklejatele. Et tagada isesõitva auto ohutust ja autole esitatud vastavat käitumist talle antud operatsioonide korral on vaja isesõitvale autole luua nõuded ning antud nõuetele vastavust testida.

Auto reaalses keskkonnas testimine võib olla väga kulukas, eriti kui tarkvaras esineb vigu. Vead võivad kahjustada autot ennast, reisijaid ning kaasliiklejaid või keskkonda kus auto liikleb.

Üheks võimaluseks kuidas testimist muuta ohutumaks, avastada vigu juba enne auto testimist reaalsusele sarnanevas keskkonnas ning kärpida auto arendamisega seotud kulusid, oleks testida antud auto tarkvara simulaatoris kus on võimalik erinevaid testilugusid läbi mängida ning antud teste automatiseerida.

Simuleeritud keskkond on ohutu isesõitvale autole, kolmandatele osapooltele ja keskkonnale. Simuleeritud keskkonnas on võimalik läbi mängida erinevaid stsenaariume ja jälgida kuidas auto etteantud olukordades käitub.

Testide automatiseerimisega on võimalik tõsta tarkvara testimise efektiivsust, koodikatvust ning tõhusust (taaskasutatavad testid) seda ka funktsionaalseid teste automatiseerides, mille puhul testitakse simuleeritud keskkonda teatud sisendeid andes, kas süsteem saavutab oodatud väljundid.

Kahjuks on antud valdkonnas kättesaadaval vähe materjale ning uurimuste tulemusi, mis puudutavad autonoomsete sõidukite testimist. Puuduvad ka fikseeritud standardid, mis dikteeriksid kuidas autonoomsete sõidukite tarkvara luua ja testida, et tagada ohutust ning tõsta tarkvara loomise kvaliteeti [1]. Üldiselt on kõigi inimelule ohtu seadvate süsteemide tarkvara arendus rangelt reglementeeritud, et tagada ohutust (nt lennunduses). Mudelipõhine testimine võiks lihtsustada tarkvara arenduse kooskõlla viimist tulevikus loodava standardiga.

Antud magistritöö eesmärk on rakendada mudelipõhist testimist TTÜ isesõitva auto loomisel kasutatava Autoware'i [2] tarkvaraga loodava süsteemi integratsiooni testide läbiviimiseks, et automatiseerida *musta kasti* (ingl.k black box) testimist ning testide

loomist Autoware'i tarkvarale, mille tulemusena on võimalik avastada varajasemas arendus faasis vigu tarkvaras, vähendada arendamisele ning testimisele/testide loomisele kuluvat aega ja arendusega seotud kulusid [3].

Lisaks on eesmärgiks mõõta testide koodikatvust, et selgitada välja kui palju koodi antud mudelipõhise testimise tulemusena kasutati ning milliseid koodijuppe käitati. Selle abil on hiljem võimalik analüüsida koodikatvuse tulemusi vaadates, kas kõik vajalikud koodiosad käivitati ning kas süsteem käitus nõuetele vastavalt või mitte.

Kuna autonoomsete sõidukite tarkvara peab arvestama väga paljude muutujatega ja töötama pidevalt muutuvas keskkonnas, rakendatakse antud magistritöös käigupealt mudelipõhist testimist testitava süsteemi peal [4].

Magistritöö tulemusena on võimalik Autoware'i tarkvaral mudelipõhiseid integratsiooni teste läbi viia ning koodikatvust mõõta. Praktilise tulemusena on loodud süsteemi kirjeldavad mudelid ning adapterid, mille abil mudelipõhiseid testimisi testitavale süsteemile simulaatoris läbi viia ja koodikatvuse ning vigade kohta arendajatele infot edastada. Magistritöö eesmärgiks ei ole kogu süsteemi katmine integratsiooni testimise ning koodikatvusega, vaid näite (vähemalt ühe kriitiliselt tähtsaks peetava süsteemi osa kohta) loomine, mille põhjal hiljem oleks võimalik seda ülejäänud süsteemi osadele võimalikult lihtsalt edasi rakendada magistritöö käigus loodud töö baasil.

Magistritöös koosneb neljast peatükist, millest esimeses peatükis antakse lühike ülevaade probleemi taustast ja isesõitvate autode ning robotite mudelipõhist testimist puudutavatest teadus artiklitest. Lisaks antakse ülevaade töös kasutatud metoodikast ja vahenditest.

Teises peatükis kirjeldatakse kuidas on üles seatud keskkond milles teste käivitataks, millised nõuded on testitavale süsteemile esitatud ja millised komponendid on kaasatud testimisse. Peatüki lõpus on loodud näidis mudelprogramm mudelipõhise testimise tutvustamiseks ja meelde tuletamiseks.

Kolmandas peatükis käsitletakse Autoware süsteemi jaoks mudelipõhiste testide loomist, testide käivitamist ja koodikatvuse mõõtmist. Samuti viiakse läbi nõuetest tulenevate stsenaariumite testid ja kirjeldatakse saadud tulemusi.

Neljandas peatükis tehakse kokkuvõte läbiviidud testidest ja analüüsitakse saadud tulemusi, et anda hinnang antud testimisviisile.

1 Taust ja kirjanduse ülevaade

Autonoomsete sõidukite ja mobiilsete robotite tarkvara muutub arvutite arvutusvõimsuse kasvades keerukamaks. Sellest tulenevalt muutub keerukamaks autonoomsete sõidukite testimine. Keerukust autonoomsete sõidukite testimise osas lisab juurde ka fakt, et erinevad autonoomsed sõidukid liiklevad ning suhtlevad inimestega aina rohkem samas keskkonnas. Sellest tuleneb ka vajadus testida, et see suhtlus toimuks mõlemale osapoolele võimalikult ohutult.

Kasutades mudelipõhist testimist ja simulatsiooni keskkonda saab luua testi stsenaariume, väiksemate kulude ning riskidega. Lisaks ei ohusta antud testimise viis reaalset keskkonda.

Kindlasti ei asenda simulatsioonikeskkonnas testimine täielikult reaalsetes tingimustes testimist, kuid võimaldab tarkvaras/riistvaras olevaid vigu varem tuvastada ja ohutumalt erinevaid testi stsenaariume eelnevalt läbi mängida.

Magistritöö raames rakendatakse mudelipõhist integratsiooni testimist simulatsioonikeskkonnas TTÜ isesõitvas autos kasutatava tarkvara komponentide integratsioonitestimiseks. Isesõitva auto loomisel kasutatakse robootilist operatsiooni süsteemi ROS ja tarkvara Autoware.

1.1 Metoodika

Magistritöös järgitakse *disainiteaduse metoodikat* [5] (ingl.k design science research) töö struktuuri loomisel ja probleemi lahendamisel (vt Tabel 1).

Tabel 1. Disainiteaduse metoodika rakendamist kirjeldav tabel [5].

Disainiteaduse tegevused	**Tegevuse kirjeldus**	**Vajalikud teadmised**
Probleemi püstitus ja motivatsioon (ingl.k Problem	**Mis on probleem?** Puudub lahendus/tööriistade komplekt, mille abil viia läbi integratsiooni teste Autoware'i tarkvaraga loodavatel süsteemidel simulatsioonikeskkonnas, et muuta arendajate tööd efektiivsemaks,	Ülevaade teemat puudutavast kirjandusest ja tehtud töödest. Arusaam sellest, kuidas testitav süsteem töötab ja

identification and motivation)	integratsiooniteste reprodutseeritavaks ja isesõitvate autode arendust ning testimist vähem kulukamaks.	milliseid võimalusi see hetkel pakub testimiseks.
Lahendus käigu esitamine (ingl.k Define the objectives of a solution)	**Kuidas probleemi lahendada?** Kasutada mudelipõhist testimist, et luua reprodutseeritavad, automatiseeritavad testilood ja stsenaariumid, mille abil oleks võimalik Autoware'i tarkvaraga loodavaid süsteeme testida. Tagastada võimalikult palju infot vigade kohta arendajatele, et vigu oleks lihtsam tarkvarast eemaldada nende avastamisel. Mudelid luuakse kasutades raamvara NModel. Testitava süsteemiga suhtlemiseks kasutame ROSBridgeClient'it.	Teadmised mudelipõhisest testimisest, Autoware'i tarkvarast, vahevarast ROS ja simulatsiooni tarkvarast Gazebo. Lisaks teadmised sellest, kuidas testitava süsteemi jaoks adapterit luua, mis oleks võimeline suhtlema testitava süsteemiga.
Disain ja arendus	**Tehise loomine probleemi lahenduseks.** Mudelipõhise testimis tööriistade ning vahendite komplekt, mille abil testida Autoware'i ja ROS'iga loodavaid autonoomseid sõidukeid. Mudel mida saab kasutada erinevate Gazebo maailmadega ning millele on võimalik anda sisendina .csv formaadis fail, mis sisaldab teekonnapunkte, mille auto läbima peab. Vahendi gcov kasutamine koodikatvuse mõõtmiseks ja arendajale tagasiside andmiseks.	Teadmised järgnevatest tööriistadest ja vahenditest: NModel, C#, F#, RosBridgeClient'i,ROS'i gcov ja gcovr. Teadmised sellest milliste sõlmedega suhelda et autot liigutada.
Demonstratsioon	**Tehise relevantsuse tõestamine ühe või enama probleemi lahedamise näitel.** Testida vähemalt ühte Autoware'iga loodavat komponenti, mis on kriitilise tähtsusega ning seotud TTÜ isesõitva auto projektiga.	

Valideerimine ja hindamine	**Kui hästi antud lahendus töötab?** Analüüsitakse mudelipõhisel testimisel leitud vigu, tulemusi ja nende õigsust et anda hinnang loodud mudelprogrammile.	Efektiivsuse mõõtmise meetrikate defineerimine.

1.2 Mudelipõhine testimine

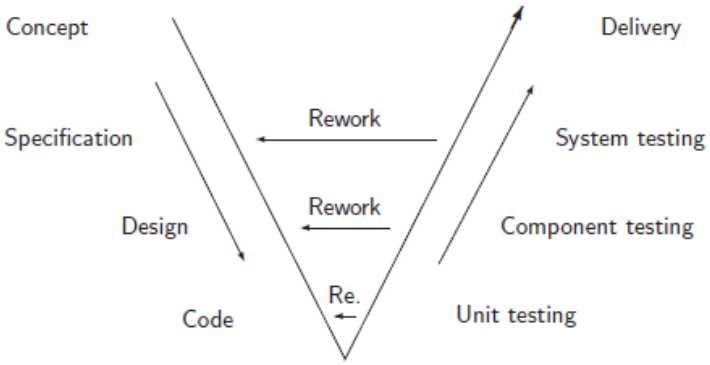
Mudelipõhine testimine on testimine, kus mudeli abil kirjeldatakse, kuidas testitav süsteem peaks käituma. Mudeli abil genereeritakse testlood ja mudelit saab kasutada testioraaklitena, mille abil kontrollida kas *testitav süsteem* (ingl. IUT/Implementation Under Test või SUT/System Under Test) läbib testid edukalt [6]. Mudel on lihtsalt öeldes tarkvara käitumise kirjeldus, kus käitumist väljendatakse tarkvara lubatud sisendite, toimingute, üleminekutingimuste ja väljundite abil [4].

Mudelipõhise testide käivitamine testitava süsteemi vastu jaguneb kaheks.

- Offline-testimiseks, kus testlood genereeritakse eelnevalt valmis ja seejärel käivitatakse. Lisaks võimaldab antud viis genereeritud teste soovikorral manuaalselt läbi viia kui need on genereeritud inimloetavana [4].

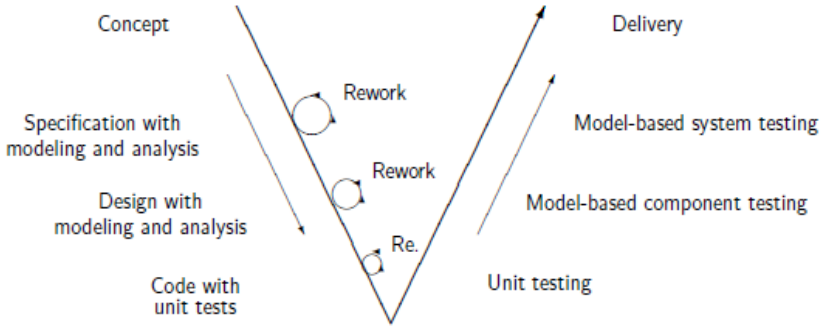
- *Käigupealt testimiseks* (ingl.k online testing), kus genereeritaks ja käivitatakse testid kohe . Mõlema testimise metoodika puhul luuakse mudeli baasil lõplik automaat, mille läbimisel luuakse testi stsenaariumid. Offline-testimise puhul salvestatakse testi stsenaariumi faili ja hiljem käivitatakse. Käigupealt testimise puhul käivitatakse antud stsenaarium käigupealt [6].

Mudelprogrammide abil on võimalik valideerida ja testida arendusprojekte juba varajasemas arendusfaasis. Selle illustreerimiseks kasutatakse tihti V-mudeli diagrammi (vt Joonis 1), mille abil näidatakse kuidas hilisemas faasis leitud vead nõuavad rohkem ressursi, et parandusi sisse viia. V diagramm näitab, kuidas iga arendusfaasi tegevusele (diagrammis vasakul pool) vastab seda toetav testimis meetod.



Joonis 1. V-diagramm mis näitab traditsioonilise tarkvaraarenduse projekti tegevusi ja protsesside järjestust [6].

Probleem traditsiooniliste arendus tegevuste puhul, nagu joonisel (vt Joonis 1) välja toodud on see, et kõige varajasemas faasis valminud projekti osasid (nt spetsifikatsiooni ja süsteemi disaini) testitakse viimastena [6].



Joonis 2. V-diagramm mis näitab mudelipõhise testimise ja analüüsimise võimalusi [6].

Mudelipõhise testimise ja analüüsimise abil on võimalik juba varajasemas arendusfaasi etappides teste läbi viia (vt Joonis 2). See ei tähenda, et alati tuleks kõiki süsteemi osasid ja komponente testida, vaid mudelipõhise testimise osa võib suunata kõige kriitilisemate, uudsemate või keerukamate osade testimiseks [6].

Mudelite loomiseks ja mudelitest testide genereerimiseks kasutan NModelit ja Autoware sõlmede koodikatvuse mõõtmiseks gcov'i, sest Autoware'i sõlmed on kirjutatud programmeerimiskeeles C++. Simulatsiooni keskkonnana kus teste läbi viia kasutan Gazebo't [7], [8].

Piirangud seab testimisvahendite valimisele robootilise operatsioonisüsteemi ROS [9] (ingl.k Robotic Operating System) kasutamine, mille peale TTÜ isesõitva auto tarkvara luuakse. Seega tuleb antud töö käigus lähtuda tööriistade valimisel sellest, et tööriistad oleksid ühilduvad ROS'i ja Autoware'iga.

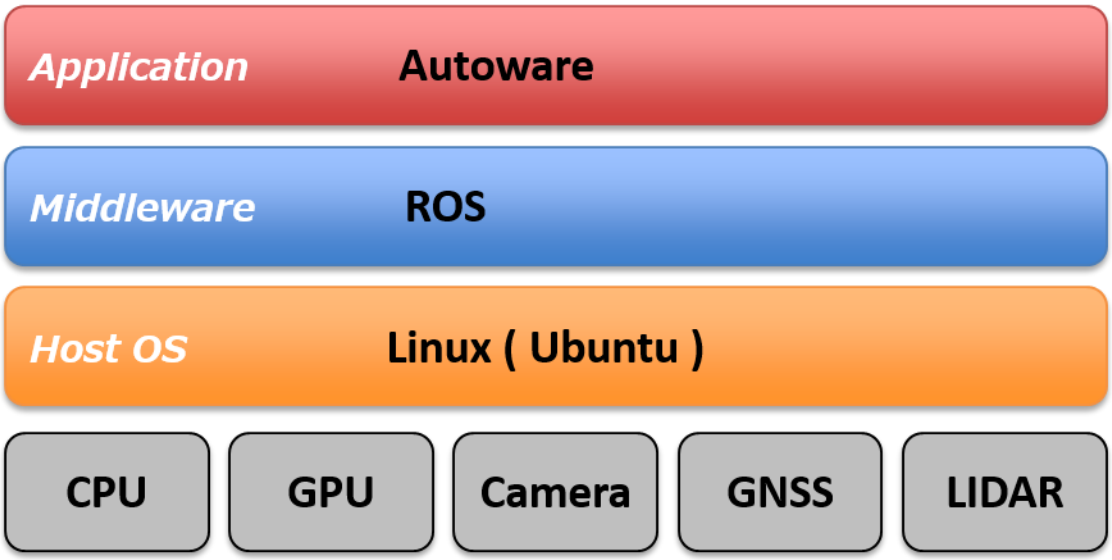
1.3 NModel

Mudelipõhise testimise ja analüüsimise raamistik mudelprogrammide jaoks [10]. Magistritöö raames kasutatakse haru NModel'ist, mis toetab lisaks programmeerimis keelele C# ka F#'i. F# võimaldab hõlpsasti genereerida testide jaoks vajalikke test andmeid, kasutades F# andmestruktuure, andmetüüpe ja funktsioone [11]. Täpsemalt kirjeldab NModel'i kasutamist ja kasutamise võimalusi oma magistritöös Rivo Roo [4].

Mudelipõhise testimise jaoks on loodud mitmeid programme, NModel sai peamiselt valitud seetõttu, et see toetab käigupealt testimist, on vabavaraline ja võimaldab mudel programme luua kõrgetaseme keelt C# ning funktsionaalset programmeerimis keelt F# kasutades [11]. Magistritöös kasutatakse NModel'i mugandatud versiooni, mis toetab ka mudeli loomist keeles F#. Käigupealt testimine on oluline sest keskkond, milles ka isesõitvad autod liiguvad on pidevas muutumises. Käigupealt testimisel on võimalik pidevalt simulatsioonikeskkonnas muutusi esile kutsuda, erinevaid mudelprogramme paralleelselt jooksutades.

1.4 ROS, Gazebo ja Autoware

ROS on vabavaraline tarkvara mis koosneb erinevatest teekidest ja tööriistadest mille abil on võimalik robootilist tarkvara luua. Hetkel on kõige uuemaks ROS'i versiooniks ROS Lunar Loggerhead. TTÜ isesõitva auto puhul kasutatakse ROS Kinetic Kame versiooni, sest ROS Lunar Loggerhead ei toeta veel täielikult Autoware'i. Kuigi ROS kätkeb oma nimes OS'i ei ole tegu operatsioonisüsteemiga vaid vahevaraga mis jookseb UNIX'il baseeruvatel operatsiooni süsteemidel [12] (vt Joonis 3).



Joonis 3 . Autoware'i platvormi struktuur [12].

Autoware on vabavaraline tarkvara autonoomsete sõidukite jaoks, mis on ehitatud ROS'i peale. Autoware annab vajalikud vahendid 3D kaartide genereerimiseks, lokaliseerimiseks, objekti tuvastuseks ja autonoomsete sõidukite juhtimiseks [12].

Gazebo võimaldab simuleerida roboteid ja nende sensorite rakendusi kolmemõõtmelises keskkonnas [7]. Gazebo kasutab klient/server arhitektuuri, tänu millele on võimalik panna läbi RosBridgeClient'i ning Autoware'i suhtlema Gazebo ja mudelprogramm, mida *testitava süsteemi* (ingl.k System Under Test/SUT) vastu käivitatakse. Autoware ja Gazebo kasutavad vahemaa mõõtühikuna meetreid ning nurkade jaoks radiaane.

1.5 RosBridgeClient ja ROS#

ROS# on vabavaraline tarkvara, mis koosneb teekidest ja tööriistadest millega saab C# programmeerimiskeele abil ROS'iga suhelda, kasutades RosBridgeClient'i. RosBridge Client abil on võimalik läbi JSON'i rakendusliidese kasutada ROS'i funktsionaalsusi välistel programmidel (nt saata mudelipõhise testimise käigus sõnumeid adapterist robotile). Suhtlus toimub läbi erinevate ROS'is defineeritud *kanalite* (ingl.k topic), *tellijate* (ingl.k subscribe) ja *kuulutajate* (ingl.k publish) kaudu. Suhtluse käigus kasutatakse alusena *ROS'i standardseid sõnumi tüüpe* (ingl.k Ros standard message types) [13]. Magistritöö käigus tuleb lisada uusi sõnumitüüpe RosBridgeClient'i, et oleks võimalik suhelda vajaminevate kanalitega kuna RosBridgeClient'is on defineeritud vaid ROS'i standardsed sõnumitüübid.

1.6 RViz

RViz on vahevara ROS 3D visualiseerimise vahend, mille abil saab visualiseerida sensorite andmeid ja olekuid vahevarast ROS. Magistritöös kasutame RViz'i teekonnapunktide ja teekonna läbimise visualiseerimiseks. Samuti on antud vahend abiks loodud mudeli vigade leidmisel ning silumisel.

1.7 Mobiilsete robotite ja autonoomsete sõidukite testimise alased uurimustööd

Mudelipõhisest testimisest leidub mitmeid artikleid kuid vähesed neist on otseselt seotud autonoomsete sõidukite mudelipõhise testimisega. Enamasti kasutati mobiilsete robotite ja autonoomsete sõidukite testimist puudutavates artiklites vahevara ROS.

Ernits, Halling, Kanter ja Vain keskendusid oma artiklis [3] sellele, kuidas tõsta integratsiooni testimise võimekust mobiilsete robotite testimisel, mis kasutavad roboti juhtimisega seotud kõrgema taseme *ROS'i pakke* (ingl.k ROS package). Artiklis viidi läbi eksperiment, mille käigus modelleeriti ja testiti STRANDS [14] projektis loodud navigatsiooni ja lokaliseerimise komponente. Pythoni moodulite koodikatvuse mõõtmiseks kasutati tööriista Coverage.py [15] [14]. Eksperimendis kasutati käigupealt mudelipõhist testimist, süsteemile esitatud nõuete testimiseks ja modelleerimiseks. Eksperimendis loodi ka stsenaarium, kus kasutati mudelit, mis modelleeris inimest, keda on võimalik simulatsioonikeskkonnas liigutada. Antud stsenaariumi testimisel ilmnes aga viga, mis võis olla seotud nii koodi kui simulatsiooni keskkonna konfigureerimisega. Eksperimendi tulemusena jõuti järeldusele, et topoloogiliste kaartidest automaatselt mudelite genereerimise ja olekute jadana testi stsenaariumite kirjeldamisega on võimalik tõsta koodikatvust. Artiklis kasutati mudelite loomiseks, valideerimiseks ja verifitseerimiseks tööriista Uppaal-Tron. Testide käivitamiseks kasutati Uppaal Tron'i ja dTron'i.

Takaya, Asai, Kroumov ja Smarandache artiklis [7] uuriti, kas simulatsiooni keskkonna põhjal loodud tarkvara saab rakendada reaalsetele mobiilsetele robotitele, jõuti järelduseni et roboti käitumine simulatsioonikeskkonnas ei erinenud drastiliselt reaalses keskkonnas roboti käitumisest ja tarkvara, mida kasutati simulatsioonis saab rakendada reaalse roboti peal. Eksperimendi läbiviimisel kasutati tarkvarasid ROS ja Gazebo [8].

Uurimustöö eksperimendi tulemusena märgiti ära, et siiski protsesside vaheliste mehhanismide kommunikatsiooni ja arvutamise kiirused erinesid, ning selle jaoks tuli modifitseerida *navigatsiooni teekide* (ingl.k navigation stack) *sõlmede* (ingl.k node) parameetreid.

Krejčí ja Novák kirjeldasid oma artiklis [16] metoodikat, mille abil oleks võimalik prioritiseerida mudelipõhiseid teste automaatselt autonoomsete sõidukite baasil. Prioriteedi määramisel hinnati kui tõenäoliselt uuritavas funktsionaalsuses esineb viga, iga muutusega funktsionaalsuse tuli uuesti uuendada vea indikaatorite väärtuseid, mille abil *tugivektorklassifitseerija* (ingl.k Support Vector Machine) algoritmiga loodi mudel, millega hinnati kui tõenäoliselt test läbi kukub. Lisaks kasutati subjektiivset hinnangut testija poolt kahele meetrikale, kus tuli anda hinnang auto mugavusi ja ohutust puudutavale funktsionaalsusele. Saadud mudeli tulemusi ja testimise kogemusest tulenevaid hinnanguid kasutati, et määrata perseptron neuronite abil testidele prioriteedid skaalal 1-10. Mida kõrgem väärtus seda kõrgem prioriteet. Antud metoodika rakendamisel peaks väidetavalt vähenema testilugude loojate ja läbiviijate nõutavavad teadmised testitavast süsteemist.

Laval, Fabresse ja Bouraqadi artiklis [17] arutleti selle üle, et robotite testid peaksid olema taaskasutatavad ja korratavad. Autorid viisid läbi eksperimendi ning tutvustasid metodoloogiat, mille eesmärk oli et testid peaksid olema võimalikult automatiseeritud, taaskasutatavad ja korratavad. Eksperiment viidi läbi kasutades kahte robotit, mis kasutasid ROS'i . Eksperimendi tulemusena näidati, et mobiilsete robotite jõudluse hindamiseks on testide taaskasutatavuse aspekt oluline.

Abdelgawad, McLeod, Andrews ja Xiao artiklis [18] presenteeriti juhtumianalüüsi, kus mudelipõhist testimist rakendati RAMP'i (ingl.k Real-Time Adaptive Motion Planning) süsteemi kasutavale robotile integratsiooni ja *süsteemi tasandi* (ingl.k system-level) komponentide jõudluse testimisel. RAMP on liikumis trajektoori planeerimise raamvara, mille abil on võimalik paralleelselt planeerida ja läbi viia liikumisi mingitel trajektooridel robotiga, millel on *suur arv vabadusastmeid* (ingl.k high degrees of freedom). Mudelipõhise testimise abil genereeriti testid ja testide käivitamiseks kasutati *Google Test Library* teeki (gtest). Testid viidi läbi masinal, mis kasutas operatsioonisüsteemi Ubuntu ja vahevara ROS. Eksperimendi tulemuste põhjal jõuti järeldusena, et mudelipõhine

testimine on sobilik viis komponentide integratsiooni ja süsteemi tasandi testide genereerimiseks ning testimiseks.

Marinescu, Saadatmand, Bucaioni, Seceleanu ja Petterson [19] pakkusid oma artiklis välja metodoloogia koodi valideerimiseks EAST-ADL'iga loodud mudelite baasil. EAST-ADL on isejuhtivate sõidukite sardsüsteemide arhitektuuri kirjeldamiseks loodud keel, mille abil loodud mudelite abil genereeriti artiklis testilood. Artiklis rakendati metodoloogiat Volvo Brake-by-Wire süsteemi lihtsustatud prototüübil, milles testiti mudelipõhise testimise abil antud süsteemi korrektsed käitumist. Testide käivitamiseks loodi Pyhton'is skriptid ja UPPAAL PORT'i abil genereeriti formaalsed mudelid.

Käesolev magistritöö erineb eelpool välja toodud teadustöödest kõige enam seetõttu, et see keskendub mudelipõhise integratsiooni testimise rakendamisele Autoware'iga loodava süsteemi näitel. Teiseks keskendutakse magistritöös sellele, et töös kasutatavaid tööriistasid, teste ning testimis keskkonda oleks võimalikult lihtne hiljem teistes Autoware'i ja ROS'i kasutavates projektidesse integreerida. Samuti kasutatakse enamikes artiklitest mudelipõhise testimise tööriistana UPPAAL'i , mis ei ole täielikult vabavaraline nagu NModel. Eelpool mainitud artiklid on aga kindlasti abiks automatiseeritud testimise keskkonna ning tööriistade loomiseks, mis oleks taaskasutatav Autoware'i ja ROS'i projektides.

2 Eksperimendi keskkond ja nõuded testitavale süsteemile

Selles peatükis vaatleme kuidas on üles seatud keskkond millel on testid läbi viidud. Vaatame kuidas täpsemalt toimub auto iseseisev juhtimine süsteemis Autoware. Kirjeldame vajalikud nõuded mida soovime testida ja seletame lahti milliseid komponente me täpsemalt testime ning kasutame. Peatüki lõpus on loodud näidis mudelprogramm mudelipõhise testimise tutvustamiseks ja meelde tuletamiseks programmi „turtlesim“ näitel.

2.1 Keskkond

Magistritöös kasutati ROS'i versiooni ROS Kinetic. Autoware tarkvara ja ROS vahevara on paigaldatud järgmiste spetsifikatsioonidega masinale:

Tabel 2. Autoware'i poolt soovitatud ja test masina spetsifikatsioonid.

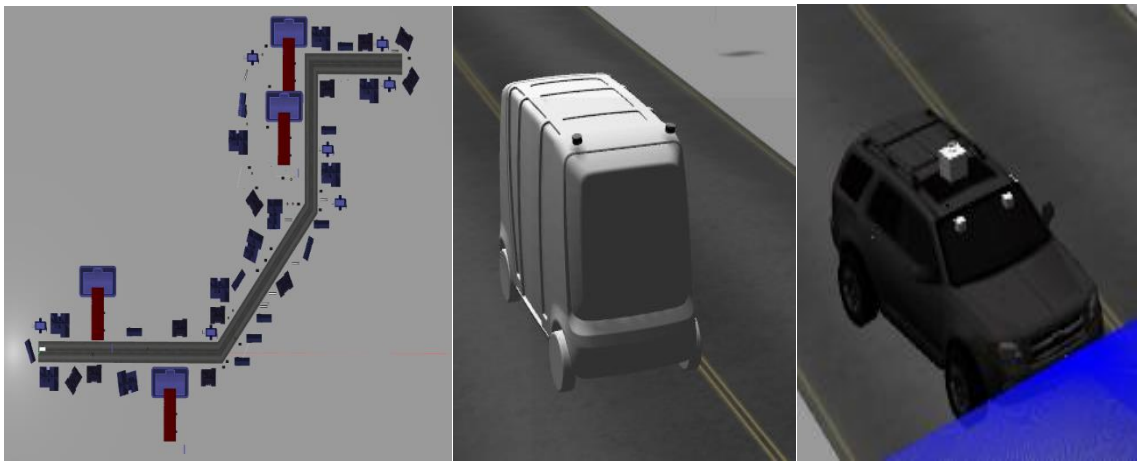
Süsteemi spetsifikatsioon	**Test masinal**	**Soovitatav Autoware'i jaoks**
Operatsioonisüsteem	Ubuntu 16.04	ROS indigo (Ubuntu 14.04) ROS jade (Ubuntu 15.04) ROS kinetic (Ubuntu 16.04)
Graafika kaart	Quadro K4000	Soovitatakse kasutada NVIDIA graafikakaarte, mis toetavad CUDA tehnoloogiat.
Muutmälu	16	32
Protsessor tuumade arv:	8 (Intel Xeon(R) CPU E5-1620v2 @ 3.70GHz)	8

Autoware paigaldus juhend on kättesaadav aadressilt https://github.com/CPFL/Autoware ja ROS'i oma aadressilt http://wiki.ros.org/kinetic/Installation. Magistritöö raames kasutatud Autoware'i versiooni salv on kloonitav aadressilt https://gitlab.cs.ttu.ee/iseauto/autoware. Mudelprogrammi arendamisel on kasutatud Visual Studio 2017 integreeritud programmeerimiskeskkonda.

Lisaks on vaja Linux'i operatsiooni süsteemis testide käivitamiseks installeerida MONO raamistik mille abil on võimalik kompileeritud DLL-teeke Linux'i operatsiooni süsteemi kasutavatel masinatel käivitada. Mono on allalaetav Linux'i operatsiooni süsteemides käsurealt järgneva käsuga:

```
sudo apt install mono-devel
```

Gazebo maailm ja kasutatud auto mudel milles testid läbi viidi oli loodud TTÜ Isesõitva auto meeskonna liikmete poolt (vt Joonis 4). Testimisel kasutati nii kahe lidari kui ühe lidariga seadistatud autot. TTÜ isesõitva auto mudel on loodud Arizona Ülikooli isesõitva auto mudeli baasil CAT ("Cognitive and Autonomous Test Vehicle" ehk kognitiivne ja autonoomne testi sõiduki), mida kasutasin Autoware staabilses harus testide läbiviimisel [20].



Joonis 4 . Testimisel kasutatav gazebo maailm, TTÜ Iseauto mudel ja CAT sõiduki mudel [20].

2.2 ROS# ja RosBridgeClient'i installeerimine

Selle jaoks et adapter saaks suhelda testitava süsteemiga, kasutame RosBridgeClientit, mis on C# implementatsioon ROS'i rosbridge_suite'ist. Rosbridge pakkub võimalust

suhelda ROS'i funktsionaalsusega läbi JSON API. Suhtlus toimub läbi WebSocket protokolli ja kasutab sõnumite edastamisel JSON formaati.

```
{ "op": "subscribe",
  "topic": "/cmd_vel",
  "type": "geometry_msgs/Twist"
}
```

Joonis 5. Näide rosbridge'iga kanali kuulamise protokollist [21].

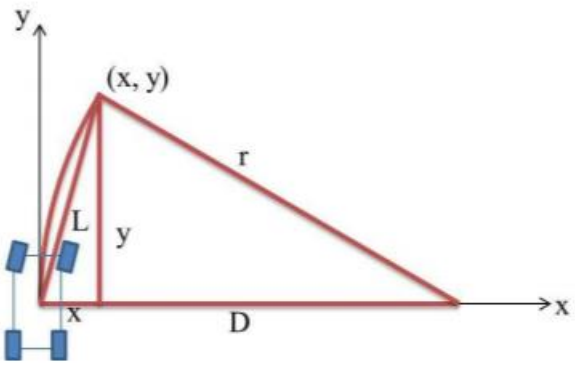
RosBridgeClient'i paigaldamiseks kasutame terminalis järgnevat käsku:

```
sudo apt-get install ros-kinetic-rosbridge-server
```

RosBridgeClient'is ei ole defineeritud Autoware sõnumite tüübid, seetõttu kloonime github'i salve järgnevalt aadressilt [22]. Kloonitud salve kaustast navigeerime kausta nimega ROS (ros-sharp/ROS), kust kopeerime kausta nimega „file_server" oma Autoware'i projekti *catkin workspace* src kausta. Kaust „file_server" sisaldab vajalikke käivitus skripte, mille abil on võimalik failide andmeid vahetada ROS'i teenustega. *Catkin workspace* on kaust kus on võimalik modifitseerida, kompileerida ja paigaldada *catkini* pakke. Kui vajalikud kaustad on kopeeritud navigeerime *catkin workspace*'i juur kausta ja käivitame terminalis käsuga `catkin_make` pakkide kompileerimise.

2.3 Testitavad komponendid

Autoware kasutab auto liigutamiseks mööda teekonnapunkte mitmeid sõlmi ja kanaleid, mida kujutab, Lisa 1. Testimise jaoks on meil vaja publitseerida teekonnapunktid kanalisse final_waypoints, mida kuulab sõlm pure_pursuit. Sõlm pure_pursuit saadab omakorda välja sõnumid kanalitele ning sõlmedele, mis juhivad autot simulatsioonikeskkonnas.



Joonis 6. Pure pursuit algoritmi geomeetria [23].

Sõlmes pure_pursuit kasutatakse algoritmi nimega pure pursuit (vt Joonis 6), mille abil arvutatakse välja kaar mida järgides auto sõidab hetke asukohast järgmisesse punkti. Algoritm toimib inimesega analoogiliselt, vaadeldes teatud kaugusesse valitud punkti ning juhtides autot nii, et see jõuaks punkti mida vaadeldakse [23]. Antud punkte mille abil pure pursuit arvutamist sooritab kutsutakse Autoware'i tarkvaras *teekonnapunktideks* (ingl.k waypoints).

2.4 Nõuded

Nõuete loomisel kasutame *käitumisekeskse arendamise* (ingl.k behaviour driven development) metoodikat [25]. Käitumiskesksete arendamise eesmärgiks on kirjeldada kuidas süsteem peaks käituma, kuid see ei anna ette täpseid suuniseid kuidas antud käitumist tuleks implementeerida. Samamoodi loome me ka mudelipõhise testimise puhul süsteemi kirjeldava mudeli, teadmisega kuidas süsteem peaks teatud sisendite korral käituma. Lisaks võimaldab see meie loodud mudelitest pöördkonstrueerida stsenaariume ja nõudeid või neid täpsustada, kuna testid ei keskendu tehnilisele aspektile vaid oodatud käitumisele.

Käitumiskeskses arenduses kasutatakse *given-when-then* malli nõuete kirja panemiseks [25].

- Sõnaga „EELDUSEL" (ingl.k given) antakse stsenaariumile esialgne kontekst
- Sõna „KUI" (ingl.k when) kasutatakse sündmuste kirjeldamiseks
- Sõna „SIIS" " (ingl.k then) kasutatakse väljundi kirjeldamiseks

Mudelist on tuletatud järgmine testi stsenaarium ja nõue:

Stsenaariumi nr	1
Teenus	Teekonna jälgimine. Auto peab suutma sõita mööda ette antud teekonnapunkte.
Stsenaarium	EELDUSEL et autole on määratud teekonnapunktid mida jälgida SIIS auto läbib ette antud trajektoori JA jõuab lõpp-punkti

Nõue on loodud mudeli põhjal, mis valmis selle jaoks et testida auto korrektset liikumist ühest teekonnapunktist teise.

2.5 Näide mudelprogrammi loomisest kasutades „turtlesim“ programmi ja ROS

Selles alampeatükis loome näitena mudelprogrammi, mille abil tutvustame lugejale mudelprogrammi loomist. Näite loomise jaoks kasutame ROS'i „turtlesim“ programmi, mis on loodud ROS'i õppimiseks. Antud näite kaasategemine eeldab et on installeeritud RosBridgeClient ja ROS. NModel'i kasutatud versioon on kättesaadav tööle lisatud „Iseauto.rar“ failist koos juba muudetud RosBridgeClient'i versiooniga. Samuti on antud näide leitav „Iseauto.rar“ failist. Lisaks tuleks oma arenduskeskkonnas viidata NModel'i (kaustas nimega „bin“, mis asub „Iseauto.rar“ failis) ja RosBridgeClient'i (kaustas nimega „ros-sharp“, mis asub „Iseauto.rar“ failis) DLL-teekidele.

Programmi „turtlesim“ abil testime, kas meie juhitav kilpkonn liigub meile soovitud viisil, ehk ei ürita ekraanist välja liikuda. Ekraanist välja liikumise korral saadetakse kanalist rosout sõnum „Oh, no! I hit the wall“.

Kõigepealt kirjeldame ära seisundi ja üleminekutingimuse mis peab olema täidetud, et vastavat toimingut meie näites sooritada.

Toiming: Kui kilpkonnale saadetakse sõnum turtle1/cmd_vel kanalisse siis kilpkonn liigub.

Tingimus: Kilpkonn tohib edasi liikuda ainult siis, kui tema ees ei esine takistusi.

Mudelis loome kaks toimingut (vt Koodinäide 1), millega sooritame kilpkonna liigutamise ja kontrollime et liigutamise tulemusena kilpkonn ei ole ekraanist välja liikunud ehk „seina“ sõitnud. Üleminkutingimustena kirjeldame järgnevad tingimused:

- Move – Kilpkonna on lubatud liigutada.
- didMove – Kilpkonna on liigutatud ja me saame kontrollida kas ta on seina liikunud toimingus checkCollision().

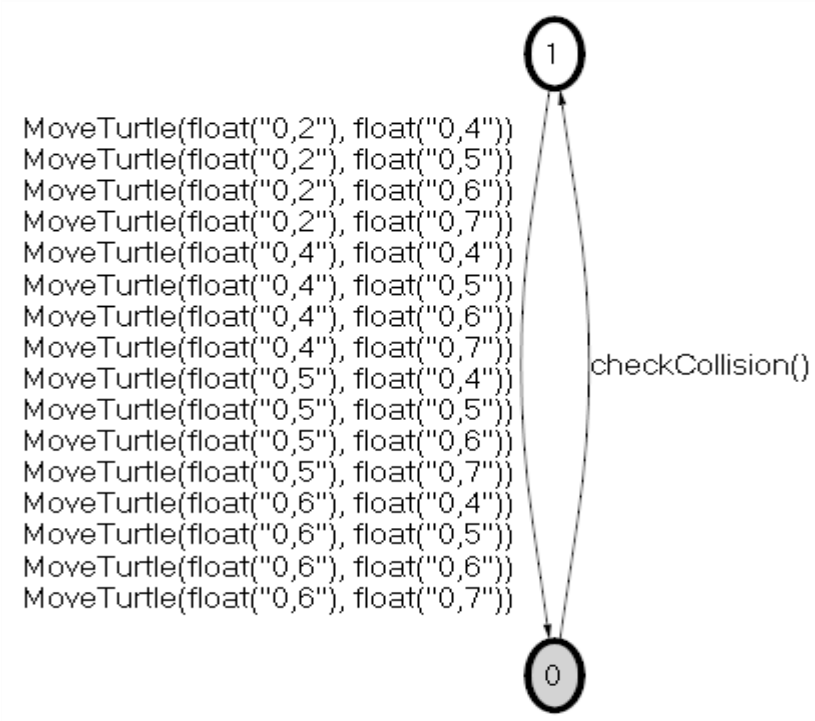
```
namespace TurtleModel

open NModel.Attributes
open NModel.Execution
open System
open RosSharp.RosBridgeClient

//0- liigutame konna 1- konn on liikunud
type turtleState=Move=0|didMove=1


type TurtleModelTest()=
  static member val velocity=0.5 //<-kilpkonna liikumise kiirus
  static member val angular=0.0  //<-kilpkonna liikumise nurk
  static member val turtleState=turtleState.didMove with get,set //kilpkonna
positsioonid

  //Kõik olekud on lubatud mille tulemusna kilpkonn ei ole seina sõitnud


  [<Action>]
  static member MoveTurtle([<Domain("Coordinates")>] xcoord:float32,
[<Domain("CoordinatesX")>] zcoord2:float32)=
     TurtleModelTest.turtleState<-turtleState.didMove

   //Kilpkonn saab liikuda siis ja ainult siis kui tingimuseks on määratud Move,
ehk kilpkonn ei ole seina sõitnud
   static member
MoveTurtleEnabled()=(TurtleModelTest.turtleState=turtleState.Move)

   //Kilpkonna võimalikud andmed liikumiseks, kiirus ja suund
   static member Coordinates () : NModel.Set<float32> =
        new
NModel.Set<float32>((float32)0.5,(float32)0.6,(float32)0.2,(float32)0.4,(float32)
0.5)
   static member CoordinatesX () : NModel.Set<float32> =
        new
NModel.Set<float32>((float32)0.7,(float32)0.6,(float32)0.4,(float32)0.4,(float32)
0.5)

  //Saame vastuse adapterist, kas konnn on vastu seina või mitte toimingu
sooritades
```

```
  [<Action>]
  static member checkCollision()=
    TurtleModelTest.turtleState<-turtleState.Move

  //Kilpkonna on liigutatud või soovitakse liigutada ja me saame kontrollida kas
ta on kokku põrganud või mitte
  static member
checkCollisionEnabled()=(TurtleModelTest.turtleState=turtleState.didMove)

  //Mudeli konstruktor
type Contract()=
    static member Create()=
        LibraryModelProgram(typedefof<Contract>.Assembly,"TurtleModel")
```

Koodinäide 1. Kilpkonna programmi mudeli näidis.

Olekugraafist on näha kuidas on kujutatud meie poolt loodud mudelit (vt Joonis 7). Olekugraafi on võimalik luua kasutades NModel'i vahendit MPV. Olekud on tähistatud numbritega 1 ja 0. Lubatud olekutele on lisatud ka rasvane piirjoon. Nagu olekugraafilt näha on välja toodud ka kõik võimalikud toimingud ja nende parameetrid.

Olekugraafist näeme et meie loodud mudel kontrollib kõigepealt toiminguga checkCollision(), kas kilpkonn on seina liikunud ja seejärel liigutab toiminguga MoveTurtle() kilpkonna ning tsükkel hakkab uuesti peale.



Joonis 7. Olekugraaf kilpkonna näitel.

Adapteri DoAction() meetodis käivitame vastavad tegevused süsteemis ja tagastame toimingust väärtuse null, kui toiming ei eelda et me tagastaks väärtust tagasi süsteemist

mudelisse. Lisaks tagastame mitte lubatud toimingu checkCollision(true), kui kilpkonn on seina sõitnud (vt Koodinäide 2).

```
using System;
using System.Threading.Tasks;
using RosSharp.RosBridgeClient;
using NModel.Conformance;
using NModel.Terms;
using System.Threading;
namespace StepperImp

{
    public class Stepper : NModel.Conformance.IStepper
    {

        public static float xposition { get; set; }
        public static bool isCollision = false;

        public static float yposition { get; set; }
        //Defineerime ja initsialiseerime kanalid mida kuulame ja millesse
kuulutame
        static RosSocket rosSocket = new RosSocket("ws://127.0.0.1:9090");
        static int publish_vel = rosSocket.Advertize("/turtle1/cmd_vel",
"geometry_msgs/Twist");
        static int subscribe_rosout = rosSocket.Subscribe("rosout",
"rosgraph_msgs/Log", rosoutCallback);
        static int reset_pub = rosSocket.Advertize("reset", "std_srvs/Empty");
        public CompoundTerm DoAction(CompoundTerm action)
        {


            switch (action.Name)
            {
                case ("Tests"): return null;
                //Liigutame konna
                case ("MoveTurtle"):
                    GeometryTwist message = new GeometryTwist();
                    var speed = (float)action[0];
                    var angleX = (float)action[1];
                    if (speed <= angleX)
                    {
                        message.linear.x = speed;
                    }
                    else
                    {
                        message.linear.x = 7.0f;
                        message.angular.z = 7.0f;
                    }

                    rosSocket.Publish(publish_vel, message);
                    isCollision = false;
                    return null;
                //Kontrollime ette antud aja pärast kas konn on jõudnud kohale
                case ("checkCollision"):
                    Console.WriteLine(isCollision);
                    var task = Task.Run(() => moveOrFail());
                    if (task.Wait(TimeSpan.FromSeconds(10)))
                        return task.Result;
                    else
```

```
                    throw new Exception("Timed out");


                default: throw new Exception("Unexpected action " + action);

            }
        }
        //Kontrollime et konn ei oleks seina liikunud
        static CompoundTerm moveOrFail()
        {
            Thread.Sleep(5000);
            if (isCollision==false)
            {
                Console.WriteLine("´Kas sein on ees  {0}", isCollision);
                return null;
            }
            else
            {


                return NModel.Terms.Action.Create("checkCollision",true);
            }
        }
        //Testi algoleku taastamine - tearDown
        void IStepper.Reset()
        {

        }
        //Adapteri konstrukto
        public static IStepper Create()
        {
            return new Stepper();
        }

        //Kuulame kas konn on liikunud seina
        private static void rosoutCallback(Message message)
        {

            RosGraphMsgsLog geo = (RosGraphMsgsLog)message;


            Console.WriteLine(geo.msg);
            if (geo.msg.Contains("Oh no! I hit the wall!"))
            {
                Console.WriteLine("Hit the wall");
                isCollision = true;
            }
            else
            {

                Console.WriteLine("Did not hit the wall");



            }
        }
    }
}
```

Koodinäide 2. Kilpkonna programmi adapteri näidis.

Meetodis rosoutCallback() kuulame kanalist rosout pidevalt, kas meie kilpkonn on sõitnud vastu seina või mitte.

Kui vajalikud mudel ja adapter on loodud tuleks projekt kompileerida. Enne testi käivitamist tuleks kõik vajalikud vahendid ja kompileeritud mudeli ning adapteri DLL-teegid kopeerida ühte kohta. NModel'i vahendid on kopeeritavad „Iseauto.rar" kaustast nimega „bin" või github salvest aadressil https://github.com/neira24/IseautoMBT.

Kui kõik eelnev on sooritatud tuleks järgmisena tabelis toodud käskude abil käivitada viidatud komponendid (vt Tabel 3).

Tabel 3. Käsud vajalike komponentide käivitamiseks kilpkonna näites.

Käivitatav komponent	**Käsk käsureal**
Roscore'i käivitamine	`roscore`
„turtlesim"'i käivitamine ja kompileerimine	`rosmake turtlesim` `rosrun turtlesim turtlesim_node`
Rosbridge'i käivitamine	`rosrun rosbridge_server rosbridge_websocket`
Testi käivitamine (Käivitada kaustas kus on kõik vajalikud DLL-teegid ja Nmode'i vahendid)	`Windows: ct.exe @ct_args.txt` `Ubunut: mono ct.exe @ct_args.txt`

Veel täpsemalt saab lugeda NModel'i kasutamisest Rivo Roo magistritöös [4].

Kilpkonna käivitamise argumendifail näidis on leitav lisadest, Lisa 4.

3 Autoware'iga loodava tarkvara mudelipõhine testimine

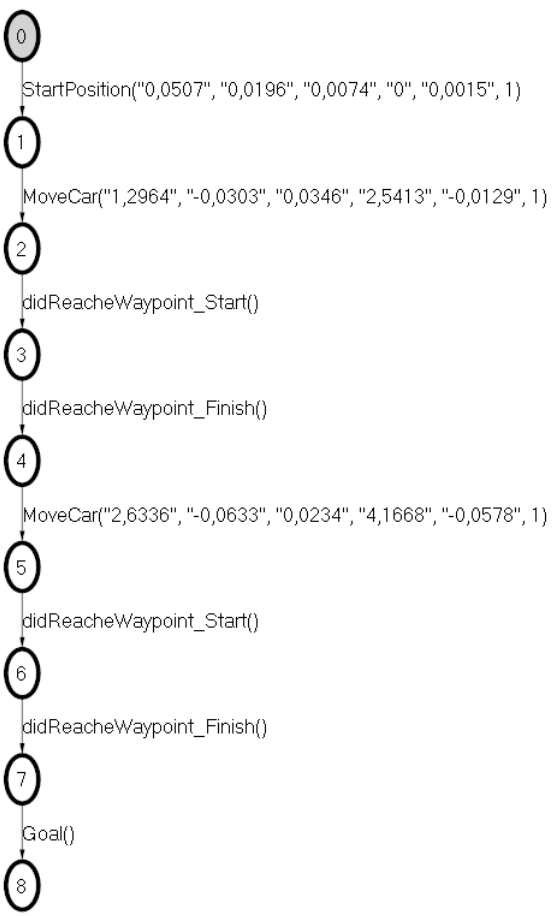
Selles peatükis vaatame kuidas on loodud testitavat süsteemi kirjeldavad mudelid ja mudelite adaptereid, mille abil on võimalik mudeli ja testitava süsteemi vahel infot vahetada. Samuti vaatame kuidas teste käivitada kasutades NModel'i testide käivitamise vahendit CT (ingl.k Conformance Tester), mille abil on võimalik sooritada käigupealt testimist ja eelsalvestatud testlugudega testimist. Lisaks vaatame peatüki lõpus kuidas mõõta koodikatvust.

3.1 Iseliikuva auto juhtimise mudel

Auto liikumise juhtimiseks ja korrektse juhtimise kontrollimiseks lõin mudeli, mille abil saame kontrollida kas auto on kindla ajavahemiku jooksul läbinud ette antud teekonnapunkti. Kui seda juhtunud ei ole, on tõenäoliselt auto teekonnalt eksinud, või on esinenud mõni muu põhjus, mis takistas autol etteantud punkti jõudmist. Täpsemalt kirjeldame punkti kontrollimise meetodit alampeatükis 3.2.

Joonis 8 kujutab olekugraafi Autoware'i süsteemi auto liikumisest teekonnapunktide abil. Joonisel on näha et algselt laetakse sisse teekonnapunkte sisaldavast failist üks teekonnapunkt (toiming StartPosition()), mis on meie algpunkt kust masin edasi liikuma hakkab. Seejärel tehakse toiming MoveCar() millega saadetakse autole järgmine punkt kuhu auto liikuma hakkab algpunktist. Viimasena kontrollitakse jälgitava toiminguga didReacheWaypoint() kas auto on jõudnud järgmisesse teekonnapunkti. Kui kõik punktid on läbitud liigutakse toiminguga Goal() viimasesse olekusse (nr 8), mille abil teame, et kõik punktid läbiti edukalt.

„Jälgitav toiming, on toiming mida ei saa testimisvahendiga käivitada ja mille teeb testitav süsteem muude tegevuste tulemusena. Testimisvahend saab selliste toimingute esinemist jälgida" (Roo 2010, 59).



Joonis 8. Auto liikumist kirjeldava mudeli olekugraaf.

Loodud test on asünkroonne, võrreldes sünkroonsete testidega võimaldab see teatud toiminguid algatada testitaval süsteemil endal. Seda on vaja et mudelile saata adapterist infot selle kohta, kas auto on jõudnud oodatud teekonnapunkti või mitte. Kui auto on punkti jõudnud käivitatakse toiming `didReacheWaypoint_Finish()` või ebaõnnestumise korral `didNotReacheWaypoint()`. Jälgitav toiming `didNotReacheWaypoint()` tagastab ka logi faili punkti koordinaadid millesse üritati jõuda. Salvestatud logifailiga on võimalik hiljem teste reprodutseerida ning täpsemalt tuvastada, miks test ebaõnnestus ning mis võis olla põhjus mille pärast soovitud punkti ei jõutud. Joonisel välja toodud mudelis saadetakse iga kord toiminguga MoveCar ühe teekonnapunkti andmed. Saadetavate teekonnapunktide arvu saab muuta mudelis, muutes atribuudi numOfWpPubAtOnce väärtust (vt Koodinäide 3). Mudeli kood asub IseautoModel projektis PortableLibrary1.fs failis (vt Lisa 5).

```
[<Action>]
 static member StartPosition([<Domain("CoordinatesX")>]
xcoord:string,[<Domain("CoordinatesY")>]
xcoord2:string,[<Domain("CoordinatesZ")>] ycoord2:string,[<Domain("Velocity")>]
zcoord2:string,[<Domain("Yaw")>] yaw:string,[<Domain("CountOfWp")>]wp:int32)=
```

```
   iseautoTest.stepcount<-iseautoTest.stepcount+1
   iseautoTest.numOfWpPubAtOnce<-4 //<-- Ette antavate teekonnapunktide arvu
muutmine
   iseautoTest.transition<-Transitions.Move
   iseautoTest.positionState<-PositionState.ReacheWp
   iseautoTest.startedWaitingMsg<-true
```

Koodinäide 3. Teekonnapunktide arvu muutmine PortableLibrary1.fs failis.

Teekond mida soovitakse testis läbida loetakse mudelisse sisse .csv formaadiga failist, milles on kirjeldatud järgnevad teekonnapunktide parameetrid:

Tabel 4. Teekonnapunktide parameetrid.

Parameetrid	**Selgitus**
X, Y ja Z koordinaadid	Koordinaat punktid kaardil.
Velocity	Kiirus millega antud teekonnapunkti läbitakse.
Yaw	Lengerdusnurk. Määrab auto suuna.

Uute teekonna punktide sisse lugemiseks võib asendada olemas oleva wp.csv faili sisu uute koordinaatidega või muuta sisse loetava faili nime mudelis (vt Koodinäide 4).

```
type iseautoTest()=
 static let mutable pass : double=1.0
 static let waypointsFilePath="../wp.csv" // <- Sisseloetav teekonnapunktide fail
 static let _waypointData = CsvProvider<Schema = "x (float), y (float), z
(float), yaw (float),velocity(float),change_flag(float)",
HasHeaders=false>.Load(waypointsFilePath)
static let waypointCount=(_waypointData.Rows|>Seq.length)
```

Koodinäide 4. Teekonnapunktide faili määramine ja sisselugemine.

Üleminekutingimusi mis peavad olema täidetud et käivitada toiminguid kirjeldab Tabel 5. Tabelis on plussiga (+) märgitud üleminekutingimused, mis peavad kindlalt lubatud olema et toimingut sooritada. Miinusega (-) on märgitud tingimused, mis ei pea olema konkreetsete toimingute teostamiseks lubatud.

Tabel 5. Toimingute üleminekutingimusi kirjeldav tabel.

Üleminekutingimused/toiming	**StartPostition**	**MoveCar**	**didReacheWaypoint_Start**	**didReacheWaypoint_Finish**	**Goal**
Start	+	-	-	-	-
Move	-	+	-	-	+
Wait	-	-	+	-	-
ObserveMsg	-	-	-	+	-
startedWaitingMsg	-	-	-	+	-
stepcount<=waypointCount	-	+	-	-	-
Stepcount>=waypointCount	-	-	-	-	+

3.2 Testi adapter

Adapter toimib ühenduslülina testitava süsteemi ja mudeli vahel. Selle joaks et meie mudel saaks suhelda läbi adapteri Autoware'ile spetsiifiliste sõnumi tüüpidega, tuleb meil defineerida uued sõnumi tüübid RosBridgeClient'is . Sõnumi tüüpide lisamiseks tuleb modifitseerida RosBridgeClient'i projektis Message.cs ja MessageTypes.cs faili ning projekt uuesti kompileerida. Autoga suhtlemiseks kasutame final_waypoints kanalit ja auto hetke asukoha määramiseks simulatsioonikeskkonnas current_pose kanalit. Algselt kasutasin auto asukoha määramiseks kanalit gazebo/link_states ja selle tulemusena testid pidevalt ebaõnnestusid. Probleemiks osutus see, et kaartide orientatsioonid olid oma vahel nihkes ja seetõttu ei ühtinud Autoware ja gazebo simulatsiooni keskkonna asukoha punktid. Ajutise lahendusena hakkasin kuulama current_pose kanalit, mis andis korrektselt teisendatud auto asukoha punktid.

Selle jaoks et teekonnapunkte kuulutada ning auto hetke asukoha kohta infot kuulata tuleb lisada uued sõnumi tüübid, mida kirjeldab Tabel 6.

Tabel 6. Autoware ja Gazebo sõnumi tüübid ning väljad.

Sõnumi tüüp	**Sõnumi väljade nimed ja andmetüübid**
autoware_msgs/waypoint	public int aid; public uint lanechange_state; public uint steering_state; public uint STR_LEFT; public uint STR_RIGHT; public uint STR_STRAIGHT; public uint accel_state; public uint stopline_state; public uint event_state;
autoware_msgs/lane	public StandardHeader header; public int increment; public int lane_id; public waypoint[] waypoints;
autoware_msgs/dtlane	public float dist; public float dir; public float apara; public float r; public float slope; public float cant; public float lw; public float rw;
autoware_msgs/WaypointState	public int gid; public int lid; public GeometryPoseStamped pose; public GeometryTwistStamped twist; public dtlane dtlane; public int change_flag;
gazebo_msgs/LinkStates	public string [] name; public GeometryPose[] pose; public GeometryTwist[] twist;

Adapteris (IseautoStepper/Class1.cs) tuleb luua ühendus testitava süsteemiga WebsSocket'i abil, mida ROS# kutsutakse RosSocket'iks ja defineerida kanalid mida tellime ja millesse kuulutame.

Adapteris on loodud funktsioon calculateTimeTravelingBetweenWaypoints() mis arvutab välja vahemaa läbimiseks kuluva aja teekonnapunktide vahel.

Vahemaa välja arvutamiseks kahe teekonnapunkti vahel kolmemõõtmelises keskkonnas kasutame valemit $\sqrt{(x1-x2)^2+(y1-y2)^2+(z1-z2)^2}$, kus x_1, y_1, z_1 on teekonnapunk kus auto antud hetkel asub ja x_2, y_2, z_2 teekonnapunkt kuhu auto siirdub. Teekonna läbimise aja välja arvutamise jaoks kasutame valemit $\frac{distants}{kiirus}$, kus kiiruseks on

auto hetke seisul oleva teekonnapunkti määratud kiirus millega antud vahemaad läbitakse ning algpunkti ja lõpp-punkti vahemaa.

Kontrollimaks kas auto on saabunud oodatud teekonnapunkti kasutame valemit $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \leq r$, kus x_1, y_1 on current_pose või gazebo/link_states kanalilt saadud auto hetke seisu koordinaadid, x_2, y_2 auto järgmise teekonnapunkti koordinaadid ja r on raadius, millesse auto peab jõudma kindla aja jooksul.

Selle jaoks et veenduda kas auto teekonnapunkti raadiuses olemise kontrollimisviis töötab, viisin läbi katsetuse mudelprogrammi ja testitava süsteemiga. Katsetuse käigus paigutasin kõigepealt isesõitva auto teekonnapunkti raadiusesse, mille raadiuses ta olema peab. Seejärel käivitasin testi ja jälgisin kas test annab oodatud tulemuse ning väljastab et auto on nõutud raadiuses. Testi väljundiks anti vastus et auto on oodatud punktis ehk meie poolt paika pandud teekonnapunkti raadiuses (vt Joonis 9).

```
TestResult(0, Verdict("Success"), "",
    Trace(
        StartPosition("0.0507", "0.0196", "0.0074", "0", "0.0015", 1),
        MoveCar("1.2964,2.6336", "-0.0303,-0.0633", "0.0346,0.0234", "2.5413,2.5413", "-0.0129,-0.0578", 2),
        didReacheWaypoint_Start(),
        didReacheWaypoint_Finish(),
        Goal()
    )
```

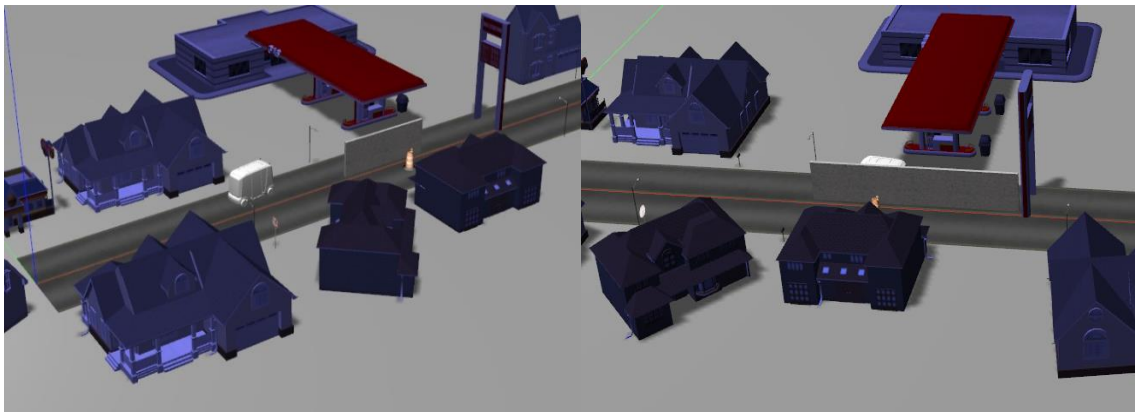
Joonis 9. Auto raadiuses olemise valideerimine.

Järgmisena liigutasin auto antud teekonnapunkti raadiusest välja ja käivitasin testi uuesti, mille puhul anti tulemuseks et auto ei asu oodatud punktis, ehk on oodatud teekonnapunkti raadiusest väljas (vt Joonis 10). Mõlemal puhul saavutasid katsetused oodatud tulemused (vt Lisa 5).

```
TestResult(0, Verdict("Failure"), "Action symbol 'didNotReachWaypoint' not enabled in the model",
    Trace(
        StartPosition("0.0507", "0.0196", "0.0074", "0", "0.0015", 1),
        MoveCar("1.2964,2.6336", "-0.0303,-0.0633", "0.0346,0.0234", "2.5413,2.5413", "-0.0129,-0.0578", 2),
        didReacheWaypoint_Start(),
        didNotReachWaypoint(float("2.6336"), float("-0.0633"), float("0.0234"))
    )
```

Joonis 10. Auto raadiusest väljas olemise valideerimine.

Lisaks viisin läbi testi, mille abil kontrollisin Gazebo maailmas auto teekonnale ette seatud takistuse abil, kas test annab korrektse tulemuse, kui auto järgmisesse teekonnapunkti ei jõua.



Joonis 11. Gazebo maailmas auto teekonnale ette seatud takistus.

Testi tulemusena anti oodatud väljund ehk test ebaõnnestus, sest auto ei jõudnud järgmisesse teekonnapunkti. Gazebo maailmas paigaldati sein auto ette testi jooksutamise ajal.

```
TestResult(2, Verdict("Failure"), "Action symbol 'didNotReachWaypoint' not enabled in the model",
    Trace(
        StartPosition("0.0507", "0.0196", "0.0074", "0", "0.0015", 1),
        MoveCar("1.2964,2.6336", "-0.0303,-0.0633", "0.0346,0.0234", "5,5", "-0.0129,-0.0578", 2),
        didReacheWaypoint_Start(),
        didReacheWaypoint_Finish(),
        MoveCar("4.1964,5.4969", "-0.1747,-0.3255", "0.0272,0.027", "5,5", "-0.1096,-0.1693", 2),
        didReacheWaypoint_Start(),
        didReacheWaypoint_Finish(),
        MoveCar("6.807,8.1961", "-0.5872,-0.8647", "0.0325,0.0301", "5,5", "-0.1905,-0.2471", 2),
        didReacheWaypoint_Start(),
        didReacheWaypoint_Finish(),
        MoveCar("9.7437,11.4063", "-1.267,-1.7011", "0.0362,0.0275", "5,5", "-0.2589,-0.2797", 2),
        didReacheWaypoint_Start(),
        didReacheWaypoint_Finish(),
        MoveCar("13.0199,14.7706", "-2.2301,-2.8816", "0.0256,0.0159", "5,5", "-0.3261,-0.3504", 2),
        didReacheWaypoint_Start(),
        didReacheWaypoint_Finish(),
        MoveCar("16.4648,18.3692", "-3.4777,-4.1363", "0.0082,-0.0009", "5,5", "-0.35,-0.3499", 2),
        didReacheWaypoint_Start(),
        didReacheWaypoint_Finish(),
        MoveCar("19.378,20.947", "-4.4723,-5.0503", "-0.009,-0.0208", "5,5", "-0.3508,-0.3581", 2),
        didReacheWaypoint_Start(),
        didReacheWaypoint_Finish(),
        MoveCar("22.2616,23.5496", "-5.5841,-6.2179", "-0.0275,-0.0362", "5,5", "-0.4301,-0.4913", 2),
        didReacheWaypoint_Start(),
        didReacheWaypoint_Finish(),
        MoveCar("24.9078,26.1998", "-6.9799,-7.8343", "-0.0503,-0.0643", "5,5", "-0.5592,-0.6342", 2),
        didReacheWaypoint_Start(),
        didReacheWaypoint_Finish(),
        MoveCar("27.3985,28.4952", "-8.7801,-9.7697", "-0.0709,-0.088", "5,5", "-0.7029,-0.7538", 2),
        didReacheWaypoint_Start(),
        didNotReachWaypoint(float("28.4952"), float("-9.7697"), float("-0.088"))
    )
```

Joonis 12. Testi tulemus simulatsiooni keskkonda lisatud takistuse korral.

Testimiskeskkonna algoleku taastamise jaoks lõin kesta skripti (ingl.k shell script) nimega reset.sh, mis seab auto tagasi algpunkti ja taastab gazebo maailma algse keskkonna seadistuse. Skript käivitatakse adapteris meetodis Reset() (vt Koodinäide 5). Antud fail on lisatud tööga kaasa pandud „Iseauto.rar“ faili kausta nimega „Skriptid“.

```
//Testi restartimine
     void IStepper.Reset()
     {
         Console.WriteLine("Reset test");
         lanelist.Clear();
         nextToReach = null;
         wpToReachList.Clear();
         ExecuteCommand("cd ~/lauri-autoware/abi; ./reset.sh");

     }
```

Koodinäide 5. Algse test keskkonna taastamine reset.sh käivitamise abil adapterist.

3.3 Testide käivitamine

Käigupealt testide käivitamine toimub NModeli vahendi CT abil. Kuid enne seda on vaja ülesse seada vajalikud Autoware sõlmed ja Gazebo simulatsioonikeskkond.
Selle jaoks et autot mööda teekonnapunkte liikuma panna on vaja käivitada ja laadida Tabel 7 kirjeldatud komponendid kas Autoware'i graafiliseliidese või roslaunch'i abil. Samamoodi on käivitatavad ka simulatsioonikeskkond Gazebo ja visualiseerimis vahend RViz. Stsenaariumi nr 1 testimisel valiti kontrollimis raadiuseks 3,2 meetrit.

Tabel 7. Auto liigutamiseks vajalike sõlmede ja andmete kirjeldus.

Map (kaardid)	
TF	Koordinaatide teisendamise teek.
Point cloud	Laeb sisse punkti pilve andmed .pcd faili vorminguga failist.
Vector map	Laeb sisse vektorkaardi andmed
Computing (andmetöötlus)	
vel_pose_connect	Käivitab sõlme, mis tagastab infot kanalisse curret_pose, mille abil on võimalik saada teada auto asukoht.
ndt_matching	Laeb punktipilve andmed kaardina
pure_pursuit	Käivitab pure pursuit algoritmi sõlme

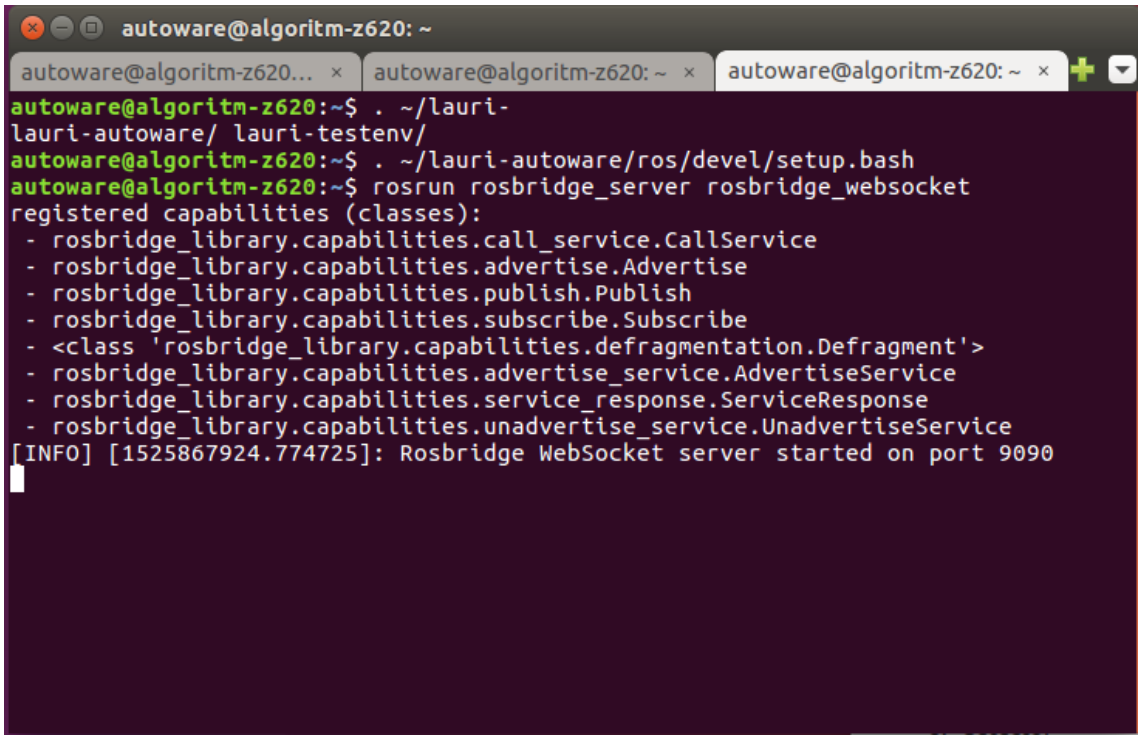
twist_filter	Käivitab sõlme, mis saadab auto liigutamiseks vajalikke andmeid simulatsiooni keskkonda
Sensing (Sensoorika)	
voxel_grid_filter	Kasutab VoxelGrid filtrit punktipilve vähendamiseks.
points_concat	Ühendab kahe punktipilve punktid üheks punktipilveks. (Vajalik ainult kahe lidari kasutamise puhul)

Kui Autoware' vajalikud sõlmed ja kanalid on tööle pandud seame üles keskkonna milles käivitame rosbridge_server'i. Rosbridge_server'i abil saame RosBridgeClient'iga panna suhtlema mudeli ja testitava süsteemi. Selle jaoks avame uue terminali akna, millesse sisestame käsu:

```
source ~/[Autoware projekti kaust]/ros/devel/setup.bash
```

Eelpool toodud käsuga saame käivitada Autoware setub.bash kesta antud terminali keskkonnas. See on vajalik selle jaoks, et RosBridgeClient saaks eelnevas alampeatükis defineeritud sõnumi tüüpe kasutada Autoware'i tarkvaraga suhtlusel. Rosbridge_server'i käivitamiseks anname käsureal järgneva käsuga (vt Joonis 13):

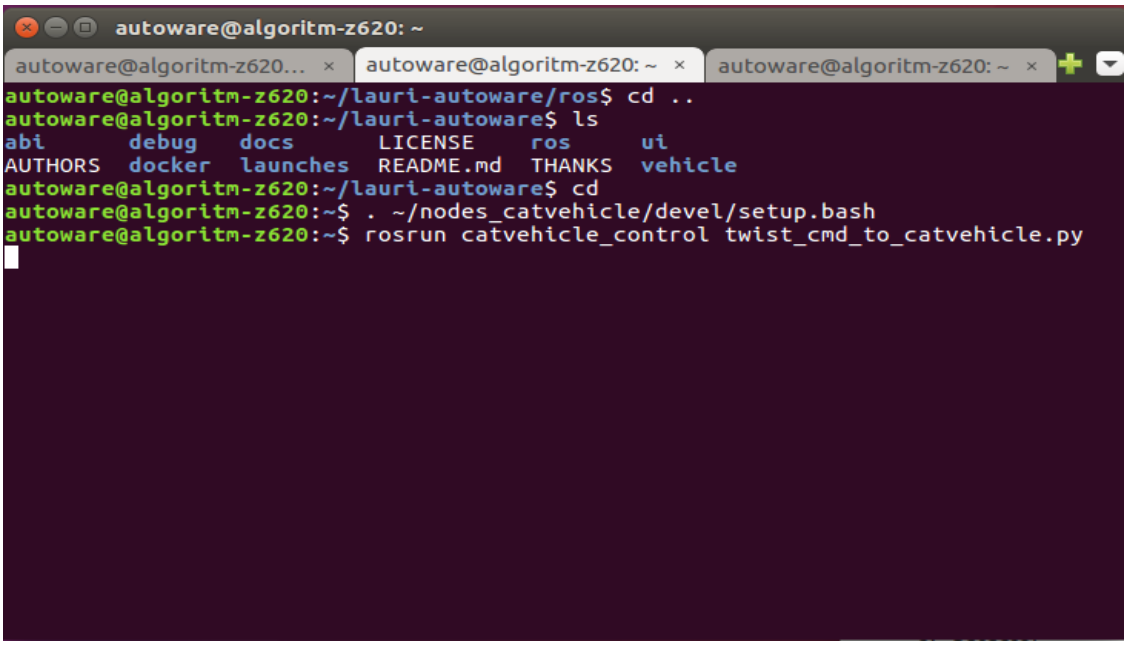
```
rosrun rosbridge_server rosbridge_websocket
```



Joonis 13. RosBridgeClient'i käivitamise näide.

Selle jaoks et auto oleks võimalik liikuma panna lõi Priit Trink oma magistritöö raames uue sõlme, mida kasutame auto liikuma panemiseks. Enne auto liikuma panemise sõlme käivitamist käsuga `rosrun catvehicle_control twist_cmd_to_catvehicle.py` tuleks käivitada järgneva käsuga kesta fail samas terminali keskkonnas (vt Joonis 14):

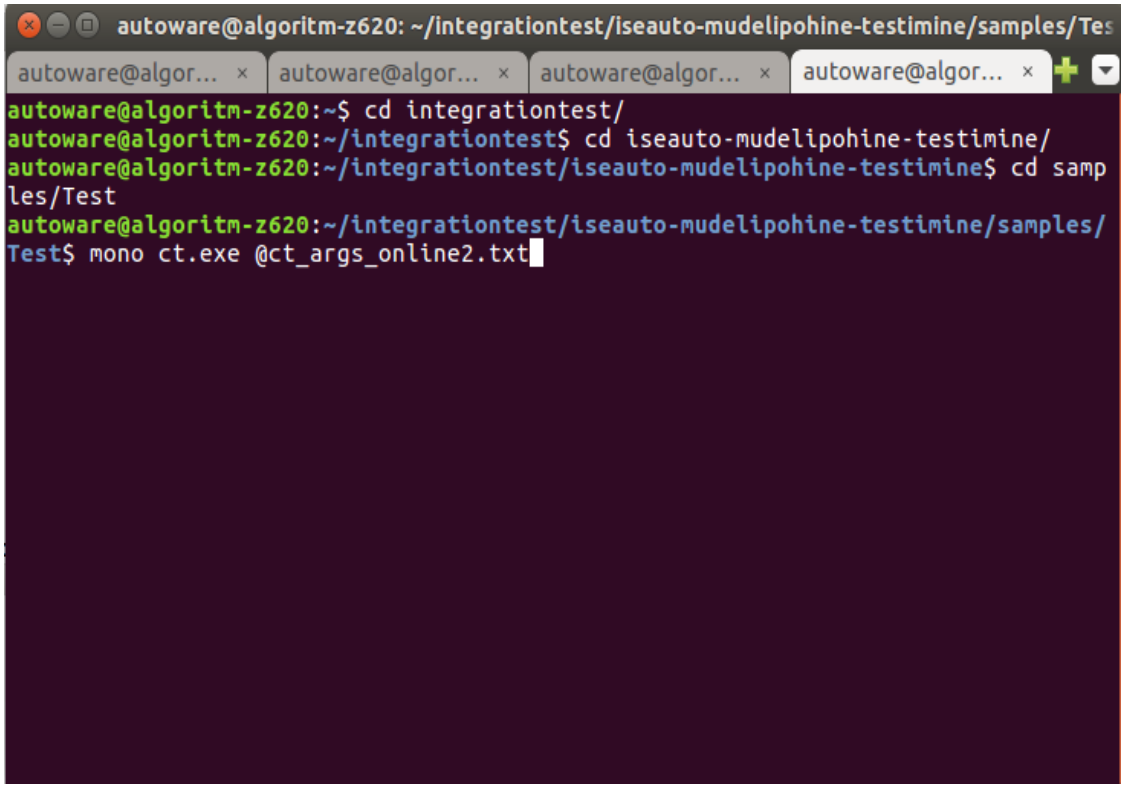
```
. /nodes_catvehicle/devel/setup.bash
```



Joonis 14. Auto liikuma panemise näide.

Viimase sammuna tuleks kloonida github salv aadressilt https://github.com/neira24/IseautoMBT või kopeerida kaasa pandud „Iseauto.rar“ failist testid ja navigeerida kausta nimega „Test“. „Test“ kaustas on adapteri ja mudelprogrammi DLL-teegid ning vajalikud NModel’i vahendid testide käivitamiseks. Testide käivitamiseks avame uue akna terminalis ja käivitame käsuga (vt Joonis 15):

```
mono ct.exe @[argumendifaili nimi].txt testi
```



Joonis 15. Testide käivitamine vahendi CT abil.

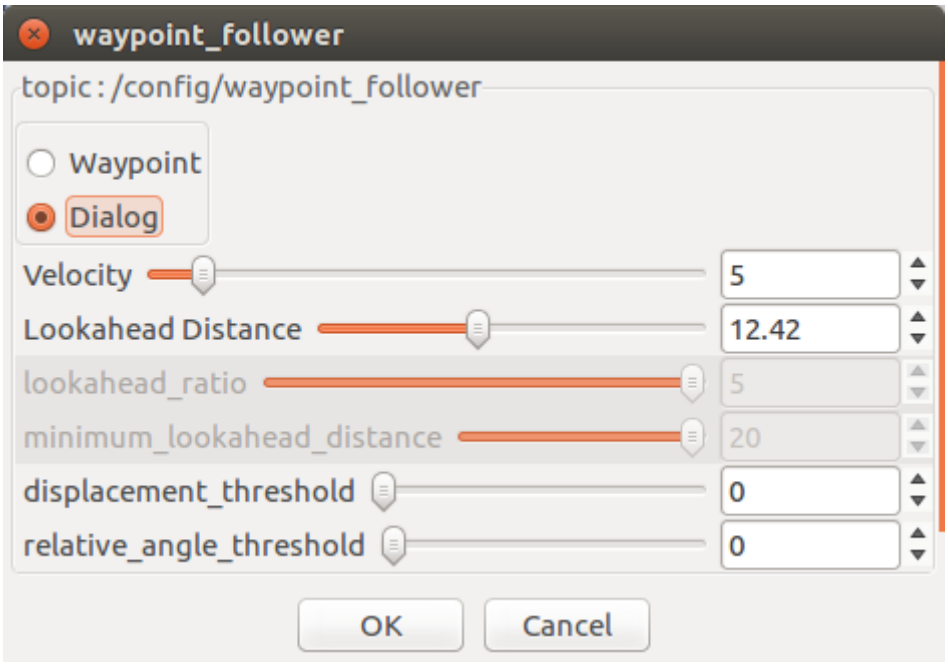
Argumendifailis on kirjeldatud argumendid mida NModel’i vahend CT kasutab testide käivitamiseks ja läbiviimiseks (vt Joonis 16).

```
## ct_args_online.txt
## Viited DLL-teekidele, mudelile, adapterile ja testitavale süsteemile
/r:IseautoStepper.dll
/r:IseautoModel.dll
/iut:IseautoStepper.Stepper.Create
/mp:IseautoModel

# Jälgitav toiming
/o:didReacheWaypoint

# Määrame kas - et pärast ebaõnnestumist testimine lõpetatakse
/continueOnFailure:-

# Testide käivituste arv

/runs:100

# Logi failinimi, kuhu logi kirjutatakse

/logfile:Iseautologi.txt
```

Joonis 16. Argumendifaili näide.

3.4 Stsenaariumi nr 1 testimine

Testides kasutati eelnevalt läbitud teekonnast wp.csv faili salvestatud teekonnapunkte. Stsenaarium testimisel saadud tulemused sõltuvad Autoware'i pure pursuit algoritmi seadistusest ja sellest millal me adapterist anname autole ette järgmised teekonnapunktid. Kõigepealt viisin läbi katsetuse, kus kasutasin "Dialog" režiimi (vt Joonis 17) , mis määrab autole kindla kiiruse ja vaatluskauguse, mille abil auto antud teekonda läbib. Vaatluskauguse abil valib pure pursuit välja järgmise teekonnapunkti talle ette antud teekonnapunktidest, kuhu ta suunduma hakkab.



Joonis 17. Näide pure pursuit algoritmi konfiguratsioonist stsenaariumi testimisel.

Antud testis suutis auto läbida kogu teekonna nii Autoware'i stabiilses versioonis kui ka TTÜ iseauto projekti arenduse haru versioonis Autoware'ist (vt Joonis 18 ja Joonis 19). Antud katsetus viidi läbi nii Autoware'i stabiilses harus kui isesõitva auto projekti arenduse harus.

```
Test Metrics(
    General Summary(
         10 tests Executed,
         0 tests Failed,
         Pass Rate: 100.0%
    ),
    Toatal time spent in each action(
         Goal(Executed 10 times, Average execution time: 00:00:00) : 00:00:00.0003210,
         MoveCar(Executed 880 times, Average execution time: 00:00:00) : 00:00:00.3299730,
         StartPosition(Executed 10 times, Average execution time: 00:00:00.0020000) : 00:00:00.0262760,
         didReacheWaypoint_Start(Executed 880 times, Average execution time: 00:00:03.5460000) : 00:52:00.6633360
    )
)
```

Joonis 18. Autoware'i stabiilses harus läbiviidud testi tulemuste statistika.

```
Test Metrics(
    General Summary(
         10 tests Executed,
         0 tests Failed,
         Pass Rate: 100.0%
    ),
    Toatal time spent in each action(
         Goal(Executed 10 times, Average execution time: 00:00:00) : 00:00:00.0003260,
         MoveCar(Executed 890 times, Average execution time: 00:00:00) : 00:00:00.3213900,
         StartPosition(Executed 10 times, Average execution time: 00:00:00.0030000) : 00:00:00.0323590,
         didReacheWaypoint_Start(Executed 890 times, Average execution time: 00:00:02.6170000) : 00:38:49.5407100
    )
)
```

Joonis 19. TTÜ iseauto projektis läbiviidud testi tulemuste statistika.

Mõlemal juhul käivitati teste kümme korda ja ainus erinevus mis leiti, oli aeg mis kulus testide läbiviimiseks. Kokku kulus antud testide läbiviimiseks aega 1 tund ja 30 minutit. Isesõitva auto projekti auto testide läbiviimiseks kulus 14 minutit rohkem kui Autoware'i stabiilse harus testide jooksutamiseks. Testide läbiviimise aeg, on seetõttu erinev et arendus harus oleva TTÜ isesõitva auto mudeli mõõtmed ja ratta raadius on väiksem kui stabiilses harus kasutataval standardsel auto mudelilt.

Järgmisena kasutasin "Waypoint" režiimi, milles on võimalik määrata minimaalne vaatluskaugus, vaatluskauguse suhe kahe teekonnapunkti vahel, teekonnast eemale nihkumise lävi ja kahe teekonnapunkti vahele joonistatava kaare nurga lävi. Samuti võetakse auto kiiruse andmed "Waypoint" režiimis teekonnapunktide failist. Testimise jaoks määrati minimaalseks vaatluskauguseks 20 meetrit ja suhteks 5. Ülejäänud väärtused jäeti samaks. Mudelis muudeti ette antavate teekonnapunktide arv kahe punkti asemel viiele punktile.
Test ebaõnnestus, seetõttu et auto lõikas järgmisesse teekonda siirdudes liiga palju kurvi ja sõitis simulatsioonikeskkonnas paiknevale majale otsa. Testi jooksutati kahel korral ja mõlemad korrad ebaõnnestusid.

```
Test Metrics(
    General Summary(
         2 tests Executed,
         2 tests Failed,
         Pass Rate: 0.0%
    ),
    Failed Actions(
         (didNotReachWaypoint, Reason: Action symbol 'didNotReachWaypoint' not enabled in the model( 2 times ))
    ),
    Toatal time spent in each action(
         didReacheWaypoint_Start(Executed 104 times, Average execution time: 00:00:01.5620000) : 00:02:42.4584930,
         MoveCar(Executed 104 times, Average execution time: 00:00:00) : 00:00:00.0518130,
         StartPosition(Executed 2 times, Average execution time: 00:00:00.0360000) : 00:00:00.0730600
    )
)
```

Joonis 20. Waypoint režiimi testi tulemuste statistika.

Kõik testimisel tekkinud logifailid on leitavad tööle juurde lisatud „Iseauto.rar" failis kaustas „Tulemused" (vt Lisa 5).

3.5 Koodikatvuse mõõtmine

Koodikatvuse mõõtmise jaoks kasutame vahendit gcov [26], mis võimaldab mõõta meil Autoware'i sõlmede koodikatvust. Näitena on valitud pure_pursuit'i sõlm, mille koodikatvust mõõtma asume. Koodikatvuse mõõtmise jaoks tuleb kompileerida soovitud

sõlmed *silumis* (ingl.k debug) võtmega ja käivitada terminali keskkonnas vajaminevad kesta failid käsuga source. Magistritöö raames muudeti selle jaoks Autowares tarkvara kompileerimiseks kasutusel oleva skripti catkin_make_release viimast rida järgnevalt:

```
catkin_make -DCMAKE_BUILD_TYPE=Debug -DCMAKE_CXX_FLAGS="-fprofile-arcs -ftest-coverage" -DCMAKE_C_FLAGS="-fprofile-arcs -ftest-coverage" -DCMAKE_SHARED_LINKER_FLAGS="-lgcov --coverage" $@
```

Catkin_make_release fail tuleks kopeerida kausta kuhu soovime panna sõlmed, mille koodikatvust mõõtma hakkame. Näites on kopeeritud catkin_make_release ümber nimetatud catkin_make_debug failiks. Seejärel tuleks käivitada catkin_make_debug käsk, mis kompileerib soovitud sõlmed silumis võtme ja GCC valikutega (vt Joonis 21).

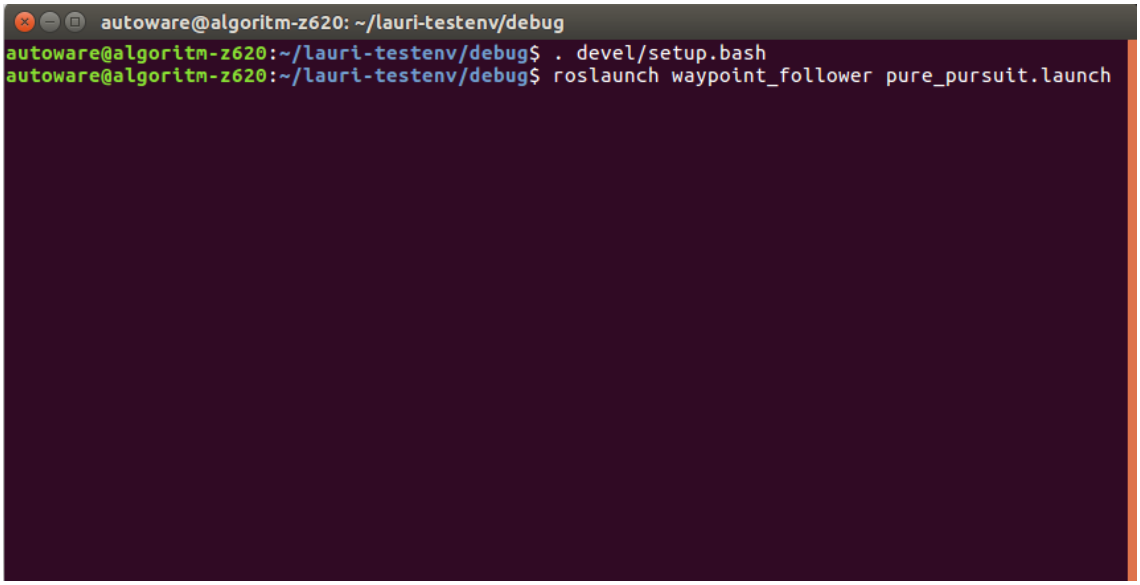
Joonis 21. GCC valikute ja silumis võtmega kompileerimine caktin_make_debug faili abil.

Järgmisena tuleks alla laadida gcovr [27] mille abil saame hiljem kuvada koodikatvuse tulemusi html failina [1].

Et koodikatvuse mõõtmist alustada, tuleks kõigepealt käivitada kõik Autoware'i vajalikud sõlmed, välja arvatud sõlmed mille koodikatvust soovime mõõta. Need tuleks käivitada kaustast kus me kompileerisime nad gcov'i jaoks vajalike parameetritega.

[1] Gcovr'i paigaldamise juhend on saadaval järgmiselt aadressilt: https://gcovr.com/guide.html.

Käivitamine toimub samamoodi nagu tavaliste roslaunch failide puhul, kuid enne seda peab laadima keskkonda antud kaustas oleva setup.bash kesta (nt `. /lauri-testenv/debug/devel/setup.bash`). Seejärel võib käivitada käsurealt pure_pursuit sõlme (vt Joonis 22).



Joonis 22. Pure_pursuit sõlme käivitamine.

Testide käivitamine toimub samamoodi nagu eelnevalt. Kui testid või soovitud tegevused on sooritatud, tuleks sulgeda käivitatud pure_pursuit sõlm. Kui sõlm veel töötab ei ole võimalik saada andmeid koodikatvusest, sest andmed antakse alles siis kui programm on oma töö lõpetanud.

Viimasena viime saadud andmed html kujule, et arendajatel oleks võimalikult lihtne tuvastada, milliseid koodiridu programmi töökäigus jooksutati. Selle jaoks anname käsurealt käsu (vt Joonis 23):

```
gcov -r . –html –html-details -o [Siia väljundfaili nimi].html
```

Joonis 23. Koodikatvuse kuvamine gcovr'i abil.

Tulemuseks luuakse html fail (vt Lisa 2), kus punasega tähistatakse koodirida, mida ei ole programmi töö jooksul käivitatud või meetodit, mille sees on osaliselt või üldse mitte käivitatud koodiridasid. Kõik käivitatud koodiread ja meetodid mille sees on kõik koodiread käivitatud on märgitud rohelise värviga (vt Lisa 3).

4 Tulemused

Selles peatükis analüüsime Autoware süsteemiga loodava isesõitva auto testimisel saavutatud tulemusi.

4.1 Testimise tulemuste analüüs

Mudelipõhise testimise valideerimisel õnnestus jälile jõuda sellele, et pure pursuit algoritmi kasutamisel on väga tähtis osa häälestamisel. Sellest tulenevalt on võimalik kasutada loodud mudelprogrammi pure pursuit algoritmi häälestamiseks ja häälestuse kontrollimiseks.

Osade mudelipõhiste testide ebaõnnestumise põhjuseks oli pure pursuit algoritmi seadistus, mida testi läbiviimise ajal kasutati. Testimisel aga ei ilmnenud probleeme „Dialog“ režiimi kasutades, küll aga „Waypoint“ režiimi kasutades. Selle jaoks et viimase režiimiga oleks võimalik teste õnnestunult jooksutada tuleks täiendada ka loodud adapterit. “Waypoint” režiimiga ei ilmnenud autol sõidu ajal simulatsioonikeskkonnas otsesõidul ja väikestel pööretel probleeme, kuid suurtel kiirustel kurvide võtmine osutus probleemseks. Probleem tuleneb pure pursuit’i algoritmist ja sellele ette antud seadistusest, millega vahel valitakse ette antud teekonnapunktide hulgast liiga kaugel olev punkt millesse auto siirduma hakkab. Selle tulemusena lõigatakse kurvi ja võidakse millelegi otsa sõita või mööduda järgmisest teekonnapunktist, liiga suure kaarega kurvis. Probleemi lahendamiseks tuleks adapteris sisse viia järgnevad muudatused/parandused:

- Tuleks võimaldada autole järskudes kurvides anda vähem teekonnapunkte suurema lengerdusnurga korral.
- Teine lahendus mida võiks kasutada oleks piirata teekonnapunktide kuulutamine ainult ühele teekonnapunktile korraga, kuid suurtel kiirustel võib seetõttu auto hakata vingerdama.

Positiivseks võib pidada loodud stsenaariumi (stsenaarium nr 1) testide jooksutamist, mille tulemusena jooksid testid rohkem kui pool tundi järjest, ning mille tulemusena ei õnnestunud tuvastada vigu. See näitab et antud parameetrite ja komponentidega suudab testitav süsteem vähemalt 52 minutit järjest ilma probleemideta töötada.

Töötulemusena õnnestus mõõta ka jooksutatud testi ajal pure pursuit’iga seotud sõlmede koodikatvust.

4.2 Hinnang mudelipõhise testimise rakendamisele

Mudelipõhise testimise abil on loodud struktureeritud viis ja töövoog integratsiooni testide läbiviimiseks Autoware'iga loodavale isejuhtivale süsteemile. Süsteemi jaoks loodud nõude testimisel ei ilmnenud tarkvaralisi vigu, kuid leiti et auto teekonna läbimine sõltub pure pursuit algoritmi häälestusest. Drastilisi kõrvalekaldeid süsteemi tavapärases töös testimise ajal ei ilmnenud.

Testid milles meelega manipuleeriti simulatsiooni keskkonna või kasutatud tarkvaraga, õnnestus mudelipõhist testimist kasutades tuvastada, loodud vead. Samuti ei tuvastanud ka mudelipõhine testimine valesid toiminguid õigeteks.

Seetõttu et Autoware'i dokumentatsioon ei olnud teatud sõlmede ning komponentide kirjeldamisel piisavalt detailne, muutis ka süsteemi tundma õppimise keerukamaks. Sellest tulenevalt võttis kauem aega adapteri loomine, sest ei olnud teada, kuidas täpselt testitav süsteem või selle komponendid töötavad. Näiteks kasutasin adapteris algselt auto positsiooni kuulamiseks gazebo kanalit gazebo/link_states, mille tulemusena testid pidevalt läbi kukkusid. Probleemiks osutus see et Autoware teisendab TF teegi abil koordinaate erinevate komponentide vahel sobilikuks ja seetõttu ei ühtinud antud kanali koordinaadid ülejäänud komponentide koordinaatidega.

Kõige suurem aeg kulus Autoware'ile mudelipõhist testimist rakendades süsteemi mõistmisele ja adapteri loomisele.

Pure pursuit "Waypoint" režiimiga testimise paremaks toetamiseks tuleks viia sisse järgnevad muudatused adapteris:

- Tuleks võimaldada autole järskudes kurvides anda vähem teekonnapunkte suurema lengerdusnurga korral.
- Teine lahendus mida võiks kasutada oleks piirata teekonnapunktide kuulutamine ainult ühele teekonnapunktile korraga, kuid suurtel kiirustel võib seetõttu auto hakata vingerdama.

Testimise käigus leiti järgnev viga NModel'i mugandatud versioonist, mis toetas programmeerimis keelt F#:

- Jälgitava toimingu loomisel ei teisendatud väärtusi tagastatavaid toiminguid automaatselt jälgitavateks vaid seda tuli teha manuaalselt.

Kokkuvõte

Käesoleva magistritöö tulemusena on välja valitud vahendid ja tutvustatud viisi kuidas mudelipõhise testimise abil viia läbi integratsiooni teste Autoware tarkvara kasutavates projektides. Näidisena loodud mudelprogrammi abil on võimalik testida korrektset liikumist isesõitvale autole ette antud teekonna puhul simulatsioonikeskkonnas. Teekond ja selle andmed on sisse laetavad .csv failist. Lisaks on tutvustatud viisi kuidas mõõta Autoware sõlmede koodikatvust, mille abil anda arendajatele tagasisidet käivitatud koodiridade kohta. Koodikatvuse andmete põhjal on võimalik kitsendada vea otsimisel vea leidmise piirkonda. Töö käigus loodi ka mudeli põhjal esmased nõuded, mida seejärel ka testiti. Nõuete loomiseks kasutati käitumiskesksest arendusest tulenevat *given-when-then* malli.

Testimine viidi läbi kahe erineva režiimiga mida pakuti pure pursuit algorimit puhul ja testiti stsenaariumit mis tulenes loodud nõudest. Testiti nii Autoware stabiilses harus ning TTÜ isesõitva auto arenduse harus. Kui mõlemal viisil ja režiimil suutsid autod läbida ühtlasel kiirusel teekonnad ilma probleemideta, siis „Waypoint“ režiimis ilmnesid vead. Viga võib tuleneda nii Autoware poolt kasutatava pure pursuit algoritmi seadistusest, kui ka loodud adapterist. Vea lahendamiseks adapteri poolelt pakuti välja ka järgnevad lahendused:

- Tuleks võimaldada autole järskudes kurvides anda vähem teekonnapunkte suurema lengerdusnurga korral.
- Teine lahendus mida võiks kasutada oleks piirata teekonnapunktide kuulutamine ainult ühele teekonnapunktile korraga, kuid suurtel kiirustel võib seetõttu auto hakata vingerdama.

Töö tulemusena on dokumenteeritud ja loodud näide, mis on laiendatav teistele Autoware’i projektidele ja simulatsiooni keskkondadele.

Antud töö edasiarendamise võimalused:

1. Luua uus mudelprogramm loodu näitel mille abil liigutada objekte gazebo maailmas ja jooksutada seda paralleelselt magistritöös loodud mudelprogrammiga.

2. Arendada magistritöös loodud mudelprogrammi edasi, nii et see toetaks ka objektidest mööda põikamist. Selle jaoks tuleks viia teatud muudatused sisse nii mudelis kui adapteris.
3. Lisada koodikatvuse toetus pythoni failidele kasutades Coverage.py [15] koodikatvuse mõõtmis vahendit.
4. Arendada loodud mudelit edasi nii, et see tuvastaks kokkupõrkeid gazebo maailmas oleva auto ja objektide vahel.

Kasutatud kirjandus

[1] J. Martin , N. Kim, D. Mittal ja M. Chisholm, „Certification for Autonomous Vehicles,“ [Võrgumaterjal]. Available: https://www.cs.unc.edu/~anderson/teach/comp790a/certification.pdf. [Kasutatud 14 11 2017].

[2] „Autoware,“ [Võrgumaterjal]. Available: https://github.com/CPFL/Autoware. [Kasutatud 10 11 2017].

[3] J. Ernits, E. Halling, G. Kanter ja J. Vain, „Model-based integration testing of ROS packages: A mobile robot case study,“ *2015 European Conference on Mobile Robots (ECMR)*, 2015.

[4] R. Roo, *Veebirakenduste mudelipõhine testimine asukohapõhise tarkvara näitel*, Tartu Ülikool, 2010.

[5] G. L. Geerts, „A design science research methodology and its application to accounting information systems research,“ *International Journal of Accounting Information Systems*, kd. 12, pp. 142-151, 2011.

[6] J. Jacky, M. Veanes, C. Campbell ja W. Schulte, Model-Based Software Testing and Analysis with C#, Cambridge: Cambridge University Press, 2007.

[7] K. Takaya, T. Asai, V. Kroumov ja F. Smarandache, „Simulation environment for mobile robots testing using ROS and Gazebo,“ *2016 20th International Conference on System Theory, Control and Computing (ICSTCC)*, 2016.

[8] „Gazebo,“ [Võrgumaterjal]. Available: http://gazebosim.org/. [Kasutatud 28 10 2017].

[9] „Robotic Operating System,“ [Võrgumaterjal]. Available: http://wiki.ros.org/. [Kasutatud 10 11 2017].

[10] „NModel,“ [Võrgumaterjal]. Available: https://nmodel.codeplex.com/. [Kasutatud 8 11 2017].

[11] „NModel: Library for model-based testing in C# (and F#),“ [Võrgumaterjal]. Available: https://github.com/juhan/NModel. [Kasutatud 25 March 2018].

[12] N. University, „Autoware-Manuals,“ [Võrgumaterjal]. Available: https://github.com/CPFL/Autoware-Manuals/blob/master/en/Autoware_UsersManual_v1.1.md#chapter-2---ros-and-autoware.

[13] „ROS standard message types,“ [Võrgumaterjal]. Available: http://wiki.ros.org/std_msgs. [Kasutatud 25 March 2018].

[14] „STRANDS,“ [Võrgumaterjal]. Available: http://strands.acin.tuwien.ac.at/software.html. [Kasutatud 28 March 2018].

[15] „Coverage.py,“ [Võrgumaterjal]. Available: https://coverage.readthedocs.io/en/coverage-4.5.1/. [Kasutatud 28 March 2018].

[16] L. Krejčí ja J. Novák, „Model-based testing of automotive distributed systems with automated prioritization,“ *2017 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, 2017.

[17] J. Laval, L. Fabresse ja N. Bouraqadi, „A methodology for testing mobile autonomous robots,“ *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems,* 2013.

[18] M. Abdelgawad, S. McLeod, A. Andrews ja J. Xiao, „Model-based testing of real-time adaptive motion planning (RAMP),“ *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR),* 2016.

[19] R. Marinescu, M. Saadatmand, A. Bucaioni, C. Seceleanu ja P. Pettersson, „A Model-Based Testing Framework for Automotive Embedded Systems,“ *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications,* 2014.

[20] „Proceedings 2nd International Workshop on Safe Control of Autonomous Vehicles, SCAV@CPSWeek 2018, Porto, Portugal, 10th April 2018,“ 2018.

[21] „Rosbridge JSON API,“ [Võrgumaterjal]. Available: http://wiki.ros.org/rosbridge_suite. [Kasutatud 12 04 2018].

[22] „ROS#,“ [Võrgumaterjal]. Available: https://github.com/siemens/ros-sharp.git. [Kasutatud 12 01 2018].

[23] W. J. Wang, T. M. Hsu ja T. S. Wu, „The improved pure pursuit algorithm for autonomous driving advanced system,“ *2017 IEEE 10th International Workshop on Computational Intelligence and Applications (IWCIA),* 2017.

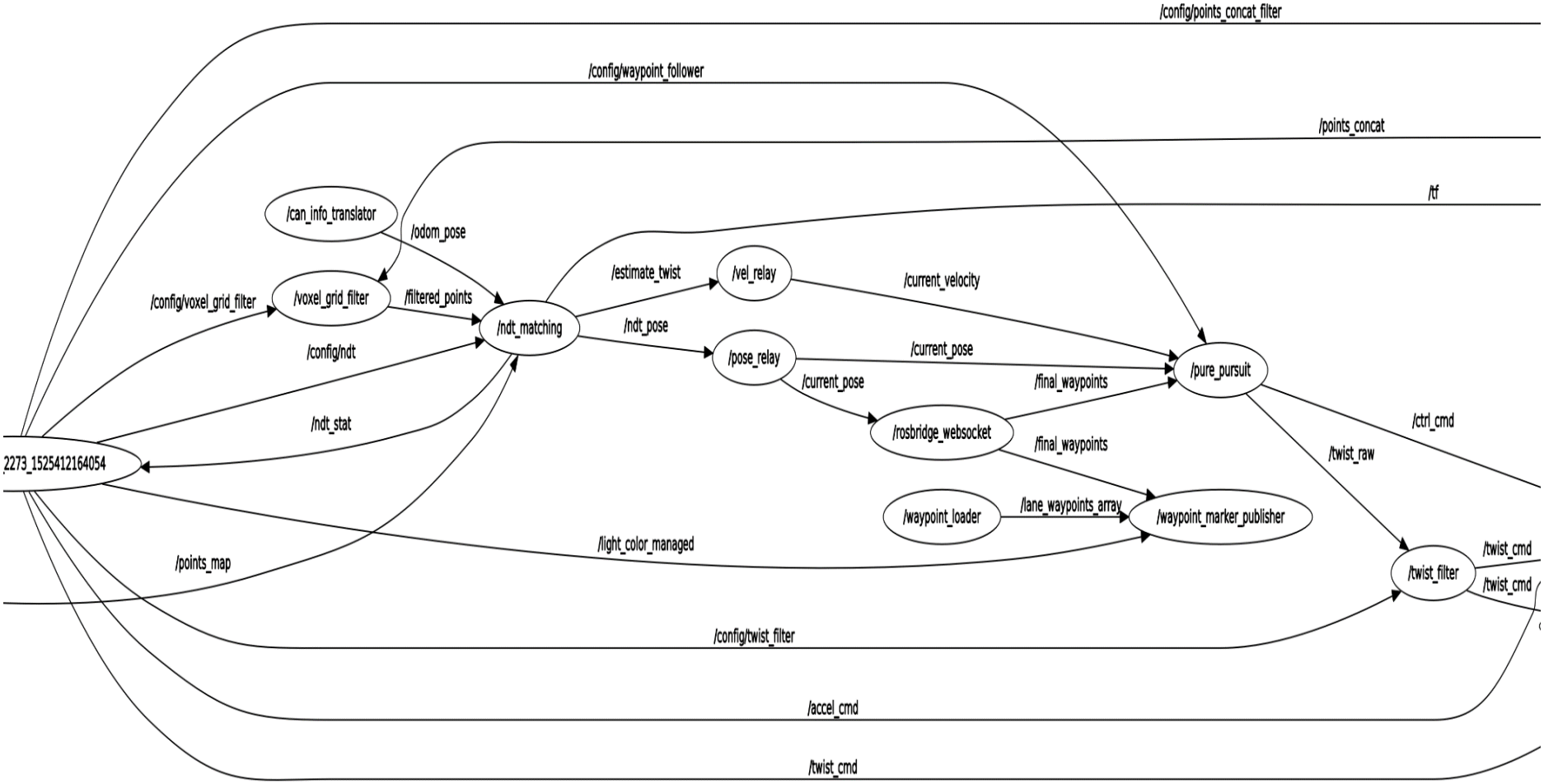
[24] „Närvivõrkude ja masinõppe sõnastik,“ [Võrgumaterjal]. Available: http://datasci.ee/masinoppe-sonastik/. [Kasutatud 28 March 2018].

[25] „Dan North & Associates,“ [Võrgumaterjal]. Available: https://dannorth.net/introducing-bdd/#translations. [Kasutatud 24 04 2018].

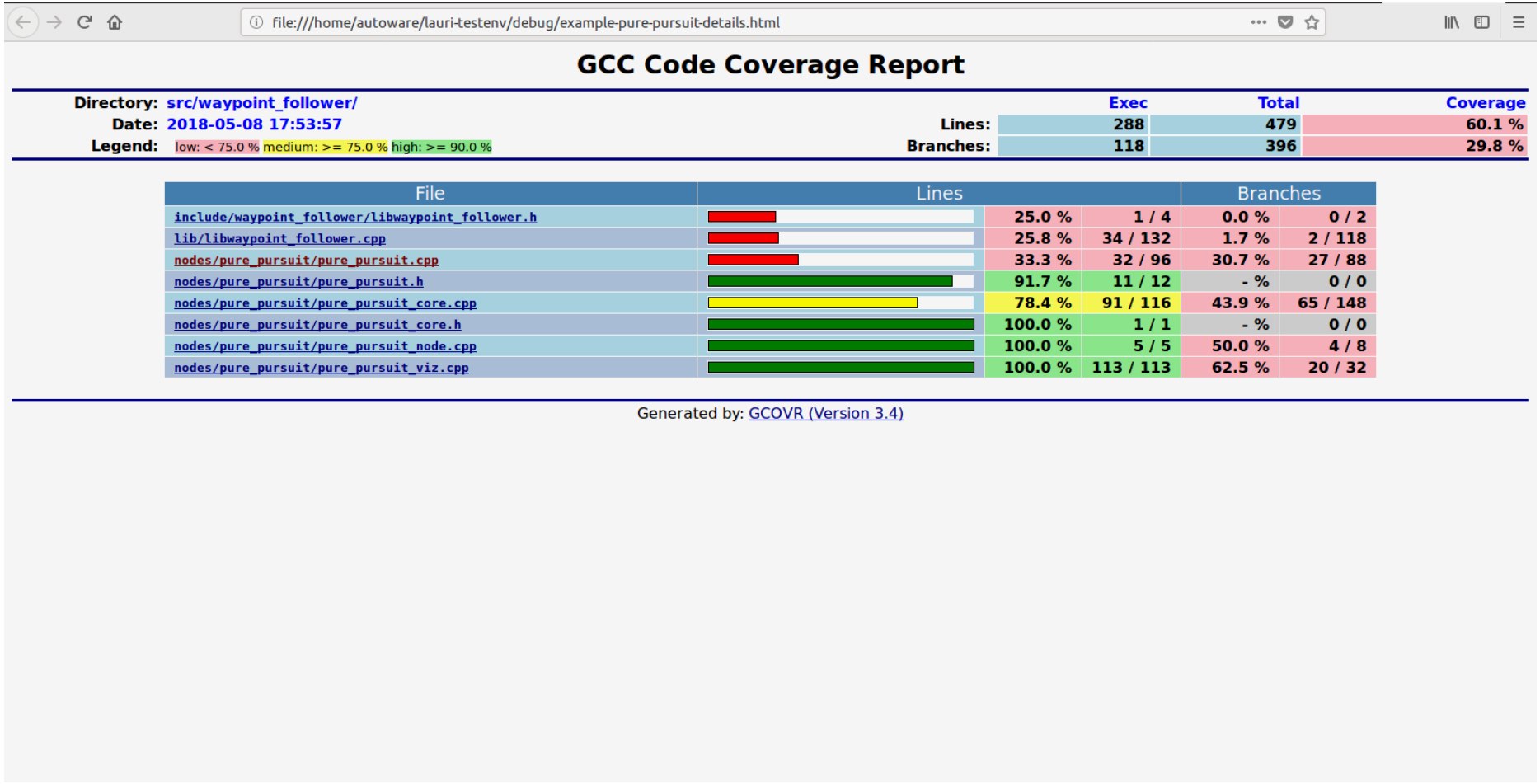
[26] „gcov,“ [Võrgumaterjal]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html. [Kasutatud 15 03 2018].

[27] „gcovr,“ [Võrgumaterjal]. Available: https://gcovr.com/. [Kasutatud 1 05 2018].

Lisa 1 – Autoware'i süsteemiga auto juhtimiseks kasutavate sõlmede ning kanalite graaf



Lisa 2 – Koodikatvuse raport pure pursuit sõlmest



file:///home/autoware/lauri-testenv/debug/example-pure-pursuit-details.html

GCC Code Coverage Report

			Exec	Total	Coverage
Directory:	src/waypoint_follower/	**Lines:**	288	479	60.1 %
Date:	2018-05-08 17:53:57	**Branches:**	118	396	29.8 %
Legend:	low: < 75.0 % medium: >= 75.0 % high: >= 90.0 %				

File	Lines			Branches	
include/waypoint_follower/libwaypoint_follower.h		25.0 %	1 / 4	0.0 %	0 / 2
lib/libwaypoint_follower.cpp		25.8 %	34 / 132	1.7 %	2 / 118
nodes/pure_pursuit/pure_pursuit.cpp		33.3 %	32 / 96	30.7 %	27 / 88
nodes/pure_pursuit/pure_pursuit.h		91.7 %	11 / 12	- %	0 / 0
nodes/pure_pursuit/pure_pursuit_core.cpp		78.4 %	91 / 116	43.9 %	65 / 148
nodes/pure_pursuit/pure_pursuit_core.h		100.0 %	1 / 1	- %	0 / 0
nodes/pure_pursuit/pure_pursuit_node.cpp		100.0 %	5 / 5	50.0 %	4 / 8
nodes/pure_pursuit/pure_pursuit_viz.cpp		100.0 %	113 / 113	62.5 %	20 / 32

Generated by: GCOVR (Version 3.4)

Lisa 3 – Faili pure_pursuit.cpp koodikatvuse näide

Lisa 4 – Kilpkonna argumendifaili näide

```
#-------------------------
# Kilpkonna näidis
#-------------------------

#
# Viitame DLL-teekidele
#
/r:TurtleModel.dll
/r:StepperImp.dll

#
# Mudeli konstruktor
#
TurtleModel.Contract.Create

#
# Adapteri konstruktor
#
/iut:StepperImp.Stepper.Create

#
# Testide jooksutuse arv ja sammude arv, ehk mitu sammu testis tehakse
#
/runs:10
/steps:30

#
# Timeout
/timeout:6000000

#
# Logifaili määramine, kuhu testi tulemused salvestatakse
#
#/log:testruns.txt

#
# Määrame kas test jookseb edasi pärast ebaõnnestumist (+) või mitte (-)
#
/continueOnFailure-
```

Lisa 5 – Loodud mudelprogrammid, logifailid ja lähtekood

Magistritööle on lisatud „Iseauto.zip“ fail kust on leitavad loodud mudelprogrammid, nende lähtekood ja testide logifailid. Samad töömaterjalid on leitavad github salvest aadressil https://github.com/neira24/IseautoMBT. Töö materjalid on leitavad järgnevatest kaustadest:

- Logifailid asuvad kaustas nimega „Tulemused“
- NModel’i DLL-teegid ja vahendid asuvad kaustas nimega „bin“
- Mudeli ja adapteri kood asuvad kaustas nimega „iseautoMudel“
- Kompileeritud mudel ja adapter asuvad „Iseauto“ kaustas nimega „Test“
- Koodikatvuse tulemused ja kasutatud sõlmede graaf asuvad kaustas nimega „Koodikatvus_Pildid“