Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Software Engineering

A Software Solution for Preprocessing Objects in Object-Level Multisensor Fusion for Self-Driving Vehicles Perception

Author: Ícaro Pires de Souza Aragão

Supervisor: Prof. M.Sc. Giovanni Almeida Santos

Brasília, DF

2021

Ícaro Pires de Souza Aragão

A Software Solution for Preprocessing Objects in Object-Level Multisensor Fusion for Self-Driving Vehicles Perception

Research thesis submitted in partial fulfillment of the requirements for the bachelor degree in Software Engineering from Universidade de Brasília.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Supervisor: Prof. M.Sc. Giovanni Almeida Santos

Brasília, DF

2021

Ícaro Pires de Souza Aragão
A Software Solution for Preprocessing Objects in Object-Level Multisensor Fusion for Self-Driving Vehicles Perception/ Ícaro Pires de Souza Aragão. – Brasília, DF, 2021-
81 p. : il. (algumas color.) ; 30 cm.

Supervisor: Prof. M.Sc. Giovanni Almeida Santos

Bachelor's Thesis – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2021.

1. Autonomous Driving Applications. 2. Multisensor Data Fusion. I. Prof. M.Sc. Giovanni Almeida Santos. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. A Software Solution for Preprocessing Objects in Object-Level Multisensor Fusion for Self-Driving Vehicles Perception

CDU 02:141:005.6

Ícaro Pires de Souza Aragão

A Software Solution for Preprocessing Objects in Object-Level Multisensor Fusion for Self-Driving Vehicles Perception

Research thesis submitted in partial fulfillment of the requirements for the bachelor degree in Software Engineering from Universidade de Brasília.

Approved Bachelor Thesis. Brasília, DF, May 25, 2021:

Brasília, DF
2021

Abstract

Self-driving cars is a trending topic. Much of the safety and security concerns in traffic may be reduced when cheap and functional versions from these vehicles become widespread. These vehicles are highly complex and advanced robots that, in order to avoid accidents and make the best decisions, use many sensors to perceive their surroundings in the best way possible. Multisensor data fusion can contribute to this task by joining each sensor's best pieces of information, building a more complete and more accurate version of this information. Software that perform such tasks are frequently reimplemented between companies and research centers for their specific hardware configurations and objectives. Object-level fusion performs fusion at a higher level of abstraction and, for this reason, contributes to the modularity and reuse of the solution. A software solution to address part of this reimplementation problem is developed in this work using the ROS 2 framework. It implements the object lists preprocessing from the fusion layer of an object-level fusion architecture. This preprocessing is composed of the spatial and temporal alignments, plus the objects associations. Finally, the solution design and implementation were validated using the CARLA self-driving simulator, using the number of failed associations as the primary metric assessed.

Key-words: Self-driving; Autonomous; ROS; Autonomous Driving Applications; Autonomous Driving Systems; Multisensor Data Fusion; Object-level fusion; CARLA; Multi-object tracking

Resumo

Carros autônomos é um assunto atual e popular. Muitas preocupações relativas a segurança no tráfego poderão ser reduzidas quando versões baratas e funcionais desses veículos se popularizarem. Esses veículos são robôs altamente complexos e avançados que, para evitar acidentes e tomarem as melhores decisões possíveis, usam uma variedade de sensores para perceber suas redondezas da melhor maneira possível. A fusão de dados de multi-sensores pode contribuir para essa tarefa unindo os melhores pedaços de informação de cada um dos sensores para formar uma versão mais completa e mais acurada dessa informação. Softwares que realizam essas tarefas são frequentemente reimplementados dentre diferentes empresas e centros de pesquisa para suas próprias configurações de hardware e objetivos. A fusão no nível dos objetos é realizada num nível mais alto de abstração e, por essa razão, contribui para modularidade e reuso da solução. Uma solução de software para resolver parte desse problema é desenvolvida neste trabalho utilizando o framework ROS 2. Ela implementa o preprocessamento das listas de objeto na camada de fusão de uma arquitetura de fusão à nível de objetos. Esse processamento é composto do alinhamento espacial e temporal, mais a associação de objetos. Por fim, este preprocessamento foi validado utilizando o simulador de veículos autônomos CARLA, utilizando como métrica principal o número de associações que falharam nos casos de teste.

Palavras-chave: Veículos Autônomos; Veículos auto-dirigidos; ROS; Aplicações de Direção Autônoma; Sistemas de Direção Autônoma; Fusão de Dados Multi-sensores; Fusão à nível de objetos; CARLA; Rastreio de múltiplos objetos;

List of Figures

List of Tables

List of Source Codes

List of abbreviations and acronyms

ADAS	*Advanced Driver Assistance Systems*
ADS	*Autonomous Driving System*
API	*Application Programming Interface*
DCNN	*Deep Convolutional Neural Network*
DDS-XRCE	*DDS For Extremely Resource Constrained Environments*
DDSI-RTPS	*DDS Interoperability Wire Protocol*
DDS	*Data Distribution Service*
DST	*Dempster-Shafer Theory*
EKF	*Extended Kalman Filter*
ENU	*East-North-Up*
FPS	*Frames Per Second*
GNSS	*Global Navigation Satellite System*
GPS	*Global Positioning System*
IDL	*Interface Definition Language*
IMU	*Inertial Measurement Unit*
IoT	*Internet of Things*
IoU	*Intersection over Union*
JDL	*Joint Directors of Laboratories*
LTS	*Long Term Support*
OEM	*Original Equipment Manufacturer*
OS	*Operating System*
QoS	*Quality of Service*
ROS	*Robot Operating System*

RTOS	*Real-Time Operating System*
SLAM	*Simultaneous Localization and Mapping*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*

Contents

1 Introduction

The usage of technology in the world is increasing. New technologies become popular every year by facilitating activities from many aspects of everyday life, such as recreation, study, work, and healthy. Undoubtedly, more and more complex electronic devices are responsible for much of these advances.

In this context, self-driving vehicles are emerging. In Brazil, 90% of road accidents are caused by human fault. Therefore, if we had fully functional and fully automated vehicles on the roads, these accidents could be avoided. Hence, it is not just about comfort and convenience. It is also about traffic safety and public health.

It is already a reality that the more modern the car, the more features it has to facilitate driving it. These features are called Advanced Driver Assistance Systems (ADAS). They are systems designed to assist a driver in certain situations or even controlling the vehicle (AEBERHARD, 2017). A vehicle that could assist the driver the most would be autonomous, "The ultimate ADAS is an autonomous driving system." (AEBERHARD, 2017).

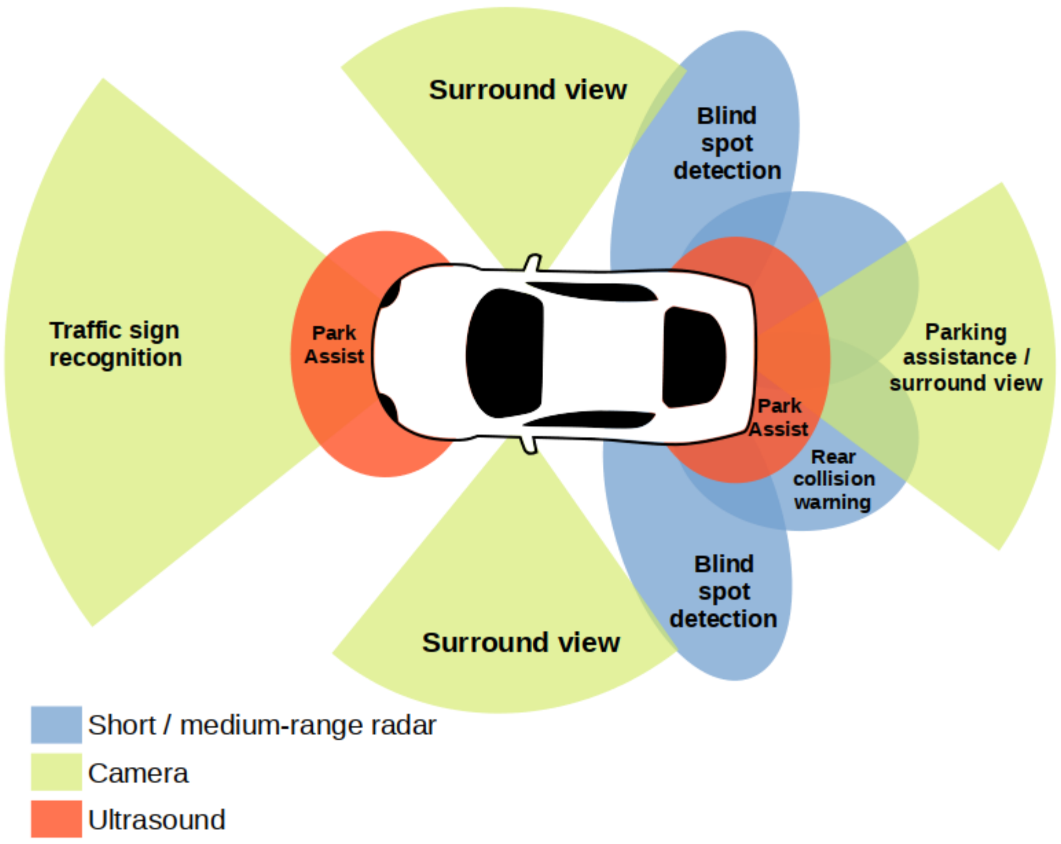
Since each sensor has its limitations and capabilities, multiple ADAS with different purposes may be developed. In this case, the vehicle can be seen as a hardware platform in which ADAS can be implemented on top, choosing relevant sensors for its features. An overview of the relationship between some commonly implemented ADAS in modern cars, and its used sensors, are illustrated in the Figure 1.

Usually, more capable devices, considering the number and complexity of features, are equipped with more sensors and actuators. As a consequence, there exist many scenarios in which different sensors generate constant flows of data. These data need to be transformed and merged so that the most amount of relevant information can be obtained and used to make decisions. This is the situation in the development of self-driving vehicles and is where sensor data fusion meets the subject.

1.1 Problem

Multisensor data fusion implementations, which are part of autonomous driving applications, usually inherit the complexity of embedded systems. The implementation is highly coupled with the sensors in use and the general setup. This sort of situation makes modifications to the project slow and expensive (MUNZ; MAHLISCH; DIETMAYER, 2010). That coupling also makes it harder to develop reusable solutions, like libraries or frameworks, which could spare time from several researchers when making experiments

Figure 1 – ADAS and Sensors Range



Source: Adapted from (CARPENTER, 2018)

and companies developing their products.

1.2 Objectives

1.2.1 Main Objective

The main objective is to propose and develop a modular and reusable software solution capable of preprocessing lists of objects generated by environment perception sensors, making the lists ready to be fused and used in self-driving applications.

1.2.2 Specific Objectives

1. Perform a study about autonomous driving applications, multisensor data fusion, and the relationship between them;
2. Define the data fusion architecture and describe its main characteristics;

3. Define the general design of the software solution, based on the defined data fusion architecture;
4. Define the communication interfaces with the proposed software solution;
5. Implement the software solution capable of performing spatial and temporal alignment, and object association;
6. Validate that the software solution is capable of preprocessing the objects lists and return results to another module;
7. Make the software solution available to be integrated with other compatible projects;

1.3 Methodology

To achieve the proposed objectives, the following steps are defined:

1. **Problem and domain understanding:** Study and describe the primary points related to this work goals: sensors, data fusion, and the autonomous driving scenario;
2. **Description of the fusion architecture:** Choose a data fusion architecture with well-defined components and potential to be used in a modular and reusable implementation, then describe the components of this chosen architecture;
3. **General requirements gathering:** Formulate the main functional and non-functional software requirements of the software solution;
4. **Search for helper software tools:** A software solution can be of different forms depending on the tools available, e.g. libraries, frameworks. This step is about looking for available free tools that can help to build the solution and meet the requirements for this specific context, autonomous driving;
5. **Architecture design:** Based on the found helper tools and defined fusion architecture, design the architecture overview from the implementation point of view, considering the interfaces and the communication between the components;
6. **Interfaces implementation:** An initial implementation of the architecture to test its feasibility. The chosen tools will be integrated into the project and the interfaces implemented;
7. **Preprocessing implementation:** Implement and describe the preprocessing algorithms and their interaction. The development will adopt an iterative and incremental approach, so there will not be a separation between the implementation and testing stages. Each increment will be designed, implemented and tested before implementing the next one (LARMAN; BASILI, 2003);

8. **Validation:** Validate, through an experiment, that the software solution can be used, as a separate module, interacting with some self-driving software simulator, and describe the obtained results.

9. **Repository documentation:** Write the repository documentation, including project description, instructions for usage and building, and usage examples.

10. **Packaging:** Package the repository so that users can integrate it into their projects through a package manager.

1.4 General Structure

This work is organized into two parts and nine chapters. This is the first chapter Chapter 1 and the chapter that presents the rest of the work.

Part I is mainly about the theory necessary to understand the development and designing of the software solution. It is divided into five chapters. Chapter 2 presents sensors in the context of self-driving applications, Chapter 3 introduces data fusion in the broader sense, Chapter 4 explains how usually data fusion is used to fuse data from sensors, Chapter 5 presents the object-level fusion architecture used in this work, and Chapter 6 explains the main aspects from the ROS framework and its application to the self-driving context.

Part II is divided into three chapters and is about the results achieved in this work, based on the theory from the last part. Chapter 7 explains aspects about the software solution definition and implementation, Chapter 8 describes the experiment used to validate the solution, and Chapter 9 brings the main conclusions about this work and suggestions for future works.

Part I

Theoretical Framework

2 Sensors

Sensors are physical devices developed to measure some attribute of the physical world. As the world is a very diverse place, there are many sensors with different costs, sizes, precision, measured attribute, and other aspects. Among the problems they can solve are detecting surrounding objects, estimating positions, movement detection, and systems state monitoring.

Taking the temperature attribute as an example, sensors about air temperature can be very different from engine temperature or computer processor temperature. It is needed to question, not exclusive but: if the sensor fits the structure, supports the weather condition, can deliver results with the expected precision, and has coherent costs. It is not always possible to measure some attribute directly. This situation requires using a sensor that measures another attribute than the desired one and then using a mathematical model to estimate the desired. These are indirect measurements. (MOUZINHO et al., 2006).

2.1 Self-Driving

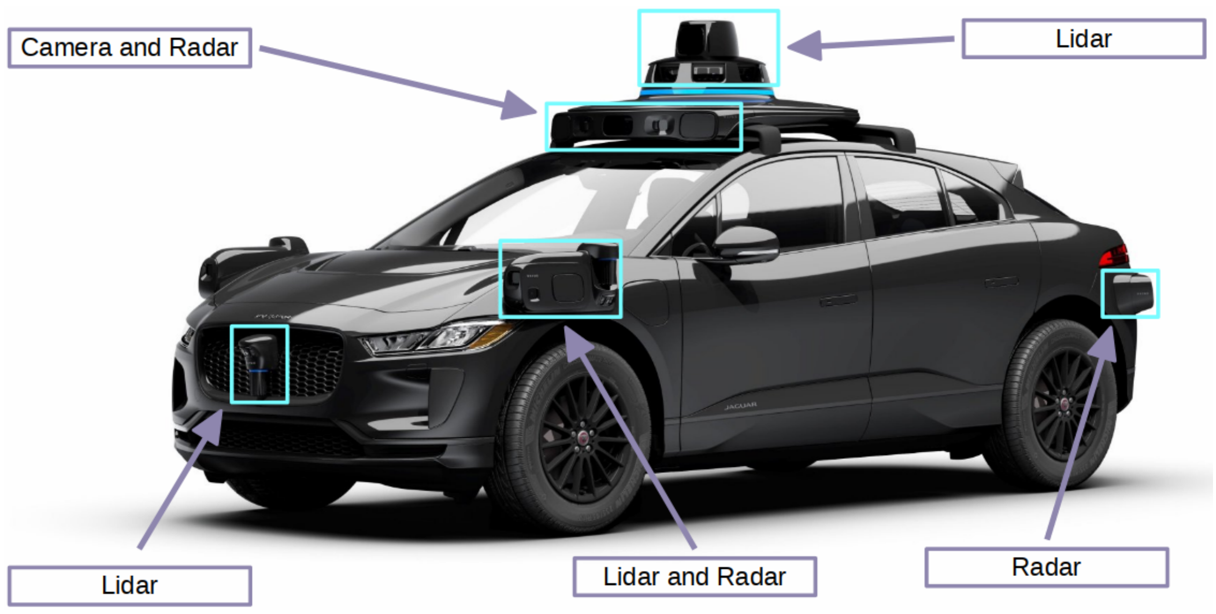
Despite the numerous kinds of sensors, complex applications, such as self-driving, besides not relying on a single sensor for some attribute estimation, eventually measure attributes that are not a physical quantity. It may be such a complex attribute as a whole surrounding reading. In this context, different kinds of sensors are capable of providing different readings depending on their functioning, current weather conditions (rain, fog, snow), luminosity conditions, among others.

Recent research summarized core functions for self-driving, or ADS, in *localization and mapping*, *perception*, *assessment*, *planning and decision making*, *vehicle control*, and *human-machine interface* (YURTSEVER et al., 2020). From these, localization and mapping, and perception are the functions that receive data directly from the sensors and generate higher-level information that the other functions can use.

Commonly used sensors for self-driving vehicles are cameras, lidar, radar, sonar, Global Navigation Satellite System (GNSS), Inertial Measurement Unit (IMU), and wheel odometer (KOCIĆ; JOVIČIĆ; DRNDAREVIĆ, 2018). GNSS, IMU, and wheel odometer are often used for localization and mapping and the others for perception.

The Figure 2 shows how the Waymo company is positioning some perception sensors in the Jaguar I-Pace model. It is a great example of which sensors are used by the industry in autonomous cars, and how they are positioned.

Figure 2 – Waymo's autonomous Jaguar I-Pace - Some of the sensors



Source: Adapted from (LI, 2021), and (BALDWIN, 2020)

2.1.1 Localization and Mapping

Localization and mapping are processes used to acquire the vehicle's position respecting the requirements demanded by ADS. The natural guess for drivers when it comes to positioning is to use the (Global Positioning System) GPS, but the provided accuracy and availability are not enough for ADS applications (KUUTTI et al., 2018). Consequently, it is necessary to use more sensors and reference maps to have complete positioning information using IMUs and odometry sensors combined with GNSS.

GNSS sensors communicate with a constellation of satellites to provide global positioning and is a more general term than GPS (European GSA, 2017). Communicating with the satellites may not always be possible, for example, when inside a tunnel or near high buildings.

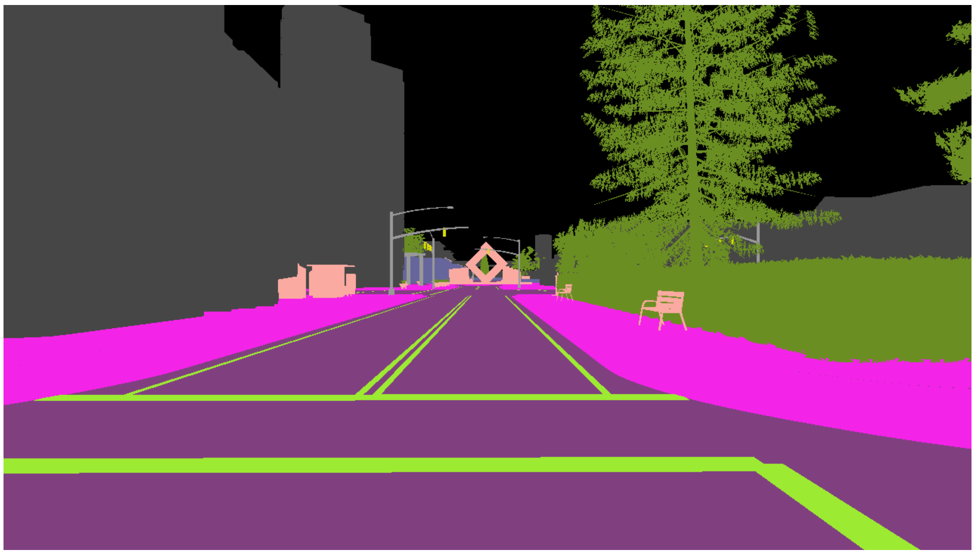
On the other hand, IMU sensor readings are always available because the sensor is composed mainly of embedded accelerometers and gyroscopes. Therefore, to estimate the position is possible to use the dead-reckoning method. It consists of periodically estimating the new position based on the last one using heading and distance sensors (KAO, 1991). Consequently, errors accumulate over time due to the estimation recursive nature (KAO, 1991; LIU et al., 2020). Concerning the wheel odometry sensor, it is a sensor for measuring the traveled distance. It can improve the dead-reckoning results as, otherwise, this distance information would be calculated based on accelerations measured by IMUs.

Even using results from GNSS, IMU, and wheel odometry altogether, as ADS performance requirements are very demanding, it may not be enough. A mapping technique as Simultaneous Localization and Mapping (SLAM) or a priori map-based localization is required for ADS. The first generates the map while localizing the vehicle in this same map simultaneously, which currently is very resourcing intensive for outdoor environments. The latter is about continuously localizing the vehicle in a map generated previously, which can be achieved, for example, using landmark search or point cloud matching techniques (YURTSEVER et al., 2020).

2.1.2 Perception

It is about gathering enough information through processing sensor data to build a description of the vehicle's surroundings. There are object detection, semantic segmentation, road/lane detection, and object tracking between its tasks (YURTSEVER et al., 2020). Object detection is responsible for detecting the size and location of objects of interest. At the same time, semantic segmentation classifies every pixel of images with class labels (YURTSEVER et al., 2020). Figure 3 illustrates this process as provided by a specific sensor in the CARLA simulator (CARLA, 2021). Road and lane detection goes beyond classifying the lanes in images. It is also responsible for understanding the road semantics and the relationship between lanes and intersections (YURTSEVER et al., 2020).

Figure 3 – Semantic Segmentation Example



Lastly, object tracking can be defined as "the task of estimating over time the state of a single or multiple objects based on noisy measurements received from one or several sensors" (KREBS; DURAISAMY; FLOHR, 2017). The obtained information complements the perception and, along with motion models, can be used to predict trajectories of surrounding dynamic objects (YURTSEVER et al., 2020).

Regarding perception sensors' limitations, Table 1 compiles some of them by kind of sensor. All sensors have limitations, but one can cover other weak points when using sensors with different strengths. In addition to different kinds of sensors, one may also combine variations of the same kind. Figure 2 illustrates, for example, the use of lidar variations, one on the top of the vehicle (with 360º view), and others at the front and lateral.

Table 1 – Perception sensors limitations

Sensor	**Limitations**
Camera	Highly affected by weather/luminosity conditions (YU; MARINOV, 2020);

Radar	No color differentiation;
	Low resolution (YU; MARINOV, 2020);
	Highly affected by reflections in walls (KIM et al., 2018);

Lidar	No color differentiation; Highly affected by weather/luminosity conditions (YU; MARINOV, 2020); Expensive when compared to others;

Sonar	No color differentiation;
	Only short range (YU; MARINOV, 2020);
	Low resolution (YU; MARINOV, 2020);
	Disruptions in noisy environments (YU; MARINOV, 2020);

2.2 Redundancy

In addition to each sensor's characteristics, the whole sensor set should work collaboratively to achieve fault tolerance and effectiveness. Redundancy, in this case, is not a negative point. Quite the opposite, often it is needed to provide some alternative to misreadings or unavailability of some other sensor.

Between sensors, redundancy is not necessarily replicating a sensor of the same type (e.g., more cameras). Perception tasks, for example, often have their results improved using different sensor types. The vehicle in Figure 2 uses multiple sensors of a same type, but it also has at least three types of sensors (radar, camera, and lidar). There are, inclusive, situations of using the same sensor for different ADAS (Figure 1) or multiple sensors by a single ADAS. Table 2 shows how some individual perception tasks can have their rating improved by using a combination of sensors. These results can be achieved with **multisensor data fusion**.

Table 2 – Perception Features vs Sensors

Legend: Good (G); Fair (F); Poor (P)

	Camera	**Radar**	**Lidar**	**Ultrasonic**	**Radar + lidar**	**Lidar + camera**	**Radar + camera**
Object detection	F	G	G	G	G	G	G
Object classification	G	P	F	P	F	G	G
Distance estimation	F	G	G	G	G	G	G
Object-edge precision	G	P	G	G	G	G	G
Lane tracking	G	P	P	P	P	G	G
Range of visibility	F	G	F	P	G	F	G
Functionality in bad weather	P	G	F	G	G	F	G
Functionality in poor lighting	F	G	G	G	G	G	G
Cost	G	G	P	G	P	P	G
Production readiness	G	G	P	G	P	P	G

Source: Adapted from (BURKACKY et al., 2018)

3 Data fusion

Understanding the general concept of *Data Fusion* is part of understanding *Sensor Data Fusion*. It is about fusing, that is, combining data. When introducing the Joint Directors of Laboratories (JDL) model, the definition by Hall (HALL, 2001) is much more precise: "Achieve a consistent, comprehensive estimate and prediction of some relevant portion of the world state. In such a view, data fusion involves exploiting all sources of data to solve all relevant state estimation/prediction problems, where relevance is determined by the utility in forming plans of action." (HALL, 2001).

For data fusion, the critical point arises at "exploiting all sources of data". Data fusion, most of the time, is about **combining data from different sources**, pursue relevant information to understand (estimate or predict) an environment (a portion of the world state) and, finally, make decisions (plans of action (HALL, 2001)).

3.1 Reasons to fuse data

3.1.1 Fault tolerance

Fusing data from sensors that can provide the same information can generate redundancy, which, as discussed in section 2.2, can provide fault tolerance in case of misreadings or unavailability of the other sensor (e.g. GNSS in tunnels).

3.1.2 Increase data quality and reliability

Fusing data from multiple independent sensors can improve spatial resolution (RAOL, 2009) and, when using different kinds of sensors, the system becomes less vulnerable to interferences. Some data sources, like sensors, can compensate for faults in the collecting process of another one, enhancing the confidence in the inferred results.

3.1.3 Estimate unmeasured states

For some reason, like a physical limitation, a state may not be directly measured. However, it might be possible to measure this state through the measurement of another one, followed by applying some mathematical or dynamics model.

3.1.4 Increase coverage area

Each sensor has a limited range (see Figure 1), but using more than one and fusing their information may lead to a sum of their ranges if they have different positions (or rotations), see lidars and radars from Figure 2.

3.2 Joint Directors of Laboratories Model

The JDL data fusion model was proposed by the Data Fusion Working Group from JDL and emerged from the need to unify terms related to data fusion (HALL, 2001). From the time that preceded the formalization (1986), data fusion studies from researchers and system developers from different study fields were following different paths. In this context, JDL was conceived as a general and functional model without application field restrictions.

A functional model is not meant to define specific inputs, outputs, or rigid order of execution like a process model (STEINBERG; BOWMAN; WHITE, 2008). It just provides a visualization of the whole and defines what should be implemented. Even with that, visual representations of the JDL data fusion model tend to cause some confusion in this aspect (see Figure 4) and gives the idea of a pipeline, which is not the case.

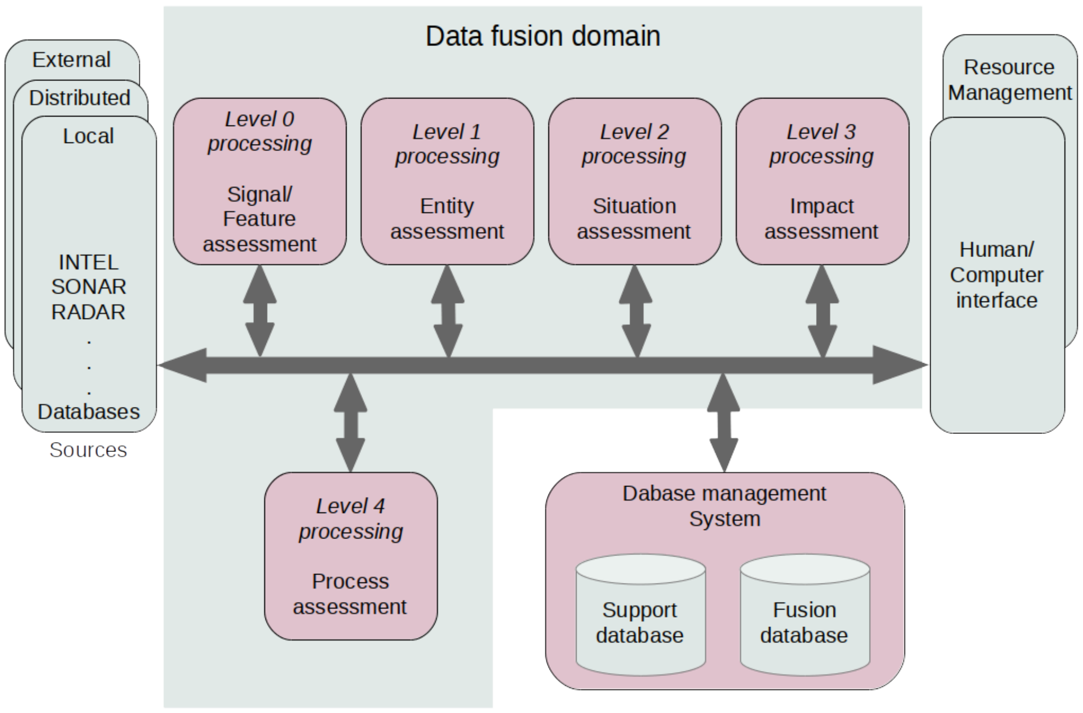
The initial version of the model could not solve all the problems it was meant to solve. Consequently, multiple posterior revisions were made to address problems with terminology, layers responsibilities, and even some definitions.

Moreover, the data model by JDL provides a holistic vision of the scenario in which the data fusion process is inserted. It describes steps (layers) that go from the raw signal input, or the data sources, to the computer-human interface, which could also be actuators.

Observing Figure 4 and considering the revised model (STEINBERG; BOWMAN; WHITE, 2008): *Level 0* handles the raw signal, with *Level 1* entities are already available, *Level 2* is capable of identifying relationships between these entities, composing situations, while *Level 3* is concerned about how possible decisions based on the new readings will impact the external scenario. It is even predicted a dedicated module to check how the whole process performs and optimize this performance, *Level 4*.

However, even though the JDL data fusion model is an essential reference for data fusion, it is a high-level specification and not a detailed architecture on how to implement Data Fusion concepts for specific use cases.

Figure 4 – JDL Recommended Revised Model



Source: Adapted from (STEINBERG; BOWMAN; WHITE, 2008)

4 Multisensor Data Fusion

Electronic devices interact with the physical world using sensors and actuators. Sensors generate measurements, and usually, these measurements are made periodically with fixed intervals defined by their Original Equipment Manufacturer (OEM). Consequently, there is a constant flow of data being generated by different sources (sensors) while the electronic device is turned on.

Following the ideas from Section 3.1, the information obtained from the sensors' data can be increased (or improved) once they are fused between them, i.e., once the *Multisensor Data Fusion* is applied to the data. This kind of fusion can be accomplished in multiple different ways, with different algorithms, and through different levels of abstraction.

4.1 Environment Model

In ADS, many sensors collaborate to understand the surrounding environment by completing tasks related to perception functionality. Even with capable sensors, and a well-defined hardware platform, there is still a need to create a software approach to make this information available to all modules that need it to make decisions. The environment model is an alternative to achieve this goal.

4.1.1 Directly accessing sensors

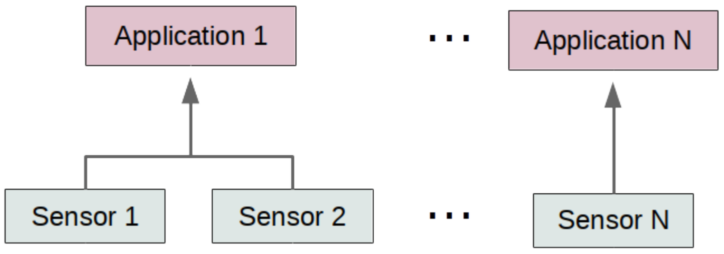
Modern vehicles (not only autonomous) may have several applications requiring data from specific sensors to perform their functions. Figure 5 shows the scenario using a basic architecture with a naive and direct approach. Each ADAS application gets data from the sensors it needs in this scenario, performing fusion whenever necessary. This kind of architecture, among other problems, can lead to high complexity code, high redundancy, and high bandwidth usage (AEBERHARD, 2017).

4.1.2 Collecting sensor data from Environment Model

Contrasting with the last architecture, architectures with *environment model* concentrates all data fusion inside a specific module. This module will produce a single description of the environment (AEBERHARD, 2017).

Assuming the idea that the state of an environment can be decomposed at the states of the objects (entities) in that environment (STEINBERG; BOWMAN; WHITE,

Figure 5 – ADAS directly collecting data from sensors

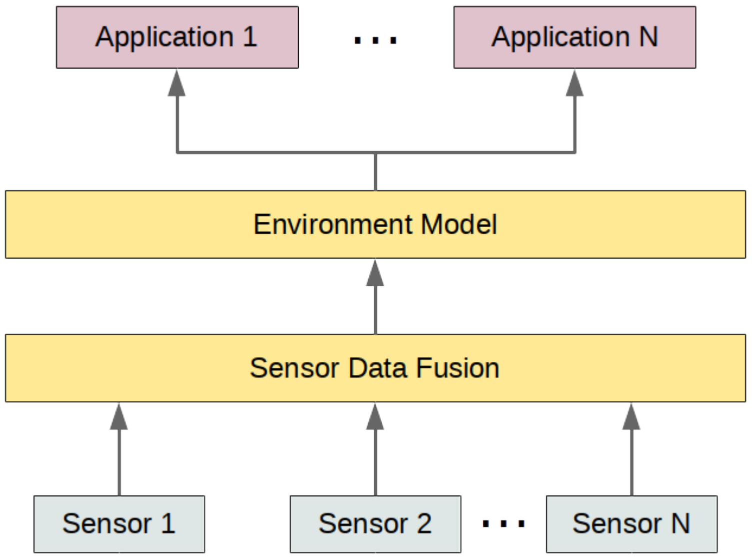


Source: Adapted from (AEBERHARD, 2017)

2008), from the environment model, it is possible to estimate the whole "world state" for that environment.

Thus, ADAS can directly get all information about the surrounding state from the environment model (see Figure 6). All sensors' complexity and specific details are transparent to the applications.

Figure 6 – ADAS collecting data from environment model



Source: Adapted from (AEBERHARD, 2017)

4.2 Architectures

There are several variations in organizing the flow of data inside a software application before they can be finally fused. This process can become a complex task as the number of sensors generating data and ADAS consuming that data increases. Predefined specifications for organizing this scenario can be called *architectures* for multisensor data fusion.

Three basic architectures are initially described by Hall (HALL, 2001): from least to most abstraction: *(i) direct fusion*, *(ii) fusion of feature vectors* and *(iii) fusion of decisions*. The first is about fusing data as it comes from the sensors. For the second approach, feature extraction algorithms are first applied to data, and then the results are fused. The last approach is about processing the data until much-derived information is extracted, and then they are fused.

Aeberhard categorizes these and other variations of this approaches in classes with different levels of abstraction: *low-level*, *high-level* and *hybrid*.

4.2.1 Low-Level

The most basic form of low-level fusion, *(i)*, is also the most intuitive. Essentially, a module will get data directly from the sensors, process this data, fuse it and return it to other modules, a completely centralized approach. Nevertheless, some other alternatives are still low-level, as the fusion of feature vectors, *(ii)*, can also fit this category.

4.2.2 High-Level

The opposite of the last one, this type of architecture depends on the fact that each sensor will have its data filtered and tracked by specific modules. When data gets to the fusion module, its responsibility is to combine them. Because each sensor will have separated filtering and tracking modules, this can also be considered a decentralized type of architecture. Since the input for the fusion itself are outputs from tracking algorithms (tracks), usually algorithms categorized as *track-to-track* are used. From the first mentioned three types of architecture, the fusion of decision, *(iii)*, fits here.

4.2.3 Hybrid

A hybrid fusion architecture is an attempt to aggregate the advantages of both other options. However, it comes with the risk of the architecture becoming too complex for real-world scenarios.

Table 3 is a comparison table by Aeberhard, who does a great job explaining each of these evaluated criteria in detail.

The most noticeable points are: first, about the unique positive point for *Low-Level* being accuracy. Accuracy for the other options can be decreased due to model assumptions, even for Hybrid, so applications that cannot tolerate this kind of loss should keep using a *Low-Level* approach. Considering only the positives and negatives, *High-Level* appears to be the best. However, each of the evaluated points needs to be evaluated specifically for the scenario in which they will be applied. It is also important to note that variations in

Table 3 – Architectures Comparison

Legend: Positive (P); Negative (N); Neutral (T)

	Low-Level	**Feature-Level**	**High-Level**	**Hybrid**
Sensor encapsulation	N	T	P	T
Bandwidth	N	T	T	T
Accuracy	P	T	T	P
Complexity	N	T	T	T
Modularity	N	T	P	T
Sensitivity to errors/bias	N	N	T	T
Practicality	N	T	P	T

Source: Adapted from (AEBERHARD, 2017)

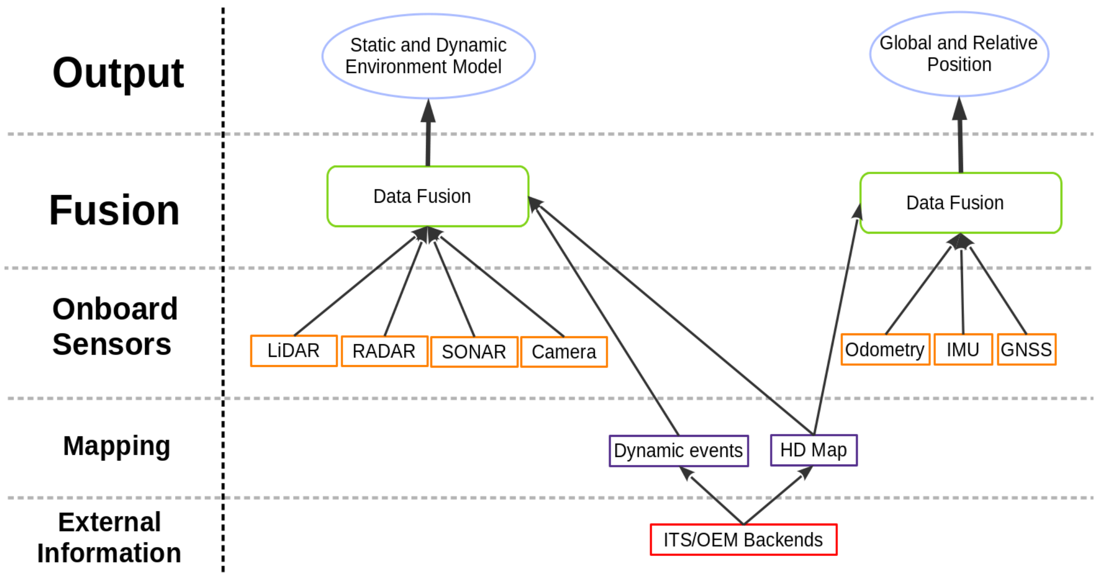
the implementations of these architectures can also lead to better or worse performance than the registered in the table.

4.3 Self-Driving Vehicle Sensor Fusion

For multisensor data fusion for self-driving vehicles, data must be fused to get the best possible results for both the core functions detailed in section 2.1, which gets data from the vehicle sensors *perception*, and *localization and mapping*.

More objectively and considering all the information from the other sections, Figure 7 abstracts *perception* as *Static and Dynamic Environment Model* and *localization and mapping* as *Global and Relative Position*. Mainly, ADS requires distinct fusion modules for each of these goals, collecting data from the indicated data sources, that may not always be physical sensors.

Figure 7 – Self-Driving Vehicle Sensor Fusion



4.4 Algorithms

Depending on the chosen architecture and hardware platform, many different types of algorithms can be used, starting with filtering algorithms that can be applied to the raw sensor data, to artificial intelligence ones to make decisions in the planning step.

To cite some examples, Bayes filters, like Kalman Filter variations, usually are enough to obtain dynamics information (YURTSEVER et al., 2020). For filtering and feature extraction, it depends on the target sensor, e.g., from Fourier Transforms to YOLO, and Deep Convolutional Neural Network (DCNN) might be applied to cameras data. Furthermore, from basic statistics to Evidence Theory can be applied to merge data, calculate the existence probability of elements, control weights based on the reliability of the data, and others.

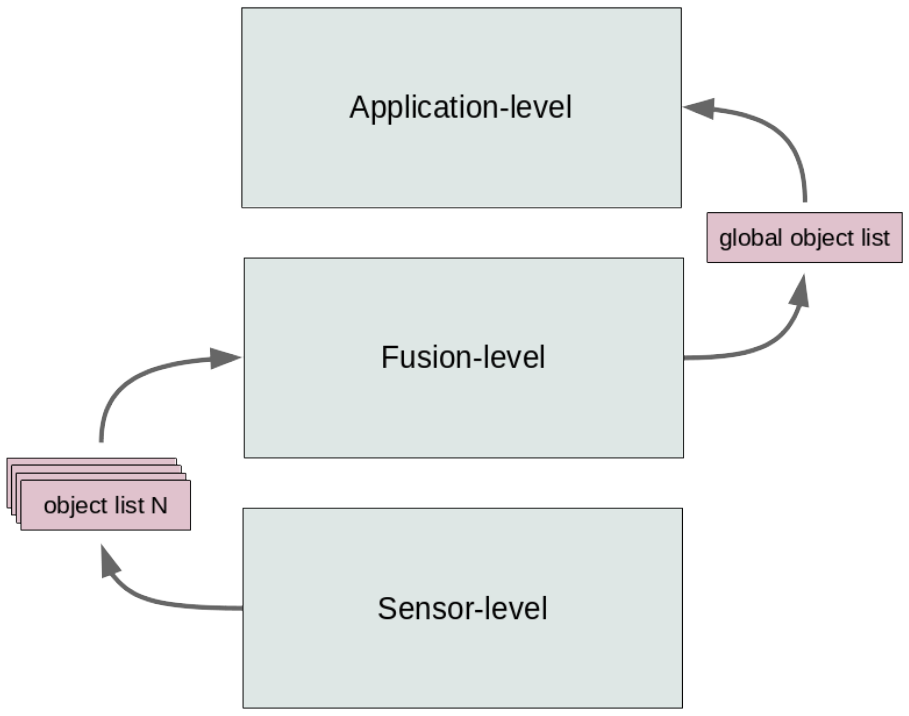
Using four main kinds of problems, Khaleghi groups algorithms: data imperfection, data correlation, data inconsistency, and disparateness (KHALEGHI et al., 2013). Besides the kinds of problems, algorithms may also be classified between frameworks as probabilistic, evidential, fuzzy reasoning. Despite these classifications, algorithms are chosen based on details of the chosen approaches for each step of this whole ecosystem.

5 Object-Level Fusion Architecture

To estimate the Dynamic Environment Model (see Figure 7) for developing ADAS applications, Aeberhard (AEBERHARD, 2017) proposed a novel architecture for fusing data from perception sensors (see subsection 2.1.2). This chapter is dedicated to giving an overview of such architecture, a central point of this work.

As seen in section 4.2, there are multiple different architectures to fuse data from multiple sensors. Aeberhard proposes a hybrid one, mainly based on high-level architectures, but that eventually uses low-level or feature-level fusion techniques for specific situations. The proposed architecture is essentially a layered architecture with three layers (levels), the *sensor-level*, *fusion-level* and *application-level* (see Figure 8).

Figure 8 – Aeberhard Proposed Architecture Overview



Source: Adapted from (AEBERHARD, 2017)

Considering this architecture, as necessary as the levels (layers) definitions are the interfaces definitions between these layers. The Object Model defines these interface definitions, modeling relevant information about each detected object, both between sensor-level and fusion-level, and fusion-level and application-level.

The following sections provide a better understanding of the functioning and responsibilities of all elements of this architecture. Moreover, due to the preprocessing steps implemented in this work being part of the fusion-level, it is described better.

5.1 Object model

The object model is the interface between the layers of the proposed architecture. Hence, it needs to carry all relevant information about all the detected dynamic objects. It was modeled as a list of objects in which each object is a structure with information about its track, dimensions, existence probability, classification results, and shape features.

Track is composed of two attributes, the estimated state vector and its covariation matrix. Similarly are the object dimensions, it also contains the dimensions themselves and another attribute to hold the variances. From left to right in Equation 5.1, the state vector $\boldsymbol{x}$ is composed by: position in x, position in y, velocity in x, velocity in y, acceleration in x, acceleration in y, yaw angle and yaw rate. Dimension $\boldsymbol{d}$ (see Equation 5.2) is composed only by the length and width of the object.

$$\mathbf{x} = [x, y, v_x, v_y, a_x, a_y, \psi, \dot{\psi}] \tag{5.1}$$

$$\mathbf{d} = [l, w] \tag{5.2}$$

Classification is the output of a classification algorithm for each of the classes: *car, truck, motorcycle, bicycle, pedestrian, stationary* and *other*. Each sensor will run feature descriptor algorithms to extract features from its measurements and then run classification algorithms to fill this vector.

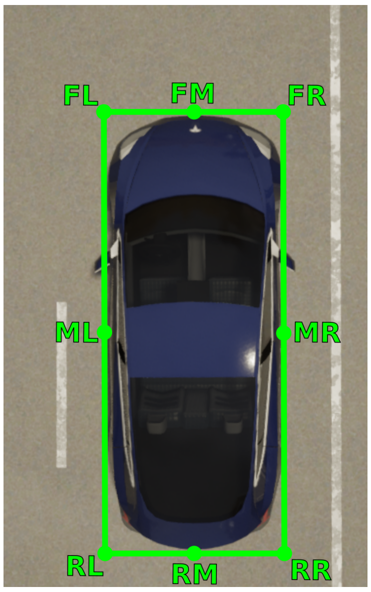
Lastly, shape features vary according to the choice of how to represent an object's shape. The reference work uses rectangular shapes, with each feature representing one point of the rectangle. As illustrated in Figure 9, the features are the relevant combinations of *rear (R), front (F), middle (M), left (L),* and *right (R)*: *FL, FR, RL, RR, FM, RM, ML and MR.* Each of these points is a boolean value, representing if the sensor detected that position of the object in the current reading. It is also worth noting that if considering three dimensions, the shape features, object dimensions and the state vector could be adjusted accordingly.

Despite the Object Model supporting all these data, not all sensors can fill every single attribute. Therefore, a software implementation using these definitions should also provide some mechanism to specify which attributes a sensor is responsible for filling and which ones it cannot.

5.2 Layers

Aeberhard proposes the three processing levels that are abstracted as layers in this work. It is common to design software architectures based on layers to retain specific

Figure 9 – Shape Features



responsibilities to each layer. When layering, the information should flow only from bottom to top, from the least to the most abstraction, keeping the layer on the top unaware of the layers below.

Between the advantages of this approach, each layer can be understood as a whole coherent module, and it can be modified without prior knowledge about other layers. On the other hand, there can be performance degradation caused by transformations between layers, and it is not always possible to keep the higher layers unaware of lower ones (FOWLER, 2002). The following subsections highlight some of the proposed architecture's layers characteristics.

The proposed architecture meets these requirements for layered architectures. Each processing level can be considered a layer, with self-contained and well-defined responsibility, also, the information, object lists, go only from lower layers to higher layers (see Figure 8).

5.2.1 Sensor-Level

The sensor-level is the base layer, the least abstracted. It communicates directly with the sensors, and its responsibility is to, for each sensor, transform the sensor's raw data into the object list, filling the attributes from the object model according to the sensor capabilities.

Eventually, better results may be achieved by performing low-level (or feature-level) fusion in this layer. With this approach, data from more than one sensor would be fused to generate one object list, and the group of used sensors would be called a *virtual sensor*. Performing low-level fusion at this level is the reason that makes the proposed architecture a hybrid one.

The transformation of the raw data to the object list includes the steps of *feature extraction, existence estimation, tracking* and *classification*. Moreover, there are several options of algorithms that could be used for each one of these steps. In this layer, the choice of algorithm and its implementations are also highly dependent on the selected sensors.

5.2.2 Fusion-Level

As illustrated in Figure 10, the layer receives all the object lists from the sensor-level and must fuse them into a single object list, the global one. The global object list represents the surrounding environment state at a given instant. The fusion is accomplished by performing the lists alignments (*spatial alignment* and *temporal alignment*), *object association*, and the fusion steps (*state and covariance fusion, existence fusion* and *classification fusion*).

The steps of alignment (spatial and temporal) plus the object association will be implemented in this work and will be called ***objects preprocessing***. The following subsections explain the main ideas about these processes.

5.2.2.1 Temporal Alignment

Usually, sensors do not generate measurements with the same frequency. Then, different object lists received in fusion-level do not represent the vehicle surroundings at the same instant, although probably close. The temporal alignment step addresses this by predicting the most recent measurement to the desired point in time.

In Figure 11, *sensor 1* generates measurements every time step, *sensor 2* every three and *sensor 3* every two. In the figure, the global measurement is intended to be generated at the instant t_k by predicting the last measurements of *sensor 1* and *sensor 2*. Without the prediction, only *sensor 3* would have measurement value.

5.2.2.2 Spatial Alignment

Spatial alignment refers to the process of bringing all spatial data from the object list to a common coordinate system, a unified spatial frame. As can be observed in Figure 12, this process is needed because each sensor may be at different positions and may have its axis rotated or reflected compared with another sensor. After the alignment, the objects' state must be relative to the same origin point (often in the middle of the vehicle rear-axis) and with positive x and y pointing to the same directions.

Figure 10 – Fusion-level overview

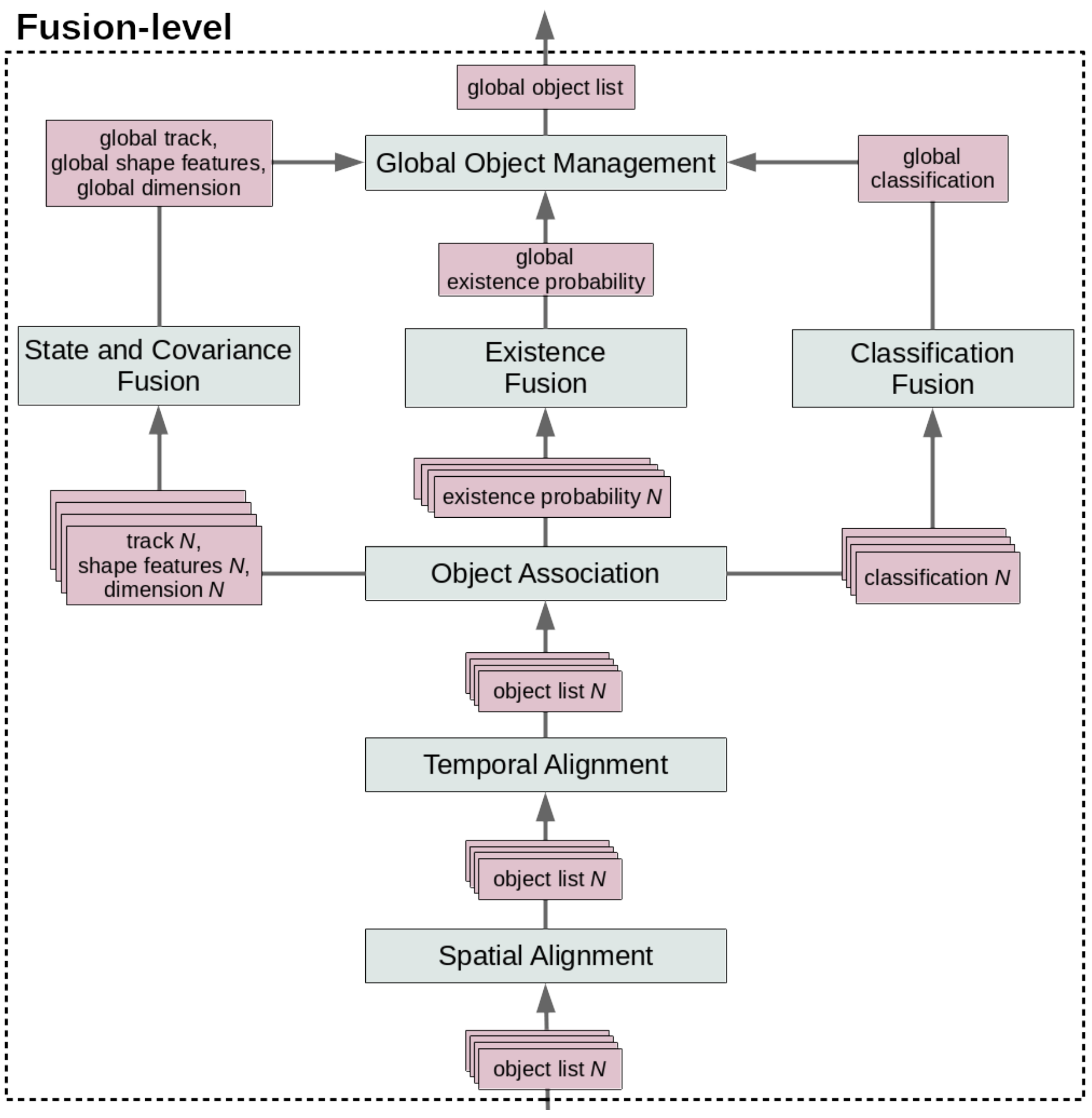


Figure 11 – Temporal Alignment

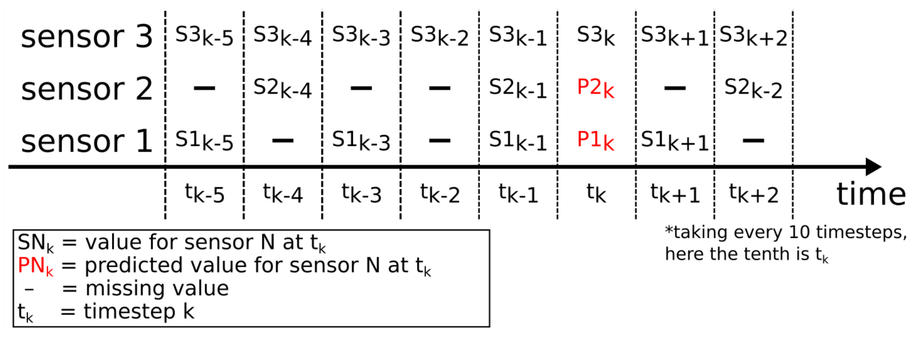
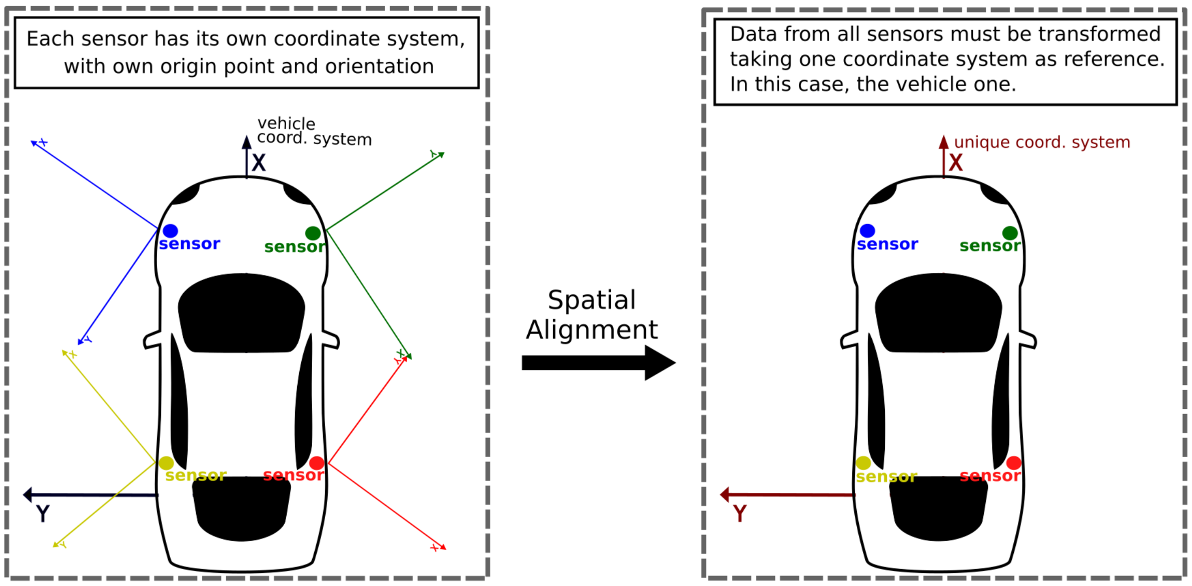


Figure 12 – Spatial Alignment


Each sensor has its own coordinate system, with own origin point and orientation
vehicle coord. system
X
Y
sensor
sensor
sensor
sensor
Spatial Alignment
Data from all sensors must be transformed taking one coordinate system as reference. In this case, the vehicle one.
unique coord. system
X
Y
sensor
sensor
sensor
sensor

5.2.2.3 Object Association

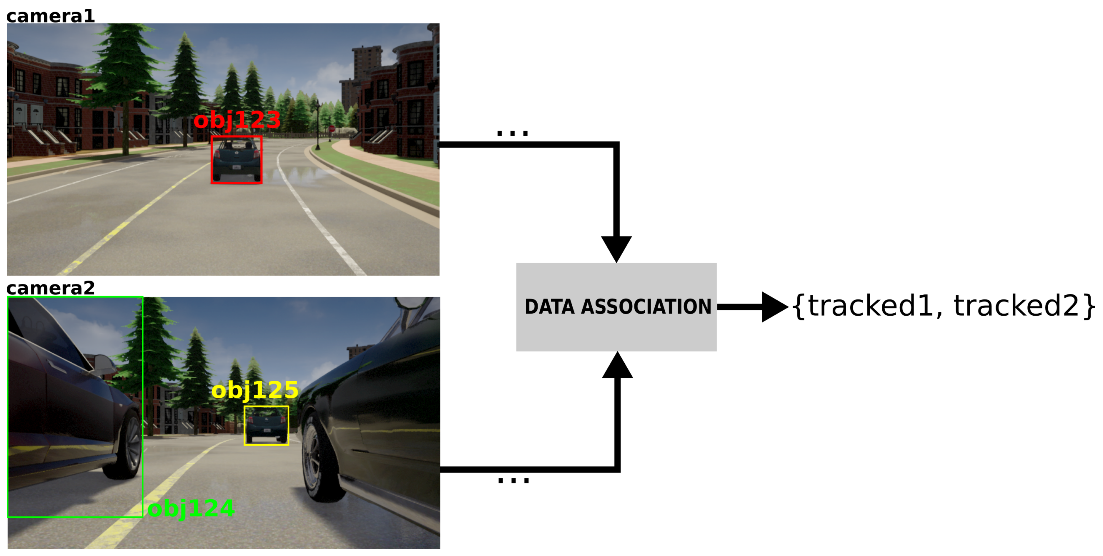
Data association is the process of identifying which pieces of information, throughout data from multiple sensors, refers to the same object (BEHERE; TORNGREN, 2015). In this layer, these pieces of information are the objects from different object lists.

Consider a scenario in which a car has two cameras, one on the hood and the other is lateral. The lateral one can visualize two cars (and the ego), while the other can visualize just one, as shown in Figure 13. Suppose also that only *obj123* is already known and is being tracked as *tracked1* and, therefore, the tracked objects is the set *{tracked1}*. After the object association, both cameras will recognize *obj123* and *obj125* as *tracked1*, and *obj124* will begin to be tracked as *tracked2*. Consequently, the tracked objects will become the set *{tracked1, tracked2}*.

5.2.2.4 Fusion

After the association, the fusion itself is performed in three steps *state and covariance fusion*, *existence fusion* and *classification fusion* (see Figure 10). The first is handled in two parts, the *track-to-track fusion*, which is solved using an information matrix fusion algorithm, and *geometry fusion* solved with geometrical fusion applied to a parallel state vector. For existence and classification, are proposed different novel approaches using Dempster-Shafer Theory (DST).

Figure 13 – Object Association



5.2.3 Application Level

In this layer, the global object list can be used to implement variations of ADAS, even ADS. However, depending on the application that is going to be implemented, more sources of data and algorithms may be needed. The fusion layer only provides the list of surrounding dynamic objects (with all the information presented in the object model), taking the ego-vehicle as the reference.

5.3 Fusion Strategy

Two typical fusion approaches could be adopted for fusing the object lists generated by the sensor layer, *sensor-to-sensor* and *sensor-to-global*. The first is mainly based on cycles, and for each cycle, the lists from each sensor would be fused between them, generating a global list. With this method, a temporary list is maintained for the current cycle, and when a new list arrives, it is merged to the temporary one until the cycle is completed and the temporary list becomes the global one. It can be said that this approach is memory-less, as the subsequent cycles do not consider the global lists from the previous ones (AEBERHARD, 2017).

By contrast, with the *sensor-to-global* approach, which is the approach chosen in the proposed architecture, whenever that any sensor generates a new list, it is fused to a global object list right away. This global object list is maintained while the application is running, and there are no cycles or completely renewing the global list. An advantage of this approach is that a recently generated list is not buffered or keeps waiting for other sensors, making the solution more flexible to removing or adding sensors. Since the global object list is not completely replaced, it considers previous values, and it can be said that this approach has memory.

It is also important to note that both options require temporal alignment (see Subsubsection 5.2.2.1). The current global list, in *sensor-to-global* case, or the temporary list, in *sensor-to-sensor* case, must be predicted in time whenever any sensor generates a new list.

6 Robot Operating System (ROS)

ROS 2 is the second generation of ROS, a modular, open-source framework for software development for robots (ROS, 2020c; Red Hat, 2020). It is not an operating system like a Real-Time Operating System (RTOS) or Linux. It is described as a meta-operating system that runs on top of another operating system providing its level of abstractions, tools, libraries, conventions, inter-process communication, and packaging system (ROS, 2020b). **In this document, ROS 2 is assumed as the default version of ROS and will be called just *ROS***. When referring to the first generation, *ROS 1* will be used instead. The following sections contain more information about specific points in the use of ROS.

6.1 Applications Architecture

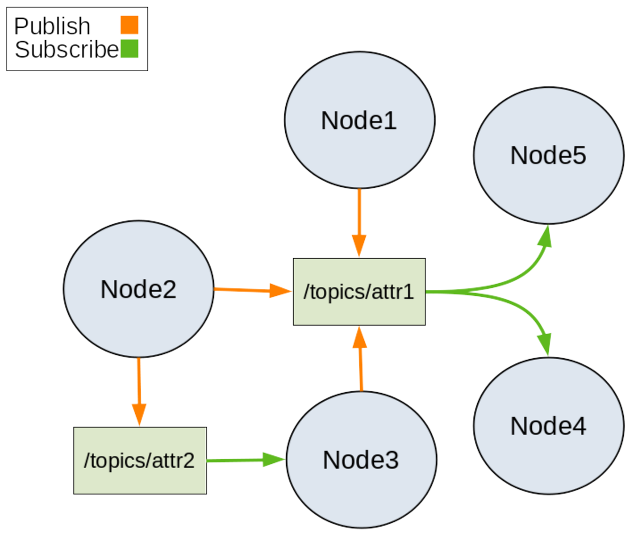
Systems developed on ROS are entirely distributed and decentralized, with no central server or broker. It forms a mesh of *nodes* that can communicate between them mainly through the use of publish/subscribe (pub/sub) pattern, and also request/response using *services* and *actions*. With a suitable Quality of Service (QoS) configuration, this structure can make these applications achieve space, time, and synchronization decoupling (EUGSTER et al., 2003).

6.1.1 Publish-Subscribe

Essentially, the publish/subscribe communication pattern (or paradigm) consists of *subscribers* capable of demonstrating interest in specific events, and *publishers* capable of generating these events. Once an event is generated, all subscribers registered will be notified, which forms a full decoupled interaction (EUGSTER et al., 2003). This pattern can be implemented in many forms. It is common to have *topics* and *messages*, messages are units of data published to topics, and then, all subscribers receive these units. ROS abstracts this behavior with the use of *nodes*, which are the entities capable of publishing and subscribing.

Figure 14 illustrates this pattern. As examples, *Node5* and *Node4* will receive all messages published by *Node1* and *Node3*, that are publishers only from */topics/attr1*. *Node3* may receive messages from *Node2* through topic */topics/attr2* and *Node2* is also able to send messages to *Node5* and *Node6* through */topics/attr1*.

Figure 14 – Publish/Subscribe



6.1.2 Elements

Understanding the main elements which compose ROS applications also makes its functioning clearer. These elements are:

- **Nodes:** ROS abstraction for processes. Each node should have one responsibility. Nodes can communicate with the rest of the system by publishing messages, subscribing to topics and making requests to services and actions.

- **Topics:** Receive messages published to them and then forward to nodes subscribed to them. A path separated by slashes identifies topics (e.g. *house/kitchen/temperature/1*), and this structure can be used to abstract composition or hierarchy, for example.

- **Messages:** Data transmitted between the nodes. A message must have a well-defined type that defines its serialization. New types can be created by composing existing ones or from primitive types.

- **Services:** Another option to communicate between nodes, but instead of pub/sub, services follow a request-response synchronous model. A node makes a request and keeps waiting until the response arrives.

- **Actions:** One more option for communicating between nodes, similar to services, and follows the request-response model, but asynchronously. A request is made (usually long-running), and while it is being executed, periodic feedbacks are returned. Additionally, the action can be canceled before it ends.

- **Parameters:** Similar to global or environment variables inside a node. They can be kept in a file and have their value changed dynamically during the application execution.

- **Bag:** ROS feature that enables to record data published to topics and play it back. Useful to repeat experiments, debugging or mocking components in the system.

6.2 Packages

One of the advantages of using ROS is to have access to easily pluggable, open-source packages from the community. Currently, in May 2021, including all versions of ROS, there are almost 2300 repositories with more than 6300 packages available for usage and study (ROS, 2020a).

ROS officially provides resources for developing packages in Python and C++ programming languages, with all the packages being able to communicate with each other. Usually, ROS projects are composed of several packages which may have different dependencies and need to be built before the execution. A project usually is organized in a *workspace*.

Over time, some tools, as *ament* and *catkin*, were developed to address these problems of dependency resolving and project building. For the last ROS Long Term Support (LTS) version available, *Foxy Fitzroy*, the official tutorials recommend using *colcon* to build the packages and *rosdep* to resolve the dependencies (ROS, 2020).

These tools can complete such tasks by interacting with standard tools already well known by programmers, e.g., *CMake*, *apt*, and *pip*. All ROS packages must have a *package.xml* file in the repository root directory for this to be possible. Plus, or a *CMakeLists.txt* file for C++, or a *setup.py* and a *setup.cfg* files for Python.

6.3 Data Distribution Service (DDS)

DDS is a middleware protocol and Application Programming Interface (API) standard for Industrial Internet of Things (IoT), i.e., it is a specification for the software layer that will lay between the Operating System (OS) and the application, intermediating the communication and sharing of data. Some of the features it has are about the automatic discovery of the parts, QoS configurations, and security (DDS Foundation, 2020).

In the transition from ROS 1 to ROS 2, most of the middleware was replaced by a DDS implementation. Additionally, another abstraction layer has been built on top of DDS to keep interfaces and features similar to the previous ROS 1 version. This DDS layer can be configured or replaced by another DDS implementation.

Due to the use of DDS, ROS uses DDS Interoperability Wire Protocol (DDSI-RTPS) as the wire protocol for communication between the nodes. It is a protocol built, by default, on top of User Datagram Protocol (UDP) and provides some of the reliability of Transmission Control Protocol (TCP). However, in a highly flexible way, allowing it to be customized for the more diverse use cases through several QoS parameters (WOODALL, 2020).

6.4 Interfaces

At the user level, the interfaces to ROS are *messages*, *services* and *actions*. For messages, one data structure is transmitted, for services are defined one data structure to the request and another for the reply, and for actions, are defined request, reply, and feedback. ROS defines that these data structures should be specified in *.msg* files. These files are then converted to Interface Definition Language (IDL) (THOMAS, 2020), which can be used as a mapping to different programming languages code or even serialization (OBJECT MANAGEMENT GROUP, 2018).

6.5 RTOS

ROS 1, as a popular framework, has been extensively used for building modular prototypes for robot applications. In the transition to ROS 2, a decentralized approach through the use of DDS was designed and developed to make it more robust, meeting use cases not met before, e.g., real-time applications and production environments (GERKEY, 2020).

The most critical differences between an RTOS and a general-purpose OS are mainly regarding the execution of the tasks being deterministic and critical deadlines, i.e., missing a deadline is considered a severe fault. To accomplish this, RTOS usually provides mechanisms that give the developer better control over the execution of the system, as a scheduler with predictable behavior based on priorities for tasks (SIEWERT; PRATT, 2015).

Eventually, embedded software applications need to be run on top of RTOS, mainly when are used in safety-critical applications. Discussing which modules of an ADS should run on RTOS is not a goal of this work. However, as the slowness to perform some action can cause injuries or deaths, an RTOS will likely be used at some point. For example, as a time requirement, for localization estimation, it has been estimated a minimum of 10 Hz frequency (Connected Places Catapult, 2019). Therefore, it is essential to check the ROS capabilities regarding real-time applications as ROS might need to interact or run on top of an RTOS in a real-world scenario.

The improvement of these real-time capabilities was one reason for the migration from ROS 1 to ROS 2 (GERKEY, 2020). However, despite the intentions, these capabilities were not implemented since the beginning. Later, there were official (recent) discussions about the subject and plans to adapt the code base aiming real-time (PANGERCIC, 2019b). As a result, an official proposal was written (KAY, 2020b), yet it is not clear how much of it is already implemented.

On the other hand, for the real-time scenario, there are official instructions for using ROS with Linux using the kernel patch RT_PREEMPT. Additional specific configurations (ROS 2, 2020) and best practices (KAY, 2020a) are also provided. Some works had even met the requirements for their soft real-time applications by using similar approaches: (PARK; DELGADO; CHOI, 2020; GUTIÉRREZ et al., 2018). The latter even for firm real-time application. Not exhaustively, but like other options, there are also commercial solutions as the eMCOS RTOS, which claims to be directly compatible with ROS (eSOL, 2020) and Apex.OS, which can be used as a direct replacement for ROS (Apex.AI, 2020).

For highly constrained environments, it was released micro-ROS, a solution with seamless integration with ROS 2 architecture and ready to be used in microcontrollers (micro-ROS, 2020a). Between its adaptations is the use of the DDS For Extremely Resource Constrained Environments (DDS-XRCE) instead of the default ROS DDS implementation. Currently, for software platforms, micro-ROS supports Nuttx, FreeRTOS, and Zephyr RTOSs. The hardware platform also supports popular platforms, e.g., ESP32 (micro-ROS, 2020b).

6.6 Usage for Autonomous Driving

Autonomous vehicles are robots. They have sensors and actuators and, essentially, common requirements with other robotic applications. As ROS is a framework for developing software for robots, it is expected that it also can be used in ADS. However, autonomous driving is a very complex robotic application, one with critical requirements for security and safety, complex sensors, and complex decision mechanisms. Therefore, extra care must be taken when using ROS in this context, which might require some extra steps.

Nevertheless, ROS has been used in the autonomous driving scenario by companies like BMW (AEBERHARD et al., 2015) and Apex.AI, which has developed autonomous cars (HO et al., 2018) and studied ROS real-time performance (PANGERCIC, 2019a). In the academy, among other works, some researchers are proposing new software architectures (REKE et al., 2020) and others considering ROS in the test and validation process (KERNER, 2017). On Google Academic (2020 December), the query *"ROS" ("autonomous* *driving" OR "self-driving")* returns almost four thousand results since 2019. Besides, there are also big open-source projects for ADS using ROS, e.g., Autoware.Auto, currently focused on the *Autonomous Valet Parking* and *Autonomous Depot Maneuvering* use cases (The Autoware Foundation, 2020).

Part II

Results

7 Software Solution

Previous chapters contain studies about all of the aspects deemed relevant for delimiting this work's main characteristics and scope. This chapter describes how all this information was used to achieve the objective defined in Subsection 1.2.1.

The fusion architecture proposed by Aeberhard and explained in Chapter 5 is chosen for this solution. Its high-level characteristics, well-defined interfaces, and processing levels (mapped to layers in this work) make it possible, to a certain extent, to build a generic and modular software solution. Another important point is that it was designed to be used in the same kind of applications that this work, autonomous driving applications.

The focus of this work is the preprocessing in the entrance of the fusion layer, comprised by the **spatial alignment**, **temporal alignment**, and **object association**. The scope to implement the whole fusion layer would not fit the time available for this work.

It is also worth noting that implementing all three layers would require other long and more complex to generalize tasks. The sensor layer would require the processing and extraction of features from different kinds of sensors' raw data. In addition, the application layer would require to implement some application that consumes the global object list (e.g., autonomous driving). The fusion layer was considered the part of the architecture more suitable for a generic and modular software solution.

The software solution will use ROS as the main framework. ROS has been widely used for robot software, eventually even considered the *de-facto* standard (MALAVOLTA et al., 2020). It was designed to make feasible a structure where independent packages (modules), developed by different groups, could be integrated to compose a complete robotic application. Additionally, it has been used for autonomous driving applications (see Section 6.6). All of these aspects make it a proper tool to be used in this solution.

7.1 Requirements

Software requirements identify what is expected from the software once it is ready to be used. Among other possibilities, it is common to classify requirements between *functional* and *non-functional*. Functional requirements refer to what the software will do (i. e., functionalities), and non-functional requirements are about constraints on how to achieve such functionalities (e.g., performance, accuracy, security, interoperability) (WOHLIN et al., 2005).

When designing the software solution, the following functional requirements were gathered, based on the studies from Part I:

- For each sensor object list that arrives, a global object list must be published;
- The solution must allow the registration of sensors;
- The solution must allow the removal of sensors;
- The solution must allow the specification of sensors positions;
- The solution must allow the specification of which attributes from the object model a sensor is capable of generating.

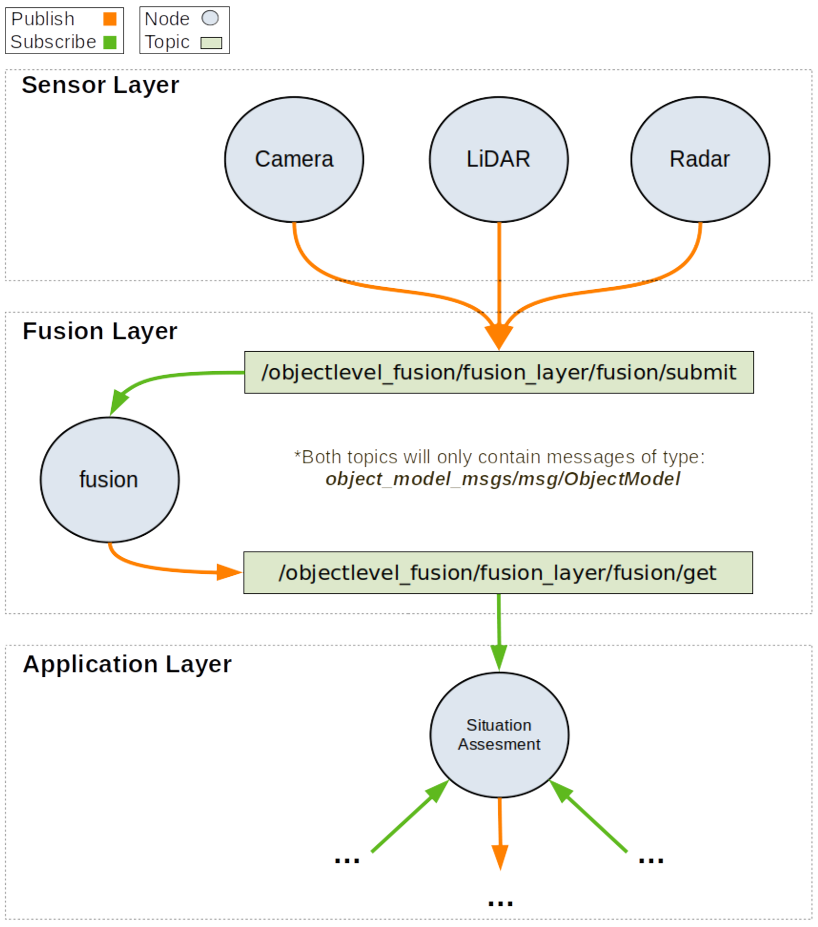
Additionally, the following non-functional requirements:

- The preprocessing implementation must be coherent to the layers specification (Figure 8);
- The solution must be compatible with ROS workspaces using ROS Foxy Fitzroy version;
- The interfaces must be independent of the specific type of sensor (e.g. camera, radar, lidar);
- The solution must be able to receive constant streams of object lists without the need to be restarted;
- The solution must be able to receive data from multiple sensors simultaneously;
- The solution must be executed independently from the application using it;
- The implementation should follow real-time best practices as much as possible.

7.2 Architecture

Figure 15 illustrates the publishing/subscribing from the three layers described in Section 5.2. The *fusion layer* entry is the */objectlevel_fusion/fusion_layer/fusion/submit* topic. It will receive the object lists produced by each of the sensors through publishes from the *sensor layer*. The preprocessing steps from the fusion layer (see Figure 10) will be implemented inside the *fusion* node and published to the */objectlevel_fusion/fusion_layer/fusion/get* topic. Nodes from the *application layer* must subscribe to this topic to receive the global object lists. In the figure, nodes from the sensor and application layers are just an example of integration for better understanding, and different configurations can be adopted.

Figure 15 – Solution Architecture Overview


Publish
Subscribe
Node
Topic
Sensor Layer
Camera
LiDAR
Radar
Fusion Layer
/objectlevel_fusion/fusion_layer/fusion/submit
fusion
*Both topics will only contain messages of type:
object_model_msgs/msg/ObjectModel
/objectlevel_fusion/fusion_layer/fusion/get
Application Layer
Situation Assesment
...
...
...

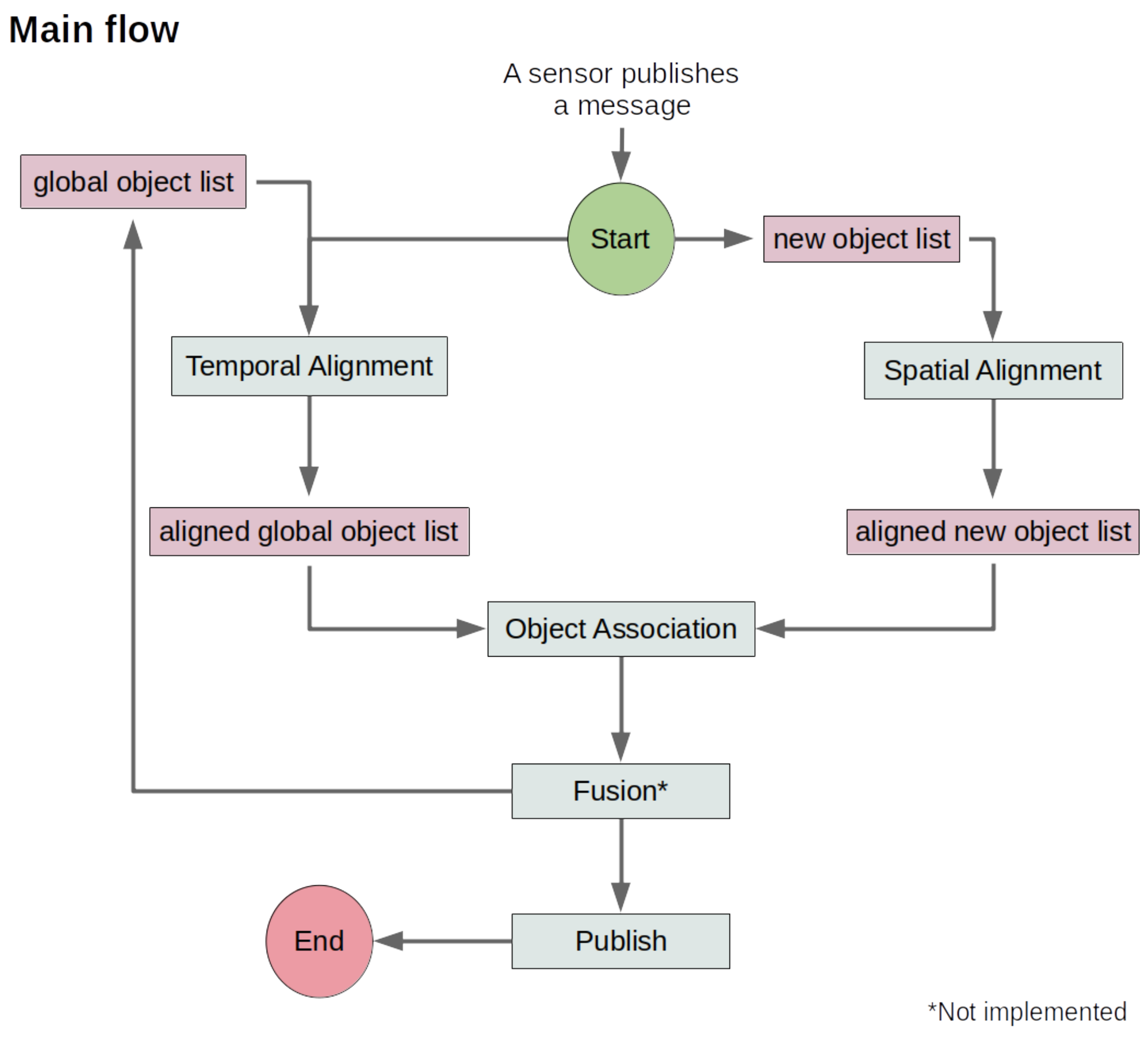
It is also worth mentioning that the fusion strategy chosen by the used fusion architecture (Chapter 5) is sensor-to-global (Section 5.3). Therefore, an object list from a sensor is fused to the global state as soon as it is submitted. It is **not buffered** until receiving lists from all the sensors, i.e., a global object list will be published whenever any object list from any sensors is published.

7.3 Fusion Node

The ideal is that all steps from Figure 10 would be implemented in the *fusion node*, but it would be a scope too big for this work. Thus, the grouping of *spatial alignment*, *temporal alignment*, and *object association* is being named **object preprocessing**, and it is the scope to be implemented in this work. The name preprocessing is because they are the steps to be performed before the fusion steps.

Following the definitions from Chapter 5, the fusion module will start the flow indicated in Figure 16 whenever a new object list is published by any sensor node in the sensor layer. By the end of the flow, the global list will have incorporated all the objects from the new list that arrived and will be up to date.

Figure 16 – Fusion Node flow



7.3.1 Sensor Registration

The fusion layer must know the attributes from each sensor in the sensors layer to perform some of the calculations, e.g., sensor position to perform the spatial alignment. These attributes are provided through the registration of the sensor.

In this solution, the registration is performed by calling ROS services. There are two services, one for adding a sensor and the other for removing it. Sensors not registered will have their messages ignored by the fusion module, and a sensor can be added or removed at any time during the application execution.

The attributes considered in the registration are the sensor name (arbitrary), position (x and y), yaw angle, which attributes from the object model it is capable of providing (as a boolean array), and the measurement noise matrix (see Source Code B.1). The measurement noise matrix is a covariation matrix about the errors expected, from that sensor, for each attribute from the object model. Moreover, all the attributes about the position are relative to the ego vehicle.

7.3.2 Spatial Alignment

When arriving in the fusion node, the values in the object list are relative to the sensor that sent the list. The objective of the *spatial alignment* is to make them relative to the ego-vehicle position (see Figure 12). In this solution, as defined in (AEBERHARD, 2017), this procedure is performed by applying a transformation matrix to the state vector.

The transformation matrix is as Equation 7.1. The $\pm\Delta x$ and $\pm\Delta y$ are the x and y provided in the sensor registration, and θ is the yaw angle, also provided in the same way. The matrix is applied to the state vector as Equation 7.3, and the aligned state will have its reference frame the frame from the ego vehicle.

$$H = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 & 0 & 0 & 0 & 0 & \pm\Delta x \\ \sin(\theta) & \cos(\theta) & 0 & 0 & 0 & 0 & 0 & 0 & \pm\Delta y \\ 0 & 0 & \cos(\theta) & -\sin(\theta) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sin(\theta) & \cos(\theta) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cos(\theta) & -\sin(\theta) & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sin(\theta) & \cos(\theta) & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \theta \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.1}$$

$$state = \begin{bmatrix} x \\ y \\ v_x \\ v_y \\ a_x \\ a_y \\ \psi \\ \dot{\psi} \\ 1 \end{bmatrix} \tag{7.2}$$

$$aligned\_state = H \cdot state \tag{7.3}$$

7.4 Temporal Alignment

In (AEBERHARD, 2017), it is used only the prediction step from a simple Kalman Filter, with a constant velocity model, to perform the temporal alignment from the track (state vector and its covariance). The same work also suggests using a more complex model with Extended Kalman Filter (EKF) and, to attempt to get better results, this is the approach adopted in this work.

This solution uses a model with a constant turn rate and constant acceleration in an approach similar to (BALZER, 2021). To avoid modifications in the known equations, the object model state is converted to Equation 7.4 by calculating the magnitude of the velocity and acceleration vectors (v_x and v_y become velocity, and analogously to the acceleration), and the heading direction. Note that this state and dynamic matrix do not use ψ, it uses the *heading* (h) instead. The h angle is the direction of the velocity and is calculated as in Equation 7.6. According to the chosen model, the Equation 7.5 is the dynamic matrix that updates the state of all global objects whenever a new message arrives.

$$\begin{bmatrix} x \\ y \\ h \\ v \\ \dot{\psi} \\ a \end{bmatrix} \tag{7.4}$$

$$\begin{bmatrix} x + \frac{-\dot{\psi}\ v\sin(h) - a\cos(h) + a\cos(\Delta t\ \dot{\psi} + h) + (\Delta t\ \dot{\psi}\ a + \dot{\psi}\ v)\sin(\Delta t\ \dot{\psi} + h)}{\dot{\psi}^2} \\ y + \frac{\dot{\psi}\ v\cos(h) - a\sin(h) + a\sin(\Delta t\ \dot{\psi} + h) + (-\Delta t\ \dot{\psi}\ a - \dot{\psi}\ v)\cos(\Delta t\ \dot{\psi} + h)}{\dot{\psi}^2} \\ \Delta t\ \dot{\psi} + h \\ \Delta t\ a + v \\ \dot{\psi} \\ a \end{bmatrix} \tag{7.5}$$

$$h = arctan(v_y, v_x) \tag{7.6}$$

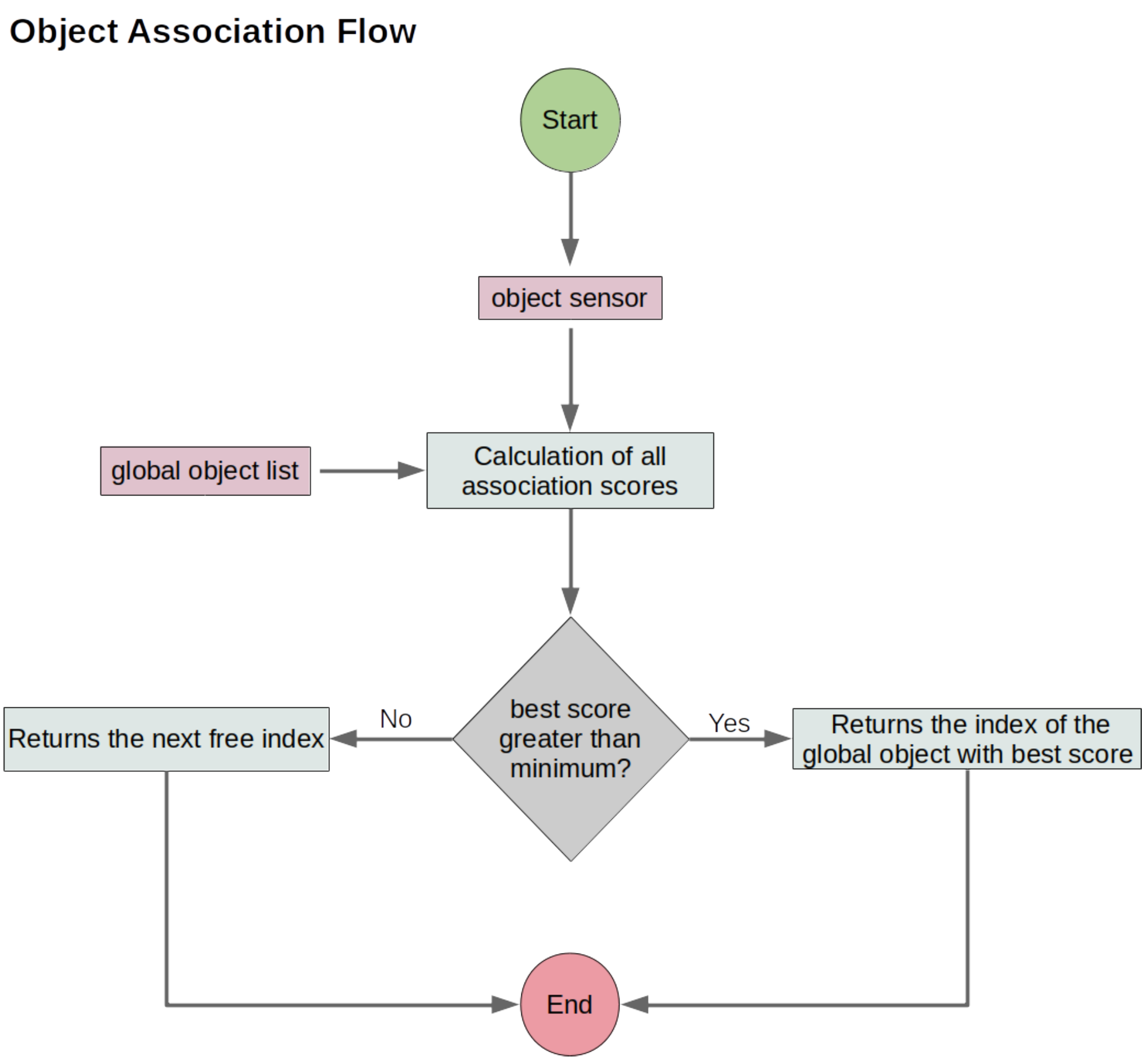
7.5 Association

Subsubsection 5.2.2.3 describes the process of object association. Aeberhard proposes a complete approach that handle partial observations of objects, considers the object classification, probability of existence and the errors covariances. Other techniques for *track-to-track* association could also be used.

With the objective of reducing this work scope, it was implemented a much simpler technique that is functional only for the best cases. It is considered that the registered sensors are always capable of providing enough information to estimate the dimensions and state of the objects, Equation 5.1 and Equation 5.2, respectively. Moreover, this is the only information considered when associating.

In the solution implementation, all of the global objects have an associated index. The task of the implemented association is to, for each object from a list arriving from a sensor, return the index from the corresponding global object. The arriving object is considered a new global object if the association score is not greater than a minimum defined. In this case, the association returns the next available index in the global list. Figure 17 illustrates this process.

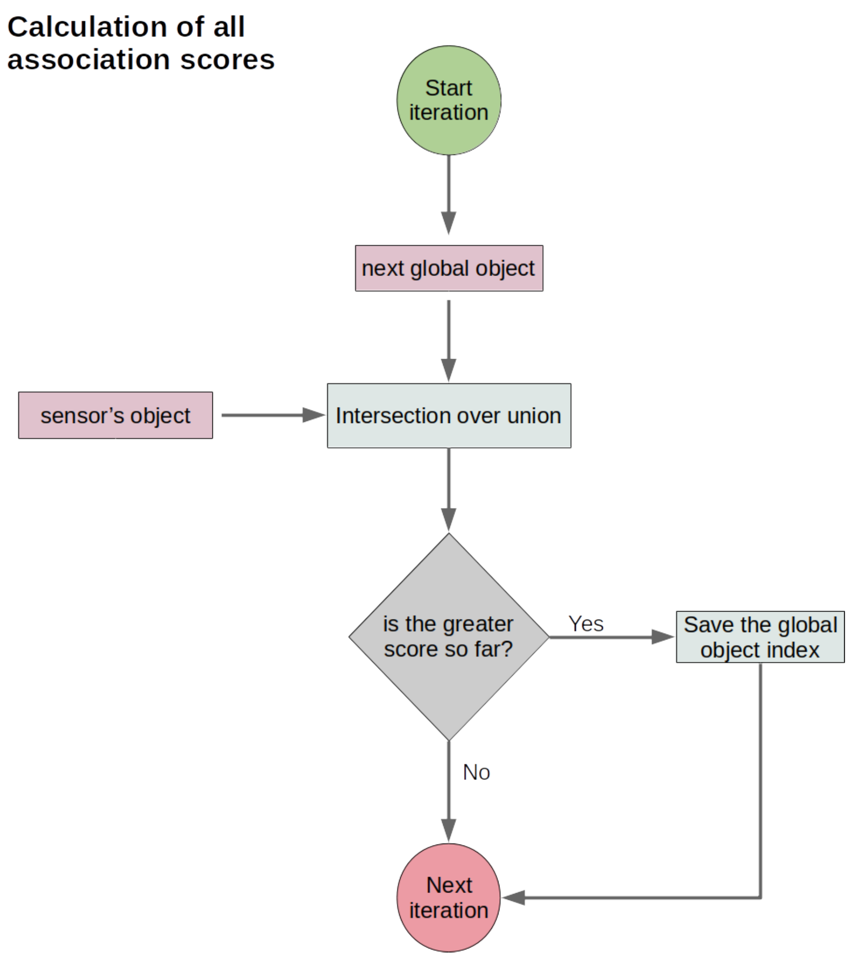
Figure 17 – Association Flow


Object Association Flow
Start
object sensor
global object list
Calculation of all association scores
best score greater than minimum?
No
Yes
Returns the next free index
Returns the index of the global object with best score
End

If the association returns the index of an existing global object, it means that, in the real world, they might be the same object, and their information should be fused. Otherwise, it means that the arriving object was not being tracked but now will.

The association scores are calculated as illustrated in Figure 18, and uses IoU (Intersection over Union) as the metric. IoU is "the most commonly used metric for comparing the similarity between two arbitrary shapes" (REZATOFIGHI et al., 2019). These shapes are two rotated rectangles in this solution, one representing an object from the sensor object list, and the other representing one object from the global object list. The object from the sensor list is compared with all the objects from the global list, and the index from the global object with the best score is returned.

Figure 18 – Association score calculation


Calculation of all association scores
Start iteration
next global object
sensor's object
Intersection over union
is the greater score so far?
Yes
Save the global object index
No
Next iteration

7.6 Packages file structure

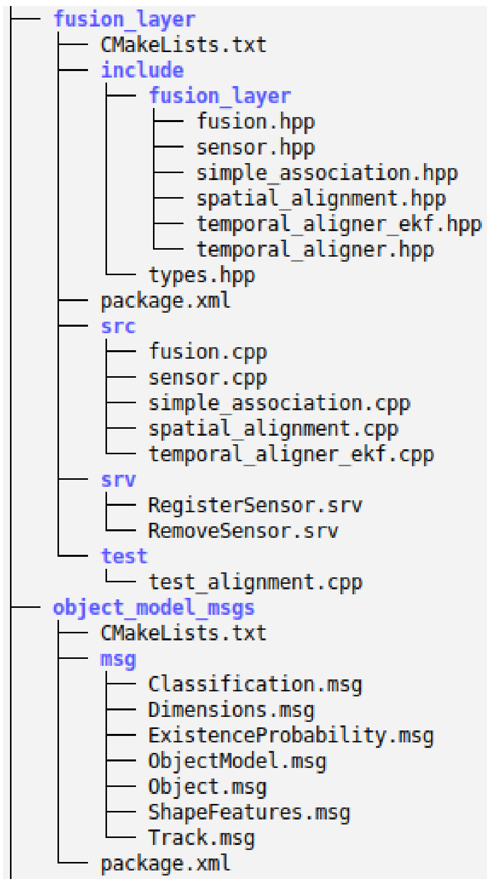
The repository [1] was created in the GitHub platform using the Git version control system. It is a repository with two ROS C++ packages compatible with the Foxy Fitzroy ROS release (see Section 6.2). The *fusion_layer* package contains the implementation of

[1] <https://github.com/icaropires/objectlevel_fusion>

the preprocessing of the object lists, and the *object_model_msgs* holds the definitions of the messages which compose the object model (described in Section 5.1). The messages definitions are available in the repository and in Section A.

The main part of the file structure of the repository is defined as shown in Figure 19. In the repository root, there is one directory for each package and configuration files. Inside each package directory, the structure is as defined by ROS itself, *src*, *include*, *package.xml*, *CMakeLists.txt* are auto-generated and made available for customization. The *object_model_msgs* do not have any *headers* or *src* files, it will only contain the definition files for the messages inside *msg*. The *srv* directory in the *fusion_layer* package holds the definition files for the services of the fusion node, currently only the services to register or remove sensors.

Figure 19 – Packages in the Repository Structure



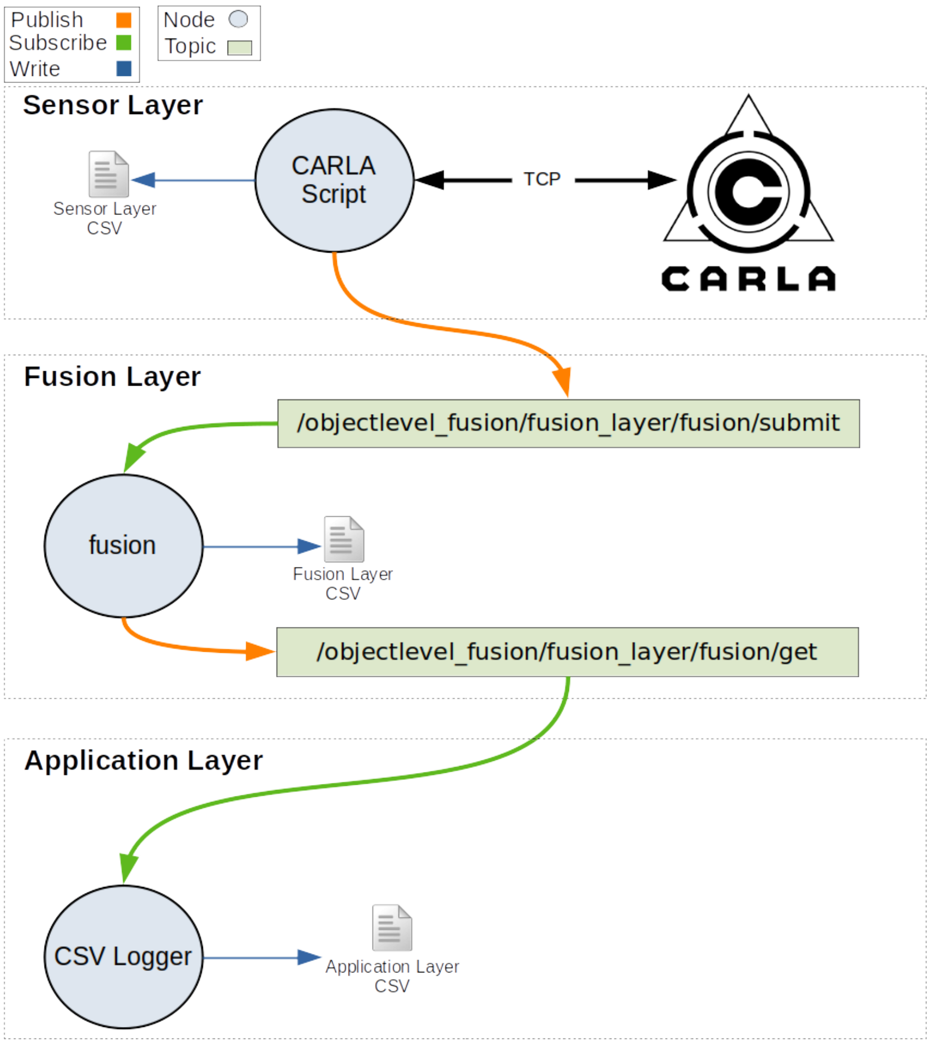
```
├── fusion_layer
│   ├── CMakeLists.txt
│   ├── include
│   │   ├── fusion_layer
│   │   │   ├── fusion.hpp
│   │   │   ├── sensor.hpp
│   │   │   ├── simple_association.hpp
│   │   │   ├── spatial_alignment.hpp
│   │   │   ├── temporal_aligner_ekf.hpp
│   │   │   └── temporal_aligner.hpp
│   │   └── types.hpp
│   ├── package.xml
│   ├── src
│   │   ├── fusion.cpp
│   │   ├── sensor.cpp
│   │   ├── simple_association.cpp
│   │   ├── spatial_alignment.cpp
│   │   └── temporal_aligner_ekf.cpp
│   ├── srv
│   │   ├── RegisterSensor.srv
│   │   └── RemoveSensor.srv
│   └── test
│       └── test_alignment.cpp
├── object_model_msgs
│   ├── CMakeLists.txt
│   ├── msg
│   │   ├── Classification.msg
│   │   ├── Dimensions.msg
│   │   ├── ExistenceProbability.msg
│   │   ├── ObjectModel.msg
│   │   ├── Object.msg
│   │   ├── ShapeFeatures.msg
│   │   └── Track.msg
│   └── package.xml
```

Keeping a separated package containing the messages definitions favors a better semantic and keeps them decoupled from the *fusion_layer*. Worth mentioning that there are more directories in the repository, like *examples*, and *docker*. These, on the figure, are only the directories for the created ROS packages.

8 Experiment

Some experiment scenarios were performed, as Figure 20 illustrates, using the CARLA self-driving simulator (CARLA, 2021). In the fusion layer, a default script from CARLA was customized to generate object lists according to object model specifications and to publish them to */objectlevel_fusion/fusion_layer/fusion/submit*. In the fusion layer there is the fusion node, which is the core of this software solution, as described in Chapter 7. The node in the application layer is a Python script listening on the */objectlevel_fusion/fusion_layer/fusion/get* topic. Note also that all the three nodes log information to a separated CSV file for posterior assessment. The following sections enter in more details about how these steps are implemented.

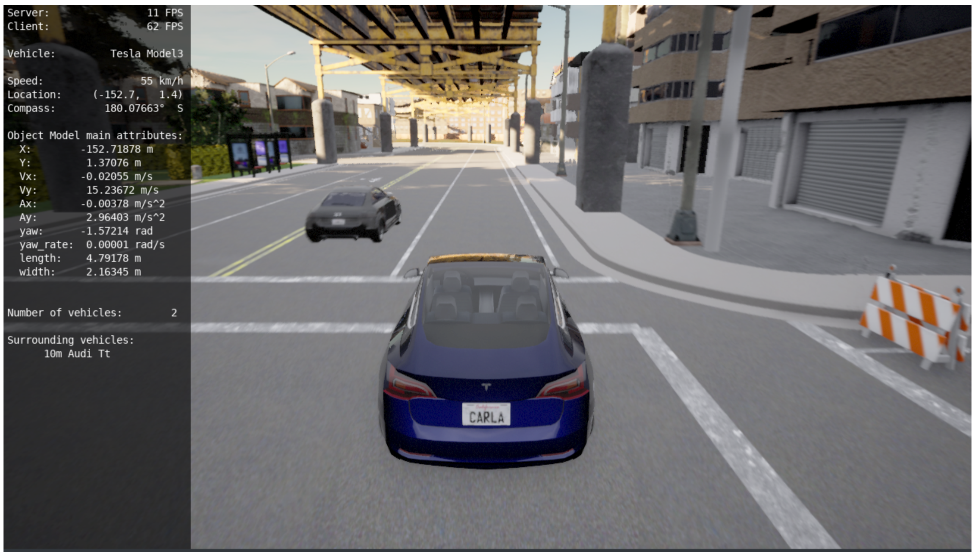
Figure 20 – Experiment Modules



8.1 Customized Manual Control Script

In addition to the main application, CARLA also provides some example Python scripts on how to interact with the simulation. The *manual_control.py* is one of these scripts, developed with *pygame* (PYGAME, 2021) and that enables the user to control a vehicle (or pedestrian) as if it was a game. This script was customized by removing some unnecessary features (which could cost performance), by adding two surround sensors, and by implementing the integration with the fusion node (see Section 7.3). The script in execution is illustrated in Figure 21.

Figure 21 – Custom manual control script in execution



```
Server:                 11 FPS
Client:                 62 FPS

Vehicle:          Tesla Model3

Speed:                 55 km/h
Location:      (-152.7,   1.4)
Compass:        180.07663°   S

Object Model main attributes:
  X:         -152.71878 m
  Y:          1.37076 m
  Vx:        -0.02055 m/s
  Vy:         15.23672 m/s
  Ax:        -0.00378 m/s^2
  Ay:         2.96403 m/s^2
  yaw:       -1.57214 rad
  yaw_rate:   0.00001 rad/s
  length:     4.79178 m
  width:      2.16345 m


Number of vehicles:         2

Surrounding vehicles:
      10m Audi Tt
```

Another option would be to use the CARLA ROS bridge, which also provides another version of *manual_control.py* that, by default, publishes information about the simulation to ROS (CARLA, 2021b). This option would also require customizations to interact with the fusion node, but after assessing the source code, it seemed easier to customize the default *manual_control.py*.

In this customization, a kind of sensor was also implemented to produce object lists of the ego vehicles surroundings. In this work, this new kind of sensor is called *surround sensor* (Subsection 8.1.1).

8.1.1 Surround Sensor

The software solution in this work only implements part of the fusion layer. It does not have a sensor layer, making experimentation harder because the sensor layer is responsible for transforming the raw data from sensors into object lists, following the object model specification. This problem was addressed by implementing the *surround sensor*.

The surround sensor is an entity that mocks the behavior of a real-world sensor by publishing surrounding perception information. To be instantiated, it receives all the information defined in Subsection 7.3.1. It uses that information plus the value of the attributes of the closest vehicles (and pedestrians, depending on the configuration) to produce object lists. Its range is arbitrarily defined as 50 meters but could be any value. As this experiment is about the best scenarios, the precision of the sensor is absolute, but noises could be added to the data to produce more realistic scenarios.

Currently, the object lists produced by the surround sensor do not have all the attributes filled, only the attributes that are being used by the current implementation of the fuse node. The filled attributes are all from Equation 5.1 and Equation 5.2, but that could be extended when implementing more components or improving the existing ones. Table 4 concentrates the summary of the surround sensor attributes.

Table 4 – Surround sensor attributes

Attribute	**Value**
range	50 m
accuracy	100%
coordinate frame	X forward, Y left
length	-
width	-
readings	object lists

8.1.1.1 Readings

The attributes measured by the sensor are obtained from CARLA using methods and attributes from the *carla.Actor* class (CARLA, 2021a), according to Table 5. The negative sign in yaw and yaw rate are needed to get positive rotation in the counter-clockwise direction. With these transformations, the values follow the right-hand system and use the east north up (ENU) convention (FOOTE; PURVIS, 2010). However, the fusion node needs the attributes relative to the sensor position, and, at this point, they are relative to the CARLA map.

These attributes can be made relative to the sensor with some additional steps described in the sequence. First, considering the ego vehicle state as $state^{ego}$ and $\alpha = \psi^{ego}$, the rotation matrix R (Equation 8.1) is applied to the ego state resulting in the $state^{r-ego}$

Table 5 – CARLA attributes to object model, in Python

object model attribute	**CARLA attribute (class carla.Actor)**
x	actor.get_location().x
y	actor.get_location().y
v_x	actor.get_velocity().x
v_y	actor.get_velocity().y
a_x	actor.get_acceleration().x
a_y	actor.get_acceleration().y
ψ	-math.radians(actor.get_rotation().yaw)
$\dot{\psi}$	-math.radians(actor.get_angular_velocity().z)
w	actor.bounding_box.extent.y*2
l	actor.bounding_box.extent.x*2

(Equation 8.2).

$$R = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 & 0 & 0 & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \cos(\alpha) & -\sin(\alpha) & 0 & 0 & 0 & 0 \\ 0 & 0 & \sin(\alpha) & \cos(\alpha) & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ 0 & 0 & 0 & 0 & \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.1}$$

$$state^{r\_ego} = R \cdot state^{ego} \tag{8.2}$$

The H_{rel} transformation matrix (Equation 8.3) is then applied to the $expanded\_state^{obj}$ (Equation 8.4) resulting in the $expanded\_state^{rel\_obj}$, a state that has all the attributes relative to the ego vehicle (Equation 8.6). This transformation matrix concentrates a rotation, subtraction from the $state^{r\_ego}$, and mirroring of the y axis. At this point, the attributes are relative to the ego vehicle. Lastly, to make these attributes relative to the sensor which measured them, the same steps from Subsection 7.3.2 are applied, but instead of H, the inverse matrix of H is used. After all these steps, the state that is published to the fusion module is obtained. Note that the $expanded\_state^{rel\_obj}$

is to $state^{rel\_obj}$ the same as $expanded\_state^{obj}$ is to $state^{obj}$ (see Equation 8.5).

$$H_{rel} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 & 0 & 0 & 0 & 0 & -x^{r\_ego} \\ -\sin(\alpha) & -\cos(\alpha) & 0 & 0 & 0 & 0 & 0 & 0 & y^{r\_ego} \\ 0 & 0 & \cos(\alpha) & -\sin(\alpha) & 0 & 0 & 0 & 0 & -v_x^{r\_ego} \\ 0 & 0 & -\sin(\alpha) & -\cos(\alpha) & 0 & 0 & 0 & 0 & v_y^{r\_ego} \\ 0 & 0 & 0 & 0 & \cos(\alpha) & -\sin(\alpha) & 0 & 0 & -a_x^{r\_ego} \\ 0 & 0 & 0 & 0 & -\sin(\alpha) & -\cos(\alpha) & 0 & 0 & a_y^{r\_ego} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -\psi^{r\_ego} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -\dot{\psi}^{r\_ego} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (8.3)$$

$$expanded\_state^{obj} = \begin{bmatrix} x^{obj} \\ y^{obj} \\ v_x^{obj} \\ v_y^{obj} \\ a_x^{obj} \\ a_y^{obj} \\ \psi^{obj} \\ \dot{\psi}^{obj} \\ 1 \end{bmatrix} \quad (8.4)$$

$$expanded\_state^{obj} = \begin{bmatrix} state^{obj} \\ 1 \end{bmatrix} \quad (8.5)$$

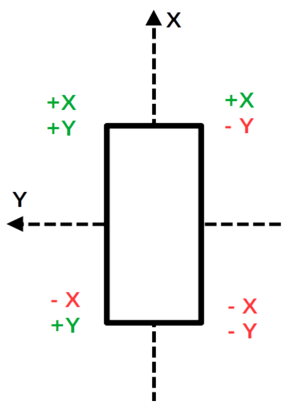
$$expanded\_state^{rel\_obj} = H_{rel} \cdot expanded\_state^{obj} \quad (8.6)$$

In the state to be published, the signs from the values x, y, v_x, v_y, a_x, and a_y will be according to Figure 22. It is important to note that all the values published by an instance of a *surround sensor* follow this same instance reference frame. Therefore, unless two instances are at the same position and with the same rotation, they will publish different values for the same observed object. Later, the fusion node will spatially align these values (see Subsection 7.3.2).

8.2 Obtained Information

While running each scenario from the experiment, a series of information is collected. For each execution are saved a ROS bag, a simulation recording file, and three CSV files (one for each layer). The ROS bag records all the information transmitted between the sensor and the fusion layers for posterior re-execution. Consequently, the fusion layer

Figure 22 – Surround sensor reference frame



can be modified, and the same data replayed from the bag without the need to rerun the simulator.

The CARLA simulator also has a feature to record the simulation, generating *.rec* files. However, when using this feature, attributes as acceleration and velocity were not being replayed, only the position of each object (CARLA, 2020). This problem was the primary motivation also to use ROS bags. Nevertheless, it was decided to keep the record file, facilitating the possibility of watching an experiment again visually.

The information contained in the CSV files is according Table 6 for the sensor layer, Table 7 for the fusion layer, and Table 8 for the application layer. Figure 23 illustrates the current directory structure for two experiments: *both_moving*, and *stopped_car*. Note the *simulation.rec* files about the carla recordings, the CSV files, and the bag directories *input_bag*.

Table 6 – Fields description from sensor layer's CSV file

attribute	**description**
timestamp	Timestamp in nanoseconds measured since 00:00:00 UTC, at 1 January 1970
sensor_name	Name of the sensor which made the measurement
n_objects	Number of objects in the object list produced by the sensor at this measurement
<ego vehicle state>	One column for each attribute from the state from the ego vehicle, according to the object model

8.3 Results

There are five test scenarios: two main experiments that have two variations each, and a last experiment with only one variation. The experiments are the *observed vehicle stopped*, *both vehicle moving*, and *city tour*. The variations from the first two experiments

Table 7 – Fields description from fusion layer's CSV file

attribute	**description**
timestamp	Timestamp in nanoseconds measured since 00:00:00 UTC, at 1 January 1970.
sensor_name	Name of the sensor which made the measurement.
<detected vehicle state>	One column for each attribute from the state from the detected vehicle, according to the object model.
idx_to_associate	The index from the global list returned by the association function. Which global object this detected object corresponds to.
association_score	The best association score calculated when compared this detected object to every global object.
association_failed	Indicates if the association for this detected object failed.

Table 8 – Fields description from application layer's CSV file

attribute	**description**
timestamp	Timestamp in nanoseconds measured since 00:00:00 UTC, at 1 January 1970.
global_list_id	The id of the list that contains this object. This id starts at one and is incremented for each global list that arrives.
<global object state>	One column for each attribute from the state from the global object, according to the object model.
length	Length of the detected object.
width	Width of the detected object.

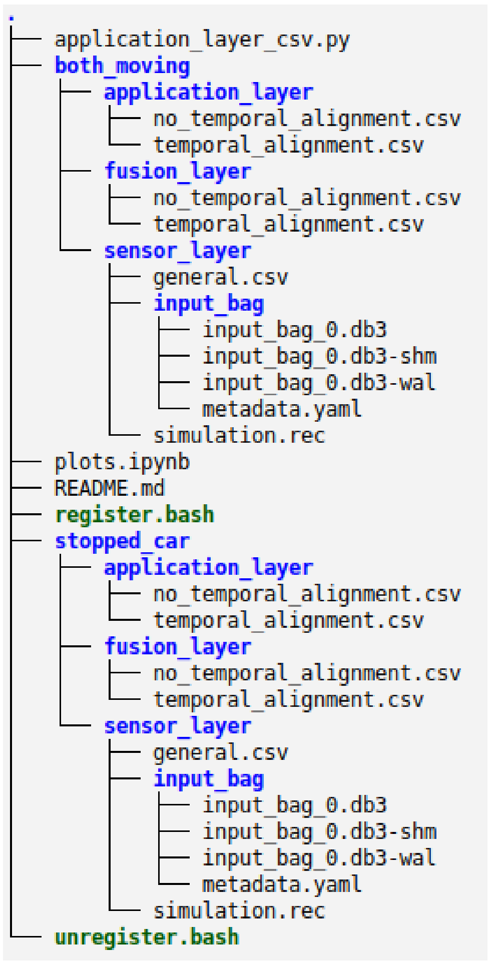
are the use or not of the temporal alignment, and in the last experiment, the temporal alignment is used. All of the experiments use object association and spatial alignment.

The vehicles and sensors used in all test cases are according to Table 9 and Table 10, respectively. The software and hardware environment used are as in Table 11. To make the *observed* vehicle be stopped or moving were used, respectively, the default *manual_control.py* and the *automatic_control.py* scripts. Another essential point is the association score, which had 0.5 as the minimum acceptable value in these experiments, but this value can be further calibrated.

Table 9 – Vehicles used in the experiments

vehicle	**model**	**length (m)**	**width (m)**
ego	Tesla Model 3	4.791	2.163
observed	Audi TT	4.181	1.994

Figure 23 – Experiments directory structure



```
.
├── application_layer_csv.py
├── both_moving
│   ├── application_layer
│   │   ├── no_temporal_alignment.csv
│   │   └── temporal_alignment.csv
│   ├── fusion_layer
│   │   ├── no_temporal_alignment.csv
│   │   └── temporal_alignment.csv
│   └── sensor_layer
│       ├── general.csv
│       ├── input_bag
│       │   ├── input_bag_0.db3
│       │   ├── input_bag_0.db3-shm
│       │   ├── input_bag_0.db3-wal
│       │   └── metadata.yaml
│       └── simulation.rec
├── plots.ipynb
├── README.md
├── register.bash
├── stopped_car
│   ├── application_layer
│   │   ├── no_temporal_alignment.csv
│   │   └── temporal_alignment.csv
│   ├── fusion_layer
│   │   ├── no_temporal_alignment.csv
│   │   └── temporal_alignment.csv
│   └── sensor_layer
│       ├── general.csv
│       ├── input_bag
│       │   ├── input_bag_0.db3
│       │   ├── input_bag_0.db3-shm
│       │   ├── input_bag_0.db3-wal
│       │   └── metadata.yaml
│       └── simulation.rec
└── unregister.bash
```

Table 10 – Sensors positions (relative to the ego vehicle)

name	**type**	**x (m)**	**y (m)**	**yaw angle (rad)**	**range (m)**
sensor1	*surround sensor*	1	-2	$\pi/4$	50
sensor2	*surround sensor*	-1	0.5	$-\pi/3$	50

Additionally, the following topics are facts for the first two experiments (*observed vehicle stopped*, *both vehicle moving*):

- the primary metric used to validate the fusion node implementation is the number of failed associations performed;
- an association is failed when there is an object in the global list that corresponds to a object detected by a sensor, but the association algorithm is not able to indicate it;
- in case of a successful association, a newly detected object replaces the global object that the association indicates;

Table 11 – Experiments environment

Attribute	**Description**
Laptop model	Acer Nitro AN515-52
GPU	Geforce GTX 1050 Ti Mobile
GPU configuration method	PRIME Render Offload
NVIDIA driver version	450.80.0.2
CPU	Intel Core i7-8750H
Operating System	Linux, Kernel version 5.8.0
Desktop Environment	KDE Plasma 5.14.5
Python version	3.7.3
CARLA version	0.9.11
CARLA start command	*DISPLAY= ./CarlaUE4.sh -opengl*
CARLA Map	Town03
ROS version	Foxy

- the test scenarios were elaborated with two vehicles, the *ego* and the *observed*;
- the configuration used in the *surround sensors* is to only detect vehicles;
- the *observed* vehicle never returns to the sensor range once that it is out;
- each failed association generates a new global object.

8.3.1 Observed Vehicle Stopped

In this first experiment, there is a stopped vehicle on the road (the *observed*), then the ego vehicle comes already accelerated in its direction and keeps accelerating. The ego vehicle passes next to the *observed* and continues forward until the observed vehicle gets out of range. Figure 24 illustrates the trajectory and velocity of both vehicles during both variations of this experiment. The trajectory and velocity for both the variations are the same because a ROS bag is recorded with the simulator data, and the same bag is played when executing each of the variations. To complement the understanding, Figure 25 illustrates the distance between both vehicles over time.

First, the variation with the temporal alignment disabled was executed. Not using the temporal alignment in the first execution allows checking how much the current implementation of this alignment is really contributing to the final result. The summary of this execution is in Table 12. The number of association errors is 23, and, consequently, the number of global objects detected was 24. Calculating the relevant Pearson correlation coefficients (using (PANDAS, 2021)), as shown in Figure 26, reveals that the *time_increment* is almost proportional to the *association_failed*.

However, this experiment favors this correlation due to its high relative velocity and the observed vehicle not performing complex movements. Figure 27 shows that one

Figure 24 – Vehicles travel for *stopped car* experiment

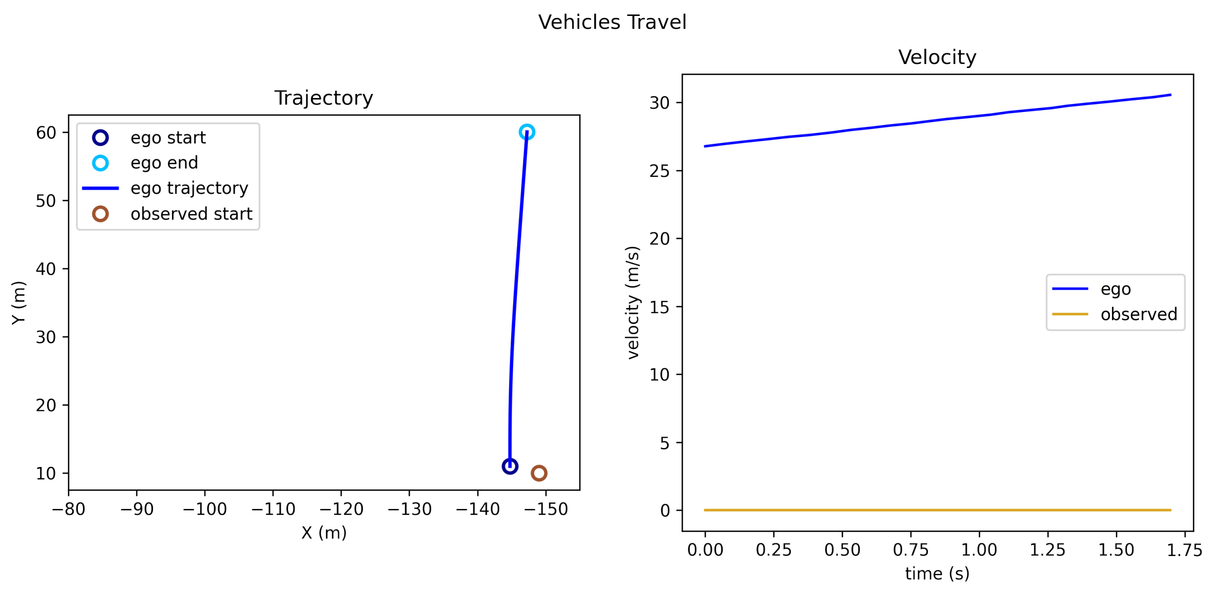


Figure 25 – Ego distance for *stopped car* experiment

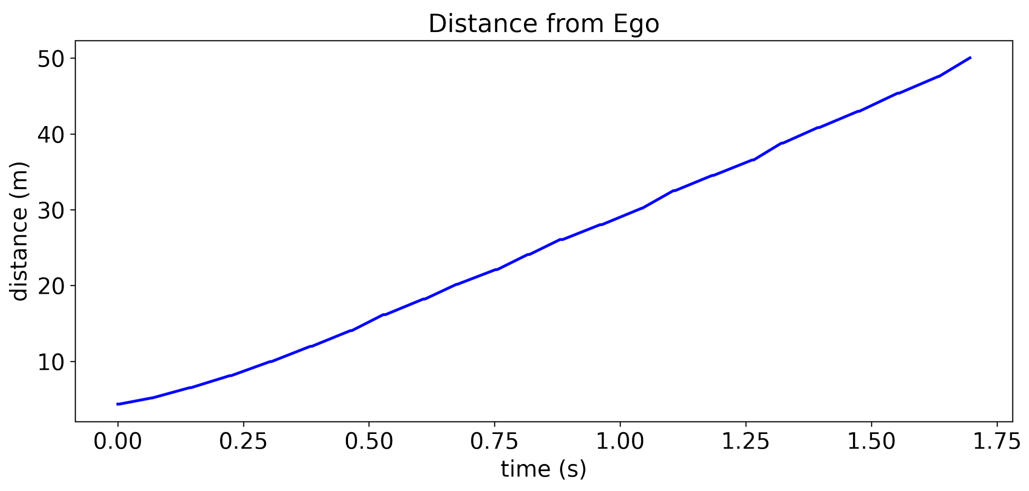
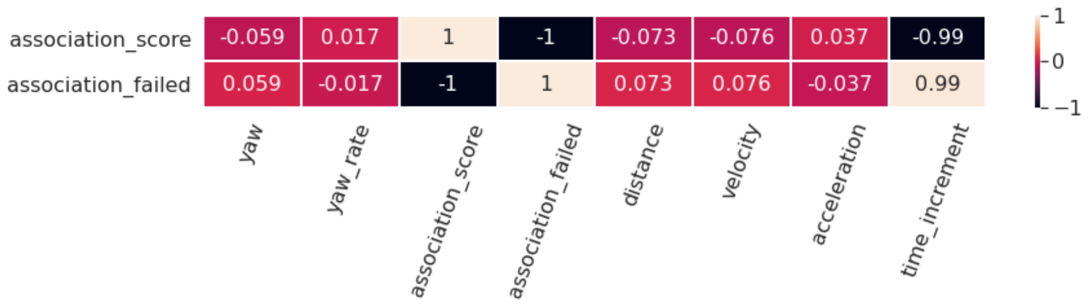


Table 12 – Results summary for *stopped car* experiment - Not using temporal alignment

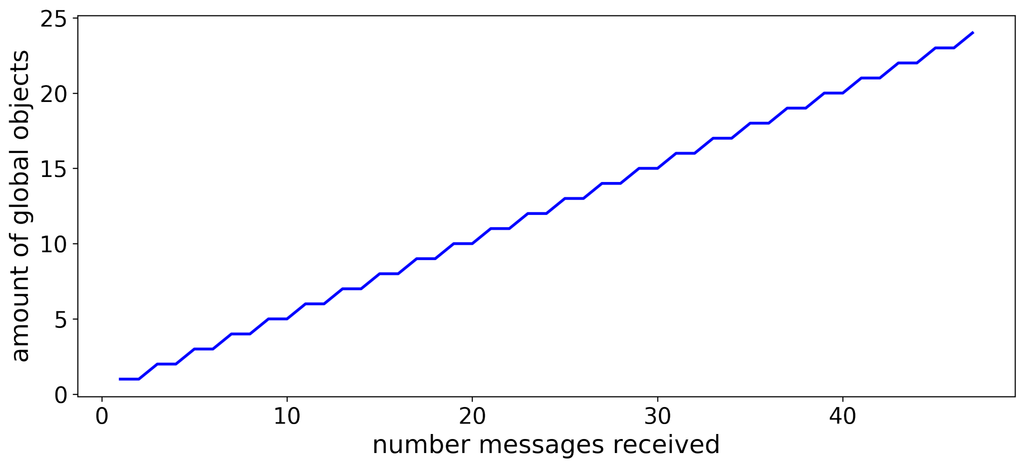
Attribute	**Value**
number of association errors	23
number of detected global objects	24
number of measurements	47
mean of measurements frequency (Hz)	27.708
duration (s)	1.696
maximum relative velocity magnitude (m/s)	30.555

Figure 26 – Errors correlation for *stopped car* experiment



new global object is created for every two messages received from the fusion layer. Every two messages because the interval between measurements from the two *surround sensors* is significantly smaller than the time between ticks: average of ˜0.003s between sensors against average of ˜0.074s between the ticks. In this execution, the association is failing for the bigger interval, and succeeding for the smaller. To decrease this interval between ticks, it is needed to increase the Frames Per Second (FPS) from the CARLA server (CARLA, 2021).

Figure 27 – Number of global objects for *stopped car* experiment



Now, also using the temporal alignment, the execution summary is as in Table 12. It can be observed that the temporal alignment decreased the number of association errors to zero, detecting only one global object, which is the correct information and the best possible result.

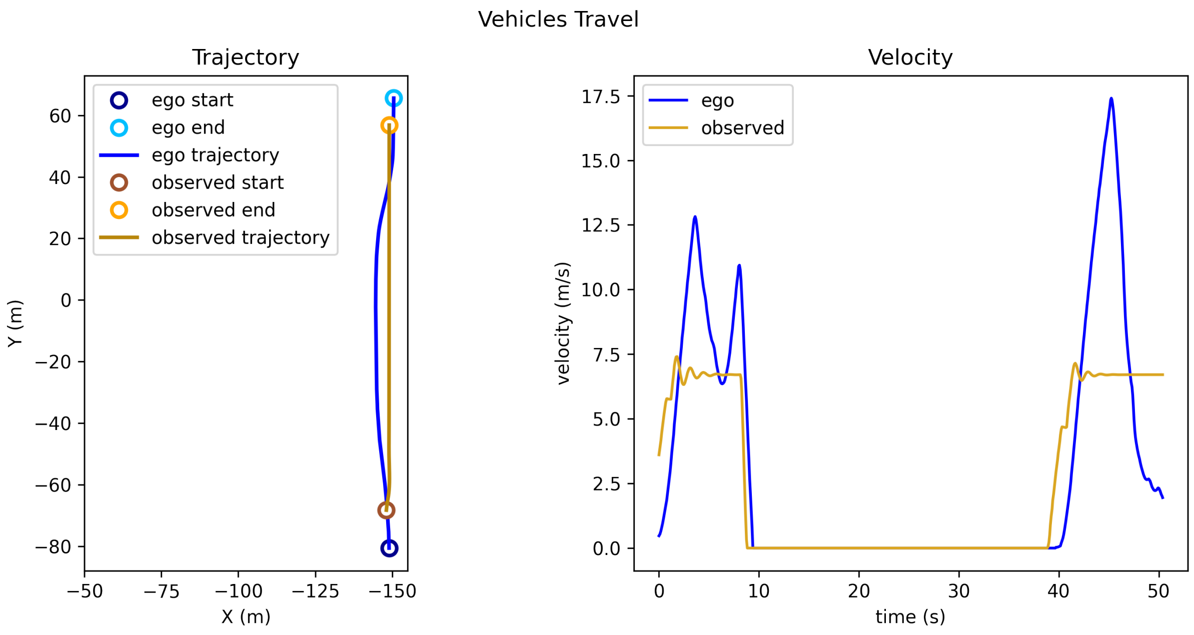
Table 13 – Results summary for *stopped car* experiment - Using temporal alignment

Attribute	**Value**
number of association errors	0
number of detected global objects	1
number of measurements	47
mean of measurements frequency (Hz)	27.708
duration (s)	1.696
maximum relative velocity magnitude (m/s)	30.555

8.3.2 Both vehicle moving

For the second experiment, both vehicles are moving in the same direction. Compared with the first experiment, there is a smaller relative velocity between the vehicles, but more variety of movements. Analogously to the *observed vehicle stopped* experiment, Figure 28 illustrates the movement and velocity from both vehicles, and Figure 29 contains information about the distance between them. Note that in the velocity graph, approximately in the period between 10 and 40 seconds, the vehicles are stopped. This period was when the vehicles were waiting to cross the traffic light.

Figure 28 – Vehicles travel for *both vehicles moving* experiment



In the first execution, with the temporal alignment disabled, the result is showed in Table 14. There were 17 association errors, detecting 18 global objects. Comparing with the same variation in the last experiment, there were less association errors for more than 30 times the experiment duration. In addition to the time the vehicles were stopped, this is probably due to the smaller values for *maximum relative velocity magnitude*, 10.988 against 30.555 meters per second.

Figure 29 – Distance from ego vehicle for *both vehicles moving* experiment

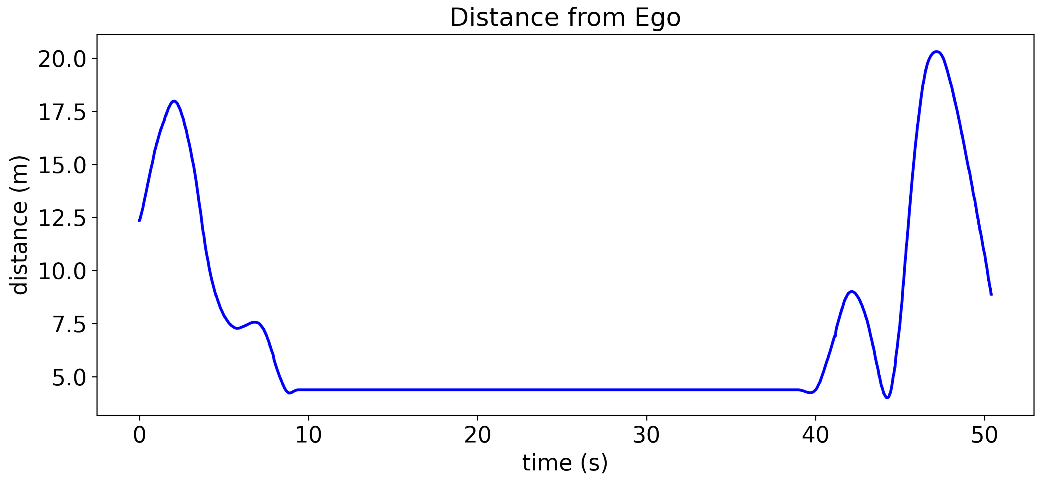


Table 14 – Results summary for *both vehicles moving* experiment - Not using temporal alignment

Attribute	**Value**
number of association errors	17
number of detected global objects	18
number of measurements	1308
mean of measurements frequency (Hz)	25.953
duration (s)	50.397
maximum relative velocity magnitude (m/s)	10.988

Unlike the last experiment, the correlation table (presented in Figure 30) is not conclusive. The reason is that the correlation is being affected by the time the vehicles remained stopped. To reduce this disturbance, Figure 31 shows the correlation from only measurements in the period after the first 40 seconds, and it is observable a greater inverse correlation between *time_increment* and *association_score*, which, by definition, has an inverse correlation with the *association_failed*. Considering this, the impact from *time_increment* on the *association_failed* attribute is consistent with the last experiment, when the observed vehicle was stopped.

Figure 30 – Errors correlation for *both vehicles moving* experiment

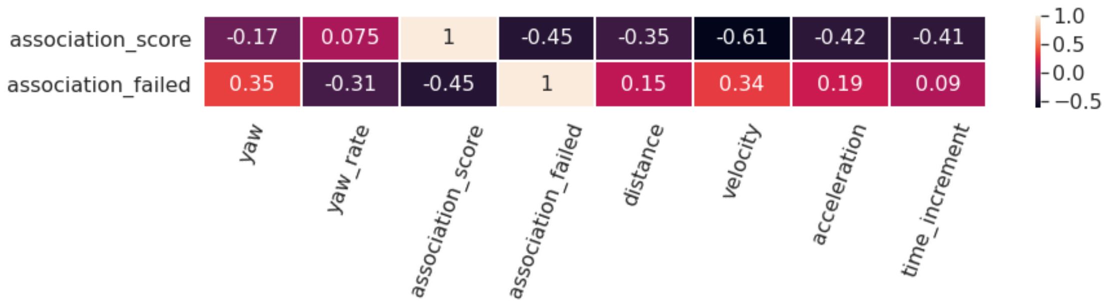
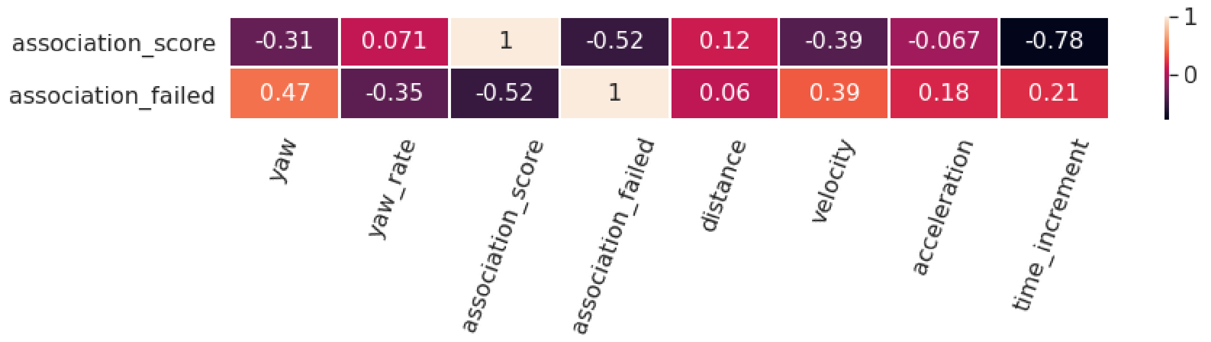


Figure 31 – Errors correlation for *both vehicles moving* experiment - After first 40 seconds



After enabling the temporal alignment, the result in Table 15 shows zero association errors and one global object detected, the same as in the last experiment (*observed vehicle stopped*). This result demonstrates that the current implementation of the temporal alignment is being effective even when both vehicles are moving.

Table 15 – Results summary for *both vehicles moving* experiment - Using temporal alignment

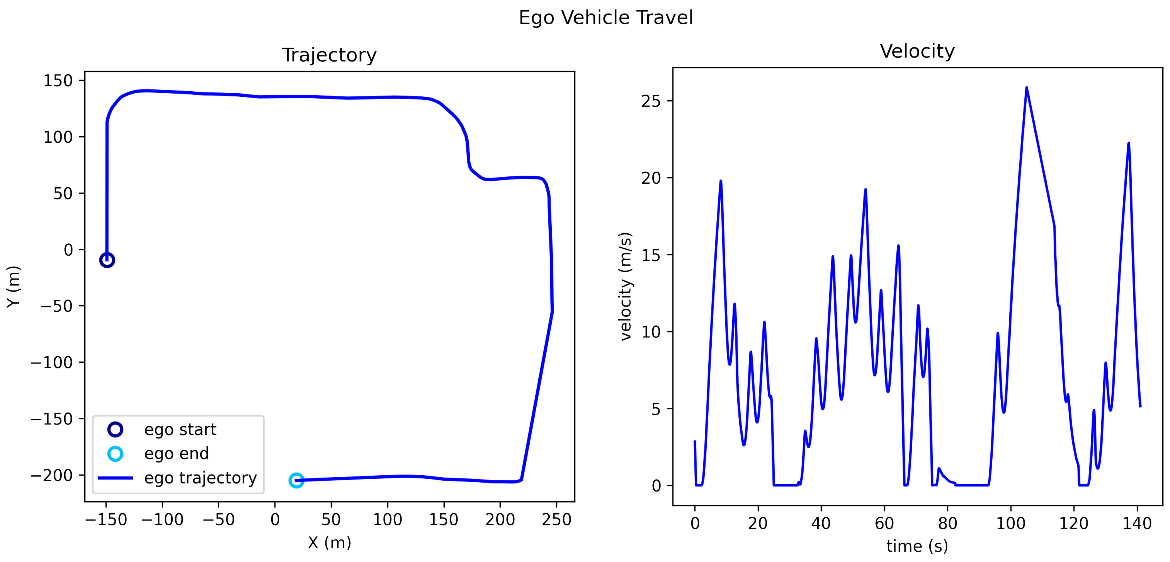
Attribute	**Value**
number of association errors	0
number of detected global objects	1
number of measurements	1308
mean of measurements frequency (Hz)	25.953
duration (s)	50.397
maximum relative velocity magnitude (m/s)	10.988

8.3.3 City tour

In the *city tour* experiment, 30 pedestrians and 30 vehicles are first spawned on the map. Then the ego vehicle is manually driven, observing some of the pedestrians and vehicles. These vehicles and pedestrians are spawned using the default *spawn_npc.py* script and follow random trajectories in the map. Another difference when comparing with the last experiments is the *surround sensors* configuration. Previously, the surround sensor was capable of detecting only vehicles, it was now configured to detect vehicles and pedestrians. Same as with the last experiments, Figure 32 shows the trajectory and velocity from the ego vehicle, but only from the ego. Another point

In the last experiments, there was only one more object to be detected on the map, and then it was not a problem to identify failed associations. For this experiment, a different approach was taken. Table 16 contains the summary of the execution, with each row representing objects with a specific dimension. The *best case* column is the number of different objects detected by the ego vehicle with the same dimension. *Worst case*

Figure 32 – Ego vehicle travel for *city tour* experiment



indicates how many measurements were made for objects with this dimension, and *result* is the number of global objects with this dimension after passing by the fusion node. Finally, *score* is how close the *result* is from *best case*, and it is calculated according to the Equation 8.7. It ranges from zero, when the result is the *worst case*, to one, when it is the *best case*. For more information about the objects that have these dimensions, check Appendix C.

Table 16 – Results summary for *city tour* experiment

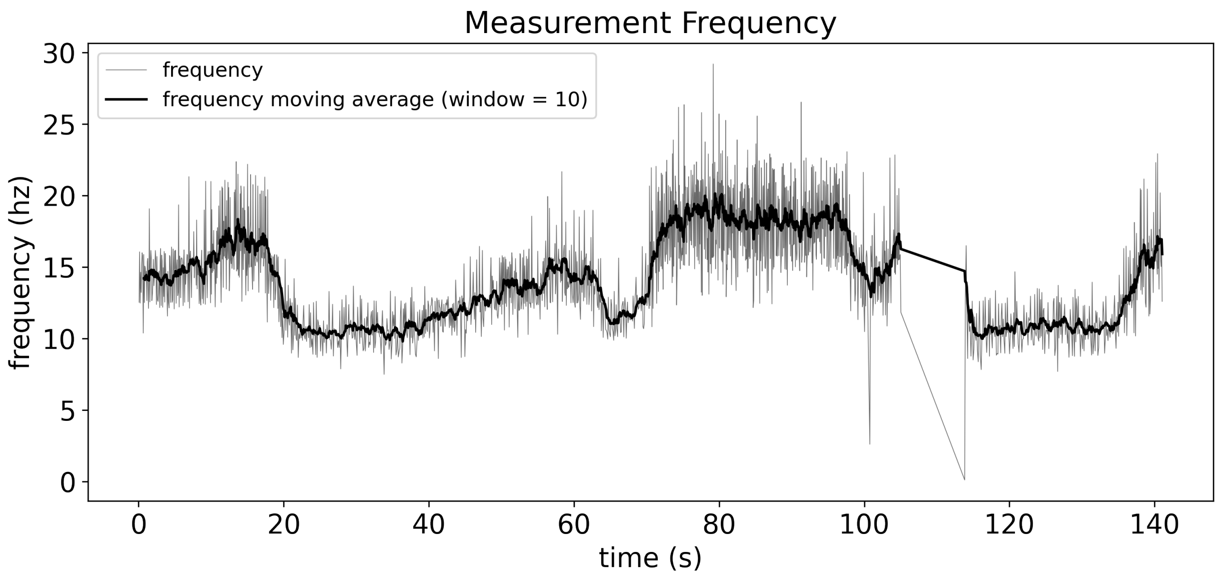
dimensions (m) (length, width)	**result**	**best case**	**worst case**	**score**
(0.38, 0.38)	438	7	2336	0.81
(0.50, 0.50)	93	1	328	0.72
(1.49, 0.86)	42	1	948	0.96
(2.21, 0.87)	14	1	234	0.94
(3.63, 1.84)	9	1	224	0.96
(3.71, 1.79)	5	1	127	0.97
(3.81, 1.97)	88	1	1494	0.94
(3.87, 1.91)	9	3	490	0.99
(4.19, 1.82)	22	2	348	0.94
(4.67, 2.00)	36	1	1491	0.98
(4.72, 1.90)	31	2	432	0.93
(4.79, 2.16)	3	1	148	0.99
(4.86, 2.03)	51	2	1570	0.97
(4.97, 2.04)	7	1	730	0.99
(5.20, 2.62)	10	1	535	0.98
(5.24, 2.10)	7	1	736	0.99

$$score = 1 - \frac{result - best\_case}{worst\_case - best\_case} \tag{8.7}$$

Eventually, it is possible that *best case* can not be reached because the current association algorithm does not handle object occlusions. Anyhow, even using the temporal alignment, *score* did not reach the *best case* value even once. The worst results were from the smaller objects, as expected. It is expected because bigger objects have a bigger chance of getting better association scores for the same time interval between the measurements. Another problem that should be tackled is old objects in the global object list not being removed after some time. This behavior can cause new objects to be mistaken with old ones.

As demonstrated in the last experiments, the time between measurements directly impacts the association failures. Figure 33 shows the measurement frequency during this experiment execution. The mean is 14.53 Hz considering only the *sensor1*. Considering both sensors, it would be nearly double, which is not very different from, for example, the average from *both vehicles moving* (which was 27.70 Hz). However, more association errors will probably occur in the moments this frequency decreases.

Figure 33 – Measurements frequency for *city tour* experiment, considering only *sensor1*



9 Conclusion

The accuracy and reliability requirements for ADS applications are very demanding, which increases the need for multisensor data fusion. Sensors solutions that are enough for more straightforward robotic applications do not meet the requirements for the self-driving scenario. The redundancy between sensors of the same type and subsequent fusion may increase the data's reliability through fault tolerance. Moreover, the fusion between different types of sensors may increase the accuracy, taking advantage of the best capabilities from each type of sensor.

The multisensor data fusion may be implemented in different types of fusion architectures, according to the requirements of the specific problem it is meant to solve. For ADS, despite the accuracy from high-level and hybrid architectures possibly being enough, it may vary depending on the implementation and other aspects. Low-level architectures often provide better accuracy at the cost of less maintainability, more bandwidth usage, among others. Furthermore, strategies like the environment model may be used to make an architecture more robust and scalable.

Despite the usefulness and popularity of multisensor fusion in self-driving contexts, it is still common for companies and research centers to implement their solutions from the ground up. The software solution implemented in this work offers a partial and initial solution because not all modules needed to perform fusions are implemented, and the solution is not currently ready to receive data from real sensors.

However, the used fusion architecture decouples the algorithms in the fusion layer from details from each perception sensor, making the solution more generic. Additionally, the use of ROS favors modularity, integration, and packaging. Moreover, in the future, using the correct configurations and tools, it might be possible even to meet the requirements for firm real-time applications.

Additionally, it is currently already possible to run the solution as an independent module, register and remove sensors, and communicate to it through the defined topics. It receives object lists, applies spatial and temporal alignments, association, and returns global lists. During the validation, it performed associations in scenarios with data coming directly from a self-driving simulator, from two sensors in two different positions.

Nevertheless, this validation was performed in a well-controlled environment, with mocked sensors publishing ground truth data. It was not used data from real-world sensors at any moment, neither errors were emulated. Furthermore, the association implementation needs to be extended and improved to support such scenarios. It currently considers that the sensors can always send accurate information about the object dimensions, position, velocity, and acceleration, which is not a real-world scenario. Thus, the solution still need to be evolved to support simulations closer to the reality.

9.1 Suggestions for Future Works

This work offers an initial implementation based on solid foundations for object-level fusion. The related future works should be increments to this foundation in order to build a robust solution capable of handling complex scenarios from the real world.

Thereby, the initial steps are to finish the implementation of the fusion layer and to improve (or substitute) the current object association implementation to consider partial observations of an object and the sensor's errors (see (AEBERHARD, 2017)). Once the application receives these improvements, it could also be validated with data from real-world sensors. Additionally, using the readings from the surround sensor, it is even possible to generate object lists to mock a specific behavior, adding to them the desired noise or omitting specific attributes.

Other improvements related to performance can also be made, such as measuring the current implementation's real-time capabilities and improving it, maybe testing in an RTOS. Another suggestion is to delete global objects that are not relevant anymore, improving the algorithm's performance and saving resources.

Bibliography

AEBERHARD, M. *Object-level fusion for surround environment perception in automated driving applications*. Tese (Doutorado), 2017. Cited 9 times in pages 1, 17, 18, 20, 23, 29, 43, 44, and 66.

AEBERHARD, M. et al. Automated driving with ros at bmw. *ROSCon 2015 Hamburg, Germany*, 2015. Cited in page 35.

Apex.AI. *Apex.OS | Developer*. 2020. <https://www.apex.ai/apexos-developer>. (Accessed on 11/29/2020). Cited in page 35.

BALDWIN, R. *Waymo Reveals Details on 5th-Gen Self-Driving Jaguar I-Pace*. 2020. <https://www.caranddriver.com/news/a31228617/waymo-jaguar-ipace-self-driving-details/>. (Accessed on 05/27/2021). Cited in page 8.

BALZER, P. *Extended Kalman Filter Implementation for Constant Turn Rate and Acceleration (CTRA) Vehicle Model in Python*. 2021. <https://github.com/balzer82/Kalman/blob/master/Extended-Kalman-Filter-CTRA.ipynb>. (Accessed on 05/08/2021). Cited in page 44.

BEHERE, S.; TORNGREN, M. A functional architecture for autonomous driving. In: IEEE. *2015 First International Workshop on Automotive Software Architecture (WASA)*. [S.l.], 2015. p. 3–10. Cited in page 28.

BURKACKY, O. et al. Rethinking car software and electronics architecture. *McKinsey & Company*, 2018. Cited in page 11.

CARLA. *No velocity or acceleration on log files*. 2020. <https://github.com/carla-simulator/carla/issues/3145>. (Accessed on 05/18/2021). Cited in page 54.

CARLA. *CARLA actor*. 2021. <https://carla.readthedocs.io/en/0.9.11/python_api/#carlaactor>. (Accessed on 05/15/2021). Cited in page 51.

CARLA. *Carla Manual Control*. 2021. <https://carla.readthedocs.io/projects/ros-bridge/en/latest/carla_manual_control/>. (Accessed on 05/08/2021). Cited in page 50.

CARLA. *CARLA Simulator*. 2021. <https://carla.readthedocs.io/en/latest/>. Cited 2 times in pages 9 and 49.

CARLA. *Client-server synchrony*. 2021. <https://carla.readthedocs.io/en/latest/adv_synchrony_timestep/#client-server-synchrony>. (Accessed on 05/08/2021). Cited in page 59.

CARPENTER, S. *Autonomous Vehicle Radar: Improving Radar Performance with Simulation*. 2018. <https://www.ansys.com/about-ansys/advantage-magazine/volume-xii-issue-1-2018/autonomous-vehicle-radar>. (Accessed on 10/31/2020). Cited in page 2.

Connected Places Catapult. *Autonomous Valet Parking*. [S.l.], 2019. Disponível em: <https://avp-project.uk/wp-content/uploads/2019/12/AVP-Requirements-Car-Park.pdf>. Cited in page 34.

DDS Foundation. *What is DDS?* 2020. <https://www.dds-foundation.org/what-is-dds-3/>. (Accessed on 12/06/2020). Cited in page 33.

eSOL. *ROS/ROS 2 Engineering Service | Embedded Software Solutions | eSOL - Real-time embedded software platform solutions*. 2020. <https://www.esol.com/embedded/ros.html#solution>. (Accessed on 11/29/2020). Cited in page 35.

EUGSTER, P. T. et al. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, ACM New York, NY, USA, v. 35, n. 2, p. 114–131, 2003. Cited in page 31.

European GSA. *What is GNSS?* 2017. <https://www.gsa.europa.eu/european-gnss/what-gnss>. (Accessed on 10/31/2020). Cited in page 8.

FOOTE, T.; PURVIS, M. *Standard Units of Measure and Coordinate Conventions*. 2010. <https://www.ros.org/reps/rep-0103.html#chirality>. (Accessed on 05/08/2021). Cited in page 51.

FOWLER, M. *Patterns of enterprise application architecture*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 2002. Cited in page 25.

GERKEY, B. *Why ROS 2?* 2020. <https://design.ros2.org/articles/why_ros2.html>. (Accessed on 11/29/2020). Cited 2 times in pages 34 and 35.

GUTIÉRREZ, C. S. V. et al. Towards a distributed and real-time framework for robots: Evaluation of ros 2.0 communications for real-time robotic applications. *arXiv preprint arXiv:1809.02595*, 2018. Cited in page 35.

HALL, J. L. D. L. *Handbook of multisensor data fusion*. 1. ed. [S.l.]: CRC Press, 2001. v. 2 Volume Set. (The Electrical engineering and applied signal processing series, v. 2 Volume Set). ISBN 9780849323799,0849323797. Cited 3 times in pages 13, 14, and 19.

HO, C. et al. *ROS 2 on Autonomous Vehicles*. 2018. <https://roscon.ros.org/2018/presentations/ROSCon2018_ROS2onAutonomousDrivingVehicles.pdf>. (Accessed on 12/05/2020). Cited in page 35.

KAO, W.-W. Integration of gps and dead-reckoning navigation systems. In: IEEE. *Vehicle Navigation and Information Systems Conference, 1991*. [S.l.], 1991. v. 2, p. 635–643. Cited in page 8.

KAY, J. *Introduction to Real-time Systems*. 2020. <https://design.ros2.org/articles/realtime_background.html>. (Accessed on 11/29/2020). Cited in page 35.

KAY, J. *Proposal for Implementation of Real-time Systems in ROS 2*. 2020. <https://design.ros2.org/articles/realtime_proposal.html>. (Accessed on 11/29/2020). Cited in page 35.

KERNER, B. J. Software testing for embedded applications in autonomous vehicles. 2017. Cited in page 35.

KHALEGHI, B. et al. Multisensor data fusion: A review of the state-of-the-art. *Information fusion*, Elsevier, v. 14, n. 1, p. 28–44, 2013. Cited in page 21.

KIM, Y.-D. et al. On the deployment and noise filtering of vehicular radar application for detection enhancement in roads and tunnels. *Sensors*, Multidisciplinary Digital Publishing Institute, v. 18, n. 3, p. 837, 2018. Cited in page 10.

KOCIĆ, J.; JOVIČIĆ, N.; DRNDAREVIĆ, V. Sensors and sensor fusion in autonomous vehicles. In: IEEE. *2018 26th Telecommunications Forum (TELFOR)*. [S.l.], 2018. p. 420–425. Cited in page 7.

KREBS, S.; DURAISAMY, B.; FLOHR, F. A survey on leveraging deep neural networks for object tracking. In: IEEE. *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. [S.l.], 2017. p. 411–418. Cited in page 9.

KUUTTI, S. et al. A survey of the state-of-the-art localization techniques and their potentials for autonomous vehicle applications. *IEEE Internet of Things Journal*, IEEE, v. 5, n. 2, p. 829–846, 2018. Cited in page 8.

LARMAN, C.; BASILI, V. R. Iterative and incremental developments. a brief history. *Computer*, IEEE, v. 36, n. 6, p. 47–56, 2003. Cited in page 3.

LI, A. *Waymo's autonomous Jaguar I-Pace looks so much sleeker 'painted black'*. 2021. <https://9to5google.com/2021/03/03/waymo-jaguar-i-pace-black/>. (Accessed on 05/27/2021). Cited in page 8.

LIU, L. et al. Computing systems for autonomous driving: State-of-the-art and challenges. *arXiv preprint arXiv:2009.14349*, 2020. Cited in page 8.

MALAVOLTA, I. et al. How do you architect your robots? state of the practice and guidelines for ros-based systems. *Proceedings of ICSE-CEIP. ACM*, v. 10, n. 3377813.3381358, 2020. Cited in page 39.

micro-ROS. *Features and Architecture | micro-ROS*. 2020. <https://micro-ros.github.io//docs/overview/features/>. (Accessed on 11/29/2020). Cited in page 35.

micro-ROS. *Supported Hardware | micro-ROS*. 2020. <https://micro-ros.github.io/docs/overview/hardware/>. (Accessed on 11/29/2020). Cited in page 35.

MOUZINHO, L. et al. Indirect measurement of the temperature via kalman filter. In: CITESEER. *XVIII IMEKO World Congress*. [S.l.], 2006. v. 9. Cited in page 7.

MUNZ, M.; MAHLISCH, M.; DIETMAYER, K. Generic centralized multi sensor data fusion based on probabilistic sensor and environment models for driver assistance systems. *IEEE Intelligent Transportation Systems Magazine*, IEEE, v. 2, n. 1, p. 6–17, 2010. Cited in page 1.

OBJECT MANAGEMENT GROUP. *Interface Definition Language*. [S.l.], 2018. Version 4.2. Disponível em: <https://www.omg.org/spec/IDL/4.2/PDF>. Cited in page 34.

PANDAS. *pandas.DataFrame.corr*. 2021. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html>. (Accessed on 05/18/2021). Cited in page 57.

PANGERCIC, D. *Doing Real-time With ROS 2: Capabilities And Challenges*. 2019. <https://www.streetscooter.com/wp-content/uploads/2020/03/dejan-pangercic-intro-use-cases-for-realtime.pdf>. (Accessed on 12/05/2020). Cited in page 35.

PANGERCIC, D. *ROS 2 and Real-time - Next Generation ROS - ROS Discourse*. 2019. <https://discourse.ros.org/t/ros-2-and-real-time/8796>. (Accessed on 11/29/2020). Cited in page 35.

PARK, J.; DELGADO, R.; CHOI, B. W. Real-time characteristics of ros 2.0 in multiagent robot systems: An empirical study. *IEEE Access*, IEEE, v. 8, p. 154637–154651, 2020. Cited in page 35.

PYGAME. *About*. 2021. <https://www.pygame.org/wiki/about>. (Accessed on 05/08/2021). Cited in page 50.

RAOL, J. R. *Multi-sensor data fusion with MATLAB*. [S.l.]: CRC press, 2009. Cited in page 13.

Red Hat. *How to Start a Robot Revolution*. 2020. <https://www.redhat.com/en/open-source-stories/how-to-start-a-robot-revolution>. (Accessed on 11/19/2020). Cited in page 31.

REKE, M. et al. A self-driving car architecture in ros2. In: IEEE. *2020 International SAUPEC/RobMech/PRASA Conference*. [S.l.], 2020. p. 1–6. Cited in page 35.

REZATOFIGHI, H. et al. Generalized intersection over union: A metric and a loss for bounding box regression. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. [S.l.: s.n.], 2019. p. 658–666. Cited in page 46.

ROS. *ROS Index*. 2020. <https://index.ros.org/stats/>. (Accessed on 12/06/2020). Cited in page 33.

ROS. *ROS/Introduction - ROS Wiki*. 2020. <http://wiki.ros.org/ROS/Introduction>. (Accessed on 12/02/2020). Cited in page 31.

ROS. *ROS.org | About ROS*. 2020. <https://www.ros.org/about-ros/>. (Accessed on 11/19/2020). Cited in page 31.

ROS. *Writing a simple publisher and subscriber (C++)*. 2020. <https://index.ros.org/doc/ros2/Tutorials/Writing-A-Simple-Cpp-Publisher-And-Subscriber/>. (Accessed on 12/06/2020). Cited in page 33.

ROS 2. *Real-time programming in ROS 2*. 2020. <https://index.ros.org/doc/ros2/Tutorials/Real-Time-Programming/>. (Accessed on 11/29/2020). Cited in page 35.

SIEWERT, S.; PRATT, J. *Real-Time Embedded Components and Systems with Linux and RTOS*. [S.l.]: Stylus Publishing, LLC, 2015. Cited in page 34.

STEINBERG, A. N.; BOWMAN, C. L.; WHITE, F. E. Revisions to the jdl data fusion model. *Handbook of multisensor data fusion*, 2008. Cited 3 times in pages 14, 15, and 18.

The Autoware Foundation. *Autoware.Auto*. 2020. <https://www.autoware.auto/>. (Accessed on 12/05/2020). Cited in page 36.

THOMAS, D. *Interface definition using .msg / .srv / .action files*. 2020. <http://design.ros2.org/articles/legacy_interface_definition.html>. (Accessed on 11/29/2020). Cited in page 34.

WOHLIN, C. et al. *Engineering and managing software requirements*. [S.l.]: Springer Science & Business Media, 2005. Cited in page 39.

WOODALL, W. *ROS on DDS*. 2020. <http://design.ros2.org/articles/ros_on_dds.html>. (Accessed on 11/19/2020). Cited in page 34.

YU, X.; MARINOV, M. A study on recent developments and issues with obstacle detection systems for automated vehicles. *Sustainability*, Multidisciplinary Digital Publishing Institute, v. 12, n. 8, p. 3281, 2020. Cited in page 10.

YURTSEVER, E. et al. A survey of autonomous driving: Common practices and emerging technologies. *IEEE Access*, IEEE, v. 8, p. 58443–58469, 2020. Cited 3 times in pages 7, 9, and 21.

Appendix

APPENDIX A – Messages definitions

This appendix contains the ROS message definitions used in the software solution.

A.1 Object Model

The object model, defined as in Source Code A.1 is the main message type for this software solution. It is both, the input and output for the fusion layer. All the other source codes of this section were defined in order to be used as attributes from the object model.

Source Code A.1 – Object Model message definition

```
# Using frame_id from header to specify the sensor name
std_msgs/Header header

Object[] object_model
```

Source Code A.2 – Object message definition

```
Track track
Dimensions dimensions
ExistenceProbability existence_probability
Classification classification
ShapeFeatures shape_features
```

Source Code A.3 – Track message definition

```
# Enumeration
uint8 STATE_X_IDX = 0
uint8 STATE_Y_IDX = 1
uint8 STATE_VELOCITY_X_IDX = 2
uint8 STATE_VELOCITY_Y_IDX = 3
uint8 STATE_ACCELERATION_X_IDX = 4
uint8 STATE_ACCELERATION_Y_IDX = 5
uint8 STATE_YAW_IDX = 6
uint8 STATE_YAW_RATE_IDX = 7

uint8 STATE_SIZE = 8

# Size is STATE_SIZE
float32[8] state # [x, y, velocity x, velocity y, acceleration x, acceleration y, yaw, yaw rate]

# Square matrix, size is the size of state squared
# float32[64] covariation # Only 1d array supported. Access with `covariation[i*8 + j]`

# Using covariation as size of CTRA squared for now, until remodel CTRA or figure out
#   how to transform from 6x6 to 8x8 and back
float32[36] covariation
```

Source Code A.4 – Dimensions message definition

```
# Enumeration
uint8 DIMENSIONS_LENGHT_IDX = 0
uint8 DIMENSIONS_WIDTH_IDX = 1

float32[2] values # [length, width]
float32[2] uncertainty
```

Source Code A.5 – Classification message definition

```
# Enumeration
uint8 CLASS_CAR_IDX = 0
uint8 CLASS_TRUCK_IDX = 1
uint8 CLASS_MOTORCYCLE_IDX = 2
uint8 CLASS_BICYCLE_IDX = 3
uint8 CLASS_PEDESTRIAN_IDX = 4
uint8 CLASS_STATIONARY_IDX = 5
uint8 CLASS_OTHER_IDX = 6

float32[7] probabilities # [car, truck, motorcycle, bicycle, pedestrian, stationary, other]
```

Source Code A.6 – Existence Probability message definition

```
float32 probability
```

Source Code A.7 – Shape Features message definition

```
# Indicates which corner and side features from the object were observable

# Enumeration
uint8 FEATURE_FL_IDX = 0
uint8 FEATURE_FR_IDX = 1
uint8 FEATURE_RL_IDX = 2
uint8 FEATURE_RR_IDX = 3
uint8 FEATURE_FM_IDX = 4
uint8 FEATURE_RM_IDX = 5
uint8 FEATURE_ML_IDX = 6
uint8 FEATURE_MR_IDX = 7

bool[8] features # [FL FR RL RR FM RM ML MR]
```

APPENDIX B – Services definitions

B.1 Sensors Registration

These are the files for the definitions of the services responsible for adding or removing sensors. Only messages from the registered sensors are considered in the fusion layers, according to subsection 7.3.1.

Source Code B.1 – Register Sensor service definition

```
# Service to register sensors in the fusion layer
# sensors not registered will have its data ignored in fusion
string name

# All relative to the vehicle coordinate frame
float32 x
float32 y
float32 angle

# Which attributes from object_model_msgs/msg/Track.state this sensor is able to provide
# Size is object_model_msgs::msg::Track::STATE_SIZE
bool[8] capable

# The R matrix from EKF
# 6x6 matrix, according to CTRA model
# Row/columns are about attributes [x, y, v, a, yaw, yaw_rate], order matters
float32[36] measurement_noise_matrix
---
# Success/error message
string status
```

Source Code B.2 – Shape Features service definition

```
# Service to remove from registered sensors

string name
---
string status
```

C Observed Objects Dimensions

Table 17 shows the *type_id* of the CARLA objects observed in the *city tour* experiment.

Table 17 – CARLA object dimensions by type_id

dimensions	**type_id**
(0.38, 0.38)	walker.pedestrian.0014 walker.pedestrian.0022 walker.pedestrian.0026 walker.pedestrian.0025 walker.pedestrian.0016 walker.pedestrian.0004 walker.pedestrian.0019
(0.50, 0.50)	walker.pedestrian.0010
(1.49, 0.86)	vehicle.bh.crossbike
(2.21, 0.87)	vehicle.yamaha.yzf
(3.63, 1.85)	vehicle.nissan.micra
(3.71, 1.79)	vehicle.audi.a2
(3.81, 1.97)	vehicle.mini.cooperst
(3.87, 1.91)	vehicle.jeep.wrangler_rubicon
(4.19, 1.82)	vehicle.seat.leon
(4.67, 2.00)	vehicle.mercedesccc.mercedesccc
(4.72, 1.89)	vehicle.mustang.mustang
(4.79, 2.16)	vehicle.tesla.model3
(4.86, 2.03)	vehicle.audi.etron
(4.97, 2.04)	vehicle.dodge_charger.police
(5.20, 2.61)	vehicle.carlamotors.carlacola
(5.24, 2.10)	vehicle.chargercop2020.chargercop2020