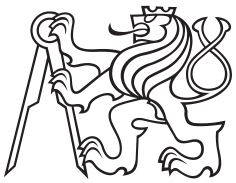
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Software testing for embedded applications in autonomous vehicles

Bc. Jiří Kerner

Supervisor: Ing. Michal Sojka, Ph.D.
Field of study: Cybernetics and Robotics
Subfield: Systems and Control
May 2017

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Kerner Jiří**

Studijní program: Kybernetika a robotika
Obor: Systémy a řízení

Název tématu: **Testování softwaru pro vestavné aplikace v autonomních vozidlech**

Pokyny pro vypracování:

1. Seznamte se s pravidly soutěže autonomních modelů aut F1/10.
2. Proveďte rešerši metod pro testování vestavného (embedded) softwaru zejména v oblasti automobilového průmyslu a autonomních vozidel.
3. Navrhněte vhodné způsoby testování softwaru pro autonomní řízení auta v projektu F1/10.
4. Navržené testovací metody implementuje a použijte je pro testování softwaru vyvíjeného v systému ROS. Zaměřte se na možnost tzv. kontinuální integrace.
5. Výsledky zdokumentujte a zhodnoťte efektivitu a účelnost jednotlivých metod testování v projektu F1/10.

Seznam odborné literatury:

http://f1tenth.org/
ISO 26262-1:2011(en) Road vehicles - Functional safety

Vedoucí: Ing. Michal Sojka, Ph.D.

Platnost zadání: do konce letního semestru 2017/2018

L.S.

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze, dne 30. 1. 2017

Acknowledgements

I would like to thank my family and my girlfriend for their support during my studies.

I would also like to thank my thesis supervisor Ing. Michal Sojka Ph.D. for his guidance and feedback during the writing of this thesis.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used in compliance with ethical principles in the preparation of University theses.

.....

V Praze, 26. května 2017

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

.....

V Praze, 26. května 2017

Abstract

Diploma thesis Software testing for embedded applications in autonomous vehicles describes methods for software testing in the automotive industry and presents possible ways how to verify and validate software for autonomous vehicles. Robotics simulation is used to develop and test software for autonomous vehicles.

The opening part of this thesis is an introduction to the topic of software testing with a focus on automotive industry.

The second part of this thesis introduces Robot operating system (ROS) and Gazebo simulator and how this software is used to test the autonomous behavior of the developed 1/10th scale car. Simulation test scenarios are developed to evaluate car behavior in situations that can occur in the F1/10 race. Car software correct operation is tested using these scenarios.

Last part of the thesis describes the use of Jenkins automation server to automate routine tasks such as package building and test running. The quality of project source code is tested and verified by static analysis.

Keywords: software testing, robotic simulator, autonomous vehicles, continuous integration

Supervisor: Ing. Michal Sojka, Ph.D.
G-203,
Karlovo náměstí 13,
120 00 Praha 2

Abstrakt

Diplomová práce Testování softwaru pro vestavné aplikace v autonomních vozidlech se zabývá popisem metod pro testování softwaru v automobilovém průmyslu a představuje možnosti, jak v budoucnu přistoupit k verifikaci a validaci softwaru v autonomních vozidel. Práce prezentuje způsob, jakým testovat software pro autonomní vozidla za použití robotických simulací.

Úvodní část práce je teoretickým úvodem do problematiky testování softwaru se zaměřením na automobilový průmysl.

V druhé části práce je pak představen Robotický operační systém (ROS) a simulátor Gazebo a jejich využití pro testování autonomního chování auta v měřítku 1:10. Za pomoci tohoto softwaru jsou následně vytvořeny simulační scénáře pro testování chování auta v situacích, které mohou nastat během závodu F1/10. Software v autě je otestován za pomoci vytvořených simulačních scénářů.

Poslední část práce se věnuje systému Jenkins a automatizaci rutinních úkolů, jako je například kompilace balíčků a spouštění testů. Zdrojový kód je staticky analyzován pro zajištění kvality kódu a včasnou detekci chyb.

Klíčová slova: testování software, robotický simulátor, autonomní vozidla, kontinuální integrace

Překlad názvu: Testování softwaru pro vestavné aplikace v autonomních vozidlech

Contents

1 Introduction 1

2 Automotive software engineering 3
2.1 History of software in cars 4
2.2 ISO 26262 4
2.2.1 Software development according to ISO26262 6
2.2.2 Initiation of Product development at the software level . 6
2.2.3 Specification of software safety requirements 6
2.2.4 Software architectural design . 7
2.2.5 Software unit design and implementation 7
2.2.6 Software unit testing 8
2.2.7 Software integration and testing 8
2.2.8 Verification of software safety requirements 8

3 Automotive software testing 11
3.1 Software testing 11
3.2 Testing classes 11
3.2.1 White-box testing 12
3.2.2 Black-box testing 12
3.3 Test Levels 12
3.3.1 Unit Testing 12
3.3.2 Integration testing 12
3.3.3 System testing 13
3.3.4 Acceptance testing 13
3.4 Testing practices 13
3.4.1 Peer code review 13
3.4.2 Code coverage 13
3.4.3 Static code analysis 14
3.4.4 Test automation 14
3.5 Test methods 15
3.5.1 Model based testing 15
3.5.2 Simulation 15
3.5.3 X-the-in-loop methods 15
3.6 Challenges of testing software in autonomous vehicles 16
3.6.1 Sensing 16
3.6.2 Localization and navigation . 16
3.7 Software testing for autonomous vehicles 17
3.7.1 Stochastic systems validation 17
3.7.2 Machine learning validation . 18
3.7.3 Simulation testing 19

4 Car testing 21
4.1 F1/10 competition 21
4.1.1 Car 22
4.2 ROS 22
4.2.1 ROS architecture 23
4.2.2 Communication 24
4.2.3 Catkin 24
4.2.4 Roslaunch 25
4.2.5 ROS test support 25
4.3 Test methodology 26
4.3.1 System requirements 27
4.3.2 Test plan 27
4.4 Robotics simulation 28
4.4.1 Gazebo simulator 28
4.4.2 URDF car model 32
4.4.3 Car control 35
4.4.4 Spawn car into simulation . . . 36
4.4.5 Simulation of two cars 37
4.4.6 Simulated worlds 38
4.5 ROS packages 41
4.5.1 Navigation manager 41
4.5.2 Race manager 42
4.5.3 Race starter 43
4.5.4 Perfect odometry 43
4.5.5 Collision detection 44
4.6 Test cases 44
4.6.1 Test node 45
4.6.2 Test node build 45
4.6.3 Planner tests 46
4.6.4 Overtaking 46
4.6.5 Two cars race 48
4.6.6 HIL testing 49
4.7 Test execution and results 50
4.7.1 Navigation with perfect localization 50
4.7.2 Full system test 51
4.7.3 Verification results 51
4.8 Future work 52

5 Continuos integration with Jenkins 53
5.1 Continuous integration 53
5.2 Jenkins 54
5.3 Jenkins jobs 55
5.3.1 Build workspace and tests . . . 55
5.3.2 Static analysis 56
5.3.3 Automated simulation test . . 57

5.4 Results 59
6 Conclusion **61**
Bibliography **63**
A CD content **67**
B Installation instruction **69**
C Jenkins job example **71**

Figures

2.1 Reference phase model for the software development [1] 7

3.1 Synthetized image with object labeled from game data [2] 19

4.1 CTU car 22
4.2 Rosgraph displays nodes as ellipses connected by topics with names in rectangles 23
4.3 High level overview of system architecture and topic interface . . . 26
4.4 URDF car model visualization . . 32
4.5 Visualization of the simulated car in Gazebo 35
4.6 Tf tree 38
4.7 Examples of simulated race circuits 39
4.8 Car in Gazebo and Rviz 40
4.9 Rviz control toolbar 42
4.10 Race manager high level workflow 43
4.11 Test flowchart 45
4.12 Gazebo worlds for navigation testing 47
4.13 Simulation scenario for overtake testing 47
4.14 Two cars pursuing the same goal 48
4.15 Schema of hil test case 49

5.1 Jenkins web based user interface 54
5.2 List of warnings issued by compiler presented in Jenkins 56
5.3 Jenkins test results 58

Tables

4.1 Gazebo launch parameters 30
4.2 upload_f1tenth.launch parameters 33
4.3 Planner test cases description . . 47
4.4 Test results 1 50
4.5 Test results 2 51

Chapter 1

Introduction

In recent years autonomous vehicles have evolved from research topic to development goal of almost every major car manufacturer and other technological companies such as Google or Apple [3]. While these efforts did already bring many improvements in the field of autonomous driving, there is still a long way to go before autonomous vehicles will be commercially available. One of the obstacles to adoption of autonomous cars is the lack of testing approach that would make it possible to verify the reliability of algorithms used for autonomous behavior in the vehicle. This thesis does not aim to solve this challenge completely but tries to provide a direction and insight into the topic.

Work on this thesis was conducted within the Industrial Informatics Research Center, which is a research group in Department of Control Engineering at Czech Technical University. One of the research topics in this group is autonomous driving, and a part of the research efforts is work on the prototype of Formula 1/10. The work involves designing, building, and testing an autonomous 1/10th scale car capable of speeds more than 60 km/h. The project goal is to create a research platform that can be used to work on the topic of autonomous vehicles and enable cooperation with partners from industry. Any work done in the realm of autonomous driving has to take into account the development practices that automotive industry uses to allow deployment of developed solutions into production vehicle in the future. This thesis summarizes methods and practices used in automotive industry for software development and testing to provide the theoretical foundation for second part of the thesis that sets the goal to create tools that enable construction of test cases for verification of autonomous car behavior. Test cases are generated in robotics simulation software in combination with Robot operating system (ROS), which is also used as the development framework for software in the car.

ROS was successfully used on multiple autonomous cars projects such as Marvin autonomous car from the University of Austin and Junior autonomous car from Stanford University [4]. BMW has been using ROS for the development of their autonomous car, and after about two years of experience, they praise its open source nature, distributed architecture, existing selection of software packages, as well as its helpful community. On the other hand, they

mention barrier for adoption in the automotive industry such as lack of official real-time support, problematic management of configurations for different robots and difficulty in enforcing compliance with industry standards like ISO 26262, which will be necessary for software that's usable in production vehicles[5]. ROS community is aware of this fact and proposes a set of code quality guidelines and quality assurance process, but they come in the form of non-strict recommendation mostly addressing stylistic concerns, such as naming convention and code formattings and are not actively maintained. Use of ROS for building robotics systems is growing and with it increases the need for quality assurance and better testing [6]. A subgoal of this thesis is to provide infrastructure for testing and code quality assurance through automation server and static analysis of source code.

This thesis is structured as follows. Chapter 2 introduces the practice of automotive software engineering. First, the role of software in the modern car is described, and a brief history of car software is given. Then, software processes and standards that automotive industry uses are described to be an introduction to the type of requirements for development of software that goes into production cars. Chapter 3 describes software testing in general, as well as software testing practices that are used in automotive industry and challenges of testing software for autonomous vehicles, are discussed. Chapter 4 briefly describes F1/10 competition rules and car in the beginning. Then an approach to testing car software in the simulated environment that can verify the behavior of the car in the race like conditions and situations is presented. Multiple tests scenarios are introduced with emphasis on automation of the testing process. Automation of testing and activities tied to code quality is further developed in Chapter 5 which introduces a way how to use Jenkins automation server to run jobs in periodic and automatic manner and how to report the results back in readable and organized form.

Chapter 2

Automotive software engineering

This chapter introduces challenges that software engineers in automotive industry are solving today. A brief history of software in cars is given in 2.1 to provide the perspective of how software in modern cars evolved. Then ISO 26262 and software development according to it are outlined in 2.2.

Software plays the dominant role in many technical products today, and this fact also applies to modern cars. It is estimated, that 90% of current car innovations are based on electronics and software [7]. Software in the car is a dominant factor for innovations in automotive industry, as well as it is decisive for competitiveness in the automotive market. With the arrival of autonomous vehicles and car-to-car communication, the role of software in vehicles will only grow. Most of the new features in cars will be software-enabled and software will remain the innovation driver for the next years. While it is expected that autonomous driving will bring increased safety and comfort for the passengers, it also presents challenges for developers as well as for testers responsible for validation and verification of the software inside the car. Automotive industry is well aware of the need to innovate their practices to address the changes that are happening.

As the infotainment systems adoption grows, cars get connected to the internet and with the question of cybersecurity is becoming more pressing. It has been already revealed by cybersecurity experts that it is possible to take over some types of cars remotely [8]. The cybersecurity of connected cars will have to be ensured before wide adoption will be possible. Last but not least, if the car is driving itself, who is responsible for the accident? Legal issues are a huge topic in autonomous vehicles and many questions have to be answered before mass adoption will be possible.

While the focus is currently on self-driving versions of today's automobiles, any vehicle that meets the fundamental need to move passengers and cargo will follow soon. Challenges of development technology to enable full autonomy are tackled with the usage of modern technologies, such as machine learning, artificial intelligence, and new sensors. Use of these technologies requires new methodology for verification and validation of software components before they can be sent into production, especially with critical applications such as driving.

2.1 History of software in cars

The first software found its way into cars only about thirty years ago. From one generation to the next, the software amount in a number of lines of code was growing by a factor of ten, or even more. Today we find in premium cars more than ten million lines of code, and we expect to find ten times more in the next generation [9].

Software components in cars used to be very isolated and local. That means that every different task has its dedicated controller(Electronic control unit or ECU) as well as dedicated sensors and actuators. To optimize wiring, bus systems were deployed in the cars, and ECUs became connected. Today premium cars feature not less than 70 ECUs connected by more than five different bus systems. Up to 40% of the production cost of the car are due to electronics and software development, and up to 70% of software/hardware systems development costs are software costs [9].

Software is the most crucial innovation driver for technical systems. Innovative functions are realized by software, and it also allows new, better solutions to known tasks [9]. The automotive industry is facing transition to becoming software companies, and this transition is difficult. The automotive industry has long tradition in finding and using proprietary solutions, which leads to steeper learning curve for new engineers as well as little to no reuse of code from one car to another [10]. On the other hand, the size of the challenge ahead inspires cooperation between manufacturers as in the case of Automotive Grade Linux[1], to give one example. Automotive Grade Linux promises to enable rapid innovation in infotainment, connected car, advanced driver assistance systems (ADAS) and autonomous driving.

The results of this fast transition to software-intensive development is that car recalls are increasingly happening because of mistakes found in source code of the vehicle software. These bugs are not just life threatening to car users, but also cost automakers billions of dollars [11]. Software new position as first class citizen in automotive industry is addressed by development of standardized methods for software development and software testing to ensure quality and safety in critical application.

2.2 ISO 26262

ISO 26262, also titled Road vehicles – Functional safety is international standard for functional safety of electrical and electronic systems in production automobiles defined by the International Organization for Standardization (ISO) in 2011. It is aimed at reducing risks associated with software for safety functions to a tolerable level by providing feasible requirements and processes [1].

The need for standardization has increased with the role of software in car. To ensure quality and safety, common standard to evaluate implementation

[1] Automotive Linux: https://www.automotivelinux.org/

and processes has to be given. Another reason to standardize processes is globalization of the automotive market and the fact that contractors are operating worldwide and standard for quality verification must be established to enable quality control and unified standards. ISO 26262 seek to standardize software development process in automotive industry by providing a foundation for implementing engineering concepts in software development process. While it takes time for company to achieve compliance with these standards, the cost of failure associated with software defects is much greater than the cost of ensuring quality.

ISO 26262 is an adaptation of the Functional Safety standard IEC 61508. It is intended to be applied to safety-related systems that include one or more electronic and electrical (E/E) systems and that are installed in series production passenger cars with a maximum gross vehicle mass up to 3 500 kg [1]. ISO 26262 consists of 9 normative parts which are listed below.

- Part 1: Vocabulary
- Part 2: Management of Functional Safety
- Part 3: Concept Phase
- Part 4: Product Development: System Level
- Part 5: Product Development: Hardware Level
- Part 6: Product Development: Software Level
- Part 7: Production and Operation
- Part 8: Supporting Processes
- Part 9: ASIL-oriented and Safety-oriented Analyses

ISO 26262 is a risk-based standard. The risk of hazardous operational situations is assessed, and safety measures are defined to detect or control random hardware failures and avoid or control systematic failures. Approach to determine risk classes, known as Automotive Safety Integrity Levels (ASIL) is provided by ISO 26262 in the form of hazard analysis and risk assessment, which takes place in Concept Phase of product development. Hazard is assessed based on the relative impact of hazardous effects related to a system, as adjusted for relative likelihoods of the hazard manifesting those effects. That is, each hazardous event is assessed regarding the severity of possible injuries within the context of the relative amount of time a vehicle is exposed to the possibility of the hazard happening as well as the relative likelihood that a typical driver can act to prevent the injury [1]. The ASIL levels range from ASIL D to QM. ASIL D represents the highest degree of automotive hazard and highest degree of rigor applied in the assurance the resultant safety requirements. Quality management (QM) represents application with no automotive hazards and no safety requirements to manage the ISO 26262 safety processes. The intervening levels (A, B and C) are simply a range

of intermediate degrees of hazard and degrees of assurance required. The higher the ASIL level, the more rigorous development and testing processes are required. Safety analysis in this matter also enables to allocate enough time and resources to project to reflect amount of work which it will take to develop product for given ASIL level.

2.2.1 Software development according to ISO26262

This section introduces Part 6 of ISO 26262, Product Development: Software Level. ISO 26262 addresses possible hazards caused by malfunctioning behavior of E/E safety-related systems, including interaction of these systems and specification of phases for product development at the software level. These phases are initiation of product development at the software level, specification of software safety requirements, software architectural design, software unit design and implementation, software unit testing, software integration and testing, and verification of software safety requirements [1]. Each development phase is described in its respective section below.

2.2.2 Initiation of Product development at the software level

The objective of phase Initiation of Product development at the software level is to plan and initiate functional safety activities as well as define supporting processes in the form of project plan, safety plan, etc. Qualified tools and guidelines to be used in the project are picked according to ASIL level that was determined by hazard analysis and risk assessment for given product.

The software development process for the software of an item, including lifecycle phases, methods, languages, and tools, shall be consistent across all the sub-phases of the software lifecycle and be compatible with the system and hardware development phases, such that the required data can be transformed correctly [1]. ISO26262 parts at the Software level of the standard are mapped to development V-Model as defined by Automotive SPICE, Figure 2.1.

V-Model captures the whole software development lifecycle from system design to item integration and testing and how each design and implementation phase on the left side of the model is mapped to verification and validation phase on the right side of the model.

2.2.3 Specification of software safety requirements

Objectives of phase Specification of software safety requirements are to specify the software safety requirements from the technical safety requirements (including their ASIL) and the system design specification, detail the hardware-software interface requirements and verify that the software safety requirements are consistent with the technical safety requirements and the system design specification [1].

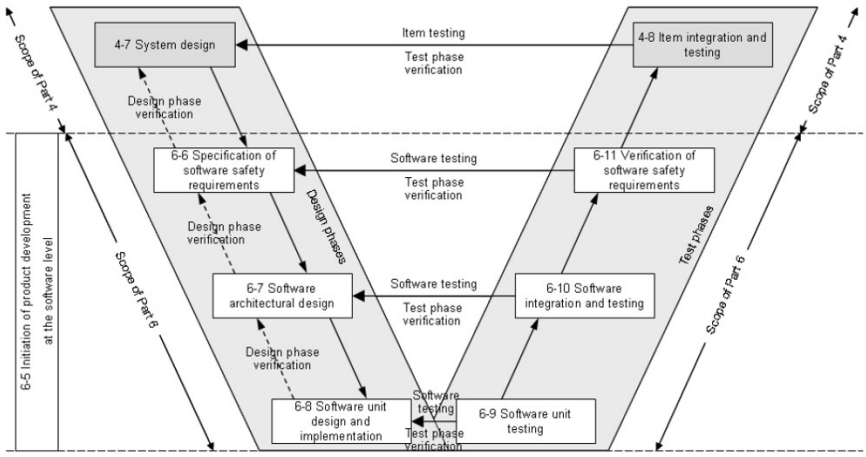


Figure 2.1: Reference phase model for the software development [1]

2.2.4 Software architectural design

Objectives of phase Software architectural design are to develop a software architectural design that realizes the software safety requirements and verify the software architectural design. It gives requirements for notations for software architectural design with goal to use appropriate levels of abstraction. It includes design principles to apply to achieve modularity, encapsulation and minimum complexity. These design principles can be met by use of hierarchical structure, restricted size of software components and interfaces, etc. Safety software requirements are allocated to software components. Software components of differents ASILs are treated as belonging to the highest ASIL if one software component can affect the other software component.

Safety analysis applied to software architecture helps to identify and confirm safety-related characteristics and supports specification of the safety mechanisms. Software architecture shall also address requirements for error detection such as plausibility checks, detection of data errors, control flow monitoring, etc.

Error handling shall also be addressed in software architecture with mechanisms such as static recovery, graceful degradation, correcting codes for data, etc. Last but not least software architecture shall specify verification requirements including control flow analysis, data flow analysis, and inspections.

2.2.5 Software unit design and implementation

Objectives of phase Software unit design and implementation are to specify the software units by the software architectural design and the associated software safety requirements after that implement the software units as specified and finally verify the design of the software units and their implementation.

Requirements for notation based on ASIL level shall be specified to allow

subsequent development activities to be performed correctly and effectively. Design principles shall be specified and applied to achieve robustness, testability, and simplicity of source code. Examples of these design principles are one entry and one exit point in subprograms and functions, limited use of pointers or usage of a subset of a programming language that is used for development of the product.

Software unit design and implementation also specify verification strategies such as control flow analysis, data flow analysis, static code analysis, inspections and more. Verification shall prove compliance with coding guidelines, compliance with source code specification and compliance with a hardware-software interface. Verification shall also prove compatibility with target hardware.

2.2.6 Software unit testing

Objectives of phase Software unit testing are to demonstrate that the software units satisfy their specification and do not contain undesired functionality. Software unit testing addresses planning, selection of test methods such as interface test, resource usage test, etc. It describes methods of deriving test cases to demonstrate appropriate specification of test cases such as analysis of requirements and boundary values analysis.

It specifies test environment requirements that should ideally be as close as possible to target environment, and it also specifies evaluation criteria such as compliance with expected results and pass or fail criteria for tests.

The main goal of software unit testing is to demonstrate compliance with the software unit design specification and the HW/SW interface. Unit tests shall also show the correctness of the implementation, absence of unintended functionality and robustness.

2.2.7 Software integration and testing

Objectives of phase Software integration and testing are to integrate software components into software system. Integration testing shall demonstrate that the embedded software correctly realizes software architectural design. It addresses planning of activities related to integration testing and selection of test method to show that software components and embedded software achieve compliance with the architectural design and the HW/SW interface.

It shall demonstrate correctness of implementation and robustness of embedded software system. Examples of test methods at integration testing level are fault injection, resource usage test, etc. Methods for deriving tests cases such as analysis of requirements and boundary value analysis shall be documented as well as requirement for the test environment.

2.2.8 Verification of software safety requirements

Objectives of phase Verification of software safety requirements are to demonstrate that the embedded software system fulfills the software safety require-

ments in the target environment. It addresses selection of test environments and practices such as Hardware-in-the-loop (HIL). It shall cover execution on the target hardware and evaluation criteria that demonstrates compliance with expected results, coverage of the software safety requirements and pass/fail criteria.

Chapter 3

Automotive software testing

This chapter provides overview of software testing as an engineering discipline, Software testing terminology is introduced and defined. Practices and methods that are used in automotive industry for software testing are described. Then, challenges for testing software in autonomous cars are outlined, and possible solutions to these challenges are given.

3.1 Software testing

Software testing is an activity with a goal to provide stakeholders with information about the quality of the product under test. Software quality has multiple definitions in literature given different authors. In the end software quality boils down to proving compliance with functional and non-functional requirements. Functional requirements may be calculations, technical details, data manipulation and processing and other specific functionality that define what a system is supposed to accomplish [12]. Non-functional requirements impose constraints on the design or implementation regarding performance requirements, requirements for real-time performance, reliability or security.

Software testing involves the execution of the software system on target hardware to evaluate properties of interest. As the number of possible test cases even for simple software unit is practically infinite, all software testing practices have to adopt strategy how to design test cases on given test level. Test cases are designed to evaluate that the tested system does comply with functional and non-functional requirements.

3.2 Testing classes

On the broadest level, there are two kinds or classes of testing known as white-box testing and black-box testing. These approaches describe the point of view that is used to design a test case.

3.2.1 White-box testing

White-box testing is a process where system is tested, and the tester has full knowledge of the internals of the system under test. This knowledge of internals of the system is required to design test cases that test the robustness of the implementation and correctness of it.

3.2.2 Black-box testing

Black-box testing examines the functionality of the input/output system without knowledge of the internal structure. The tester does not need to know the details of the system internals. The test cases are derived based on high–level system requirements. Test cases consist of setting a test of inputs and outputs that are expected and observed outputs of the system under test with the set of defined inputs.

3.3 Test Levels

Software testing is usually performed at different levels throughout the development and maintenance process. Definition and description of these test levels can be different through the literature and internal test guides in companies. The levels described below follows the description given in Software Engineering Body of Knowledge (SWEBOK)[13].

Four test levels, Unit, Integration, System, and Acceptance test, are defined in following sections.

3.3.1 Unit Testing

Unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use [14]. Unit Testing falls into white-box testing class. The requirement for unit test is that software element under test is isolated from the rest of the code and tested separately.

Each unit test must be a standalone case independent from the rest of the tests. A good rule of thumb is that unit test should only work with objects that are in working memory of the machine that the test is run on. That means unit does not need network access, I/O operations or database access.

3.3.2 Integration testing

Integration testing is the level of software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before system testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing [15].

Integration test can fall to either black-box or white-box testing class. It depends on software module that is being tested and what information does the tester need to create test case for software requirement that is being tested.

3.3.3 System testing

System testing of software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of black-box testing, and as such, should require no knowledge of the inner design of the code or logic [16].

System testing takes as its input an integrated software system that has passed integration testing phase. System testing is usually considered appropriate for assessing the nonfunctional system requirements [13].

3.3.4 Acceptance testing

Acceptance testing verifies that system fulfills all customer requirements. It is a black-box testing method which is done on target hardware in the operation like environment to prove the readiness of the product for release. Acceptance testing is performed by the customer or user of the system under test, and the goal is to establish confidence in the system under test.

3.4 Testing practices

Following sections introduce testing practices used to ensure software quality which were picked with correlation to whitepaper [7], which brings an overview of software testing practices and methods used by car manufacturers and researchers in automotive. These practices are supported by specialized tools, following section gives theoretic introduction and specific tools used in this thesis are described in 5.3.

3.4.1 Peer code review

Peer code review is a procedure in which code is submitted to review process. Another developer from the team goes through the code. It is intended to find mistakes overlooked in the initial development phase, improving the overall quality of software. Code review is also used to check compliance with company style guide for code writing or with industry coding standards.

3.4.2 Code coverage

Code coverage analysis measures how much of the code is covered by unit tests. When the unit test suite is run, the execution of the program is monitored, and code coverage tool generates a coverage report with percentage of code that

has been executed. This percentage has to be accompanied by information which coverage criteria has been used. Coverage criteria are listed below.

- **Function coverage** – Has each function (or subroutine) in the program been called?
- **Statement coverage** – Has each statement in the program been executed?
- **Branch coverage** – Has each branch of each control structure (such as in if and case statements) been executed? For example, given an if statement, have both the true and false branches been executed?
- **Condition coverage** – Has each Boolean sub-expression evaluated both to true and false?

While high code coverage does not necessarily imply good quality of code, it is one of the measures that is used to show to which level the code is tested [17] Branch coverage is mostly used as it ensures that all paths have been tested while keeping the test suite smaller than if condition coverage was used.

3.4.3 Static code analysis

Static code analysis is automated analysis of software that is performed on the source code without actually executing the program. Static analysis programs can range from tools that check source code line by line up to tools that analyze complete source code and can find violations across multiple files.

Use of static analysis tools is growing in commercial sphere as part of verification process, mostly used in safety-critical computer systems such as medical software, nuclear software, aviation and automotive software. Static analysis software can check for known dangerous language constructs, compliance with software standards or compliance with code style guide. The range of open-source and commercial solutions is available for many programming languages.

3.4.4 Test automation

Software testing is iterative process. Fixing software bug in one part of the system can generate or illuminate the other bug in different part of the system. That is why it is necessary to have as many as possible test cases performed automatically to ensure that bug fixing or new feature implementation does not break existing code base.

Automation servers are used for automated testing, which means running given test cases automatically. These automatic runs can be done periodically or every time source code is changed by connection to Source Control Management (SCM) program. These tools help to avoid cumbersome repetitive

task and help improve software quality by giving quick feedback if test cases fail and help to discover regressions.

3.5 Test methods

There exists a wide variety of test methods as much as types of software exists. Testing methods described below are selected based on results of survey [7]. Methods that are used in practical part of this thesis are described in more depth.

3.5.1 Model based testing

Model-based testing is used to test systems developed with model-based development approach. Software components are no longer handwritten in C or Assembler code but modeled with MATLAB/Simulink^{TM}, Statemate, or similar tool[18].

Model-based development in Matlab and Simulink and therefore model-based testing is commonly used in automotive industry and makes up to third of all used development languages[7]. Even though some work on the topic of model-based development for autonomous vehicles exists [19], different development approach was used in project F1/10, and model-based testing is therefore not covered in bigger detail.

3.5.2 Simulation

The simulation does imitate the operation of real-world process or system over time. Simulation can be used to show the eventual real effects of alternative conditions and courses of action. Simulation is also used when the real system cannot be engaged, because it may not be accessible, or it may be dangerous or unacceptable to engage, or it is being designed but not yet built, or it may simply not exist [20].

A computer simulation uses a model of real-world or system on a computer to run simulation scenarios to see and study how the system works. By changing variables in the simulation, predictions may be made about the behavior of the system. It is a method to virtually investigate the behavior of the system under study [21].

3.5.3 X-the-in-loop methods

In-the-loop methods are simulation techniques that are used in development and testing of complex real-time embedded systems. They are namely Model–in–the–Loop (MiL), which uses e.g. a MATLAB/Simulink Model, Software–in–the–Loop (SiL) uses compiled software for the target machine, but runs within an emulator hosted on the development machine, Processor–in–the–Loop (PiL) uses the SiL Software and the target processor (e.g. on an evaluation board) but not the contemplated ECU. Hardware–in–the–loop (HiL) uses the

target ECU with it specified the hard-wired interface and a simulation input values via defined signals (I/O and Bus messages) [7].

3.6 Challenges of testing software in autonomous vehicles

To discuss testing methodology for autonomous vehicles, engineering principles and technologies that are used for development of autonomous vehicles have to be described first. An autonomous vehicle is evolutionary goal of Advanced Drives Assistance System (ADAS) development. The path taken by most car manufacturers is incremental betterment of ADAS technology up to a point, where the vehicle become fully autonomous. This goes hand in hand with development in modern robotics which combines sensing, localization and mapping and navigation tasks. On the other hand, researchers are working on the second approach which is called end-to-end learning and is also described further.

3.6.1 Sensing

The first requirement for autonomous vehicles is to have a set of sensors to use. Different approaches exist, but in the end, most of the parties working on autonomous vehicles use one of the following types of sensors. While Google uses Light Detection And Ranging (LIDAR) sensors with their car[22], Tesla, which has arguably the most advanced ADAS capability in current market equips all cars built and delivered starting October 2016 with eight cameras, 12 ultrasonic sensors, and radar [23].

Historically LIDAR and radar technologies were used heavily in development of self-driving cars, with the advancement in deep learning and hardware, cameras and computer vision are expected to make bigger and bigger impact in the field of autonomous driving.

3.6.2 Localization and navigation

ADASes that we can meet in modern cars are for example lane detection, pedestrian detection, road signs detection and blind-spot monitoring. While tasks such as automatic cruise control and lane keeping in constrained environment such as highway make it possible to achieve good performance with deterministic algorithms, most of the research and development in autonomous cars is currently using machine learning.

What all these tasks have in common is that they are vision-based tasks and deep learning techniques have been applied to computer vision problems recently. To ensure full autonomy, other capabilities such as traffic light detection, car detection, and obstacle detection need to be developed. There also a different approach and that is end-to-end learning. Sensor data are linked through neural network directly to steering commands [24].

While this approach has generated interesting results, the challenge of machine learning, in general, is that the developer has minimal information about how the system works inside and makes the decision. If the system makes an error, the developer may not know why it made the error, which shows the main disadvantage of these systems. How do we prove that the system we have designed is safe under all circumstances? These challenges are discussed in section 3.7.

3.7 Software testing for autonomous vehicles

Methodical approach rather than a simple cycle of system-level test-fail-patch-test will be required to deploy safe autonomous vehicles at scale. The ISO 26262 development process (V model, Figure 2.1) sets up a framework that ties each type of testing to a corresponding design or requirement document but presents challenges when adapted to deal with the sorts of novel testing problems that face autonomous vehicles [25].

ISO 26262 is mandatory for software development in automotive and as such it is expected that autonomous car development will have to comply with this standard. This area is currently a topic of research [26] and is also widely discussed by professionals inside the automotive industry. Functional safety defined by ISO 26262 requires human driver to oversee the function of the vehicle, and therefore ISO 26262 in its current form brings challenges of defining the safety and functional requirement for autonomous vehicles [27]. This challenges should be addressed by the second edition of ISO 26262 standard which is due for release in early 2018 [28].

The challenge of testing for autonomous vehicles lies in the nature of technology that is being used to program autonomous vehicles. Non-deterministic algorithms and statistical algorithms with combination with machine learning systems are used. Some of the challenges of testing these systems are discussed in following chapters.

3.7.1 Stochastic systems validation

Non-deterministic computations include algorithms such as planners, perception algorithms and pedestrian detection. These systems are classification tasks which by nature have to trade between false positives and false negatives in classification tasks. Systems with this property have to be setup in order what rate of false positives to false negatives is acceptable for proper system function.

This tradeoff with combination of stochastic nature of the system makes verification and validation of the system difficult. One way this can be illustrated is testing the behavior of the system in specific edge case scenario. The challenge here is to know that such a scenario does exist and then design the specific test case because the scenario can be triggered by narrow set of input in very specific situation which can be hard to reproduce. What is more, non-deterministic system can have very different output for similar

inputs. This implicates that the decision about pass/failure of the test is not binary because the correctness of behavior of the system is measured in percentage rather than a true/false statement.

3.7.2 Machine learning validation

In machine learning, there are many approaches such as supervised vs. unsupervised learning and active learning. All of them use inductive learning, where training examples are used to derive a model. To test the performance of trained model, data sets used for learning are usually divided into three parts, namely training set, validation set, and test set. The training set is used to train the model, and the validation set is used to estimate prediction error. The test set is used to evaluate the trained model performance.

Two sets used for testing the model, namely validation and test set are used to prevent and catch overfitting to training data. Overfitting is when model learns to describe the noise in the data rather general underlying relationship in the data. Overfitted model does work very well on the training set and validation set but works poorly with test set because of novel data points in this set.

The main takeaway should be, that complete end-to-end testing of the whole system is infeasible for the task of autonomous driving. Full vehicle testing is not enough, and not all scenarios can be tested before deployment. Taken complexity of requirements of autonomous vehicles into account, is seems likely that rare edge cases will be discovered during use and underlying software components will have to be retrained to accommodate these new realities.

An Approach that seems to be applicable to address all the challenges described is extensive testing and validation through use. To evaluate the system fit for use, extensive use in target real-world environment has to be conducted. For example, aircraft permissible failure rates are one in 10^9 hours[29]. This implies that at least 10^9 hours of operation testing has to be conducted, but probably even longer to increase the statistical significance of such testing. Building a fleet of physical vehicles big enough to run billions of hours in representative test environments without endangering the public seems impractical. Thus, alternate methods of validation are required, potentially including approaches such as simulation, formal proofs, fault injection, bootstrapping based on a steadily increasing fleet size, gaining field experience with component technology in non-critical roles, and human reviews [25].

Another significant part of testing the fleet of the car through use is the amount of data that will be collected by the cars. Intel predicts that autonomous car can create up to 4 000 GB od data for one hour of driving [30]. This illustrates the need not just for back-end infrastructure, but also the connection to the car with sufficient bandwidth. The data acquisition in commercially deployed cars also offers legal questions regarding data safety and customer privacy.

While there are many questions yet to solve, it is probable that some hybrid approach will be used. New systems can be tested in predefined simulation scenarios to validate their operation fast before deployment into real car, where additional assessment is done with continuous monitoring of the car operation during its use by customer.

3.7.3 Simulation testing

A simulation is a tool that can be used for testing as described in 3.5.2. Robotic simulator software can be used to test robot, autonomous vehicles in our case, without depending physically on the actual machine. One of the main advantages of simulation to the testing of autonomous vehicles through usage is that simulation does not necessarily need to run in real time and can run much faster, therefore making the validation and testing time shorter.

As main disadvantage of robotic simulators is often stated their inability to simulate real environment realistically enough, therefore things that run in simulation flawlessly, might encounter problems in real world because of factors that simulation did not simulate such as the wind and other weather conditions.

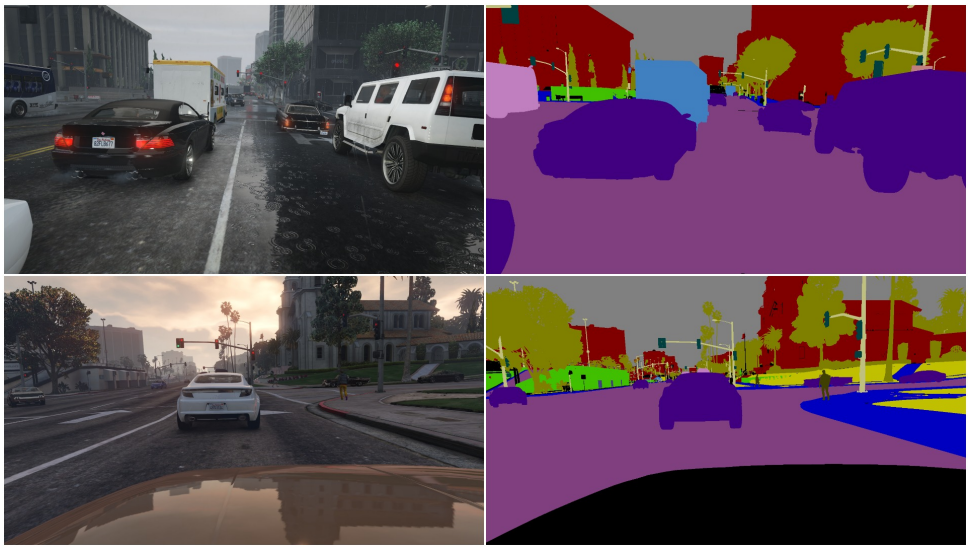


Figure 3.1: Synthetized image with object labeled from game data [2]

While this argument is still valid these days, there are first signs of things changing. With growing quality of computer graphics and photo-realistic like rendering, simulated environments are used for research into autonomous vehicles with games such as GTA V. Researches from TU Darmstadt and Intel Labs developed an approach to automatic labeling images synthesized by the game. The result of the work shows that model trained on synthesized images with real word data improves the accuracy of the model against data set without synthesized images [2]. Microsoft developed open-source AirSim simulator using Unreal engine for use in drone research. It enables hardware

in the loop testing with popular drone controllers, and more vehicles are to be implemented as well soon [1].

There are research labs that try to take advantage of images synthesized in simulation, use them for training of models and transfer the knowledge into real world [31],[32]. As the computer graphics get better over time and better simulation software is developed, we can expect this approach to take bigger part in development. Use of robotics simulator greatly saves development times as is enables fast prototyping and prove of concept. Modern robotic simulators enable simulation of robot control, used sensors and are equipped with advanced physics engines and quality rendering and 3D modeling. To get used from these tools it is important to study how simulation scenarios can be automated and evaluated without the oversight of the developer to enable regression testing. Ways how to achieve this autonomy in test execution and evaluation are described in 5.3.3.

[1] AirSim: https://github.com/Microsoft/AirSim

Chapter 4

Car testing

In this chapter, first, the car and the F1/10 competition rules are presented in 4.1. Robot operating system (ROS) is introduced, and selected parts of ROS which are relevant for the rest of this thesis are then described in bigger detail as well as ROS built-in support for software testing in 4.2.

In section 4.3 we define what is the system under test and test methods that are used for car testing. We chose robotics simulation as the main tool for car testing. Motivation behind this decision is given followed by description of robotics simulator software in section 4.4 .

Original work starts from section 4.4.2 where car model for simulation is introduced. Instructions for usage of simulated car and description of created simulated worlds is given. ROS packages that were developed for creation of test cases is given in 4.5. Test cases are derived and described in 4.6 and finally, test results are presented and discussed in 4.7.

Implemented functionality described in following sections is distributed with printed copy of thesis on enclosed CD, its content, and directory structure is described in A. Instructions how to install all required software to be able to run commands described in this thesis is given in Appendix B.

4.1 F1/10 competition

Original document with F1/10 competition rules is located on F1/10 competition website [33]. Main points that are relevant for understanding what F1/10 competition is about are described in this section to show what kind of autonomous behavior is required from car and what kind of testing methodology is appropriate to use and brings benefits to the project.

F1/10 competition is a race of autonomous cars built from prescribed parts that use hardware approved by competition organizer. The race is held in the hallways of the University which are roughly 2–5 meters wide. The racing track consists of multiple turns in both directions, uneven walls, varying lighting conditions and static objects on the track. The race will comprise of time trials and the car with the fastest time wins. Participating teams will be split into pools according to their lap times in the qualifying lap. Cars in each pool will compete in race of two cars against every other car in in that pool, and the top cars will qualify for the next round.

The competition rules speak in length about collisions between cars during the race of two cars and how to judge who is guilty of the collision. The main take away from it is that a car will be considered responsible for collision if it hits another car from behind, or it appears to have lost control and hits opponent car during the race.

4.1.1 Car

This section briefly introduces car and components that it is built from. Car for F1/10 competition is built on Traxxas Rally 1/10 chassis and can be fitted with Suspension, Axle Conversion, and Tires that are picked from list approved by organizers of the competition. NVIDIA Jetson TK1 is used as main computing unit for planning and perception, and Teensy board is used for motor control. The car can be fitted with multiple sensors in compliance with list of approved sensors given in F1/10 competition rules. Our car is shown in Figure 4.1. The car is fitted with ZED camera, SICK Tim551-2050001 LIDAR, Razor IMU 9DOF (P/N: SEN-10736 and Netis WF2190 WiFi adapter. A detailed description of the car and its assembly is given in [34].

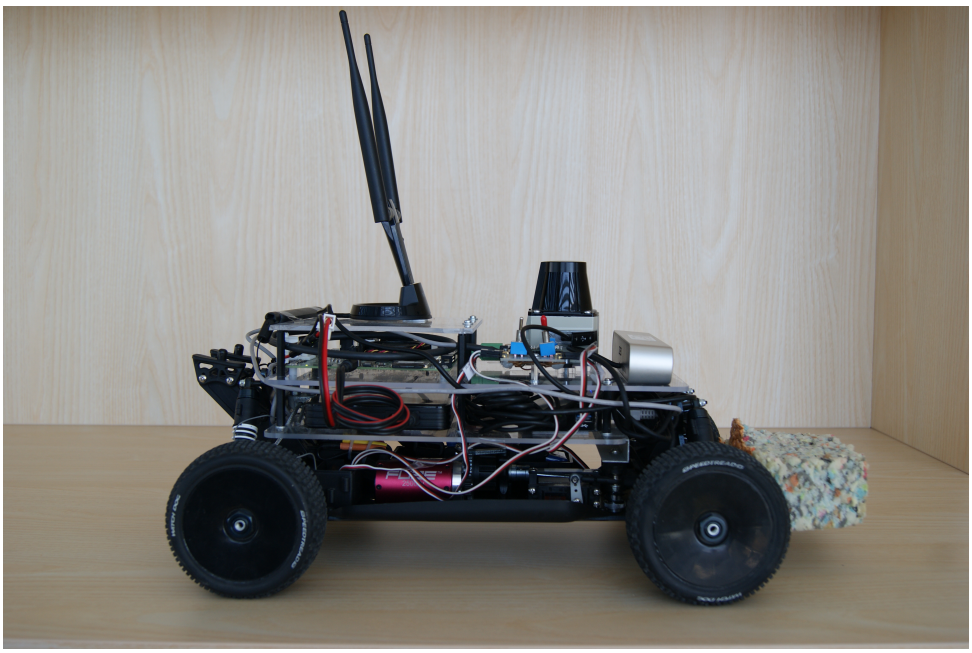


Figure 4.1: CTU car

4.2 ROS

It is required by the F1/10 competition rules to use ROS as part of software running in the race car. The minimal is to use ROS to listen to start command

over ROS topic. ROS in version Indigo, which was released in 2014 and targets Ubuntu 14.04 LTS (Trusty) distribution is used in race car and for this thesis.

4.2.1 ROS architecture

This section gives brief overview of what ROS is and for what application it is suitable. ROS robot software development framework licensed under standard three-clause BSD license. It includes hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. The ROS runtime graph is a peer-to-peer network of processes (potentially distributed across machines) that are loosely coupled using the ROS communication infrastructure [35].

In ROS, a process that performs computation is called node. *Nodes* are processes that perform computation. Term node arises from visualizations of ROS-based systems at runtime: when many nodes are running, it is convenient to render the peer-to-peer communications as a graph (figure 4.2) with processes as graph nodes and the peer-to-peer links as arcs. Lookup mechanism that allows processes to find each other at runtime is called *Master* [36].

Modular design helps to reduce code complexity in comparison to monolithic systems. A system which is distributed in this manner is more tolerant to faults as crash at one node does not result in crash of the whole system. What is more, implementation details are hidden as nodes expose minimal API. This also enables communication of nodes written in different programming languages or running on different machines which are connected over network.

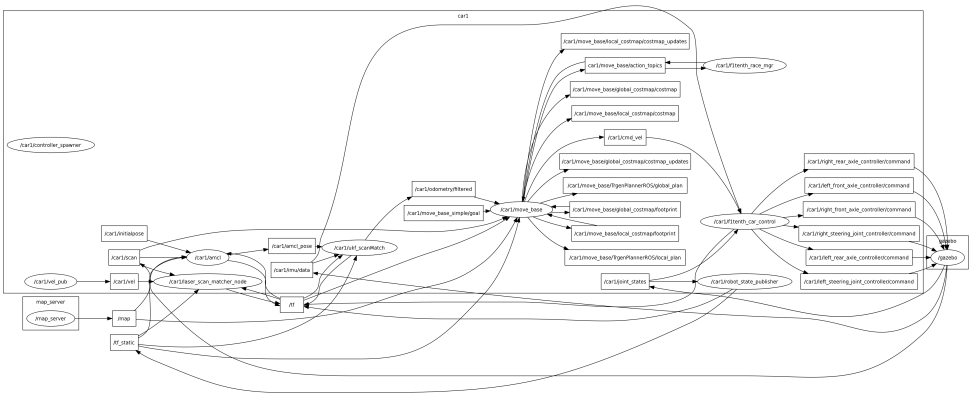


Figure 4.2: Rosgraph displays nodes as ellipses connected by topics with names in rectangles

ROS framework has support for Python, C++ nad Lisp programming languages and has experimental libraries for Java nad Lua. ROS has a custom unit/integration test framework called rostest. ROS currently runs only on Unix-based platforms and is primarily tested in Ubuntu and Mac OS X

systems. ROS consists of core functionalities and ecosystem of ROS packages. Parts of ROS that are relevant for understanding the rest of this thesis are introduced in following sections.

4.2.2 Communication

Nodes communicate with each other by passing messages. A *message* is a strictly typed data structure. Messages can be composed of other messages, and arrays of other messages nested arbitrarily deep. ROS has four main communication paradigms which are in brief described in this chapter. *Topics* are named buses which nodes use to exchange messages. Topics are work on publisher/subscriber architecture. Each topic can have multiple publishers and subscribers[1].

Services are used for one-way transport and are build as request/reply system. Service is defined by pair of messages. One message is used for the request and second for the reply. ROS node offering a service is called provider. Client calls the service using request message and after that waits for the reply[2].

While services enable sending a request and receiving a reply with result, they do not offer the ability to monitor the state of execution of that request or cancel the request. For this, *actionlib* package can be used. It implements server and client pair which communicates over set of predefined messages, namely goal, feedback, and result[3]. This functionality is used for example in navigation system where the goal is sent to the action server, and client gets periodic feedback on the goal pursue status.

The last way how to share data between nodes in ROS is *Parameter server*. It is shared, multi-variate dictionary that is accessible to all nodes via network API and can be used to store and retrieve parameter at runtime. It is low-performance application best used for static data such as configuration parameters[4].

4.2.3 Catkin

ROS package management and build system. It includes build system macros and infrastructure for ROS and offers features such as workspace creation, package creation and compilation, test running and support for static analysis[5]. Catkin workspace is a folder in which catkin packages are built. Catkin uses CMake, a cross-platform, open-source build system for package building. While initially targeted compiler is GNU C++ compiler, catkin packages can be compiled with other compilers such as Clang.

Each catkin package must include catkin compliant package.xml which provides meta information about the package CMakeList.txt. CMakeList.txt

[1] Topics: http://wiki.ros.org/Topics
[2] Services: http://wiki.ros.org/Services
[3] Actionlib: http://wiki.ros.org/actionlib
[4] Parameter server: http://wiki.ros.org/Parameter%20Server
[5] Catkin: http://wiki.ros.org/catkin

provides information how to build the package. There can be no more than one package in each folder, and nested packages are not allowed.

Catkin also provides functionality for registering and running tests in catkin workspace. Each package can also be registered for static analysis with integrated roslint tool which uses cpplint for C++ code and pep8 for Python code. Tests and analysis can be run from catkin workspace on one by one basis or all at once.

4.2.4 Roslaunch

Roslaunch – command line tool which is used to run launch files. *Launch* file specify the parameters to set to Parameter server and nodes to instantiate, as well as the machines that they should be run on. Launch files are written in XML. A Minimal working example of file foo.launch is presented.

```
<launch>
  <node name="foo" pkg="foo_pkg" type="foo_app">
</launch>
```

This launch file runs binary foo_app from package foo_pkg under name foo and is run by roslaunch as.

```
roslaunch foo.launch
```

Several tags are defined for launch files that can be used to combine nodes that are written to coexist as group such as assisgning the same namespace to group of nodes, give a group of nodes the same name remapping or specify on which machine they should run on[6].

4.2.5 ROS test support

ROS package and build system offers build in support for testing in form of googletest[7] for C++ language and unittest[8] for Python language. Use of these supported frameworks is recommended, but it is possible to use different unit test framework as long as it is compatible with xUnit framework.

ROS has its own integration test suite called rostest which is based on roslaunch and is compatible with xUnit framework. Rostest[9] enables to use roslaunch files as test fixtures and do full integration testing across multiple nodes. Rostest interprets <test> tag in launch file, which specifies test node to run. Test nodes are nodes that execute defined tests while the system is running. The advantage of <test> tag is that if file is run with roslaunch, <test> tags are ignored, and system launches without invoking tests. When the .launch file is run with rostest command, all the test nodes are invoked. It is possible and good practice to integrate the rostests into catkin build and

[6]Roslaunch: http://wiki.ros.org/roslaunch
[7]Googletest: https://github.com/google/googletest
[8]UnitTest: https://docs.python.org/2/library/unittest.html
[9]Rostest: http://wiki.ros.org/rostest

run them with catkin tools in catkin workspace. Further details are given in section 4.6.2.

4.3 Test methodology

This section describes architecture of car software system and which parts of the system will be tested. Requirements for autonomous behavior that the car has to have to enable participation in F1/10 competition are derived, and ways how to test these requirements is discussed. Finally, test methodology for F1/10 project is established.

F1/10 project is a collaboration of a group of students. Software components are developed in separation by another team member. To derive test methodology, brief overview of software architecture is shown in Figure 5.3. Named arrows describe ROS topics over which ROS nodes communicate. Localization, Navigation, and Map are high-level components which group ROS nodes together. Main communication topics between these components are displayed. Implementation details for these components are described in [34]. Race component is a group of packages that were developed for this thesis and are introduced in 4.5. Vertical dashed line shows interface between car software and input from sensors and output to motor control.

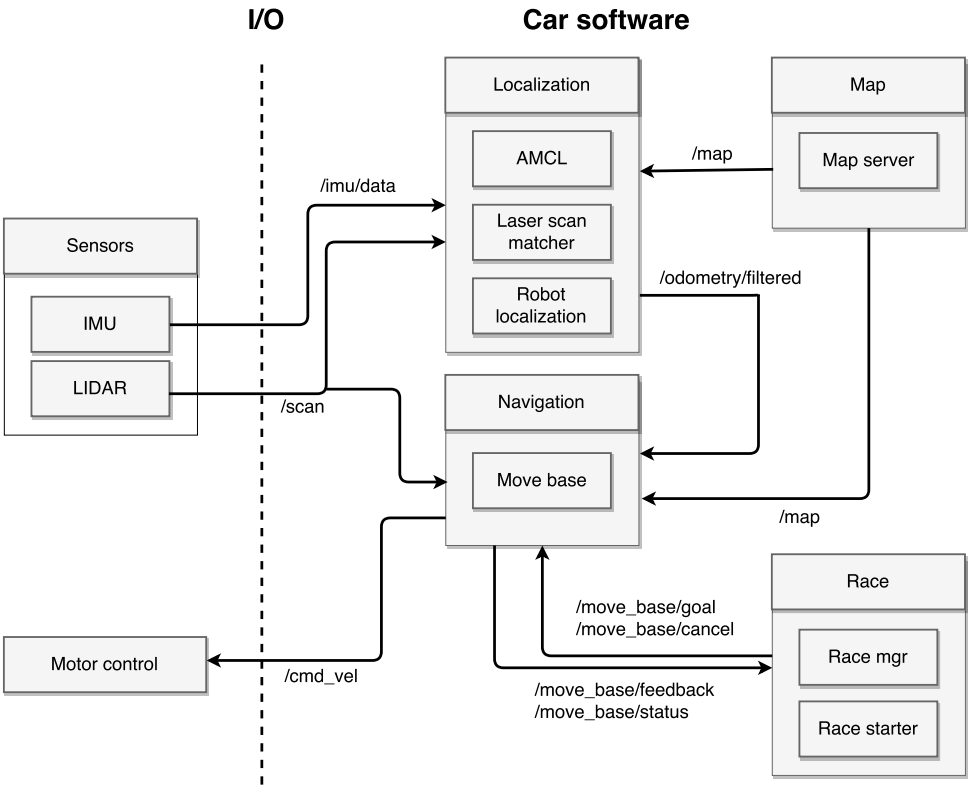


Figure 4.3: High level overview of system architecture and topic interface

4.3.1 System requirements

This section defines system requirements for car software system. Requirements for car autonomous behavior are derived from F1/10 competition rules [33].

1. Car can persistently complete a mission defined by the safe traversal of ordered series of checkpoints with the objective of minimizing completion time.
2. Car can interpret static obstacles within the environment in order to maintain collision free progress
3. Car can exhibit context-dependent speed and angular velocity control in a static environment.
4. Car can interpret dynamic obstacles within the race environment in order to enable predictive controls and planning, such as is necessary to ensure collision-free progress.
5. Car is able to Exhibit context dependent speed and angular velocity control in a dynamic environment.
6. Car is able to navigate in areas where sensors may not provide map-based localization (i.e., LiDAR range is insufficient).

4.3.2 Test plan

This section describes which test levels are used to test car software, how requirements listed in 4.3.1 are verified and which test methods are used to verify them.

Test levels were described theoretically in 3.3. Figure 5.3 shows, that car software is built by combining existing ROS packages. These packages are used in multiple ROS projects and are already unit tested, therefore no more testing on this level is done, and packages are expected to function as described in their respective documentation. Apart from using existing ROS packages, new local planner plugin was written for move_base. This plugin uses predefined interface to integrate with move_base and does not change move_base API, therefore can be tested as part of system tests.

Communication between nodes is done over ROS communication stack which was described in 4.2.2. Topics, services, and actionlib provide well-defined interface which fails at compile time if incompatible type is assigned to message. Because ROS communication stack is used, specialized integration tests do not have to be written to test communication between nodes.

This shows that it is possible to view car software as black-box and evaluate requirements defined in 4.3.1 by testing the system using its input/output (I/O) interface in prepared test scenarios. Because car software implementation was done parallel to work on this thesis, it was required that test method which will be used to verify requirements can also be used for testing during

development. Robotics simulator was selected as tool which will enable to develop required autonomous behavior and at the same time design test cases to verify functional requirements.

4.4 Robotics simulation

In this section firstly Gazebo simulator and its integration with ROS is described, then challenges of modeling car with Ackermann steering are discussed, and car model is written in Unified Robot Description Format (URDF) is introduced. Once robot model is established, ways how to control it in simulation are outlined and at last process how to spawn one or multiple car models into simulation is documented.

4.4.1 Gazebo simulator

This section introduces Gazebo[10] simulator, open source robot simulation tool licensed under Apache 2.0. Its development is currently stewarded by Open Source Robotic Foundation[11] (OSRF). Gazebo comes prepared with multiple robots that are ready to be used in simulation. It takes advantage of multiple open source projects to provide high-performance physics engines including ODE, Bullet, Simbody, and DART. OGRE 3D library is used to render realistic environments including high-quality lighting, shadows, and textures. The Gazebo provides ways to generate sensor data for most sensors used in modern robotics such as LIDARs, IMUs, high-resolution cameras and more. These sensor data can be augmented with noise to better simulate performance of real world sensors.

With recent development in cloud computing, Gazebo provides possibilities to run simulation on remote servers and interface to Gazebo through socket-based message passing using Google Protobufs. Cloud simulation and web browser GUI are currently under development in the form of CloudSim and Gzweb. Version 7.5 is used in this thesis as it is the highest compatible version with ROS Indigo.

ROS and Gazebo Interface

Gazebo and ROS communication interface is described in this section. The Gazebo has been a standalone project since 2015 a no longer has direct code dependencies to ROS. To integrate Gazebo and ROS, a set of ROS packages named gazebo_ros_packages provides wrappers around the standalone Gazebo[12]. Gazebo_ros_pkgs are crucial part that enables ease of use and integration of simulation into ROS development workflow. Because ROS topics, services and controllers in the form of ros_control are used to interface with

[10]Gazebo simulator: http://gazebosim.org/
[11]OSRF: https://www.osrfoundation.org/
[12]ROS integration: http://gazebosim.org/tutorials?tut=ros_overview

the simulation as well as with the real robot, the switch between simulation and real world use can be written as one simple parameter in roslaunch file.

While every distribution of ROS comes with support for given version of Gazebo, it is possible to use newer version of gazebo with older version of ROS if gazebo_ros_packages are properly recompiled. The only downside to this approach is that all packages using Gazebo and ROS have to be compiled manually inside catkin workspace, because packages in ROS ppa are compiled for Gazebo version coming with given ROS distribution, such as Gazebo 2 for ROS Indigo. This downside is most of the time worth the effort as newer version of Gazebo simulator offer significantly more features and better performance than older versions.

Roslaunch tool, described in 4.2.4, can be used in combination with gazebo_ros_package to start Gazebo simulator as ROS node. This node can be called with multiple startup parameters which are described in Table 4.1. To run Gazebo simulation, launch file for empty world, which is default for Gazebo session is included. To launch a custom world, world_name parameter is set. It can be absolute path to .world file or relative path to GAZEBO_RESOURCE_PATH environmental variable. Example usage in launch file is shown in following code snippet.

```
<include file="$(find gazebo_ros)/launch/empty_world.launch">
  <arg name="world_name" value=
  "$(find f1tenth_gazebo)/worlds/$(arg world_file_name).world"/>
  <arg name="debug" value="$(arg debug)" />
  <arg name="gui" value="$(arg gui)" />
  <arg name="paused" value="$(arg paused)"/>
  <arg name="use_sim_time" value="$(arg use_sim_time)"/>
  <arg name="headless" value="$(arg headless)"/>
</include>
```

Description format

Universal Robot Description Format[13] (URDF) is introduced in this section. URDF is used in ROS to write robot description for use in simulation and visualization software. Gazebo, on the other hand, switched to Scene Definition Format[14] (SDF). While URDF can be parsed into SDF and used with Gazebo simulator, it is not possible to convert SDF to URDF. To use robot described with URDF in Gazebo, special measures has to be taken into account. There are efforts in the community to update URDF format or provide a way to convert from SDF to URDF, but currently, no straightforward solution exists. ROS tools such as Rviz and Robot state publisher needs to have robot description in URDF format, that is why URDF has been chosen to write f1tenth car description for simulation.

[13] URDF: http://wiki.ros.org/urdf
[14] SDF: http://sdformat.org/

Parameter name	default	description
world_name	empty.world	Path to .world file, that will be used for simulation.
debug	false	Start gzserver (Gazebo Server) in debug mode using gdb.
gui	true	Launch the user interface window of Gazebo.
paused	false	Start Gazebo in a paused state/
use_sim_time	true	Tells ROS nodes asking for time to get the Gazebo-published simulation time, published over the ROS topic /clock.
headless	false	Disable any function calls to simulator rendering (Ogre) components. Does not work with gui:=true.

Table 4.1: Gazebo launch parameters

Xacro

This section describes Xacro[15] (XML Macros), XML macro language. Xacro enables to write shorter and better structure robot description in URDF by writing XML macros that expand XML expression on call. A typical example of use is writing a macro for creation of wheel for the robot which is later instantiated for each wheel that the robot has, this reduces the amount of code that has to be written to describe the robot and also makes the description files more readable.

Xacro enables to create properties and property blocks which are named values that can be inserted anywhere in the XML document. Property blocks are snippets of XML code that can be reused at multiple places. Xacro enables to evaluate math expressions inside XML documents and also allows conditional blocks which make it possible to change robot description by use of configuration files. This enables to quickly change the set of sensors the robot is using or gazebo plugins which are loaded when the robot is simulated. Xacro macros are most commonly used to define sensors and other repeating parts of the robot and enable creation of multiple blocks with different names and attached to different links.

Links and joints in URDF

URDF has a tree structure with one root link, and the root link in robots used with ROS is usually called base_link. The URDF tree is composed

[15] Xacro: http://wiki.ros.org/xacro

of links and joints. Links are created by usage of simple shapes such as sphere, box, and cylinder. These links consist of visual, inertial and collision description. Links are joint together using joints[16]. Joints are described by its type, which can be revolute, continuous, prismatic, fixed, floating or planar. They also have to specify parent and child link that they are connecting. This is shown in code snippet below which is describing base_link and fixed joint that is connecting base_footprint and base_link. In the code snippet below we can see the use of xacro functionality, referencing parameters such as chassis dimensions in the form of chassisLength,chassisWidth and chassisHeight. Another example is the use of box_inertia that is a call to macro that computes moment of inertia for given box.

```
<joint name="base_footprint_joint" type="fixed">
  <origin xyz="0 0 ${centerOfGravity}" rpy="0 0 0" />
    <parent link="base_footprint"/>
    <child link="base_link" />
</joint>

<link name="base_link">
  <inertial>
    <mass value="${chassisMass}" />
    <origin xyz="0 0 ${centerOfGravity}" />
    <xacro:box_inertia m="${chassisMass}"
                       x="${chassisLength}"
                       y="${chassisWidth}"
                       z="${chassisHeight}"/>
  </inertial>

  <visual>
    <origin xyz="0 0 ${centerOfGravity}" rpy="0 0 0" />
    <geometry>
      <box size="${chassisLength}
                 ${chassisWidth}
                 ${chassisHeight}"/>
    </geometry>
  </visual>

  <collision>
    <origin xyz="0 0 ${centerOfGravity}" rpy="0 0 0" />
    <geometry>
      <box size="${chassisLength / 2}
                 ${chassisWidth / 2}
                 ${chassisHeight / 2}"/>
    </geometry>
  </collision>
</link>
```

[16]Joint: http://wiki.ros.org/urdf/XML/joint

4.4.2 URDF car model

ROS and its packages have been mainly developed for use with differential steering robots. This provides challenges for F1/10 project because F1/10 race car is a car-like robot which is using Ackermann steering. Interest group for development and producing generic interfaces for control, navigation and simulation of Ackermann robots exists in ROS, but so far did not produce a solution that would enable to take a ROS package with defined robot and start simulating.

Describing robot with Ackerman-steering in URDF description format is problematic because of the tree-like structure of the format. Theoretical background on why these challenges exist and how to solve them is given in [37]. For the use in this thesis rbcar_description[17] and rbcar_robot_control[18] packages from project RBCAR[19] that has source code available on Github under BSD-2-Cause license have been used to jump start the writing of simulated car description. Rbcar_description is URDF description of four wheel robot with Ackermann steering and rbcar_robot_control is Gazebo plugin which enables control of Ackerman-steered robots and emulates Ackermann steering in software as described in [37]. In the end, most of the software has been overwritten to suit F1/10 project better and extended with new functionality required for automation of testing scenarios. Visualization of robot URDF description is in figure 4.4.

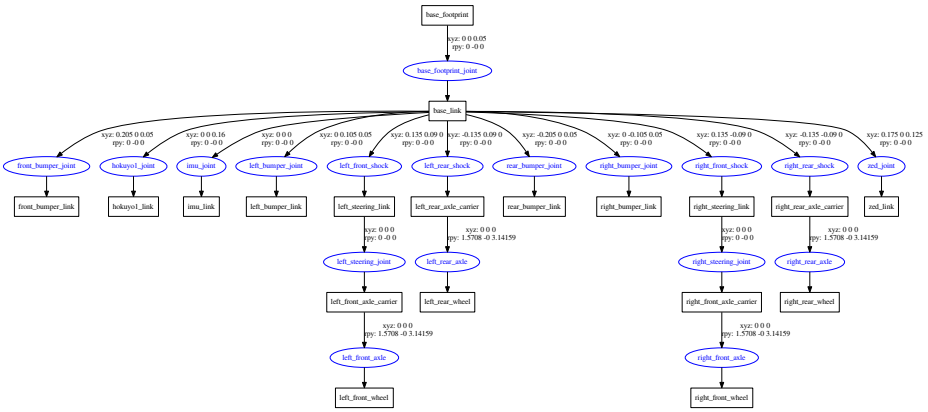


Figure 4.4: URDF car model visualization

All files that describe the car in simulation are stored in f1tenth_description package. Launch file upload_f1tenth.launch is provided with this package. This launch file takes parameters that enable to change robot description

[17] RBcar common: https://github.com/RobotnikAutomation/rbcar_common
[18] RBcar sim: https://github.com/RobotnikAutomation/rbcar_sim
[19] RBCAR: http://wiki.ros.org/Robots/RBCAR

based on what is needed for which scenario. Description of parameters that can be passed to upload_f1tenth.launch is given in Table 4.2.

Parameter name	default	description
perfect_odom	false	If this parameter is true, Gazebo publishes exact location of simulated car on odometry topic perfect_odom.
has_zed_camera	false	If this parameter is true, simulated car has stereo camera sensor.
has_bumpers	false	If this parameter is true, simulated car has contact sensors on each side of chassis.
has_lidar	true	If this parameter is true, simulated car has lidar sensor.
display_laser	true	If this parameter is true, laser rays from lidar are displayed in Gazebo simulator.

Table 4.2: upload_f1tenth.launch parameters

When upload_f1tenth.launch is called, robot description is parsed by the xacro interpreter, and robot description in URDF format is produced. This URDF file is stored on parameter server as robot_description. Simulated car inside Gazebo simulation is shown in Figure 4.5.

base

The base of the car is described by two files. F1tenth_base.gazebo.xacro is defined where Gazebo plugins are loaded. Simulation of car uses gazebo_ros_control library which provides interface to send commands to actuators via simulated controllers. The second Gazebo plugin used is called f1tenth_perfect_odometry and is used to read odometry from Gazebo simulation and provide it as ROS odometry topic. This plugin was written for the need of the thesis and is described in 4.5.4.

The second file is f1tenth_base.urdf.xacro which provides description of car chassis in the form of base_link and base_footprint. These are coordinate frames which are used as reference frames for navigation and localization and should always be defined for every robot description. According to REP120 [20] base_footprint is the representation of the robot footprint projected on the floor. In most use cases this frame is not used, and base_footprint is represented as small, weightless box attached directly to base_link, that is also the case for F1/10 car description. Base_link is rigidly attached to mobile robot base, and it is one of basic ROS coordinate frames defined in

[20] REP120: http://www.ros.org/reps/rep-0120.html#base-footprint

REP 105 [21].

bumpers

It is required to know if the simulated car has taken part in any collision during the automatic execution of the test to automate simulation scenarios. The Gazebo offers contact sensor that registers contact between two links and predefined gazebo_ros_bumper plugin that publishes gazebo_msgs::ContactsState over ROS topic. The bumper is implemented as the xacro macro with parameters name, parent, number, origin and box. Origin parameter provides position reference against base_link and box specify size and shape of the bumper. The bumper macro definition is in bumper.urdf.xacro. Following code snippet shows an example of referencing xacro macro.

```
<!-- front bumper -->
<xacro:bumper name="front_bumper"
              parent="base_link"
              number="1">
  <origin xyz="${chassisLength/2+0.005}
               0.0
               ${centerOfGravity}"/>
  <box size="0.01
             ${chassisWidth }
             ${chassisHeight / 2}"/>
</xacro:bumper>
```

wheels

Wheels are defined as cylinders with given size and moment of inertia is computed for them as for solid cylinder. Each wheel is equipped with shock absorber, which is modeled as the prismatic joint. Front wheels are also equipped with steering link which is modeled as the revolute joint. Each wheel joint, shock absorber joint and steering joint is equipped with the actuator which enables the simulation of movement via robot control Gazebo plugin which is further described in 4.4.3.

sensors

The simulated car is equipped with the same set of sensors as the real car. Sensors in the simulated car are build by using Gazebo sensor plugins and parameters are set for each sensor to reflect the performance of the real sensors counterparts. IMU sensor and LIDAR sensors description come from the robotnik_sensors package which offers set of predefined set of sensors in the form of xacro macros[22]. Stereo camera sensor description in file zed.urdf.xacro

[21] REP 105: http://www.ros.org/reps/rep-0105.html
[22] Robotnik sensor: https://github.com/RobotnikAutomation/robotnik_sensors

was written based on the template from LIDAR sensor and tutorial on Gazebo wiki[23]

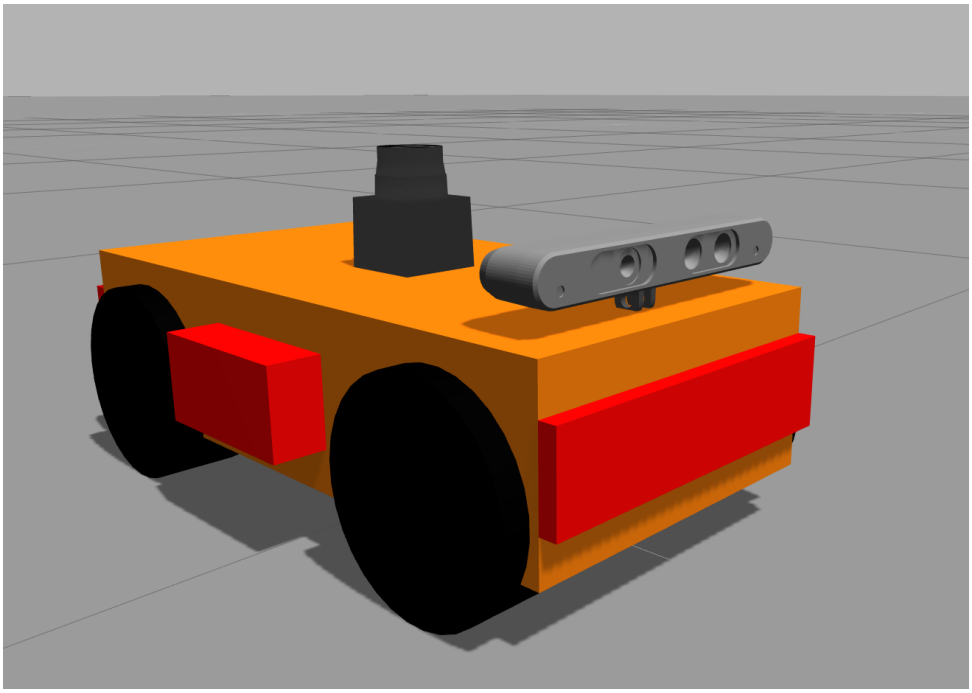


Figure 4.5: Visualization of the simulated car in Gazebo

4.4.3 Car control

Two levels of control have to be issued to enable car movement in simulation, At one level, each joint capable of movements such as wheels and steering has to be actuated. At second level, high-level commands from teleoperation controller or navigation node have to be taken and transformed to command for wheels and steering. Functionality that implements these requirements is stored in f1tenth_car_control ROS package.

Actuators for shock absorbers, steering and axles need to have controllers assigned to them. For this in ROS exists controller_manager package. Actuators in Gazebo have the form of transmission interfaces which are mapped to joints they are actuating. Controller_manager provides infrastructure to load, unload, start and stop controllers. Each actuator is controlled by PID controller and control constants for each controller are stored in configuration file f1tenth_controllers.yaml in f1tenth_car_control ROS package.

Controller spawner is used to load and start a set of controllers at once. When the spawner is run, the listed controllers are loaded, and starter and will get unloaded once the spawner process is killed. Then the controllers are uploaded to parameter server so they can be interfaced by f1tenth_car_control package which takes commands from planner or teleoperation device and

[23]Gz plugins: http://gazebosim.org/tutorials?tut=ros_gzplugins

the commands are transferred to commands for axle and steering actuators. Once joints of the car are actuated, their states are published by robot_state_publisher on topic joint_states. Functionalities described in this paragraph are implemented in launch file f1tenth_controllers.launch and are run using roslaunch.

```
roslaunch f1tenth_car_control f1tenth_controllers.launch
```

F1tenth_car_control node listens on cmd_vel topic for geometry_msgs::Twist which contains command from planner or teleoperation device which consists of linear velocity and steering angle. When this message is received, the commandCallback function is called which checks if the requested speed and steering angle is reasonable and if not, the values are saturated to maximum values for the simulated car. The f1tenth_car_control operates in the infinite loop with the frequency of 50 Hz. The steering command is recalculated for each front wheel according to Ackermann steering geometry and values are sent to actuators in Gazebo. Node f1tenth_car_control is run by call to f1tenth_car_control.launch file using roslaunch.

```
roslaunch f1tenth_car_control f1tenth_car_control.launch
```

4.4.4 Spawn car into simulation

In order to spawn the car into simulation, upload robot_description to parameter server as described in 4.4.2 and start Gazebo simulator as described in 4.4.1. Once simulator is running, launch file spawn_car.launch from package f1tenth_gazebo is used to place car into simulation and load all required controllers. Typical call inside launch file is given below.

```
<group ns="car1">
 <param name="tf_prefix" value="car1_tf" />
<include file="$(find f1tenth_gazebo)/launch/spawn_car.launch">
   <arg name="init_pose" value="-x 0 -y 0.0 -z 0.0" />
   <arg name="init_twist"  value="-R 0 -P 0 -Y 0.0"/>
   <arg name="robot_name"  value="car1" />
  </include>
</group>
```

Group element enables to apply namespace setting to the group of nodes. A namespace is attached as the prefix to each topic in the group. Parameter name tf_prefix is used to set the prefix for tf coordinate frames. Usage of namespaces is not consistent in ROS packages, and while most of the packages enable the use of namespaces, the group tag is not always enough, and manual remapping of ros topics is required. That is why robot_name parameter is passed to spawn_car.launch. For all system to function is important, that robot_name, group ns parameter, and tf_prefix use the same robot name. While namespacing would not be required if only one car is simulated, it is vital when more then one car is simulated. Launch file spawn_car.launch enables to put the car into the simulation with specified initial pose and twist.

3D pose with coordinates x,y and z are in meters, while twist Roll, Pitch, and Yaw is in radians.

4.4.5 Simulation of two cars

This section describes the challenges of simulating more than one ROS controlled car in the Gazebo simulator. Spawning of a single car into simulation has been described in the previous chapter. This section describes why it is important to be able to simulate more than one car and what additional steps have to be taken to enable this functionality. A complete example is given in file sim_two_cars.launch in in f1tenth_gazebo package and can be launched as.

```
roslaunch f1tenth_gazebo sim_two_cars.launch
```

F1/10 competition rules as described in 4.1 state which behavior must car execute while racing against opponents car at one track. There is a need to have a way how to safely evaluate, how the car will behave once it is on the race track with another car to develop required behavior in various situations that can happen during the race of two cars. This is hard to test in the real world for multiple reasons such as not having a second race car to test the algorithms with and also the potential danger of damaging the cars.

ROS architecture is optimized for use with on robot, which is connected to the single master node. While there has been work on multi-master project, for which interest group has been formed, no standard way of to use ROS with the multi-robot system has emerged. This is a known issue which will be addressed in ROS 2 [38], but currently, the workaround of the single-master structure of ROS has to be made. Another challenge is to spawn two different cars into Gazebo simulation. Gazebo_ros, as well as ros_control and robot_state_publisher packages, expect to work with single robot_description parameter. This limitation has not been overcome, therefore both simulated cars use the same description and are separated by using namespaces for all topics and coordinate frames. The second car is spawned into the simulation the same way as described in 4.4.4, only the robot_name, group ns parameter and tf_prefix has to be changed to different values.

The previous paragraph has described how to spawn the second car to Gazebo simulator and load its controllers. The second challenge of the multi-robot system is to keep track of coordinate frames for each car. Package tf[24] is used in ROS to keep track of multiple coordinate frames over time. Base_link and base_footprint coordinate frames are described in 4.4.2. To explain how two cars are tracked by tf, other standard coordinate frames have to be introduced. Odom frame is a short-term local reference which provides pose of the mobile platform that evolves in a continuous way without discrete jumps. Map frame is a world fixed frame with Z-axis pointing upwards and is used as long-term global reference point. Tf frames form a tree structure,

[24] tf package: http://wiki.ros.org/tf

and only such structure can exist for one master node. Part of typical tf tree is on the left side of Figure 4.6.

To control two cars on single tf tree, these cars have to be tied together with single reference parent frame. Because each of the cars has to have unique odometry and base link frame to be able to be navigated by navigation software, the common reference point is the map frame as shown on the right side of Figure 4.6

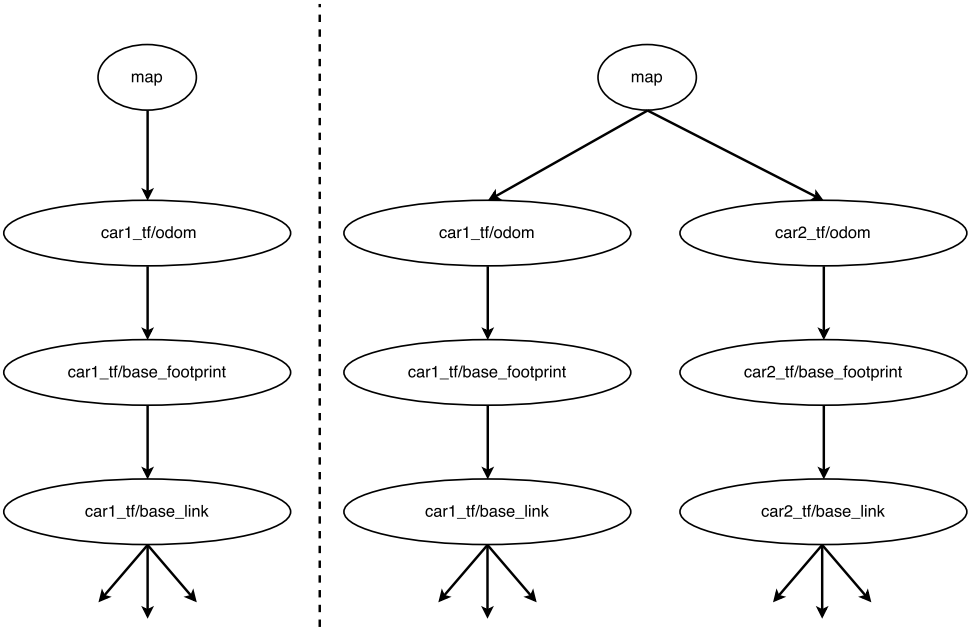


Figure 4.6: Tf tree

It has to be ensured, that map frame has no tf_prefix, and map messages are published on map topic. In practice, the map is loaded with the map_server package and it must ensured that this node is not placed inside group tags.

```
<node name="map" pkg="map_server" type="map_server" args=
"$(find f1tenth_navigation)/maps/$(arg world_file_name).yaml"
output="screen">
  <param name="frame_id" value="/map"/>
</node>
```

4.4.6 Simulated worlds

Five racing circuits were created for Gazebo simulator to test the car in the environment that resembles racing track as described in 4.1. While it is technically possible to build realistic simulations of the real-world environment with 3D file meshes, this does affect simulation time, and the more detailed world does not have to be beneficial for car performance. While the race car is equipped with the stereo camera sensor, it is not currently used in current version of the software, and that does mean that more details and

realism in the simulated world does only lower simulation performance while not bringing practical use. Most of the created circuits are therefore very simple and use only walls to mark the racing track. On the other hand, for future use and to show results that are achievable with Gazebo simulator, the detailed circuit was created for use with the camera and visual odometry testing. An example of simulated worlds are shown in Figure 4.7.

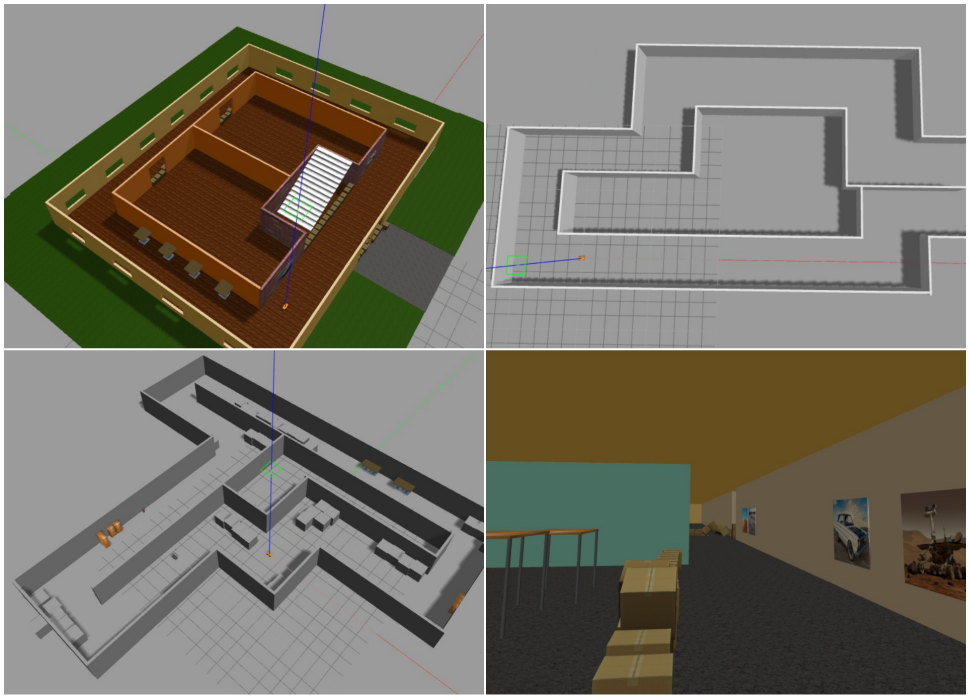


Figure 4.7: Examples of simulated race circuits

All simulated circuits have been built with Gazebo Building Editor[25], which is a graphical interface for creating indoor environments. It offers a 2D view where a floor plan can be created by inserting walls, windows, and doors. The walls can be colored or fitted with predefined or custom texture. Additional models such as furniture and boxes have been taken from Gazebo Model Database[26], which is available for use directly from Gazebo simulator GUI. Models are free to use under Apache License Version 2.0.

Run simulation

This section describes how to run the simulation with the simulated car. Example launch file has been prepared that spawns the car into simulated world. To start the simulation of circuit1.world run command.

```
roslaunch f1tenth_gazebo sim_one_car.launch \
world_file_name:=circuit1
```

[25] Gazebo building editor: http://gazebosim.org/tutorials?tut=building_editor
[26] Gazebo models: https://bitbucket.org/osrf/gazebo_models

Once the simulation is started, we need a way how to send commands to the car and observe how it behaves. For this Rviz[27], which is the 3D visualization tool for ROS, is used. Rviz enables to display messages from sensors such as scans from LIDAR, local global plans from navigation software and odometry information. Rviz uses configuration files which stores setup for given car. To run Rviz with configuration for simulated car run command.

```
roslaunch f1tenth_navigation rviz.launch
```

Rviz opens and automatically loads map and topics from the car and visualizes them. Figure 4.9 shows visualization on the left side, and Rviz view on the right side, user interfaces of both Gazebo and Rviz are cropped from the figure. Rviz top toolbar features a 2D Nav Goal button that can be used to pass navigation goal directly to move_base. Click on 2D Nav Goal button and then click on the map and the car will start driving to that goal.

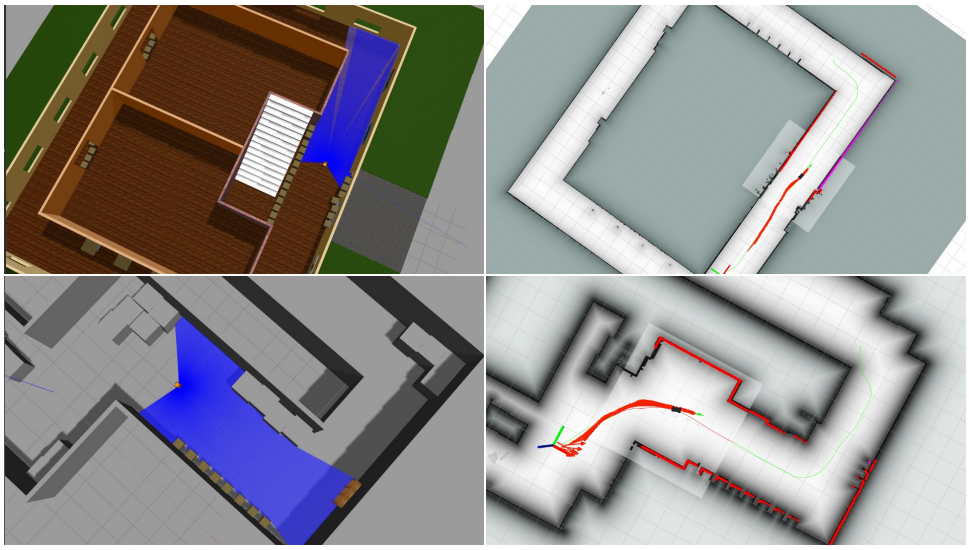


Figure 4.8: Car in Gazebo and Rviz

Create map

Map for each simulated circuit was created. Launch file create_map.launch is prepared for map creation in the f1tenth_gazebo package. The map is created using simulated lidar sensor and gmaping[28] ROS package. Example usage for world circut1 is given below.

```
roslaunch f1tenth_gazebo create_map.launch \
world_file_name:=circuit1
```

Teleoperation such as keyboard or joystick is used to drive the car trough the world and once the map is complete. It can be saved with the map_saver

[27] Rviz: http://wiki.ros.org/rviz
[28] Gmapping package: http://wiki.ros.org/gmapping

tool from map_server[29] ROS package. An example of the map created this way is in Figure 4.9.

```
rosrun map_server map_saver -f circut1
```

4.5 ROS packages

Following sections describe ROS packages that have been implemented as part of this thesis to enable the creation of test cases described in 4.6 to verify requirements stated in 4.3.1.

These packages are navigation manager, the shared library which provides API for communication with move_base action server. Race manager which uses navigation manager library to follow ordered series of checkpoints. Race starter which implements countdown mechanism that sends information about time to start the race to other nodes. Perfect odometry is Gazebo plugin that provides exact information about pose and twist of the car in the simulation. Collision detection implements node that listens to all contact sensors from the simulated car and transforms it into boolean information for test node.

4.5.1 Navigation manager

This section describes shared library written in C++ which implements simple action client that enables to load navigation goals, send them to move_base actionlib server and monitor the execution of these goals. The move_base package provides an implementation of an action server that, given a goal, will attempt to reach it with a mobile base. Details of planner implementation are given in [34].

Navigation manager source code is stored na ftenth_navigation_mgr package. Navigation manager API usage is shown in following code snippet.

```
NavigationMgr navmgr{"robot1"};
navmgr.LoadGoals("navigation_goals.txt");

while (!ros::isShuttingDown()) {
  while (ros::ok() && nh.ok()) {
    navmgr.Spin();
  }
}
```

An instance of NavigationMgr class is created in namespace car1. Navigation goals are loaded from the text file and stored in the queue, instructions on how to create the file with navigation goals are at the end of this section. Then, the Spin function of navigation manager is called. Spin function with committed error message implementation is shown.

[29] Map server package: http://wiki.ros.org/map_server

```
void NavigationMgr::Spin() {
  if (ac.isServerConnected()) {
    if (HasActiveGoal()) {
      CheckGoalState();
    } else if (SizeGoalsQueue() != 0) {
      SendGoal(FrontGoalsQueue());
    } else {
      ROS_INFO("Goals queue is empty");
    }
  }
}
```

Spin function checks that move_base action server is connected, then check if navigation manager has an active goal, if it does, goal state is read from the server in order to check if goal succeeded, if navigation manager has no active goal and goals to pursue are still in queue, next goal is send. To create navigation goals file for world circuit1, run.

```
roslaunch f1tenth_gazebo create_navigation_goals.launch \
world_file_name:=circuit1
```

Figure 4.9: Rviz control toolbar

Rviz window opens, use 2D Nav Goal button from upper toolbar to select goals inside the map, once finished, close the application. Goals are stored in navigation_goals.txt in f1tenth_gazebo/scripts.

4.5.2 Race manager

This section describes race mgr node from f1tenth_race_mgr package. This node first read the configuration file and then loads navigation goals from file described in the configuration file. The configuration file is written in YAML format, and example is given here.

```
f1tenth_race_mgr:
     use_absolute_fpath: false
     package_name: f1tenth_race_mgr
     fpath: /config/navigation_goals.txt
     loop_mode: false
```

The first line is named under which configuration parameters are all found on parameter server. If use_absolute_fpath is true, parameter fpath is taken as the absolute path. Otherwise, it is used relative to package position in the file system. If race manager is configured to work in loop mode, goals are not popped from the queue once they are reached, and the car starts to pursue

the first goal once the last goal is reached. Flowchart of high level logic is displayed in Figure 4.10.

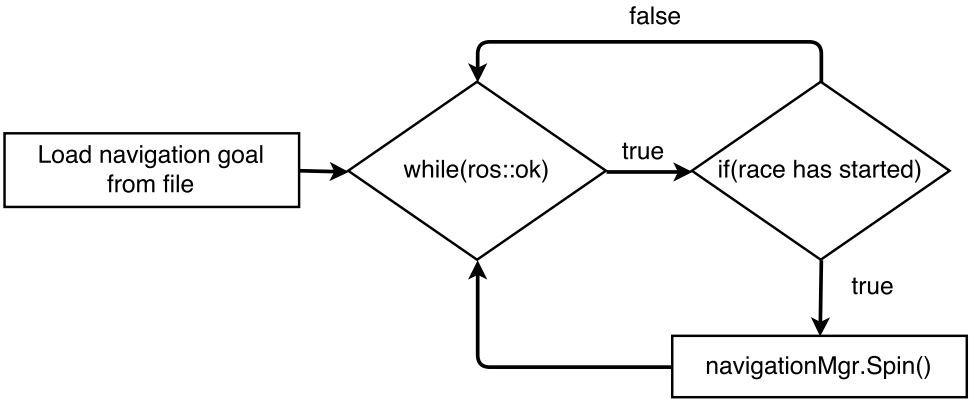


Figure 4.10: Race manager high level workflow

Navigation manager library is used to load goals. Once the goals are loaded, race manager waits for the signal to start racing which is send by race starter node. Once the signal to start racing comes, nav_flag is set to true, and navigation manager Spin function is called periodically.

4.5.3 Race starter

This section describes race starter node from f1tenth_race_starter package. This node implements countdown mechanism. The countdown starts from time_to_start variable which can be setup by the configuration file or default value 5 seconds is used. Race starter node publisher on topic time_to_start every second and after every message internal variable is decremented. Once message with zero value is sent, race starter node terminates. This node enables to start the race of two cars at the same time.

4.5.4 Perfect odometry

This section describes Gazebo plugin perfect odometry which is part of package f1tenth_gazebo_plugins. This plugin enables to read simulated car position from Gazebo simulator and publish it as nav_msgs/Odometry. It is beneficial to have the source of exact odometry for test purposes and development of navigation algorithms. Developers and testers can focus on the evaluation of navigation algorithm without interference from localization inaccuracy. Of course, once the navigation algorithm is developed with exact localization, proper localization algorithm is used to see how the system works together.

Gazebo plugin is compiled as the shared library. It is referenced in robot description of base which is described in 4.4.2. Perfect odometry plugin is written as xacro macro and inserted into robot description as shown here.

```
<xacro:macro name="perfect_odometry">
```

```
    <gazebo>
     <plugin filename="libf1tenth_perfect_odometry.so"
      name="f1tenth_perfect_odometry">
      <alwaysOn>true</alwaysOn>
      <updateRate>200</updateRate>
      <odometryTopic>perfect_odom</odometryTopic>
      <odometryFrame>odom</odometryFrame>
      <robotBaseFrame>base_footprint</robotBaseFrame>
      <rosDebugLevel>na</rosDebugLevel>
     </plugin>
   </gazebo>
</xacro:macro>
```

Perfect odometry plugin listens to Gazebo update event which is broadcasted every simulation iteration. If the time since the last update is longer than update period, Gazebo API is used to read pose and twist of base_footprint, saves this information into nav_msgs/Odometry message and publishes it on topic perfect_odom with update rate 200 Hz.

4.5.5 Collision detection

To automate test of navigation software to the state where simple logic about pass or failure of the test can be derived is to have the ability to evaluate if the car reached its goal, how long it did take and if the car managed to reach the goal without collision with other objects in the simulation. The simulated car is equipped with four sensors for collision detection which are described in 4.4.2.

These sensors publish gazebo_msgs/contact_state messages, which contain an array of states which holds info about models, which are colliding together. This states array is empty when no collision is happening. To track the information from all the collision sensors, dedicated node f1tenth_collision_detection inside package f1tenth_gazebo_plugins was created. This node does subscribe to all the topics from collision sensors and publishes boolean information about the occurrence of the collision. If the car hits anything during automated test, it is a sign that something is wrong and the failed test should be reviewed by the developer.

4.6 Test cases

This section describes test cases that were defined to verify car software against the set of requirements that were set in 4.3.1. Firstly, the structure of test node for automatic tests of car software is described, than instructions on how to build and run tests is given. Finally, test cases created in Gazebo simulator are described.

4.6.1 Test node

This section describes test node for the automated test of planner software. A diagram that shows how the test works is in Figure 4.11.

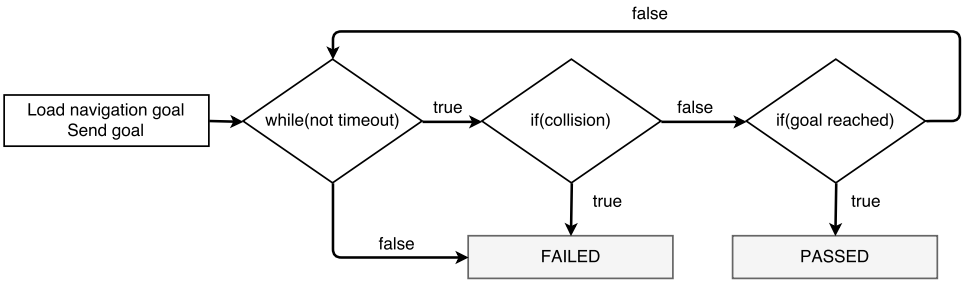


Figure 4.11: Test flowchart

Test node loads goals for given simulation test scenario. It continuously checks that the car did not take part in any collision and if it succeeds in pursuing its goal in reasonable time. Tests like this are run on continuous integration server which presents test results and measures how long did the test take. Each test is equipped with timeout mechanism which ensures that the test does not hang in the undefined state if a bug is committed to the source code.

4.6.2 Test node build

This section describes how to build and run tests. Tests that are defined in catkin workspace, which was described in 4.2.3 have to be built before they can be executed. The build of the tests is done by calling following command in the catkin_ws folder.

```
catkin_make tests
```

The tests itself can be executed in three ways. First and secondly is by using catkin_make as in the case of the build of the tests. Catkin offers a test runner that can be invoked by calling:

```
catkin_make run_tests
```

This command runs all the test registered in catkin workspace and can also be tab completed to run only tests for selected package. Another way how to run tests is to call

```
catkin_make test
```

The difference between these two commands is hard to find in official documentation and other sources are also scarce. The main differences are that run_tests command provides debugging info during test execution, but tries to run multiple test cases at once. This is a problem when tests work with the same resource that can exist in only one instance such as Gazebo simulator and make test fail in the unexpected, not deterministic way. Therefore usage

of run_tests command is not recommended for use with system tests proposed in this thesis.

Call to test command, on the other hand, runs test cases one after the other and makes sure that all resources used for test execution are closed before next test is executed. This way of test execution provides stable results for every run. This is and encouraged way to run the whole test suite for the F1/10 project. Test command does not provide debug info on test execution, but this not a problem since a call to rostest can be done while the test case is developed and debugged.

The third way of test execution is to use rostest. The advantage against usage of catkin tools is that rostest enables passing parameters from the command line to test launch file and also can redirect output to stdout instead of XML file with the test result. This makes the development of test nodes and their execution with different parameters possible without recompiling them every time with catkin or changing parameters in test launch files. An example of rostest call is given below. Test launch file test_f1tenth_planner_direct_path.test from package f1tenth_gazebo is executed with parameter gui set to true.

```
rostest f1tenth_gazebo test_f1tenth_planner_direct_path.test \
gui:=true
```

4.6.3 Planner tests

This section describes test suite that is used to test local planner developed in [34]. Planner tests are automated simulation tests that are used to evaluate planner performance in simple navigation tasks ranging from going straight down the narrow hall to driving through the simple circuit which simulates the possible operational environment. Test node for planner tests are described in 4.6.1. Test cases for planner test suite are described in Table 4.3.

Gazebo worlds for direct path test, left and right turn test, static obstacle test, and simple circuit test are shown in Figure 4.12.

4.6.4 Overtaking

This section describes how the scenario in which one car overtakes another during the race can be simulated. The rules of F1/10 competition described in 4.1 expects that two cars will race on the same track and describes what the expected behavior for the autonomous car is. Since the car sensors are pointed onwards and cannot, therefore, discover another car approaching from behind. Only relevant scenario for testing is when the opponent car is before our car, and we plan to make the overtaking maneuver. This situation is difficult to test with the real car because of the lack of hardware and potential risks involved. Therefore simulation scenario can be helpful in this case.

The simulation scenario that simulates overtaking is shown in figure 4.13. Two cars are spawned into the simulation, and both cars start operating once the race_starter node sends the signal that race has started. One example

Test name	Description
planner_collision	Test that collision detection works. Drive simulated car against wall, test passes once car hits the wall and collision is detected.
planner_direct_path	Test that car can drive in narrow hall and navigate to set goal.
planner_left_turn	Test that car can drive in hall with left hand turn and navigate to set goal.
planner_right_turn	Test that car can drive in hall with right turn and navigate to set goal.
planner_static_obstacle	Test that car can drive through narrow hall with static obstacle that is not in the map.
planner_simple_circuit	Test that car can finish driving on circuit while following multiple goals.

Table 4.3: Planner test cases description

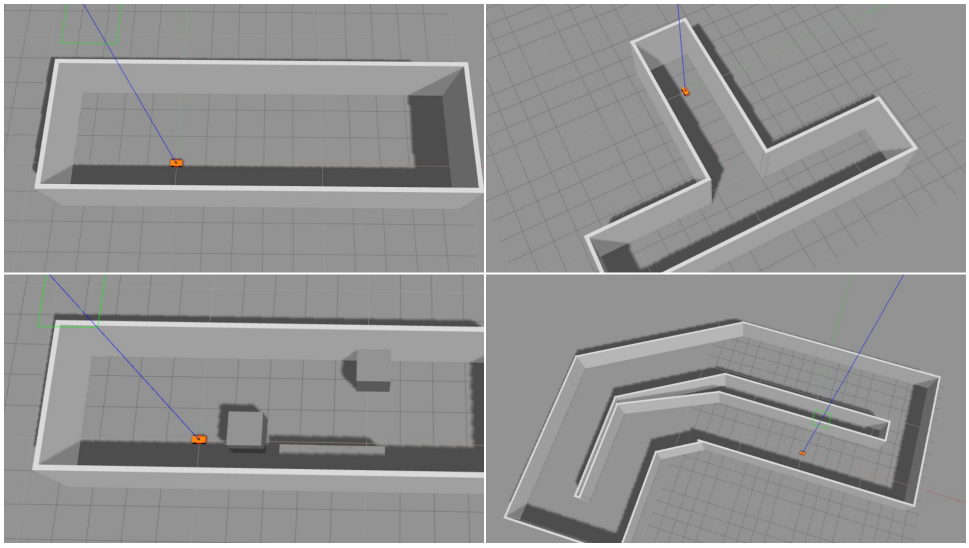


Figure 4.12: Gazebo worlds for navigation testing

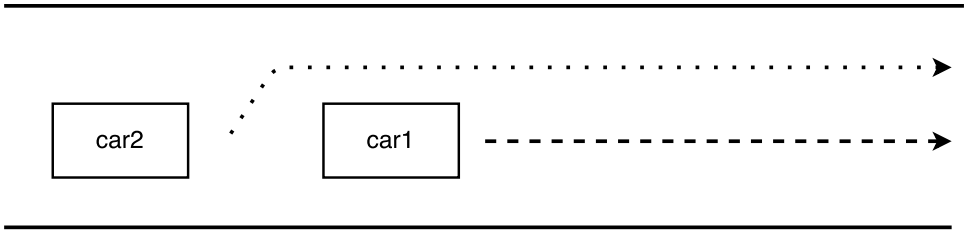


Figure 4.13: Simulation scenario for overtake testing

scenario is created for this thesis, but other scenarios are easy to create using the same approach. A complete example of this simulation scenario is in the f1tenth_gazebo package under test race_overtake. To run this scenario run command.

```
rostest f1tenth_gazebo test_f1tenth_race_overtake.test
```

Gazebo and Rviz opens, and race_starter node sends the signal to start after the countdown. The current version of the software in the car uses the constant velocity of 0.3 m/s compiled into the local planner, therefore both cars go at the same speed and overtaking is not possible. Base_local_planner API which is used for local planner integration into move_base enable to specify maximum linear velocity as the configuration parameter, once this feature is implemented in car software, cars in the simulation can be spawned with different maximum velocity to ensure that car2 can go faster than car1 and actually overtake it.

4.6.5 Two cars race

This section describes how to run simulated test scenario in which two cars race against each other. Once autonomous behavior for the car which is racing on the track against its opponent is developed by using overtaking scenarios for testing, race of two cars can be run to see how the cars will behave in the race like conditions. An example simulated scenario is prepared in the f1tenth_gazebo package in test race_circuit1. To run this scenario run following command.

```
roslaunch f1tenth_gazebo test_f1tenth_race_circuit1.test
```

Gazebo and Rviz opens, and race_starter node sends the signal to start after the countdown. In this scenario both cars are autonomous. The current version of the software in the car is not able to interpret static and dynamic obstacles, and therefore this scenario ends in the crash. The situation is illustrated in Figure 4.14.

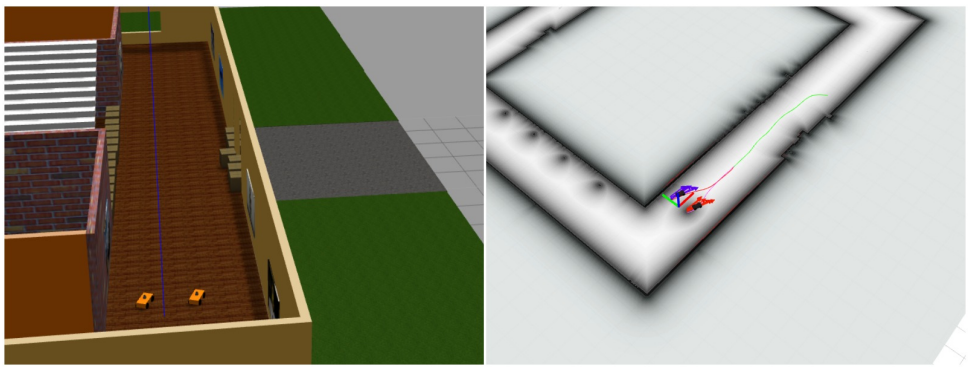


Figure 4.14: Two cars pursuing the same goal

4.6.6 HIL testing

This section describes how to run Hardware–in–the–loop test with car software running on NVIDIA Jetson TK1 and simulation scenario that runs on the computer on the same network. Hardware–in–the–loop test method as described in 3.5.3 enables evaluation of software performance on real hardware while running with the simulated environment. This approach has multiple benefits. Software that is being developed is compiled and run on target hardware which enables to mitigate risks the that arise when porting software between computers with x86-64 processors architecture and ARM processor architecture which is used in car. The computational load of used algorithms is something that has to be taken into account, and some algorithms can work fine on powerful processors of modern desktop computers but struggle with performance on embedded devices used on the car. All these risks can be mitigated by testing on the device early and often. What is more, once car software runs on the target hardware, Gazebo simulator has more resources on the development computer and enables to run simulation software faster. Example use case for HIL is given in Figure 4.15.

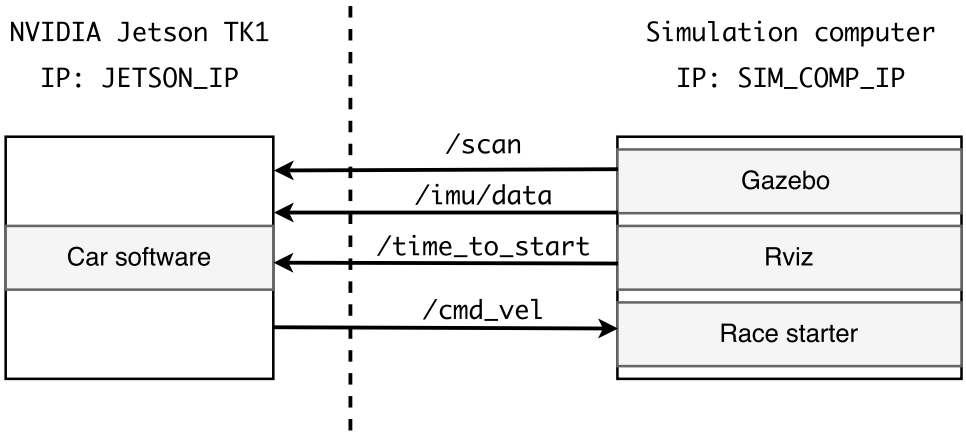


Figure 4.15: Schema of hil test case

Examples of HIL tests is given in f1tenth_gazebo package in the hil_circuit1 directory. In order the run the tests act as follows, both NVIDIA Jetson TK1 and Simulation computer has to be connected to the same network. Then connect over ssh to NVIDIA Jetson TK1 and run.

```
roscore
```

To establish rosmater. Simulation computer needs to connect to rosmater running on Jetson board. On simulation computer export ROS_MASTER_URI and ROS_API and run some ROS command line command to check that the simulation computer is connected to rosmater running on Jetson.

```
export ROS_MASTER_URI=http://JETSON_IP
export ROS_IP=SIM_COMP_IP
rosnode list # check connection to rosmater
```

Once the connection is established, run HIL simulation test launch file on simulation computer. Example call for hil_circuit1 would look like.

```
roslaunch f1tenth_gazebo test_f1tenth_hil_circuit1.test
```

Once the simulation is up and running, switch to Jetson and run

```
roslaunch f1tenth_gazebo \
test_f1tenth_hil_circuit1_system_bringup.test
```

Since simulated sensors and real sensors publish on topics that are named the same, it is easy to switch between real and simulated world. The important thing to remember while doing HIL testing in this manner is to set use_sim_time parameter to true.

4.7 Test execution and results

Real world testing was done in cooperation with Martin Vajnar and report of achieved results is given in his thesis [34]. This chapter puts the system under test using simulated test scenarios as described in 4.6.

Two modes of operation are used. Firstly, to evaluate Trgen_local_planner, tests are run with localization information provided by perfect odometry Gazebo plugin which was described in 4.5.4. This setup enables to evaluate planner performance without interference from inaccurate localization. The second mode of operation is when the test is run in the simulation with the same setup as in real world testing.

4.7.1 Navigation with perfect localization

Firstly, automated planner tests which were described in 4.6.3 were performed, and the results are shown in table 4.4.

Test name	Time[s]	Status
planner_direct_path	58.05	PASSED
planner_left_turn	84.00	PASSED
planner_right_turn	79.51	PASSED
planner_static_obstacle	14.05	FAILED
planner_simple_circuit	534.52	PASSED

Table 4.4: Test results 1

Trgen_local_planner, which is the system under test, uses ROS base global planner to create global trajectory and acts as the local planner. It works as the regulator that keeps the car as close to global plan as possible. This enables to drive the car in environments without static and dynamic obstacles. The ability to spot and to avoid obstacles, which should be part of the local planner, is not currently implemented. That is a reason why test planner_static_obstacle fails. While the rest of the tests passes, their

execution time is very slow. This is happening because trgen_local_planner drive race car at the constant speed which is set to 0.3 m/s.

Because obstacle avoidance is not yet implemented, the car has no capacity to perform overtaking maneuver or race against another car on the circuit, therefore simulation scenarios introduced in 4.6.4 and 4.6.5 cannot be tested and evaluated with trgen_local_planner.

4.7.2 Full system test

The second mode of operation switches from perfect odometry to odometry given by localization system developed for the F1/10 project and the same set of tests is executed.

Test name	Time[s]	Status
planner_direct_path	64.47	PASSED
planner_left_turn	90.10	PASSED
planner_right_turn	89.11	PASSED
planner_static_obstacle	14.13	FAILED
planner_simple_circuit	581.32	PASSED

Table 4.5: Test results 2

We can see that the results are the same while time the test time is longer. This is caused by bigger processor load because localization algorithm needs more computational power than Gazebo perfect odometry plugin. This additional computational load slows down the simulation which makes the test run longer.

4.7.3 Verification results

This section provides results of verification for requirements stated in 4.3.1

1. This requirement was tested and verified by planner test suite 4.6.3 and results are given in 4.7.1 and 4.7.2. Car is able to complete a mission and traverse ordered series of checkpoints. This was tested also with real car and results are desribed in [34].

2. Current implementation of navigation component and local planner is not able to interpret static obstacles. This was tested by planner_static_obstacle test from planner test suite 4.6.3 and also observed in tests with the real car. Therefore this requirement is not met.

3. Current version of navigation software has constant velocity and does not exhibit context-dependent speed. Therefore this requirement is not met.

4. Car is currently not able to interpret dynamic obstacles. This was observed while running simulation scenario with two cars which with current implementation ends in the crash. This requirement is not met.

5. Current version of navigation software has constant velocity and does not exhibit context-dependent speed. Therefore this requirement is not met.

6. In the current implementation, only LIDAR and IMU sensors are used, and the car is fully dependent on them, therefore this requirement is not met.

4.8 Future work

This section describes future work that has to be done on the car to meet all requirements of F1/10 competition and continues with propositions for enhancements to testing in the simulation.

To meet the requirements for F1/10 competition, detection for static and dynamic obstacles has to be added. Detection of other cars during the race should be supported by computer vision system since the opponent car can not be reliably detected only by LIDAR sensor. With the addition of computer vision to the car, visual odometry can be implemented to enable navigation and localization in environments where LIDAR range is insufficient. To exhibit context-dependent speed, car navigation software will have to be updated. Simulated worlds and scenarios developed in this thesis can be used for development of functions mentioned in the previous paragraph. It is a question if simulated environment, as it is presented in this thesis will be suitable for testing computer vision algorithms, or if development work will have to be done in the real world only.

Test node architecture described in 4.6.1 provides test result in form of simple PASSED/FAILED boolean information. While this sufficient format for the result of the unit and integration testing, it turned out to not be ideal for testing of complex car system inside simulated environment. Simulation test takes time to run and can fail for various reasons which cannot be determined instantly, and tester has to rerun the test and watch what happens. Test that takes 10 minutes to complete and where failure happens before end of the test is more of a burden than a benefit. A possible solution for this would be to run Rviz during simulation in the program such as Xvfb, which provides virtual framebuffer and enables to run all graphical operations in memory without showing any screen. At the end of the test, the screenshot of Rviz window would be created and added to test result. We get the graphical representation of test run with this approach, and the reason of failure can be pinpointed faster.

Once software functionality for overtaking and racing against other car is developed, simulation scenarios for these cases can be automated using the similar approach as with tests of planner software in 4.6.3. This with combination with automated test run enables to run regression test suite often and to discover software bugs faster.

Chapter 5

Continuos integration with Jenkins

This chapter follows the theory of testing practices introduced in 3.4 and after continuous integration development practice is described, Jenkins automation server usage for automation of repetitive tasks like package building, test running and also to achieve better code quality in the form of static analysis tools is documented.

5.1 Continuous integration

This section introduces development practice continuous integration which requires developers to commit code to source code management (SMC) tool multiple times a day. All changes are committed to the single branch which is connected to automation tool that performs automated build. This setup enables fast detection and localization of errors and their fast repair.

The main feature of practicing continuous integration is that it enables to iterate and integrate fast. For continuous integration to work, the automated build has to be fast to keep feedback loop short. Other jobs such as application testing can also be performed as part of the automated build. Because automation server is visible for all members of the development team, it is easy for everyone to see the status of the project, what is currently happening and what has changed in the project. Last but not least, usage of automation tool enables to automate deployment process and perform deployment the same way every time and mitigating mistakes.

Continuous integration and test automation, in general, has become widely used in automotive industry as can be seen in [7] and also in software engineering in general [39]. Multiple open source solutions have been reviewed for use in this thesis, namely Jenkins, Travis and Team City. Travis CI is full featured continuous integration server, but its usage is tied to Github service and therefore is not usable for private repositories that are not hosted on GitHub. Team city is free service to some degree, but a number of executors and jobs is limited. Jenkins is open source solution that is widely used and is completely free without any restrictions. Therefore it was chosen to be used in F1/10 project.

5.2 Jenkins

Jenkins is automation server written in Java programming language. It is a fork of Hudson project, which continued to exist as commercial version under Oracle development and was discontinued in 2012. Jenkins was originally developed for use with projects written in Java, but it is a plugin based system and plugins developed by community members enable extending the system for many new use cases.

At the time of writing, Jenkins stable long term supported version is 2.46.2, which is a version that is used in this thesis. Jenkins uses a web-based user interface for both administrations, job definition, and job execution. An example of Jenkins user interface is in Figure 5.1. Jenkins also enables to use its custom command line tool for operations such as adding the job, getting job descriptions, etc.

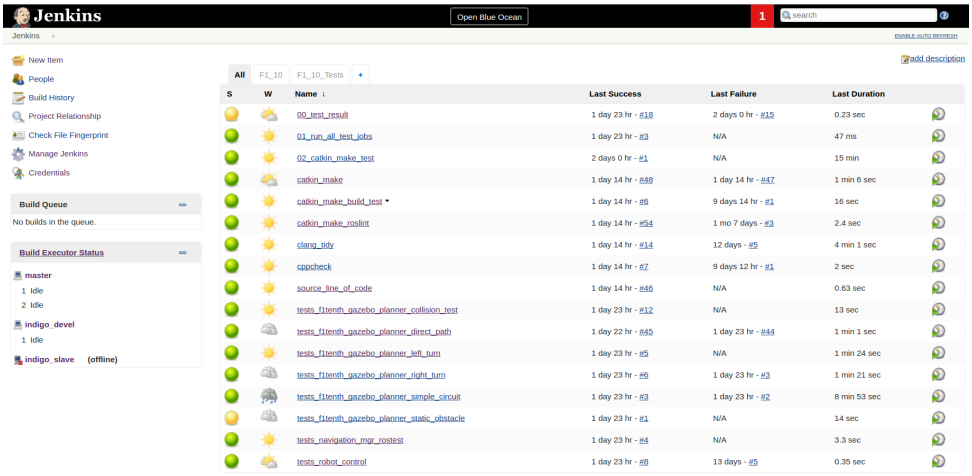


Figure 5.1: Jenkins web based user interface

Jenkins simplest execution unit is a job, also called a project. Jenkins job can range from Freestyle job, that can be used for project or test build up to Multibranch pipelines which can run jobs in parallel from multiple SCM branches in one repository. Jenkins job run can be triggered in multiple ways. The job can be run from Jenkins UI or its command line tool. It can also be setup to run automatically in the periodic manner or every time a change is committed into SCM repository, which the job is monitoring and using for getting the data which it needs for completing the job.

Jenkins is currently in transition from old style job definitions that were created using web UI and saved in XML format, which can be downloaded from Jenkins by using its command line tool. Job definition created from web UI in XML format is hard to change in the text editor, so web app has to be used for that every time, what is more, job definitions are stored at Jenkins master and specialized measures have to be taken to backup data, which add more work to the development team.

The solution for this should be Jenkins pipeline, which is a job definition

written as code that can be stored into SCM. Jenkins team is encouraging users to uses pipelines in their project. With this comes transition to the new version of web user interface called Blue Ocean and pipeline editor and linter tools, but these projects at the time of writing are far from complete. For these reasons, Jenkins jobs described in 5.3 are built as freestyle jobs.

Jenkins jobs are executed on execution nodes. Jenkins execution nodes are divided between one master and potentially multiple slave machines. Master is responsible orchestrating job execution, sending orders to slave and keeping track of running jobs and list of connected slaves. Each master can service multiple slave nodes. Jenkins as a system is used for large scale deployments, but the underlying hardware infrastructure needs to reflect how the system is used and what computational load it produces. Jenkins slave node is connected to master node either using Java web start, or SSH connection. The latter is used for connection slave nodes in the F1/10 project.

5.3 Jenkins jobs

Following sections provide the overview of jobs that have been created for the F1/10 project, their usage and usefulness are discussed. Jenkins jobs that enable to build catkin workspace, run static analyzer tools on the codebase and automatic test runner have been created. Example of job XML definition is given in Appendix C. All Jenkins jobs created for this thesis are saved on enclosed CD in jenkins/jenkins_jobs_definitions directory.

5.3.1 Build workspace and tests

Jenkins job catkin_make and catkin_make_build_test are described in this section. Catkin_make is used to build software packages located in catkin_ws folder and job named catkin_make_build_test is used to build tests.

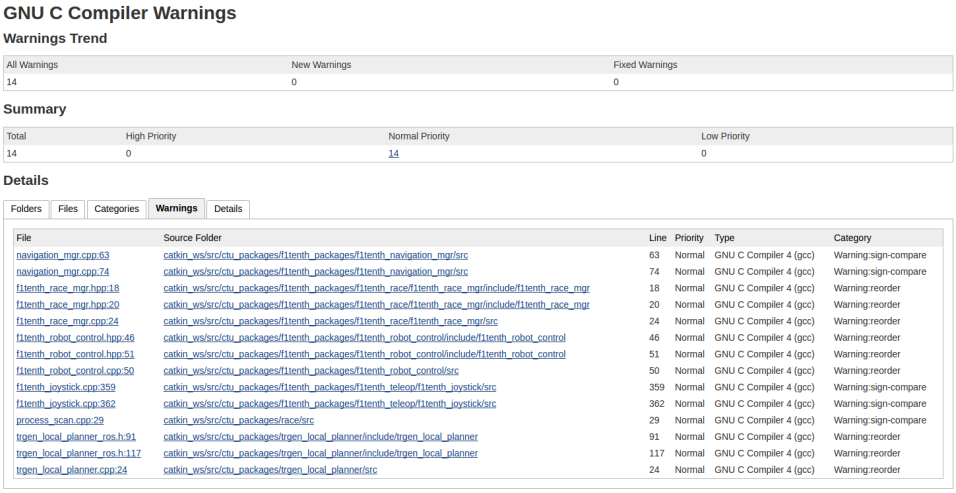
Each Jenkins job creates its workspace by default. This is not always desirable because some jobs might need to use data which were created in another job. That is why the only catkin_make job has its workspace and all other Jenkins jobs described below operates in it. Catkin_make operates as follows. Firstly, the old workspace of catkin_make Jenkins job is deleted to ensure clean build environment without any artifacts from other job runs. Git repository with F1/10 code is fetched for development branch ctu, and environment variables with the path to header files which are not in standard system locations are set. Catkin workspace is build using the following shell script.

```
#!/bin/bash
source /opt/ros/indigo/setup.bash
source /usr/share/gazebo-7/setup.sh
CPATH=$CPATH:/usr/include/gazebo-7:
/usr/include/sdformat-4.3:
/usr/include/ignition/math2
export CPATH
```

```
cd catkin_ws
catkin_make -DCMAKE_CXX_FLAGS=-Wall -DCMAKE_C_FLAGS=-Wall
catkin_make install
```

If the build is finished successfully, the catkin_make_build_test job is run automatically, and all tests in catkin_ws are built.

Warning setting for the compiler is set to '-Wall' during compilation and warnings issued by the compiler are scraped from stdout by Jenkins Warnings plugin. This plugin displays all warnings acquired during compilation in the clear web–based presentation form. In bigger projects with the greater number of developers, build of the application can be reviewed from this view and decisions about what needs to be fixed can be made effectively. An example of Warnings plugin output presented in Jenkins is in Figure 5.2.



GNU C Compiler Warnings

Warnings Trend

All Warnings	New Warnings	Fixed Warnings
14	0	0

Summary

Total	High Priority	Normal Priority	Low Priority
14	0	14	0

Details

Folders | Files | Categories | **Warnings** | Details

File	Source Folder	Line	Priority	Type	Category
navigation_mgr.cpp:63	catkin_ws/src/ctu_packages/f1tenth_packages/f1tenth_navigation_mgr/src	63	Normal	GNU C Compiler 4 (gcc)	Warning:sign-compare
navigation_mgr.cpp:74	catkin_ws/src/ctu_packages/f1tenth_packages/f1tenth_navigation_mgr/src	74	Normal	GNU C Compiler 4 (gcc)	Warning:sign-compare
f1tenth_race_mgr.hpp:18	catkin_ws/src/ctu_packages/f1tenth_packages/f1tenth_race/f1tenth_race_mgr/include/f1tenth_race_mgr	18	Normal	GNU C Compiler 4 (gcc)	Warning:reorder
f1tenth_race_mgr.hpp:20	catkin_ws/src/ctu_packages/f1tenth_packages/f1tenth_race/f1tenth_race_mgr/include/f1tenth_race_mgr	20	Normal	GNU C Compiler 4 (gcc)	Warning:reorder
f1tenth_race_mgr.cpp:24	catkin_ws/src/ctu_packages/f1tenth_packages/f1tenth_race/f1tenth_race_mgr/src	24	Normal	GNU C Compiler 4 (gcc)	Warning:reorder
f1tenth_robot_control.hpp:46	catkin_ws/src/ctu_packages/f1tenth_packages/f1tenth_robot_control/include/f1tenth_robot_control	46	Normal	GNU C Compiler 4 (gcc)	Warning:reorder
f1tenth_robot_control.hpp:51	catkin_ws/src/ctu_packages/f1tenth_packages/f1tenth_robot_control/include/f1tenth_robot_control	51	Normal	GNU C Compiler 4 (gcc)	Warning:reorder
f1tenth_robot_control.cpp:50	catkin_ws/src/ctu_packages/f1tenth_packages/f1tenth_robot_control/src	50	Normal	GNU C Compiler 4 (gcc)	Warning:reorder
f1tenth_joystick.cpp:359	catkin_ws/src/ctu_packages/f1tenth_packages/f1tenth_teleop/f1tenth_joystick/src	359	Normal	GNU C Compiler 4 (gcc)	Warning:sign-compare
f1tenth_joystick.cpp:362	catkin_ws/src/ctu_packages/f1tenth_packages/f1tenth_teleop/f1tenth_joystick/src	362	Normal	GNU C Compiler 4 (gcc)	Warning:sign-compare
process_scan.cpp:29	catkin_ws/src/ctu_packages/race/src	29	Normal	GNU C Compiler 4 (gcc)	Warning:sign-compare
trgen_local_planner_ros.h:91	catkin_ws/src/ctu_packages/trgen_local_planner/include/trgen_local_planner	91	Normal	GNU C Compiler 4 (gcc)	Warning:reorder
trgen_local_planner_ros.h:117	catkin_ws/src/ctu_packages/trgen_local_planner/include/trgen_local_planner	117	Normal	GNU C Compiler 4 (gcc)	Warning:reorder
trgen_local_planner.cpp:24	catkin_ws/src/ctu_packages/trgen_local_planner/src	24	Normal	GNU C Compiler 4 (gcc)	Warning:reorder

Figure 5.2: List of warnings issued by compiler presented in Jenkins

5.3.2 Static analysis

This section describes Jenkins jobs that are used to run static analysis software on project code base. Static analysis of the source code was theoretically introduced in section 3.4.3. Commercial programs exist that enable to analyze code for compliance with various standards such as MISRA or High Integrity C++, which are required for industry products. Open source static analyzers for C++ language have been reviewed for this thesis and roslint, cppcheck and clang tidy were chosen. Each static analysis tool has different domain of expertise and various level of noise (false positive warning) that they return. This implies that it is beneficial to use multiple tools that are complementary in their abilities.

Three static analysis tools have been reviewed and prepared for use within F1/10 project. First of them is roslint[1], which is a catkin integrated tool build

[1] roslint: http://wiki.ros.org/roslint

with cpplint program and enables to check Python and C++ source code for errors and compliance with ROS coding standards. Roslint is good in checking ROS coding standards compliance such as missing license information, wrong style of include headers or indentation. Roslint analysis is run by Jenkins job roslint

Another tool that is used is cppcheck[2]. Cppcheck does not concern itself with syntax errors in the code and primarily tries to detect bugs that are not caught by the compiler. This makes it a good addition to code checking during the build and to roslint which mainly cares about syntax checks. Cppcheck analysis is run by Jenkins job Cppcheck.

The last tool that is used is clang tidy[3]. Clang tidy is a linter tool that aspires to be an extensible framework for diagnosing and fixing programming errors, interface misuse and style violations. Thanks to its modularity, it enables to add new checks or setup the checker for custom code style guide. Checks in clang tidy are gathered into groups such as Google group that checks, if Google coding conventions are used, cppcoreguidelines which checks against the set of C++ Core Guidelines issued by C++ language creator Bjarne Stroustrup or modernize, which advocates use of modern (C++11) language constructs and much more. Clang tidy is run by Jenkins job clang_tidy.

5.3.3 Automated simulation test

Running automated tests from Jenkins turned out to be more challenging than expected. Reason for this is that even though the same computer, which was used for development and test running in 4.7, is used as Jenkins slave, invocation of commands works differently while running from Jenkins job.

Jenkins jobs are run under jenkins account which is a service account, and it doesn't have a shell assigned to it by design in Linux. A service account is used to execute services (daemon) with restricted scope and privilege. This might be an issue for applications that expect to be invoked from shell environment, as was the case when running planner test suite. Solution to this complication is to use tool that enables to disassociated program from the original terminal. Examples of these tools are screen, tmux or dtach. Dtach was used for this thesis. An example call to rostest with the dtach tool is given.

```
dtach -n socket bash -c "source /opt/ros/indigo/setup.bash; \
source ~/workspace/catkin_make/catkin_ws/devel/setup.bash; \
source /usr/share/gazebo-7/setup.sh; \
rostest f1tenth_gazebo test_f1tenth_planner_direct_path.test \
2>error_log" while test -e socket; do sleep 1; done
```

Dtach tool run with -n argument creates a new session, without attaching to it. A new session is created in which the specified program is executed. Dtach does not try to attach to the newly created session, however, and exits

[2]Cppcheck: http://cppcheck.sourceforge.net/
[3]Clang tidy: http://clang.llvm.org/extra/clang-tidy/

instead. To keep the socket session open, while loop is run that keeps the session running while the test runs inside it. Environment variables such as paths to shared libraries are not passed to the dtach session, therefore must be sourced before any ros functionality can be used.

This approach enables to run automated simulation tests from Jenkins but comes with some costs, while it is possible to redirect standard error output to file and print it after the test is finished, standard output from dtach session can't be retrieved. Test results are saved into typical xUnit result XML file and are loaded by Jenkins to display test results. Standard output from test run can be in theory retrieved from log files that ROS automatically produces, but in practice, it turns out to be difficult because the output of each ROS node is log is kept in the separate file, and therefore the sequence of commands for the complete system cannot be retrieved.

Results from all tests run by jenkins are stored in catkin workspace in build/test_results directory. Job 00_test_result crawls this directory and looks for all XML files with test results and creates the graph of test results. Each test can review individually, and information about its history which hold information about failure and how long it takes to execute can be displayed.

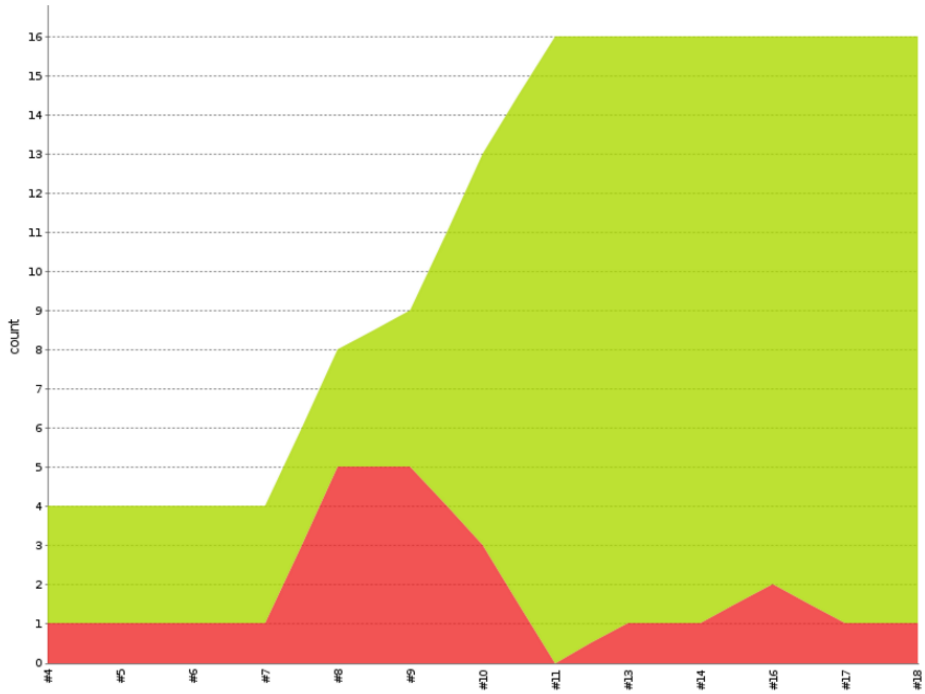


Figure 5.3: Jenkins test results

5.4 Results

This section describes how continuos integration was used in F1/10 project and what results it provided for the development team. In the time of writing this thesis, two students including the author of this thesis have been working on F1/10 project. Since each student was working on its part of the code base, continuous build of application did not bring practical benefits since merge conflicts and broken build after commit rarely happened.

All C++ code developed in F1/10 project have been continuously analyzed by tools described in 5.3.2. While some warnings have been reported by analyzers and code was reworked to solve them, no serious bugs have been found. ROS nodes developed for this thesis and described in 4.5 are written with basic C++ constructs, without dynamic allocation, complicated use of pointers and only standard data structures and containers are used, which might be the cause for the low number of warnings from analyzer tools. The complexity of the system is not at individual nodes, but in the distributed nature of the system and how all the nodes operate together.

Way to run automated simulation test from Jenkins have been developed. This enables to run planner test suite described in 4.6.3 automatically and use them as regression tests to check that changes to car software do not introduce bugs into already developed functionality.

Chapter 6

Conclusion

This thesis goal was to design the appropriate methods to test the software of autonomous car in the F1/10 project, implement these methods and use them to test car software. Robotics simulation was selected as the main testing method. Car model for simulation and necessary control software was created for use in Gazebo Simulator in combination with ROS. Simulation of multiple cars was described that is used to test car in the race like condition. Multiple simulated worlds were created to enable development and testing in various environments. ROS packages to support test automation in Gazebo simulator and creation of custom simulation scenarios were developed.

The test cases in the form of automated simulation scenarios were presented and run with software developed for the car in F1/10 project. Results of these tests were evaluated against a set of system requirements. Simulation scenarios to simplify future development of autonomous behavior needed for the race against opponent car were developed, and use case for hardware-in-the-loop was presented. To support code quality and team cooperation in F1/10 project, continuos integration development practice was introduced, and jobs for Jenkins automation server were set up to automate test running, code compilation and static analysis.

Presented approach can be modified to accommodate needs for the testing full-scale car. Gazebo simulator enables to create countryside or city maps and can be used for development of autonomous vehicles, especially in the phase of prototyping, testing new ideas and architectures, the variety of simulated sensors provides a wide range of options, and different configuration can be quickly tested with the simulated car. Last but not least, simulation can be used for Hardware-in-the-loop testing to verify software on target hardware. The simulation software will keep improving in the future, and this will bring new possibilities as well as better portability of solutions developed inside simulation directly to the real world.

Bibliography

[1] ISO, "26262-1:2011 road vehicles – functional safety," International Organization for Standardization, Geneva, CH, Standard, 2011.

[2] S. R. Richter, V. Vineet, S. Roth, and V. Koltun, "Playing for data: Ground truth from computer games," in *European Conference on Computer Vision*. Springer, 2016, pp. 102–118.

[3] D. Wellers. (2016) Autonomous vehicles: Accelerating into the mainstream. [Online]. Available: https://www.forbes.com/sites/sap/2016/12/08/autonomous-vehicles-accelerating-into-the-mainstream/#6269155c4ba5

[4] R. News. (2016) Autonomous cars. [Online]. Available: http://www.ros.org/news/robots/autonomous-cars/

[5] S. Paepcke, "Michael aeberhard (bmw): Automated driving with ros at bmw," https://www.osrfoundation.org/michael-aeberhard-bmw-automated-driving-with-ros-at-bmw/, 2016.

[6] A. Santos, A. Cunha, N. Macedo, and C. Lourenço, "A framework for quality assessment of ros repositories," in *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*. IEEE, 2016, pp. 4491–4496.

[7] H. Altinger, F. Wotawa, and M. Schurius, "Testing methods used in the automotive industry: Results from a survey," in *Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing*, ser. JAMAICA 2014. New York, NY, USA: ACM, 2014, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/2631890.2631891

[8] S. Currie, "Developments in car hacking," SANS Institute, Tech. Rep., 2015. [Online]. Available: https://www.sans.org/reading-room/whitepapers/ICS/developments-car-hacking-36607

[9] M. Broy, "Challenges in automotive software engineering," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 33–42.

[10] B. Hardung, T. Kölzow, and A. Krüger, "Reuse of software in distributed embedded automotive systems," in *Proceedings of the 4th ACM International Conference on Embedded Software*, ser. EMSOFT '04. New York, NY, USA: ACM, 2004, pp. 203–210. [Online]. Available: http://doi.acm.org/10.1145/1017753.1017787

[11] S. Currie, "The impact of vehicle recalls on the automotive market," NADA, Tech. Rep., 2014. [Online]. Available: http://www.autonews.com/Assets/pdf/NADA%20UCG_WhitePaper_Impact%20of%20Vehicle%20Recalls.pdf

[12] U. S. G. U. Army, *Systems Engineering Fundamentals*. CreateSpace Independent Publishing Platform, 2013.

[13] I. C. Society, *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, 2014.

[14] D. Huizinga and A. Kolawa, *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Pr, 2007.

[15] M. Ould, C. Unwin, and B. C. S. W. G. on Testing, *Testing in Software Development*, ser. British Computer Society Monog. Cambridge University Press, 1986.

[16] IEEE, "Ieee standard computer dictionary: A compilation of ieee standard computer glossaries," *IEEE Std 610*, 1991.

[17] G. J. Myers, *The Art of Software Testing, Second Edition*. Wiley, 2004.

[18] E. Bringmann and A. Krämer, "Model-based testing of automotive systems," in *2008 1st International Conference on Software Testing, Verification, and Validation*, April 2008, pp. 485–493.

[19] C. Berger, M. Chaudron, R. Heldal, O. Landsiedel, and E. M. Schiller, "Model-based, composable simulation for the development of autonomous miniature vehicles," in *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative M&S Symposium*, ser. DEVS 13. San Diego, CA, USA: Society for Computer Simulation International, 2013, pp. 17:1–17:8. [Online]. Available: http://dl.acm.org/citation.cfm?id=2499634.2499651

[20] J. A. Sokolowski and C. M.Banks, *Principles of Modeling and Simulation: A Multidisciplinary Approach*. Wiley, 2011.

[21] J. Banks, J. S. Carson, B. L. Nelson, and D. M. Nicol, *Discrete-Event System Simulation (3rd Edition)*. Prentice Hall, 2000.

[22] S. Higgins. (2017) Google has its own lidar for autonomous cars. [Online]. Available: http://www.spar3d.com/news/lidar/google-lidar-autonomous-cars/

[23] J. Condliffe. (2016) Tesla announces new sensors and puts the brakes on autopilot. [Online]. Available: https://www.technologyreview.com/s/602703/tesla-announces-new-sensors-and-puts-the-brakes-on-autopilot/

[24] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," *CoRR*, vol. abs/1604.07316, 2016. [Online]. Available: http://arxiv.org/abs/1604.07316

[25] P. Koopman and M. Wagner, "Challenges in autonomous vehicle testing and validation," *SAE International Journal of Transportation Safety*, vol. 4, no. 2016-01-0128, pp. 15–24, 2016.

[26] T. Weigl, "Development process for autonomous vehicles," Master's thesis, Technical University of Munich, Arcisstraße 21, Munich, Bavaria, 80333, Germany, 2014.

[27] C. Bergenhem, R. Johansson, A. Söderberg, J. Nilsson, J. Tryggvesson, M. Törngren, and S. Ursing, "How to reach complete safety requirement refinement for autonomous vehicles," in *CARS 2015-Critical Automotive applications: Robustness & Safety*, 2015.

[28] P. Els. (2017) Iso 26262 - addressing concerns around emerging technologies. [Online]. Available: https://www.automotive-iq.com/electrics-electronics/articles/iso-26262-addressing-concerns-around-emerging-technologies

[29] F. A. Administration, "Ac 25.1309-1a - system design and analysis," Federal Aviation Administration, ANM-112, Northwest Mountain Region - Transport Airplane Directorate, Tech. Rep., 1988.

[30] P. Nelson. (2016) Just one autonomous car will use 4,000 gb of data/day. [Online]. Available: http://www.networkworld.com/article/3147892/internet/one-autonomous-car-will-use-4000-gb-of-dataday.html

[31] P. Christiano, Z. Shah, I. Mordatch, J. Schneider, T. Blackwell, J. Tobin, P. Abbeel, and W. Zaremba, "Transfer from simulation to real world through learning deep inverse dynamics model," *CoRR*, vol. abs/1610.03518, 2016. [Online]. Available: http://arxiv.org/abs/1610.03518

[32] M. Johnson-Roberson, C. Barto, R. Mehta, S. N. Sridhar, and R. Vasudevan, "Driving in the matrix: Can virtual worlds replace human-generated annotations for real world tasks?" *CoRR*, vol. abs/1610.01983, 2016. [Online]. Available: http://arxiv.org/abs/1610.01983

[33] F. Tam. (2017) F1/10 competition rules. [Online]. Available: http://f1tenth.org/rules

[34] M. Vajnar, "Model car for the f1/10 autonomous car racing competition," Master's thesis, Czech Technical University, FEE, 2017.

[35] ROS.org. (2014) What is ros? [Online]. Available: http://wiki.ros.org/ROS/Introduction

[36] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, 2009, p. 5.

[37] E. Kaltenegger, "Physical and graphical simulation of an ackermann steered vehicle," Master's thesis, TU Wien, 2016.

[38] B. Gerkey. (2016) Why ros 2.0? [Online]. Available: http://design.ros2.org/articles/why_ros2.html

[39] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 2016, pp. 426–437.

Appendix A

CD content

```
/
├── jenkins
│   └── jenkins_jobs_definitions
├── catkin_ws
│   └── src
│       ├── 3rd_party ...  This directory holds
│       │                  3rd_party ROS packages
│       │                  with Gazebo sensor
│       │                  definitions and packages
│       │                  developed by other
│       │                  member of F1/10 project.
│       └── f1tenth_packages ...  Software packages
│           │                     developed for this
│           │                     thesis.
│           ├── f1tenth_description ...  URDF description of
│           │                            simulated car.
│           ├── f1tenth_gazebo ...  Gazebo worlds, test
│           │                       cases and launch files
│           │                       for simulation.
│           ├── f1tenth_gazebo_plugins ...  Perfect odometry plugin
│           │                               and collision detection
│           │                               node.
│           ├── f1tenth_navigation ...  Maps for simulated
│           │                           worlds and Rviz
│           │                           configuration file.
│           ├── f1tenth_navigation_mgr ...  Navigation manager.
│           ├── f1tenth_race
│           │   ├── f1tenth_race_mgr ...  Race manager.
│           │   └── f1tenth_race_starter ...  Race starter.
│           └── f1tenth_teleop ...  3rd_party teleoperation
│                                   packages for keyboard
│                                   and joystick.
└── software_testing_for_embedded_applications_in_autonomous_vehicles.pdf
```

Appendix B

Installation instruction

This section describes how to install the software required to run simulations as described in the thesis. All software was tested on ROS Indigo with Gazebo 7. Target operating system for ROS Indigo is Ubuntu 14.04, if you do not own machine with this operating system, virtualization can be used.

Installing Ubuntu 14.04 on the Virtual box is straightforward, and many tutorials can be found online. This is preferred an option for Windows users. Linux users can create Chroot environment with Ubuntu 14.04 as described in[1]. Use the second chapter to install Ubuntu 14.04 to Chroot.

All packages needed to build and use the software in F1/10 project have been placed into the debian package that enables to install all required software and its dependencies at once. Once you have Ubuntu 14.04 installed and running, open terminal and add rtime Debian package repository[2].

```
echo deb http://rtime.felk.cvut.cz/debian unstable main  \
> /etc/apt/sources.list.d/rtime-debs.list
wget -O - https://rtime.felk.cvut.cz/debian/archive-key.asc \
 | apt-key add -
```

Once the repository is added, update the package list and install.

```
sudo apt-get update
sudo apt-get install f1tenth-ctu-deb-sources
```

Package f1tenth-ctu-deb-sources adds ROS and Gazebo repositories to package list. Update package list again and install f1tenth-ctu-devel package.

```
sudo apt-get update
sudo apt-get install f1tenth-ctu-devel
```

This package will install ROS Indigo, Gazebo 7 and all dependencies required to build catkin workspace for F1/10 project. Source ROS and Gazebo setup.bash scripts and add gazebo headers to CPATH so compiler can locate required headers. It is recommended to add following commands to .bashrc file.

```
source /opt/ros/indigo/setup.bash
```

[1] ROS Indigo in Chroot: http://wiki.ros.org/ROS/Tutorials/InstallingIndigoInChroot
[2] Rtime wiki: https://rtime.felk.cvut.cz/wiki/index.php/Debian_packages_repository

```
source /usr/share/gazebo-7/setup.sh
CPATH=$CPATH:/usr/include/gazebo-7\
:/usr/include/sdformat-4.3:/usr/include/ignition/math2
export CPATH
```

Now, change directory to catkin_ws and call catkin_make to build all packages in this workspace.

```
catkin_make
```

Appendix C

Jenkins job example

Example of XML definition for Jenkins job catkin_make.

```
<?xml version='1.0' encoding='UTF-8'?>
<project>
  <actions/>
  <description>Fast catkin make, check commit,
  build if relevant changes found</description>
  <keepDependencies>false</keepDependencies>
  <properties/>
  <scm class="hudson.plugins.git.GitSCM" plugin="git@3.3.0">
    <configVersion>2</configVersion>
    <userRemoteConfigs>
      <hudson.plugins.git.UserRemoteConfig>
        <url>git@rtime.felk.cvut.cz:f1tenth</url>
        <credentialsId>cfba3c96-3dc6-4dc0-ac81-1b760bfe47d3
        </credentialsId>
      </hudson.plugins.git.UserRemoteConfig>
    </userRemoteConfigs>
    <branches>
      <hudson.plugins.git.BranchSpec>
        <name>*/ctu</name>
      </hudson.plugins.git.BranchSpec>
    </branches>
    <doGenerateSubmoduleConfigurations>false
    </doGenerateSubmoduleConfigurations>
    <submoduleCfg class="list"/>
    <extensions>
      <hudson.plugins.git.extensions.impl.DisableRemotePoll/>
    </extensions>
  </scm>
  <assignedNode>indigo_devel</assignedNode>
  <canRoam>false</canRoam>
  <disabled>false</disabled>
  <blockBuildWhenDownstreamBuilding>fals
  e</blockBuildWhenDownstreamBuilding>
```

```
  <blockBuildWhenUpstreamBuilding>false
  </blockBuildWhenUpstreamBuilding>
  <triggers/>
  <concurrentBuild>false</concurrentBuild>
  <builders>
    <hudson.tasks.Shell>
      <command>#!/bin/bash
source /opt/ros/indigo/setup.bash
source /usr/share/gazebo-7/setup.sh
CPATH=$CPATH:/usr/include/gazebo-7:
/usr/include/sdformat-4.3:
/usr/include/ignition/math2
export CPATH

cd catkin_ws
catkin_make -DCMAKE_CXX_FLAGS=-Wall -DCMAKE_C_FLAGS=-Wall
catkin_make install

</command>
    </hudson.tasks.Shell>
  </builders>
  <publishers>
    <hudson.plugins.warnings.WarningsPublisher
    plugin="warnings@4.62">
      <healthy></healthy>
      <unHealthy></unHealthy>
      <thresholdLimit>low</thresholdLimit>
      <pluginName>[WARNINGS] </pluginName>
      <defaultEncoding></defaultEncoding>
      <canRunOnFailed>false</canRunOnFailed>
      <usePreviousBuildAsReference>false
      </usePreviousBuildAsReference>
      <useStableBuildAsReference>false
      </useStableBuildAsReference>
      <useDeltaValues>false</useDeltaValues>
      <thresholds plugin="analysis-core@1.86">
        <unstableTotalAll></unstableTotalAll>
        <unstableTotalHigh></unstableTotalHigh>
        <unstableTotalNormal></unstableTotalNormal>
        <unstableTotalLow></unstableTotalLow>
        <unstableNewAll></unstableNewAll>
        <unstableNewHigh></unstableNewHigh>
        <unstableNewNormal></unstableNewNormal>
        <unstableNewLow></unstableNewLow>
        <failedTotalAll></failedTotalAll>
        <failedTotalHigh></failedTotalHigh>
```

```
        <failedTotalNormal></failedTotalNormal>
        <failedTotalLow></failedTotalLow>
        <failedNewAll></failedNewAll>
        <failedNewHigh></failedNewHigh>
        <failedNewNormal></failedNewNormal>
        <failedNewLow></failedNewLow>
      </thresholds>
      <shouldDetectModules>false</shouldDetectModules>
      <dontComputeNew>true</dontComputeNew>
      <doNotResolveRelativePaths>true
      </doNotResolveRelativePaths>
      <includePattern></includePattern>
      <excludePattern></excludePattern>
      <messagesPattern></messagesPattern>
      <parserConfigurations/>
      <consoleParsers>
        <hudson.plugins.warnings.ConsoleParser>
          <parserName>GNU C Compiler 4 (gcc)</parserName>
        </hudson.plugins.warnings.ConsoleParser>
      </consoleParsers>
    </hudson.plugins.warnings.WarningsPublisher>
  </publishers>
  <buildWrappers>
    <hudson.plugins.ws__cleanup.PreBuildCleanup
    plugin="ws-cleanup@0.33">
      <deleteDirs>false</deleteDirs>
      <cleanupParameter></cleanupParameter>
      <externalDelete></externalDelete>
    </hudson.plugins.ws__cleanup.PreBuildCleanup>
  </buildWrappers>
</project>
```