TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Elgar Lepp 152995IAPM

AUTONOOMSE ROBOTI KÕRGJUHTIMINE

magistritöö

Juhendaja: Gert Kanter
Tehnikateaduse magister

Tallinn 2017

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: Elgar Lepp

08.05.2017

Annotatsioon

Töö eesmärgiks on jõuda seisu, kus antud töö käigus valminud programm võimaldab luua robotile plaani, mille järgi on tal võimalik iseseisvalt keskkonnas ringi liikuda, kasutades selleks minimaalselt väljaspoolset abi.

Töö põhiprobleemiks on leida programmile kiire ja paindlik sisemine struktuur, millele oleks võimalik juurde lisada ka hierarhia süsteem. Lisaks sellele on vaja veel lahendada alamprobleem kuidas valitud struktuuris luua andmevahetus erinevate süsteemi osade vahel.

Töö tulemusena valmib keskkond, kus kasutajal on võimalus defineerida erineva hierarhiaga tegevusi, et muuta robot autonoomseks. Valmis ka viis, kuidas erinevad süsteemi osad saavad oma vahel andmeid vahetada ja oma muutujaid jagada. Lisaks sellele luuakse veel ka süsteemi töö jälgija, mis annab kasutajale teada, kui mingi süsteemi osa on enneaegselt lõpetanud oma töö.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 37 leheküljel, 10 peatükki, 9 joonist, 1 tabelit.

Abstract

Mission control for autonomous robots

Thesis' main goal is to develop a software package, where user can define a complex set of behaviours for a robot using the program which was developed using methods described in here. By using this program, the robot can move around in the environment by itself, requiring minimal outside help.

Thesis's main problem is to find fast and flexible data structure. Parts of the chosen structure need to have a hierarchial information. The second problem is to design an effective communication system between different parts of the system.

As a result of this thesis, an environment is created where user can define actions with different hierarchies to make the robot more autonomous. Additionally, a method to communicate and share variables between different parts of the system was also created. Lastly, a supervisor program was created which observes the system and notifies the user when some parts of the system have stopped working unexpectedly.

The thesis is in Estonian and contains 37 pages of text, 10 chapters, 9 figures, 1 tables.

Lühendite ja mõistete sõnastik

CLARAty	Coupled Layered Architecture for Robotic Autonomy
SMACH	State Machine
ROS	Robot Operating System
Travis CI	Travis Continuous Integration
OSRF	Open Source Robotics Foundation

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Roboti autonoomsus ehk roboti iseseisev töötamine on robootika valdkonnas üks tähtsamaid aspekte. Käesolevas töös luuakse keskkond kõrgjuhtimise loomiseks autonoomsele robotile. Töö eesmärgiks on jõuda seisuni, kus käesolevas töös valminud programmi abil on võimalik luua robotile plaan, mille järgi on robotil võimalik iseseisvalt keskkonnas ringi liikuda, kasutades selleks minimaalselt väljaspoolset abi.

Programmi loomiseks tuleb esmalt välja uurida ja otsustada, millist struktuuri süsteemis kasutada, nii et see sobiks ka hiljem valitud teekidega. Tuleb jälgida, et valitud struktuur oleks kiire ja paindlik ning kasutaja saaks struktuuri kasutades kasutusmalli jaoks vajalikke funktsionaalsusi implementeerida. Järgmisena tuleb leida viis, kuidas valitud struktuuris luua hierarhia, et roboti tähtsamad tegevused saaksid vabalt ja segamatult tegutseda. Viimasena on tarvis välja selgitada, kuidas oleks kõige mõistlikum valitud struktuuris teha andmevahetus erinevate süsteemi osade vahel.

Ülesande lahendamiseks kasutatakse programmeerimiskeeli Python ja C++ ning vahevara *ROS*. Valminud tarkvara testitakse kasutades Pythonis olemas olevat teeki *unittest* ja *ROS* vahevaras olevat programmi *rostest*. Kasutusmall luuakse keskkonnas *Gazebo*, millel on olemas ka liidestus vahevara *ROS*.

Töö teostati 2017. aasta esimeses ja teises kvartalis Tallinna Tehnikaülikooli Tarkvarateaduse Instituudis.

2 Taust ja varasemalt tehtud tööd

Iga firma eesmärk on teenida kasumit, seetõttu otsitakse pidevalt viise kuidas enda konkurentidest parem olla. Viimasel ajal on järjest enam hakatud panustama robootika valdkonda, mis võimaldab igapäeva töid automatiseerida. Robotite arendamine on küll kallis kuid laiemas perspektiivis on robotite töös hoidmine tihti odavam ja efektiivsem kui inimeste palkamine. Kuna robotite autonoomsus on väga varajases staadiumis, siis ei ole väga palju vabavaralisi lahendusi, mida saaks robotite arenduses kasutada. Seetõttu tekkis vajadus vabavaraliste tööriistade järele. Robootika autonoomsuse kohta on aga olemas mitmeid erinevaid teadustöid, mida üritatakse käesolevas töös ära kasutada. Vajadus käesoleva töö järgi tekkis autonoomse traktori kasutusmalli jaoks sobiva vabavaralise kõrgjuhtimistarkvara puudumise tõttu. Selline tarkvara võimaldab luua mitmetest moodulitest koosnevaid juhtimisloogikaid ja seetõttu on võimalik luua robotitele keerukaid käitumismustreid ilma individuaalsete moodulite keerukust oluliselt tõstmata.

Käesoleva töö käigus luuakse süsteem, mis võimaldab planeerida roboti tegevust erinevates moodulites. Hoides moodulite keerukust võimalikult madalal on võimalik luua roboti ülesandele sobiv juhtimisloogika, mis on komponentidena oluliselt lihtsam, kui see oleks näiteks üheainsa olekumasina või muu juhtimisloogikana realiseeritud.

2.1 Autonoomsuse struktuurid

Tavalised autonoomsed roboti arhitektuurid koosnevad kolmest kihist: planeeriv-, täidesaatev- ja funktsionaalne kiht. Planeerimise kihis toimub kõrgetasemeliste plaanide koostamine, kasutades selleks keerukaid planeerimise tehnikaid. Üldiselt võivad need algoritmid olla aega ja ressurssi nõudvad ning seega ei saa see kiht kohe igale muutusele reageerida. Täidesaatev kiht tegeleb planeerimise kihi poolt loodud plaanide täitmisega. See kiht peab ka jälgima robotit plaani täitmise käigus ning tegelema probleemidega kui need esile kerkivad. Selle tõttu on see kiht ka kiirem ja reageerib muutustele kiiremini

kui planeerimise kiht. Funktsionaalne kiht suhtleb robotiga otse reaalajas ning ütleb, mida robot tegema peab [3].

Sellise mudeli üks probleemidest on, et igal tasemel on oma nägemus ja pilt süsteemist, mis tuleb luua ja roboti töö käigus uuendada. Selline tegevus võib tekitada päris suure töö koormuse. Teiseks probleemiks on erinevate kihtide töötamine erinevate detailsuse tasemel, planeerimine kõrgeimal tasemel, täidesaatev järgmisel jne. Neid probleeme lahendab *CLARAty* arhitektuur. See arhitektuur on kahe kihiline, esimeses kihis on kokku pandud planeerimis- ja täidesaatev kiht ning teises kihis eraldi jääb funktsionaalne kiht. Planeeriva- ja täidesaatva kihi ühendamise põhjuseks oli tõusev trend, kus planeerijal on täidesaatvad omadused ja plaanide täidesaatmise kihil on planeerija omadusi. Samas lubab *CLARAty* arhitektuur igal kihil töötada erineval detailsuse ja abstraktsuse tasemel [3]. Käesolev arhitektuur lubab osalist kattuvust mõlema kihi töös. Lisaks võib ka planeerimise-, täidesaatmise kiht pöörduda funktsionaalsuse kihi poole erinevatel detailsuse tasemetel, mis laseb funktsionaalset kihti üles ehitada kõrgemal abstraktsuse tasemel [4].

2.2 Autonoomsuse strateegiad

Autonoomse roboti kõrgjuhtimiseks on välja mõeldud mitmeid erinevaid strateegiaid. Iga uue strateegiaga, kas lahendatakse mõni uus probleem, mõni vana strateegia puudujääk või siis pannakse kaks strateegiat kokku võttes mõlemast ainult positiivsed aspektid.

Puhast geomeetrilist teekonna planeerimist teada olevates keskkondades nimetatakse arutlevaks või kaalutlevaks strateegiaks. Selle strateegia plussiks on asjaolu, et kuna sellel on olemas kaardi punktide võrgustik, siis nende vahel on kindlasti võimalik leida optimaalne teekond. Lisaks sellele saab ka tagada, et teekond ühe punkti juurest teise on kokkupõrgete vaba. Sellise lähenemise miinuseks on, et kogu keskkonna mudel peab planeerijale teada olema, mis suurem osa reaalsetes rakendustes on väga keerukas või isegi võimatu [5].

Eelnevalt kirjeldatud strateegiale täiesti vastanduv on reageeriv strateegia. Teekonna planeerimiseks läheb vaja kõige uuemaid andmeid anduritelt ning nende järgi leitakse

reaalajas optimaalne teekond sihtpunkti. Sellise strateegiaga on töö käigus muutuste tegemine võrdlemisi kiire, kuna arvestatakse kohalike andurite andmeid. Miinuseks on ümbritseva keskkonna mudeli mittetäielikkus, andmete juhuslikkus ja ka plaani täideviimise ebaõnnestumine. Kõigi nende punktidega tuleb arvestada ning töö käigus üritada neid vältida või parandada. Teiseks suureks miinuseks on asjaolu, et robot võib sattuda umbseisu, kuna tal ei ole tervet kaarti ümbritsevast keskkonnast vaid väga väike osa eesseisvast lõigust [5].

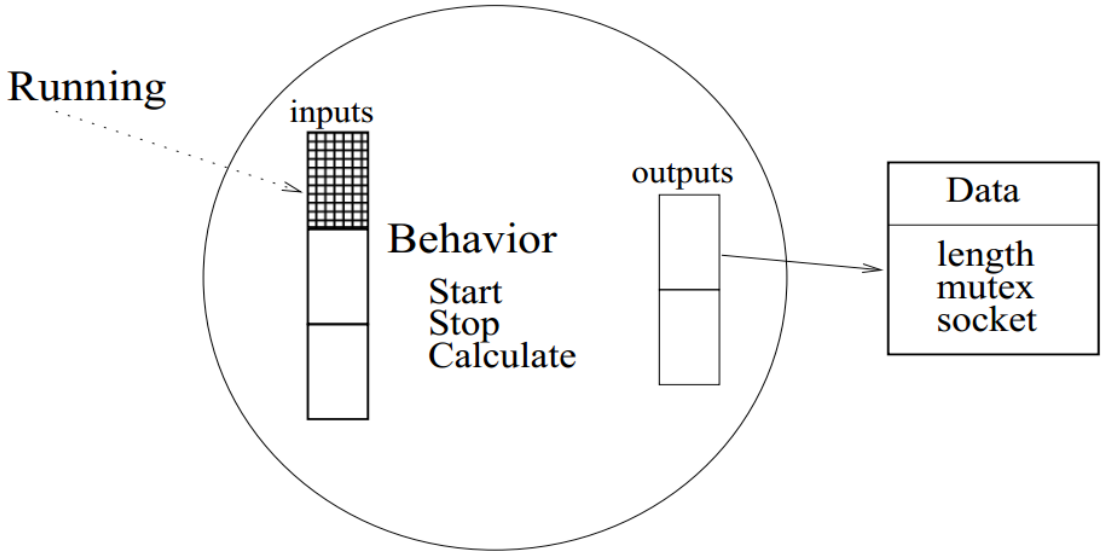
Hübriidses strateegias tuuakse kaalutlev ja reageeriv strateegia kokku ja üritatakse lahendada konfliktid nende vahel [2]. Kui hübriidses strateegias hakatakse ühest punktist teise liikuma, siis see eesmärk jaotatakse mitmeks alameesmärgiks, mida saab ükshaaval täita. Nende alameesmärkide täitmise käigus saab ümbritsevat keskkonda jaotada kahte kategooriasse, kas täielikult modelleeritud keskkond või siis mittetäielikult modelleeritud keskkond, vastavalt sellele, kas teele jääb objekte või takistusi, mis puuduvad robotile teada olevast keskkonna mudelist. Kui alameesmärk on täielikult modelleeritud keskkonnas, siis on lihtne välja arvutada robotile käsklusi, kuidas alameesmärk täita. Kui aga tegemist on mittetäielikult modelleeritud keskkonnaga, siis lisaks välistele sensoritele peab robot kasutama ka sisemisi andureid ja välja arvutama, kas käesolevat alameesmärki on võimalik täita või peab välja arvutama uue alameesmärgi [5].

Viimasena on veel roboti autonoomsuse loomiseks olemas käitumise põhilist strateegia. Sellises strateegias roboti juhtimine on organiseeritud moodulite kogumikuks, kus ühte moodulit nimetatakse käitumiseks. Need moodulid võtavad vastu erinevat informatsiooni nii roboti anduritelt kui ka teistelt moodulitelt. Moodulid samal ajal võivad ise ka andmeid saata roboti riistvarale või/ja ka teistele moodulitele. Kõik moodulid töötavad paralleelselt, samal ajal võttes vastu informatsiooni, töödeldes seda ja saates saadud informatsiooni tagasi kas robotile ja/või teistele moodulitele. Käitumised ise kõik on iseseisvad. Nad teevad omaette tööd lugedes neile mõeldud andureid ja genereerides sisse tuleva info järgi tulemusi. Ühe mooduli jaoks ei ole oluline, kas ka teine moodul tema välja antud infot vastu võtab. See annab võimaluse igat käitumist eraldi arendada ja testida ning hiljem see ühendada suuremasse võrku koos teiste moodulitega [1].

2.3 SORGIN

Käitumise põhilisele strateegiale on ka välja mõeldud raamistik *SORGIN*, mis viib selle strateegia töö ja info vahetamise turvalisemaks ja konkreetsemaks. *SORGIN*-i välja arendamisel võeti eeskujuks Allutatuse arhitektuur ja Van Breemeni raamistik. Allutatuse arhitektuur tähendab endas seda, et iga moodul peaks olema lihtne ja lisatud ükshaaval süsteemi. Van Breemeni raamistik aga väidab et arhitektuur peaks olema hästi struktureeritud, selleks et käitumismooduleid defineerida selgel ja arusaadaval viisil, samal aja lahendades keerukaid roboti juhtimise probleeme. Lisaks nende kombineerimisele täiendati mõlemat arhitektuuri vastavalt looja vajadustele [2].

SORGIN samas hoiab kinni ka tavalistest käitumise põhilistest strateegia punktidest. Selles raamistikus on kõik moodulid eraldi ja töötavad paralleelselt. Selleks ongi igal moodulil muutuja, mis näitab kas vastav moodul töötab või mitte. Seda muutujat uuendatakse iga kord mooduli nullindasse sisendisse saabuva informatsiooni järgi [2]. Üks sõlm *SORGIN*-i struktuurist on välja toodud joonisel 1.



Joonis 1. Raamistiku *SORGIN* üks käitumise sõlm [2]

Andmete vahetamiseks luuakse moodulite vahel ühendused, mis lubavad moodulitel turvaliselt informatsiooni nii lugeda kui ka kirjutada. Sellisel juhul on erinevatel moodulitel ligipääs andmetele kui nad seda vajavad ning ei saa tekkida olukorda, kus kaks erinevat moodulit üritavad samal ajal lugeda ühte ja sama allikat. Andmete vahetamisel puudub kohale jõudmise kinnitus. Kuna kõik moodulid töötavad samal ajal ja erinevatel sagedustel, siis oleks keerukas alati kõigile tagada informatsioon

kinnitusega. See-eest saab iga moodul lugeda informatsiooni, siis kui tal seda juhtimisotsuste tegemiseks vaja on [2].

2.4 SMACH

SMACH on üks kahest teegist, mis on implementeeritud kasutades mingil määral eespool välja toodud strateegiaid. Antud programm on olekupõhine arhitektuur, mille abil on võimalik luua keerulisi käitumise plaane autonoomsetele robotitele. *SMACH*-i käitumised koosnevad erinevatest detailselt defineeritud olekutest ja erinevatest siiretest olekute vahel [7]. Plaani täitmisel täidetakse olekus defineeritud tegevused ning nende tegevuste lõpus iga olek tagastab kas siis ülemineku teise olekusse või siis ülemineku plaani lõpetamiseks.

2.5 *TREX*

TREX on teek, mis jagab automaatse planeerimise väiksemateks ja lihtsamateks juhtivateks tsükliteks. Erinevalt tavalistest kolmetasemelistest arhitektuuridest, on selles teegis ühendatud planeerimine ja plaani täitmine. See väljendub asjaolus, et plaani täitja tihti segab planeerijale vahele, et planeerija oleks alati kursis robotit ümbritseva keskkonnaga [8].

2.6 Loa haldamine mitme roboti süsteemist

Autonoomne robot suurem osa kordadest ei ole suuteline tegema mitut erinevat tegevust korraga. Tihti on valitud tegevus, mis koheselt teostatakse, seotud sellega, mis olekus robot on. Erinevate tegevustele aja jaotamisel võib näite saamiseks uurida, kuidas mitme roboti süsteemid haldavad tegevusi, mida mingi robot peab tegema. Sellistes süsteemides on võimalik lahendada robotite vahelist tegevuste jagamist lubadega. Igal tegevusel on juures oma luba ning kui üks robotitest hakkab tegelema mingi tegevusega, siis ta annab teada, et vastava tegevuse luba on tema käes. Ülejäänud robotid saavad selle informatsiooni kätte ja teavad, et neil ei ole mõtet enam seda tegevust teha ning saavad liikuda uue tegevuse poole. Selline lähenemine vähendab suuresti informatsiooni

hulka ja tihedust, mida robotid peavad omavahel vahetama, et keegi neist ei hakkaks tegelema tegevusega, mida keegi teine juba täidab [6].

3 Loodava süsteemi analüüs

Loodavas süsteemis kavatsetakse kasutada sisemiseks struktuuriks käitumisepõhist strateegiat. Üheks põhjuseks on selle strateegia paindlikkus ja detsentraliseeritus. Teiseks põhjuseks on valitud strateegia struktuuri kokku langevus süsteemi struktuurile, mis on loodud kasutades *ROS* vahevara. Nimelt on võimalik defineerida sõlmi, mis töötavad paralleelselt eraldi lõimedes. Sõlmede vahel on võimalik eelnevalt defineeritud reeglite järgi saata erinevaid sõnumeid.

Süsteemis hierarhia loomiseks piisab kui igale eraldiseisva süsteemi osale anda kindel number mingist ette määratud vahemikust. Määramaks, mis süsteemi osa mingil hetkel töötada saab, kasutatakse variatsiooni eelpool mainitud loa haldamisest mitme roboti süsteemis. Kui mitme roboti asemele võtta erinevad tegevused süsteemis ning jätame ainult ühe loa kogu süsteemi peale, siis see peaks rahuldama meie süsteemi antud ajahetkel olevaid vajadusi.

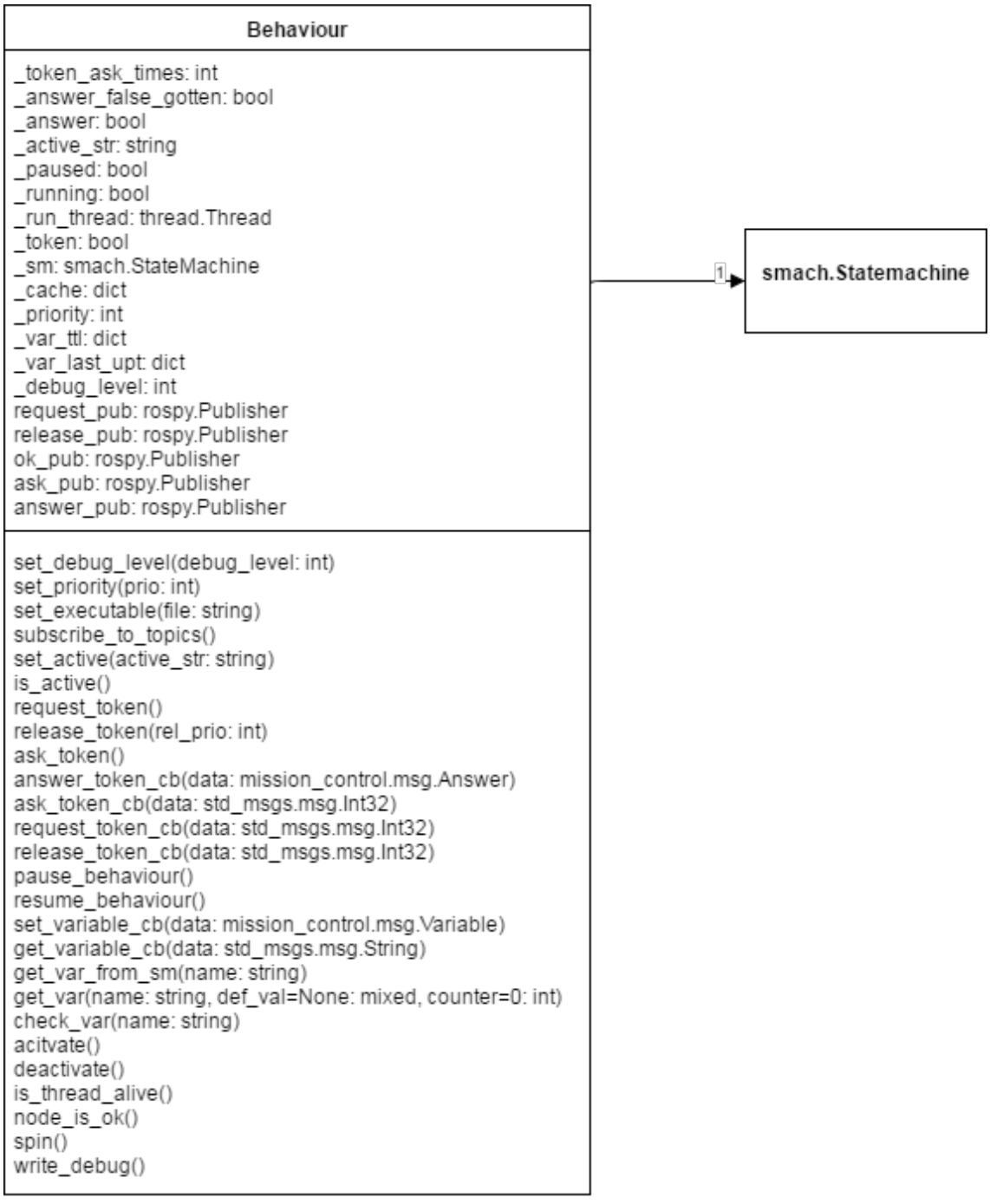
Kommunikatsiooni erinevate süsteemi osade jaoks on plaanis kasutada *ROS* teegi olemasolevat suhtlusprotokolli. Pole mõtet hakata uuesti looma spetsiaalselt meie süsteemi sobivat suhtlusvahendit kuna *ROS* vahevaras on see probleem lahendatud väga lihtsal ja paindlikul moel. Selle abil saavad erinevad süsteemi osad omavahel jagada muutujaid, hierarhia tasemeid ning pärida ja loovutada lubasid üksteisele. Lahendada on vaja ainult probleem, kuidas süsteemi osad hoiavad kommunikatsiooni tagajärjel sisse tulnud muutujate väärtusi. Seda on võimalik lahendada kasutades minimaalsuse põhimõtteid. See tähendab, et iga süsteemi osa peaks jätma meelde ainult need muutujad, mida tal reaalselt töö käigus vaja läheb. Selline lähenemine aitab säilitada süsteemi kiirust muutustele ning ka kokku hoida süsteemi mälu.

Loodava süsteemi implementeerimiseks tuleb esmalt realiseerida struktuur käitumispõhist strateegiat järgides. Järgmisena tuleb luua süsteemide vahel lubade andmine ja võtmine. Järgides tavalist lubade andmise loogikat, tuleb ka implementeerida loa jagamine süsteemi alguses. Seejärel ära defineerida kuidas ja mis

infot süsteemi osad omavahel vahetavad ning luua see võimalus. Viimasena kirjutada ka eraldiseisev programm, mis aitaks kasutajal jälgida süsteemi tööd ning annaks märku kui süsteemis mingi viga tekib.

4 Käitumise sõlm

Üks käitumise sõlm hoiab endas ühte kindlat roboti tegevust või käitumist. Lisaks sellele on igal käitumise sõlmel olemas prioriteet ja ka tingimused, mis peavad olema



Joonis 2. Klassi Behaviour klassiskeem

täidetud, et vastav käitumine saaks aktiveeruda. Ühel robotil saab olla korraga üks kuni N sõlme, kus N tähistab mingit naturaalarvu. Iga sõlm töötab eraldi lõimes ehk siis kõik sõlmed töötavad paralleelselt. Ühelgi sõlmel ei ole kindlat infot ühegi teise sõlme kohta, kas mingi teine sõlm üldse eksisteerib, hetkel töötab või mis tegevust kindel sõlm peab tegema. Käitumise sõlme funktsionaalsus on ära defineeritud klassiga Behaviour, mille klassiskeem on illustreeritud joonisel 2.

4.1 Sõlme prioriteet

Kõik tegevused robotil ei ole võrdse tähtsusega. Mõnda tegevust teeb robot koguaeg, aga see ei ole väga tähtis. Samas võib olla mõni tegevus, mida robot teeb mõne hetke, aga on niivõrd tähtis, et tal ei ole aega, et ära oodata kuna hetkel käimas olev tegevus lõpetaks. Nagu eespool ka mainitud, siis tegevused üksteisest midagi ei tea ja nii võibki jääda kriitiline tegevus ootama vähe tähtsa tegevuse taha, mis tegelikult kunagi ei lõppe. Sellise probleemi vältimiseks on tarvis sõlmede vahel tekitada mingisugune hierarhia. Selle probleemi lahendamiseks antakse igale käitumise sõlmele prioriteedi number vahemikus 1..N, kus N tähistab mingit naturaalarvu. Mida madalam prioriteedi number on, seda kõrgema tähtsusega sõlm on. Sellisel juhul sõlm prioriteedi numbriga 1 on kõige kõrgema tähtsusega ning sõlm prioriteediga N on kõige madalama tähtsusega. Juhul kui robotil on tegevusi, mis saaksid või peaksid samal ajal tegutsema, siis saab nendele tegevustele anda sama väärtusega prioriteedi numbri. Sellisel juhul kui soovib tegutseda sõlm, mille prioriteet on sama väärtusega kui hetkel tegutseva sõlme prioriteet, siis lubatakse tegutseda soovival sõlmel hetkel tegutseva sõlmega paralleelselt töötada.

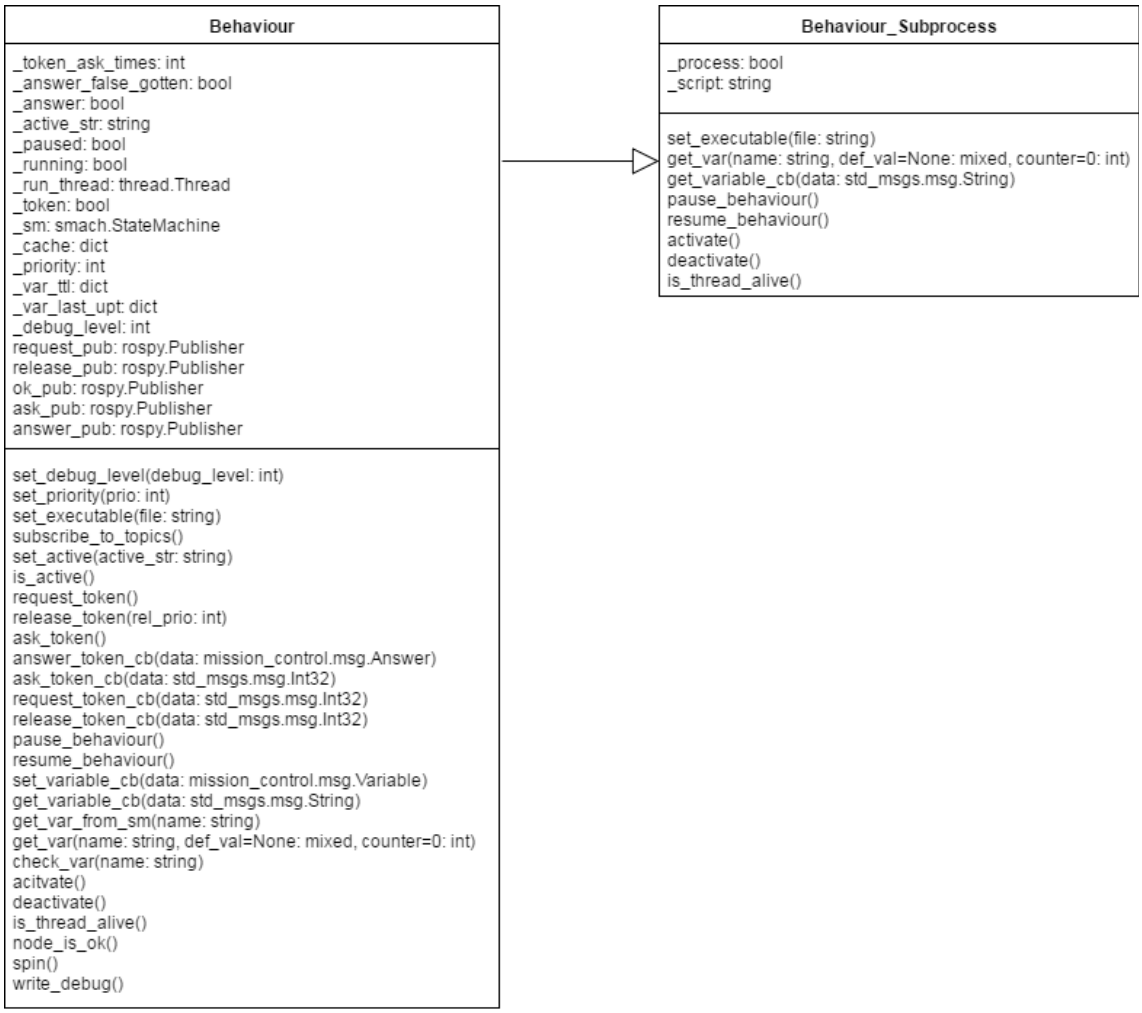
4.2 *SMACH* skriptid sõlmes

Üks võimalus defineerida tegevus, mida sõlm hakkab aktiveerudes tegema on kasutades *SMACH* objekti. Sõlmele on võimalus ette anda faili asukoht, kus asub kasutaja poolt defineeritud *SMACH* objekt. Sõlm otsib antud failist globaalsel tasemel muutujat, mis oleks StateMachine klassi objekti tüüpi. Kui sellist tüüpi objekt leitakse, siis pannakse see sõlmes eraldi muutujasse, et seda oleks võimalik hiljem vajadusel aktiveerida.

SMACH objekt pannakse tööle eraldi lõimes, kutsudes objektil välja funktsiooni *execute()*, et see ei takistaks sõlme enda edaspidist tööd.

4.3 Isetehtud skriptid sõlmes

Teine võimalus defineerida tegevust, mida sõlm saab aktiveerudes käivitada on kasutades isetehtud skripte, mis on kirjutatud kasutades programmeerimiskeeli Python või C++. Sõlmele antakse ette faili teekond, kus sees isetehtud skript asub. Sõlme aktiveerudes pannakse skript tööle alamprotsessis, et sõlm ei peaks oma töö jätkamiseks ootama kuni skript lõpuni jookseks. Kuna on erinevusi, et kas panna eraldi lõimes tööle objekti funktsioon või siis eraldi protsessis terve skript, siis isetehtud skriptide haldamiseks on tehtud eraldi sõlme klass Behaviour_Subprocess, mille klassiskeem on illustreeritud joonisel 3.



Joonis 3. Klassi Behaviour_subprocess klassiskeem

4.4 Sõlme aktiveerumine

Mainiti juba eespool, et kõik sõlmed töötavad paralleelselt, aga korraga saavad oma tegevust täita ainult sama prioriteediga sõlmed. Ülejäänud aja kui sõlmed oma tegevust ei täida on nad lihtsalt tühikäigul. Selle tühikäigu jooksul annab sõlm teada, et sõlme protsess on aktiivne. Järgmisena kontrollib sõlm temale määratud tingimust, mis laseks tal oma tegevusega alustada. Võimalik on määrata, kui tihti sellist kontrolli tehakse, aga vaikimisi tehakse seda kontrolli kaks korda sekundis. Sõlmele määratud tingimus peab olema sõne, mida on võimalik hiljem kasutatavas programmeerimise keeles välja arvutada tõeväärtusena. Selles tõeväärtuse tingimuse sõnes saab kasutada peale sisseehitatud muutujate ja andmetüüpide nii selle sama sõlme SMACH skripti või isetehtud skripti muutujaid kui ka teiste töötavate sõlmede SMACH skriptide või isetehtud skriptide muutujaid.

5 Lubade haldamine erinevate sõlmede vahel

5.1 Lubade vajadus

Nagu eespool mainitud, sai igale sõlmele antud prioriteet vahemikus 1..N, mis märgib ära sõlmes kirjeldatud käitumise tähtsuse teiste sõlmedega võrreldes. See aitab luua hierarhia sõlmede seas. Kuidas aga määrata ära, millisel prioriteedil on õigus hetkel töötada ja aktiveeruda. Selle probleemi lahendamiseks võeti sõlmedes kasutusele load. Enne aktiveerumist peab iga sõlm taotlema teiste sõlmede käest loa. Ta peab taotlema luba seni, kuni luba omav sõlm annab talle nõusoleku ja loobub oma loast. See ei pruugi juhtuda esimesel päringul, kuna luba omav sõlm võib olla kõrgema prioriteediga, kui luba taotlev sõlm ning luba omav sõlm ei pruugi oma tegevusega veel valmis olla. Tuleb meeles pidada, et sõlmed ei tea, kelle käes luba mingil hetkel on, kuna neil puudub üksteise kohta info. Seega tuleb loa taotlemine saata ühte kindlasse kanalisse, mida kõik teised süsteemis olevad sõlmed kuulavad ning oodata, et luba omav sõlm vastaks.

5.2 Lubade jaotus süsteemi töö alguses

Süsteemi töö alguses ei ole mitte ühelgi sõlmel luba tegutseda. Üks lahendus sellele probleemile oleks, et kasutaja ise määrab ära sõlme, kes saab töö alguses loa endale. See on aga võrdlemisi tülikas ning võib tekitada probleeme, kui kasutaja unustab selle määrata ja mõtleb, miks ükski sõlm süsteemi töö alguses ei alusta oma tegevusega kuigi sõlme aktiveerumise tingimused on täidetud. Selle probleemi lahendamiseks tuleks sõlmedel omavahel ära jaotada, kes süsteemi töö alguses loa saab. Ühe võimalusena võiks loa anda sõlmele, kes kohe peale süsteemi töö algust oma tegevusega alustab. See aga võib olla kahjuks ebaefektiivne kuna ükski sõlm ei pruugi kohe peale süsteemi algust oma tegevusega alustada. Kui kasutada seda meetodit, siis tuleb jälgida, et kui mitu sõlme soovib kohe oma tegevusega alustada siis milline neist on kõige kõrgema prioriteediga. Kõik selle välja selgitamine võib võtta üpriski kaua aega. Palju lihtsam

oleks kui iga sõlm küsib töö alguses läbi ühise kanali, et kas tema võiks saada süsteemi töö alguses loa tegutseda, saates igale sõlmele oma prioriteedi numbri. Peale sellist küsimist saab sõlm ise iga küsimise peale vastata, kas küsiv sõlm võib endale loa võtta või mitte. Vastus, kas sõlm võib loa võtta endale või ei, sõltub sellest, kas küsiva sõlme prioriteet on kõrgem, kui sõlme kelle käest küsitakse ehk siis kui küsiva sõlme prioriteet on kõrgem, siis antakse talle õigus luba endale võtta. Juhul kui sõlm saab kasvõi ühe negatiivse vastuse loa taotlemisele, siis ta enam uuesti luba ei taotle. Kasvõi ühe eitava vastuse saamine näitab, et süsteemis on üks sõlm kelle prioriteet on kõrgem. Sellist küsimist tehakse 2 korda enne kui sõlm saab endale loa võtta. Mitmekordne küsimine on vajalik kuna kõik sõlmed ei pruugi kohe süsteemi töö alguses loa taotlemissõnumeid vastu võtta.

5.2.1 Lubade jaotus süsteemi töö alguses sama prioriteediga sõlme puhul

Süsteemi töö alguses loa küsimisel mitme sama prioriteediga sõlme puhul toimib kõik nii nagu tavaliselt. Kui tegemist on mitme sõlmega, millel on samaväärne madal prioriteet, siis nad saavad teistel sõlmedelt sarnaselt tavalise käitumisega negatiivse vastuse ja kumbki neist uuesti luba ei küsi. Juhul kui on kaks või enam sõlme kel on kõige kõrgem prioriteet, siis peale loa taotlemist mõlemad sõlmed saavad ainult positiivseid vastuseid. Sõlm saab ka positiivse vastuse juhul kui ta küsib luba teise sõlme käest kellel on samaväärne prioriteedi tase.

5.3 Lubade jagamine süsteemi töö käigus

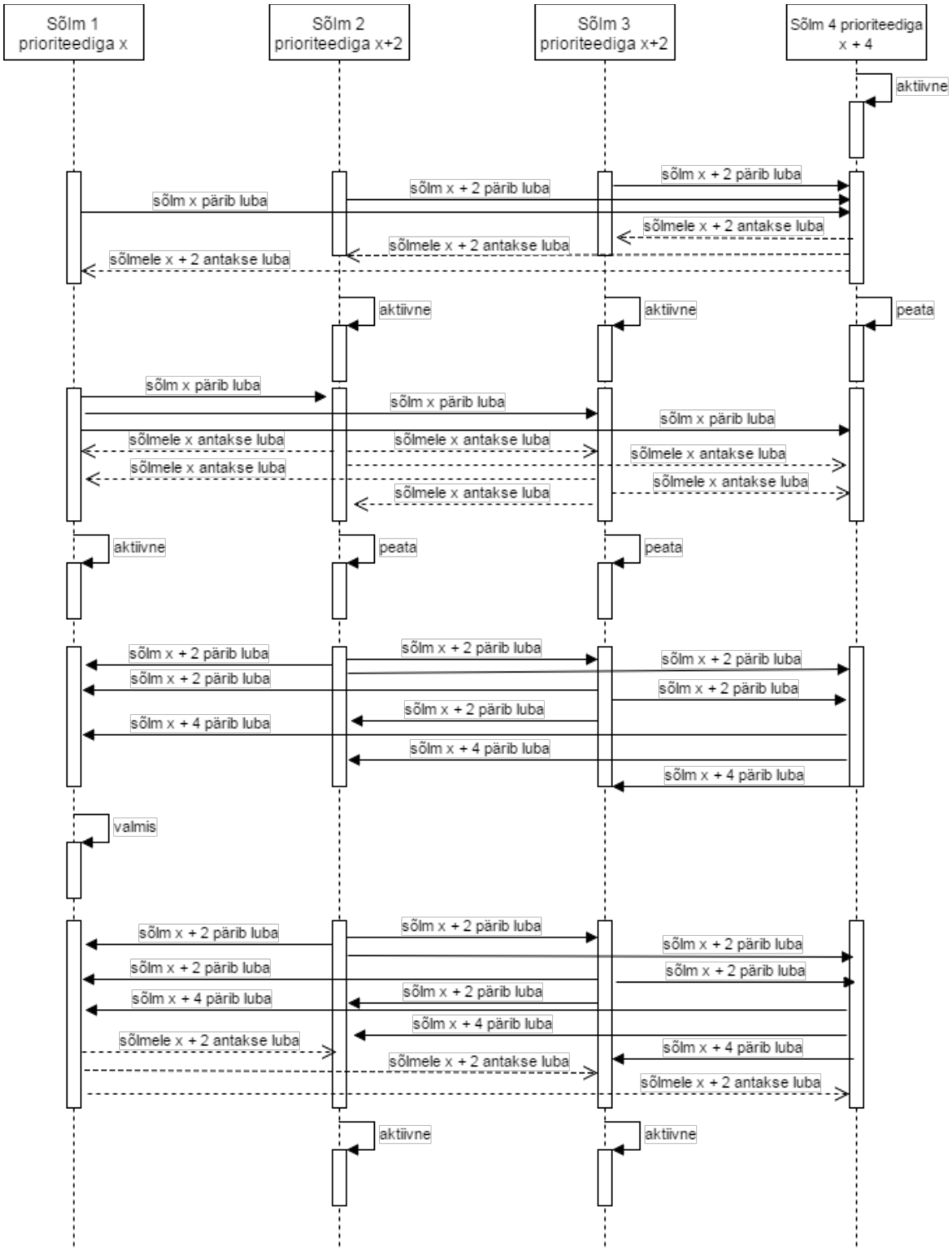
Süsteemi töö käigus kui sõlm aktiveerub ja tal puudub luba töötada, siis peab ta seda teiste sõlmede käest küsima. Selleks saadab luba sooviv sõlm oma prioriteedi numbri läbi ühtse kanali, mida kõik teised sõlmed kuulavad. Juhul kui luba omav sõlm on madalama prioriteediga või ta ei ole aktiivne, loobub ta oma loast ja annab teada, millise prioriteediga sõlmele luba loovutatakse. Kui luba küsitakse sõlme käest, kellel on madalam prioriteet ja aktiivne tegevus on pooleli, siis peab see sõlm oma tegevuse peatama ja loa loovutama. *SMACH* objekti puhul tehakse paus klassi uuendamise funktsioonis. Isetehtud skriptide puhul, mis pannakse alamprotsessides tööle, peatatakse nad SIGSTOP signaaliga ning jätkatakse SIGCONT signaaliga. Samal ajal kui sõlme

aktiivne tegevus on pausi peal, pärib ta pidevalt luba endale tagasi. Sellisel juhul saab ta oma tööga niipea jätkata kui kõrgema prioriteediga sõlm on oma töö lõpetanud ja enam luba ei vaja.

5.3.1 Lubade jagamine süsteemi töö käigus sama prioriteediga sõlmede puhul

Korduvalt on juba mainitud, et süsteemis olevad sõlmed ei tea üksteise kohta midagi. Isegi juhtudel kui kahel või enamal sõlmel on sama väärtusega prioriteet. Selle tõttu kui süsteemi töö käigus küsib sõlm endale luba, siis saavad kõik aktiivsed samaväärse prioriteediga sõlmed endale tegutsemiseks loa. Samas võib tekkida olukord, kus aktiveerub sõlm, kellel on loa omava sõlmega samaväärne eelisõigus. Aktiveerunud sõlm peab kindlasti endale loa saama ning tegutsev sõlm ei tohiks oma tööd pooleli jätta, kuna sama prioriteediga sõlmed peavad saama üheaegselt töötada. Probleemi lahendamiseks luba omav sõlm lihtsalt jagab oma luba, kus sõlm oma luba ei tühista ning aktiivset tegutsemist ei katkesta.

Loa omamine kahe või enama sõlme poolt tekitab probleemi, kus üks sõlmedest lõpetab oma töö varem kui teised ja on valmis andma oma luba ära madalama prioriteediga sõlmele. Seda aga ei tohi juhtuda, kuna siis saavad kaks erineva prioriteedi tasemega sõlme korraga tegutseda, mis on käesolevas süsteemis keelatud. Sellise probleemi lahendamiseks peab iga luba omav sõlm pidevalt luba edasi küsima. Seega kui üks sama eelisõigusega sõlm lõpetabki töö, siis peab ta kohe loa tagastama teistele tegutsevatele sõlmedele. Isegi kui mõni madalama prioriteediga sõlm peaks ette jõudma ja loa endale saama, siis kõrgema prioriteediga sõlmede pidev loa küsimine võtab selle kohe esimesel võimalusel ära.



Joonis 4. Lubade jagamine sõlmede vahel süsteemi töö käigus

Mitme madalama prioriteediga aktiivse sõlme loa ära andmine toimub eespool paika pandud reeglite järgi. Kui kõrgema prioriteediga sõlme küsib luba, siis ta saadab oma prioriteedi suhtluskanalisse, mida kõik süsteemis olevad sõlmed kuulavad. Aktiivsed luba omavad sõlmed saavad selle päringu kätte, seiskavad oma töö ja vabastavad

küsivale sõlmele oma load. Joonisel 4 on ära näidatud, kuidas süsteemis lubade jagamine toimib olukorras, kui luba küsib sõlm, millel on sama prioriteet kui mõnel teisel sõlmel süsteemis ning kui luba küsib sõlm, kellel on ainukesena vastav prioriteet.

6 Sõlmede vahel muutujate jagamine

Käitumispõhine strateegia näeb ette, et sõlmed peavad saama omavahel vahetada erinevaid muutujaid, mis tekivad töö käigus. Neid võib minna tarvis mõne teise sõlme aktiveerimiseks. Teise juhuna võib mõnel kõrgema eelisõigusega sõlmel tarvis teada millises olekus mõne madalama prioriteediga sõlme muutujad on.

6.1 Muutujate saatmine

Muutujate saatmisega peavad kõik sõlmedes olevad *SMACH* olekumasina objektid või isetehtud skriptid ise tegelema. Nimelt nad peavad neid teistele sõlmedele ise saatma, kui need algselt initsialiseeritakse või töö käigus peaksid muutuma. Muutuja teiste sõlmedele saatmiseks tuleb kasutajal välja kutsuda selleks eeldefineeritud funktsiooni, kuhu peab ette andma muutuja nime, mille järgi teised sõlmed saaksid seda pärida, ning muutuja väärtuse. Funktsioon omakorda saadab need väärtused andmevahetuse kanalisse, mida kõik süsteemis olevad sõlmed kuulavad. Käitumispõhine strateegia näeb ette, et saatja ei pea garanteerima, et muutujad kuulavatele sõlmedele kohale jõuavad. Seetõttu muutuja saatmise funktsioon ei oota ära, ega küsi teiste käest, kas nad said muutuja väärtuse kätte, vaid kirjutab selle kanalisse ära ja tagastab juhtimise kasutaja defineeritud tegevusele.

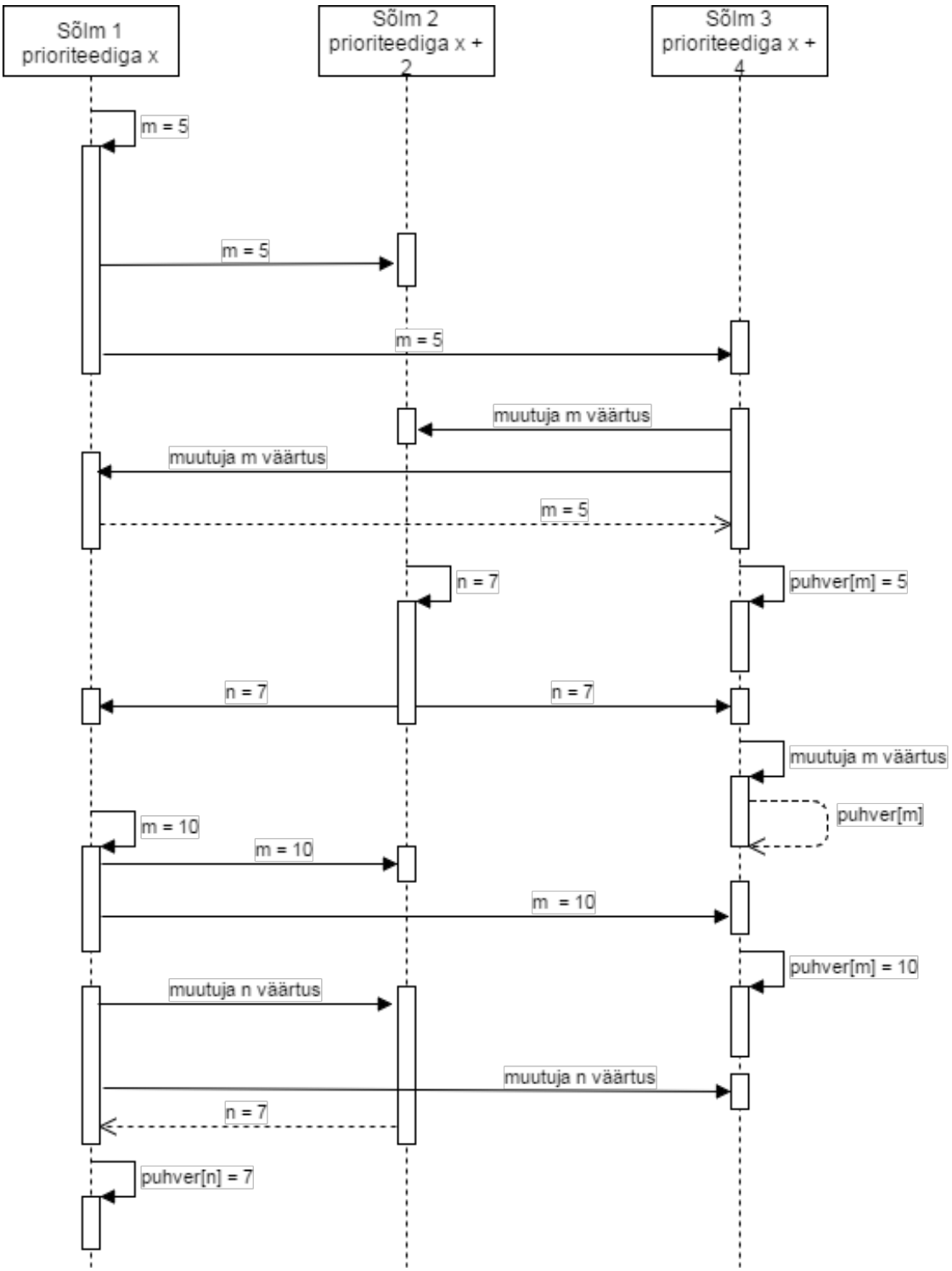
6.2 Muutujate küsimine ja vastuvõtmine

Igal sõlmel on oma puhver, kuhu ta sisse tulnud muutujate väärtuseid salvestab. Sõlme töö alguses ühtegi saadetud muutujat sinna ei salvestata juhul kui seda muutujat sõlme sees oleva tegevuse poolt ei küsita. Kui sõlm on korra ühe muutuja väärtust juhtimisloogikas kasutanud, siis järgnevatel kordadel saadetakse, uuenenud väärtused sellest muutujast, siis uuendatakse seda väärtust sõlme puhvris. Sellise lähenemise põhjuseks on asjaolu, et kui sõlme eluea jooksul kõik sissetulevaid muutujaid

salvestatakse, siis peale pikemat töötamist võib see suureks paisuda ja süsteemi mälu asjata raisata. Selleks salvestataksegi ainult muutujaid, mida sõlmel reaalselt tarvis läheb.

Kui sõlm pärib muutujat, siis esialgu vaadatakse ta enda puhvrisse, ega seal ei ole vastavat muutuja väärtust. Saab kindel olla, et sellise käitumise järel ei saada aegunud väärtust, kuna kui korra muutuja puhvrisse salvestatakse, siis iga kord ka seda uuendatakse, juhul kui muudatus on vastavasse andmevahetuse kanalisse kirjutatud. Kui puhvris küsitud muutujat ei eksisteeri, siis ei oodata kuni vastav muutuja uueneks ja seda uuesti teistele sõlmedele saadetaks. Selle asemel edastatakse päring üle terve süsteemi ning iga sõlm vaatab, ega taolist muutujat tema *SMACH* objektis või isetehtud skriptis ei eksisteeri. Kui keegi selle muutuja enda tegevuses leiab, siis ta saadab selle muutuja meeldetuletusena üle terve süsteemi, mis jõuab ka selle sõlme puhvrisse, kes algselt seda soovis. Muutujate saatmist, küsimist ja vastuvõtmist illustreerib joonisel 5 olev järgnevusskeem.

Muutuja pärimisel on võimalik küsijal ka ära määrata vaikimisi väärtus, mis tagastatakse juhul kui vastavat muutujat ei leita või selle otsimiseks kulub liiga palju aega. Selline käitumine on kasulik kasutajale, kuna ta saab teha muudatusi oma tegevuses või siis muud moodi tööga jätkata, juhul kui muutujat ei leita. Tagastatud vaikimis väärtus salvestatakse ka sõlme puhvrisse. See ei sega muutujate väärtuste korrektsust, kuna kui peaks tulema vastava muutuja uuenemise sõnum, siis leitakse see muutuja puhvrist ning seda uuendatakse. Uuesti ei ole tarvis seda üle terve süsteemi küsida. *SMACH* objekti puhul vaadatakse läbi kõik objekti defineeritud muutujad ning kui tekibki uus muutuja juurde, siis kasutaja peaks ise seda ka süsteemile teatama. Alamprotsessis olevate skriptide puhul saab pärida ainult neid muutujaid, mida on korra üle süsteemi teatatud, seega kui korra muutujat sealt ei leita, siis enne see ei saa tekkida, kui süsteemile sellest ei teatata.



Joonis 5. Muutujate saatmine ja vastu võtmine

6.3 Muutujate kustutamine

Puhvrist muutujaid ei kustutata, juhul kui kasutaja muutujast teatamisel muud moodi ei ütle. Peale muutuja nime ja väärtuse on kasutajal võimalus süsteemile teada anda ka tolle muutuja kõlblikkus sekundites, mille määramine on valikuline. Kui kasutaja ei soovi seda määrata, siis saadetakse selle asemel süsteemile kõlblikkuseks 0, mis näitab, et sellel muutujal ei ole tarvis värskust kontrollida. Sellist kontrolli tehakse iga kord enne kui see võetakse puhvrist. Juhul kui muutuja ongi aegunud, siis see kustutatakse puhvrist, seega sõlm peab seda uuesti üle süsteemi küsima, et olla kindel muutuja õigsuses. See aitab tagada, et sõlmedel on ainult värsked muutujad ja saaksid teada juhul kui tegevus on lõpetanud või tupikusse sattunud.

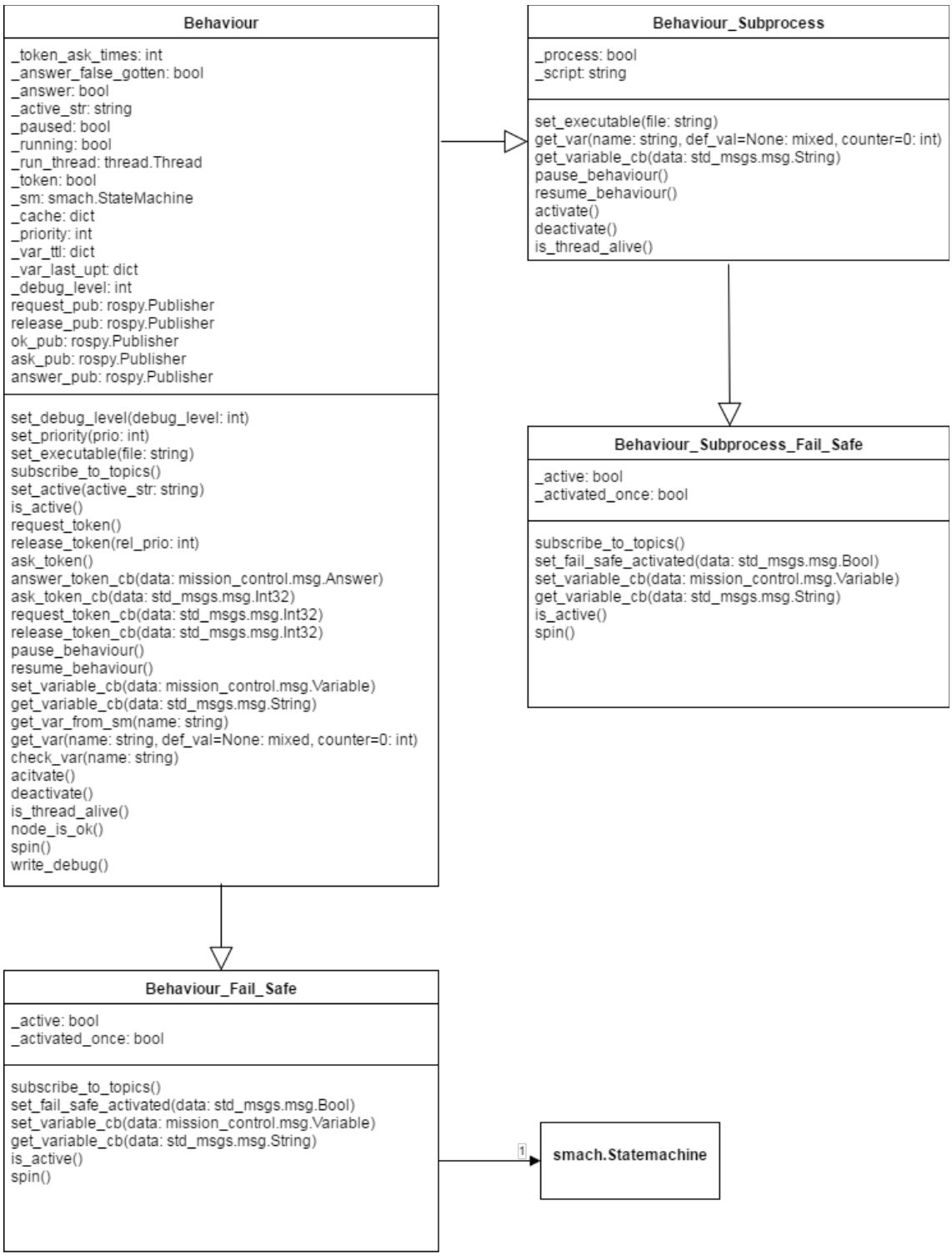
7 Sõlmede töö jälgija

Süsteemi töö käigus võib juhtuda olukordi, kus mõni sõlm lõpetab veaga töö või satub tupikusse. See võib juhtuda ilma veateadet tagastamata või siis kasutajal võib see lihtsalt märkamata jääda. Sellised vead on aga süsteemi korrektseks tööle saamiseks vaja kinni püüda ja parandada. Lahendusena on kasutajal võimalus sisse lülitada süsteemi töö jälgija, mis vaatab, et kõik sõlmed oleksid töökorras.

Iga sõlm saadab 2 korda sekundis välja sõnumi, mis näitab, et protsess on aktiivne. Sõnumis on kirjas sõlme nimi ning kas tal on luba tegutseda või mitte. Sõlmede töö jälgija püüab kõik need sõnumid kinni ning talletab endale aja, millal see jõudis ning kas tollel sõlmel oli luba. Jälgija vaatab 1 kord sekundis, kas kõik sõlmed on endast hiljuti teada andnud. Kasutaja saab ise määrata, mitu sekundit jälgija ootab, enne kui ta kuulutab sõlme protsessi lõpetatuks. Juhul kui sõlmede töö jälgija on korra mõne sõlme lõpetatuks kuulutanud, siis süsteemi vigade vabaks ei kuuluta, isegi kui lõpetatuks kuulutatud sõlm endast uuesti teada annab. Selline käitumine aitab tagada asjaolu, et tekkinud vigadega tegeletakse, et vigu ei ignoreerita.

7.1 Süsteemi tõrkel aktiveeruv sõlm

Süsteemis võib eksisteerida sõlm, mis tegeleb näiteks roboti peatamisega, kui ta on liiga kaugele sõitnud. Kui selline sõlm jääb tupikusse või lõpetab veaga töö, võib robot jäädagi liikuma. Probleemi lahendamiseks on olemas eraldi sõlm, mida saab aktiveerida ainult süsteemi töö jälgija ning mis aktiveerub ainult siis kui süsteemis olev sõlm lõpetab veaga töö või satub tupikusse. Seda sõlme kasutades on võimalik vea tekkides robot välja lülitada või siis lihtsalt robot peatada. Sarnaselt teistele süsteemis olevatele sõlmedel on ka tõrkel aktiveeruvas sõlmes võimalik tegevus defineerida *SMACH* olekumasinana või siis isetehtud skriptina. Mõlema jaoks defineeriti eraldi klass, nende klassiskeemid on illustreeritud joonisel 6.



Joonis 6. Klasside Behaviour_Fail_safe ja Behaviour_Subprocess_Fail_Safe klassiskeemid

Süsteemis vea tagajärjel aktiveeruval sõlmel on prioriteet numbriga null. Kasutajad saavad enda sõlmedele defineerida prioriteete vahemikus 1..N, kus N märgib mingit naturaalarvu, seega saab tõrkel aktiveeruv sõlm alati loa tegutseda. Võib tekkida

olukord, kus vea tagajärjel lõpetab töö sõlm, kellel oli luba. Sellisel juhul isegi kui küsida luba süsteemist, siis keegi ei vasta. Probleemi lahendamiseks, nagu ka eespool mainitud, hoiab süsteemi jälgija endas informatsiooni, kellel viimati oli luba tegutseda. Kasutades seda informatsiooni saab ta sõlme aktiveerides kaasa anda ka teabe, kas tal on luba kohe tegutseda või ta peab seda süsteemi käest küsima. Vaatamata sellele küsib tõrkel aktiveerunud sõlm süsteemilt pidevalt luba. Taoline käitumine on vajalik juhtudel, kus loaga tupikusse sattunud sõlm võib ummikust vabaneda ja üritada oma tööd jätkata. See aitab tagada, et ellu ärganud sõlm loobuks esimesel võimalusel oma loast ja jätaks oma tegevuse pooleli.

7.1.1 Muutujad süsteemi tõrkel aktiveeruvas sõlmes

Süsteemi tõrkel aktiveeruv sõlm salvestab kõik sissetulevad muutujad endale puhvrisse. Vaatamata sellele, et eespool sai mainitud, et selline käitumine raiskab asjatult süsteemi mälu, kuna ei olda kindlat, et salvestatud muutujaid üldse kunagi kasutatakse. Sellist käitumist on tarvis juhul kui peale süsteemi tõrget aktiveeruval sõlmel läheb vaja mõnda muutujat, mis oli defineeritud surnud sõlmes. Teada on, et uuesti küsimisel ei pruugi sõlm vastata, kuna ta võib olla ikka veel tupikus või siis üldse veaga oma töö lõpetanud. Juhul kui kõik muutujad salvestatakse, siis on tõrkel aktiveeruval sõlmel vähemalt viimane väärtus muutujast.

8 Testimine

Süsteemi testitakse kasutades üksuste teste ning integratsiooni teste. Kuna tehtud süsteem töötab ainult tervikuna, siis on raske kõiki funktsioone katta üksuste testidega. Seega kasutatakse selliseid teste ainult klassi funktsioonide peal, mis initsialiseerivad klassi siseseid muutujaid või tagastavad informatsiooni kasutades klassi siseseid muutujaid. Ülejäänud funktsioone saab efektiivselt testida ainult siis, kui süsteemis on rohkem kui kaks sõlme, mis kasutavad ühte ja sama testitavat klassi. Testimise käigus on tarvis initsialiseerida ka *ROS* teegi põhifunktsionaalsused, et kõige paremini simuleerida reaalset töö olukorda. Sellise probleemi lahendamiseks on *ROS* vahevaras olemas programm *rostest*, kus on võimalus ära defineerida, mis tingimustes millised sõlmed tööle pannakse ning Pythoni fail, kus sees on üksuste testimise klass. Testimise käigus pannakse kõik defineeritud sõlmed ja üksuste test paralleelselt tööle.

Integratsiooni testimise käigus on tarvis panna sõlmed kindlatel tingimustel tööle ning oodata kuni nad mingisse kindlasse seisu jõuavad, et testida kas kõik funktsionaalsused töötavad korrektselt. Selleks tuleb teha koopiad skriptidest, mis jooksutavad tsüklis klassi põhifunktsiooni *spin()*. Seejärel saab loodud koopiates saata testidele informatsiooni töötava klassi kohta. Viimasena on vaja testidele saata kinnitus, et kõik vajalik informatsioon peaks olema saadetud. Kuna siin testimisel kasutatakse ka Pythoni üksuste testimise liidest, siis peab igas üksuste testimise klassis olema üks funktsioon, mis tegeleb andmete testimisega. Seda põhjusel kuna üks kindel üksuse testi funktsioon peab teada andma, et nüüd kõik informatsioon on käes ja võib testima hakata.

Testimise käigus kogutakse ka informatsiooni selle kohta, kui suur osa koodist on testidega kaetud. Kuna kõik sõlmed töötavad paralleelselt eraldiseisvates sõlmedes, siis kõigi peal korraga ei saa koodi katvust mõõta. Korraga erinevates lõimedes koodi katvuse mõõtmisel ja selle talletamisel ette antud faili võib tekkida juhus kus kaks või enam sõlme korraga üritavad oma saadud tulemusi faili kirjutada ning üks sõlm kirjutab teise info lihtsalt üle. Sellise vea vältimiseks tuleb iga testimise käigus mõõta korraga ainult ühe sõlme koodi katvust. Võib tekkida olukordi, et ühe integratsiooni testi käigus

kasutab üks sõlm selliseid funktsioone, mida teine sõlm ei kasuta. See võib viia olukorrani, kus koodi katvus jätab välja ridu, mis tegelikult sai läbitud. Olukorra lahendamiseks tuleb teste, kus võib tekkida selline olukord, jooksutada rohkem kui üks kord. Igal jooksutamisel tuleks aga mõõta erineva sõlme koodi katvust. Testide tulemusena saadud koodi katvuse info on välja toodud lisas number 2.

Tagamaks, et koodi repositooriumis oleks koodi korrektsus nähtaval, liidestati kasutatava *Github*-i salv *Travis CI*-ga. Sellise lahenduse puhul kui salves uuendatakse koodi, siis luuakse täiesti uus Ubuntu keskkond versiooniga 14.04. Kahjuks *Travis CI* ei toeta veel Ubuntu versiooni 16.04. Seejärel installitakse vahevara teek *ROS* ilma graafiliste liidesteta ning muud tööks vajalikud teegid. Viimasena luuakse vastavad kaustad, mida *ROS*-il vaja läheb ja lingitakse salves olev kood *ROS* kaustades õigesse kohta. Lõpetuseks käivitatakse kõik testid, mis peaksid teada andma, kas salves olev kood toimib nii nagu vaja.

9 Kasutusmall

Töös kirjeldatud lahendust kasutatakse Tallinna Tehnikaülikooli biorobootika keskuse poolt väljatöötatud allveelaeva roboti peal nimega U-CAT. Hetkel on sellele ära defineeritud ainult väga algelised ringi liikumise sõlmed nagu näiteks takistuste vältimise sõlm, kiiruse reguleerimise sõlm ning sõlm, mis tegeleb pööramise, tõusmise ja sukeldumisega. Allveelaeva roboti esmaseks ja algeliseks eesmärgiks on vees ringi ujuda ja ümbrust uurida.

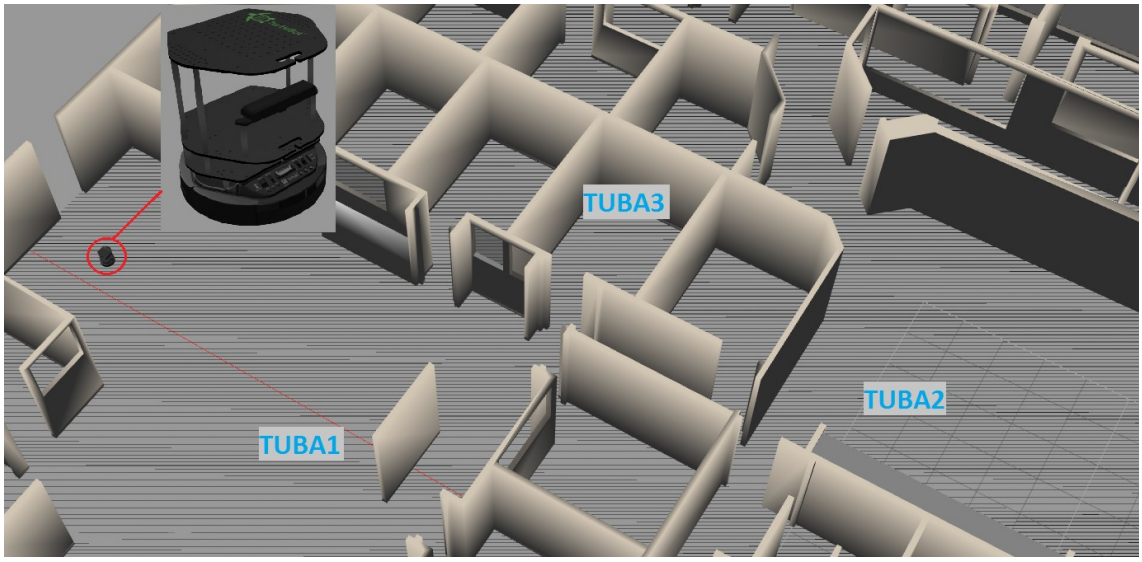
Loodud tarkvara ei ole kindla roboti spetsiifiline ja seega kasutatakse seda ka autonoomse traktori peal. Antud traktorit kasutatakse loomade söötmiseks. Kasutades antud tarkvara saab muuta traktori autonoomseks, seega ta saab ise ümbritsevas keskkonnas ring liikuda ja õigetele kellaaegadel loomadel toitu ette viia.

Joonis 7. U-CAT [9]

Joonis 8. Autonoomne traktor

Eelpool mainitud projektid on töömahukad, eraldiseisvad ja keerukad. Käesoleva süsteemi töötamisest saab kompaktse ülevaate väga lihtsa ja minimalistliku kasutusmalliga simulatsiooni keskkonnas. Sellisel juhul saab keskenduda ainult kõrgjuhtimise teegi funktsionaalustele ning ei pea kogu tervet süsteemi kirjeldama. Simulatsioonis kasutatakse ümbritsevaks keskkonnaks vabavarana saada olev *Willow Garage*-i kontori plaani ning roboti simuleerimiseks kasutatakse *OSRF* ja *Roboti*-se poolt pakutava roboti *turtlebot* plaani. *Turtlebot* plaani saab kasutada ka selle roboti simulatsiooni jaoks mõeldud programme, mis ise hoiavad meeles roboti asukohta ruumis ning ka arvutavad robotile plaani, mille järgi ruumis liikuda.

Pannes tööle programmid, mis hoiab meeles roboti asukohta ruumis ja arvutab roboti liikumiseks plaani, saab simulatsioonis olevale robotile ette anda punkte ruumis, kuhu ta liikuma peab. Lisaks eelnevale tuleks luua ka üks eraldiseisev sõlm, mis simuleeriks roboti akut, mis aja möödudes lineaarselt tühjeneb. Peale selle saab luua veel kaks sõlme, mis on implementeeritud *SMACH*-i olekumasinate peale ja kasutavad töös kirjeldatud lahendust. Esimene sõlm tegeleb sellega, et robot liiguks toast number 1 tuppa number 2, mille asukohad on näha joonisel 9. Teine sõlm tegeleb sellega, et jälgib aku taset, ning kui see on madalam kui 20%, siis viib ta roboti kolmandasse tuppa, kus toimub roboti aku laadimine seni, kuni roboti aku tase on 100% juures. Selline simulatsioon peaks näitama loodava süsteemi võimekust lahendada roboti kõige algelisemat vajadust ehk võimekust ennast laadida, et oma tegevusega jätkata.



Joonis 9. Simulatsiooni keskkonnas olev stseen

10 Kokkuvõte

Käesoleval tööl oli üks põhieesmärk, mida sai jaotada kolmeks väiksemaks eesmärgiks. Põhieesmärgiks oli arendada tarkvara, mida kasutades on võimalik luua robotile tegevusplaan, mille abil saab robot minimaalse välise abiga iseseisvalt ringi liikuda ja muid tegevusi teha, mida vastav roboti tüüp võimaldab. Alameesmärkidest esimeseks oli leida loodava süsteemi struktuur. Teiseks tuli leida viis, kuidas valitud struktuuris luua hierarhia erinevate süsteemi osade vahel. Viimaseks oli tarvis leida viis, kuidas erinevad süsteemi osad saaksid omavahel efektiivselt informatsiooni vahetada.

Töö tulemusena on loodud lahendus, kasutades töös välja toodud meetodeid. Selle programmi abil saab koostada robotile tööplaani, mida järgides peaks robot saama teha oma tavalisi tegevusi. Samas aga võib robot vajadusel katkestada oma hetkest tööd ja alustada uue tegevusega, kui roboti antud hetke seisund seda nõuab. Lisaks sellele on loodud ka programm, mis kontrollib kas kõik loodud plaani osad töötavad korrektselt. Juhul kui ükski neist peaks ootamatult oma töö lõpetama, siis vastav jälgija programm käivitab hädaabiplaani, mille kasutaja on varem ise defineerinud, et robot saaks ohutult oma töö lõpetada.

Tööga edasi minnes oleks võimalik olemas olevat programmi laiendada nii, et mitu robotit suudaksid omavahel hierarhia põhiselt tööd teha. Kui praeguses lahenduses roboti süsteem koosneb mitmest sõlmest, mis igaüks käivitub tema jaoks õigel hetkel, kui talle on luba antud. Seevastu aga võiks programmi laiendada nii, et süsteemi loogika jääks samaks, aga selles süsteemis sõlmede asemel oleksid robotid, kes koostöös teineteisega tööd teevad. Seega oleks võimalus robotitel omavahel välja selgitada, kes mingit tegevust peaks tegema ning kõrgema prioriteediga robotitel oleks võimalus teistest varem tegutseda.

10.1 Hinnang eesmärgi saavutamisele

Süsteemile on leitud struktuur, mis sobib hästi robotites kasutatava *ROS* vahevaraga ning mis lubab kergelt ja efektiivselt süsteemi laiendada ja täiendada. Süsteemi osade vahel on prioriteedi numbrite näol loodud kindel hierarhia, mille abil on võimalik ära defineerida, milline süsteemi osa peaks teatud hetkel töötama. Lisaks on leitud viis kuidas süsteemi osade vahel informatsiooni vahetada kasutades selleks *ROS* vahevaras olemasolevaid võimalusi. Kokkuvõtvalt võib öelda, et kõik töös toodud eesmärgid said edukalt lahendatud kasutades selleks ära maksmaalselt olemaolevaid lahendusi või kogudes mõtteid varem tehtud teadustöödest.

Summary

Current work contains one main goal, which was divisible into three subgoals. Main goal was to create a mission control program. By using it, user can define a plan for the robot so it can move and act by itself, needing minimal amount of help. First subgoal was to find a structure for the mission control. Secondly the mission control needed a hierarchy between different parts of the system. Lastly a way how to share information between different parts of the system was needed.

As a result a program was created using all the methods described in this thesis. User can define a plan for a robot using this program so that the robot can do it's usual activities. The robot also can stop its current activity and start executing another one if the robots current state needs it. In addition to that a supervisor program was created, which looks after all the parts of mission control's plan. When some part of mission control unexpectedly stops working, then the supervisor program can activate a fail safe program, which the user itself defined, and the robot can end its work properly.

Kasutatud kirjandus

[1] Abascal, J., Lazkano, E., Sierra B. (2005). Behavior-Based Indoor Navigation. - Ambient Intelligence for Scientific Discovery. Lecture Notes in Computer Science. 3345, 263-285. [*Online*] SpringerLink (21.03.2017)

[2] Astigarraga, A., Lazkano, E., Rańó, I., Sierra, B., Zarauz, I. (2003). Sorgin : a Software Framework for Behavior Control Implementation [WWW] http://www.sc.ehu.es/ccwrobot/papers/astigarraga03sorgin.pdf (22.03.2017)

[3] Estlin, T., Volpe, R., Nesnas, I., Mutz, D., Fisher, F., Engelhardt, B., Chien, S. Decision-Making in a Robotic Architecture for Autonomy [WWW] http://robotics.estec.esa.int/i-SAIRAS/isairas2001/papers/Paper_AM123.pdf (22.03.2017)

[4] Nesnas, I., Wright, A., Bajracharya, M., Simmons, R., Estlin, T., Kim, W. S. CLARAty: An Architecture for Reusable Robotic Software [WWW] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.9886&rep=rep1&type=pdf (22.03.2017)

[5] Zhang, J., Knoll, A. Hybrid Motion Control by Integrating Deliberative and Reactive Strategies [WWW] http://www6.in.tum.de/Main/Publications/Zhang1995a.pdf (22.03.2017)

[6] Farinelli, A., Iocchi, L., Nardi, D., Ziparo, V.A. (2006). Assignment of Dynamically Perceived Tasks by Token Passing in Multirobot Systems - Proceedings of the IEEE, 94 (7), 1271 - 1288. [*Online*] IEEEXplore (22.03.2017)

[7] SMACH teegi ametlik ROS Wiki [WWW] http://wiki.ros.org/smach (22.03.2017)

[8] TREX teegi github [WWW] https://github.com/fredpy/trex2-agent (22.03.2017)

[9] Salumäe, T., Raag R., Rebane, J., Ernits, A., Toming G., Ratas M., Kruusmaa, M. (2014). Design principle of a biomimetic underwater robot U-CAT. - *Oceans – St. John's 2014, St. John's, NL, Canada, September 14-19,* 1-5 [*Online*] IEEEXplore (08.05.2017)

Lisa 1 – Koodi *Github*-i salv

https://github.com/mission-control-ros/mission_control

Lisa 2 – Koodi katvus testide poolt

Tabel 1. Koodi katvus testide poolt

Name	Statements	Missing	Cover	Missing line
src/behaviour.py	253	0	100%	
src/behaviour_fail_safe.py	42	0	100%	
src/behaviour_subprocess.py	60	0	100%	
src/behaviour_subprocess_fail_safe.py	42	0	100%	
src/mission_control_utils.py	78	0	100%	
src/mission_control_utils_cache.py	14	0	100%	
src/mission_control_utils_constants.py	7	0	100%	
src/watchdog.py	60	1	98%	116
TOTAL	556	1	99%	