DEGREE PROJECT IN ELECTRICAL ENGINEERING,
SECOND CYCLE, 30 CREDITS
*STOCKHOLM, SWEDEN 2018*

# Model Based Systems Engineering Approach to Autonomous Driving

Application of SysML for trajectory planning of autonomous vehicle

**SARANGI VEERMANI LEKAMANI**

**KTH ROYAL INSTITUTE OF TECHNOLOGY**
**SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE**

## Author

Sarangi Veeramani Lekamani savl@kth.se
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology

## Place for Project

Södertälje, Sweden
AVL MTC AB

## Examiner

Ingo Sander
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology

## Supervisor

George Ungureanu
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology

## Industrial Supervisor

Hakan Sahin
AVL MTC AB

# Abstract

Model Based Systems Engineering (MBSE) approach aims at implementing various processes of Systems Engineering (SE) through diagrams that provide different perspectives of the same underlying system. This approach provides a basis that helps develop a complex system in a systematic manner. Thus, this thesis aims at deriving a system model through this approach for the purpose of autonomous driving, specifically focusing on developing the subsystem responsible for generating a feasible trajectory for a miniature vehicle, called AutoCar, to enable it to move towards a goal.

The report provides a background on MBSE and System Modeling Language (SysML) which is used for modelling the system. With this background, an MBSE framework for AutoCar is derived and the overall system design is explained. This report further explains the concepts involved in autonomous trajectory planning followed by an introduction to Robot Operating System (ROS) and its application for trajectory planning of the system.

The report concludes with a detailed analysis on the benefits of using this approach for developing a system. It also identifies the shortcomings of applying MBSE to system development. The report closes with a mention on how the given project can be further carried forward to be able to realize it on a physical system.

**Keywords:** *MBSE; SysML; Trajectory planning; ROS; Autonomous driving*

# Sammanfattning

Modellbaserade systemteknikens (MBSE) inriktning syftar till att implementera de olika processerna i systemteknik (SE) genom diagram som ger olika perspektiv på samma underliggande system. Detta tillvägagångssätt ger en grund som hjälper till att utveckla ett komplext system på ett systematiskt sätt. Sålunda syftar denna avhandling att härleda en systemmodell genom detta tillvägagångssätt för autonom körning, med särskild inriktning på att utveckla delsystemet som är ansvarigt för att generera en genomförbar ban för en miniatyrbil, som kallas AutoCar, för att göra det möjligt att nå målet.

Rapporten ger en bakgrund till MBSE and Systemmodelleringsspråk (SysML) som används för modellering av systemet. Med denna bakgrund, MBSE ramverket för AutoCar är härledt och den övergripande systemdesignen förklaras. I denna rapport förklaras vidare begreppen autonom banplanering följd av en introduktion till Robot Operating System (ROS) och dess tillämpning för systemplanering av systemet.

Rapporten avslutas med en detaljerad analys av fördelarna med att använda detta tillvägagångssätt för att utveckla ett system. Det identifierar också bristerna för att tillämpa MBSE på systemutveckling. Rapporten stänger med en omtale om hur det givna projektet kan vidarebefordras för att kunna realisera det på ett fysiskt system.

**Nyckelord:** *MBSE; SysML; Banplanering; ROS; Autonom körning*

# Table of Contents

**1 Introduction .......... 1**
**1.1 Background .......... 2**
**1.2 Problem Statement .......... 6**
**1.3 Purpose .......... 7**
**1.4 Goals .......... 7**
**1.5 Methodology .......... 8**
**1.6 Report Outline (Disposition) .......... 8**

**2 Introduction to Model Based Systems Engineering .......... 10**
**2.1 Advantages of Representing Systems Through Models .......... 10**
**2.2 System Modeling Language .......... 11**
2.2.1 SysML – an extension of UML .......... 12
2.2.2 Different types of SysML diagrams .......... 13

**3 Defining a Model for AutoCar .......... 15**
**3.1 Background on EAST-ADL .......... 15**
3.1.1 Purpose of EAST-ADL in project .......... 16
**3.2 Diagrams Representing Vehicle Level of the System .......... 17**
3.2.1 Top Level req .......... 17
3.2.2 Collision Avoidance uc .......... 18
**3.3 Diagrams Representing Analysis Level of the System .......... 19**
3.3.1 Detailed req .......... 20
3.3.2 AutoCar Subsystem Composition bdd .......... 20
3.3.3 Planning Subsystem Interactions ibd .......... 21
3.3.4 AutoCar Overall Functionality act .......... 25

**4 Planning Subsystem of AutoCar .......... 27**
**4.1 Path Planning .......... 28**
**4.2 Velocity generation using Dynamic Window Approach .......... 29**
**4.3 Collision avoidance .......... 30**
4.3.1 Find a new path around obstacle .......... 30
4.3.2 Follow obstacle .......... 31
4.3.3 Wait for obstacle to cross .......... 32
**4.4 Costmap maintenance .......... 32**

**5 Representation of Trajectory Planning in the System Model .......... 35**
**5.1 Diagrams Representing Design Level of Planning Subsystem .......... 35**
5.1.1 Functional design architecture of the sub-system .......... 35
*5.1.1.1 Planning req .......... 35*
*5.1.1.2 Perception req .......... 36*
*5.1.1.3 Map Server req .......... 37*
*5.1.1.4 Map Server uc .......... 38*
*5.1.1.5 Perception bdd .......... 39*
*5.1.1.6 Planning Block Diagrams .......... 39*
*5.1.1.7 Planning act .......... 42*
*5.1.1.8 Planning seq .......... 44*
*5.1.1.9 Planning stm .......... 45*
5.1.2 Hardware design architecture of planning subsystem .......... 47
*5.1.2.1 Vehicle Construction bdd .......... 47*
**5.2 Diagrams Representing Implementation Level of Planning Subsystem. 48**

5.2.1 Background on ROS ..... 49
5.2.2 Block definition diagrams representing the application of ROS ..... 50

**6 Model Evaluation ..... 56**
**6.1 Model Completeness ..... 56**
**6.2 Model Correctness ..... 65**
**6.3 Model Summary ..... 66**

**7 System Design Analysis and Conclusion ..... 70**
**7.1 Analysis on System Design Phases ..... 70**
**7.2 Analysis on System Design Methodology ..... 71**
**7.3 Conclusion ..... 72**
**7.4 Future Scope of AutoCar Project Execution ..... 73**

**References**

# List of Acronyms and Abbreviations

| | |
|---|---|
| ABS | Antilock Braking System |
| ACC | Adaptive Cruise Control |
| act | Activity diagram |
| ADAS | Advanced driver assistance systems |
| AD | Autonomous Driving |
| ADLs | Architectural Description Languages |
| AUTOSAR | Automotive Open System Architecture |
| bdd | Block definition diagram |
| CC | Cruise Control |
| DARPA | Defense Advanced Research Projects Agency |
| DAS | Driver Assistance Systems |
| DWA | Dynamic Window Approach |
| EPL | Eclipse Public License |
| GPU | General Processing Unit |
| GUI | Graphical User Interface |
| HMI | Human Machine Interaction |
| INCOSE | International Council of Systems Engineering |
| ibd | Internal block diagram |
| IMU | Inertial Measurement Unit |
| IDE | Integrated Development Environment |
| Lidar | Light Detection and Ranging |
| MBSE | Model Based Systems Engineering |
| MUX | multiplexer |
| OMG | Object Management Group |
| par | Parametric diagram |
| pkg | Package diagram |
| req | Requirement diagram |
| RGBD | Red Green Blue Depth |
| ROS | Robot Operating System |
| rpm | revolution per minute |
| RRT | Rapidly-exploring Random Trees |
| SAE | Society of Automobile Engineers |
| SCs | Software Components |
| SE | System Engineering |
| seq | Sequence diagram |
| stm | State machine diagram |
| SysML | System Modeling Language |
| uc | Use case diagram |
| UML | Unified Modifying Language |
| 4WD | Four-wheel drive |

# 1 Introduction

Autonomous driving is a popular research area to ensure safe traffic on roads with almost zero accidents. Every year, car related accidents account for the death of about 1.2 million people all over the world [1]. This is a significantly huge number which needs due attention.

Present generation vehicles have at the least 100 microprocessors managing brakes, cruise control and transmission, and communicating with each other [1]. To add on, these vehicles have as many as 5 to 10 million lines of code running. It is a complex system of several sub-systems and its complexity is expected to increase with the introduction of autonomous driving technology. It, therefore, requires a systematic approach that follows a good design methodology to ensure that the complete system is built as per the requirements laid out by a customer [30]. Model Based Systems Engineering (MBSE) is one such engineering technique that aims at designing the system from a holistic perspective and provides a common platform for developing a system without ambiguity. The thesis report focuses on deriving a system model using this approach, using System Modeling Language (SysML).

The functionality of autonomous driving can be broadly broken down into three domains: perception, planning and control [3]. *Vehicle perception* is achieved through sensors which collect data regarding vehicle's current state and its external environment. From these sensor readings, the collected data is converted into a vehicle perceivable format. *Planning* focusses on making decisions on how to move from a given position to a destination, safely by avoiding any potential collision. *Control* is responsible for ensuring that the vehicle reacts in accordance with the decisions made, by quantitatively setting values for various vehicle actuators (brake, steering, throttle valve).

In this thesis, the focus is on *vehicle planning* for autonomous driving and hence, a system model, specifically, for this module will be designed, in which a feasible trajectory for the vehicle to move along is planned. Trajectory is defined as a path followed by an object with respect to time. The approximations that will be made during the modeling process are as follows:

- Bicycle model [5] of the vehicle will be used for generating velocity values, i.e. velocity values will be individually generated for the virtual front wheel and virtual back wheel. This model is thus used for generating an executable trajectory.

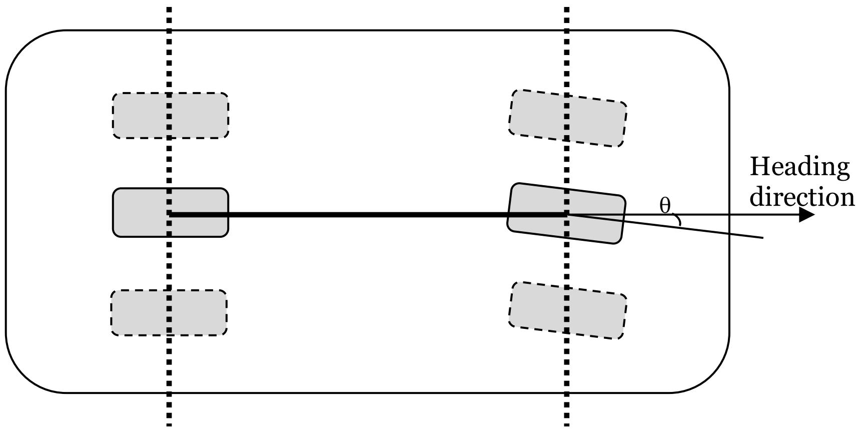


**Fig 1.1:** Bicycle model of the vehicle where the two rear wheels have been approximated to one rear wheel (in the center) and the same with the front wheels. The front wheel is steerable but not the rear wheel.

- The vehicle is considered as a moving point object while planning a path in the given map.
- The vehicle always starts driving from the location (0, 0).

## 1.1 Background

In [6], it has been mentioned that researches on autonomous driving have been existing since the 1980's but it was only in the past 12 years that this technology has caught the attention for commercialization. In 2005, when Defense Advanced Research Projects Agency (DARPA) [2] organized the Grand Challenge [7], wherein, the participating teams had to compete in driving the vehicle autonomously through a predefined dessert route, 5 out of 23 teams were successful in completing the challenge. In a following challenge by the DARPA, called the Urban Challenge [10], it required the teams to drive an autonomous vehicle capable of navigating in city traffic, performing complex maneuvers like parking, negotiating intersections, merging, and 6 teams were successful in completing it. These, in turn, have encouraged the development of Google's self-driving car, Waymo [11] and Tesla's autopilot functionality, which have caught considerable media attention. Waymo's

vehicles, however, have operational design domain, i.e. they have certain well-defined boundaries for them to operate reliably. This includes geographical boundaries, weather conditions, and speed range. Out of these well-defined boundaries the vehicles refuse to function. Expanding this operational design domain is something which is currently being worked upon by Waymo.
Society of Automobile Engineers (SAE) have defined 5 levels of vehicle autonomy depending on its functionalities and/or capabilities [15]:

**Level 0:** At this level, the vehicle lacks any control or automation. It relies purely on driver's capabilities to drive the vehicle efficiently and carefully.

**Level 1:** There are available systems in the vehicle called the driver assistance systems (DAS), which assist the driver with its minimal vehicle control functionalities and makes driving an easier experience. Examples for DAS are Antilock Braking System (ABS), Cruise Control (CC).

**Level 2:** Vehicle's control functionalities are quite advanced at this level and is capable of sensing and understanding the environment in a very limited manner. ADAS functionalities like Adaptive CC (ACC) and emergency braking are some examples.

**Level 3:** At this stage, the vehicle has partial automation capabilities where the vehicle can free the work of human driver at ideal situations but expects driver's responsibility under unexpected situations or emergencies. Vehicle's sensing capabilities are quite mature at this stage. Autopilot falls under this category.

**Level 4:** Here, human driver's intervention is expected under bad weather conditions, but the vehicle can mostly take care of driving. So, the vehicle is still like a regular one with steering, break and accelerator.

**Level 5:** This is the highest level of automation where no human intervention is expected or allowed, and the vehicle doesn't need any steering wheel, brake or accelerator at this stage as the vehicle's driving functionality is quite reliable.

Today's commercial vehicles have achieved, at the maximum, third level of autonomy. But, if trajectory planning and trajectory following modules can be completely automated, in addition to the present advancement in perception systems, vehicles can achieve level 4 autonomy.

In a journal by S. Behre and M. Törngren [8], a functional architectural overview of an autonomous vehicle is presented. It emphasizes the importance of MBSE for such a complex system development by comparing two models which have a different distribution of the various individual components responsible for autonomous driving (AD), between the functionalities of cognitive driving intelligence and vehicle platform. The comparison shows how the two models have different levels of abstraction due to the varied distributions.

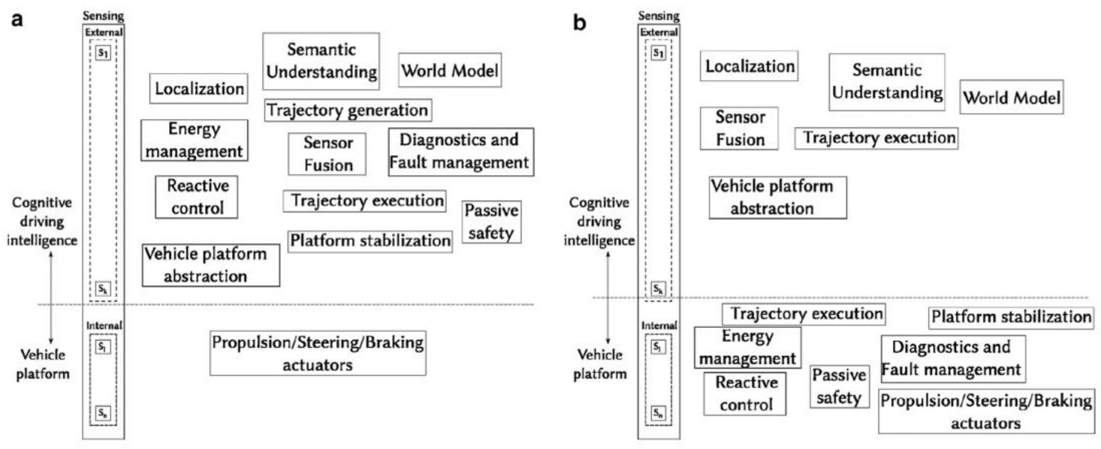


**Fig 1.2:** A comparison of distribution of various functional components between the two layers: cognitive driving intelligence and vehicle platform. (b) provides more abstraction and modularity compared to (a). (Behre et al., 2016)

It is therefore, essential to take system architecture into due consideration and approach such complex systems from a top-down perspective to ensure high level of abstraction and reusability. MBSE is one such interdisciplinary approach to engineer a system using models that capture multiple views of the underlying system through requirements, design, analysis and verification activities. In [16], a comparison between the traditional document-based approach and model-based approach to the systems engineering process is made. Here, it is claimed that in document-based approach, the system information is spread across several documents. Traceability and change impact assessment of the system becomes nearly impossible with the traditional approach. In MBSE, since, the different models are linked to one another, system maintenance becomes easier. SysML is a language that is used for developing models through the MBSE process. According to Object Management Group (OMG) [17] SysML is defined as follows:

> *"SysML is a general-purpose graphical modeling language for specifying, analyzing, designing, and verifying complex systems that may include hardware, software, information, personnel, procedures, and facilities. The language provides graphical representations with a semantic foundation for modeling system requirements, behavior, structure, and parametrics, which is used to integrate with other engineering analysis models".*

Since the specific module of interest for this thesis is trajectory planning for autonomous vehicles, several papers, reports and articles were studied to understand the decomposition of trajectory planning problem. The breakdown can be as follows:

1. **Lateral and longitudinal components**: [5], [8] and [9] discuss about breaking down the trajectory of a vehicle into lateral and longitudinal components. Where, [5] discusses in detail on modes of controlling lateral and longitudinal dynamics of a vehicle, [8] discusses on the actuators that act on controlling the lateral and longitudinal accelerations and [9] discusses about lateral and longitudinal planners for performing lane change and overtaking maneuvers.
   This decomposition is necessary from the perspective of control of actuators and for planning maneuvers like overtaking and lane departures in a multi-lane road.
2. **Velocity and path planning problem:** [4] discusses about decomposition of trajectory planning into two independent planning problems, path and velocity planning. This decomposition is applied in [12] and [13] to plan trajectory of autonomous vehicle.
   This decomposition is logical because defining a path alone is insufficient to make the vehicle move. This breakdown assists in real

time planning of vehicle's motions, thereby avoiding both static and dynamic obstacles. Therefore, in this thesis, the simultaneous decomposition of path and velocity is focused to ensure identifying a trajectory which is executable.

3. **Online and offline planning:** Offline planning is done before the start of the journey, giving the vehicle a vague idea on how to go about from start to destination (under ideal road conditions). Whereas, online planning does a detailed and/or modified planning (when encountered with new obstacles). In [14], as the environment is explored on the go (due to unavailability of a detailed map of the environment), only online planning is possible. For this thesis, as a detailed map is available before start of journey, path planning happens offline, and generation of velocity and handling of dynamic obstacles happens online.

## 1.2 Problem Statement

A miniature vehicle is required to drive autonomously within a confined space, like a building or closed work place, by avoiding collisions. The vehicle which is being considered for this purpose has a chassis of BSD Viper 1/10 [22] and it is fitted with Nvidia Jetson TX2 Developer Kit. The subject vehicle is called AutoCar.

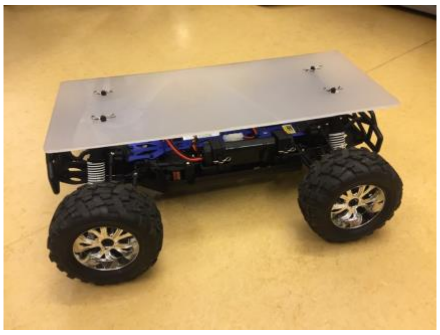


**Fig 1.3:** BSD Viper 1/10 modified to AutoCar. A basic vehicle construction is presented in the picture. Additional hardware components that are required will be decided as a part of the thesis.

The boundary conditions within which AutoCar operates autonomously are defined here. A well-defined digital map and the destination location are available to the vehicle before its start. The path through which the vehicle

drives, is a two-way road with a defined lane marking. The lanes are wide enough to enable the vehicle to turn around corners without having to cross the lane. Unmapped entities that are detected, move at a constant linear velocity (angular velocity remains zero) where velocity values equate to zero if the entity is stationary.
Such an autonomous vehicle has a high level of complexity due to the presence of various sub-systems and their communication with one another at appropriate time instances. To design such a complex system, a systematic approach is essential. The MBSE approach is expected to assist such a system design and development, thus being a solution to the development problem. In this thesis, it is expected to derive a system model, using MBSE techniques, capable of generating trajectory for the vehicle to autonomously travel from start to destination by avoiding possible obstacles.

## 1.3 Purpose

In order to assist a complex system development through systematic and well-defined processes, application of MBSE to facilitate such a system development, capable of autonomous trajectory planning, will be demonstrated through this thesis. SysML is the language that is adopted to represent the corresponding model elements.
As a final step in this thesis, an analysis on the impact of different design phases on the system development is made, and an understanding on the effects of chosen MBSE methodology is achieved.

## 1.4 Goals

The goal set for this project are broken into primary goal, that which needs to be completed by the end of the thesis project, and secondary goal, which adds value to the primary goal.
Primary goals are as follows:

1. Create a base MBSE framework for the complete system to enable future system development activities wherein, the overall autonomous vehicle requirements are specified, and the required subsystems are identified.
2. Elaborate the subsystem responsible for trajectory planning. This includes specifying detailed requirements, performing an analysis on the system

functions and defining the system architecture. Detailed diagrams representing the realization of autonomous trajectory planning onto the system are to be shown – inputs and outputs required, sequence of actions required to achieve the functionality, and the composition of trajectory planning subsystem.

3. Represent the ROS application for developing software components responsible for trajectory planning through SysML diagrams. Representing the ROS usage in the model is a novel part of the thesis project.
4. Evaluate the model in terms of its completeness and correctness.
5. Assess the impact of the different design phases on the system development and effects of the chosen methodology used for designing the system under consideration.

Secondary goals are as follows:

1. List the verification test specifications.
2. Develop the software code for the designed model.
3. Setup the simulation environment and simulate the trajectory planning functionality of the vehicle.
4. Verify the simulation results against the requirements.

## 1.5 Methodology

This is an applied research that aims at applying MBSE for the context of autonomous driving, specifically in developing trajectory planning system (functionality) for the vehicle and understanding how this application has assisted and/or hindered the system development.

The final system model is evaluated as follows:

1. The specified high-level requirements are broken down into fine executable requirements.
2. Each requirement is captured by at least one SysML diagram.
3. The system model designed is assessed for its reliability, efficiency and performance.

## 1.6 Report Outline (Disposition)

In Chapter 2, details on MBSE, its relevance to engineering a system, identified advantages of using it for development, will be discussed. This chapter will also touch upon the basics of SysML language and details on how it assists in modeling a system. Chapter 3 shows the models that assist in viewing AutoCar from an external perspective, followed by models showing the subsystem view of the vehicle. Thus, here, MBSE context for AutoCar is established. In Chapter 4, details on trajectory planning functionality and how it is being approached in this paper are explained in detail. Models representing a detailed design view and the implementation of the trajectory planning module are shown in chapter 5. Chapter 6 focusses on evaluating the system developed as a part of the thesis. The last chapter (Chapter 7) concludes on how the different design phases have assisted in the system development and understanding the benefits of applying the chosen MBSE methodology in the process of structuring the system development. It closes with the details on how the final system model can be used towards the development of a physical system.

# 2 Introduction to Model Based Systems Engineering

Present day's systems are evolving in terms of complexity, interoperability, and connectivity. They are no longer standalone units but a part of well-connected ecosystem. To develop such systems, an interdisciplinary approach is essential to ensure the systems' development at a controlled cost and within a definitive schedule. Systems Engineering (SE) is one such approach that focuses on designing a system and managing its development at every point of its lifecycle. According to International Council of Systems Engineering [18] (INCOSE), SE is defined as follows:

*"Systems Engineering is an interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem."*

Model Based Systems Engineering is a methodology in SE process wherein the different views of an underlying system are represented using different model elements. Model elements are used for representing system requirements, analysis, design, implementation, and verification right from conceptual phase to later phases of product development lifecycle.

## 2.1 Advantages of Representing Systems Through Models

In comparison to traditional document based approach to SE, where the system related artifacts are maintained across various disjoint sets of text documents, spreadsheets, diagrams, presentations etc., model based approach is advantageous [19].

As system development is an iterative process, system modifications are introduced in every iteration. Therefore, there needs to be a means by which the components that are affected by newly introduced changes needs to be analyzed, thereby performing an impact analysis. With disjoint set of artifacts, this must be manually performed. Whereas, in MBSE, a dedicated tool ensures that the diagrams are integrated, coherent and consistent. Therefore, impact analysis and traceability with every introduced change, is easy to deal with using the MBSE approach. This is possible because the diagrams designed through this process present different, yet interconnected, views of the underlying system. Below is a list of advantages of using this approach, which has been discussed in [20]:

- *Automatic generation and maintenance of system documents*: The required set of documents of the system, depending on individual stakeholder's needs, can be generated automatically from the model.
- *Complexity control and management*: Complexity of models can be measured, and thus, the project or the system can be managed accordingly.
- *Consistency of information*: Due to the coherence in diagrams, if the given system is modeled as per the defined guidelines, a consistent information is maintained across the entire system model.
- *Inherent traceability*: Given that the model is well designed, a traceability between the system artifacts and different lifecycle stages is contained within the model.
- *Improved communication and uniformity in understanding*: As the system model is represented in a diagrammatic format, it provides a common platform for a team with diverse background to work together in a reliable way without any possible chances for miscommunications.

## 2.2 System Modeling Language

System Modeling Language (SysML) is a general-purpose modeling language used for SE applications. As it is a language, it has a defined grammar (syntax) and its own vocabulary (graphical notations). This language was developed from a joint effort by INCOSE and OMG to adapt an already existing Unified Modifying Language (UML), for the purposes of SE, as UML is specific to modeling software systems. SysML can be used to model varied types of systems including hardware, software, personnel, procedures and facilities. Following artifacts of a system can be represented using this language [16]:

- Requirements and their connectivity with other model elements and their verification through testcases
- Structural compositions and their interconnections
- Message-based, flow-based and state-based behavior
- System constraints

### 2.2.1 SysML – an extension of UML

UML is a general-purpose, developmental, modelling language but specific to the domain of software engineering. It has been managed by OMG since 1997. SysML, which was initially defined in 2005, was created as a sub-set of the UML along with additional diagrams to support systems engineering. The diagram below shows a pictorial representation of the relationship between UML and SysML.

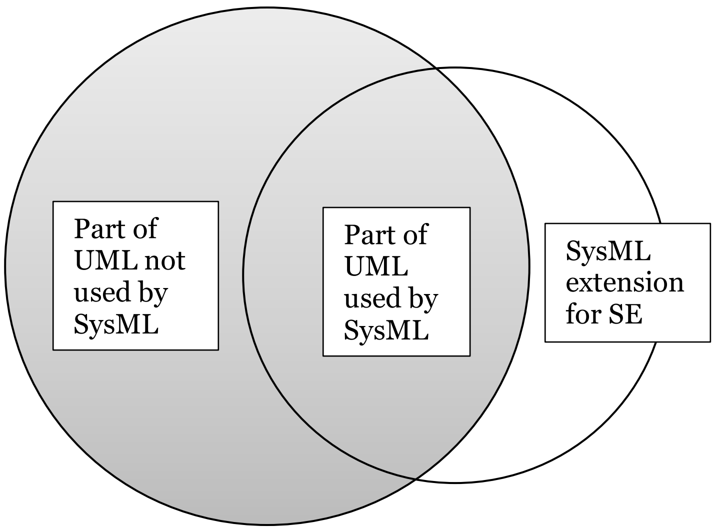


**Fig 2.1:** Relationship between SysML and UML.

The following diagrams of the UML are not used by the SysML – object diagram, communication diagram, component diagram, deployment diagram, timing diagram, interaction overview diagram and profile diagram [20]. On the other hand, the diagrams that have been added to SysML extension are parametric diagram, requirement diagram and allocation table. Additionally, the part of UML that is used by SysML is also subject to a few changes to tailor it for SE purposes. Examples for this are block definition diagram that has been modified from class diagram of UML and internal block diagram from composite structure diagram.

The figure below shows all the diagrams that are supported by the SyML and their relation to the UML has also been represented here.

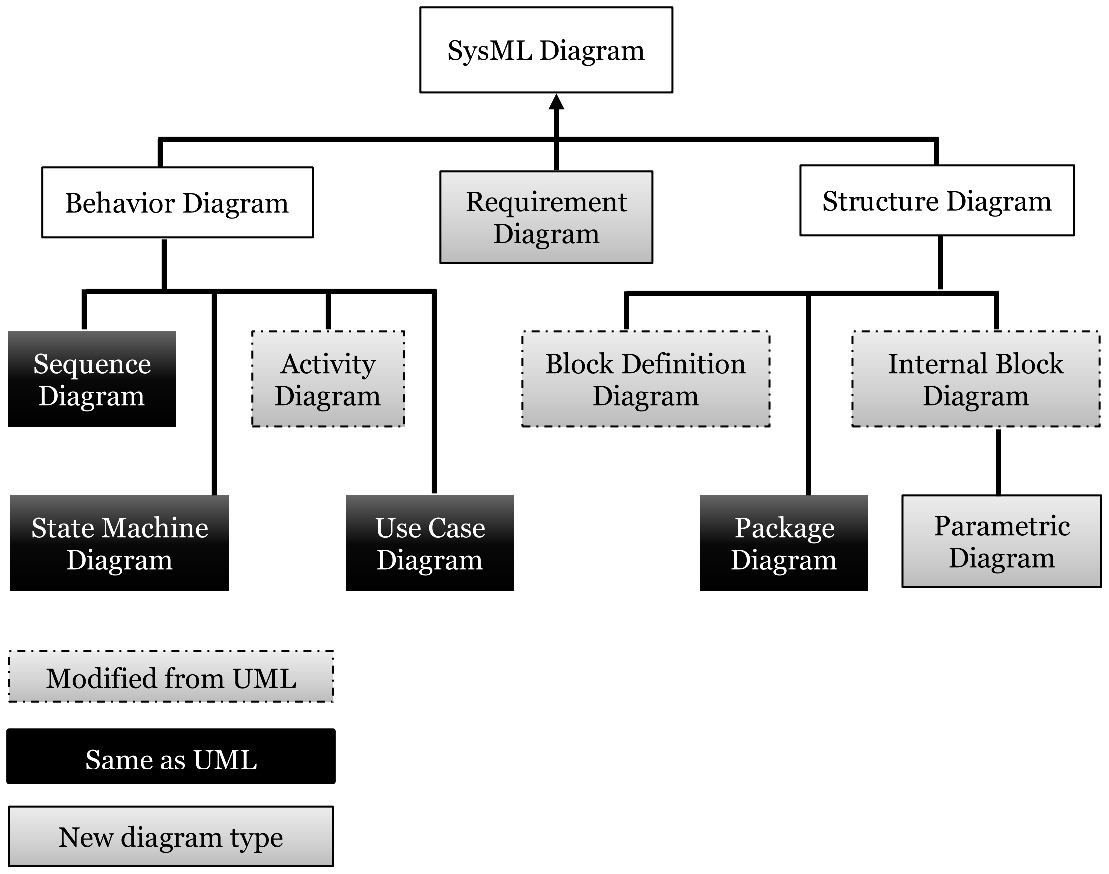


**Fig 2.2:** Types of SysML diagrams and their relation to UML diagrams.

## 2.2.2 Different types of SysML diagrams

Below is a brief description of the list of diagrams that are available in SysML:

- *Block definition diagram (bdd)*: As shown in Figure 2.2, it represents the structural composition of a system. It shows a black box view of software and/or hardware components of a system, their relationships with each other and flow specifications between these blocks. It replaces class diagram of UML.
- *Internal block diagram (ibd)*: This diagram compliments the bdd by providing a white box view of the blocks, thereby showing its internal structure. It conveys internal parts of the block and their connections with one another.
- *Use case diagram (uc)*: It represents various elements that interact with the system and shows the system's response to them. It is also used for showing services that are externally visible to interacting elements.
- *Activity diagram (act)*: It is a kind of behavior diagram which is used to represent dynamic nature of the system. Actions performed by the system,

flow of objects (energy, matter or data) and flow of control can be shown using this diagram.

- *Sequence diagram (seq)*: It is one of the three kinds of behavior diagrams. It is used for showing different kinds of interactions between system components and messages that are exchanged between them during the interactions with respect to a relative time.
- *State machine diagram (stm)*: It is used for representing various states to which the system transitions and conditions that trigger the transitions.
- *Parametric diagram (par)*: This is one of the diagrams unique to SysML which provides information about system constrains. It is also used for representing mathematical models of the system.
- *Requirement diagram (req)*: It is another diagram exclusive to SysML which establishes requirements of a system and shows various kinds of relations between the requirements, structures and behaviors to provide system traceability.
- *Package diagram (pkg)*: It is used for providing a structure to the system model. It is used for organizing the model elements into logical groups.

# 3 Defining a Model for AutoCar

To implement a model representing AutoCar's functionalities and compositions, it is important to first adopt a structure for the model that facilitates the same in a systematic manner. The structure shall approach the system from a top to down perspective, i.e. it shall ensure that the details are abstracted at higher levels. Therefore, this chapter begins with providing a background for one of the popular architectural description languages (ADLs) that is used in this project to give a basis to the model. This is then followed by a detailed explanation of the model implementation.

To model the system using SysML, an open source tool called Papyrus has been used. Papyrus is a tool that provides a platform for model based development. It uses Eclipse integrated development environment (IDE) and it is licensed under the Eclipse Public License (EPL). This tool has been chosen because of its open source nature, active community that provides good support, intuitive Graphical User Interface (GUI) and its industry level standardization.

## 3.1 Background on EAST-ADL

The architecture for AutoCar's system model has been defined using one of the available architectural description languages (ADLs) called the EAST-ADL. EAST-ADL is a type of ADL, but specific to automotive embedded systems with a built-in support for Automotive Open System Architecture (AUTOSAR). It focuses on breaking down the system model into four levels of abstraction [21]:

- **Vehicle Level**: At this level, a solution independent analysis on vehicle's contents and/or functionalities, when perceived from an external view point, are presented. The '*what*' aspect of the vehicle is shown here, and '*how*' it is achieved is not focused.
- **Analysis Level**: An abstract functional architecture of the vehicle is presented here. Entities that are defined here, must satisfy the vehicle level requirements. Vehicle's subsystems and their abstract interfaces are represented through model entities defined at analysis level.

- **Design Level**: A detailed functional architecture is derived at design level, wherein the required software components (SCs) of individual subsystems and their corresponding hardware allocation are decided.
- **Implementation Level**: System implementation's representation using AUTOSAR components is focused at this level.

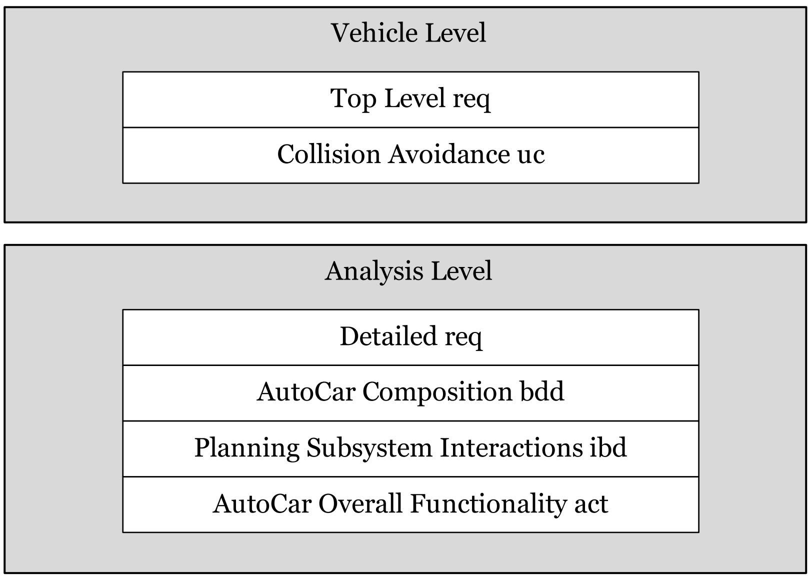
Requirement specifications, verification methods, dependencies and interactions are defined at every level of abstraction but only with the details corresponding to each level.

### 3.1.1 Purpose of EAST-ADL in project

A similar architectural breakdown and model representation will be followed for this project with the only difference at *implementation level*, as AUTOSAR is out of scope for this project. This architectural breakdown is preferred because it assists in step by step understanding, analysis and definition of the system. This top-down approach of defining the system from user specifications to a detailed implementation helps in an incremental understanding of the system at every progressive level.

As explained in chapter 2, packages are necessary to organize the model elements, thereby giving a structure to the model. Therefore, this architecture has been captured using a package structure as shown in Figure 3.1.

In this chapter, model elements representing *Vehicle Level* and *Analysis Level* are discussed. *Design Level* and *Implemetation Level* of the model will be discussed in Chapter 5, which is followed by a brief overview of the concepts used for trajectory planning presented in Chapter 4.



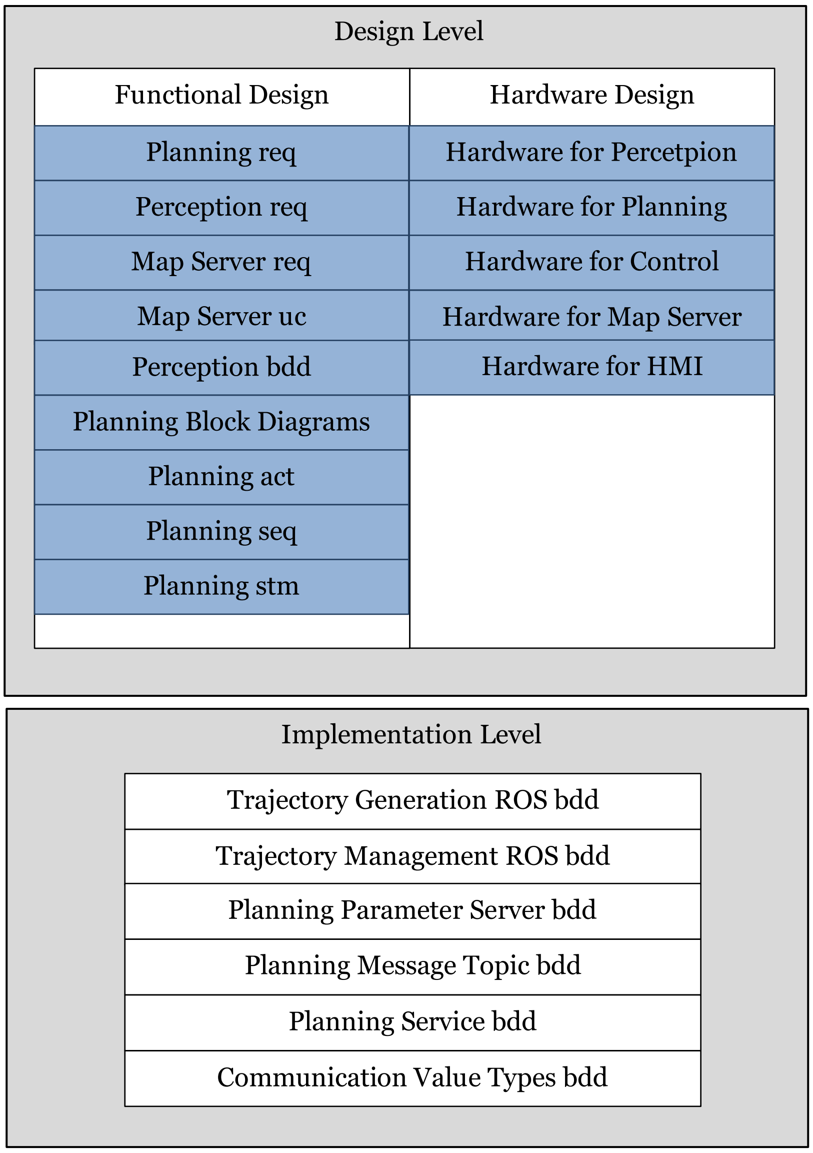


**Fig 3.1:** Architectural breakdown of AutoCar model according to EAST-ADL. The blocks that are shown in the diagram are arranged as separate and/or nested packages in the model implementation. Within these packages the respective model elements are available.

## 3.2 Diagrams Representing Vehicle Level of the System

At vehicle level, as the focus is on the vehicle functionalities as required by the user, without getting into the technical details of it, these can be sufficiently captured using *requirement diagram(s)* and *use case diagram(s)* [21].

### 3.2.1 Top Level req

This diagram captures the requirements of AutoCar when viewed from an external perspective. High level requirements, more in the form of user specifications, is shown at this level of the requirement diagram. The below shown diagram, lists the expectations for AutoCar to successfully drive autonomously to a given destination, from its start position, without colliding with encountered obstacles, in the shortest possible time. These requirements are in line with the *problem statement* specified in Chapter 1.

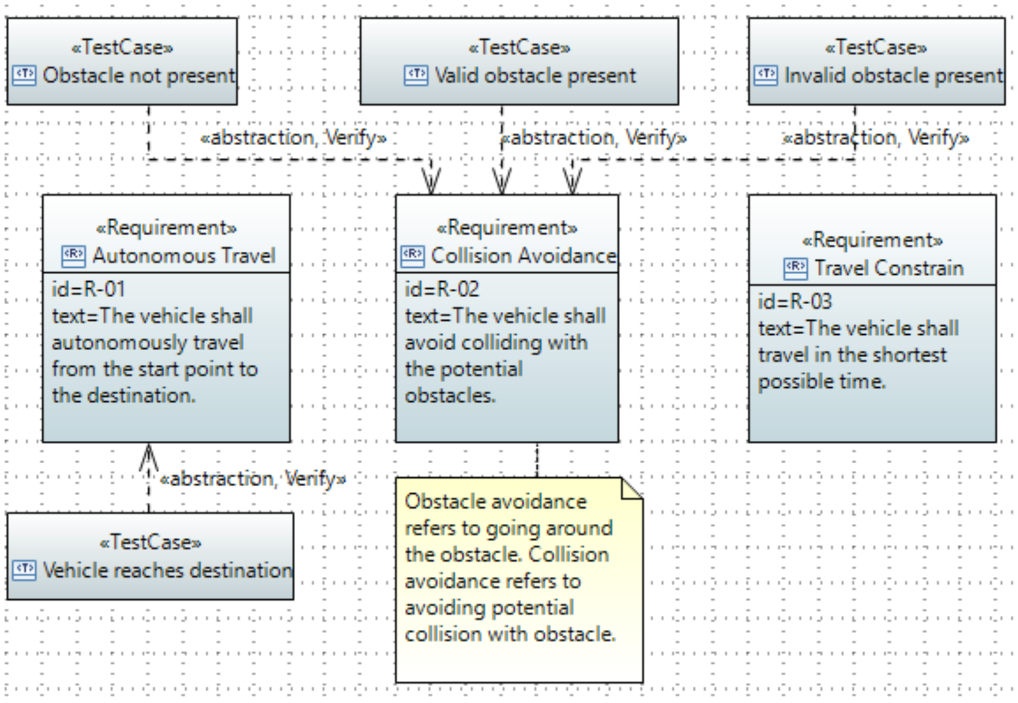

«TestCase» Obstacle not present
«TestCase» Valid obstacle present
«TestCase» Invalid obstacle present
«abstraction, Verify»
«abstraction, Verify»
«abstraction, Verify»
«Requirement» Autonomous Travel
id=R-01
text=The vehicle shall autonomously travel from the start point to the destination.
«Requirement» Collision Avoidance
id=R-02
text=The vehicle shall avoid colliding with the potential obstacles.
«Requirement» Travel Constrain
id=R-03
text=The vehicle shall travel in the shortest possible time.
«abstraction, Verify»
«TestCase» Vehicle reaches destination
Obstacle avoidance refers to going around the obstacle. Collision avoidance refers to avoiding potential collision with obstacle.

**Fig 3.2:** Requirement diagram at the *vehicle level – TopLevel_RequirementDiagram* lists overall functionalities for AutoCar.

### 3.2.2 Collision Avoidance uc

Through this use case diagram, different types of external entities (obstacles) and AutoCar's interactions with these entities are shown. Obstacle, here, is defined as any entity that is unavailable in the pre-defined digital map and is a potential candidate for collision with AutoCar. Manoeuvres corresponding to the obstacle type are shown in the diagram below. As it can be understood from the diagram, the technical details of the mentioned manoeuvres are not focused.

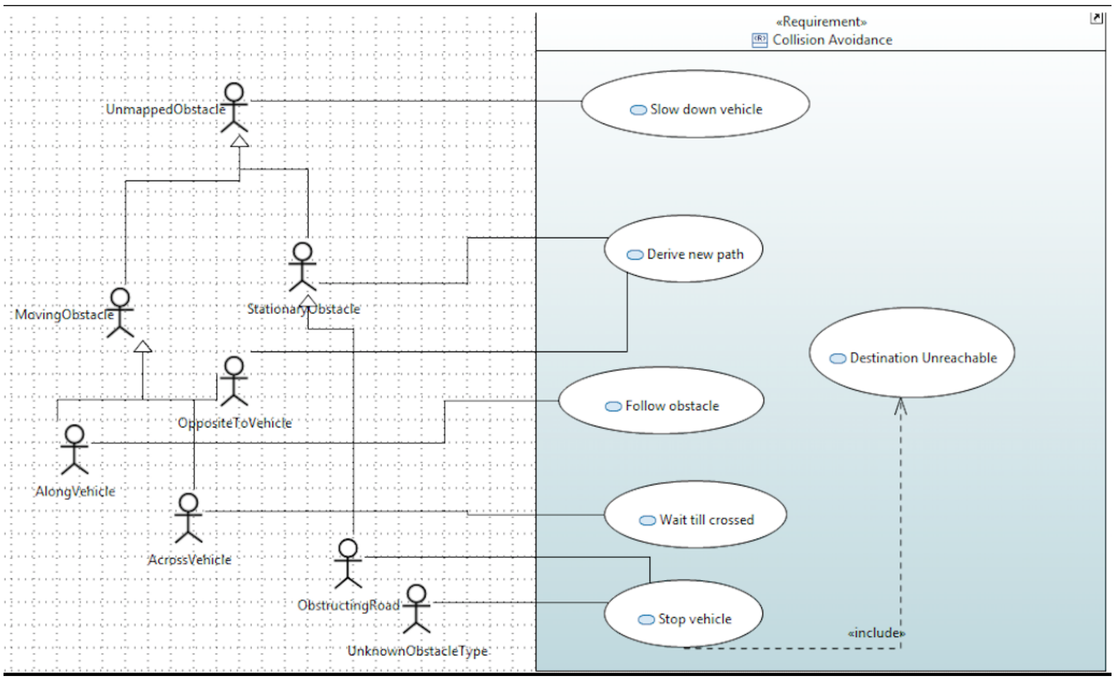


**Fig 3.3:** *ObstacleAvoidance* use case diagram shows the different manoeuvres for different types of obstacles.

In the figure above, *UnmappedObstacle* is a **generalization** of the types of obstacles that have been placed below it. Therefore, irrespective of the type of obstacle encountered by AutoCar, it is expected to slow down from its current velocity upon obstacle detection. Further, if it is *StationaryObstacle* or *OppositeToVehicle* (obstacle moving in direction opposite to the vehicle), AutoCar is expected to derive a new path for it to follow; but if it is *AlongVehicle* (obstacle moving along the direction of AutoCar), it is expected to follow the obstacle in front; and if it is *AcrossVehicle* (obstacle moving across the AutoCar), AutoCar is expected to wait till the obstacle finishes crossing.

## 3.3 Diagrams Representing Analysis Level of the System

Starting from this level, the technical details of the vehicle functionalities defined in *vehicle level* are analyzed and represented. At analysis level, subsystems that are required by the vehicle, that satisfy the requirements mentioned in the previous level, are derived. The abstract interactions between the subsystems are also specified here.

### 3.3.1 Detailed req

Requirement diagram at this level is derived from *TopLevel_RequirementDiagram*, wherein the requirement specifications of logically independent functional entities of the vehicle are derived from the user-defined functionalities of AutoCar. Individual requirement contained by each requirement specification has, in turn, been modularized.

The requirement diagram in Figure 3.4, derives the following requirement specifications:

- *Human Machine Interaction (HMI) Requirement Spec*: For the vehicle to be able to drive by itself to a user defined destination, it is required to receive the destination and a digital map as inputs, before the start of journey.
- *Map Server Requirement Spec*: To ensure that the input map is available to the vehicle in a perceivable format and the input destination is added in relation to the given map, *Map Server* is required for processing the raw input data.
- *Perception Requirement Spec*: Since it is expected by the vehicle to autonomously navigate its way to destination, it shall understand the external environment, its internal state and its location with reference to the given map.
- *Planning Requirement Spec*: To achieve autonomous driving functionality of the vehicle, it shall plan its way towards the goal by generating a virtual trajectory for the vehicle to follow. Additionally, to ensure that the obstacles are efficiently avoided from potential collisions, when encountered, the generated trajectory shall be managed according to the type of obstacle.
- *Control Requirement Spec*: It requires that the vehicle shall set the values of actuators for it to autonomously follow the derived trajectory and perform additional stabilization maneuvers to ensure that it stays on the defined track.

### 3.3.2 AutoCar Subsystem Composition bdd

The subsystems required by the vehicle and its containing modules have been shown in this diagram. These subsystems have been derived from the requirement specifications mentioned above. These subsystems are logically independent blocks which are designed for a dedicated system functionality.

This diagram gives a black-box view of the vehicle with five major subsystems: perception, planning, control, map server and HMI. The lines connecting the subsystems show the necessary interactions required between them.

### 3.3.3 Planning Subsystem Interactions ibd

A detailed view of the interactions between the subsystems, with respect to planning subsystem, is shown in the following ibd. As ibd is a complementary view of bdd, the *directed association (line with open arrow head)* relationship shown in Figure 3.5 has been elaborated in Figure 3.6 but only with reference to *planningSubsystem*.

As it can be seen, planning subsystem receives obstacle related details and odometry values (details on current vehicle state) from perception subsystem. It receives global and local map data from map server subsystem. It sends out twist message values in turn to the control subsystem to control the wheels accordingly.

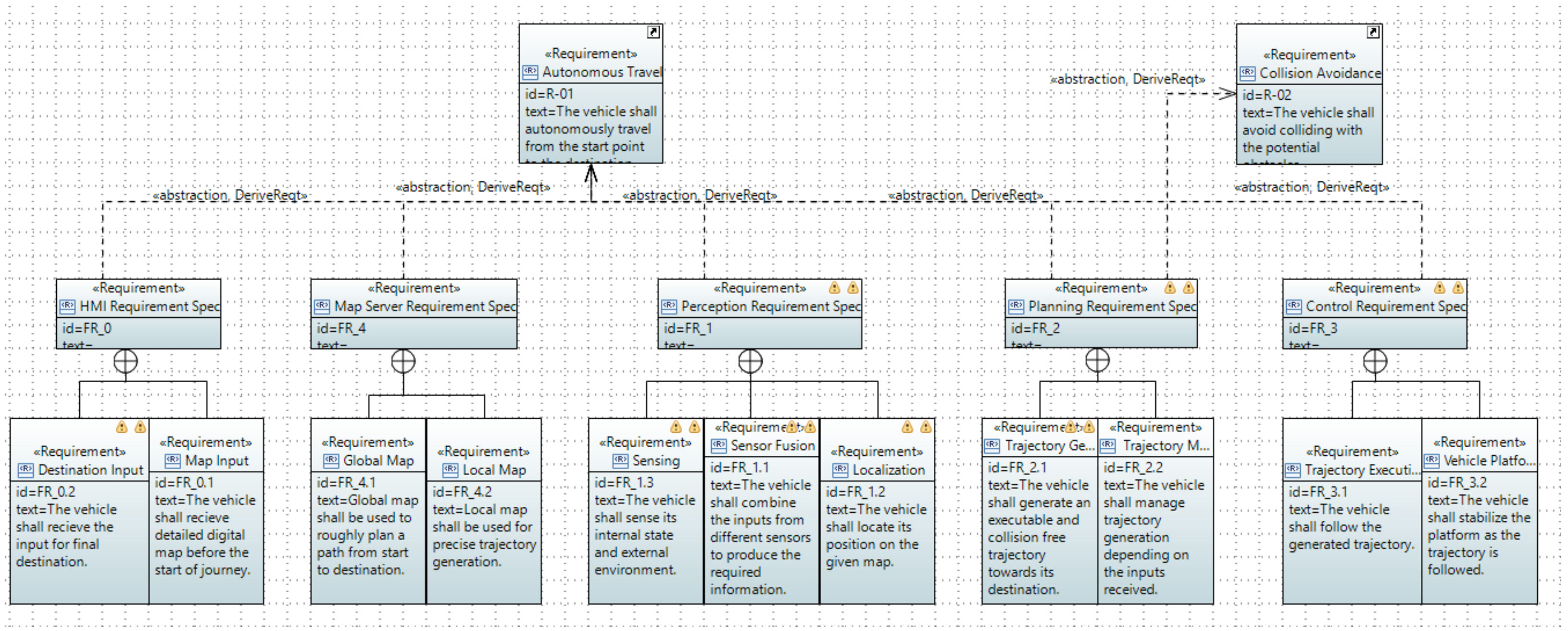

«Requirement» Autonomous Travel
id=R-01
text=The vehicle shall autonomously travel from the start point
«Requirement» Collision Avoidance
id=R-02
text=The vehicle shall avoid colliding with the potential
«abstraction, DeriveReqt»
«abstraction, DeriveReqt»
«abstraction, DeriveReqt»
«abstraction, DeriveReqt»
«abstraction, DeriveReqt»
«abstraction, DeriveReqt»
«Requirement» HMI Requirement Spec
id=FR_0
text=
«Requirement» Map Server Requirement Spec
id=FR_4
text=
«Requirement» Perception Requirement Spec
id=FR_1
text=
«Requirement» Planning Requirement Spec
id=FR_2
text=
«Requirement» Control Requirement Spec
id=FR_3
text=
«Requirement» Destination Input
id=FR_0.2
text=The vehicle shall recieve the input for final destination.
«Requirement» Map Input
id=FR_0.1
text=The vehicle shall recieve detailed digital map before the start of journey.
«Requirement» Global Map
id=FR_4.1
text=Global map shall be used to roughly plan a path from start to destination.
«Requirement» Local Map
id=FR_4.2
text=Local map shall be used for precise trajectory generation.
«Requirement» Sensing
id=FR_1.3
text=The vehicle shall sense its internal state and external environment.
«Requirement» Sensor Fusion
id=FR_1.1
text=The vehicle shall combine the inputs from different sensors to produce the required information.
«Requirement» Localization
id=FR_1.2
text=The vehicle shall locate its position on the given map.
«Requirement» Trajectory Ge...
id=FR_2.1
text=The vehicle shall generate an executable and collision free trajectory towards its destination.
«Requirement» Trajectory M...
id=FR_2.2
text=The vehicle shall manage trajectory generation depending on the inputs received.
«Requirement» Trajectory Executi...
id=FR_3.1
text=The vehicle shall follow the generated trajectory.
«Requirement» Vehicle Platfo...
id=FR_3.2
text=The vehicle shall stabilize the platform as the trajectory is followed.

**Fig 3.4:** *AutoCar requirement diagram* displaying *derive relationship* from *TopLevelRequirement* and displaying the subsystem specifications along with the modules within the subsystems.

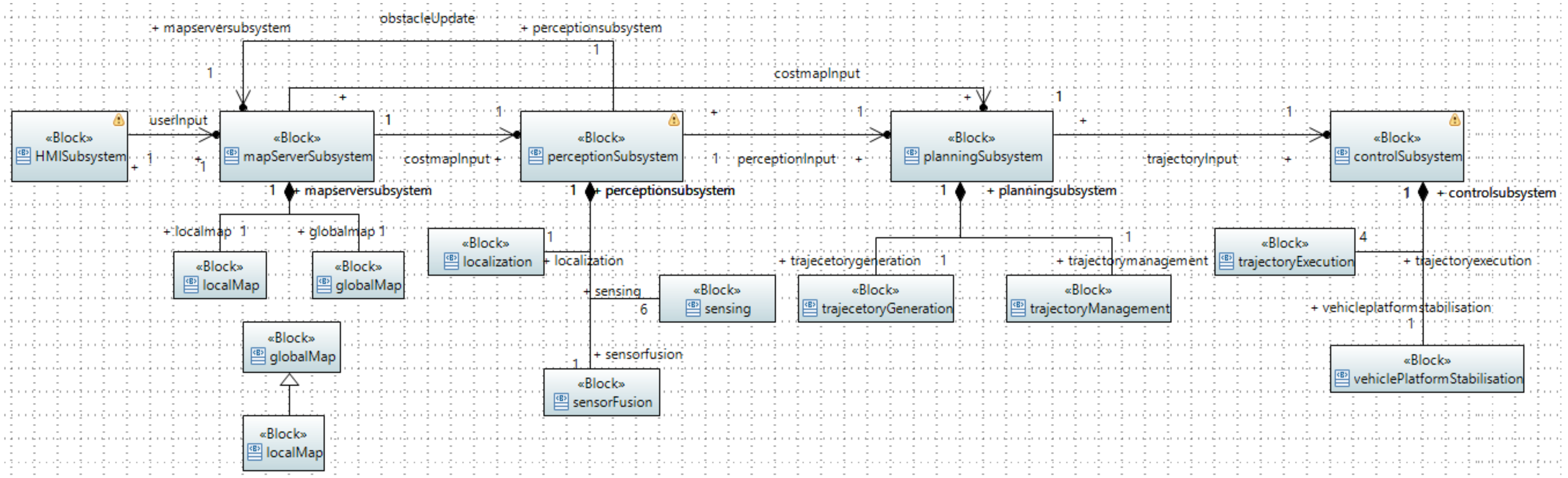


**Fig 3.5:** *AutoCarSubsystemComposition_BlockDefinitionDiagram* shows the subsystems of AutoCar and the modules contained within the respective subsystems.

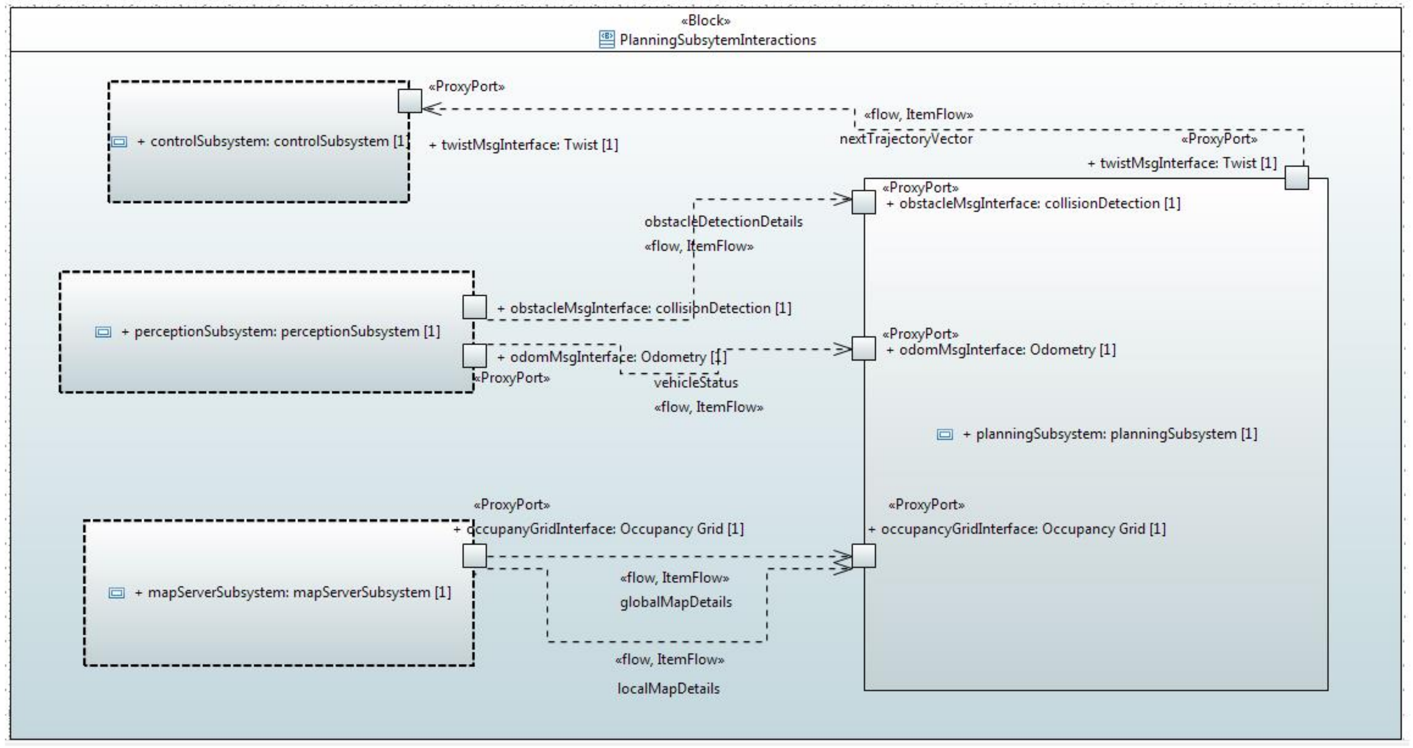

«Block»
PlanningSubsytemInteractions
«ProxyPort»
+ controlSubsystem: controlSubsystem [1]
+ twistMsgInterface: Twist [1]
«flow, ItemFlow»
nextTrajectoryVector
«ProxyPort»
+ twistMsgInterface: Twist [1]
«ProxyPort»
+ obstacleMsgInterface: collisionDetection [1]
obstacleDetectionDetails
«flow, ItemFlow»
+ perceptionSubsystem: perceptionSubsystem [1]
+ obstacleMsgInterface: collisionDetection [1]
+ odomMsgInterface: Odometry [1]
«ProxyPort»
vehicleStatus
«flow, ItemFlow»
«ProxyPort»
+ odomMsgInterface: Odometry [1]
+ planningSubsystem: planningSubsystem [1]
«ProxyPort»
+ occupanyGridInterface: Occupancy Grid [1]
+ mapServerSubsystem: mapServerSubsystem [1]
«flow, ItemFlow»
globalMapDetails
«flow, ItemFlow»
localMapDetails
«ProxyPort»
+ occupancyGridInterface: Occupancy Grid [1]

**Fig. 3.6:** Internal block diagram showing the communication of *planningSubsystem* with other subsystems

### 3.3.4 AutoCar Overall Functionality act

The flow of activities for the vehicle to be able to drive autonomously to the destination, avoiding all possible collisions is shown in the below figure. As it can be observed, the following actions run parallelly: *Perceive Environment and AutoCar, Generate trajectory values for current environment, Control actuators* and *Generate local maps*. Simple open headed arrows represent the flow of control from one action to another. Whereas, small squares containing arrows and small squares connected by open headed arrows (which are semantically the same) show flow of object which can be matter, energy or data. For actions that contain both incoming object and control flows, the action gets executed when both the control and object are available.

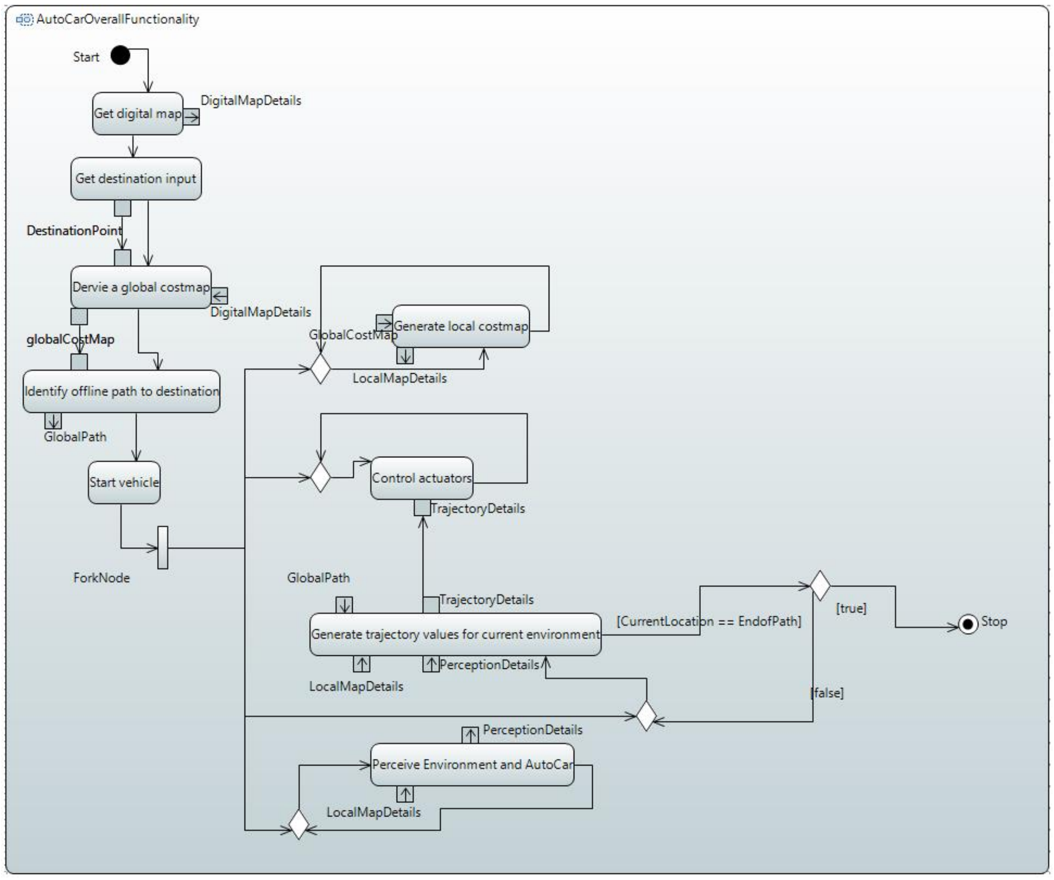


**Fig 3.7:** *AutoCarOverallVehicleFunctionality_ActivityDiagram* shows the flow of actions that enable the vehicle to travel from the start to destination autonomously by avoiding any possible collisions.

Thus, an abstract system structure of the autonomous vehicle (AutoCar) has been established through these diagrams, thereby identifying the required sub-systems to perform the desired functionalities.

*Note: Design level* and *Implementation level* of the system model are shown in Chapter 5 after providing a basic overview of *planning subsystem* and its approach for trajectory generation.

# 4 Planning Subsystem of AutoCar

As mentioned in *Introduction* section of Chapter 1, the planning subsystem of the vehicle is responsible for assigning an executable trajectory for the vehicle to follow. This subsystem receives inputs from the perception subsystem to be able to understand the vehicle's environment. From these inputs, it accordingly plans a path towards the goal and generates velocity values to enable the vehicle to follow the path. The generated trajectory values are further given as input to the control subsystem which executes the trajectory by quantitatively setting and adjusting the values of the actuators. Therefore, the focus for this subsystem is on the following functionalities:

- Path planning
- Velocity generation
- Collision avoidance
- Costmap maintenance

Trajectory is defined as a path followed by a moving object as a function of time. It is mathematically represented, for an object moving in a surface on the $x$-$y$ plane, as:

$$Position\ p(t) = (x(t), y(t))\ where\ t\ is\ time\ in\ seconds$$

As mentioned in [25], since AutoCar is a 4-wheeldrive (4WD) system (i.e. non-holonomic), its motion is constrained to $x$ and $y$ axes of a two-dimensional plane and its heading direction is defined by orientation or rotation around $z$ axis i.e. $\theta_z$. The $x$ and $y$ position and the corresponding orientation at every value of $t$ defines the trajectory of AutoCar, i.e $x$, $y$, $\theta$ values at specified time instances define the trajectory of the vehicle.

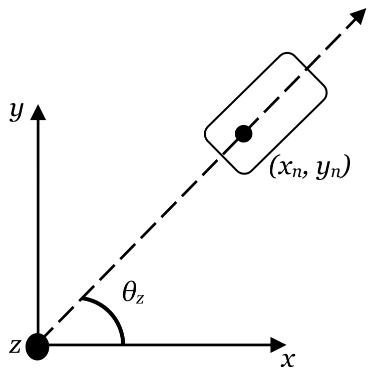


**Fig 4.1:** This diagram shows the vehicle position and orientation in $x$-$y$ plane.

The motivation behind breaking down a trajectory planning problem into path planning problem and velocity planning problem, as discussed in [4], is to define the trajectory of the vehicle as a discrete time entity. This is achieved as follows – planned path is broken into waypoints and velocity values between the waypoints are generated to enable the vehicle to follow the given trajectory.

$$\text{where, } v = \sqrt{\left(\frac{\Delta x}{\Delta t}\right)^2 + \left(\frac{\Delta y}{\Delta t}\right)^2}$$

$$\omega = \frac{\Delta\theta}{\Delta t}$$

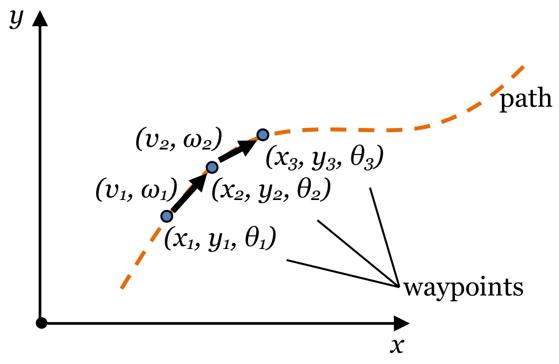


**Fig 4.2:** Breaking down trajectory planning into path planning and velocity planning.

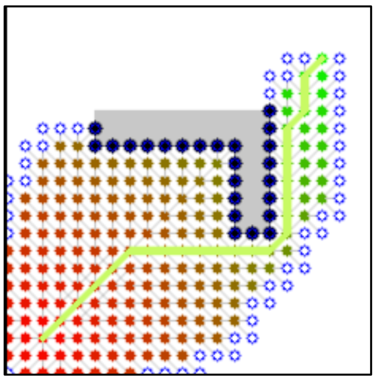
As it can be seen in Figure 4.2, velocity values for the vehicle are defined by translational velocity $v$ and angular velocity $\omega$. As the vehicle is non-holonomic in nature, it possesses only translational velocity, which is tangential to the path, and angular velocity $\omega$, which is the rate of change of vehicle's orientation [25].

As defined in the *Problem Description* section, obstacles moving at a constant velocity will also be handled by the system by avoiding potential collisions with them. To avoid collision, contingency planning is needed to ensure that new path and/or new velocity value is/are derived depending on the type of the obstacle. This contingency planning is handled by obstacle avoidance functionality of the subsystem.

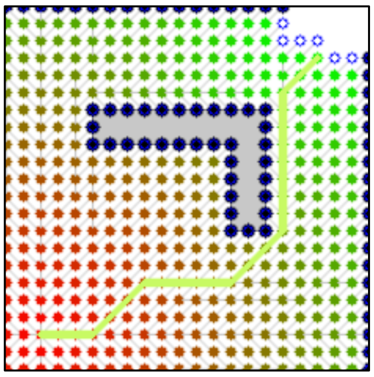
## 4.1 Path Planning

Path planning involves finding a collision free and optimal route between two points. Optimal route can mean either being the shortest path, or having less number of turnings, or requiring fewer variations in vehicle acceleration, or a specific combination of these parameters. These parameters are used in a function called the cost function to achieve the desired path. The details of which will be discussed in Section 4.4.

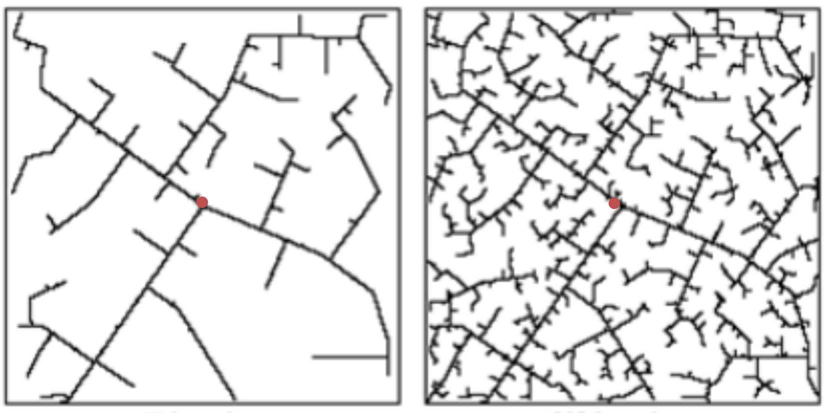
To perform path planning, a complete map of the environment (global map), vehicle's start position and its end position are required. The given map is dominantly represented in a discrete format [23] where the continuous map is divided into grids of equal or unequal sizes or into graphs. Path planning algorithms like A* (A star), Dijsktra, Rapidly-exploring Random Trees (RRT) or D* (D star) are applied onto the discretized maps to find a feasible line that connects the start position to the end position.



(a)



(b)



(c)

**Fig 4.3:** (a) Shows path search using A* algorithm (b) Shows path search using Dijkstra's algorithm (c) Shows path search using RRT. In (a) and (b) the green line shows the finally planned path, the solid grey block is the obstacle and the dots represent the search tree. In (c) the red dot represents the point of origin for search tree that grows with iterations.

## 4.2 Velocity generation using Dynamic Window Approach

After a feasible path is planned, it is important to generate velocity values which are required to follow the planned path. To generate feasible velocities ($v$, $\omega$) for the vehicle to move towards the goal, a moving window frame (local map) of fixed dimensions (height and width) is used and this approach is called Dynamic Window Approach (DWA) [24]. At any specified time instance, the window is centered on the vehicle. Within the frame, the next waypoint(s) of the planned path, acceleration limits (both linear and angular accelerations) of the vehicle and current vehicle details (position, orientation and velocity) are needed to be able to generate feasible velocity values for the next waypoint(s). The most suitable velocity value, that enables the vehicle to move towards its goal, avoids collision with possible obstacles, and reduces the travel time, is chosen from them. In DWA vehicle velocity values are generated for a short time interval. Here the vehicle trajectory is generated in accordance to the constrains on vehicle dynamics.

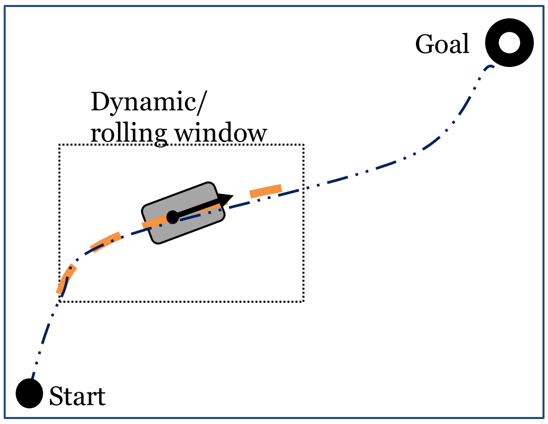


**Fig. 4.4:** A representation of DWA on a global map.

## 4.3 Collision avoidance

The planning subsystem reacts to unmapped obstacles that are encountered during the course of the vehicle and avoids any potential collision with these obstacles. This obstacle detection and avoidance is achieved through DWA, which generates vehicle velocities depending on the local map. This map gets updated with the perceived obstacles and their potential region for collision with AutoCar by using a state-time analysis [13]. Depending on the type of obstacle, as defined in Figure 3.3, this subsystem generates a suitable trajectory.

### 4.3.1 Find a new path around obstacle

For obstacles that are stationary or that move in a direction opposite to AutoCar, a new path is planned in such a way that it is closer to the globally planned path and avoids the collision region. Velocity values are generated for the new path using DWA in adherence to the dynamics of the vehicle.

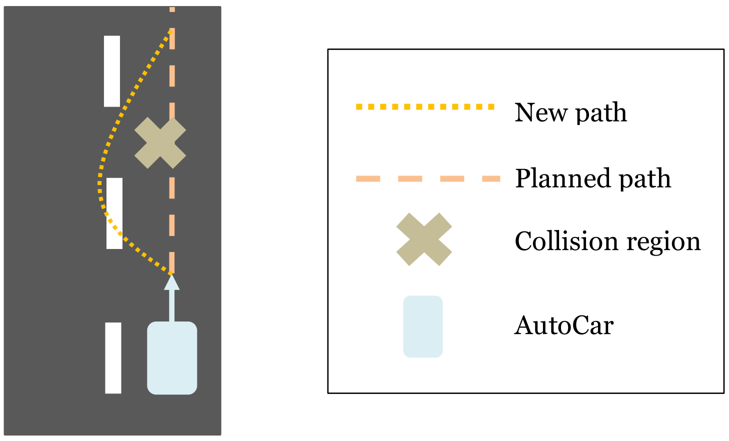


**Fig. 4.5:** Avoiding possible collision with obstacle by re-planning a new path closer to the offline/ global path. The obstacle can either be stationary or moving in a direction opposite to the vehicle. Irrespective of these two types, the collision region is identified.

### 4.3.2 Follow obstacle

For obstacles that move along AuotCar's direction, the global path is followed but the velocity values are generated in such a way that AutoCar follows the obstacle in front by maintaining a safe distance with respect to that obstacle.

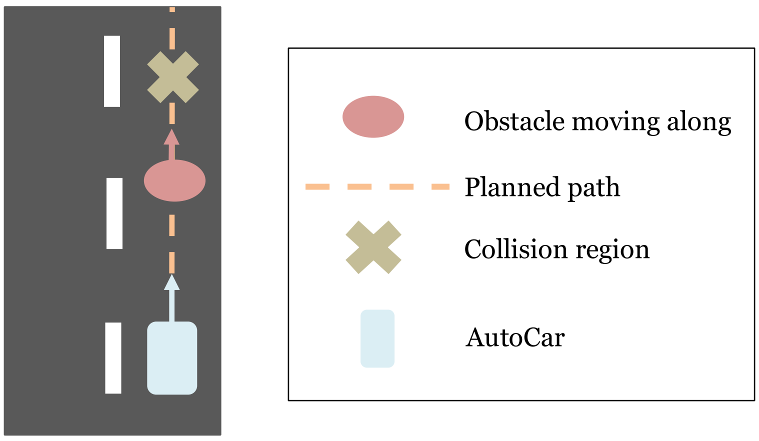


**Fig. 4.6:** Possible collision with obstacle moving along the direction of AutoCar.

Collision of AutoCar with such obstacles is possible only under the following condition:

$$v_{obs} < v_{AutoCar}$$

Where, $v_{obs}$ is velocity of obstacle and $v_{AutoCar}$ is velocity of AutoCar.

### 4.3.3 Wait for obstacle to cross

For obstacles that cross the vehicle, the global path remains the same but the velocity values are generated in such a way that AutoCar slows down to a halt, at a safe distance with reference to the possible collision region with the crossing obstacle, till it completes crossing.

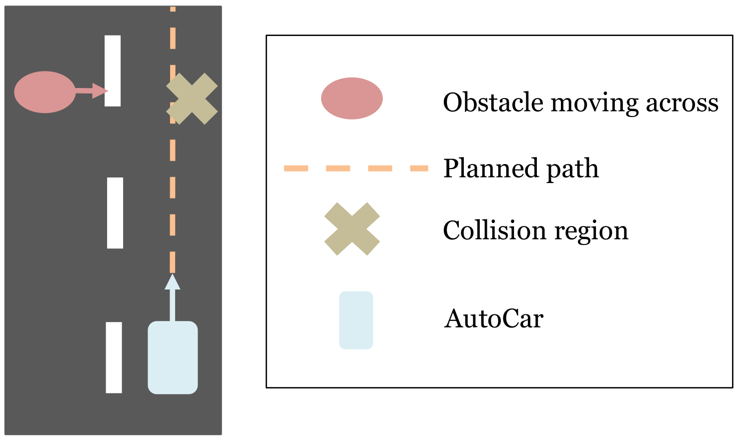


**Fig. 4.7:** Possible collision with obstacle moving across AutoCar.

## 4.4 Costmap maintenance

The map is converted into an occupancy grid of a specified resolution and each cell in the grid is assigned with a cost value from 0 to 254, where 254 is assigned to cells with definite obstacle and 0 is assigned to cells with definite free space and the values in between are assigned for varying probabilities of obstacle [26] and for penalized paths, the latter is required to ensure that the vehicle drives on the right side of the lane. The cost value of individual cell will be considered as one of the factors for calculating the total cost of the path. Path with the least cost is chosen from the set of feasible paths [rolling window paper].

Cost $C$ of path $p$ is given as follows:

$$C(p) = \alpha.GCcost + \beta.Gdist$$

Here, $\alpha$ and $\beta$ are weights assigned to the individual parameters. $GCcost$ is the sum of values assigned to each grid cell, which ensures that the desired path is free of obstacles and AutoCar stays to the right side of the lane. $Gdist$ is the distance to the goal from the current location.

As AutoCar is approximated to a point object, as mentioned in Chapter 1, planned paths can be very close to obstacles, thereby causing the vehicle to collide with them. Therefore, to ensure generation of feasible, collision free paths, the obstacles are to be inflated proportional to the size of the vehicle. Figure 4.8 shows the reason for inflating obstacles through a diagrammatic representation. Figure 4.9 shows the logic that is used for inflation which is explained in [26].

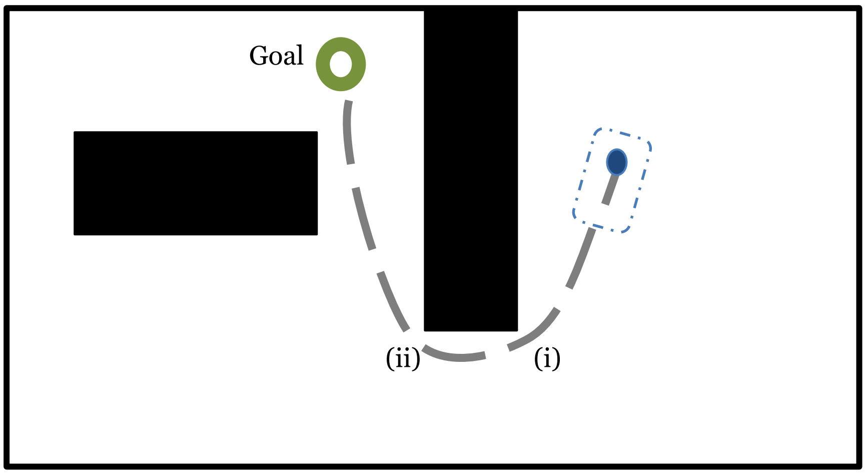


**Fig. 4.8:** Due to point approximation of the vehicle, the path is generated with respect to the point object in the map. But when the complete AutoCar is considered, it is likely to collide at points (i) and (ii). This requires that a tolerance equivalent to the size of the vehicle is added to the obstacles (black rectangle blocks).

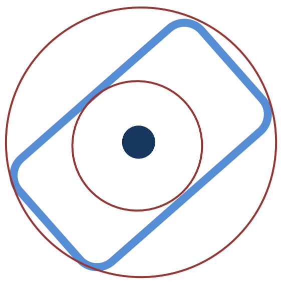


**Fig. 4.9:** Inflation cost that needs to be added to the obstacles is decided using a circle that approximately inscribes within the vehicle (blue rectangle) and a circle that approximately circumscribes the vehicle. The dot in the center is the point approximation of the vehicle.

Inflation is a process of propagating the assigned costs to cells with definite obstacle in an outward direction in such a way that the cost value gradually decreases with distance. It is performed in 4 distinct stages:

1. Cells that have obstacles are assigned with a value of 255.
2. Cells that are at a distance equal to the inscribed radius of the vehicle from the obstacle are assigned with a cost value of 254.

3. Cells that are beyond the inscribed radius but within the circumscribed radius of the vehicle are assigned with a range of values between 253 and 128 depending on the orientation of the vehicle as it possible for the vehicle to collide with the obstacle only at specific orientations.
4. Beyond the distance equal to circumscribed radius, the costs are added to the cells as a decaying function with values in between 127 and 0.

# 5 Representation of Trajectory Planning in the System Model

With the basis provided by *Analysis Level* in Chapter 3 and the understanding of basic concepts and ideas behind autonomous trajectory planning, that is provided in Chapter 4, the model is expanded with a detailed design at *Design Level* and its realization on AutoCar is represented in *Implementation Level*. The subsystems and their corresponding software components (SCs) that are defined in *Analysis Level*, are elaborated in this chapter with a focus on the *planning subsystem*.

Firstly, requirements are derived and elaborated in *Design Level* from the existing requirements laid-out in the previous level i.e. *Analysis Level*. At *Design Level*, the system is analyzed and hence, defined a step ahead with respect to the system's structural and behavioral details. Planning subsystem's reactions to various external entities are elaborated here. Its interactions with other subsystems, required inputs and outputs, are also defined.

To design *Implementation Level* of the model, the components of an open source middleware, called Robot Operating System (ROS) are used with the necessary modifications to accommodate the so far designed model. Section 5.2.1 gives a background of the ROS middleware and it is followed by a brief explanation of the necessary ROS terms.

## 5.1 Diagrams Representing Design Level of Planning Subsystem

### 5.1.1 Functional design architecture of the sub-system

Here, a detailed design of the system's functionalities required for the planning subsystem to generate executable trajectories is presented. The subsystems defined in Chapter 3 are elaborated here with a focus on trajectory planning.

#### *5.1.1.1 Planning req*

In this diagram, the requirements are laid out for *Trajectory Management* and *Trajectory Generation* components for *Planning Subsystem*. Trajectory Generation is responsible for planning a path before start of the journey and later, calculating the required velocity values to follow the path as closely as possible. Whereas, *Trajectory Management* manages *Trajectory Generation* by deciding the mode in which the later shall function. It acts as an abstraction layer for Trajectory Generation.

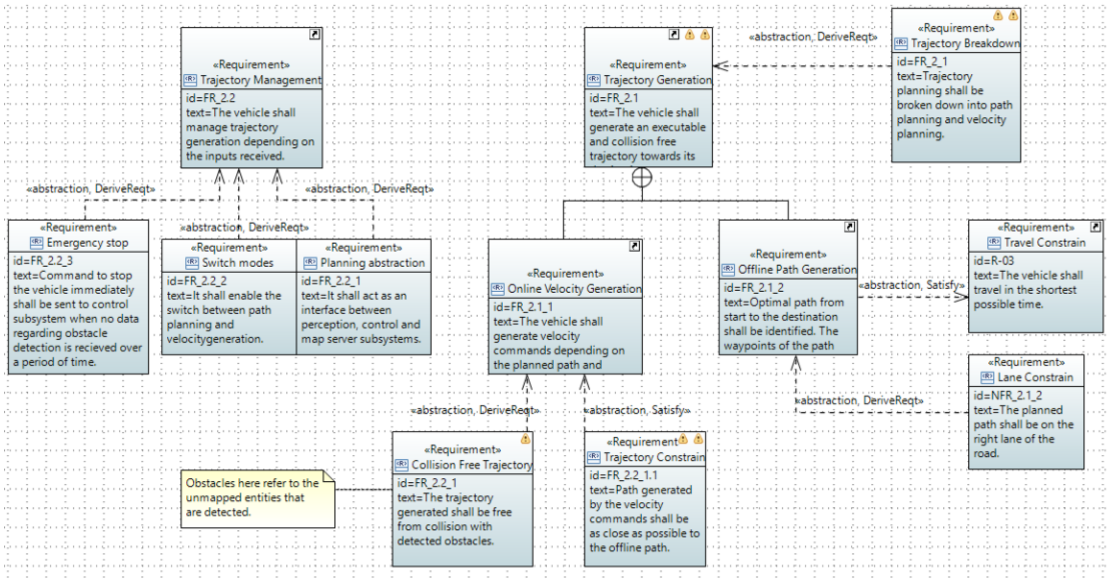

«Requirement»
Trajectory Management
id=FR_2.2
text=The vehicle shall manage trajectory generation depending on the inputs received.
«Requirement»
Trajectory Generation
id=FR_2.1
text=The vehicle shall generate an executable and collision free trajectory towards its
«abstraction, DeriveReqt»
«Requirement»
Trajectory Breakdown
id=FR_2_1
text=Trajectory planning shall be broken down into path planning and velocity planning.
«abstraction, DeriveReqt»
«abstraction, DeriveReqt»
«Requirement»
Emergency stop
id=FR_2.2_3
text=Command to stop the vehicle immediately shall be sent to control subsystem when no data regarding obstacle detection is recieved over a period of time.
«abstraction, DeriveReqt»
«Requirement»
Switch modes
id=FR_2.2_2
text=It shall enable the switch between path planning and velocitygeneration.
«Requirement»
Planning abstraction
id=FR_2.2_1
text=It shall act as an interface between perception, control and map server subsystems.
«Requirement»
Online Velocity Generation
id=FR_2.1_1
text=The vehicle shall generate velocity commands depending on the planned path and
«Requirement»
Offline Path Generation
id=FR_2.1_2
text=Optimal path from start to the destination shall be identified. The waypoints of the path
«abstraction, Satisfy»
«Requirement»
Travel Constrain
id=R-03
text=The vehicle shall travel in the shortest possible time.
«Requirement»
Lane Constrain
id=NFR_2.1_2
text=The planned path shall be on the right lane of the road.
«abstraction, DeriveReqt»
«abstraction, DeriveReqt»
«abstraction, Satisfy»
«Requirement»
Collision Free Trajectory
id=FR_2.2_1
text=The trajectory generated shall be free from collision with detected obstacles.
«Requirement»
Trajectory Constrain
id=FR_2.2_1.1
text=Path generated by the velocity commands shall be as close as possible to the offline path.
Obstacles here refer to the unmapped entities that are detected.

**Fig. 5.1:** Requirement diagram of *Planning Subsystem* defines the requirements of the two components – Trajectory Management and Trajectory Generation that are defined as a part of this subsystem.

#### *5.1.1.2 Perception req*

Here, the requirements for *Sensing* and *Sensor Fusion* components of *Perception Subsystem* are expanded. Figure 5.2(a) shows the requirement for *Sensing* which collects information from the different vehicle sensors as input. *Sensor Fusion* collects and combines the data from *Sensing*, interprets and converts them into a format that can be perceived and used by other subsystem(s). Figure 5.2(b) shows requirements for *Sensor Fusion* for the purpose of trajectory planning – this component shall combine the internal and external sensory details to differentiate the external objects from obstacles and hence identify possible region for collision with detected obstacle, preferably by using state time analysis matrix [13].

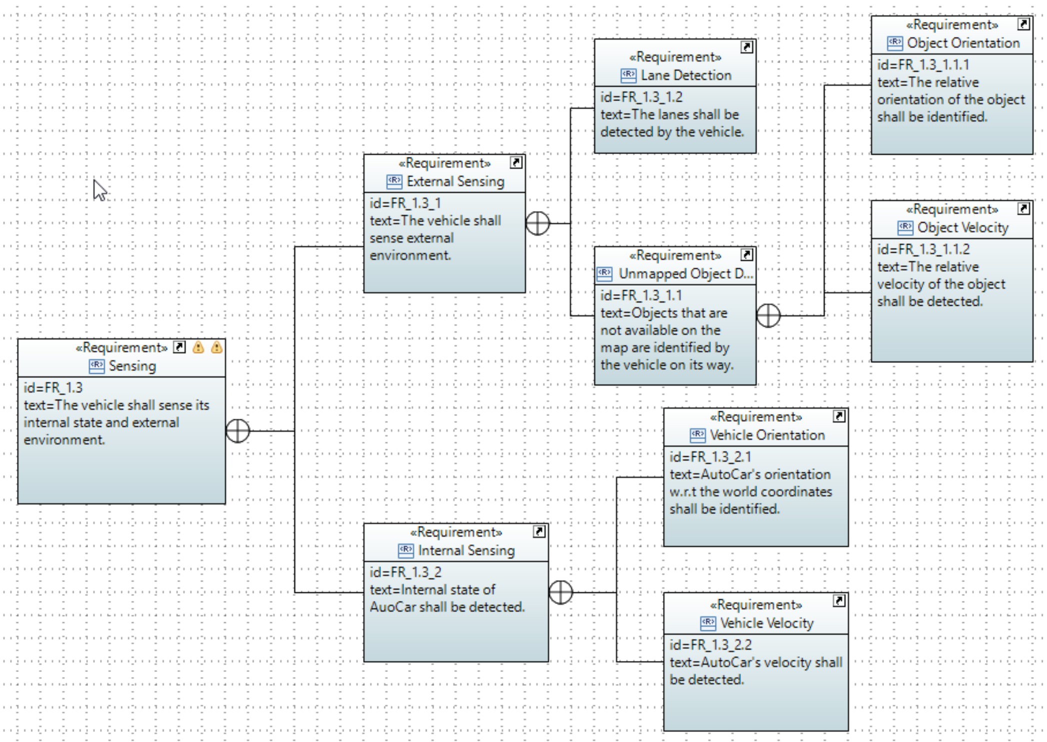

«Requirement»
Sensing
id=FR_1.3
text=The vehicle shall sense its internal state and external environment.
«Requirement»
External Sensing
id=FR_1.3_1
text=The vehicle shall sense external environment.
«Requirement»
Internal Sensing
id=FR_1.3_2
text=Internal state of AuoCar shall be detected.
«Requirement»
Lane Detection
id=FR_1.3_1.2
text=The lanes shall be detected by the vehicle.
«Requirement»
Unmapped Object D...
id=FR_1.3_1.1
text=Objects that are not available on the map are identified by the vehicle on its way.
«Requirement»
Object Orientation
id=FR_1.3_1.1.1
text=The relative orientation of the object shall be identified.
«Requirement»
Object Velocity
id=FR_1.3_1.1.2
text=The relative velocity of the object shall be detected.
«Requirement»
Vehicle Orientation
id=FR_1.3_2.1
text=AutoCar's orientation w.r.t the world coordinates shall be identified.
«Requirement»
Vehicle Velocity
id=FR_1.3_2.2
text=AutoCar's velocity shall be detected.

**Fig. 5.2 (a):** Requirement diagram of *Sensing* component of *Perception Subsystem* shows its breakdown into different sensor information that are needed by the vehicle.

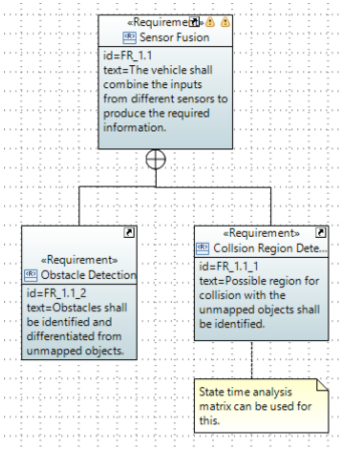

«Requirement»
Sensor Fusion
id=FR_1.1
text=The vehicle shall combine the inputs from different sensors to produce the required information.
«Requirement»
Obstacle Detection
id=FR_1.1_2
text=Obstacles shall be identified and differentiated from unmapped objects.
«Requirement»
Collsion Region Dete...
id=FR_1.1_1
text=Possible region for collision with the unmapped objects shall be identified.
State time analysis matrix can be used for this.

**Fig. 5.2 (b):** This diagram shows the requirement of *Sensor Fusion* component of *Perception Subsystem* to identify obstacles and region for collision with the same.

#### *5.1.1.3 Map Server req*

In the below requirement diagram, requirements of *Local Map* and *Global Map* are listed out. *Global Map* is the complete map that is given as an input to AutoCar before start of the vehicle whereas *Local Map* is a snippet from *Global Map* that is generated around the vehicle with respect to its current position. In *Global Map* the barriers shall be inflated as mentioned in Section

4.4 and costs shall be added to left lane to enable AutoCar to drive on right lane. Detected obstacles shall be updated on *Local Map*.

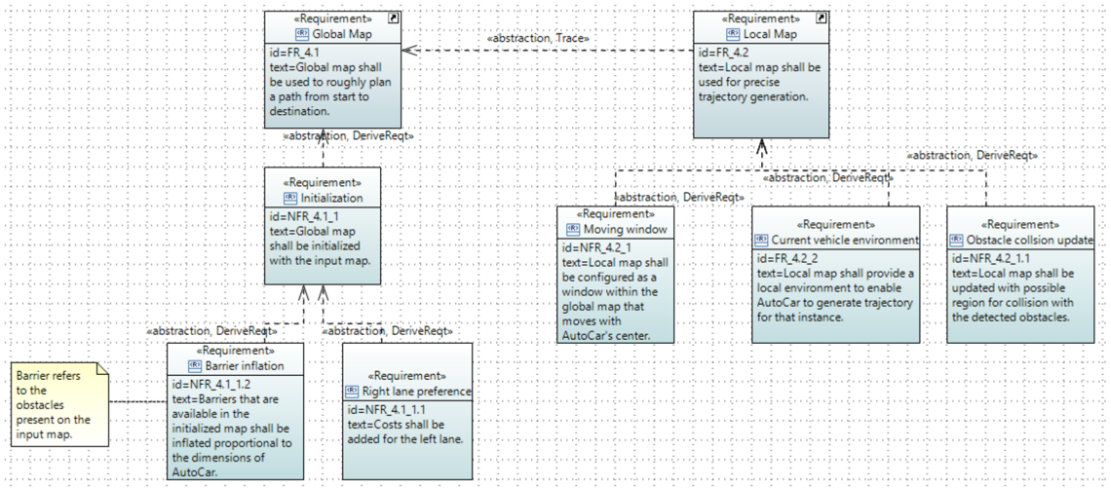

«Requirement» Global Map
id=FR_4.1
text=Global map shall be used to roughly plan a path from start to destination.
«abstraction, Trace»
«Requirement» Local Map
id=FR_4.2
text=Local map shall be used for precise trajectory generation.
«abstraction, DeriveReqt»
«Requirement» Initialization
id=NFR_4.1_1
text=Global map shall be initialized with the input map.
«abstraction, DeriveReqt»
«Requirement» Moving window
id=NFR_4.2_1
text=Local map shall be configured as a window within the global map that moves with AutoCar's center.
«Requirement» Current vehicle environment
id=FR_4.2_2
text=Local map shall provide a local environment to enable AutoCar to generate trajectory for that instance.
«Requirement» Obstacle collsion update
id=NFR_4.2_1.1
text=Local map shall be updated with possible region for collision with the detected obstacles.
«abstraction, DeriveReqt»
«abstraction, DeriveReqt»
«Requirement» Barrier inflation
id=NFR_4.1_1.2
text=Barriers that are available in the initialized map shall be inflated proportional to the dimensions of AutoCar.
«Requirement» Right lane preference
id=NFR_4.1_1.1
text=Costs shall be added for the left lane.
Barrier refers to the obstacles present on the input map.

**Fig. 5.3:** Requirements of *Local Map* and *Global Map* of *Map Server* subsystem are shown in this requirement diagram.

### 5.1.1.4 Map Server uc

Through the below represented use case diagram for Map Server subsystem, the requirements shown in Figure 5.3 are displayed through a different perspective. Here, the external entities/actors and their interactions with *Global Map* and *Local Map* are shown.

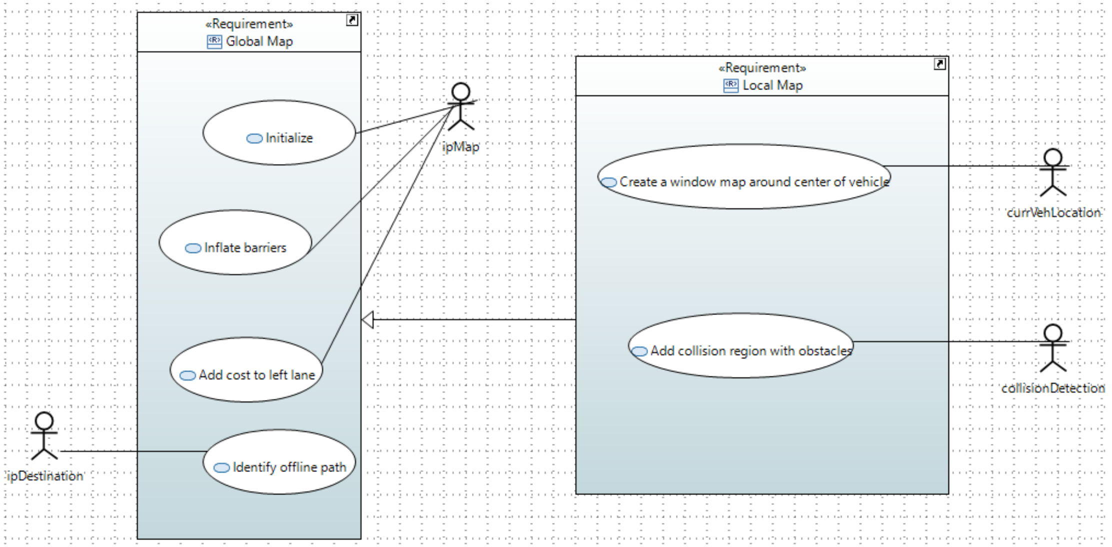


**Fig. 5.4:** Interactions between *ipMap* and *ipDestination* with *GlobalMap*, ObstacleDetection and currentVehLocation with LocalMap are shown through this use case diagram.

#### 5.1.1.5 Perception bdd

The requirements that have been identified for *Perception Subsystem* in Section 5.1.1.2 are represented as bdd with the required module breakdown for *sensing* and *sensorFusion* components. *collisionDetection* component, under *sensorFusion*, lists the attributes of detected collision in terms of types of values that are evaluated as a part of it. These values are required by *Planning Subsystem* as an input to avoid possible collisions. Enumeration of *obstacleType* shown in this bdd matches the ones listed in Section 3.2.2.

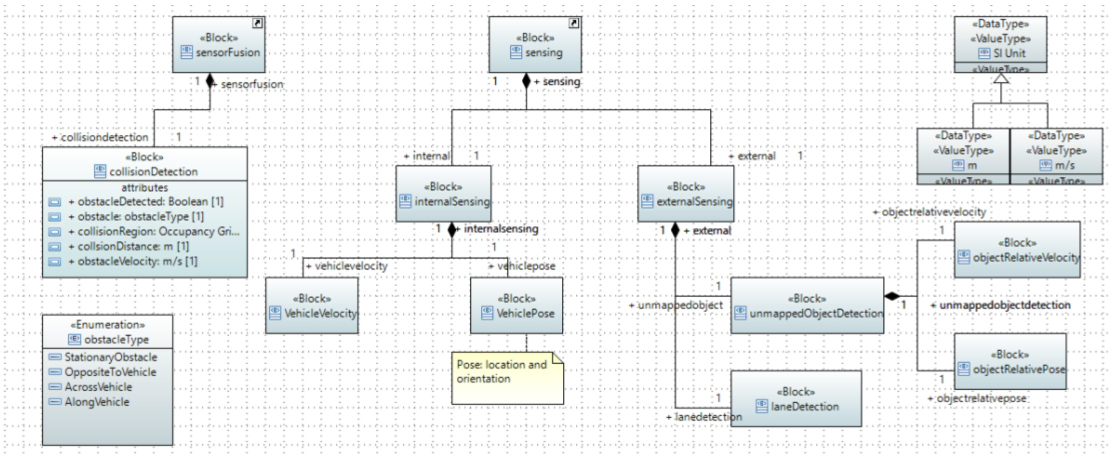


**Fig. 5.5:** Blocks showing the composition of *sensing* and *sensorFusion* modules. It also shows the necessary values that are to be computed as a part of *collisionDetection* which are needed to generate collision free trajectory.

#### 5.1.1.6 Planning Block Diagrams

Figure 5.6 (a) shows the bdd of planning subsystem whereas, Figure 5.6 (b) shows the ibd of the same subsystem. Through the bdd, further breakdown of *trajectoryGeneration* into *offlinePathPlanning* and *onlineVelocityGeneration* is shown. *planPath* is a request sent by *trajectroyManagement* but the other two are signals to which this module reacts, which are shown in Figure 5.6 (a). The enumeration values *Velocity Generation Type* are mapped to *Obstacle Types* as shown in Table 5.1. The former values decide vehicle velocity that needs to be generated as an input to *Control Subsystem.*

The association that is shown in Figure 5.6 (a) is expanded in ibd shown in Figure 5.6 (b). The latter diagram shows details on the flow of data within *Planning Subsystem*, i.e. between *trajectoryManagement* and *trajectoryGeneration* modules of the subsystem, and outside *Planning Subsystem.* It can be understood from the diagram that *trajectoryManagement* module serves as an abstraction for *trajectoryGeneration*. It abstracts the details on the type of obstacles received and manages the decision to switch between planning a path and generating velocity values, which are mutually exclusive functionalities of trajectory generation.

| *collisionDetection* inputs read by *Trajectory Management* | | | | *Planning Subsystem* |
|---|---|---|---|---|
| Obstacle Detected | Obstacle Type | Collision Distance | Obstacle Velocity | *trajectoryManagement* Velocity Generation Type |
| No | x | x | x | case 0 |
| Yes | 0 | x | x | case 0 |
| Yes | 1 | x | x | case 0 |
| Yes | 2 | Yes | x | case 1 |
| Yes | 3 | Yes | Yes | case 2 |
| Yes | unkown | x | x | case 3 |

**Table 5.1:** Shows the mapping of details on collision detection with the type of velocity generation. Here 'x' represents *don't care* as the corresponding inputs are not valuable. For the different velocity generation types, refer to Figure 5.6 (a).

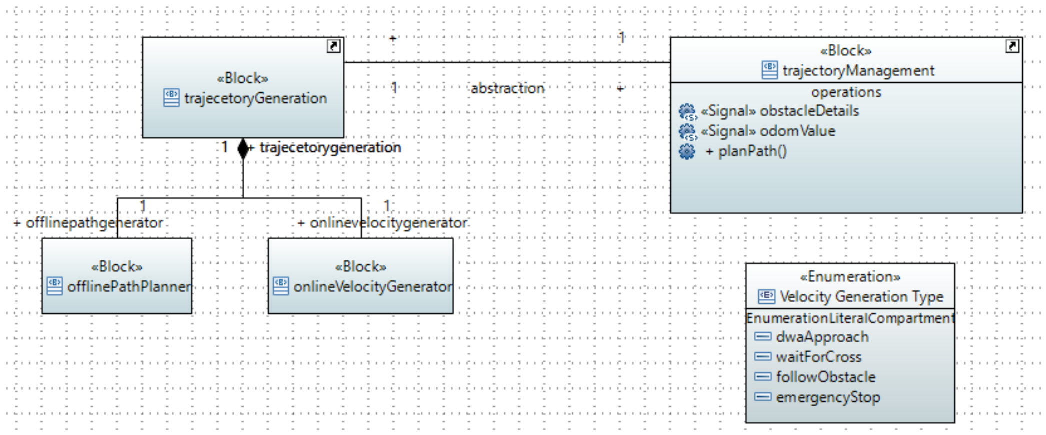


**Fig. 5.6 (a):** Block definition diagram of *Planning Subsystem* with details on *trajectoryGeneration* and *trajectoryManagement* modules are shown. The line joining these two modules shows that they are associated.

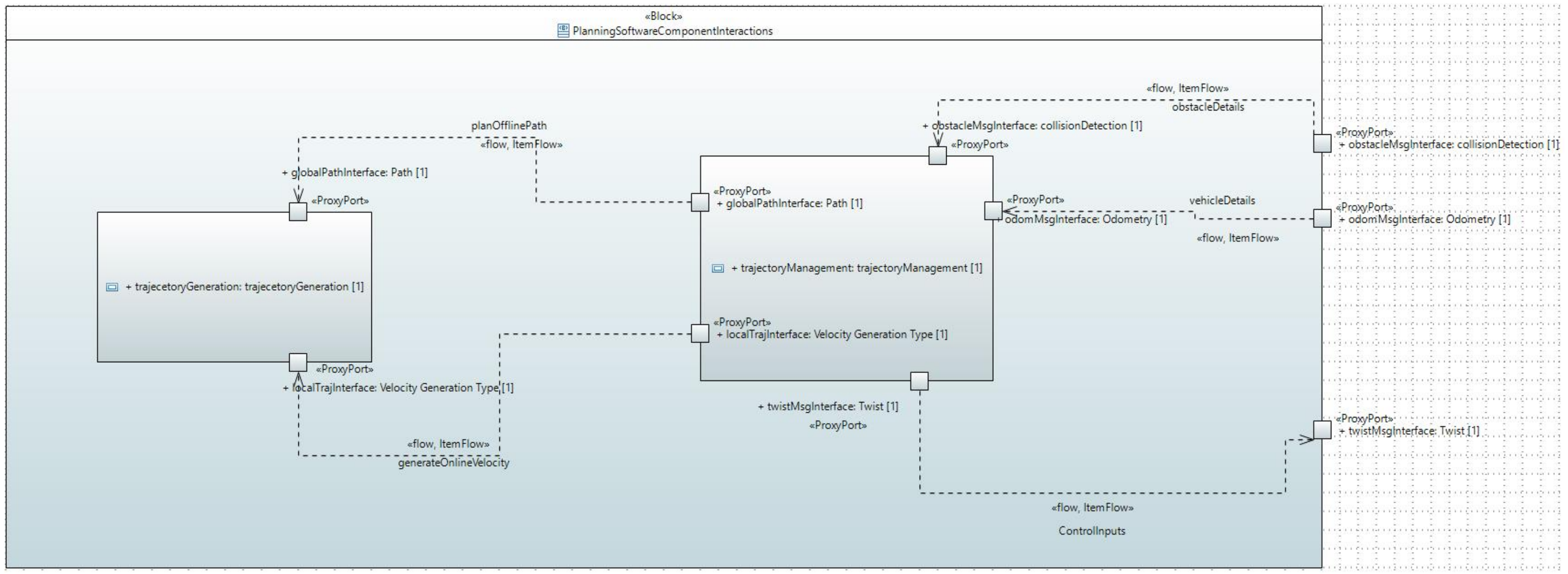

«Block»
PlanningSoftwareComponentInteractions
planOfflinePath
«flow, ItemFlow»
+ globalPathInterface: Path [1]
«ProxyPort»
+ trajecetoryGeneration: trajecetoryGeneration [1]
«ProxyPort»
+ localTrajInterface: Velocity Generation Type [1]
«flow, ItemFlow»
generateOnlineVelocity
«ProxyPort»
+ globalPathInterface: Path [1]
+ trajectoryManagement: trajectoryManagement [1]
«ProxyPort»
+ localTrajInterface: Velocity Generation Type [1]
+ twistMsgInterface: Twist [1]
«ProxyPort»
+ obstacleMsgInterface: collisionDetection [1]
«ProxyPort»
«ProxyPort»
odomMsgInterface: Odometry [1]
«flow, ItemFlow»
obstacleDetails
vehicleDetails
«flow, ItemFlow»
«flow, ItemFlow»
ControlInputs
«ProxyPort»
+ obstacleMsgInterface: collisionDetection [1]
«ProxyPort»
+ odomMsgInterface: Odometry [1]
«ProxyPort»
+ twistMsgInterface: Twist [1]

**Fig. 5.6 (b):** *PlanningSoftwareComponentInteractions* shows the internal block representation of *Planning Subsystem*. Interactions between *trajectoryManagement* and *trajectoryGeneration*; and communication of *trajectoryManagement* with other subsystems are shown.

### *5.1.1.7 Planning act*

Figures 5.7 (a) – (d) each show a flow of actions required for different behaviours of *Planning Subsystem*. Flow of actions required for planning a path is shown in Figure 5.7 (a). If a valid path is planned, the command to start AutoCar is broadcasted to different subsystems. Figure 5.7 (b) shows the activity for generating velocity values using DWA approach which falls under 'case 0' of Table 5.1. Whereas, Figure 5.7 (c) shows velocity generation to follow a moving obstacle in front (case 1) and Figure 5.7 (d) shows velocity generation to slow down the vehicle to wait for a crossing obstacle (case 2). For cases 1 and 2, once the obstacle is no longer detected by the vehicle, the respective type of velocity generation is aborted, which is represented by a circle with a cross mark inside as shown in Figure 5.7 (c) and (d), and case 0 is executed.

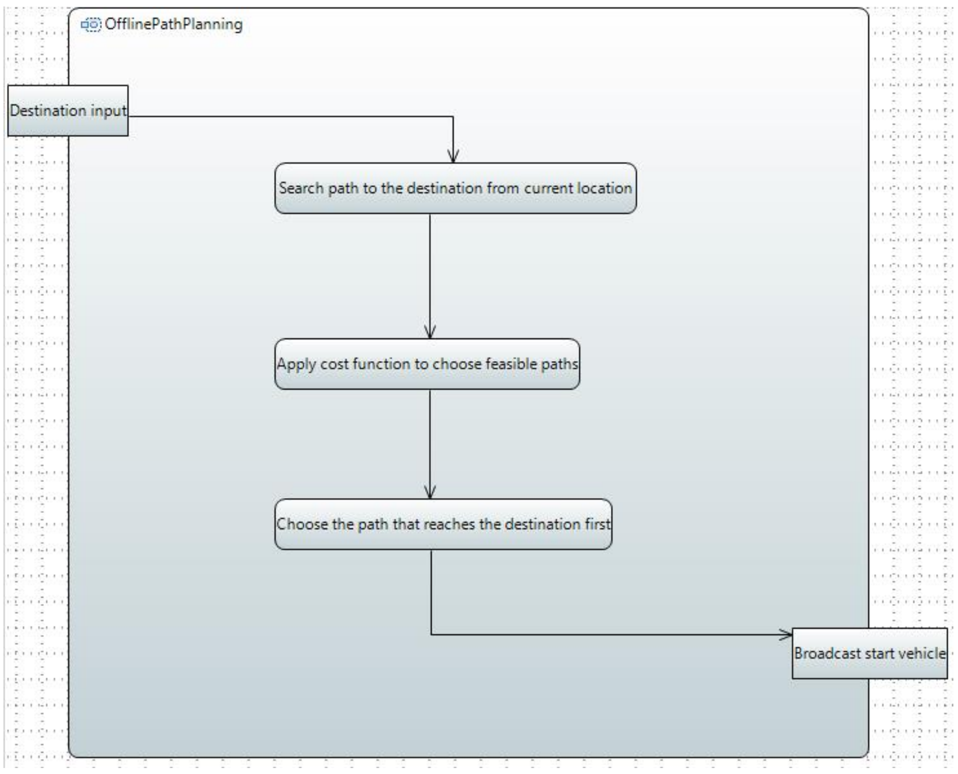

OfflinePathPlanning
Destination input
Search path to the destination from current location
Apply cost function to choose feasible paths
Choose the path that reaches the destination first
Broadcast start vehicle

**Fig. 5.7 (a):** *OfflinePathPlanning* activity diagram showing the flow of actions required to plan a path before the start of travel.

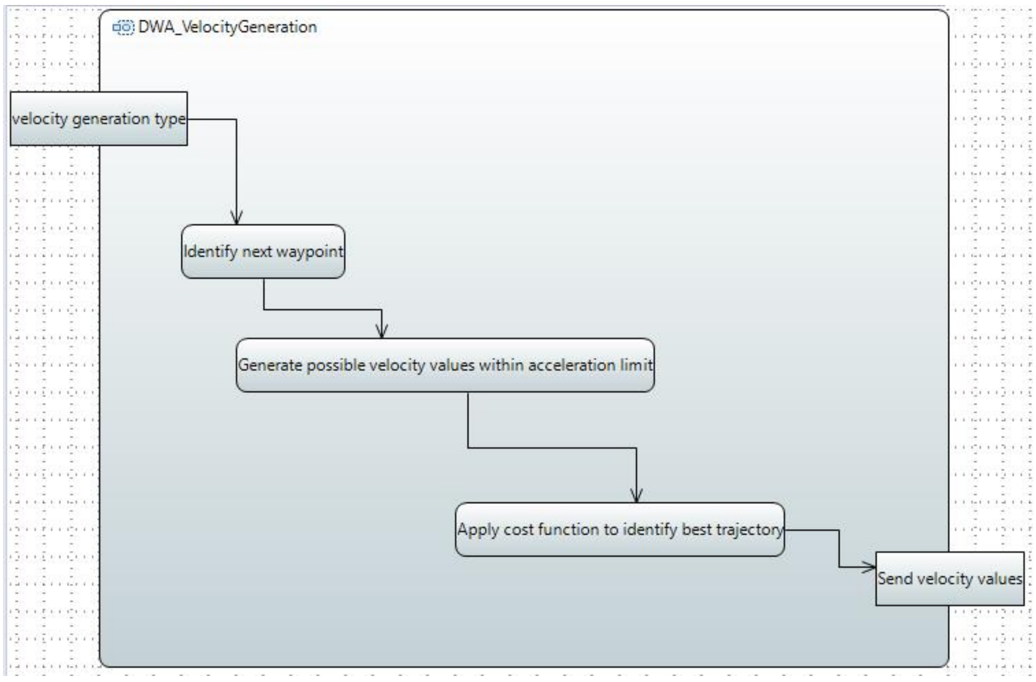

DWA_VelocityGeneration
velocity generation type
Identify next waypoint
Generate possible velocity values within acceleration limit
Apply cost function to identify best trajectory
Send velocity values

**Fig. 5.7 (b):** *DWA_VelocityGeneration* activity diagram shows the flow of actions required to generate velocity values using DWA. Here the obstacles are avoided by going around them, if present. This approach is also used when there are no obstacles present.

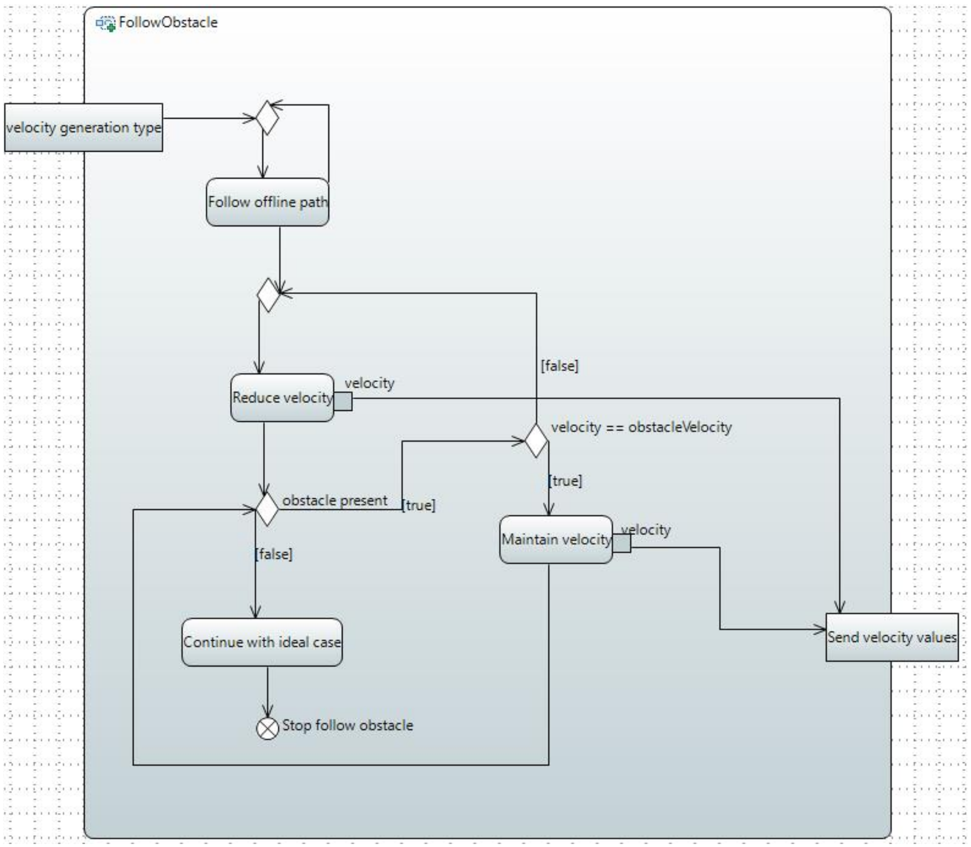


**Fig. 5.7 (c):** *FollowObstacle* activity diagram shows the flow of actions required to generate velocity values to enable AutoCar to follow the obstacle moving in front. Once the obstacle is no longer found, AutoCar shall abort this activity and resume with DWA.

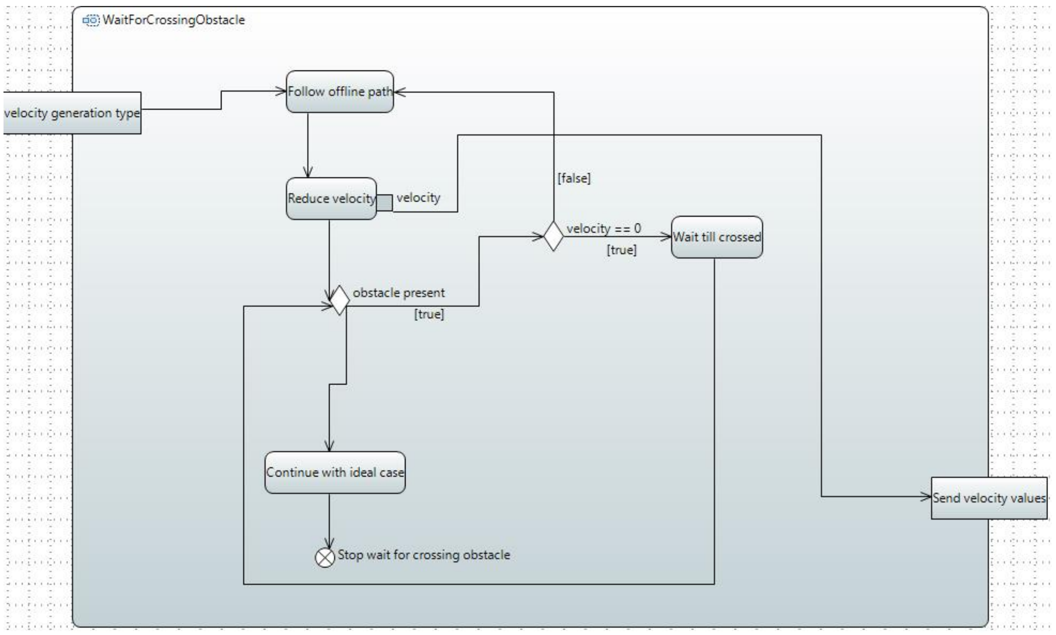


**Fig. 5.7 (d):** *WaitForCrossingObstacle* activity diagram shows the flow of actions required to generate velocity values to enable AutoCar to stop and wait for the obstacle to cross. Once the obstacle is no longer found, AutoCar shall abort this activity and resume with DWA.

#### *5.1.1.8 Planning seq*

The sequence of interactions between the modules of Planning Subsystem and interactions of the subsystem with other subsystems are shown in the sequence diagram in Figure 5.8. The blue shade partitions show the alternate conditions for different *obstacleDetected* type. For the respective type of obstacle detected, the required action is performed (as mentioned in Table 5.1). The last partition is for obstacles that do not fall in any of the defined types, i.e. obstacle type is unknown. In this case, *trajectoryManagement* sends velocity value (*twistMsg*) to stop AutoCar (execution of case 3 from Table 5.1).

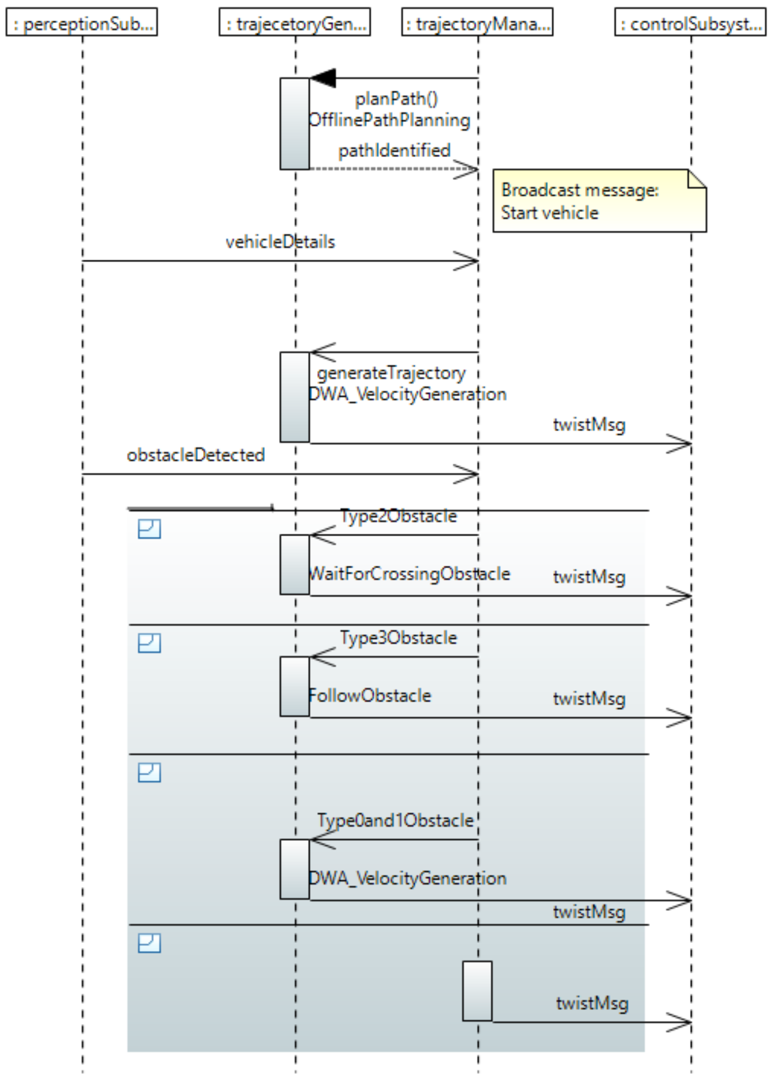


**Fig. 5.8:** *Planning_SequenceDiagram* represents the sequence of messages sent and the corresponding actions being performed.

### 5.1.1.9 *Planning stm*

In Figure 5.9, different states of *Planning Subsystem* are shown. The moment the system is started, a feasible path is identified for AutoCar in *Path planning mode*. Velocity values are generated for the vehicle to move, by default, using DWA. On obstacle detection, mapping of obstacle type with velocity generation type is performed in *Collision avoidance mode* (with reference to Table 5.1). Depending on the type of velocity generation, the subsystem gets into different modes: *Allow crossing mode*, *Follow mode* or *DWA velocity generation*. In case of unknown value of obstacle or upon arrival at the destination, the vehicle comes to a complete stop.

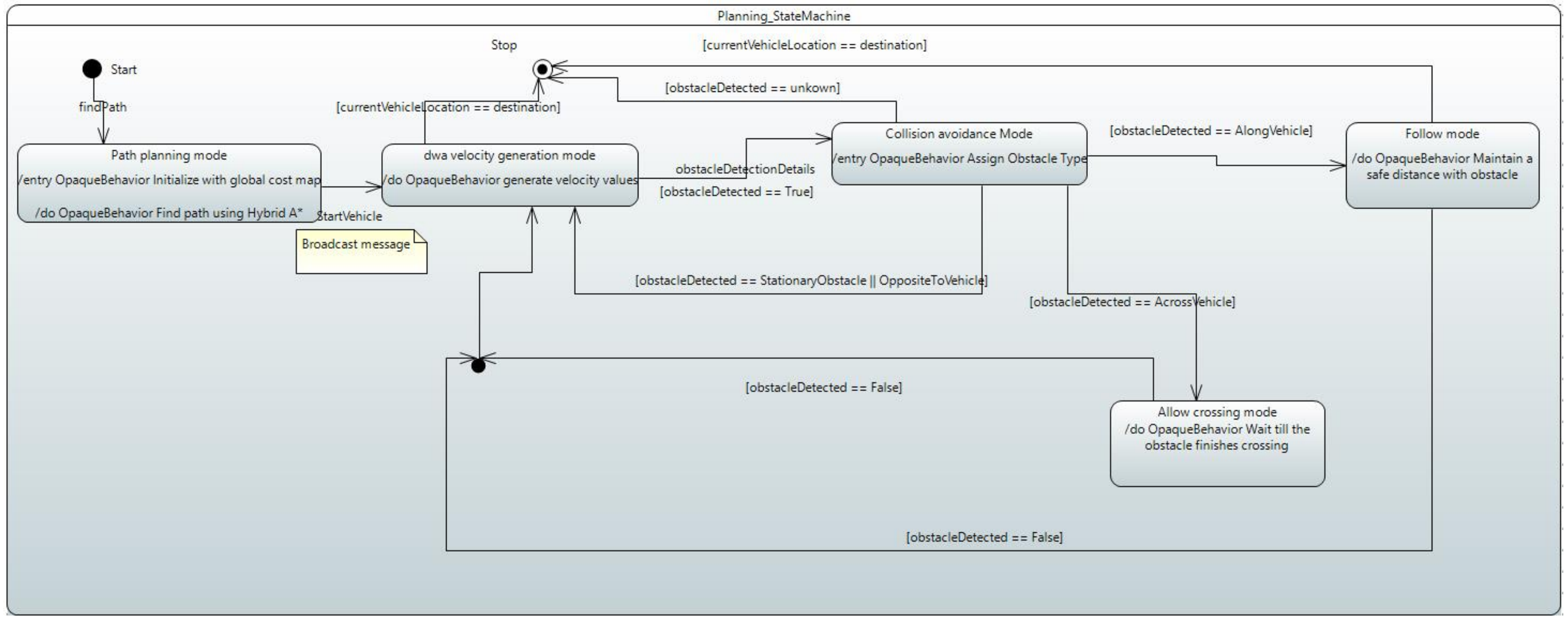

Planning_StateMachine
Start
findPath
Path planning mode
/entry OpaqueBehavior Initialize with global cost map
/do OpaqueBehavior Find path using Hybrid A*
StartVehicle
Broadcast message
Stop
[currentVehicleLocation == destination]
dwa velocity generation mode
/do OpaqueBehavior generate velocity values
obstacleDetectionDetails
[obstacleDetected == True]
[obstacleDetected == unkown]
[currentVehicleLocation == destination]
Collision avoidance Mode
/entry OpaqueBehavior Assign Obstacle Type
[obstacleDetected == AlongVehicle]
Follow mode
/do OpaqueBehavior Maintain a safe distance with obstacle
[obstacleDetected == StationaryObstacle || OppositeToVehicle]
[obstacleDetected == AcrossVehicle]
[obstacleDetected == False]
Allow crossing mode
/do OpaqueBehavior Wait till the obstacle finishes crossing
[obstacleDetected == False]

**Fig. 5.9:** State machine diagram showing the different states of *Planning Subsystem* and the trigger conditions for state transition.

### 5.1.2 Hardware design architecture of planning subsystem

The hardware components that are required by AutoCar, in general, are shown in this section. As there has been no mention of the hardware requirements for the vehicle so far, at this point, the required hardware components are listed out for individual subsystems.

#### *5.1.2.1 Vehicle Construction bdd*

Figure 5.10 (a) shows the hardware components required for *Perception Susbsystem*. Red Green Blue Depth (RGBD) camera and Light Detection and Ranging (Lidar) sensor are needed to sense the external environment. Inertial Measurement Unit (IMU) senses AutoCar's orientation and wheel encoders measure wheel's revolution per minute (rpm). Fusion of the external and internal sensor data into a vehicle perceivable format (for collision detection and classification here) is performed in the General Processing Unit (GPU).

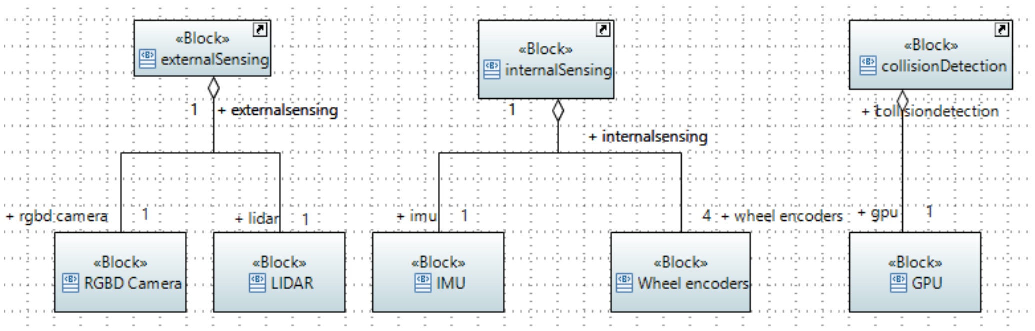


**Fig. 5.10 (a):** Hardware components required for *Perception Subsystem* are shown here.

Figure 5.10 (b) shows the hardware component required for *Planning Subsystem*. Planning requires processing of various inputs required to identify a trajectory for the vehicle to be able to move towards the goal, and this processing is performed in the GPU.

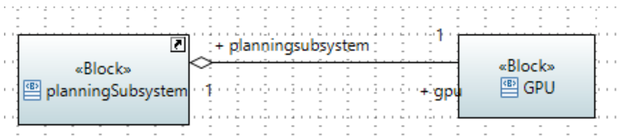


**Fig. 5.10 (b):** Hardware component required for *Planning Subsystem* is shown here.

Figure 5.10 (c) show the hardware components required for *Control Subsystem*. To execute the generated trajectory, the actuators that need to be controlled are wheels and steering angle of AutoCar.

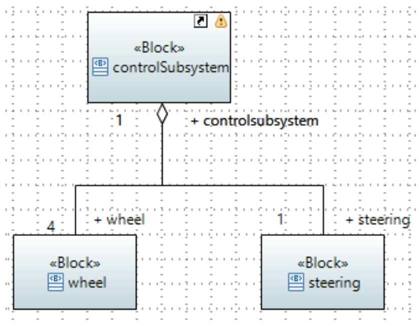


**Fig. 5.10 (c):** Hardware components required for *Control Subsystem* are shown here.

GPU maintains a global map and generates local maps depeding on current vehicle location on the global map. Thus, *Map Server Subsystem* directly depends on the GPU for its operation which is shown in Figure 5.10 (d).

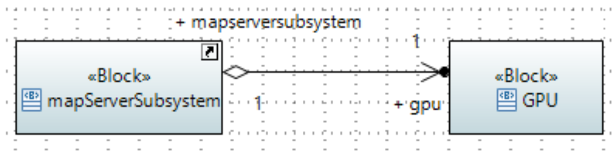


**Fig. 5.10 (d):** Hardware component required for *Map Server Subsystem* is shown here.

To enable the user to give inputs to AutoCar regarding the location of destination and the digital map, and to be able to view the vehicle's behaviour (path planned, trajectory generated), a computer is required that is connected to the GPU of AutoCar. Hence, *HMI Subsystem* requires a computer for the above-mentioned purposes as shown in Figure 5.10 (e).

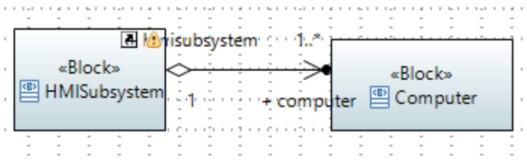


**Fig. 5.10 (e):** Hardware component required for *HMI Subsystem* is shown here.

## 5.2 Diagrams Representing Implementation Level of Planning Subsystem

In this level, ROS is used to define and hence, provide a basis for model implementation on AutoCar. ROS has readily available packages for commonly implemented functionalities. These packages are complete with the required code and documentations explaining the same. Certain concepts that are used in designing the system are directly taken from ROS and others are either modified according to the necessity or are newly defined.

### 5.2.1 Background on ROS

Though ROS is expanded to Robot Operating System, it is not an operating system but a middleware that helps provide a software framework. It provides hardware abstraction, handles low level inter process communications, supports distributed computing, provides libraries, provides tools for simulation and debugging and maintains a repository of shared source code.

ROS has been chosen for this project because it enables quick prototyping and developing a proof of concept. It is an open source programming framework with a very active community of contributors and thus, with elaborate documentation and assistance for ROS related development.

Some of the commonly used ROS terms have been listed below [27]:

**Nodes:** Nodes are individual independent processes that perform computation and are combined into a graph. These processes operate in a fine-grained scale.

**Packages:** Packages can be composed of nodes, a ROS-independent library, a dataset, configuration files or a third-party piece of software. It is a complete module that can be built and delivered for reuse.

**Messages types**: These are message descriptions that define the data structures of messages sent in ROS.

**Service types:** These are service descriptions that define the request and response data structures of services used in ROS.

**Messages:** These are the means by which nodes communicate with each other. They are data structures with type fields.

**Topics:** These are the virtual channels through which messages are shared through publish/subscribe semantics.

**Services:** As messages are means for single way communication by broadcasting, services are needed for one-to-one communication in a synchronized fashion. The client makes a service request and waits till it receives a response from the server.

*ROS navigation stack* is a readily available package that is configured specific to the robot to enable it to run autonomously. It uses inputs from sensors, understands robot's odometry, plans trajectory and thus, controls robot base with the calculated velocity values. In order to use the *navigation stack* for AutoCar's navigation, here are two major changes that needs to be made:

1. The *navigation stack* is designed for a robot to enable it to be navigating in unstructured spaces like office environment, within which it is free to move in any direction as long as it doesn't collide with obstacles. But in the case of AutoCar, it is required to move in a structured environment with 2-way roads and it is important to stick to the right side of the lane (in ideal case).
2. In ROS, the strategy for handling collision avoidance mainly involves going around the detected obstacle, i.e. avoiding obstacles, but for AutoCar different use cases are defined for collision avoidance and that needs to be taken care of in the model.

### 5.2.2 Block definition diagrams representing the application of ROS

Below are the block definition diagrams that represent application of ROS onto the system. By using *stereotype* feature of SysML, the vocabulary of this language has been extended to *ROS_Class*, *ROS_Message*, *ROS_Topic* and *ROS_Service*.

Figure 5.11 (a) shows the bdd that represents implementation of *trajectory generator* module. To develop *offlinePathPlanner* SC, a child class called *HybridAStarGlobalPlanner* is derived from the class *BaseGlobalPlanner* which is available as a ROS package. *HybridAStarGlobalPlanner* uses Hybrid A* algorithm [28] to plan a feasible path for AutoCar towards the goal. Similarly, to develop *onlineVelocityGenerator*, *BaseLocalPlanner* class is used as the base class for *DWALocalPlanner* both of which are available ROS packages. This child class uses DWA algorithm [24] for generating velocity values that are suitable for situations when no obstacle is present or when obstacles of type 0 or 1 is present (Table 5.1). For the case where the obstacle in front needs to be followed, child class called *FollowPlanner*, which is also derived from *BaseLocalPlanner*, is needed. Similarly, for the case where the vehicle needs to wait till the obstacle finishes crossing, child class called *WaitPlanner* is derived from *BaseLocalPlanner*.

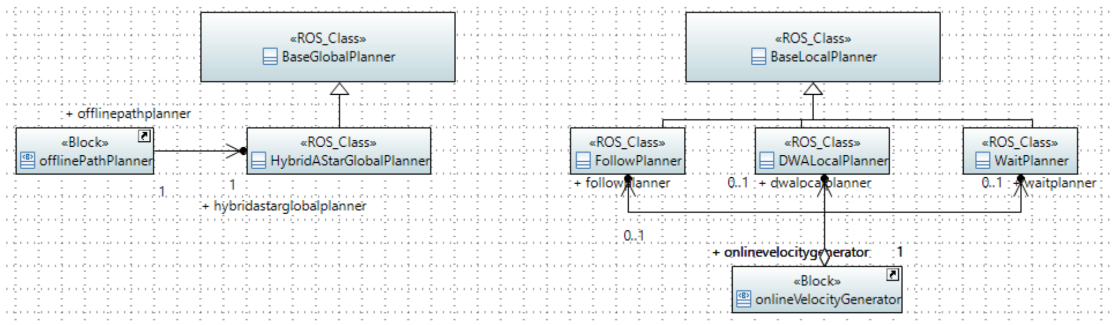

«ROS_Class» BaseGlobalPlanner
«ROS_Class» BaseLocalPlanner
+ offlinepathplanner
«Block» offlinePathPlanner
«ROS_Class» HybridAStarGlobalPlanner
1
1
+ hybridastarglobalplanner
«ROS_Class» FollowPlanner
«ROS_Class» DWALocalPlanner
«ROS_Class» WaitPlanner
+ followplanner
0..1
+ dwalocalplanner
0..1
+ waitplanner
0..1
+ onlinevelocitygenerator
1
«Block» onlineVelocityGenerator

**Fig. 5.11 (a):** Implementation of *trajectory generator* module using ROS shown as bdd.

Figure 5.11 (b) show the implementation of *trajectory manager* on AutoCar. *Trajectory manager* acts as a wrapper for *trajectory generator*. It provides an interface for interactions between *Planning Subsystem* and other subsystems. One instance of *trajectory manager*, called *pathTrigger*, requests *HybridAStarGlobalPlanner* to plan a path once ROS core starts running. Once *HybridAStarGlobalPlanner* responds back with an acknowledgement on identifying a feasible path, *pathTrigger* broadcasts ROS message to start the vehicle to the other subsystems. *pathTrigger* can be further enhanced to be able to re-plan a new path when AutoCar identifies an obstacle that completely obstructs its way towards the goal. As this is not in the scope of current operational conditions of the vehicle, it is not implemented here. Another instance of *trajectory manager*, called *obstacleMUX*, is used to map the type of obstacle received from *Perception Subsystem*, to the type of *onlineVelocityGenerator* (as shown in Table 5.1). It also decides what values need to be passed to the corresponding *onlineVelocityGenerator* type. The velocity values that are calculated are sent to *Control Subsystem*. In case, if the obstacle type is unidentifiable or doesn't fall into the defined enumeration values, *obstacleMUX* sends velocity values equating to zero to *Control Subsystem*.

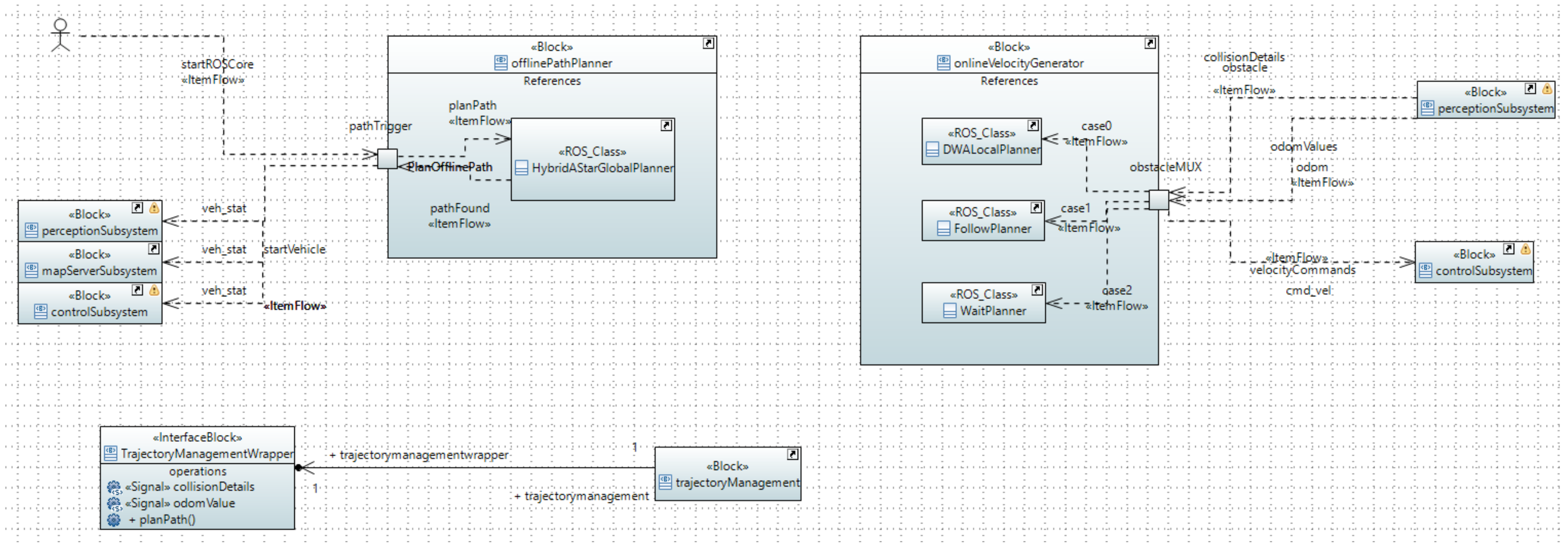


**Fig. 5.11 (b):** Implementation of *trajectory manager* module as a wrapper for *trajectory generator*. Two instances of trajectory manager are shown in the figure – *pathTrigger* and *obstacleMUX*.

Figure 5.11 (c) shows the different ROS messages, and their corresponding topics, that are broadcasted/subscribed by *Planning Subsystem*. Topics called *odom* and *obstacle* carry *odometryValue* and *collisionDetails* messages respectively. *Planning Subsystem* subscribes to these messages by listening to the respective topics. Whereas, *cmd_vel* and *veh_stat* topics carry *twistValue* and *startVehicle* messages that are broadcasted by *Planning Subsystem* by publishing them on the corresponding topics.

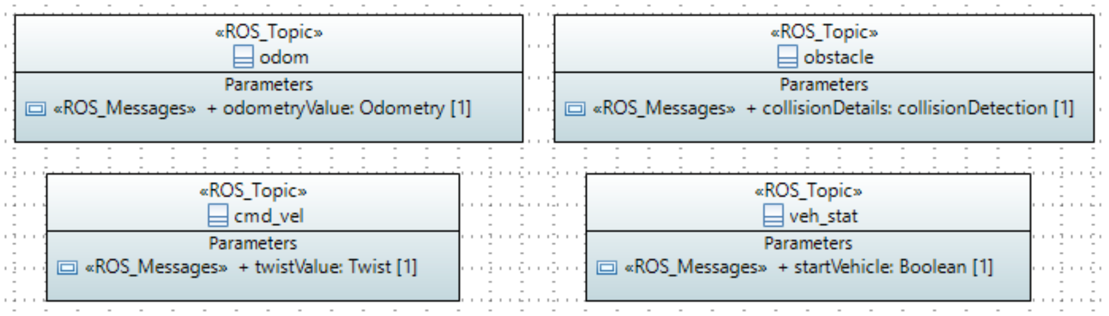

«ROS_Topic» odom
Parameters
«ROS_Messages» + odometryValue: Odometry [1]
«ROS_Topic» obstacle
Parameters
«ROS_Messages» + collisionDetails: collisionDetection [1]
«ROS_Topic» cmd_vel
Parameters
«ROS_Messages» + twistValue: Twist [1]
«ROS_Topic» veh_stat
Parameters
«ROS_Messages» + startVehicle: Boolean [1]

**Fig. 5.11 (c):** Topics and corresponding messages are shown here. *odometryValue* and *collisionDetails* are the messages that *Planning Subsystem* subscribes to and *startVehicle* and *twistValue* are the messages that it publishes.

In Figure 5.11 (d) static values that are used by the vehicle are shown. These values don't change frequently and hence can be saved in an accessible location called *parameter server*. The values related to *location/start* and *location/stop*, which contain the start and goal positions respectively; *configuration/vehicleParams*, which are the configuration parameters for AutoCar, and *configuration/mapParams*, which are the configuration parameters for both local and global maps; *internalValues/path*, which is the offline path identified for AutoCar to reach the goal, are stored in parameter server.

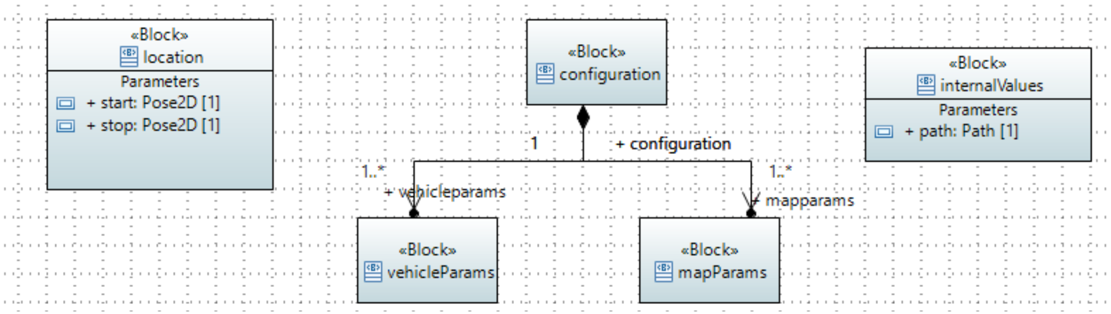


**Fig. 5.11 (d):** Values that are stored in parameter server are shown in this diagram.

Since, the vehicle cannot be started before a path from start position to the goal is identified, a client-server protocol is used to identify a feasible path, where the request for path planning is made and the client waits till the service is completed and the result is passed back as a response. Figure 5.11 (e) shows the ROS service *PlanOfflinePath* and corresponding request and response values passed.

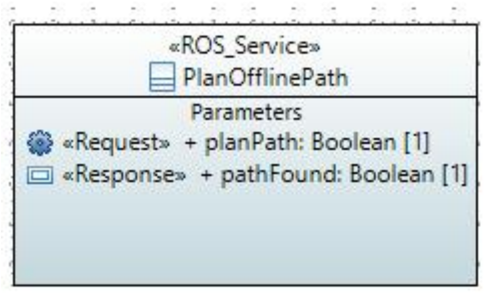

«ROS_Service»
PlanOfflinePath
Parameters
«Request» + planPath: Boolean [1]
«Response» + pathFound: Boolean [1]

**Fig. 5.11 (e):** Service request to *HybridAStarGlobalPlanner* and service response to *pathTrigger* are shown here.

In Figure 5.11 (f), value types of the data structures that are used for the purpose of trajectory planning are shown. These data structures are defined in ROS which can be readily used by including the header of the containing packages.

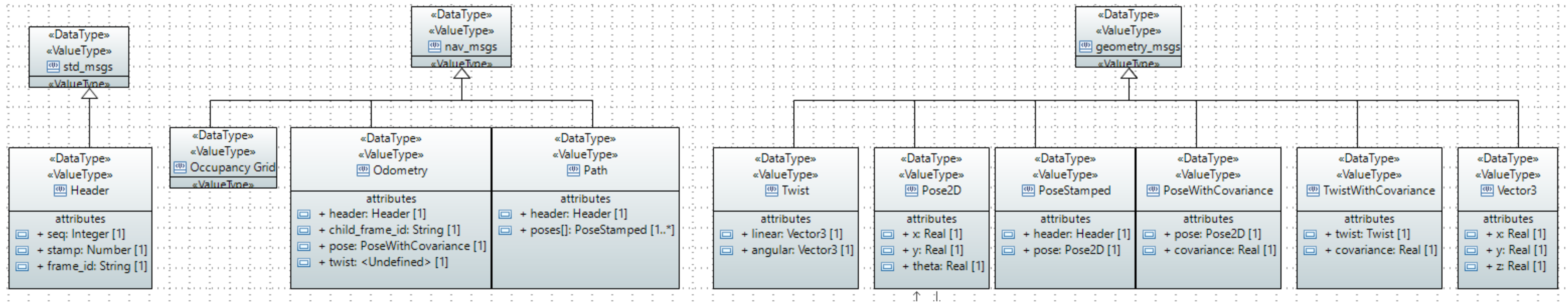

«DataType» «ValueType» std_msgs «ValueType»
«DataType» «ValueType» nav_msgs «ValueType»
«DataType» «ValueType» geometry_msgs «ValueType»
«DataType» «ValueType» Header
attributes
+ seq: Integer [1]
+ stamp: Number [1]
+ frame_id: String [1]
«DataType» «ValueType» Occupancy Grid «ValueType»
«DataType» «ValueType» Odometry
attributes
+ header: Header [1]
+ child_frame_id: String [1]
+ pose: PoseWithCovariance [1]
+ twist: <Undefined> [1]
«DataType» «ValueType» Path
attributes
+ header: Header [1]
+ poses[]: PoseStamped [1..*]
«DataType» «ValueType» Twist
attributes
+ linear: Vector3 [1]
+ angular: Vector3 [1]
«DataType» «ValueType» Pose2D
attributes
+ x: Real [1]
+ y: Real [1]
+ theta: Real [1]
«DataType» «ValueType» PoseStamped
attributes
+ header: Header [1]
+ pose: Pose2D [1]
«DataType» «ValueType» PoseWithCovariance
attributes
+ pose: Pose2D [1]
+ covariance: Real [1]
«DataType» «ValueType» TwistWithCovariance
attributes
+ twist: Twist [1]
+ covariance: Real [1]
«DataType» «ValueType» Vector3
attributes
+ x: Real [1]
+ y: Real [1]
+ z: Real [1]

**Fig. 5.11 (f):** Value types of the data structures that is used for implementation are shown here.

# 6 Model Evaluation

In this chapter, the model that has been designed so far is evaluated in terms of its completeness and correctness. An assessment of its completeness is achieved by ensuring that the vehicle level requirements are broken down into fine executable requirements and these requirements are in turn captured by different model elements. System correctness is evaluated as defined in [31]. Finally, model summary is presented showing the so far designed system model using plain blocks.

## 6.1 Model Completeness

The model is assessed to be complete under the following conditions:

1. The high-level requirements are broken down into simple and executable requirements.
2. The defined requirements are in turn captured by different kinds of SysML diagrams.

Table 6.1 (a) to 6.1 (c) represent the requirements breakdown using a requirement traceability matrix. Table 6.1 (a) shows the requirements breakdown of *Planning Subsystem*. Table 6.1 (b) shows the requirements breakdown of *Perception Subsystem*. Table 6.1 (c) shows the requirements breakdown of *Map Server Subsystem*, *Control Subsystem* and *HMI Subsystem*.

| Vehilce Level | Analysis Level | Design Level I | Design Level II |
|---|---|---|---|
| The vehicle shall autonomously travel from the start point to the destination. | The vehicle shall generate an executable and collision free trajectory towards its destination. | The vehicle shall generate velocity values depending on the planned path and current environment. | The generated velocity shall be free from collision with detected obstacle. |
| | | | Path generated by velocity commands shall be as close as possible to the offline path. |

| | | | |
|---|---|---|---|
| | | An optimal path from the start to destination shall be identified. The waypoints of the path shall be saved. | The vehicle shall travel in the shortest possible time. |
| | The vehicle shall manage trajectory generation depending on the inputs received. | Command to stop the vehicle immediately shall be sent to control subsystem when no data regarding obstacle detection is received over a period of time. | |
| | | It shall enable the switch between path planning and velocity generation. | |
| | | It shall act as an interface between perception, control and map server subsystems. | |
| The vehicle shall avoid colliding with potential obstacles. | The vehicle shall generate an executable and collision free trajectory towards its destination. | | |
| | The vehicle shall manage trajectory generation depending on the inputs received. | | |
| The vehicle shall travel in the shortest possible time. | | An optimal path from the start to destination shall be identified. The waypoints of the path shall be saved. | |

**Table 6.1 (a):** Requirement traceability matrix for *Planning Subsystem*.

| Vehilce Level | Analysis Level | Design Level I | Design Level II | Design Level III |
|---|---|---|---|---|
| The vehicle shall autonomously travel from the start point to the destination. | The vehicle shall sense its internal state and external environment. | The vehicle shall sense external environment. | The lanes shall be detected by the vehicle. | |
| | | | Objects that are not available on the map are identified by the vehicle on its way. | The relative orientation of the object shall be identified. The relative velocity of the object shall be detected. |
| | | The internal state of AutoCar shall be detected. | AutoCar's orientation w.r.t the world coordinates shall be identified. | |
| | | | AutoCar's velocity shall be detected. | |
| | The vehicle shall combine the inputs from different sensors to produce the required information. | Obstacles shall be identified and differentiated from unmapped objects. | | |
| | | Possible region for collision with the unmapped objects shall be identified. | | |
| | The vehicle shall locate its position on the given map. | | | |

**Table 6.1 (b):** Requirement traceability matrix for *Perception Subsystem*.

| Subsystem | Vehilce Level | Analysis Level | Design Level I | Design Level II |
|---|---|---|---|---|
| HMI subsystem | The vehicle shall autonomously travel from the start to the destination. | The vehicle shall receive the input for final destination. | | |
| | | The vehicle shall receive detailed digital map before start of journey. | | |
| Map server subsystem | The vehicle shall autonomously travel from the start to the destination. | Global map shall be used to roughly plan a path from start to destination. | Global map shall be initialized with input map. | Barriers that are available in the initial map shall be inflated proportional to the dimensions of AutoCar. |
| | | | | Costs shall be added to the left lane. |
| | | Local map shall be used for precise trajectory generation. | Local map shall be configured as a window within the global map that moves with AutoCar's center. | |
| | | | Local map shall provide local environment to enable AutoCar to generate trajectory for that instance. | |

| | | | | |
|---|---|---|---|---|
| | | | Local map shall be updated with possible region for collision with the detected obstacles. | |
| Control subsystem | The vehicle shall autonomously travel from the start to the destination. | The vehicle shall follow the generated trajectory. | | |
| | | The vehicle shall stabilize the platform as the trajectory is followed. | | |

**Table 6.1 (c):** Requirement traceability matrix for *Map Server Subsystem*, *HMI Subsystem* and *Control Subsystem*.

Table 6.2 (a) shows the diagrams that satisfy *Top Level* requirements. Table 6.2 (b) lists the diagrams that capture *Detailed* requirements. Table 6.2 (c) shows the diagrams that capture *Planning* requirements. Table 6.2 (d) and 6.2 (e) show the diagram that captures the requirements laid out for *Perception* and *Map Server* respectively.

| **Top Level req** | | | |
|---|---|---|---|
| **Requirement ID** | **Requirement description** | **Diagram type** | **Diagram name** |
| R-01 | The vehicle shall autonomously travel from start to destination. | Activity diagram | AutoCar Overall Functionality act |
| R-02 | The vehicle shall avoid colliding with potential obstacles. | Use case diagram | Collision avoidance uc |
| R-03 | The vehicle shall travel in the shortest possible time. | Requiement diagram | Planning req |

**Table 6.2 (a):** Diagrams that represent requirements listed in *Top Level* are shown here.

| Detailed req | | Diagram type | Diagram name |
|---|---|---|---|
| Requirement ID | Requirement description | | |
| FR_2.1 | The vehicle shall generate executable and collision free trajectory towards its destination. | Block definition diagram | AutoCar Composition bdd |
| FR_2.2 | The vehicle shall manage the generated trajectory depending on the received inputs. | Block definition diagram | AutoCar Composition bdd |
| | | Internal block diagram | Planning Subsystem Interactions ibd |
| FR_0.1 | The vehicle shall receive detailed digital map before start of journey. | Activity diagram | AutoCar Overall Functionality act |
| FR_0.2 | The vehicle shall receive the input for final destination. | Activity diagram | AutoCar Overall Functionality act |
| FR_1.1 | The vehicle shall sense its internal state and external environment. | Block definition diagram | AutoCar Composition bdd |
| | | Activity diagram | AutoCar Overall Functionality act |
| | | Internal block diagram | Planning Subsystem Interactions ibd |
| FR_1.2 | The vehicle shall combine the inputs from different sensors to produce the required information. | Block definition diagram | AutoCar Composition bdd |
| FR_1.3 | The vehicle shall locate its position on the given map. | Block definition diagram | AutoCar Composition bdd |
| FR_3.1 | The vehicle shall follow the generated trajectory. | Block definition diagram | AutoCar Composition bdd |
| | | Internal block diagram | Planning Subsystem Interactions ibd |

| | | | |
|---|---|---|---|
| | | Block definition diagram | AutoCar Composition bdd |
| | | Internal block diagram | Planning Subsystem Interactions ibd |
| FR_4.1 | Global map shall be used to roughly plan a path from start to destination. | Activity diagram | AutoCar Overall Functionality act |
| FR_4.2 | Local map shall be used for precise trajectory planning. | Block definition diagram | AutoCar Composition bdd |
| | | Internal block diagram | Planning Subsystem Interactions ibd |
| | | Activity diagram | AutoCar Overall Functionality act |

**Table 6.2 (b):** Diagrams that represent the requirements listed out in *Detailed req* are shown here.

| **Planning req** | | | |
|---|---|---|---|
| **Requirement ID** | **Requirement description** | **Diagram type** | **Diagram name** |
| FR_2.2_3 | Command to stop the vehicle shall be sent to control subsystem when no proper data regarding obstacle detection is received over a period of time. | Sequence diagram | Planning seq |
| | | Statemachine diagram | Planning stm |
| FR_2.2_2 | Trajectory management shall enable the switch between path planning and velocity generation. | Internal block diagram | Planning Bock Diagrams |
| | | Sequence diagram | Planning seq |
| | | Statemachine diagram | Planning stm |
| FR_2.2_1 | Trajectory management shall act as an interface between perception, map server subsystem and control subsystems. | Sequence diagram | Planning seq |
| | | Internal block diagram | Planning Bock Diagrams |
| FR_2.2_1 | The vehicle shall generate velocity | Sequence diagram | Planning seq |

| | | | |
|---|---|---|---|
| | commands depending on the planned path and current environment. | Block definition diagram | Planning Bock Diagrams |
| | | Internal block diagram | Planning Bock Diagrams |
| | | Activity diagram | Planning act |
| FR_2.1_2 | Optimal path from start to destination shall be identified. | Activity diagram | Planning act (Offline Path Planning) |
| | | Statemachine diagram | Planning stm |
| FR_2_1 | Trajectory planning shall be broken down into path planning and velocity planning. | Block definition diagram | Planning Bock Diagrams |
| | | Internal block diagram | Planning Bock Diagrams |
| | | Activity diagram | Planning act |
| | | Sequence diagram | Planning seq |
| | | Statemachine diagram | Planning stm |
| FR_2.2_1 | The generated trajectory shall be free from collision with detected obstacles. | Block definition diagram | Planning Bock Diagrams |
| | | Activity diagram | Planning act (DWA Velocity Generation, Follow Obstacle, Wait for Crossing Obstacle) |
| | | Sequence diagram | Planning seq |
| | | Statemachine diagram | Planning stm |
| FR_2.2_1.1 | Path generated by velocity commands shall be as close as possible to the offline path. | Block definition diagram | Planning Bock Diagrams |
| | | Activity diagram | Planning act (DWA Velocity Generation) |
| | | Sequence diagram | Planning seq |

| | | | |
|---|---|---|---|
| | | Statemachine diagram | Planning stm |
| NFR_2.1_2 | The planned path shall be on the right lane of the road. | Requirement diagram | Map Server req |

**Table 6.2 (c):** Diagrams that represent the requirements listed out in *Planning req* are shown here.

| Perception req | | Diagram type | Diagram name |
|---|---|---|---|
| **Requirement ID** | **Requirement description** | | |
| FR_1.3_1.2 | The lanes shall be detected by the vehicle. | Block definition diagram | Perception bdd |
| FR_1.3_1.1 | Objects that are not available on the map are identified by the vehicle on its way. | | |
| FR_1.3_1.1.1 | The relative orientation of the object shall be identified. | | |
| FR_1.3_1.1.2 | The relative velocity of the object shall be identified. | | |
| FR_1.3_2.1 | AutoCar's orientation w.r.t the world coordinates shall be identified. | | |
| FR_1.3_2.2 | AutoCar's velocity shall be detected. | | |
| FR_1.1_2 | Obstacles shall be identified and differentiated from unmapped objects. | | |
| FR_1.1_1 | Possible region for collision with the unmapped objects shall be identified. | | |

**Table 6.2 (d):** Diagram that represents the requirements listed out in *Perception req* is shown here.

| Map Server req | | Diagram type | Diagram name |
|---|---|---|---|
| **Requirement ID** | **Requirement description** | | |
| NFR_4.1_1 | Global map shall be initialized with the input | Use case diagram | Map Server uc |

| | | | |
|---|---|---|---|
| | map. | | |
| FR_4.1_1.2 | Barriers that are available in the initialized map shall be inflated proportional to the dimensions of AutoCar. | | |
| NFR_4.1_1.1 | Costs shall be added to left lane. | | |
| NFR_4.2_1 | Local map shall be configured as a window within the global map that moves with AutoCar's center. | | |
| FR_4.2_2 | Local map shall provide a local environment to enable AutoCar to generate trajectory for that instance. | | |
| FR_4.2_1.1 | Local map shall be updated with possible region for collision with the detected obstacles. | | |

**Table 6.2 (e):** Diagram that represents the requirements listed out in *Map Server req* is shown here.

## 6.2 Model Correctness

As mentioned by Joseph Sifakis in [31], system model correctness is defined by the following factors:

1. Trustworthiness: It is the ability of a system to perform reliably despite possible hazards (software failure, unpredictable environment, erroneous interactions).
2. Cost efficiency: It is minimization of resource usage and parameters of an execution platform.
3. Performance: It is a measure of system throughput and latency.

The system model derived in this thesis has been designed to enter a safe state (setting zero velocity values for the actuators of the vehicle) under unpredictable perception conditions. Identification of different kinds of obstacles for the given functional domain of the vehicle, ensures possible external hazard identification and handling of the same. Software failures are not focused during the model development and hence they are not handled here.

Both cost efficiency and performance cannot be improved together as they conversely affect the other, but an optimal point is to be identified for system development. For path planning, in order to ensure identification of close to ideal path, the vehicle applies *Hybrid A** algorithm. Due to non-holonomic nature of the vehicle, this algorithm ensures an executable path before the start wherein, during the run-time, under ideal conditions, velocity values are calculated for the given path without the need for identifying velocity values for all possible executable paths and assessing the cost function for each of them. This improves the real time performance of the vehicle and reduces vehicle slowdowns and jerks.

Inclusion of *Trajectory management* module in front of *Trajectory generation* module creates an overhead in terms of processor utilization and data transfer but adds a level of abstraction and enables parallel processing as *Trajectory management* subscribes to corresponding *ROS topics* and interprets the incoming data for *Trajectory generation* module which in turn generates vehicle velocity values that are published on corresponding *ROS topics* by the former module.

## 6.3 Model Summary

*Implementation Level* of the system model confirms the model feasibility of the system defined from *Vehicle Level* to *Design Level*. It is the level closest to the actual implementation, either through simulation or on AutoCar. As defined in Section 5.2, this level is achieved by application of ROS middleware.

The final system comprises of the subsystems as shown in the below figure. These subsystems majorly interact through ROS's broadcast message protocol.

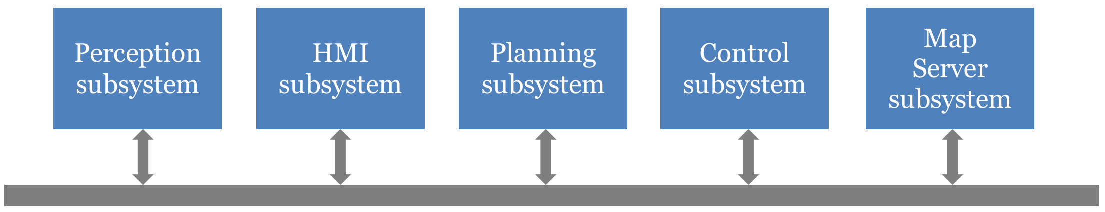


**Fig. 6.1:** Subsystems that make AutoCar and the communication between them through broadcast messages.

Since, the project mainly focuses on trajectory planning for the vehicle, *Planning Subsystem* is of major interest. Figure 6.2 shows the interactions of *Planning Subsystem* with other subsystems. *Planning Subsystem* receives *Vehicle odometry* and *Obstacle details* as inputs from *Perception Subsystem* through respective broadcast messages. Similarly, it accesses *costmap class* of *Map Server Subsystem* for *Local* and *Global costmap*. The planned trajectory is broadcasted by *Planning Subsystem* which is in turn subscribed by *Control Subsystem* to control wheel rotation.

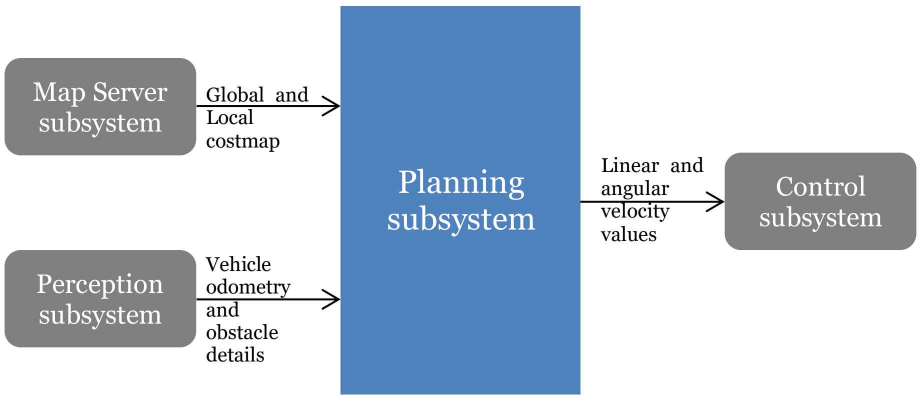


**Fig. 6.2:** Interaction of *Planning Subsystem* with other subsystems. The data being passed, and the direction of interaction are also included in the figure.

Figure 6.3 shows the overall composition of *Planning Subsystem*. This subsystem comprises of:

1. ***Trajectory generation* module:** This module is responsible for planning a rough path from start to destination using *Offline path planning* SC and generating velocity vector values ($v$, $\omega$) (refer to Chapter 4) using *Online velocity generation* SC for the roughly planned path with respect to the current environment sensed by the vehicle.

2. ***Trajectory management* module:** An instance of this module, *Plan offline path*, is responsible for initiating path planning once the required inputs are received and the trigger for the same is initiated. The other instance of this module, *Obstacle MUX* (multiplexer), multiplexes the type of velocity to be generated depending on the type of obstacle being detected.

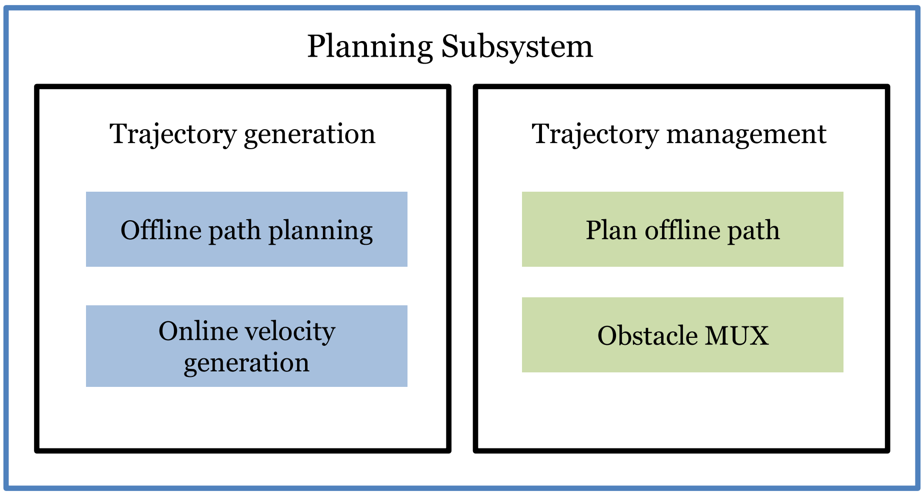


**Fig. 6.3:** Composition of *Planning Subsystem*.

In Figure 6.4, the initial path planning process is shown. As the ROS core is initiated, *Plan offline path* instance of *Trajectory management* requests *Hybrid A* path planner* to look for a feasible path. Once a path is identified, an acknowledgement is sent back, which in turn sends a message to the other subsystems requesting the start of the vehicle. *Hybrid A* path planner* is derived from an available ROS package called *Base Global Planner*. This derived path planner uses Hybrid A* algorithm to find a feasible path to the destination.

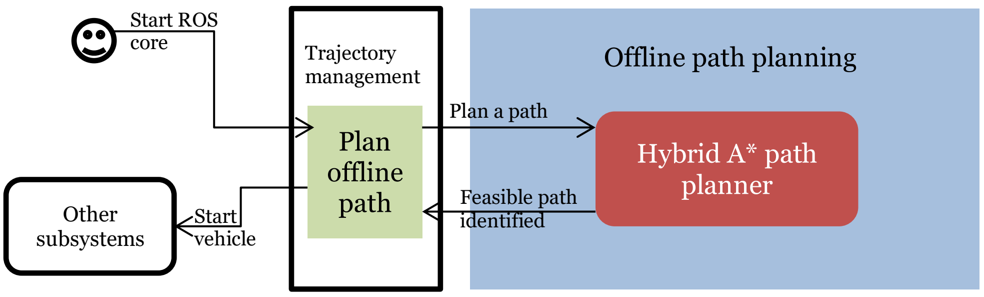


**Fig. 6.4:** Interactions within *Planning Subsystem*, and interactions of *Planning Subsystem* with external entity and other subsystems for Offline path planning using Hybrid A* algorithm.

In Figure 6.5, it can be seen that depending on the obstacle type detected by *Perception Subsystem*, *Obstacle MUX* instance of *Trajectory management* invokes one of the *Planners* of *Online Velocity Generation* SC (refer to Section 4.3 and Table 5.1 for details). Generated velocity values from the corresponding *Planner* is sent to *Control Subsystem*.

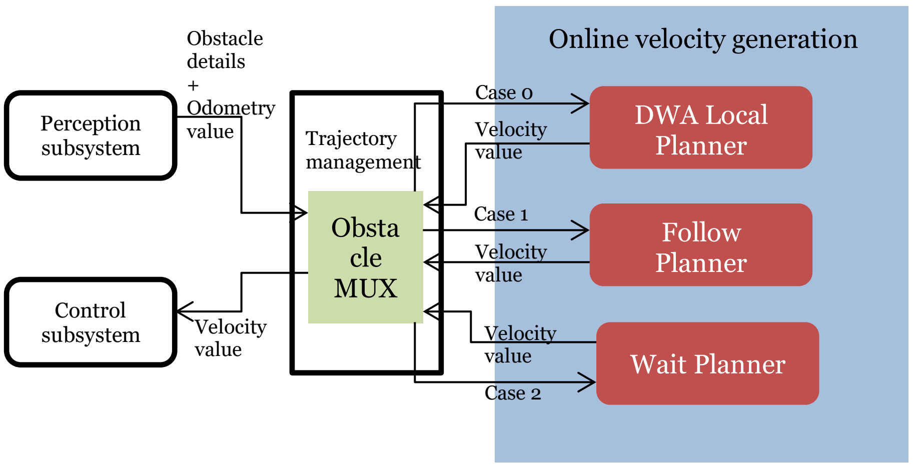


**Fig. 6.5:** Vehicle velocity generation depending on the type of obstacle being detected. One of the three *Planners* is invoked by obstacle MUX to generate velocity values based on the obstacle type.

# 7 System Design Analysis and Conclusion

The system design concludes with modeling the system from *Vehicle Level* to *Implementation Level*. This chapter presents an overview on design phases involved in the modeling process and how these phases are adopted for the chosen design methodology.

A conclusion on what is achieved at the end of this thesis through application of MBSE for the given system development and analyzing the effects on the same is presented.

This chapter ends with a section on future scope of the thesis, an overview on how the physical system can be further developed from the established model.

## 7.1 Analysis on System Design Phases

The different design phases that are focused for the vehicle model development through the thesis project are:

- Requirement analysis and requirement specification which involves defining top level system requirements from customer needs, and further deriving detailed requirements.
- Defining system architecture which involves identifying the subsystems required for system operation and their communication.
- Detailed design of *planning subsystem* for the purpose of vehicle's trajectory planning by identifying the required software components, required data for generating a feasible trajectory and the means of communication between these components.
- Implementation specification wherein usage of ROS for development of the required software components are represented through SysML diagrams.

Actual code implementation, integration and verification phases of system design are not focused in this thesis but are to be carried out as next steps towards the completion of AutoCar project (refer to Section 7.2).

At *Vehicle level*, user level requirements are derived from the given problem statement and external entities interacting with the system are identified from the vehicle's operational conditions.

Subsystem identification and representation through SysML structural diagrams; their abstract and independent functional system responsibilities and their interactions shown through SysML behavioral diagrams constitute *Analysis level* wherein the system architecture is established, and overall vehicle functionality is understood.

Detailed design of trajectory planning functionality of the vehicle involves identification of required SCs, their inputs/outputs and interactions at *Design level*.

Prior to actual code implementation, a specification of ROS usage for SC development is performed at *Implementation level*.

## 7.2 Analysis on System Design Methodology

As mentioned in [30] a design methodology is important to be adopted to help guide design decisions for developing a complex system. The methodology adopted for the system design development follows a top-down approach which is achieved by defining the system with reference to the levels of EAST-ADL: *Vehicle level*, *Analysis level*, *Design level* and *Implementation level*. System details increase as the levels are traversed and they are represented by corresponding SysML diagrams. Though the EAST-ADL methodology can be further extended by adding orthogonal activities with corresponding verification and validation activities like the *V-model* for a comprehensible system development [21], the approach used for this thesis is slightly different. In the methodology used for AutoCar modeling, requirements analysis and specification are performed at each of the above-mentioned level with reference to the corresponding level of vehicle details and only the design phases mentioned on the left side of the below shown V-model have been focused upon as it is a conceptual model development.

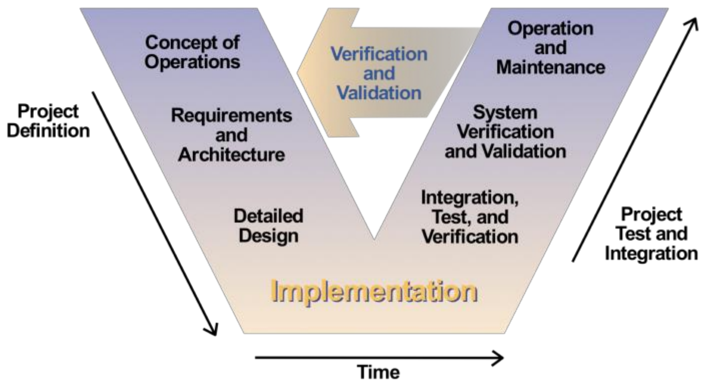

Concept of Operations
Verification and Validation
Operation and Maintenance
Project Definition
Requirements and Architecture
System Verification and Validation
Detailed Design
Integration, Test, and Verification
Project Test and Integration
Implementation
Time

**Fig. 7.1:** *V-model* representation of system development lifecycle. (source: Wikipedia)

With reference to the methodologies mentioned in [32] the applied methodology for this project closely matches the INCOSE Object-Oriented Systems Engineering Methodology (OOESM) as it is a top-down approach and focuses on designing the system to enable an object-oriented software development which is supported by ROS middleware. The methodology that is chosen for system development sometimes depends on the nature of the system being considered. For example, as listed out in [32], JPL State Analysis and Dori Object Process Methodology are appropriate for a state-based systems where the focus is mainly on identification of various system states, their transition processes and interactions; IBM Rational Unified Process (RUP) for SE is a methodology applied at enterprises for commercial product development that focuses on business processes and concurrent design phases which is possible to achieve with cross-functional teams. IBM Telelogic Harmony-SE, on the other hand, follows V-model system development approach which assists in a reliable system development (for production) that includes model driven testing too. As AutoCar model development is mainly conceptual and as the system under consideration is simpler (not state based) with restricted operational conditions, the methodology that is applied is considered sufficient with a shorter design lifecycle.

## 7.3 Conclusion

Hence, with reference to the goals stated in Section 1.4, the mentioned primary goals are achieved at the end of thesis. However, as an embedded system, with requirements for collision avoidance with different kinds of obstacles, it is important to assess the timing constrains to achieve a real time behavior of the vehicle. Requirement analysis on this aspect is missing in this thesis which needs to be performed at the design level of the model. It is also needed to perform an analysis on the different possible failure modes of the system, understanding their causes and effects, and assessing the defined system's reliability.

Defining the system from a top-down perspective helps gain a gradual and progressive understanding of the system, its functionalities and behavior. This step-by-step approach provides a basis to view the system and define it with respect to the corresponding perspective. This, apart from assisting in system definition, also provides a systematic structure for future referencing, even by those not involved in system's design development. The system model that has been developed using this methodology, thus, gives a good reference to develop the physical system capable of generating a trajectory for the vehicle to move towards its destination.

However, on the downside, in order to model a system using MBSE, it is expected that one is familiar with its concepts and also proficient with at least one modeling language. Learning a language takes considerable time for understanding the syntax and semantics, and also requires a good practice before its application. These add to a long learning period before being able to use it for a project. Choosing the right MBSE methodology, depending on the system under development, is essential which can otherwise become an overhead complicating the development process. Therefore, it is important assess the different available methodologies before choosing the appropriate one.

## 7.4 Future Scope of AutoCar Project Execution

The derived model lacks timing analysis which needs to be performed to define the timing behavior of the system as a next step. System reliability and performance need to be assessed and an analysis on their improvement is to be performed. The finally improved and reliable model is implemented using ROS middleware wherein the identified SCs (of *Implementation Level*) are coded either using C++ or Python. These components are individually verified through a simulation in Rviz (integrated simulation tool in ROS) by providing stub inputs. These SCs are in turn integrated to form *Planning Subsystem*. The subsystem functionality of identifying optimal trajectory during runtime is verified.

Any deviations identified through these simulations are captured and resolved by modifying the system model accordingly. This iteration is carried out till the expected system behavior is obtained. This way, the system model remains updated and the incremental changes are verified in a systematic manner. The final step to this is to deploy the verified software on AutoCar hardware and validate the same.

## References

[1] Hod Lipson, Melba Kurman, “*Driverless: Intelligent Cars and the Road Ahead*”, The MIT Press, 1st edition, September 2016

[2] Defense Advanced Research Projects Agency (DARPA) official website: https://www.darpa.mil/

[3] Ömer Sahin Tas, Florian Kuhnt, J. Marius Zöllner and Christoph Stiller, “*Functional System Architectures towards Fully Automated Driving*”, Intelligent Vehicles Symposium (IV), IEEE, 2016

[4] Kamal Kant, Steven W. Zucker, “*Towards Efficient Trajectory Planning: The Path-Velocity Decomposition*”, The International Journal of Robotics Research, Volume 5, Issue 3, September 1986, Pages: 72-89

[5] P. Falcone, F. Borrelli, H. E. Tseng, J. Asgari, D. Hrovat, “*A Hierarchical Model Predictive Control Framework for Autonomous Ground Vehicles*”, American Control Conference, 2008

[6] Brian Paden, Michal Čáp, Sze Zheng Yong, Dmitry Yershov and Emilio Frazzoli, “*A Survey of Motion Planning and Control Techniques for Self-Driving Urban Vehicles*”, IEEE Transactions on Intelligent Vehicles, Volume 1, Issue 1, March 2016, pages: 33-35

[7] Martin Buehler, Karl Iagnemma, and Sanjiv Singh, “*The 2005 DARPA Grand Challenge: The great robot race*”, volume 36, Springer, 2007

[8] Sagar Behere, Martin Törngren, “*A functional reference architecture for autonomous driving*”, Peer reviewed journal, Information and Software Technology, volume 73, 2016, pages: 136–150

[9] Benjami Nordell, "*Trajectory Planning for Autonomous Vehicles and Cooperative Driving*", Master Thesis report, KTH Royal Institute of Technology, 2016

[10] The DARPA Urban Challenge website: https://www.darpa.mil/about-us/timeline/darpa-urban-challenge

[11] Waymo Safety Report: On the Road to Fully Self-Driving, online: https://storage.googleapis.com/sdc-prod/v1/safety-report/waymo-safety-report-2017.pdf

[12] Julius Ziegler, Philipp Bender, and Markus Schreiber, Henning Lategahn, Tobias Strauss, and Christoph Stiller, Thao Dang, Uwe Franke, Nils Appenrodt, Christoph g. Keller, Eberhard Kaus, Ralf g. Herrtwich, Clemens Rabe, David Pfeiffer, Frank Lindner, Fridtjof Stein, Friedrich Erbs, Markus Enzweiler, Carsten Knöppel, Jochen Hipp, Martin Haueis, Maximilian Trepte, Carsten Brenk, Andreas Tamke, Mohammad Ghanaat, Markus Braun, Armin Joos, Hans Fritz, Horst Mock, Martin Hein, and Eberhard Zeeb, "*Making Bertha Drive— An Autonomous Journey on a Historic Route*", IEEE Intelligent Transportation Systems Magazine, Volume 6, Issue 2, Summer 2014

[13] Abhiram Rahatgaonkar, "*Velocity planning approach for autonomous vehicles: An approach to address traffic negotiation for autonomous vehicles*", Master Thesis report, Chalmers University of Technology, 2015

[14] Tom Schouwenaars, "*Safe Trajectory Planning of Autonomous Vehicles*", PhD Thesis report, Massachusetts Institute of Technology, 2006

[15] SAE International, 2014. Taxonomy and definitions for terms related to on-road motor vehicle automated driving systems (J3016), website: http://standards.sae.org/j3016_201401/

[16] Sanford Friedenthal, Alan Moore, Rick Steiner, "*A practical guide to SysML: The Systems Modeling Language*", Morgan Kaufmann, 3rd edition, 2015

[17] Object Management Group (OMG) website: http://www.omgsysml.org/

[18] International Council of Systems Engineering (INCOSE) website: https://www.incose.org/

[19] Lenny Delligatti, "*SysML Distilled: A Brief Guide to Systems Modeling Language*", Addison-Wesley Professional, 1st edition, November 2013

[20] Jon Holt and Simon Perry, "*SysML for Systems Engineering: A Model-Based Approach*", The Institution of Engineering and Technology, 2nd Edition, November 2013

[21] Hans Blom, Henrik Lönn, Frank Hagl, Yiannis Papadopoulos, Mark-Oliver Reiser, Carl-Johan Sjöstedt, De-Jiu Chen, Ramin Tavakoli Kolagari, "*EAST-ADL – An Architecture Description Language for Automotive Software-Intensive Systems*", White Paper, Version 2.1.12

[22] BSD Viper 1/10 instructions and specifications manual website: https://www.hobbex.com/internt/artiklar/781434/781434_BSD_1-10_BL_Truck_BS909T.pdf

[23] Nikolaus Correll, "*Introduction to Autonomous Robots*", Magellan Scientific, 2nd edition, V1.7, 2016

[24] Dieter Fox, Wolfram Burgard, and Sebastian Thrun, "*The dynamic window approach to collision avoidance*", IEEE Robotics and Automation Magazine, Volume 4, Issue 1, 1997

[25] Brian P.Gerkey, Kurt Konolige, "*Planning and Control in Unstructured Terrain*", Workshop on Planning with Cost Maps, IEEE International Conference on Robotics and Automation, 2008

[26] Robot Operating System 2-dimensional costmap documentation website: http://wiki.ros.org/costmap_2d?distro=lunar

[27] Robot Operating System basic concepts documentation website: http://wiki.ros.org/ROS/Concepts

[28] Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo and James Diebel, "*Path Planning for Autonomous Vehicles in Unknown Semi-Structured Environments*", The International Journal of Robotics Research, 2010

[29] E. Velenis and P. Tsiotras, "*Optimal Velocity Profile Generation for given Acceleration Limits: The Half-Car Model Case*", IEEE International Symposium on Industrial Electronics, 2005

[30] Marilyn Wolf, "*Computers as Components: Principles of Embedded Computing System Design*", Morgan Kaufmann, 4th Edition, 2017

[31] Joseph Sifakis, "*System Design Automation: Challenges and Limitations*", Proceedings of the IEEE, Volume 103, Issue 11, pages 2093-2103, November 2015

[32] Jeff Estefan, "MBSE Methodology Survey", INSIGHT, Volume 12, Issue 4, pages 16-18, December 2009

TRITA TRITA-EECS-EX-2018:724