Runtime Restriction of the Operational Design Domain: A Safety Concept for Automated Vehicles

by

Ian Colwell

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2018

Abstract

Automated vehicles need to operate safely in a wide range of environments and hazards. The complex systems that make up an automated vehicle must also ensure safety in the event of system failures. This thesis proposes an approach and architectural design for achieving maximum functionality in the case of system failures. The Operational Design Domain (ODD) defines the domain over which the automated vehicle can operate safely. We propose modifying a runtime representation of the ODD based on current system capabilities. This enables the system to react with context-appropriate responses depending on the remaining degraded functionality. In addition to proposing an architectural design, we have implemented the approach to prove its viability. An analysis of the approach also highlights the strengths and weaknesses of the approach and how best to apply it. The proof of concept has shown promising directions for future work and moved our automated vehicle research platform closer to achieving level 4 automation.

A ROS-based architecture extraction tool is also presented. This tool helped guide the architectural development and integration of the automated vehicle research platform in use at the University of Waterloo, and improve the visibility of safety and testing procedures for the team.

Acknowledgements

I would like to thank my supervisors, Krzysztof Czarnecki and Steven Waslander, for the wonderful learning opportunity provided under their guidance. I would also like to thank the members of the Waterloo Intelligent Systems Engineering Laboratory (WISELab), Waterloo Autonomous Vehicles Laboratory (WAVELab), and the entire Autonomoose team for an exceptional collaborative environment to build awesome things together. It has truly been an amazing couple of years working with everyone.

Dedication

I dedicate this to my parents, Jeff and Janet, who have always been there to support me, regardless of what I chose to do or the outcome of it.

Table of Contents

List of Tables

List of Figures

Abbreviations

ADS	Automated Driving System
API	Application Programming Interface
DDT	Dynamic Driving Task
ODD	Operational Design Domain
PNG	Portable Network Graphics
ROD	Restricted Operational Domain
SAE	Society of Automotive Engineers
SVG	Scalable Vector Graphics

Chapter 1

Introduction

Due to the inherent difficulty of solving the “self-driving car problem”, an Automated Driving System (ADS) requires complex software and hardware system architectures [18][33]. These systems are expected to operate safely even in the event of system failures or hazardous external conditions such as poor weather. An ADS must be able to achieve a minimal risk condition (such as pulling to the side of the road) if it detects any issues with its own functionality or external conditions that prevent further safe operation. This thesis proposes a safety concept and architectural design that integrates functional degradation and functional boundary monitoring to maintain a runtime representation of the functional boundary based on current system capabilities.

The latest recommended practice from the Society of Automotive Engineers (SAE) [5] presents the need to monitor the Operational Design Domain (ODD) at runtime. The purpose of ODD monitoring is to determine whether or not the ADS is in a situation that it was designed to handle safely. The ODD can also be described as the “functional system boundary” since a domain is defined by its boundary. For example, if the ODD contains only unsignalized intersections, the ADS must not enter intersections controlled by traffic lights. If ODD monitoring determines that the ADS has violated, or is about to violate, the bounds of the ODD, then it is expected to initiate a Dynamic Driving Task (DDT) fallback in order to achieve a minimal risk condition. A DDT fallback for a level 3 ADS is human intervention and a DDT fallback for a level 4 or 5 ADS is executed by the ADS itself. For example, if the side-facing long range sensing (long-range radar) required to turn onto a major road from a minor road (T intersection) fails before the intersection, the vehicle must not enter it and may need to pull over. There exists work on defining and monitoring the functional system boundary [15] [36] (we will use the term ODD in the sequel) and work on graceful degradation approaches [26] [25] for handling failures. However, these existing works do not address how the degraded functionality of the ADS affects the ODD or how the ODD monitoring system should handle degraded operation. This thesis contributes the concept of *restricting the ODD based on current system capabilities.* This concept is called the Restricted Operational Domain (ROD), since it represents a restricted version of the ODD in which the ADS can still operate safely. The structured representation of the ROD allows the ADS to perform runtime monitoring of the ROD, and evaluate the domain of safe operation during changing system capabilities or faults. Continuing the T intersection example: if the right side facing long-range radar fails, the ODD is restricted to exclude left turn and straight through movements at T intersections since the ADS would have insufficient detection range for vehicles approaching from the right. In this degradation scenario a right turn may still be allowed, since approaching vehicles from the front or left can be detected and the remaining right side short-range sensing will detect pedestrians and other obstacles. This restriction allows the ADS to either continue safely and re-plan a route that avoids violating the ROD, or to perform a DDT fallback to a minimal risk condition if an intersection can not be avoided.

The main contributions of this thesis are the concept of restricting the ODD at runtime based on current system capabilities and providing an architectural design that integrates both ROD monitoring and graceful degradation. Among other benefits, this integrated architectural design allows the vehicle to continue to operate within a safe domain and monitor the boundary of the safe domain during changing system capabilities.

In addition to introducing the ROD concept, this thesis presents a brief overview of the system architecture of the "Autonomoose" research platform in use at the University of Waterloo. The ROD concept was applied to the Autonomoose platform as a proof of concept. A ROS-based architecture diagramming and documentation tool is also introduced as part of this thesis. The tool was created to easily capture and document the architecture of large ROS-based systems. This tool helped guide the development of the Autonomoose architecture and generate architecture documentation directly from the implementation. It was also useful in verifying and documenting the Autonomoose-specific safety and validation procedures used by the team.

Chapter 2 explains the purpose of the ODD and reviews the importance of ODD monitoring. A summary of existing work related to fault-tolerant and graceful degradation methods for automated vehicles is also presented in Chapter 2. Chapter 3 introduces the key concepts behind the approach and explains the application of the approach with a detailed example. Chapter 4 presents an architectural design to apply the approach on an automated vehicle. Before presenting the implementation of the ROD approach on the Autonomoose research platform, an overview of the Autonomoose architecture is presented in Chapter 5. In Chapter 6, we present implementation details of the proposed architecture, and share the experiences and results of applying the architectural design to an automated driving research vehicle as a proof of concept. The unique challenges associated with an ADS undergoing continuous development is also discussed in Section 6.3. Finally, a discussion in Chapter 8 analyses the approach to determine its strengths, weaknesses, and where best to apply it. A review of remaining challenges and future work is also included in Chapter 8. The ROS architecture extraction and documentation tool used during the development of the Autonomoose software architecture is introduced and discussed in Chapter 7.

Chapter 2

Background and Related Work

The computing systems inside an ADS have already approached a level of complexity that warrants the application of autonomic computing principles [12], such as self-awareness and self-adaptation [31],[24]. This thesis addresses part of this need for autonomic computing principles by proposing an integrated concept for self-awareness and self-adaptation that maintains a runtime representation of its potentially restricted ODD and adapts its behaviour to remain within this domain. The following subsections summarize the definition and monitoring of the ODD as well as the related works regarding fault tolerance methods for automated vehicles.

2.1 Operational Design Domain

SAE J3016 defines the ODD as "The specific conditions under which a given driving automation system or feature thereof is designed to function, including, but not limited to, driving modes." [5]. Examples of conditions include geographical areas, road types, traffic conditions, and maximum speed of the subject vehicle. The ODD can be seen as an ontology of a portion of reality that is associated with driving. The main purpose of the ODD is to precisely specify the domain in which the ADS can *safely* perform the DDT. In general, the ODD is useful for the following tasks:

- **Design Process**: Defining the ODD helps identify what scenarios the ADS must be designed to handle. System-wide and functional requirements can then be defined alongside the ODD.

- **Testing and Verification**: The ODD can be sampled to generate test cases with varying levels of detail for unit testing or integration testing via simulation.
- **Online Monitoring**: The ODD can be instantiated as a runtime object to be measured and validated during operation. This is also known as functional boundary monitoring [15].

The work of Geyer et al. [14] provides a structured ontology for generating test and use-case catalogues that consist of scenario, situation, and scene descriptions. Their work approaches the problem from a testing and verification perspective, but the ontology they present serves the same purpose as an ODD. Ulbrich et al. go into more detail by defining and substantiating the terms scene, situation, and scenario. They define a scene as containing scenery (lane network and stationary objects), dynamic elements (vehicles, pedestrians, etc.), and self-representations of actors and observers (behaviours and abilities). While there are many different ways to organize an ODD ontology (depending on the use cases), we will be utilizing the detailed ontology provided by K. Czarnecki's "Operational World Model" [11] throughout this thesis. Our interpretation of an ODD ontology is very similar to the scene portion of the ontology proposed by Ulbrich et al., except that we also include a section for describing the behaviour of the subject vehicle itself.

1. **Road Structure:** This category is for completely describing the static road environment. Details such as road geometry, traffic control devices, signage, roadside structures, etc. are included.
2. **Road Users:** Road users consist of vehicles, humans, and combinations thereof. Vehicles such as cars, trains, bicycles, etc. all fall under this category. The behaviour of these road users is also described in this category.
3. **Animals:** Supervised or unsupervised animals and their behaviour. An example of a supervised animal is a dog on a leash and an unsupervised animal is a deer.
4. **Other Obstacles:** This category covers all other obstacles that might be found on a roadway. Some examples are gravel or loose material from infrastructure decay, natural objects such as tree branches or leaves, and any other possible object such as lost cargo.
5. **Environmental Conditions:** Atmospheric, lighting, and road surface conditions are all part of this category. These conditions affect the visibility of sensors and the traction of the vehicle on the road surface among others.

6. **Subject Vehicle Behaviour:** Functionality and limits of the subject vehicle such as possible manouevres, speed limits, etc. Depending on the level of detail needed, the behaviour may be broken down into specific manouevre types or simply bounded by higher-level constraints such as vehicle speed or ability to reverse, etc.

2.2 ODD Monitoring

SAE levels 1 to 4 of driving automation define an ODD for a limited range of ADS functionality. While the SAE definition for level 5 automation specifies an "unlimited ODD", a level 5 ADS must offer the same mobility that an average human driver can provide. For the purposes of this thesis, we assume that level 5 automation still requires some form of ODD definition and monitoring for determining when the ADS is outside of its designed functional boundary, which may also be beyond what a human driver could safely handle. An example of a level 5 ODD violation would be driving off of the road and through a vegetable field or encountering severe weather. Levels 3 and 4 require the system to monitor the ODD and respond to ODD violations by requesting manual control from a fallback-ready user (level 3) or by automatically performing the DDT fallback (level 4). The purpose of the DDT fallback is to achieve a "minimal risk condition" or "safe state", which depends on the situation [27]. An example of a DDT fallback to a minimal risk condition is a pullover manoeuvre to a stopped position on the side of the road. If an ADS is expected to detect whether it has left the ODD, then it must be able to monitor the ODD at runtime. Wittmann et al. highlighted the need for monitoring the functional boundary in order to ensure safe operation [36]. Their use of the term "functional boundary" is the same concept as the ODD since a domain can be defined in terms of its boundary. They specify a functional boundary in terms of five subspaces consisting of static environment, traffic dynamics, environmental conditions, state of the subject vehicle, and passenger actions. The work of Horwick et al. provides a robust architecture for performing functional boundary (ODD) monitoring and DDT fallback execution [15]. While these existing works clearly outline the need and use of ODD monitoring, system faults are treated as events that induce immediate DDT fallbacks if unhandled by redundancy. Our contribution is to introduce the ROD as a restriction of the ODD based on current system capabilities, so that the system can continue operating safely in the ROD or perform a DDT fallback in the case of ROD violations.

2.3 Fault Tolerance in Automated Vehicles

Typical safety-critical fault-tolerance strategies involve redundancy of system components, whether it be running processes, hardware components, communication pathways, or all of the above [17]. These techniques guarantee a high level of system dependability, but often come at a significant cost increase due to redundancy measures. While some level of redundancy will always be required for safety assurance, what if the ADS could continue to operate safely after suffering system faults?

Graceful degradation allows a system to continue operating with reduced functionality when parts of the system fail. While some robotic systems might not contain enough functionality or complexity to be able to degrade gracefully, automated vehicles are an excellent use case for graceful degradation strategies because:

- Automated vehicles use a diverse suite of sensors to perceive the environment. Sensor measurements are typically fused together, which provides an opportunity for the system to operate without every single sensor fully functioning at all times.
- They are expected to execute a wide range of functionality and operate in various environments with changing requirements (highway, urban, parking lots, rural, etc.).
- Automated vehicles are being developed by many competitive companies, which makes cost a major motivator in the design phase. If graceful degradation strategies can reduce the need to duplicate some hardware for redundancy reasons, this reduction could prove very profitable.
- In terms of safety, graceful degradation is essential for ensuring automated DDT fallbacks in the case of major system failures.

Several fault-tolerant safety and monitoring systems for automated vehicles exist.

The work by Reschka et al. on fault tolerance for automated vehicles presents a surveillance and safety system that influences vehicle control and driving manoeuvres in response to system failures [26]. Specific performance criteria (such as localization accuracy, sensor coverage, etc.) are maintained by surveilling the system. These performance criteria then lead to intentional degradation actions used to limit driving manoeuvres and adjust driving parameters.

Ability and skill graphs have been suggested as an approach for achieving graceful degradation in automated vehicles [25]. The runtime skill graph provides a detailed representation of the current abilities of the system, which can be used to re-configure the system or support decision making of the ADS. However, no relation is made to the functional boundary or ODD monitoring.

The existing works present ways to improve the safety of an ADS in the case of system faults. However, they do not address how system faults affect the ODD or how to integrate their approach with an ODD monitoring system.

Chapter 3

Proposed Approach

The main purpose of the proposed approach is to produce a runtime representation of the domain in which the ADS can currently operate safely, regardless of functional degradation. In order to achieve this, we must answer the question "How should the ODD be restricted if a given subsystem is degraded?". This question is addressed by creating a mapping between degraded subsystem functionality and ODD restrictions. The following is needed in order to create this mapping:

1. An ODD definition that can be monitored at runtime.
2. Predefined subsystem Degraded Operation Modes (DOM).

Item 1 will be addressed in the following section and item 2 will be discussed, after a detailed explanation of the ROD, in Section 3.4.

3.1 Defining the ODD

Ideally, the ODD should be defined at the very beginning of the design process. The initial ODD would begin from a high level perspective and become further detailed and refined as the design of the ADS progresses. For example, Figure 3.1 shows an ISO 26262 inspired design process and highlights phases where the ODD is continually refined [4]. An example of ODD refinement during the design process is starting with an ODD of "Driving within a specific city", and then refining it to the types of roads that exist within this city, or

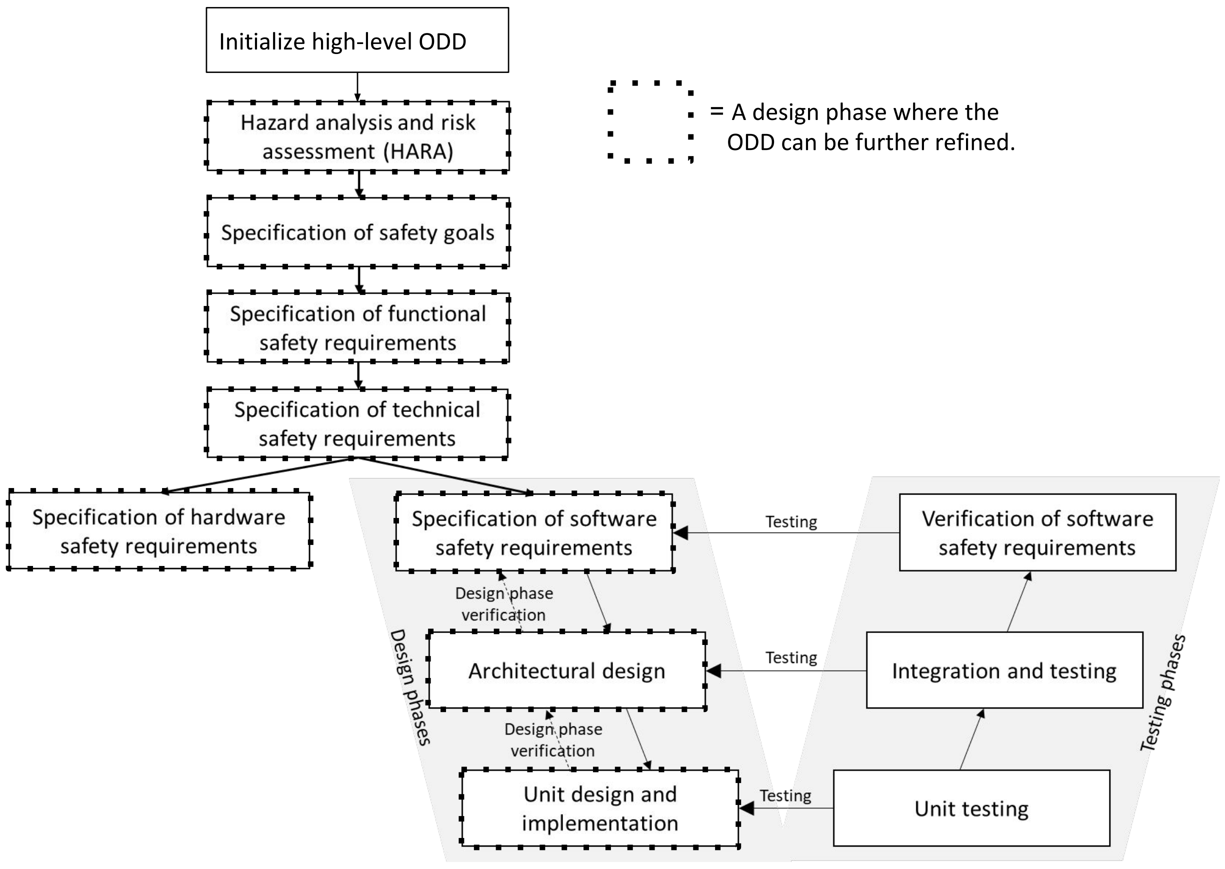

Initialize high-level ODD
Hazard analysis and risk assessment (HARA)
Specification of safety goals
Specification of functional safety requirements
Specification of technical safety requirements
Specification of hardware safety requirements
Specification of software safety requirements
Design phase verification
Architectural design
Design phase verification
Unit design and implementation
Design phases
Testing
Verification of software safety requirements
Testing
Integration and testing
Testing
Unit testing
Testing phases
= A design phase where the ODD can be further refined.

Figure 3.1: The integration of the ODD definition into the design process.

the precise geographical boundary of this city. In the end, the ODD should accurately represent the domain in which the ADS can safely perform the DDT.

The contents of the ODD also have an effect on ODD monitoring since monitoring is required for levels 3 and higher. This means the ODD defines not only the functional requirements of subsystem components responsible for performing the DDT, but also the functional requirements of the subsystem components responsible for ODD monitoring. For example, if the ODD is constrained to only clear weather, then the ADS must be responsible for detecting the current weather condition in order to determine whether or not the ADS is within the ODD.

3.2 Traceability Between Functional Requirements and the ODD Definition

In order to restrict the ODD in the case of degraded operation, it is essential to establish links between ODD elements and system functionality requirements or specifications. As soon as a rough functional architecture is defined in the design process, this linking between ODD elements and system functionality can be made. Ideally, this linking should be done using a top-down approach in parallel with the definition of the ODD as previously mentioned. However, an existing system could be analyzed in a bottom-up fashion to determine a safe ODD based on implemented functionality and components. An example of top-down would be: first defining that the ODD contains other road users, such as motorcycles, and then linking this ODD element to a functional requirement of the perception subsystem that requires the detection of motorcycles. An example of bottom-up would be analyzing the functionality of an existing perception subsystem (and possibly other subsystems) and determining that the ODD can safely include motorcycles, since the analyzed subsystems have the functionality to detect and interact with motorcycles. During a real design process, the ODD and linking to subsystem functionality will likely be established using both top-down and bottom-up approaches as the design is further refined. This combination of top-down and bottom-up aligns with the “V” portion of the design process shown in Figure 3.1, where the design is iteratively improved based on verification and testing results.

Regardless of the approach, it is key to establish and maintain traceability between subsystem specifications and system-level ODD elements. For example, a system-level ODD capability of “left turn at T intersection” would define what the perception system must be able to detect, as shown in Figure 3.2.

In the following section, we discuss how to use this traceability between subsystem functional requirements and ODD elements to restrict the ODD based on the degraded functionality of the ADS. This restricted ODD is a domain within which the degraded ADS can still function safely.

3.3 The Restricted Operational Domain (ROD)

We propose a new concept called ”Restricted Operational Domain” (ROD), which is defined as:

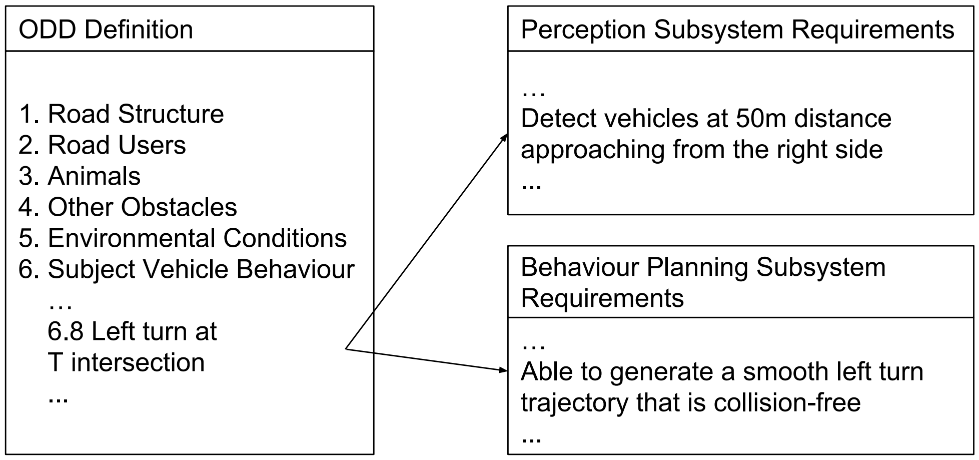

ODD Definition
1. Road Structure
2. Road Users
3. Animals
4. Other Obstacles
5. Environmental Conditions
6. Subject Vehicle Behaviour
…
6.8 Left turn at T intersection
...
Perception Subsystem Requirements
…
Detect vehicles at 50m distance approaching from the right side
...
Behaviour Planning Subsystem Requirements
…
Able to generate a smooth left turn trajectory that is collision-free
...

Figure 3.2: The mapping between ODD elements and high-level subsystem requirements.

> The specific conditions under which a given driving automation system or feature thereof is *currently able* to function, including, but not limited to, driving modes.

While the ODD is considered static throughout the operation of the ADS, the ROD may change given the degraded/restricted operation mode of the vehicle. In general the ODD can be seen as a set of constraints used to define the domain over which the system is designed to operate. The ROD constraints are a superset of the ODD constraints because more constraints are added to the domain based on the degraded functionality of the ADS. In terms of functionality, the ROD would allow a subset of the functionality that the ODD would allow as shown in Figure 3.3. When the system is fully functional the ROD is equivalent to the ODD.

Take the following failure case for example: A beam on the roof-mounted 360 degree LIDAR fails, creating a gap in the produced point cloud. The failed beam is in the middle of the range, at the height of other vehicles on the road. This failure affects three different subsystems that depend on the LIDAR data. The perception subsystem can not detect vehicles as far as it normally can, so the ROD constrains the subject vehicle speed so that the stopping distance is within the vehicle detection range. The localization subsystem is still able to perform scan matching, but the performance has been reduced and so the ROD constrains the speed of the subject vehicle to improve scan matches. This limits the type of roads to those that are under the constrained speed limit. The occupancy grid subsystem is not seriously affected as it can still create a representative grid based on the remaining beams. If more or different beams were to fail, the resulting restrictions would

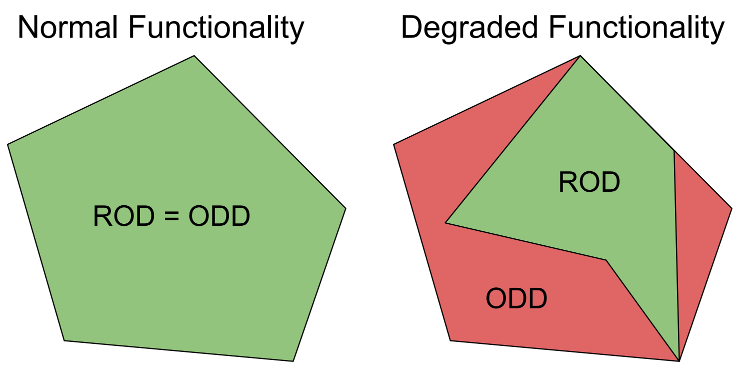


Figure 3.3: The Restricted Operational Domain.

be different depending on the unique functionality of each subsystem. For example, losing one of the lower beams may have no effect on vehicle detection or localization, but could severely limit the ability of the occupancy grid subsystem to detect small obstacles on the road, such as curbs. If the subject vehicle was travelling at high speed on a highway while the failure occurred, the DDT fallback would need to occur since the vehicle immediately exited its ROD (due to a violation of the speed constraint). If the vehicle was on a road with a minimum speed below the speed constraint, the system would continue and re-plan a route that stays within the ROD or, if no such route is possible, would have to prepare for exiting the ROD similarly to preparing to exit the ODD.

3.4 System Degradation and the ROD

In order for the ROD to represent what domain the ADS can currently handle safely, we require a monitoring system capable of detecting the current internal state of the ADS functionality. System failures and functional degradation must be identified and measured. We define the Degraded Operation Mode (DOM) as:

A sub-optimal functional operating mode of a system or subsystem.

A degraded operation mode (other than fully degraded) guarantees that some level of functionality remains. Fully degraded implies the subsystem is offline or failed. Some subsystems degrade in discrete modes, and some subsystems could degrade in a continuous fashion (such as confidence in calculated output data). A more detailed explanation of DOMs will be discussed in Section 3.4.1.

A *system impairment* is any effect on a component of the system that impacts the optimal performance of that component. Faults such as hardware or software component failures would be considered impairments. Other issues such as *confounding factors* are also considered impairments. Confounding factors are situations or events that complicate a task but are not considered faults. Examples of confounding factors are snowfall interference in a LIDAR scan or obstructions to cameras. System impairments lead to functional degradation unless they are already handled by traditional fault tolerance techniques (e.g, redundancy and compensation). In summary, degraded operation occurs when the system experiences some kind of impairment.

The DOMs of the subsystems are monitored at runtime and used to update the ROD. In order to do this, a mapping has to be created that links DOMs to the ROD constraints that must be applied to the ROD based on the occurrence of that DOM. This mapping ensures that the ROD always represents a safe domain based on current functionality, regardless of the amount of system impairments or combinations thereof. An example of this mapping would be constraining the set of manoeuvres to exclude left turns at T intersections from the ROD if the right side facing vehicle detection system were to fail, as shown in Figure 3.2.

The main benefit of keeping a runtime representation of the ROD is that the ROD can be monitored for violations the exact same way as the ODD. This means if a system impairment occurs, the degraded ADS may still be able to continue safe operation within the ROD. Also, since ROD monitoring is occurring, the system will be able to engage a DDT fallback in the case of system impairments that result in a ROD that is violated or about to be violated by the actual driving environment (indicating unsafe or imminently unsafe operation).

3.4.1 Degraded Operation Modes in Detail

As previously discussed, DOMs represent functional modes for a given system or subsystem. The following notation will be used to discuss more details about DOMs.

$$DOM_{ax} = \text{Degraded operation mode } x \text{ of subsystem } a$$

Each subsystem has at least two operation modes:

$$DOM_{a1} : \text{Fully degraded or failed}$$
$$DOM_{a0} : \text{Normal operation mode}$$

A DOM greater than 1 represents any other type of partial functionality that lies somewhere in between fully functional (DOM_{a0}) and fully degraded (DOM_{a1}). The numerical value is just an index and does not indicate increased degradation with increased value. A DOM might have a continuous value to it as well, such as degraded output quality or confidence in calculated data. This continuous value can be associated with any DOM other than 0 or 1. This value is expressed as:

$$val(DOM_{ax})$$

The DOM of a given subsystem may be dependent on another subsystem that could also degrade. For example, if subsystem A were to depend on the output of subsystem B and subsystem B became degraded, how should subsystem A respond? It is possible that subsystem A can not function without a fully functional subsystem B. If subsystem B were to become degraded, so should subsystem A. A subsystem could depend on multiple other subsystems all with their own DOMs. Each of these other subsystems provide different types of data that, when degraded, have different impacts on the resulting performance of the subsystem that depends on them. Logic expressions can be used to represent this interdependence between subsystems.

$$DOM_{ax} = DOM_{b1} \text{ OR } DOM_{c1}$$

DOM_{ax} is activated if either subsystem b or subsystem c is fully degraded. Another example is as follows:

$$DOM_{bx} = val(DOM_{d3}) > t$$
$$\text{where } t \text{ is some threshold}$$

DOM_{bx} is activated if DOM_{d3} has an associated value greater than some threshold.

3.5 Example

An example will be used in order to explain the application of the proposed approach. We will step through the process of designing an ADS using the ROD approach. The first part of the process is defining the ODD, in order to scope the requirements for the ADS. For simplicity, we will assume that the vehicle is only operating on low-speed (less than

50 km/h) roadways. We want our ADS to be able to handle unsignalized intersections (stop-sign intersections) that are either cross intersections or T intersections. Stop signs may exist on all intersection approaches (4-way, 3-way stops) or allow some approaches the right-of-way, which is typical when a minor road connects to a major road. The subject vehicle has a maximum of three manoeuvre options at any instance of this intersection as shown in Figure 3.4. In order to describe all the possible scenarios that our ADS could be exposed to, the following ODD is created as shown in Listing 3.1. The numbered categories match the categories described in K. Czarneckis "Operational World Model [11], with the addition of "subject vehicle behaviour" as the sixth category.

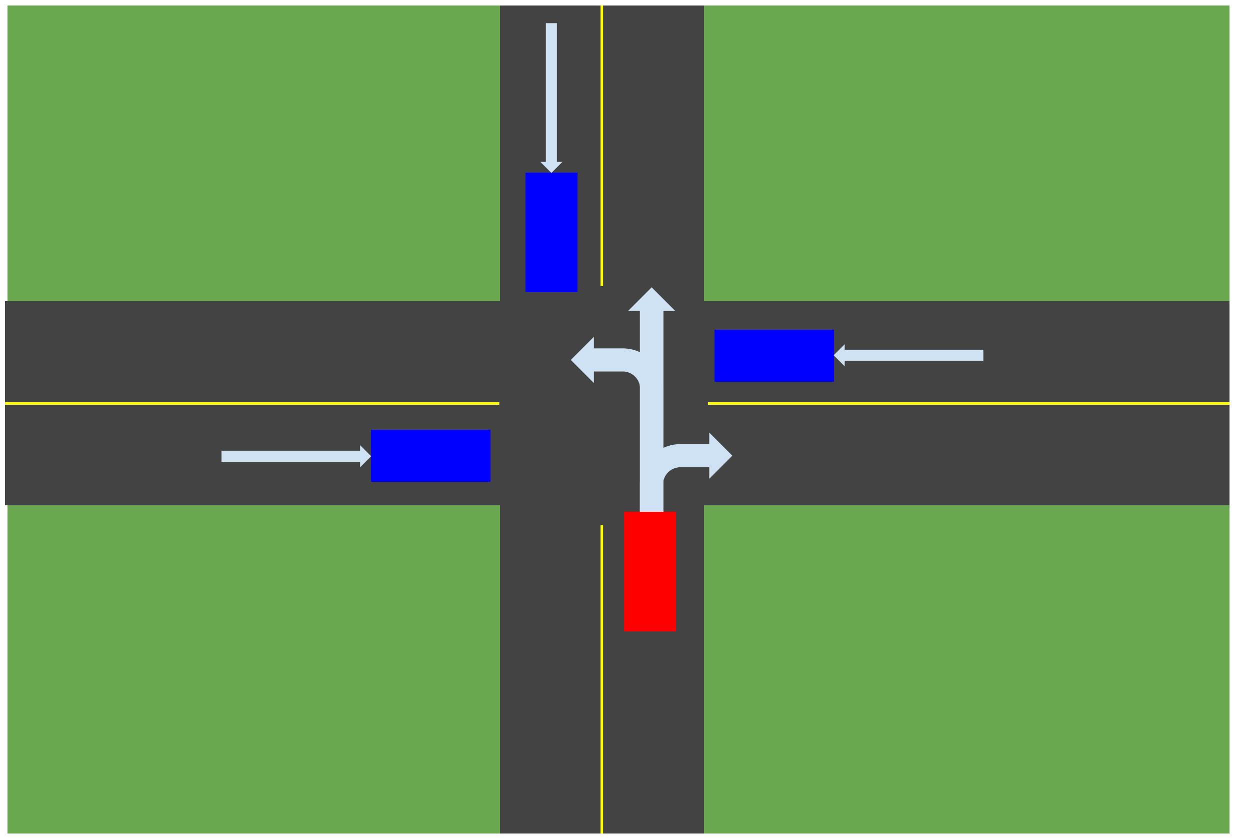


Figure 3.4: The intersection scenario with the three possible manoeuvres of the subject vehicle.

Listing 3.1: The ODD definition for an example ADS.

```
1. Road Structure
 1.8 Junctions
```

```
  1.8.1 Intersections
   - Types:
    - T-intersection, open-throat type
    - Cross intersection, open-throat type
   - Traffic Control:
    - STOP-sign controlled

6. Subject Vehicle Behaviour
 6.4 Intersection Manoeuvres
  - Left Turn
  - Straight through
  - Right Turn
```

Category 1.8.1 lists the types of intersections and traffic control devices that are allowed in the ODD. In category 6.4 we list the type of manoeuvres that the subject vehicle is allowed to take at intersections. Keep in mind that the above ODD elements are only a small portion of the overall ODD. Other elements that are not shown could include limits on weather conditions, the types of roads that are allowed (freeway vs. residential roadways), and roadway features such as pedestrian crossings or tunnels.

Now that we have defined the ODD in which our ADS is to operate, we can design a system to achieve safe operation in the specified ODD. Since it will be operating on public roads, our autonomous vehicle will have camera, LIDAR, and radar sensors to detect other road users all around the vehicle. Bussemaker's work [10] on sensing requirements is a useful starting point to determine what sensors are needed in order to safely execute certain driving objectives, and conversely, what actions can safely be performed given certain sensors have failed. Bussemaker's analysis for intersections suggests forward and side-facing sensor coverage in order to safely navigate intersections. This coverage ensures any other approaching vehicle will be detected before the subject vehicle pulls out into the intersection. Other approaching vehicles may be travelling at full speed since they may not have a stop sign and will proceed through the intersection without slowing. Therefore, when stopped at a stop sign, the ADS uses long range radars to determine if the way is clear of approaching vehicles before turning out onto the road. The radars are mounted at the front of the vehicle and are oriented to cover the area to each side of the vehicle as shown in Figure 3.5. The gaps in coverage closer to the vehicle are assumed to be covered by the camera and LIDAR sensors.

At this point in the design we can begin linking ODD elements to the subsystems present in the architecture. As previously mentioned, the side-facing radars are required for safely traversing intersections. Links will now be made between the radar subsystems and the corresponding ODD elements as shown in Figure 3.6.

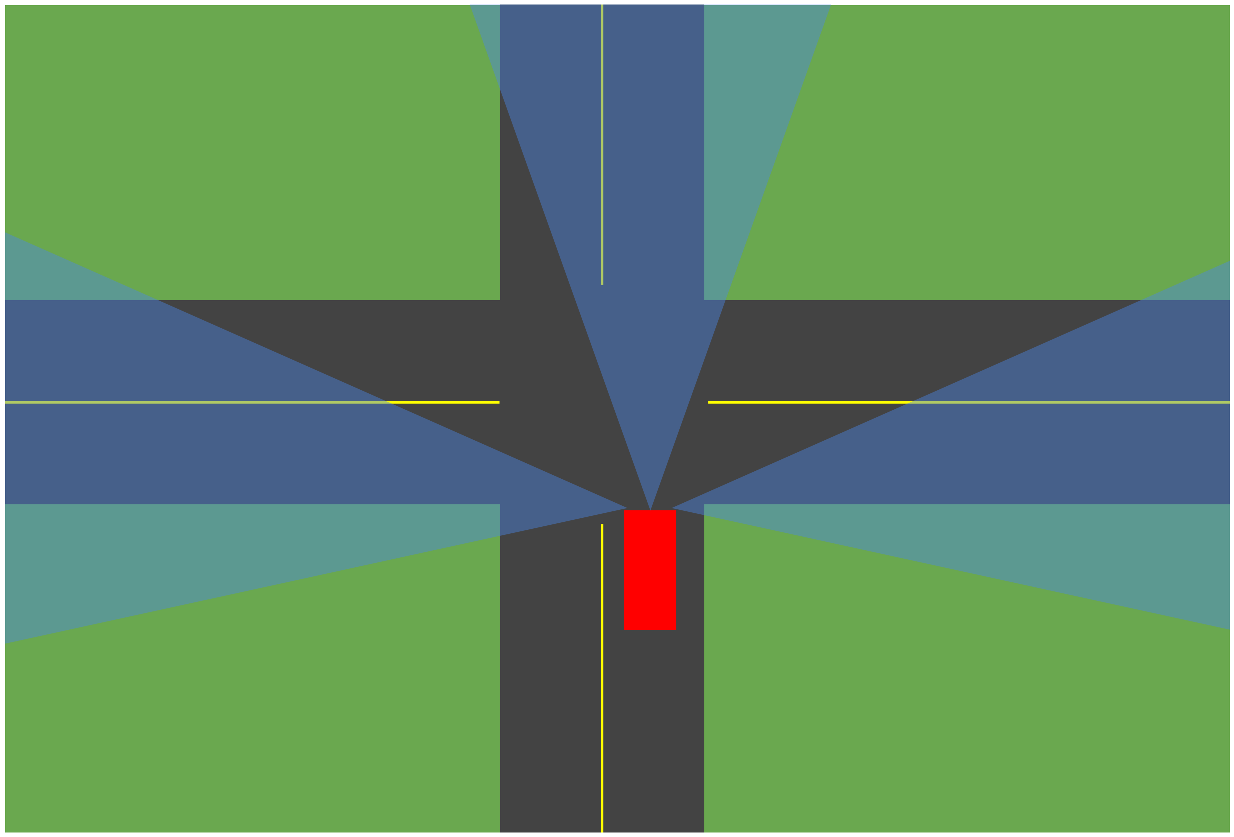


Figure 3.5: The coverage of the long range radars for detecting oncoming vehicles at intersections.

Notice how both “Left Turn” and “Straight Through” require all three radars, but “Right Turn” only requires the left and forward-facing radars. It is considered safe to turn right without the right-facing radar since the only possible obstructions to the path of the subject vehicle would be vehicles approaching from the left going straight, or vehicles approaching from the front turning left. It is still possible for other objects such as pedestrians to be in the path of the subject vehicle, but these objects are detected by shorter range cameras and LIDAR. It is assumed that vehicles approaching the intersection are not travelling on the opposite side of the road, which is possible during an overtaking manoeuvre. Now that it is clear what subsystem components enable what elements of the ODD, the linking between ODD elements and subsystem DOMs can be instantiated as runtime elements. First, we will define the DOMs for the radar subsystems as shown in Table 3.1. The DOMs of the radars are activated by monitoring the data stream produced

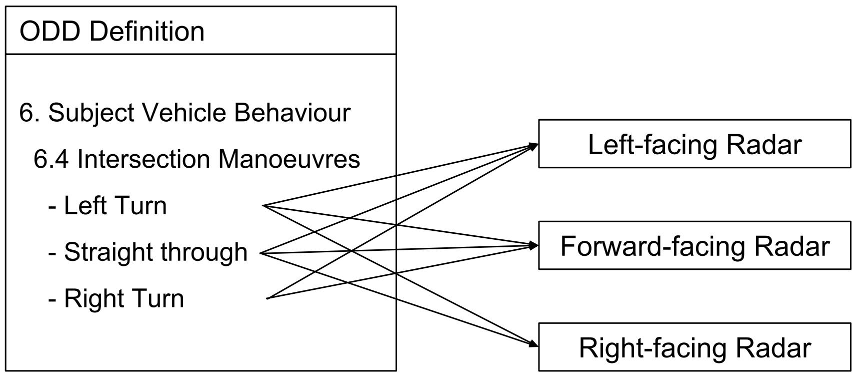


Figure 3.6: The mapping between the ODD elements and subsystems required to enable the element.

Table 3.1: The Degraded Operation Modes of the radar systems.

	DOM	
Radar	**0**	**1**
Left-Facing	Fully functional	Fully degraded/failed
Forward-Facing	Fully functional	Fully degraded/failed
Right-Facing	Fully functional	Fully degraded/failed

by the sensors. If the data were to stop, then the DOM would be updated to 1 to indicate a failure of the device.

Next, mappings between the DOMs of a given subsystem and the modifications to the ROD must be made. Often, multiple subsystem DOMs will modify similar elements of the ODD, so ROD modifications will be defined separately. There are two ROD modifications that should be made in this example:

- Modification 1: Remove the ability to perform left turn, straight through, and right turn manoeuvres at intersections (category 6.4). Also remove both "T" and cross intersections from the allowed types of junctions (category 1.8.1).
- Modification 2: Remove the ability to perform left turn and straight through manoeuvres at intersections (category 6.4).

When the modifications are activated, the ROD will be restricted according to the modifications. Finally, the DOMs are linked to the appropriate ROD modifications to achieve the desired behaviour. The DOM to ROD modification links are shown in Figure 3.7.

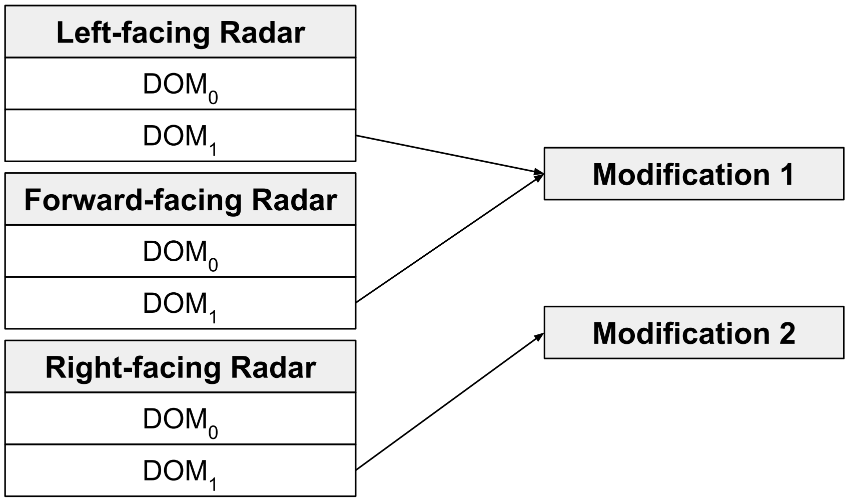


Figure 3.7: The mappings between the DOMs of the subsystems and the ROD modifications.

The ADS now has all the configurations needed to use the proposed approach. The following two failure cases will be examined to ensure the ROD will be properly restricted to a safe domain.

1. **Left-facing Radar Failure:** The failure of the left-facing radar is detected by a monitoring system and the DOM of the left-facing radar subsystem is updated to 1 according to Table 3.1. This DOM update triggers ROD modification 1 as shown in Figure 3.7. Modification 1 removes all the manoeuvre and intersection types from the ROD. If the subject vehicle was in or approaching an intersection, it will have to perform a DDT fallback since it has violated, or will soon violate, the ROD.

2. **Right-facing Radar Failure:** The failure of the right-facing radar is detected by a monitoring system and the DOM of the right-facing radar subsystem is updated to 1 according to Table 3.1. This DOM update triggers ROD modification 2 as shown in Figure 3.7. Modification 2 removes the left turn and straight through manoeuvres at intersections. The subject vehicle is still able to traverse intersections, but only perform right turns.

Restricting the ROD based on DOMs is only part of the final ADS design. While the ADS is now able to monitor the ROD and only perform DDT fallbacks when the ROD is violated, the planning subsystems need to be designed to react to changes in the ROD. In this case, a behavioural planning subsystem should be aware of the ROD elements in category 6 (which describe subject vehicle behaviours), in order to ensure future planned behaviour does not violate the ROD. A route planning system should also be aware of the allowed road structure in category 1, in order to plan a route over road types that are within the ROD.

Now that we have an idea of the steps and content required to realize the ROD approach, the following section will go into more detail regarding how the subsystems communicate all the DOM and ROD information.

Chapter 4

Proposed Architectural Design

A benefit of the ROD approach to system degradation/restriction is that we do not need to pre-define system-wide degradation modes and responses. We only need to define how a DOM in a subsystem maps to specific elements of the ROD. Once the ROD is updated, the subsystems are responsible for modifying their functionality to attempt to stay within the ROD. The subsystems need to be aware of ROD elements that are relevant to their operation and adjust accordingly. For example, the trajectory planner would need to know the ROD element that represents the subject vehicle maximum speed.

The proposed architectural design for enabling the ROD approach consists of a supervisory layer and a system layer, as shown in Figure 4.1. The system layer is responsible for executing the Dynamic Driving Task (DDT), and the supervisory layer is responsible for monitoring the system and responding to impairments by interacting with the system layer. The supervisory layer contains the core elements required to realize the ROD approach to system degradation. Each functional block of the supervisory layer can also be considered a subsystem on its own. A description of each of the subsystems in the supervisory layer follows.

4.1 System Supervisor

The System Supervisor maintains the current system state and manages requests for state changes. Some examples of system states would be “Manual”, “Automated Drive”, or “Emergency Pullover”. The System Supervisor is also responsible for coordinating the DDT fallback based on the level of remaining functionality. Here we assume that there

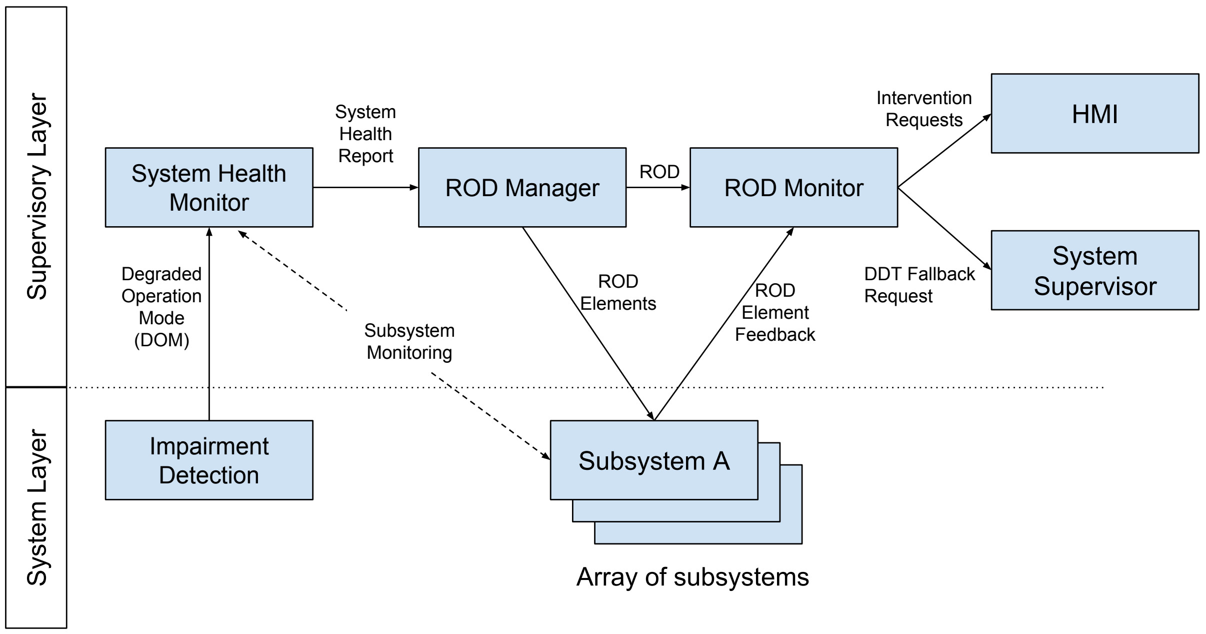


Figure 4.1: The proposed architectural design.

is redundancy in the system layer so that the ADS can always perform a DDT fallback. However, the design of this fallback system is outside the scope of this thesis.

4.2 System Health Monitor

The System Health Monitor is responsible for maintaining a complete representation of the current system health known as the *System Health Report*. The system health report is an amalgamation of all the DOMs of the subsystems. At least three different strategies can be used to obtain the current DOM for each subsystem:

- *Self-monitoring*: A subsystem is responsible for monitoring its own DOM and sending these updates to the System Health Monitor, which records it and combines it into the system health report.
- *Distributed Monitoring*: A stand-alone subsystem is designed specifically for monitoring and evaluating the current DOM of one or several other subsystems. In this case, the monitoring subsystem would report the DOM of one or more subsystems to the

System Health Monitor, while leaving the subsystem under observation completely free of any diagnostic functionality.

- *Centralized Monitoring*: The System Health Monitor is responsible for monitoring a subsystem and generating the subsystem's DOM all on its own.

It is important to note that a System Health Monitor in production would most likely use a combination of these strategies to maintain the system health report. For example, if self-monitoring is used and the subsystem itself fails then there would be no way for the System Health Monitor to know that the subsystem is offline. This problem can be avoided by having the System Health Monitor perform centralized monitoring in the form of simple heartbeat checking, while more detailed diagnostics can be obtained by distributed or self-monitoring.

The DOMs are assigned to subsystems at runtime based on appropriate metrics for a given DOM. These metrics need to be monitored at runtime in order to keep the DOM of the subsystems accurate. Some metrics are considered "raw" such as data frequency or data latency, and some metrics are calculated or derived such as a calculation of "visibility distance" or "obstruction percentage" from sensor data. The System Health Monitor is also where the logical relations between DOMs (described in Section 3.4.1) reside. When the System Health Monitor receives DOM updates or measures them itself, it uses the logical relations (if any are defined) to then update the DOMs of dependent subsystems for the system health report.

4.3 ROD Manager

The ROD Manager maintains the current ROD based on the system health report. The ROD Manager uses the mappings between subsystem DOMs and ROD elements (as mentioned in section 3.4) to keep the ROD up to date given any combination of subsystem DOMs.

4.4 ROD Monitor

The ROD Monitor is responsible for determining whether or not the ADS has exited the ROD. Naturally, it also serves as the ODD monitor. The ROD Monitor must obtain certain information about the system in real time in order to verify that the vehicle and environment state has not violated the ROD constraints or is about to violate the constraints.

The ROD Monitor is similar to the System Health Monitor in the sense that its functionality could be spread out using self-monitoring, distributed monitoring, or central monitoring. While distributed or central monitoring makes the most sense for reliability reasons (a subsystem performing self-monitoring could itself fail, which could leave some ROD violations unmonitored), self-monitoring could be applied to give additional and more detailed feedback for unavoidable upcoming ROD violations. For example if the maximum subject vehicle speed was limited to 60 km/h, the planning subsystem/s could detect that the planned route eventually traverses high speed roads that will violate the ROD. The ADS can use this advanced ROD violation information to plan a new route, plan an optimal automated DTT fallback, or at the very least give advanced warning for smoother transition to a manual DDT fallback or the eventual minimal risk condition.

If the ROD Monitor detects an immediate violation, then it notifies the System Supervisor to initiate a DDT fallback. If an unavoidable upcoming ROD violation is detected by the ROD Monitor, then an intervention request is made to the fallback-ready user via the HMI systems as per SAE level 3 [5]. Unanswered intervention requests forces the ADS to perform the DDT fallback automatically (SAE levels 4 and 5).

4.5 Subsystem

In the proposed architecture any given subsystem in the system layer could take a few forms:

1. Does not do any self diagnostics and simply performs a function. This means the System Health Monitor is responsible for detecting the DOM of the subsystem.

2. Performs self diagnostics, is aware of its own DOM, and simply reports the DOM to the System Health Monitor.

The subsystem may also need to be aware of any relevant ROD elements that affect its functionality. Subsystems may also perform some of the work of the ROD Monitor by checking if elements are violated or are about to be violated and notifying the ROD Monitor. For example, the vehicle control subsystem might subscribe to the ROD element that represents the subject vehicle speed limit, in order to ensure the vehicle stays within the limit.

Chapter 5

Overview of the Autonomoose Architecture

The system architecture of the Autonomoose automated driving research platform is presented as a contribution to existing works showcasing system architectures of automated vehicles [21], [7], [29], [22]. The ADS functionality is clearly separated from the vehicle platform functionality as suggested by [7] in order to minimize feature interaction and ensure the portability of the software onto other vehicles. The functional architecture of the ADS software is inspired by the architecture presented by Matthaei and Maurer [22].

Figure 5.1 shows the major functional blocks in the architecture and their classifications, and Figure 5.2 shows the connections between functional blocks.

The ADS software is broken down into four major categories: perception, planning, control, and supervision. The following sections will summarize each of these categories.

5.1 Perception

The perception subsystems of the architecture are responsible for measuring the external state of the world, the state of the ego vehicle, and combining them into a complete world model. The perception system is broken down into four subsystem categories as described in the following subsections.

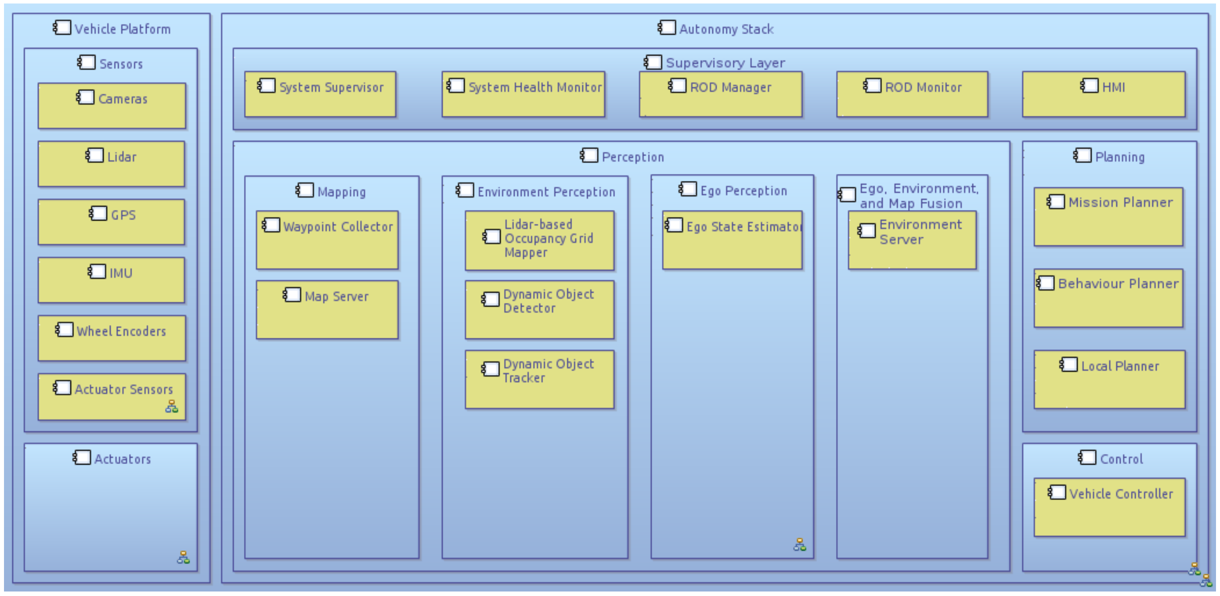


Figure 5.1: The classifications of the major functional blocks in the Autonomoose system.

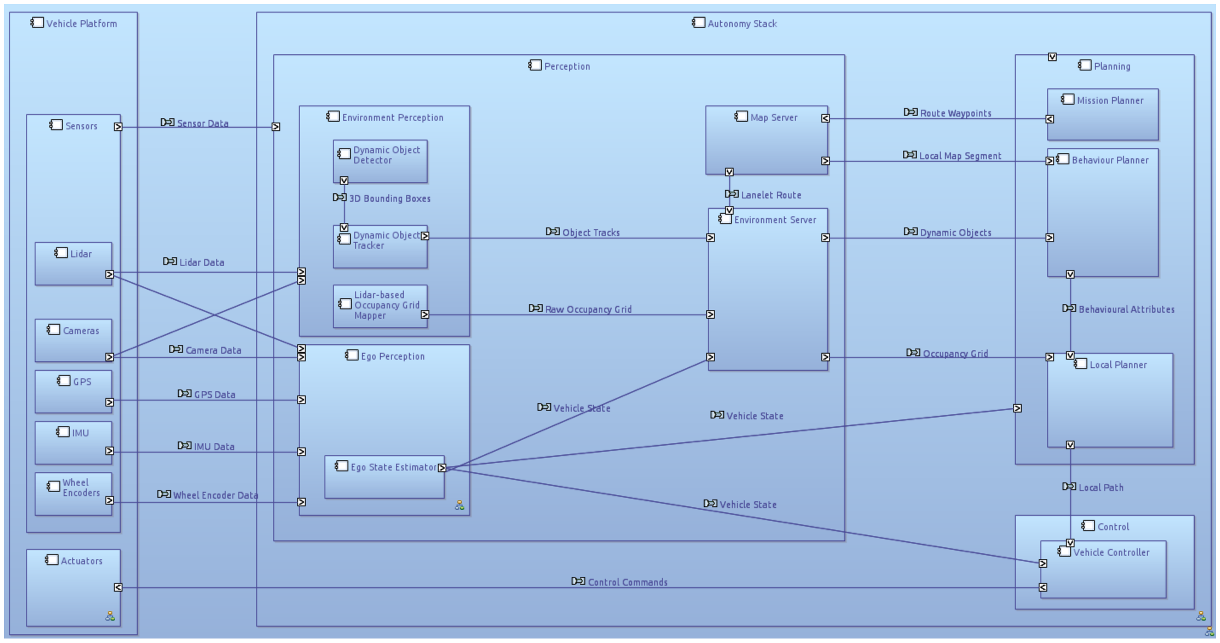


Figure 5.2: The connectivity of the major functional blocks.

5.1.1 Environment Perception

- **Dynamic Object Detector:** The Dynamic Object Detector detects pedestrians, vehicles, and bicycles using an Aggregate View Object Detection (AVOD) network [19]. LIDAR and RGB image data are inputs to the network, and 3D bounding boxes with orientation are produced.
- **Dynamic Object Tracker:** The Dynamic Object Tracker takes in the detected 3D bounding boxes from the Dynamic Object Detector and tracks their movement over time. A Kalman Filter is used to filter new detected bounding boxes with existing object tracks. The output of the tracker function is the same 3D bounding boxes with an additional speed estimations and tracks of previous positions.
- **Occupancy Grid Mapper:** The Occupancy Grid Mapper creates a probabilistic 2D occupancy grid using LIDAR data of the immediate area around the vehicle. This grid represents all static objects around the vehicle such as trees, shrubs, curbs, signs, bus shelters, etc. A height threshold is used to avoid false positives such as speed bumps and slight elevations in the roadway.

5.1.2 Ego Perception

The Ego State Estimator is responsible for determining the current location of the vehicle in the world. It consists of two major components:

- **LIDAR Scan Matcher:** The LIDAR Scan Matcher compares a measured pointcloud with a pre-existing pointcloud map. The measured pointcloud is edited to remove points landing on the ground using ground plane estimation. An area around the vehicle is also cropped to remove any points that may appear on the vehicle. Finally, the edited pointcloud is compared to the map using the Iterative Closest Point algorithm. The output of the LIDAR Scan Matcher is a pose (position and orientation) estimation in the frame of the pointcloud map.
- **Localizer:** The Localizer module fuses many sensor inputs to produce an estimation of the ego vehicle pose. The onboard GPS/INS system produces a direct pose estimation based off of the GPS and IMU sensors. This GPS/INS pose estimation is fused with all four wheel speed measurements and the Lidar Scan Matcher pose estimation using graph-based nonlinear optimization methods. The output of the localizer is a pose, linear velocity, and angular velocity state estimation.

5.1.3 Mapping

The mapping category is divided into the following subsystems.

- **Waypoint Collector:** In order to build accurate maps, GPS waypoints are first collected to indicate the centreline and left and right boundaries of lanes. The collected waypoints are then manually combined into the lanelet format as proposed by Bender et al. [8].
- **Map Server:** The map server provides a road network map in the form of lanelets. After receiving goal points from the Mission Planner, the Map Server will provide a route between the goal points in the form of lanelet IDs to traverse. A centreline path of waypoints is also produced to indicate where the ideal lane position is within the route for trajectory planning purposes.

5.1.4 Ego, Environment, and Map Fusion

The Environment Server fuses the information from environment perception, ego perception, and the map server. Tracked dynamic objects are converted from the camera frame to the odometry frame and assigned a lanelet ID according to the lanelet map. This makes the objects easier for the planning systems to work with.

5.2 Planning

The planning category is divided into the following subsystems:

- **Mission Planner:** The Mission Planner provides a series of GPS goal points to the Map Server to indicate the desired path of the vehicle.
- **Behaviour Planner:** Behaviour Planning predicts the future paths of detected dynamic objects in the environment and calculates a specific manoeuvre for the vehicle to execute. Example manoeuvres are “maintain speed”, “decelerate to stop”, or “yield to vehicles”. Theses manoeuvres are communicated to the Local Planner using attributes such as the manoeuvre type itself, stop locations, lane boundaries, and dynamic objects of interest, such as a lead vehicle.

- **Local Planner:** The Local Planner is responsible for planning a smooth trajectory between the current location of the vehicle and a future point along the desired path using bending energy optimization. Paths are evaluated for collisions with dynamic obstacles or occupied space on the static occupancy grid. The planner takes in the behavioural attributes from the Behaviour Planner, the static occupancy grid from the Occupancy Grid Mapper, and the vehicle state from the Localizer. The output of the Local Planner is a path for steering and a velocity profile along the path.

5.3 Control

Finally, the control category consists of the single Vehicle Controller subsystem. The Vehicle Controller takes in the local path and velocity profile from the Local Planner and executes it using a Stanley steering controller for lateral control [35] and a PID controller for longitudinal control. An integrated reverse controller is able to perform parallel and perpendicular parking as well. The outputs of the Vehicle Controller are throttle, brake, steering, and gear positions.

5.4 Supervisory Layer

The supervisory layer is explained in detail in Chapter 4.

Chapter 6

Realizing the Approach

This chapter discusses how the proposed architecture design was implemented and presents the results of a proof-of-concept scenario in Section 6.2. A discussion of some of the unique challenges related to an ADS undergoing continuous development is also presented in Section 6.3.

6.1 ROS Implementation

The proposed architecture design presented in Chapter 4 was implemented using the ROS framework [23]. A ROS node was created for each subsystem in the supervisory layer. The communication between the nodes is shown in Figure 6.1.

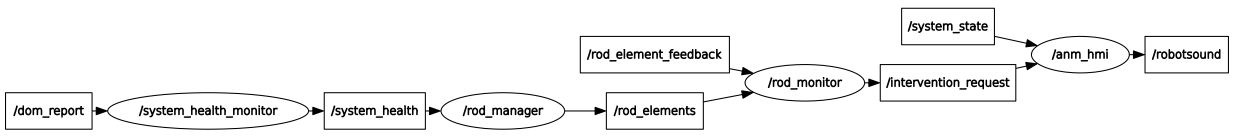


Figure 6.1: The ROS communication architecture between the supervisory layer subsystems. Ellipses represent ROS nodes and Rectangles represent ROS topics.

The ODD was stored in a hierarchical manner using JavaScript Object Notation (JSON) to be loaded by the ROD Manager at runtime. The JSON hierarchy matches that of the ODD categories described in Section 2.1. Each category has a name and either subcategories or values associated with the category. An example of the ODD format is shown in Listing 6.1.

Listing 6.1: The JSON structure used to define the ODD.

```
"1": {
  "name": "road structure",
  "1": {
    "name": "road type and capacity",
    "1": {
      "name": "road classification",
      "list": [
      "local road",
      "intersection"
      ]
    },
```

DOMs are represented using the ROS message defined in Listing 6.2. This message is passed from subsystems performing self-monitoring to the System Health Monitor. The message is also used in the system health report, which is simply a list of DOM messages published by the System Health Monitor. The fields of the message represent the different components of the DOM described in Section 3.4.1.

Listing 6.2: The DOM ROS message definition.

```
    Header header
    string subsystem_name
    uint32 DOM
    float32 value
```

The ROD Manager applies ROD modifications to the ROD based on the current DOMs of the subsystems that are obtained via the system health report. ROD modifications are also stored using the JSON format and an example modification is shown in Listing 6.3.

Listing 6.3: The JSON structure used to define ROD modifications.

```
"1": [
  {
    "odd_element_id": "1.1.1",
    "operation": "remove_list_element",
    "value": "intersection"
  },
  {
    "odd_element_id": "1.1.1",
    "operation": "remove_list_element",
    "value": "local road"
  }
],
```

For each ROD modification, some form of operation is made to an existing ODD element. In the above example, both the "intersection" and "local road" entries are removed from the ODD elements under category 1.1.1, which represent a list of allowed road types. Each ROD modification has a single ID but may contain more than one modification to the ROD. This is the case with the example shown in Listing 6.3, where the single modification with ID 1 performs two different ODD modifications.

Finally, the ROD modifications need to be activated by the DOMs of the subsystems. These mappings between subsystem DOMs and ROD modifications are implemented as "triggers" and are shown in Listing 6.4.

Listing 6.4: The JSON structure used to define ROD modification triggers.

```
"camera_LF": [
  {
    "evaluation_type": "eq",
    "dom_value": "1.0",
    "rod_modification_id": "2"
  }
]
```

The `evaluation_type` of `eq` indicates that the current DOM must be equivalent to `dom_value` in order to trigger the ROD modification identified by `rod_modification_id`. Other options such as `gt` and `lt` represent greater-than and less-than operators. Each subsystem has a list of mappings from the subsystem DOMs to ROD modifications. If a subsystem DOM is currently active, then the associated ROD modification is applied by the ROD Manager.

6.2 Proof of Concept

The ROD approach is implemented on the level 3 automated driving research platform currently in use at the University of Waterloo named the "Autonomoose". The existing platform depends on cameras and lidar for its perception of dynamic objects, therefore the goal is to make the system tolerant to camera impairments (both camera failures and camera obstructions) using the proposed architectural design.

A perception impairment monitor detects obstructions to the camera feeds used for perception. If the camera becomes partially or fully obstructed, then the DOM of the camera is reported as fully degraded to the System Health Monitor. This is an example of distributed monitoring, as previously mentioned in section 4.2, since a stand-alone subsystem (perception impairment monitor) is reporting the DOM of another subsystem (perception) to the System Health Monitor.

Figure 6.2 shows the system that exists on the Autonomoose research platform. It consists of the ROS-based supervisory layer as outlined in Section 6.1 and the newly created perception impairment monitor. The mission planner contains ROD element checking, so that it can respond to the changing ROD element that describes the types of roads that the ADS can currently traverse safely.

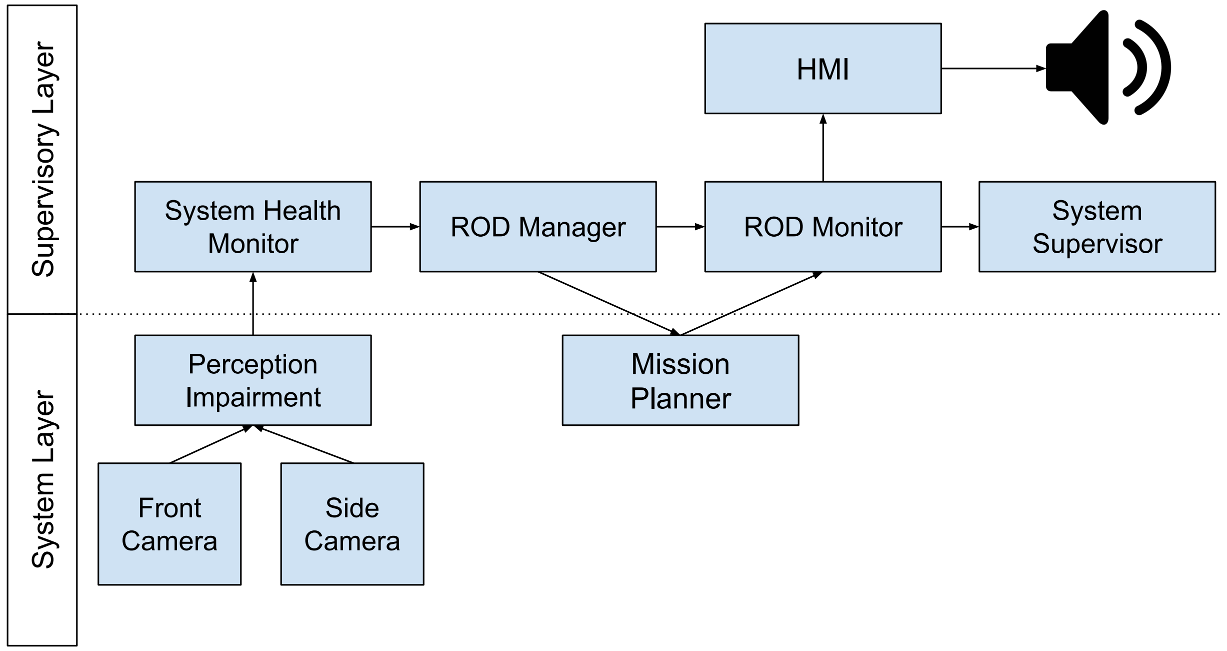


Figure 6.2: The implemented supervisory layer and impairment detection.

The implemented supervisory layer responds to camera impairments in either the front or side facing camera. This setup allows the system to intelligently react in a context-aware fashion to camera impairments. The functionality of the implemented supervisory layer is explained using the following impairment scenarios which are shown in Figure 6.3.

1. **Front facing camera impairment:** The perception impairment monitor detects the impairment and updates the DOM of the camera subsystem. The System Health Monitor receives the DOM and updates the system health report. The ROD Manager becomes aware of the degraded state of the camera subsystem via the system health report and triggers the appropriate ROD modification to update the ROD elements. In this case,

Table 6.1: Proof of concept results.

	Situation and ADS Response	
Failure Type	**Driving on road segment**	**Within or approaching an intersection**
Left-Facing Camera	Warning only	DDT Fallback
Forward-Facing Camera	DDT Fallback	DDT Fallback

the "road types" ROD element is modified to exclude all public roads, since the system requirements for safely traversing a public road can not be achieved without the front facing camera (and therefore forward perception) system. The mission planner determines that the car is currently traversing a road type that is not allowed and issues a ROD violated message to the ROD Monitor. The ROD Monitor issues a request to intervene via the HMI, and the fallback-ready user takes control of the vehicle completing the DDT fallback.

2. **Side facing camera impairment:** A side facing camera impairment is detected in the same way as the front facing camera impairment; however, the system response differs. Since the side facing camera is not absolutely necessary for lane keeping on an unobstructed two-lane road segment, the "road types" ROD element is updated to remove intersections from the allowed road types. Thus, the system response will differ depending on the location of the vehicle. In the case where the vehicle is currently on a public local road segment ("local road" meaning two-lane road, 50 km/h maximum), the system issues a warning to the driver that the camera is impaired, but the system is still able to perform the DDT safely. In the case where the vehicle is approaching or within an intersection, the system issues a request to intervene, since side-facing perception is required to safely navigate intersections.

The results of applying the proposed approach to the Autonomoose research platform are shown in Table 6.1. A video demonstration of the previously mentioned scenarios is available online [1]. This implementation is only a proof of concept that relies on intervention requests to a fallback-ready user; however, it paves the way for automated DDT fallback manoeuvres. Expanding the functionality of the supervisory layer and planning subsystems (to perform automated DDT fallbacks) will enable level 4 autonomy.

6.3 Challenges of an ADS in Development

Complex systems are typically created using a design process. System requirements and capabilities are defined up front by domain experts that are already aware of what requirements are in the realm of feasibility. A system is then iteratively designed and developed to meet those requirements. Subsystem components only exist to help achieve system-level requirements.

However, in the case of research-oriented projects, systems may develop for the purposes of achieving research objectives. In the case of our Autonomoose project, major subsystems are developed to test out a new research concept or algorithm. These subsystems often don't have formal requirements, specifications, or safety documents defined before beginning implementation of a subsystem. This presents a challenge for safety validation. How do we know the current system-wide capabilities and safe ODD of the system if the system was not designed with these concerns already in mind?

We approach this challenge in a bottom-up fashion. Rather than first defining the ODD in full, each subsystem is analysed to determine which areas of the ODD ontology are relevant to its operation. For example, our perception system is only currently able to detect and classify pedestrians, cyclists, and specific types of vehicles. The vehicle control subsystem is currently designed to go no faster than 50 km/h. An overall ODD is then compiled based on the capabilities of the various subsystems. No element of the ODD is included unless it is linked to a justification from a subsystem. This ensures the ODD is a conservative representation of what the vehicle can currently handle safely. Based on the existing perception and control subsystems, the compiled ODD would exclude roads with a minimum safe speed faster than 50 km/h, limit the subject vehicle behaviour to 50 km/h, and exclude all the types of dynamic object that the perception subsystem is unable to detect. As development continues and the functionality of the ADS is changed, the ODD can be recompiled to reflect the current state of the ADS. This ability to recompile the ODD based on the capabilities of the subsystems is also useful to systems that were created in a top-down design process. Modern software is rarely released and then left untouched. Bug fixes and feature updates are typically applied post release and this practice is currently observed in vehicles with ADAS features such as the Tesla product line [3]. ODD recompilation could be performed at each software update to ensure the vehicle will monitor an accurate and up-to-date ODD.

In the case of the Autonomoose project, one major value of the updated ODD is to help the safety driver understand the current abilities of the system. The Autonomoose does not have an extensive ODD monitoring system, therefore it is the role of the safety driver to perform the majority of ODD monitoring. The following chapter introduces a documentation tool that was created to help verify and explore the safety-related documentation of the Autonomoose project (in addition to its architecture diagramming capabilities).

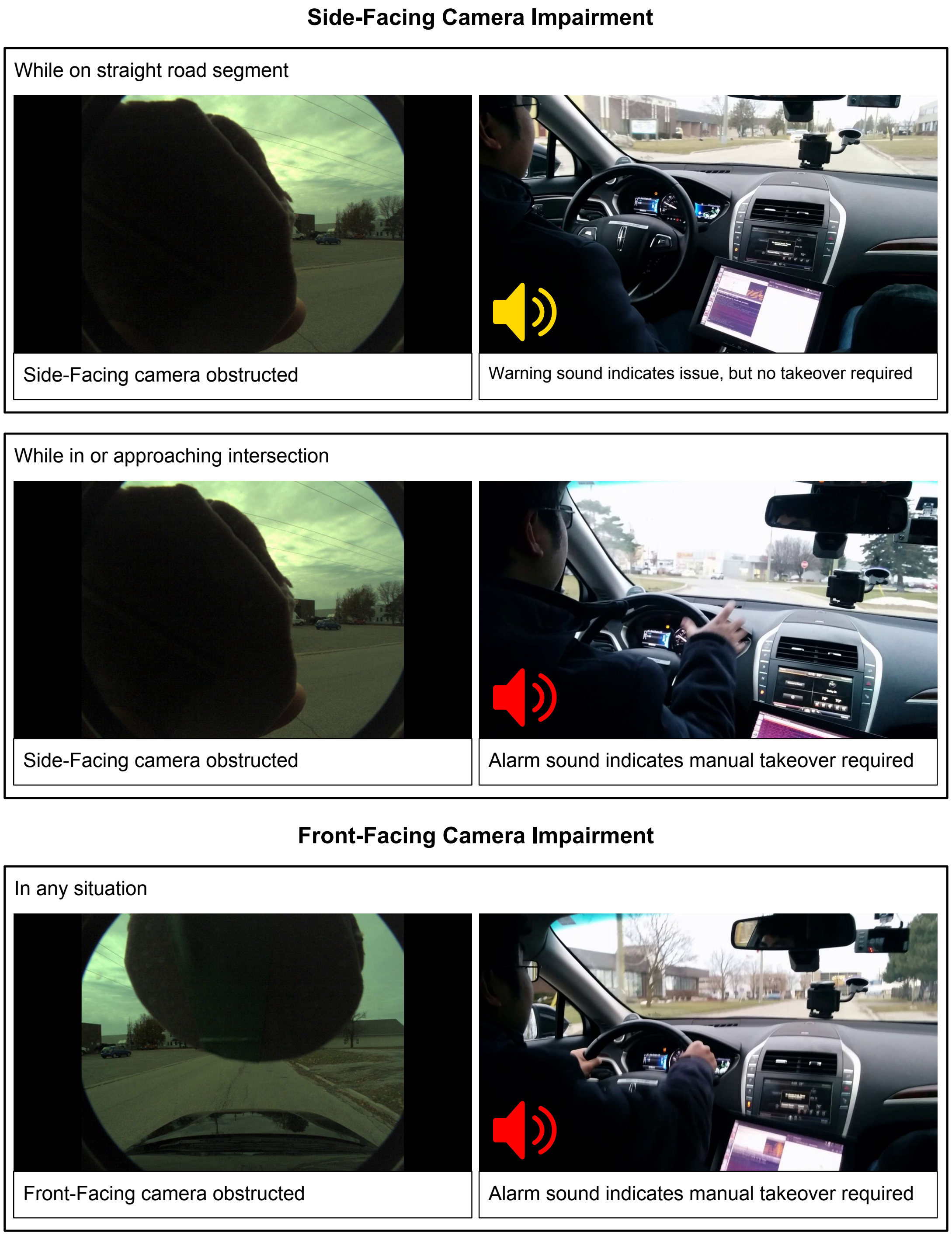

Side-Facing Camera Impairment
While on straight road segment
Side-Facing camera obstructed
Warning sound indicates issue, but no takeover required
While in or approaching intersection
Side-Facing camera obstructed
Alarm sound indicates manual takeover required
Front-Facing Camera Impairment
In any situation
Front-Facing camera obstructed
Alarm sound indicates manual takeover required

Figure 6.3: The proof of concept handling different failures in multiple scenarios.

Chapter 7

ROS Architecture Extraction Tool

This chapter introduces the ROS Database tool, which was created to help guide the architectural development of complex ROS-based systems. The tool was created to address several practical needs:

1. ROS-based architectures consist of nodes that primarily communicate via topics in a publish-subscribe architectural pattern. In order for nodes to communicate with each other, they must agree on the message types and topic names used. The team needed a way to quickly explore the ROS Application Programming Interface (API) of the many ROS nodes that exist in the system, in order to integrate new nodes and updates.

2. System-wide architecture diagrams slowly became difficult to keep up to date manually. It was a time consuming task that only became harder as the system grew in complexity. Ensuring that the manually-created architecture diagrams actually matched the implemented code was also a challenge. The need for automatically generating system architecture diagrams based on the existing codebase, and easily specifying future planned architectures, quickly became clear.

3. A ROS system can be launched in different configurations depending on the use case. Our ADS software was designed to operate on multiple different vehicles with different sensor suites. Therefore, the ability to automatically capture the system architecture of multiple different configurations was required.

4. The Autonomoose team needed to address the challenges discussed in Section 6.3. Namely, the ability to compile a system-wide ODD definition in a bottom-up fashion from subsystem documentation, and the ability to generate a safety summary

document for the safety driver that contains the most important safety-relevant information from each subsystem.

This Chapter reviews the existing solutions for ROS-based modelling and diagramming, and presents the ROSDB tool, our solution to capturing, modelling, and documentating ROS-based system architectures.

7.1 Existing Solutions

7.1.1 rqt_graph

An existing ROS tool called "rqt_graph" provides a graphical representation of the ROS system that is currently running on a user's machine [34]. The tool uses ROS information to populate graphviz [13] diagrams which are then displayed using the qt framework [2]. An example of one of these diagrams is shown in Figure 7.1. Ellipses represent nodes and rectangles represent topics.

The rqt_graph tool is very useful when debugging a system or trying to capture an overall diagram of the architecture. Diagrams can be exported as their dotcode source file (dotcode is used by graphviz to generate images), or as image formats such as Scalable Vector Graphics (SVG) or Portable Network Graphics (PNG). The rqt_graph tool addresses some of our diagramming needs but leave out details such as the message type being published to a topic, which package the nodes are related to, and the ability to create custom diagrams with nodes that do not yet exist. Arguably, one could create custom diagrams with the exported dotcode from rqt_graph, but the dotcode language is time consuming to edit manually.

7.1.2 Capella

Capella is an open-source model-based system engineering tool for guiding the execution of the Arcadia method [28]. Arcadia is a model-based engineering method for systems, hardware and software architectural design. While the application of the Arcadia method is outside the scope of this thesis (and the needs of our project team), the Capella tool is useful, with or without the use of Arcadia, as a diagramming tool. Since Capella is model-based, the system architecture is described using an overarching model. Diagrams are able to show different views into the system architecture model as needed. The main

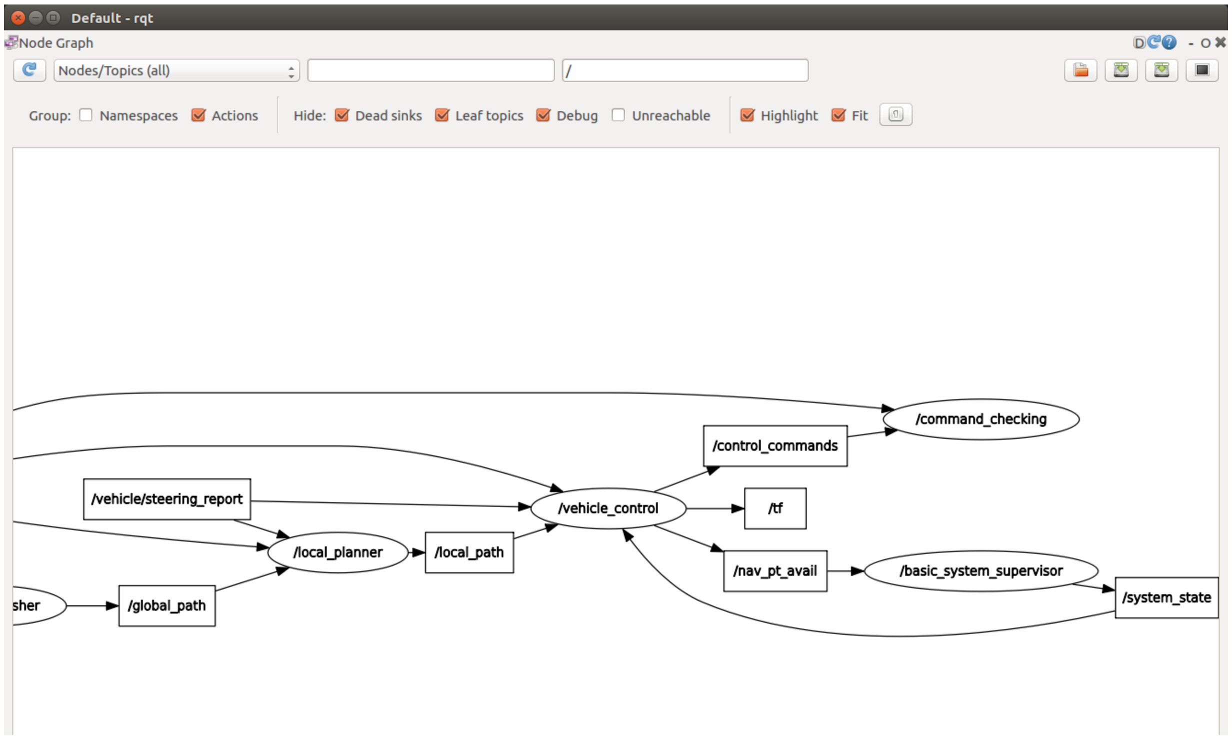


Figure 7.1: The rqt_graph user interface and an example diagram of a ROS system.

benefit of model-based diagrams is that changes to the underlying model are reflected on all diagrams automatically. Capella was used with some success for modelling the high-level functional architecture of the Autonomoose system and some of the hardware systems as shown in Figures 5.1 and 5.2.

The diagrams created in Capella served their purpose for guiding high-level architectural decisions and documenting the hardware architecture. However, lower level software details related to ROS concepts would have to be modelled manually, which was not ideal.

7.1.3 Model-Based Engineering Tools for ROS

There exist tools for performing model-based engineering with the ROS framework. Following is a brief overview of these existing tools.

BRIDE is a toolchain for framework-independent model-based development of robotic systems [9]. The tool is based in Eclipse and allows users to model their system and generate code from the model. BRIDE is ideal for teams that are in the design phase and have yet to write any code, because it helps guide them through the model-based engineering process. The ROSMOD tool is another option for modelling ROS systems and generating code from models [20]. Deployment and process monitoring are other useful features of this tool.

While these tools are powerful and useful for model-based engineering, they are unable to quickly import existing ROS packages and generate models of them. Since our primary concern was documenting the existing ROS architectures automatically, these modelling tools were not considered.

7.2 ROS Database (ROSDB)

The solution was to combine the benefits of model-based diagramming with the ROS integration of the rqt_graph tool. The idea is to be able to capture all the information from a running system into a model format, and then use that model to generate diagrams and documentation that describes the ROS system in detail. Models can be manually edited or created to produce documentation of future planned architecture designs, so that they can be analysed before implementation. The choice of term "database" in the name comes from the ability of the tool to store models of ROS packages in a central location to be used later in documentation generation. The following subsections go into the details of the stages shown in Figure 7.2, where models are captured, stored, edited if needed, and used to generate documentation.

7.2.1 Capturing ROS Data

The existing rqt_graph tool is able to generate a graph of the system architecture for any running ROS system using the rosgraph package, which uses a graph-based data structure to store ROS-related data. A populated rosgraph contains node names, topic names, and connections between nodes. This graph is traversed to capture all available information on the running system. Topic types (message types used for the topic) are captured using the rostopic API from the rostopic package. Determining the type of a running node, and the package that it belongs to, is not available from existing ROS APIs. A ROS node can be launched with a different name than the one it was compiled with, therefore the name of the node assigned at runtime can not be trusted to determine the type of the node. This problem was solved by retrieving the process ID of the running node using

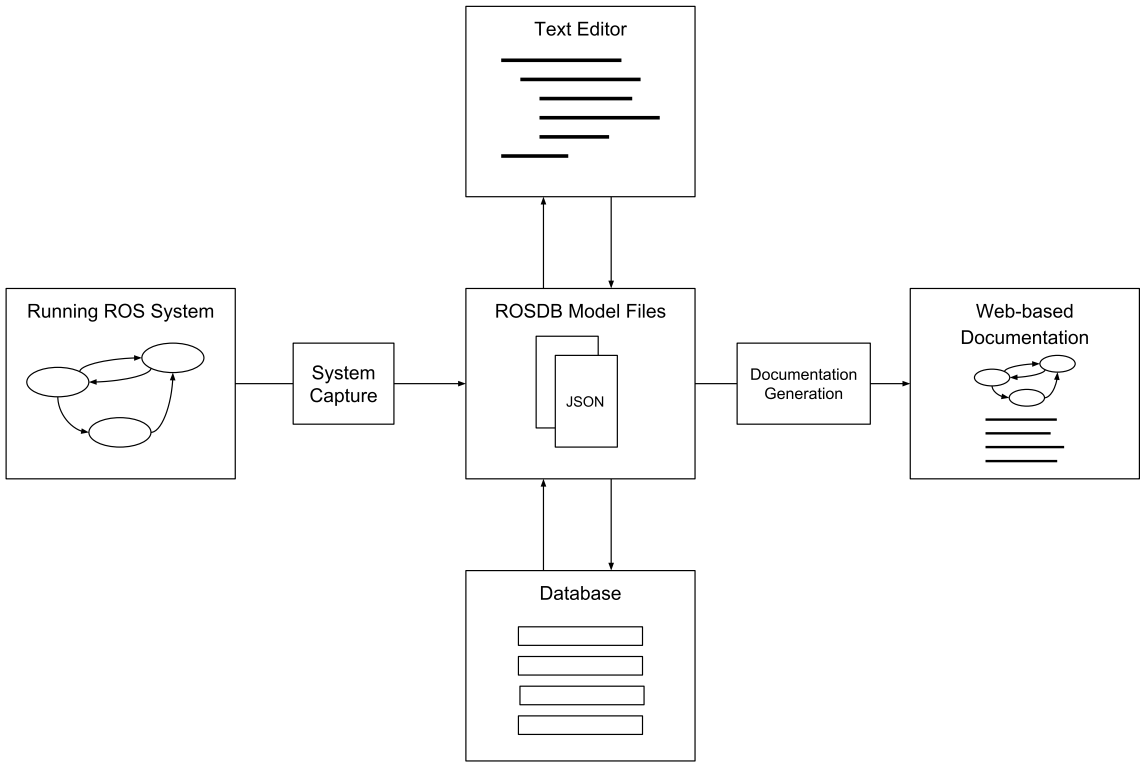


Figure 7.2: ROS information can either be captured or created manually, and then used to generate documentation.

an API from the rosnode package. The process ID reveals the command that was used to start the process which contains the full path to the compiled node. Fortunately, in the case of ROS packages, a compiled node binary resides within a folder named after the package name. Parsing the process command made it possible to determine the type of the running node and link which package it belonged to. Nodelets and nodes written in python complicated the parsing slightly as their process commands are different, but both were able to be properly linked to corresponding packages. Finally, package version information was obtained via the rospkg API. All this information is referred to as a "snapshot" of the system since it represents an instance of a running ROS system under a specific configuration.

A forked version of the rqt_graph tool was modified in order to easily save all this information to a file, and also load a file to view other captured systems. Figure 7.3 shows the modified rqt_graph tool and the added buttons for loading and saving this information. The following section discusses the captured information in more detail.

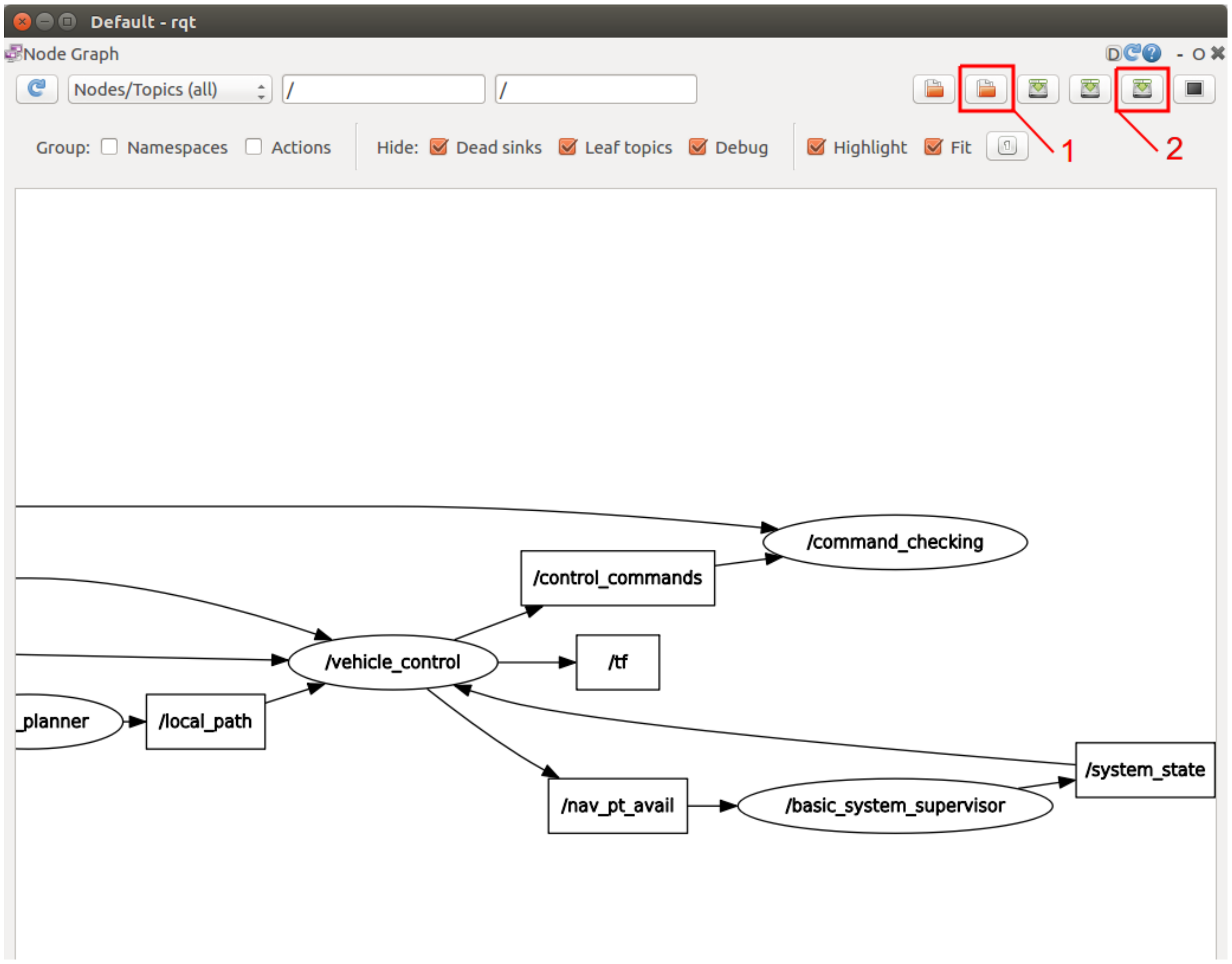


Figure 7.3: A screenshot of the rosdb rqt plugin with a rosdb file load button (1), and a rosdb file save button (2).

7.2.2 Storing the Data

A model structure was defined to contain all information necessary for describing a ROS system. The model is arranged in terms of ROS packages and the nodes or nodelets that they contain. An example of the information contained in the model is shown for the robot_state_publisher package in Listing 7.1. JSON was used as a storage format due to its hierarchical nature and relative ease of manual editing. These JSON files are referred to as “rosdb” files.

Listing 7.1: The captured rosdb file of the robot_state_publisher package.

```
{
  "packages": [
    {
      "name": "robot_state_publisher",
      "nodes": [
        {
          "name": "/robot_state_publisher",
          "node_type": "robot_state_publisher",
          "publications": [
            {
              "name": "/tf",
              "topic_type": "tf2_msgs/TFMessage"
            },
            {
              "name": "/tf_static",
              "topic_type": "tf2_msgs/TFMessage"
            }
          ],
          "subscriptions": [
            {
              "name": "/joint_states",
              "topic_type": "sensor_msgs/JointState"
            }
          ]
        }
      ],
      "parameters": [
        {
          "name": "/robot_state_publisher/example_param",
          "value": 3.0
        }
      ],
      "version": "1.13.4"
    }
  ]
}
```

A rosdb file may have multiple package descriptions in the same file (as is the case when capturing a snapshot of the whole system), or just a single package. The “database” aspect of the tool allows the user to register the contents of a rosdb file to a common database, which is simply a standard location on the user's computer. When registering the contents to the database, the rosdb file is divided up into each individual package and stored by package version number.

7.2.3 Generating Documentation

The captured data can be used to generate web-browser-based documentation for entire ROS system snapshots or single packages. The following pieces of information are generated when creating documentation for ROS system snapshot.

Overview Page: This page serves as the "home page" for the documentation and features an interactive diagram of the entire system. Clicking on ROS nodes in the diagram will bring the user to the documentation of the package to which that node belongs. A list of all the packages in the system is also provided with documentation status details, such as whether or not the package contains supporting documentation. All ROS parameters that were registered on the parameter server are also available for browsing in a list. Finally, an API spreadsheet table is available for browsing or downloading. The API table serves as a fast way to quickly view all the nodes in the system and what topics they publish or subscribe to. An example of the generated overview page is available in Appendix A.1.

Package Page: Each ROS package included in the snapshot documentation has its own page to present all the package information. A diagram showing all the package nodes and the topics they subscribe or publish to along with a listing of the topics and their messages types is included for each package. Other information that may also be shown is a readme, subsystem specification, and validation document. Our team requires each major subsystem to have certain documentation kept up-to-date for safety purposes. These documents and their uses are explained in Section 7.3. An example of a generated package page is available in Appendix A.2 for the local_planner package.

7.3 Autonomoose Safety Documentation

While the ROSDB tool was created to be useful for any ROS-based system, some custom functionality was implemented to serve the unique needs of the Autonomoose team. The team has defined software development and testing procedures in order to ensure that software running on the ADS has been sufficiently reviewed and validated. Since ROS-based systems are organized as a collection of ROS "packages" containing functional ROS "nodes", each major functional subsystem in the architecture has its own ROS package.

Therefore, terms “package” and “subsystem” are used interchangeably. Our team required each ROS package to have at least three documents that ensure the software testing and validation procedures are being performed:

1. **Readme:** As is the case with most software, this document is useful for providing an overview of the package functionality and how to install and run the software.

2. **Subsystem Specification:** This document is used to fully specify the functionality of the subsystem. It consists of sections describing the functional requirements, input/output requirements, any relevant configuration parameters, important safety driver considerations, dependencies on other subsystems, and finally: constraints to the ODD based on the functionality of the subsystem. Overall, the subsystem specification document should provide a complete picture of what the subsystem does, and how it affects the ODD or safety of the vehicle during operation.

3. **Validation:** The validation document describes the testing procedures for the subsystem, and a record of when the latest tests were performed and by who. Details for unit testing, integration/simulation testing, and in-vehicle testing are required so that someone unfamiliar with the subsystem is at least able to validate its functionality. An example of both the validation and subsystem specification documents are available in the generated package page in Appendix A.2.

The above documents provide detailed information for all subsystems. However, from a safety driver’s perspective, the most important information is the overall system ODD in order to know what the system is capable of performing safely, so that the driver can perform ODD monitoring and take over control when the ODD is violated. Any other relevant safety-related concerns from the subsystems should also be combined into an overall “Safety Driver Summary” for the safety driver to review before tests. Ideally, each safety driver would read all the above documentation for each subsystem, however this documentation is exhaustive and not all details are immediately relevant to the operation of the vehicle. The ROSDB documentation generation functionality was augmented to include this documentation so that it can be browsed easily and compiled into an overall ODD and safety driver summary. This functionality ensures that the safety driver is always able to read the latest version of this information and it reflects the functionality that is currently running on the vehicle.

7.4 Future Improvements

The ROSDB tool was able to satisfy the four practical needs of creating detailed ROS architecture diagrams and documentation, ensuring diagrams matched existing code, capturing entire ROS systems under different configurations, and compile summarized safety documents by combining information from all the subsystems. However, there are still improvements that can be made to the usability and automation features of the tool.

- **Complete ROS information capture:** There are other ROS-related architectural details that could be captured by the tool, such as ROS services and actions, the ROS transform tree, and any topic name re-mappings. Adding these details to the ROSDB model format would provide a complete model of all ROS-related architectural features for captured packages.
- **Analyze differences between models:** Since the models are text-based, the differences between two models can be determined relatively easily. A small amount of pre-processing would be needed to ensure the two models are arranged in the same order, such as alphabetically, so that a difference tool is comparing the same content at the same location in the files. This difference is useful to check whether an implemented ROS package, or system, matches its prescribed model. Mismatches found during an automated review could improve the system integration phase by helping keep the code implementation on track, or update the system architecture to necessary changes in the implementation.
- **Integration with static code analyser:** One downside to the tool is that a ROS system must be running in order to perform a system snapshot. While this may be unavoidable, and possibly desired when capturing system-wide configurations, it can be a drawback to capturing details of a single package. Integrating the tool with a static code analyser, such as HAROS [30], could allow the capture of ROS-related information without the need to launch the nodes.
- **Graphical editing of architecture model:** Currently the only way to manually edit a ROSDB model is using a text editor. The ability to create detailed models from scratch is not a need the tool was originally designed to address (only the ability to make small edits to existing snapshots, combine snapshots, etc., was originally desired). However, a graphical editor would allow the tool to serve as a system modelling tool to plan out ROS architectures before they are created. Since package snapshot information is stored in a database, existing nodes could be dropped in

from the database to help guide the integration of a complete system. Integration with other tools such as the previously mentioned BRIDE or ROSMOD tools should be considered as they already serve the purpose of system modelling.

Chapter 8

Discussion

8.1 Monitoring the ODD at Runtime

The ROD approach presented in this thesis depends on the ability to monitor the ODD at runtime. However, the ODD is complex and contains a variety of elements that may not be monitored easily. A review of the six-category ODD structure defined in Section 2.1 was performed to determine how much of the ODD can be feasibly monitored. The review assumes that existing publicly-showcased technology is utilized to build the ADS and ODD monitoring systems. The terms “difficult” and “easy” are used in the review. An ODD category or element being “difficult” to monitor is considered as “unable to monitor without significant increase in cost incurred by the longer design process or additional in-vehicle hardware resources required to accomplish the monitoring”. An ODD category or element being “easy” to monitor means there is very little impact to the overall design of the ADS and monitoring systems, and to the resulting hardware requirements of the monitoring systems. Following are the results of the review.

1. **Road Structure**: The majority of road structure details would be difficult to monitor accurately at runtime, with the exception of critical details such as lane markings, signs, or traffic lights. Semantic segmentation could be used to monitor some of these critical elements, but other details such as road geometries, intersection types and configurations, etc., are considered difficult to monitor with today’s technology. However, if the ADS has access to a map, the entire road structure category could be monitored depending on the level of detail and accuracy of the map. Measures would have to be taken to ensure the map data is sufficiently up-to-date and accurate.

Most modern ADS systems have extensively detailed maps making this category of the ODD easy to monitor at runtime. Overall, the ODD elements of the Road Structure category are well-defined making the design of their monitoring easy (compared to the following three categories which include a variety of objects that can take many shapes and forms).

2. **Road Users**: This category was determined to be difficult to monitor due to the variety of possible road users. Object detection and classification is currently an enormous challenge for automated vehicles and including it in a monitoring system is considered a non-trivial effort. Adding the ability to monitor this category of the ODD (in addition to the existing on-board perception systems), would require a significant increase in on-board computing resources. It is possible that some elements within this category could be monitored using semantic segmentation, which could provide the existence of elements in the environment without needing to position them accurately. However, the category as a whole is considered difficult to monitor due to the wide variety of road users that may not be well-defined.

3. **Animals**: Similar to road users, monitoring the existence of a variety of animals at runtime is considered difficult.

4. **Other Obstacles**: Again, monitoring the existence of other obstacles is considered difficult to do at runtime.

5. **Environmental Conditions**: This category is a mix of conditions that are between easy and more difficult to monitor. On the easier side would be conditions such as illumination levels and temperature. If the ADS has an active internet connection (which is a reasonable assumption with today's automated vehicles), then various high-level weather conditions become easy to monitor. Some road surface conditions such as surface friction or pavement temperature are considered difficult to monitor. Localized weather conditions such as visibility in fog or sudden precipitation events are also more difficult to monitor accurately.

6. **Subject Vehicle Behaviour**: The behaviour of the subject vehicle is considered the easiest category to monitor since it is simply a matter of monitoring the current vehicle state and future plans. This is typically something the ADS has to monitor in order to operate the vehicle, therefore access to this information is of no concern.

In summary (as shown in Table 8.1), assuming an ADS has a sufficiently detailed road/environment map and an internet connection for weather updates, an ADS is only

Table 8.1: The feasibility of monitoring certain categories of the ODD.

ODD Category	**Monitoring Feasibility**
1. Road Structure	Easy
2. Road Users	Difficult
3. Animals	Difficult
4. Other Obstacles	Difficult
5. Environmental Conditions	Mix
temperature, lighting conditions	Easy
surface friction, localized weather events	Difficult
6. Subject Vehicle Behaviour	Easy

capable of monitoring three of the six categories of the ODD without overly complex monitoring systems. The three categories are road structure, environmental conditions, and subject vehicle behaviour. This suggests that the ROD approach should be focused around these three categories since restrictions to the ODD are only useful if the ADS can monitor and respond to those restricted portions of the ODD. Consider the following two examples:

1. The ADS has multiple perception subsystems for detecting and classifying objects. One perception subsystem is for detecting vehicles and another is for detecting pedestrians. If the pedestrian detection subsystem were to fail, then the ROD should be restricted to exclude pedestrians because the ADS can no longer safely handle situations containing pedestrians.

2. The ADS has a subsystem dedicated to detecting the state of traffic lights. If this subsystem were to fail, then the ROD should be restricted to exclude intersections with traffic lights that do not also have V2X communications.

In both examples a perception subsystem failed; however, the restrictions to the ROD are different. In the first example, pedestrians are excluded from the ROD. Checking for the violation of the ROD element excluding pedestrians would require the ROD monitoring system to detect pedestrians. This creates a contradiction in design, if the ADS was still able to detect pedestrians (for monitoring purposes), then the ADS should use those detections and the ROD should not have been restricted in the first place. This design contradiction exists for most of categories two, three, and four that consist of objects in the environment that must be detected by the ADS. In the second example, traffic lights are excluded from the ROD. Assuming traffic lights are included in an on-board map, since they are static elements of the road structure, the monitoring system can easily determine where and when the ROD will be violated. The ROD monitoring system can traverse the planned route and check if traffic lights are encountered along the way.

Overall, the ROD approach is best applied to system degradation that results in ROD restrictions that consist of road structure elements, environmental conditions or subject vehicle behaviours. Additionally, in order to take full advantage of the ROD approach, a detailed on-board map should be used along with a robust planning subsystem. The ability of the route planning subsystems to re-plan routes in order to avoid violating the ROD or provide detailed warning for when and where it will be violated, is a main benefit of the ROD approach. These two aspects (detailed maps and flexible route planning) are particularly important as they allow the vehicle to stay on the road longer under various degradation situations.

8.2 Architectural Considerations

It is important to consider the architectural implications of the proposed supervisory layer. Consider the following example: The localization solution begins to deteriorate (perhaps due to a failed wheel-speed sensor or IMU) resulting in higher covariances in the state estimate. Should the localization subsystem enter a DOM in response to the poor localization solution, should the state covariance be passed along to other subsystems along with the state estimate, or both? If the localization subsystem were to enter a DOM, the ROD Manager could update the ROD in response to this degradation. The subsystems that depend on the localization data could then adjust their operation based on the restrictions in the ROD. For example, the controller might have to respond to a new speed limit restriction from the updated ROD. In this case, the response to this impairment occurs via the supervisory layer after the System Health Monitor updates the system health report and after the ROD Manager updates the ROD. This information path could be too slow depending on the criticality of the impairment. The other option of passing the state covariance along with the state estimate would result in a more direct path to the required response. The controller could immediately slow down based on the state covariance. However, if other subsystems need to be aware of current restrictions (such as speed limitations), then they would also need to subscribe to this state covariance information or to a speed limit from the controller. For example, the mission planner needs to know the current speed restriction in order to re-plan a route that satisfies this restriction. Without the supervisory layer and structured ROD, the system architecture could become complicated with large amounts of subsystem connections used only for coordinating responses to system restrictions. In the end, it is likely that a combination of architectural approaches is necessary. The subsystems that directly rely on state estimates can receive the covariance along with the estimate, and the supervisory layer can also update the ROD to allow other subsystems (such as the mission planner), that might not normally use the state estimate, to adjust to the restrictions of the ROD.

Using the Architecture Tradeoff Analysis Method (ATAM) [16] to evaluate an architecture that uses the proposed supervisory layer, such as the Autonomoose, is a direction for future work. ATAM could help clarify when to integrate the supervisory layer with the functional subsystems, and when it is better to coordinate faster, more direct responses to failures within the subsystems themselves.

8.2.1 Integrity of the Supervisory Layer

Another architectural aspect to consider is the fact that the supervisory layer itself may experience a fault. The failure of a subsystem within the supervisory layer would leave the ADS unmonitored and possibly unable to trigger a DDT fallback. This issue could be resolved by adding hardware redundancy to the entire supervisory layer. Another possible solution is adding lightweight monitoring capabilities, and redundancy, to the System Supervisor to execute immediate DDT fallbacks if anything in the supervisory layer fails.

8.3 Application on Other Systems

The ROD approach presented in this thesis is focused around its usage on an automated vehicle. However, the approach could be applied to any complex cyber-physical system with the following conditions:

- **The system operates in a varied domain with changing requirements.** A system designed to operate under changing requirements often results in portions of the system not being required for certain environments. For example, a long range forward-facing radar may only be required for high-speed roadways. Changing requirements provide an opportunity for the system to continue operating in the case of degradation to portions of the system not currently required.

- **Opportunity to plan alternate routes.** Another important aspect is the ability of the system to re-plan and take other actions to achieve the same goal. If the system operates in a limited environment without many, or any, alternate means to achieve the same goal, the system may not be able to continue operating much longer, simply because there are no less-demanding alternatives that allow the system to continue operating under degradation.

- **System architecture consists of a complex arrangements of subsystems.** If the system architecture is not very complex, it is likely that responses to system degradation can be designed directly into the system rather than coordinated by the supervisory layer. A benefit of the ROD approach is the use of the supervisory layer for responding to system degradation, since it removes all functionality related to handling of system degradation from the functional subsystems themselves. The subsystems only need to be aware of what possible ODD restrictions they would need to comply with. This allows for a more modular design and simplifies the functional subsystems as they do not need to be concerned with detecting, coordinating, and executing responses to system degradation (unless it makes sense for them to do so).

Assuming the above three conditions apply to a cyber-physical system, it is likely that the system will benefit from the application of the ROD approach.

8.4 Remaining Challenges and Future Work

Traceability between subsystem functionality and ODD: It may not always be easy to establish clear traceability between ODD elements and system functionality (or vice versa). Some elements of the ODD may be enabled by multiple subsystems working together. Therefore, if a link between an ODD element and every required subsystem is missed, it could cause the system to fail to restrict that element in the case of an impairment. It is also possible that subsystem functionality may impose very specific limitations to the ODD. For example, since the Autonomoose is a research platform, it is common for subsystems to be developed according to research objectives rather than being derived from system level requirements in a top-down fashion. In this case, a subsystem may have very specific or limited operating conditions. Translating these specific conditions into ODD elements can be difficult without a detailed representation of the ODD.

Future work could address this challenge by using a highly detailed and standardized ontology of the ODD defined by a structured language, such as the Web Ontology Language

(OWL), as suggested by [6]. Mappings between DOMs and ODD elements could then be defined using a structured language that integrates with the OWL representation of the ODD. This structured representation would also be used at runtime for ODD monitoring without the need to modify it.

Safety Validation: Another important consideration when determining traceability between DOMs and ODD element restrictions is safely validation. The proposed approach depends upon the assumption that the DOM to ROD element mappings are valid. This assumption guarantees that as long as the DOM is detected and the correct mapping is applied, the ROD will accurately represent a domain in which the degraded ADS can operate safely within. The challenge lies within the safety analysis and verification of these mappings. Each mapping will need to be evaluated and verified to ensure that the ODD restrictions being applied do indeed result in a safe ROD given the degraded functionality. The safety verification process for all mappings could potentially increase up-front costs in the overall design process. However, if this process can ensure safe operation under system degradation, it provides the opportunity to reduce other methods of safety assurance. For example, the reduction of hardware redundancy measures could be highly profitable when mass-producing vehicles.

Future work could explore the impact of the proposed approach on the overall design process and propose concrete methods for evaluating the safety of the ADS within the ROD under various DOMs. In addition to exploring the verification of single mappings, the verification of combinations of mappings will also need to be addressed. The work of Stewart et al. provides a compositional verification tool for proving high level safety properties using architectural models of the system [32]. The tool splits the safety analysis of large systems into verification tasks based on the structure of the architecture. The use of this tool to both link ODD elements to subsystems, and to verify the safety of the ADS under different system degradation and ODD restriction scenarios is considered a promising direction for future work.

Interaction with vehicle controls: The ROD is dynamic and can change during runtime to reflect the current capabilities of the system. Intermittent hardware failures could lead to randomly changing DOMs and possibly result in fluctuating ODD restrictions. The possibility for restrictions to change or fluctuate frequently could result in problems for the lower-level vehicle control systems that operate at high frequencies. These issues could lead to vehicle stability problems if not addressed properly. It is also possible for unintended feedback loops to emerge via the supervisory layer. For example, the ROD may be restricted to decrease the current speed of the vehicle in order to improve the localization results. The vehicle then slows down and the localization results improve, which then increases the restricted speed limit causing the vehicle to increase speed. This increased speed could then reduce the quality of the localization solution. Overall, the effect could be an unstable feedback loop between the vehicle speed controller and the ROD restrictions based on localization accuracy.

Future work could address these challenges by latching certain restrictions to the ODD with timeouts to prevent unwanted fluctuations. Subsystem monitoring (especially hardware subsystems) would also have to be designed to account for the possibility of intermittent failures. A control system analysis could also be performed to ensure the low-level controllers always remain stable given certain restrictions by the supervisory layer.

ROD-integrated map and route planner: As discussed in Section 8.1, the onboard map and route planning subsystems are very important to taking advantage of the benefits of the ROD approach. The route planning subsystem must be able to traverse the map while being aware of which portions of the road structure (category 1 of the ODD) are restricted according to the ROD. At the same time, the subject vehicle behaviour (category 6 of the ODD) must also be considered when planning routes and future manoeuvres. To achieve this, the map data must be linked to portions of the ODD and be able to respond to the changes in the ROD. Assuming the road network is represented in sufficiently detailed segments such as lanelets, proposed by Bender et al. [8], each lanelet has an associated manoeuvre and important regulatory details such as speed limits or stop lines. The onboard map provider could combine categories 1 and 6 of the ROD in order to deliver a version of the map that only contains lanelets that do not violate the ROD. The route planner would then be free to plan a route throughout the map with the assurance that the route will not violate, or at least provide precise locations where the ROD will be violated and a DDT fallback will have to occur.

Maps could also be used to transform certain elements of the ODD into others. Perhaps some ODD elements are difficult to monitor at runtime, such as the existence of railway crossings, for example. Rather than having the ADS check for the existence of railway crossings in the planned route, the map could be pre-processed to exclude lanelets that lead to railway crossings. This way, the onboard monitoring systems would not have to worry about monitoring the railway crossing element of the ODD whatsoever, as it has been transformed into the boundary of the lanelets, which could be monitored more easily.

The user experience: An important aspect to consider with the ROD approach is the user's experience during system degradation. While the vehicle may be able to continue operating safely and re-plan a route to the destination, the vehicle user needs a way to choose the options available. It is possible that a user would prefer to take over manual control rather than losing time on the trip by taking other routes. An HMI that is able to clearly communicate information such as what degradation the system experienced, how it affects the current driving mission, and what options there are for continued safe driving, is considered a key aspect to making the ROD approach integrate smoothly into the overall user experience.

Restricting the ODD for other reasons: Currently the ROD is modified based on the system health report only (in order to respond to system degradation). Future work could explore modifying the ROD based on other factors such as energy saving goals or user preferences. The context-aware nature of the architecture could be used to dynamically adapt the system based on the current operational domain. For example, if the vehicle is traversing a highway it could shut down certain sensors or subsystems that are not required for highway driving. The mechanism for turning these sensors back on when leaving the highway could be activated by the existing mechanism for identifying an upcoming exit of the ROD. Another example is a user preferring to sacrifice time for ride comfort. The user's "preference profile" could modify the ROD to constrain certain path planning requirements to satisfy the needs of the rider. If the vehicle was unoccupied, the profile might allow less comfortable manoeuvres in order to reduce the overall duration of the trip.

Chapter 9

Conclusion

This thesis identified the need to integrate self-awareness and fault tolerance aspects into an ODD monitoring system to maintain a runtime representation of the ODD. To address this integration need, we introduced the concept of restricting the runtime ODD based on subsystem degradation modes. The proposed approach allows an ADS to continue to operate within a safe domain and monitor the boundary of the safe domain during changing system capabilities and faults. A proposed architectural design was implemented in ROS as a proof of concept. The proof of concept demonstrated the viability of the proposed approach, but also exposed some remaining challenges. Some of these challenges include the definition and safety validation of mappings between DOMs and ROD elements, and system latencies induced by the proposed supervisory layer. Future work could address these challenges by investigating the use of architectural modelling tools to help perform safety validation on the DOM to ROD mappings, and analyze the architectural trade-offs associated with the proposed supervisory layer. As determined in the discussion, the map and route planning subsystems are very important for maximizing the benefits of the ROD approach. Exploring the integration of ROD elements with the map and route planning subsystems could allow the ADS to stay on the road longer and is another possible direction for future work.

The presented ROSDB architecture extraction tool was successful in producing detailed documentation of our ROS-based Autonomoose software. The tool guided the development and integration of our system architecture and increased visibility of our safety and validation procedures/documents. Continued improvements of the tool, such as complete ROS API capturing, will allow it to become a useful tool for any complex ROS-based system.

References

[1] Autonomoose public drive. youtu.be/i7S-JZYdb74?t=396, December 2017.

[2] Qt. www.qt.io/what-is-qt/, Retrieved April 2018.

[3] Tesla software updates. tesla.com/support/software-updates, Retrieved April 2018.

[4] *Road vehicles – Functional safety*, ISO 26262, 2011.

[5] *SURFACE VEHICLE RECOMMENDED PRACTICE Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*, SAE J3016, 2016.

[6] Gerrit Bagschik, Till Menzel, and Markus Maurer. Ontology based scene creation for the development of automated vehicles. *arXiv preprint arXiv:1704.01006*, 2017.

[7] Sagar Behere and Martin Trngren. A functional reference architecture for autonomous driving. *Information and Software Technology*, 73:136 – 150, 2016.

[8] P. Bender, J. Ziegler, and C. Stiller. Lanelets: Efficient map representation for autonomous driving. In *2014 IEEE Intelligent Vehicles Symposium Proceedings*, pages 420–425, June 2014.

[9] A. Bubeck, F. Weisshardt, and A. Verl. Bride - a toolchain for framework-independent development of industrial service robot applications. In *ISR/Robotik 2014; 41st International Symposium on Robotics*, pages 1–6, June 2014.

[10] KJ Bussemaker. Sensing requirements for an automated vehicle for highway and rural environments. 2014.

[11] K. Czarnecki. Operational World Model Ontology for Automated Driving Systems. Parts I and II. available at researchgate.net, November 2017.

[12] Alan G Ganek and Thomas A Corbi. The dawning of the autonomic computing era. *IBM systems Journal*, 42(1):5–18, 2003.

[13] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30(11):1203–1233, September 2000.

[14] Sebastian Geyer, Marcel Baltzer, Benjamin Franz, Stephan Hakuli, Michaela Kauer, Martin Kienle, Sonja Meier, Thomas Weißgerber, Klaus Bengler, Ralph Bruder, et al. Concept and development of a unified ontology for generating test and use-case catalogues for assisted and automated vehicle guidance. *IET Intelligent Transport Systems*, 8(3):183–189, 2013.

[15] Markus Hörwick and Karl-Heinz Siedersberger. Strategy and architecture of a safety concept for fully automatic and autonomous driving assistance systems. In *Intelligent Vehicles Symposium (IV), 2010 IEEE*, pages 955–960. IEEE, 2010.

[16] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The architecture tradeoff analysis method. In *Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No.98EX193)*, pages 68–78, Aug 1998.

[17] Andre Kohn, Rolf Schneider, Antonio Vilela, Andre Roger, and Udo Dannebaum. Architectural concepts for fail-operational automotive systems. Technical report, SAE Technical Paper, 2016.

[18] Philip Koopman and Michael Wagner. Transportation CPS safety challenges. In *NSF Workshop on Transportation CyberPhysical Systems*, 2014.

[19] Jason Ku, Melissa Mozifian, Jungwook Lee, Ali Harakeh, and Steven Lake Waslander. Joint 3d proposal generation and object detection from view aggregation. *CoRR*, abs/1712.02294, 2017.

[20] P. S. Kumar, W. Emfinger, A. Kulkarni, G. Karsai, D. Watkins, B. Gasser, C. Ridgewell, and A. Anilkumar. Rosmod: a toolsuite for modeling, generating, deploying, and managing distributed real-time component-based software using ros. In *2015 International Symposium on Rapid System Prototyping (RSP)*, pages 39–45, Oct 2015.

[21] F. Kunz, D. Nuss, J. Wiest, H. Deusch, S. Reuter, F. Gritschneder, A. Scheel, M. Stbler, M. Bach, P. Hatzelmann, C. Wild, and K. Dietmayer. Autonomous driving at Ulm university: A modular, robust, and sensor-independent fusion approach. In *2015 IEEE Intelligent Vehicles Symposium (IV)*, pages 666–673, June 2015.

[22] Richard Matthaei and Markus Maurer. Autonomous driving–a top-down-approach. *at-Automatisierungstechnik*, 63(3):155–167, 2015.

[23] Morgan Quigley, Josh Faust, Tully Foote, and Jeremy Leibs. Ros: an open-source robot operating system.

[24] Andreas Reschka, Gerrit Bagschik, and Markus Maurer. Towards a system-wide functional safety concept for automated road vehicles. In *Automotive Systems Engineering II*, pages 123–145. Springer, 2018.

[25] Andreas Reschka, Gerrit Bagschik, Simon Ulbrich, Marcus Nolte, and Markus Maurer. Ability and skill graphs for system modeling, online monitoring, and decision support for vehicle guidance systems. In *Intelligent Vehicles Symposium (IV), 2015 IEEE*, pages 933–939. IEEE, 2015.

[26] Andreas Reschka, Jürgen Rüdiger Böhmer, Tobias Nothdurft, Peter Hecker, Bernd Lichte, and Markus Maurer. A surveillance and safety system based on performance criteria and functional degradation for an autonomous vehicle. In *Intelligent Transportation Systems (ITSC), 2012 15th International IEEE Conference on*, pages 237–242. IEEE, 2012.

[27] Andreas Reschka and Markus Maurer. Conditions for a safe state of automated road vehicles. *it-Information Technology*, 57(4):215–222, 2015.

[28] Pascal Roques. MBSE with the ARCADIA Method and the Capella Tool. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, January 2016.

[29] mer Sahin Tas, Florian Kuhnt, J Zllner, and Christoph Stiller. Functional system architectures towards fully automated driving, 06 2016.

[30] A. Santos, A. Cunha, N. Macedo, and C. Loureno. A framework for quality assessment of ros repositories. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4491–4496, Oct 2016.

[31] Johannes Schlatow, Mischa Moostl, Rolf Ernst, Marcus Nolte, Inga Jatzkowski, Markus Maurer, Christian Herber, and Andreas Herkersdorf. Self-awareness in autonomous automotive systems. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1050–1055. IEEE, 2017.

[32] Danielle Stewart, Michael W Whalen, Darren Cofer, and Mats PE Heimdahl. Architectural modeling and analysis for safety engineering. In *International Symposium on Model-Based Safety and Assessment*, pages 97–111. Springer, 2017.

[33] Ömer Şahin Taş, Florian Kuhnt, J Marius Zöllner, and Christoph Stiller. Functional system architectures towards fully automated driving. In *Intelligent Vehicles Symposium (IV), 2016 IEEE*, pages 304–309. IEEE, 2016.

[34] Dirk Thomas. rqt_graph. wiki.ros.org/rqt_graph, Retrieved April 2018.

[35] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, et al. Stanley: The robot that won the darpa grand challenge. *Journal of field Robotics*, 23(9):661–692, 2006.

[36] David Wittmann, Cheng Wang, and Markus Lienkamp. Definition and identification of system boundaries of highly automated driving. In *7. Tagung Fahrerassistenz*, 2015.

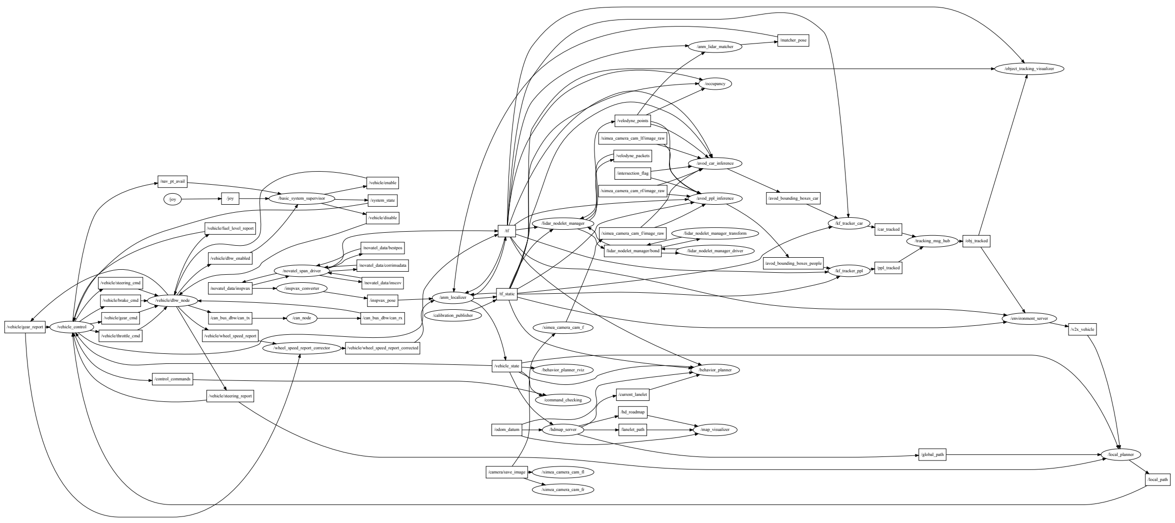
APPENDICES

Appendix A

Generated Documentation from the ROSDB Tool

A.1 System Overview Page

public_drive_target snapshot



In This Snaphot:

Packages

- anm_lidar_matcher **R** **V**
- environment_server **R** **S** **V**
- tracking_msg_hub **R** **S** **V**
- anm_localizer **R** **V**
- calibration_publisher **R** **S** **V**
- novatel_span_driver
- inspvax_converter **R**
- behavior_planner **R**
- kf_tracker **R** **S** **V**
- vehicle_control **R** **S**
- wheel_speed_report_corrector **R**
- avod_ros_integration
- occupancy **R** **S** **V**
- basic_system_supervisor **R**
- command_checking **R**
- ximea_ros_cam **R**
- object_tracking_visualizer **R**
- map_visualizer **R**
- hdmap_server **R**
- local_planner **R** **S** **V**

- dataspeed_can_usb
- velodyne_driver
- nodelet
- velodyne_pointcloud
- joy
- dbw_mkz_can

ROS Parameters

All ROS Parameters

Compiled Documentation

Safety Driver Summary (markdown)
System-wide ODD (markdown)

API Table

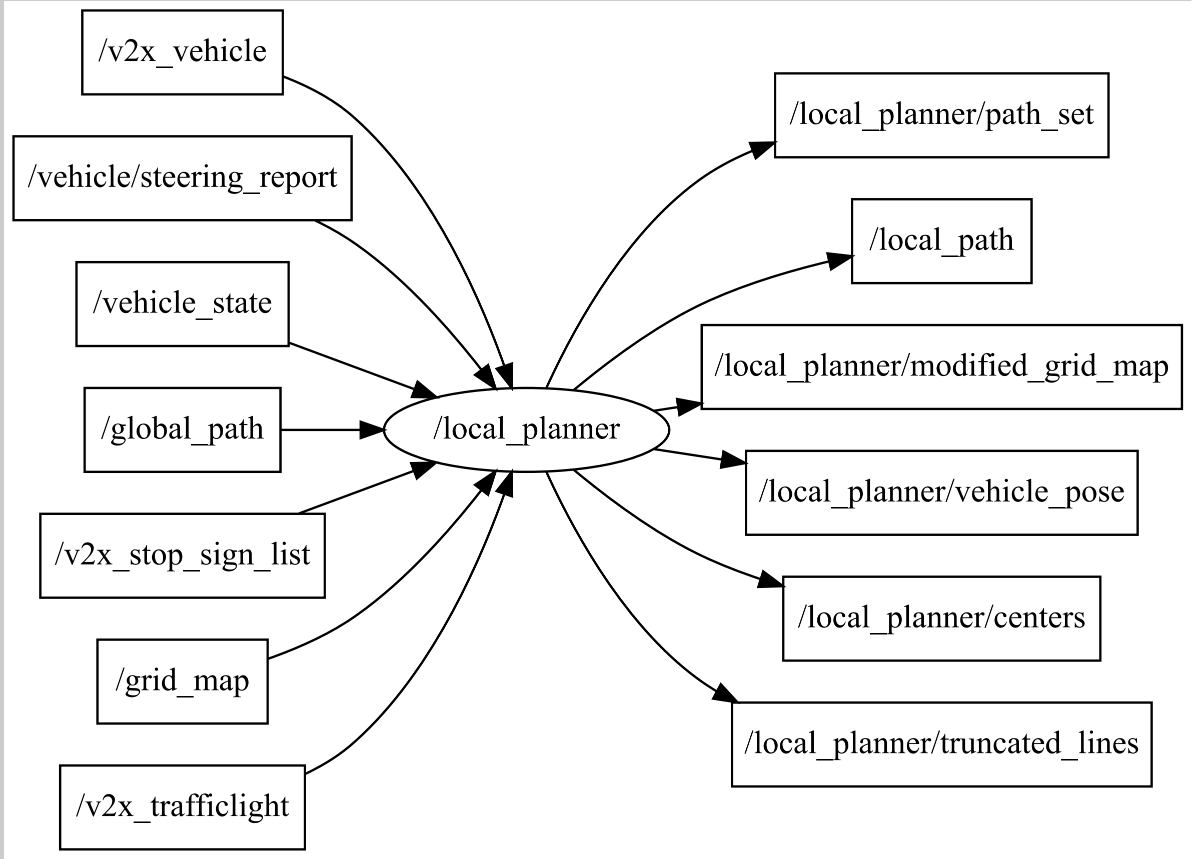
View API table
Download API table in csv

Date generated: 2018-04-25

A.2 ROS Package Documentation

local_planner

Version 1.1.1



Nodes

local_planner_node

Topics subscribed to:

- /v2x_stop_sign_list [anm_msgs/V2XStopSignList]
- /vehicle/steering_report [dbw_mkz_msgs/SteeringReport]
- /v2x_trafficlight [anm_msgs/V2XTrafficLightList]
- /v2x_vehicle [anm_msgs/V2XVehicleList]
- /grid_map [nav_msgs/OccupancyGrid]
- /global_path [nav_msgs/Path]
- /vehicle_state [anm_msgs/VehicleState]

Topics published to:

- /local_planner/vehicle_pose [visualization_msgs/Marker]

- /local_planner/centers [visualization_msgs/MarkerArray]
- /local_planner/modified_grid_map [nav_msgs/OccupancyGrid]
- /local_planner/path_set [visualization_msgs/MarkerArray]
- /local_planner/truncated_lines [visualization_msgs/Marker]
- /local_path [nav_msgs/Path]

Associated Parameters

- /local_planner/base_distance_dobs: 6.0
- /local_planner/check_for_static_collisions: True
- /local_planner/effective_heading_angle_reg_elem: 45
- /local_planner/enable_obstacle_list_subscriber: False
- /local_planner/enable_path_set_viz: True
- /local_planner/enable_pointcloud2_subscriber: True
- /local_planner/enable_v2v_subscriber: True
- /local_planner/end_deceleration: 0.3
- /local_planner/end_speed_limit: 3.0
- /local_planner/extend_length: 0.0
- /local_planner/goal_threshold: 0.1
- /local_planner/lateral_acceleration_limit: 0.4
- /local_planner/local_path_pp: 0.1
- /local_planner/local_path_tn: /local_path_desc
- /local_planner/lookahead: 10.0
- /local_planner/min_goal_separation: 0.005
- /local_planner/moving_vehicle_effect_radius: 30.0
- /local_planner/parked_vehicle_effect_radius: 10.0
- /local_planner/path_number: 15
- /local_planner/safe_distance: 0.5
- /local_planner/safe_stop_distance: 8.0
- /local_planner/slow_speed: 0.5
- /local_planner/speed_threshold: 8.0
- /local_planner/state_subscriber_type: anm
- /local_planner/truncate_path: True
- /local_planner/viz_enabled: True

README File

Overview

This package constantly searches the `global_path` and finds the best goal state based on the lookahead distance and the pose of the car relative to the global path. Then it generates a rich set of paths and selects the best according to the objective functions(for the ces demo, it only generates one path). The speed profile will be generated according to the dynamic constraints, limits from recorded speed profile, the speed of the detected vehicle and v2x information.

Ros Interface

Subscribers:

- `global_path` (nav_msgs/Path)
- `vehicle_state` (anm_msgs/VehicleState)
- `v2x_stopsign` (anm_msgs/V2XStopSignList)
- `v2x_vehicle` (anm_msgs/V2XVehicleList)
- `v2x_trafficlight` (anm_msgs/V2XTrafficLightList)

Publishers:

- `local_path` (nav_msgs/Path)
  It represents the local path in global frame(odom frame).

- `local_planner/centers` (visualization_msgs/MarkerArray)
  It shows the circles that cover the footprint of the car.

- `local_planner/vehicle_pose` (visualization_msgs::Marker)
  It publishes the vehicle state received from `ref_ekf` and recent goal state on the global path for the local planner.

- `local_planner/path_set` (visualization_msgs/MarkerArray)
  It shows all the candidate paths generenated by local planner.

- `local_planner/truncated_lines` (visualization_msgs::Marker)
  This is the projection point marker of the moving vehicle on the local path. If there is no moving vehicle, the marker will stay at the rear center of the car.

Important Parameters

- `speed_threshold` (float in `m/s`)
  This is the max speed threshold of the local planner.

- `state_subscriber_type` (string)
  anm: anm_msgs/VehicleState msg with dataspeed steering report msg.

  odom: nav_msgs/odometry msg that inludes steering angle information.

- `enable_v2v_subscriber` (boolean)
  In order to getting the moving vehicle information, this value should be `true`.

- `end_deceleration` (float in `m/s^2`)
  Adjust the brake force at the stop sign, traffic light and end of the global path.

Test, Visualization and Troubleshooting

Individual Launch File

roslaunch local_planner local_planner_node.launch

Visualization

If you want to see all the features of the local planner, please load `mkz_viz.rviz` file in `local_planner/config/` folder or the rviz icon on the desktop.

Troubleshooting

If you launch the whole system, there is no local path being published. There are several cases that may cause the problem.

- Time synchronization
  This usually happens when you do multiple machines simulation or distributed system communication. Check if the timestamps of `vehicle/steering_report` and the one of `vehicle_state` are close enough. If these messages come from separate machines, you need to synchronize the time of the two machines with the NTP tool.

- The local planner doesn't get all the necessary inputs from ROS.
  Please check every topic needed by local_planner.

Subsystem Specification

Behaviour and Local Planner

WARNING

Several complicated bugs have been discovered in the planner that affect its ability to safely follow leading vehicles as well as manage stop signs. Until further notice, assume that it can do neither of these things reliably.

If the vehicle is too far from the goal path, the planner will terminate.

Package Version

1.1.1

Overall Function Description

Generates a path and a velocity profile, subject to static objects, dynamic obstacles, stop signs, traffic lights, expected driving behaviours and dealing with physical constraints of the car. To be divided into two modules in the future.

Functional Requirements

Path Definition

- Path is at least c2 smooth
- Path is curvature constrained
    - Minimum turning radius constrained - ignores speed
    - Minimizes bending energy
- Path tries to reach final x, y and heading (derived from consecutive points) of goal state
    - Trying to reach path is enforced as soft constraints
    - WARNING: Does not account for intermediate points on the global path and may cut corners as a result, or loop out into other lane
- Path set is a set of paths perturbed laterally from the original goal state
    - In a second optimization function, paths closer to the goal state have higher weights
- Attempts to find a collision free path from a path set
    - If this is not possible returns the truncated path just in front of obstacle along original goal path
    - Collision checking is with respect to static obstacles from occupancy grid
    - Collision checking is done through circle intersections; a path may be marked as infeasible even if it is feasible

Velocity Profile Definition

- Speed along path is always limited by universal speed limit of 50 kph (14 m/s)
    - To be converted to depend on road segment speed limit
- Speed along path is always limited by maximum lateral acceleration, defined by path curvature
- If current speed is below maximum speed, acceleration from current speed to maximum allowed is limited by constant acceleration limit forward from current location until maximum speed is reached.
    - Should have velocity dependent acceleration profiles for smooth transitions.
- Speed limit at terminal point of path is set to zero if vehicle must stop at the end of the path, as required for parking and stop sign/traffic light intersection
- If within safe stopping distance, velocity set to zero and controller manages stopping
    - WARNING: Safe Stopping Distance is fixed, not ego-velocity or obstacle dependent
- If beyond safe stopping distance, constant deceleration to zero speed over the same fixed stopping distance.
    - Should have a velocity dependent safe stopping distance and a comfortable deceleration profile (min jerk is an option), and an emergency stopping profile.
- If the path is infeasible, speed profile is set to stop within Safe Stopping Distance
    - WARNING: no alert to driver, paths on rviz go red.

Behaviour Definition (to move to behaviour planner module)

- Path following (unconstrained driving)

    - Path and speed profile output based on above without modifications

- Vehicle following (lead vehicle constrained driving)

    - Vehicle is commanded to track lead vehicle target speed when target vehicle is within tracking distance
    - If the vehicle in front is going faster, our vehicle speeds up, up to the minimum of the target speed, speed limit and curvature constrained speed
    - If the vehicle in front is going slower

        - If within a fixed range to collision threshold, perform hard deceleration (emergency stop per above)
        - Else smooth deceleration (constant deceleration) to match speed
    - WARNING: Does not handle stopped vehicle correctly
        - Should be resolved
    - WARNING: No prediction of target speed over path

- Stop Sign Intersection

    - Upon approach, the speed profile along path is set to decelerate to zero and stop at the stop line end point.
        - Should be stopped at correct location in lane
    - Once the vehicle is stopped at the stop line end point, this stopped state is held for 3 seconds, after which the vehicle plans a path and speed profile to turn right.
    - If there is a dynamic obstacle within 7m of the vehicle, the speed profile for the turn maneuver is held to zero speed until no dynamic obstacles are present in the region of interest
        - Should be â€œIf there is an insufficient time to execute a turn without slowing down an oncoming vehicle from west approach, , the speed profile for the turn maneuver is held to zero speed until a sufficient time gap exists
    - Intersection restrictions for stop sign
        - Only considers a vehicle coming from the left and proceeding straight.
        - Assumes all vehicles coming from the left are proceeding straight through the intersection, blocking progress.
        - Expected usage is t-intersection, two lane road, with a stop sign for ego vehicle only
        - May need to be conservative and wait for any traffic from the East turning left as well.
        - COLLISION RISK: vehicles turning left from the right are ignored.
- Traffic Light Intersection
    - Upon approach, if traffic light state is RED or YELLOW, the path endpoint is set on intersection stop line if carâ€™s global path intersects with stop line drawn perpendicular to lane heading. Speed profile is set to zero at end point.
    - Only considers vehicle directly in front of itself when in vehicle following mode. No other vehicles impact plan or profile at traffic light intersection.
    - Does not detect or avoid vehicles running red light in perpendicular direction.
    - TODO: Revisit

Configuration Parameters

- Maximum speed
- Fixed Acceleration Rate (Normal Operation)
- Maximum Path Curvature = 0.5 m^-1
- Path Arc Length Limit = 100 m
- Goal State Set Offset = 0.2 m
- Number of Paths Generated = 21 (10 on each side of center)
- Goal State Lookahead Distance = 10 m (nominal, changes with vehicle speed)
- Safe Stopping Distance To Static Collision on Infeasible Path = 5 m
    - When a collision is detected on the path, this is the threshold determining whether an attempt to smoothly brake will be made instead of a hard stop
    - Ignores speed so does not account for actual braking distance

- Safe Distance from Dynamic Obstacle = 10 m
    - The minimum distance the base link of the ego vehicle can be from the location of a dynamic obstacle
    - Used to calculate â€œdynamic safe distanceâ€ in vehicle following velocity calculation, is empirical based on ego vehicle speed (not obstacle speed), corresponds to the distance from the leading vehicle where we would like the ego vehicleâ€™s speed to match the leading vehicleâ€™s speed. Probably should be renamed to distance gap, as it is approximately the distance gap we would like to maintain.
- Safe Distance Buffer (Collision Clearance) from Static Obstacle = 0.5 m
    - WARNING: Total buffer may be double the above plus the circle approximation bias for an upper limit of 1.25m
- Speed Limit = 14 m/s
    - TODO: switch to 14 m/s in code
- Speed Limit near stopped cars = 0.5 m/s
    - This should be dead code for Colby as it is only used for v2x cues for another vehicle possibly skipping a stop sign without visibility
    - To be removed and relocated to BP
- Speed Limit for final path segment at end of global path but before final point = 2.5 m/s
    - Should be OK for Colby
    - Should become a failure handling mode for when the BP stops publishing without telling LP to stop
- Moving Vehicle Effect Radius = 20 m
    - Sight distance for dynamic obstacles at stop sign intersections
    - Just for stop sign intersections
- Parked Vehicle Effect Radius = 10 m
    - Sight distance for CES specific scenario (see next point)
    - Paired only with â€œSpeed Limit near stopped carsâ€ above?
- Grid Map Size = 80 m longitudinal x 40 m lateral
    - Internal representation used to fuse occ grid, lane boundaries (future)

Input Requirements

- Global path point sequence in odom frame, always available?
    - x and y coordinates only
    - Global path curvature not required, computed from point sequence
    - Global path cannot loop on itself
    - Minimum global path length: K points
    - Minimum point spacing: subsequent points closer than a configurable threshold are dropped
    - Local path unable to track a three point curvature estimate than is greater than max curvature
        - WARNING: Currently fails silently
- Vehicle state from EKF in odom frame at least 10 Hz
    - Data age requirement TBD, should be very low (much less than 150 ms)
- Stop sign list and traffic light list along global path
    - Stop line location along path for all intersections
    - Traffic light state at least 15 m from intersection stop line
- Static Occupancy grid in base link frame at 10 Hz with <150 ms data age
    - Dynamic object must removed from the grid
- Car, pedestrian and bicycle tracks at 10 Hz with <150 ms data age

    - X and Y coordinates to within 25 cm for all cars, bicycles and pedestrians within occupancy grid extent
    - Speed and heading estimates for all cars, bikes and pedestrians
    - TODO: Other modules (TBD) need to address and fix the latency and propagation of dynamic object timing via prediction. Local Planner shouldnâ€™t be addressing predictions based off of 150ms latencies

Output Requirements

- 2D pose path packaged into nav_msgs/path which is 3D pose sets using quaternions for orientation
    - X, y, yaw define planar vehicle positions
    - z is used for speed
    - Roll, pitch, are ignored
- Output frequency = 10 Hz
- Average Latency < 50 ms
    - Ball park figure dependent on hardware limitations
- Max driving speed of 14m/s (50 km/h)
- Max arc length path is less than 100m

Important Safety Considerations

- Does not account for intermediate points on the global path and may cut corners as a result, or take wide turns out into other lane.
- Safe Stopping Distance is fixed, not ego-velocity or obstacle dependent.
- If all paths are infeasible there is no alert to driver. Paths on rviz go red.
- Does not handle stopped vehicle correctly. A hard stop may happen if a vehicle is parked on the side of the road.
- No prediction of the lead vehicle (vehicle being followed) target speed over path.
- Local path unable to track a three point curvature estimate that is greater than max curvature and fails silently.
- Safe Distance Buffer (Collision Clearance) from Static Obstacle = 0.5 m
    - Total buffer may be double the above plus the circle approximation bias for an upper limit of 1.25m
- The `moving_vehicle_effect_radius` parameter is set to 40m. As a result, any dynamic obstacles have to be quite a ways out of sight before the car will disregard them. This may need tuning after in-car testing in the future.
- If the goal path is entirely beyond the lookahead, the planner will fail. Currently this is via an unhandled exception crash. A subsequent version will fix this.

Operational Design Domain Constraints

- Weather Conditions
    - Non-slippery road surface
- Roadway Types
    - Any roadway for which global path provided can be followed (not blocked by obstacles such as parked cars)
    - No tight loops

    - Three way stop sign intersections and two lane four-way traffic light intersections only
        - To be migrated to Behavioural Planner
    - Limited levels of banking due to the bicycle model
    - No speedbumps
        - To be revisited by BP or other modules
    - No significant crest and sag curves (hills)
- Traffic Conditions
    - No merging traffic
    - No traffic pulling out of side roads/driveways
    - No vehicles violating centre line of roadway
    - Only one vehicle in an intersection at a time
    - Parked cars remain parked
    - Leading cars must not stop and stay stopped
- Ego vehicle limits
    - Maximum 50 kph

Subsystem Dependencies

```
- Route publisher
- Environment server
    - Tracker
- Occupancy grid mapper
- Ref_EKF
- Behaviour planner (future)
```

Validation

Local Planner

Validation Procedure

Most of the testing to be done on this module are behavior in simulation or on board vehicle testing.

Unit Testing

While the local planner has unit testing, they are limited in three important ways:

- The existing unit tests do not test for corner cases
- Due the simplicity of the unit tests the systems may still produce unexpected behavior even if all unit tests pass.
- The Unit tests have very low coverage.

Thus even if all unit tests pass the system should still not be trusted. That being said no changes to the local planner should ever result in the unit test not passing.

Simulation Testing

There are 3 main simulation scenario that should be using to test all the functionality of the local planner:

- obstacle_avoidance used to make sure the car is generating a correct path with an expected velocity profile, as well as being able to avoid obstacles when required.
- colby_drive_v2x_objects used to make sure that the car still correctly stops at a stop sign, and does proper vehicle following.
- adm_ces_from_B22_right used to check traffic light intersection behavior.

Vehicle Testing

When testing on the car RVIZ should be used at all times as this is the first indication of an error. RVIZ should have the following displays the global_path, local path, all possible local, the location and heading of the ego vehicle, the location and heading of all regulatory elements, the location and heading of any vehicles, and an post for the current vehicle following if any.

It should be noted that RVIZ is not always sufficient and the behavior of the car should be closely monitored by the safety driver in the vehicle currently. The car should be stop at the first behavior fault detected if this fault could cause injury.

Validation Logs (DD.MM.YYY)

- Unit Testing is verified by Travis CI builds
- Simulation Testing was performed on 6.04.2018 by MichaÅ, Antkiewicz on version 1.1.1
- Vehicle Testing was performed on 15.12.2017 by MichaÅ, Antkiewicz on version 1.0.5
- Waypoint Following (partial vehicle test) was performed on 25.01.2018 by MichaÅ, Antkiewicz on version 1.0.10