The University Of Sheffield.

Verification-Driven Design, Simulation and Implementation of a Self-Driving Vehicle

by

Mohammed Yaseen Hazim Al-Nuaimi

A thesis submitted as partial fulfilment for the degree of Doctor of Philosophy

Department of Automatic Control and Systems Engineering

Faculty of Engineering

The University of Sheffield

March 2020

Abstract

The evolution of driving technology has recently progressed from active safety features and advanced driver assistance system (ADAS) to fully sensor-guided Autonomous Vehicle (AV). Bringing such robotic vehicles to roads requires not only simulation and testing but formal verification to account for all possible traffic scenarios.

It is important to guarantee safety for all agents moving in the same environment (in our case study; people, vehicles and our AV) while the AV is exploring the surroundings. Comparing with static or controlled environments, high dynamic environments present many other difficulties: the detection and tracking of the moving objects, the prediction of their future state in the world, and the run-time planning and navigation.

To demonstrate the feasibility of our safety system and to contribute to the field of self-driving vehicles, we have designed our open-source AV software using the Robot Operating System (ROS) and Gazebo simulator. The overall structure of our AV system framework consists of (i) A perception system of sensors that feeds data into (ii) a Rational Agent (RA) based on a Belief-Desire-Intention (BDI) architecture and designed using sEnglish based Natural Language Programming (NLP). The RA used for decision-making is connected to (iii) a verification system, and (iv) a feedback control system for following a self-planned path.

A new hybrid verification approach, which combines the use of two well-known model checkers: a Model Checker for Multi-Agent Systems (MCMAS) and Prism, is presented for this purpose. MCMAS is used to check the consistency and stability of the agent logic during design-time. Prism is used to provide the RA with the probability of success while it decides which action to take during run-time operation. This then allows the RA to select movements of the highest probability of success from several alternatives.

The AV system as mentioned above has been implemented on a prototype electric vehicle as a fully functional AV (level-4 autonomy) in collaboration with Tata Motors European Technical Centre (TMETC) to test the feasibility of our AV system. Both the simulation and practical implementation considers a parking lot environment to check the feasibility of this approach. A demonstration of our AV system is shown in a video link below. [1]

[1] `https://tiny.cc/mohammed-av-2019`

Acknowledgements

First and foremost, all thanks and praise are due to ALLAH the almighty for his blessing that made this work possible.

This research project was funded by the Higher Committee for Education Development in Iraq (HCED), also partially funded and supported by Tata Motors European Technical Centre (TMETC), AMRC research centre and The University of Sheffield, UK.

I would like to thank my dear supervisor, Prof Sandor Veres, for his excellent supervision, and for all the great opportunities he provided me with. Without his support, guidance and knowledge through the stages of the work, this thesis would never have been completed.

I also would like to thank Dr Hongyang Qu for the support and guidance with the verification theory and the verification tools. Dr Jonathan Aitken for his comments on the rational agent design. My colleague Dr Sapto Wibowo for his valuable support, suggestions and lengthy discussions on the topic of self-driving vehicles.

Special thanks to my thesis examiners, Professor Alice Miller / University of Glasgow and Dr Paul Trodden / University of Sheffield, who provided constructive feedback for my thesis and research.

Thanks must also go to all the academic, technician, and administrative staff of the Automatic Control and Systems Engineering department. Also, to my course mates for providing me with guidance, support and a friendly environment.

A thank you also goes to Dr Mark Tucker, Dr Maradona Rodrigues and Suresh Arikapudi / Tata Motors European technical centre, for their valuable guidance, comments and suggestions on the hardware implementation of the autonomous vehicle system.

Last but not least, a big thank you to all my family members and friends for always being there in my life, surrounding me with your care and support.

March 2020, Sheffield, UK

Mohammed Al-Nuaimi

myhazim2@gmail.com

Contents

List of Figures

List of Tables

List of Acronyms

3D	Three Dimensions
AC	Alternative Current
Ace-EV	Tata Ace Electric Vehicle
ACF	Aggregated Channel Features
ADAS	Advanced Driver Assistance Systems
AI	Artificial Intelligence
AMC	Auto Master Controller
AOP	Agent Oriented Programming
ATL	Alternating-time Temporal Logic
AV	Autonomous Vehicle
BDD	Binary Decision Diagram
BDI	Belief-Desire-Intention
BES	Boolean Evolution System
BRF	Breadth-First Search
BRF	Belief Review Function
BUF	Belief Update Function
CAN	Controller Area Network
CAT	Cognitive Agent Toolbox
CTL	Computational Tree Logic
CTMC	Continuous-Time Markov Chain
CUDA	Compute Unified Device Architecture
CV	Computer Vision
DARPA	Defence Advanced Research Projects Agency
DOF	Degree Of Freedom
DTMC	Discrete-Time Markov Chain
EKF	Extended Kalman Filter
EV	Electric Vehicle
FOV	Field Of View
FPS	Frame Per Second
FSM	Finite State Machine
GPS	Global Positioning System
GPU	Graphics Processing Unit
HS	Hybrid System
ISPL	Interpreted Systems Programming Language
LiDAR	Light Detection And Ranging
LISA	Limited Instruction Set Agent

LOAM	LiDAR Odometry and Mapping
LTL	Linear Temporal Logic
LTS	Labelled Transition System
MC	Markov Chain
MCMAS	Model Checker for Multi-Agent Systems
MDP	Markov Decision Processes
MOL	Machine Ontology Language
MPC	Model Predictive Control
NHTSA	National Highway Traffic Safety Administration
NLP	Natural Language Programming
OBDD	Ordered Binary Decision Diagrams
OEM	Original Equipment Manufacture
OOP	Object-Oriented Programming
OS	Operating System
PA	Probabilistic Automata
PCTL	Probabilistic Computation Tree Logic
POMDP	Partially Observable Markov Decision Processes
PRS	Procedural Reasoning System
PTA	Probabilistic Timed Automata
PTP	Probabilistic Timed Programs
PTCTL	Probabilistic Timed Computational Tree Logic
RA	Rational Agent
RAM	Random Access Memory
ROI	Region Of Interest
ROS	Robot Operating System
RPM	Rotation Per Minute
RRT	Rapidly-exploring Random Tree
RViz	ROS Visualisation
SAE	Society of Automotive Engineering
sEnglish	system-English
SLAM	Simultaneous Localisation And Mapping
SoM	System on Module
TEB	Timed-Elastic Band
ToF	Time of Flight
V2I	Vehicle-to-Infrastructure
V2V	Vehicle-to-Vehicle
V2X	Vehicle-to-Everything
VCL	Vehicle Control Language
VR	Virtual Reality
YOLO	You Only Look Once

Chapter 1

Introduction

1.1 Background

Autonomous robots are now widely used in various industries, and their operational space is well protected to prevent harm from direct interaction with them. In order to introduce robots into our daily life, we need to create safe systems of robot navigation in their dynamic and crowded environments. Recently, different mobile robots such as the ones shown in figure 1.1 have been developed to work autonomously among humans. These robots have been used in department stores, hospitals, museums, office buildings and other public places.

Current robots are capable of serving various purposes such as the delivery of goods, guidance, assistance in workshops and homes, providing telepresence, entertainment, and cleaning. With regards to transport vehicles, both industry and governments are interested in the development of assisted driving for safety. They are working towards the ultimate goal of the creation of fully autonomous vehicles that could move freely and safely on roads among different objects and other road users. Roads are considered dangerous environments where road accidents show a high rate of deaths in many countries [13] where studies show that more than 90% of all car accidents are caused by human errors and only 2% by vehicle failures [14]. In urban environments, pedestrians and cyclists are the victims in a high percentage

of accidents. Recently, technologies pertinent to safety, have been increasingly added to cars to reduce accidents, courtesy to the ongoing research towards safe autonomy. In reality, only a few examples of fully autonomous driving systems exist under constrained infrastructures, such as autonomously driven metros and some prototypes of autonomous lorries and cars on instrumented streets and motorways. For safety limitations, most of these prototypes are still restricted to tests or under the condition that the driver has to be behind the driving wheel at all times.

Robot navigation in a static and controlled environment is different from dynamic and changing environments where it still represents a significant challenge for robotics research. In a static environment, global path planning is sufficient. In contrast, in highly dynamic environments, difficulties arise which necessitate run-time motion planning and navigation, the detection and tracking of moving objects and their prediction and influence on the future state of the world.

GPS navigation can be complemented by the run-time perception of the vehicle environment, which takes into account possible sources of uncertainty involved in the sensing process. In the past decade, the problem of uncertain, incomplete, and changing information in the field of navigation has obtained further attention in the robotics community; the aim is to develop and include probabilistic frameworks for integration and precise elaboration of the available information.

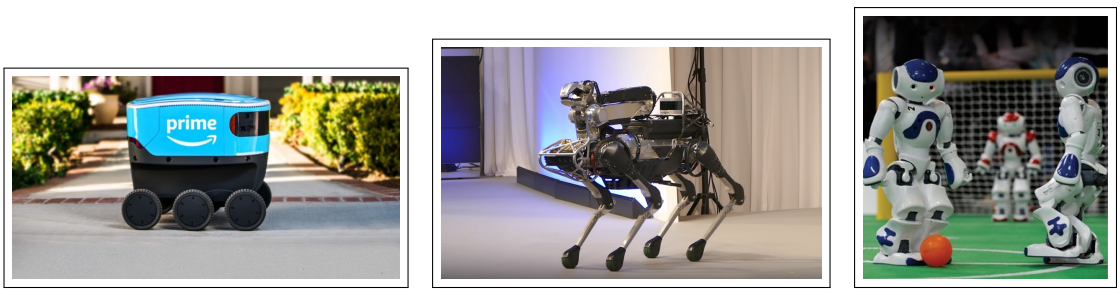


Figure 1.1: Three different kinds of autonomous robots.

1.2 Motivation

Autonomous Vehicles (AVs) are capable of driving without human intervention or supervision in an unpredictable, uncertain, and complex driving environments. It represents the highest level of autonomy and thus the ideal, future end state of these vehicles.

In 2004–2007 Defence Advanced Research Projects Agency (DARPA) sponsored competitions [15, 16] have made a big impact on the development of autonomous ground vehicles [17, 18]. However, results remain limited to non-complex driving environments [19]. AVs which operate in complex active environments require methods that generalise to unpredictable circumstances and for reasoning promptly in order to reach human-like ability and react safely even in complex urban situations, where informed decisions require accurate perception. Some of the DARPA Grand and Urban winning teams vehicles with their perception system onboard are shown in figure 1.2.

The development and deployment of AVs on some of our roads in the near future is realistic and can also bring significant benefits. In particular, it can solve problems related to (i) improvement of traffic congestion, (ii) reduction of the number of accidents, (iii) help in the parking process, and reduction in traveller's time by identifying a free parking space, (iv) encouraging shared use of AVs to reduce overall fuel consumption [20].

Significant efforts are undertaken in industry and academia on hardware and algorithmic research. AVs have to cope with different challenges such as perception, planning and control. Decision-making while driving is a vital process that also needs attention [21]. The primary cause of human accidents comes from wrong decision-making, and there will be no point in developing an AV if those wrong decisions continue to be made at a similar rate as by humans. In [22, 23, 24] we can have a glance at some of the accidents that the self-driving cars have been involved in so far, there are different reasons for those accidents, a common reason

is the misinterpretation of the vehicle's environment, also the wrong decisions to deal with the driving scenario at that particular moment. Hence we focused on making sure that the decision process of our vehicle has been thoroughly verified according to various criteria of safety that has been mentioned in chapter 3.

There are already several AV tests that are being undertaken. Some of them will lead to practical vehicles on our streets quite soon, a few examples are: in Phoenix (USA) the use of a fully driverless taxi service is expected to commence soon [25]; in Singapore's university district there is the world's first self-driving taxi service, which has been operated by NuTonomy since August 2016 [26]; while there are a large number of cars equipped with autonomous driving technology that are expected to see on the streets of South Korea in 2020 [26].

Figure 1.2: Four self-driving vehicles represent top-ranked teams participated in DARPA Urban (top two figures) and Grand (Bottom two figures) challenges.

Software agents have been rapidly developed during the past two decades. Some well-known agent types are reactive, deliberative, multi-layered, and Belief-Desire-Intention (BDI) rational agents [27]. Limited Instruction Set Agent (LISA) [4] which we used in this work is a new multi-layered implementation to Rational Agents (RAs) based on the BDI agent architecture, which is particularly suitable for

achieving goals by autonomous systems. Most approaches to agents of autonomous robotic depend on logic-based reasoning cycles to keep the robot safe and within acceptable behaviour limits [28]. Within the pre-programmed set of rules, it is essential that the robot can establish consistency between its perception-based beliefs, its rules, its planned actions and their consequences.

Despite the increased research activity in machine learning techniques and advanced planning and decision-making methods, verification and guaranteed performance of the autonomous driving process remain challenging problems [29]. Reconfigurable and adaptive RA based control systems have proved capable of robustly progressing a vehicle in space and time to avoid other vehicles and people [30]. Nevertheless, to make decisions with foresight, and consideration to other traffic participants in a social context, integration is essential within overall decision-making based on behaviour rules and experience. Rational agents, which could also be referred to as Cognitive or Autonomous agents, have exhibited significant robustness in their implementation of various applications. However, for real-world critical applications, some safety concerns remain after extensive testing, creating the need for an appropriate verification framework.

The gap for a level 4 AVs is the lack of a simple yet efficient platform for decision-making operation that could be connected to a verification platform to make sure that all the decisions are safe and feasible for run-time vehicle operation. This is what our new rational agent is made for, where we tried to fill the gap for the lack of such a system.

Testing of these systems through prototype development is one approach that attempts to partially answer operational safety questions. The best that can be done in testing is a representative set of scenarios on real vehicles. Simulations can provide validation and illustrations of correct social behaviour of the AV, but they cannot take into account rare combinations of events that may arise during real deployment of the AV. Hence the gap here is the lack of verification methods which are needed to account for low probability scenarios. If good dynamical models are available to represent robotic skills of sensing and action, then formal

verification can rely on a finite interaction model of the vehicle with the dynamics of the environment [31, 32].

We filled this gap by providing a new verification methodology for safe decision-making onboard a Tata Ace vehicle (Ace-EV). Safety aspects are also developed for a prototype system designed for an autonomous parking scenario with the ability to deal with the two most vulnerable type of traffic participants: vehicles and pedestrians. The prototype autonomy level is 4, in that our system can work autonomously in a specific environment until it is interrupted for the task of parking. Here the Society of Automotive Engineering (SAE) levels [33] of autonomy have been used, which can vary from a human-control (level 0) to a fully autonomous system (level 5).

The car manufacturers nowadays depend on their closed-source software tools and platforms to design their autonomous vehicle systems. This considered as a barrier for researchers who want to design an open-source AV system. We have tackled this gap by using the open-source robot operating system (ROS) to design a comprehensive open-source platform for a realistic self-driving vehicle. Autonomous vehicle and self-driving vehicle are used interchangeably in this thesis, and both refer to the same meaning.

In our work, we first designed an AV system and its environment, which are programmed and modelled in ROS [34] and Gazebo simulator [35]. Second, we investigate how a robotic software agent can use model checking through MCMAS [36] to examine the consistency and stability of its rules set during design-time, which involves beliefs and actions specified in Computational Tree Logic (CTL) for a RA that has been implemented within the LISA agent architecture [3, 4]. Third, some of the required RA properties are formally specified through Probabilistic Timed Programs (PTPs) and Probabilistic Computation Tree Logic (PCTL) formulae then formally verified with the Prism model checker [37] in onboard vehicle operations which also referred to as run-time verification process.

1.3 Research questions and methodology

Abstractly, the design of an AV can be divided into two main parts: the high-level and the low-levels [38]. The latter is responsible for the sensors, planners, actuators, and their related devices and algorithms. The former, however, captures the critical decision-making capability that the AV must exhibit while there is no responsible human 'driver'. This high-level decision-making is software responsible (called Rational Agent) for clearly determining the actions that will be invoked at the low-level (referred to in this thesis as Agent Skills).

There are three main research questions that we focus on in this thesis; those are:

1. How we can ensure the safety and feasibility of decisions made by the autonomous vehicle while driving, by using three different approaches: verification, validation and testing?

2. How we can design a simple, feasible, realistic and reconfigurable autonomous vehicle system using the Robot Operating System (ROS)?

3. How to use the ROS-based system to drive a real vehicle in real life driving scenario in a parking lot environment?

The main question and theoretical contribution we have answered in this work is the safety and feasibility of decisions made by a decision-maker onboard the AV. To explain this in details, we have presented in this section the reasons behind the complexity and the sources of possible errors of decision-making operation. This point is also fully covered in chapter 3.

The two other questions or problems that we tackled in this thesis are the simulation and hardware implementation of the AV system, and both are explained in details in chapter 4 and 5.

1.3.1 Autonomous control in a dynamic environment

We designed a new AV in simulation (using ROS) and hardware implementation (on a Tata Ace electric vehicle) that can navigate in an uncertain dynamic and restricted environment represented by a parking lot. This example of a level 4 autonomy in a parking lot environment is mentioned in [9, 39]. We have also designed a new Rational Agent (RA) to drive the vehicle, and we developed techniques to allow the RA to check its decisions while driving in a priori unknown parking lot environment among static and moving objects. The AV's task is to generate and perform a continuous sequence of steering, accelerating and braking actions that lead to its intended destination while avoiding collisions with the other objects around it. Our novel autonomous vehicle system in both simulation and implementation involves perception, prediction and planned action taking while constrained by rules, and is briefly summarised as follows:

- Perception: The AV is equipped with sensors; the odometry data coming from the LiDAR laser scanner gives the vehicle the knowledge about its position relative to the world (heading and position) while the mono cameras and a stereo camera (visual sensors) are used for recognition.

- Prediction: In order for the AV to make decisions, it should make predictions of the immediate future. A dynamic model of the AV's environment is continuously updated, and real-time re-planning is applied to avoid any collisions whiles reaching the desired destination within reasonable time duration and on the shortest path.

- Action: When a movement decision is taken, the sequence of operations obtained by the planner is sent to the vehicle's actuators for steering, movement and braking.

1.3.2 AV perception system

AVs depend on many sensors to find their way among static and dynamic objects; each of those sensors has its strengths and deficiencies. Both cameras and LiDAR are usually used together. LiDAR provides excellent odometry, localisation, mapping and range information but with limits to object identification; on the other hand, cameras provide better recognition but with limits to spatial information obtained [40]. A multi-sensor system can provide reliable information for perception when it performs computations in parallel to provide timely processing of the sensory data [41].

In this thesis, the proposed perception system is used for localisation and mapping, also for calculating the positions of objects around the AV. The architecture of our AV's perception system is divided into three subsystems: The LiDAR subsystem, vision pre-processing, and tracking-classification system with coordinate transformation. The cameras take images for object recognition and aid free parking space detection that the LiDAR complements by checking distances to obtain a more precise occupancy grid than it would be possible by camera-based vision only. The position of the objects found by perception is converted into data in the global coordinate system. In vision pre-processing, a region of interest (ROI) is identified in the image space that is processed further with classification, depth information by LiDAR and object tracking.

1.3.3 Dynamic environment modelling

The AV needs to generate a map for its dynamic environment to enable it to navigate safely in the current and future time. This takes the form of a Spatio-temporal model that takes into account the position, direction as well as the shape of moving objects around it. The initial position and predicted trajectory of these objects can be estimated based on a probability distribution model. To counter the issues of prediction, the perception system needs to make some hypotheses, such as the representation of the surrounding objects in the sensor data, including their behaviour

and motion model. With these hypotheses, the mapping system will be able to distinguish the occupied and free spaces in the surroundings, decide which objects are static and which are dynamic, register the entry of new objects, follow them and stop monitoring the objects that are beyond a defined distance.

1.3.4 Dealing with consistency and stability

In order for the AV to keep itself within a safe and permitted behaviour, the reasoning cycle of its rational agent uses logic inference with a set of rules that the AV should follow. The agent should also be responsible for maintaining the consistency between its rules, perception-based beliefs, the planned actions and their consequences. In this project, we investigate how the RA for the AV can use model checking techniques to check the consistency and stability of its different predicates. A set of rules has been modelled using a Boolean Evolution System (BES) with synchronous semantics that can be interpreted as a Labelled Transition System (LTS). The consistency and stability of the logic system have been formulated in Computation Tree Logic (CTL). In this project, the RA of our AV has a set of rules, beliefs, and actions, and hundreds of logic statements to be programmed during design-time. In our approach, the MCMAS model checker is used to check the consistency and stability of the logic statements during design-time, by going through all possible combinations of logic states and making sure that all states are stable and consistent. In case of inconsistency, the MCMAS model checker will generate a counterexample to show a possible conflict between the predicates for sensing, consequences and actions.

1.3.5 Dealing with uncertainty

The navigation problem of the AV traditionally assumes that the geometry, position and motion of the AV and the other objects are accurately known. Also, it proposes that the AV can perform the desired actions without any errors. In the real world, the AV and its environment cannot satisfy those assumptions: the

known information is at best imperfect, actions taken have an error bound, the future trajectory of the moving objects is unknown. Furthermore, the AV could have little or no knowledge about its environment, so it needs to rely on its perception system (sensor observation) to build or refine different models. To deal with the uncertainty of the environment, we adopted a probabilistic approach by generating Probabilistic Timed Program (PTP) models which were regularly updated during run-time to reflect the changes in the environment. Those will be verified online using the Prism model checker to provide the RA with the sequence of actions that will lead to the highest probability of success.

1.4 Contributions

The contribution of this thesis is divided into four categories, which together represent a new safety approach for the decision-making of the designed AV system presented in simulation and hardware implementation:

1. Development and implementation of a Rational Agent (RA) in Limited Instruction Set Agent (LISA) based on Belief-Desire-Intention (BDI) principles in order to control the Autonomous Vehicle (AV) and make decisions in real-time based on feedback control and perception systems. This agent is applied to both the simulated and the real vehicle to guide vehicle movements under physical and rule-based constraints of vehicle movements and the state of the environment. This is the first time a RA is designed in LISA to drive a full-size ground vehicle in a complex environment. This work is presented in chapter 3.

2. A novel theoretical framework, called 'hybrid' for it merges both the design-time verification by the MCMAS model checker and the run-time verification by the Prism model checker. This verification theory provides a comprehensive approach to the verification of AV decisions. These results present a new and efficient way to design a safety system for a real AV. This work is presented in chapter 3.

3. Design, development and implementation of a new open-source physics-based model of an AV system for the Tata Ace electric vehicle and the intended environment based on Robot Operating System (ROS) and Gazebo simulator. This includes design, modelling, and simulation of the installed sensors and the vehicle body and dynamics with error bounds. This model represents a new and complete platform in the simulation of an AV that could be used by vehicle developers and researchers for further developments or to check different algorithms related to autonomous driving. This work is presented in chapter 4.

4. The engineering development and implementation work of the vehicle perception and actuation systems on the Tata Ace electric vehicle to test the feasibility of the above ROS-based control system for future use on self-driving cars. This real AV implementation and its simulation model share the same rational agent and associated planning and control abilities designed for a parking lot environment. This work is to satisfy the testing requirements of the industry and to check the feasibility of the above-described system on a real testbed. This work is presented in chapter 5.

1.5 List of publications

A number of papers and poster presentations were published during the course of the work described in this thesis, which are listed below. Those are mainly my work and written by myself, the other names mentioned apart from my supervisor have reviewed the papers and gave their comments.

Papers:

1) Mohammed Al-Nuaimi, and Sandor M. Veres. "ROS-based hardware implementation of a self-driving vehicle", this journal paper is still under preparation for future publication.
This paper is presented in chapter 4 and 5 of this thesis.

2) Mohammed Al-Nuaimi, Sapto Wibowo, Hongyang Qu, Jonathan M. Aitken, and Sandor M. Veres. "Hybrid verification technique for decision-making of self-driving vehicles", submitted to IEEE transaction on intelligent vehicles. 2019.
This paper is presented in chapter 3, 4 and 6 of this thesis.

3) Mohammed Al-Nuaimi, Hongyang Qu, and Sandor M. Veres. "Towards a verifiable decision making framework for self-driving vehicles." In FLOC: Federated Logic Conference. Verification and Validation of Autonomous Systems (VaVAS) 2018.
This paper is presented in chapter 3 of this thesis.

4) Mohammed Al-Nuaimi, Hongyang Qu, and Sandor M. Veres. "A stochastically verifiable decision making framework for autonomous ground vehicles." In IEEE International Conference on Intelligence and Safety for Robotics (ISR). IEEE, 2018.
This paper is presented in chapter 3, 4 and 6 of this thesis.

5) Mohammed Al-Nuaimi, Hongyang Qu, and Sandor M. Veres. "Computational Framework for Verifiable Decisions of Self-Driving Vehicles." In IEEE Conference on Control Technology and Applications (CCTA). IEEE, 2018.
This paper is presented in chapter 3, 4 and 6 of this thesis.

6) Mohammed Al-Nuaimi, Hongyang Qu, and Sandor M. Veres. "Testing, Verification and Improvements of Timeliness in ROS processes." In Annual Conference Towards Autonomous Robotic Systems (TAROS). Springer, Cham, 2016.
This paper is presented in chapter 2 and 3 of this thesis.

Poster presentations:

1) Mohammed Al-Nuaimi, and Sandor M. Veres. "Verification-driven design, simulation and implementation of a self-driving vehicle." Talk and Poster Presentation in UKACC showcase day. 2019.

2) Mohammed Al-Nuaimi, and Sandor M. Veres. "Verifiable decision-making framework for Autonomous Ground Vehicles." Poster Presentation in Midlands Intelligent Mobility Conference. 2018.

3) Mohammed Al-Nuaimi, and Sandor M. Veres. "Formal analysis and verification of Autonomous Vehicles." Poster Presentation in ACSE PGR symposium. 2016.

Chapter 2

Background

2.1 Autonomous control systems

In systems engineering, continuous research on feedback control over the past decade has led to the development of autonomous or semi-autonomous 'intelligent' controllers. These controllers have the ability of making decisions and executing those decisions in terms of actions, parameters and performance at a certain level of autonomy. The term 'intelligence' is considered as one of the hardest concepts to define, and apply. A general definition of intelligence can however be stated as [1, 42]:

> "An *Intelligence* is the ability of a system to act appropriately in an uncertain environment, where a proper action is that which increases the probability of success, and success is the behavioural sub-goals achievement that supports the system's final goal".

A key feature of an autonomous control system is the ability to behave appropriately under significant uncertainty in both the system and its environment for extended periods of time, and it must have the capability of compensating for system failure without external intervention. Such control systems are evolved from conventional control systems by adding intelligent components. Those autonomous

control systems consist of software and hardware, which can perform the necessary control functions, without external intervention, over extended periods. There are several degrees of autonomy; a fully autonomous controller should perhaps have the ability even to perform hardware repair to its own system when needed; this could be particularly important for autonomous systems to work in extreme terrains such as space and in-depth oceans explorations. Note that conventional (automatic) controllers have a low degree of autonomy since they can only tolerate and control a few plant parameters. To achieve a high degree of autonomy, the controller must be able to perform some functions in addition to conventional control functions such as tracking and regulation. Autonomous controllers can be used in a variety of systems; hence, they enable the host machine to work autonomously in different terrains without the need for observation or supervision [43]. The general architecture of an autonomous system is shown in figure 2.1.

In this chapter, a description of the architecture of our autonomous controller is presented through a rational agent for an autonomous ground vehicle. The control system processes the data coming from the perception system and interacts with the environment through system actuators. The goal of the autonomous controller is to provide a high level of adaptation to changes in vehicle dynamics and the environment and to provide a high level of tolerance to faults to deal with unexpected situations.

An example of an automatic controller is the autopilot of an aircraft that can maintain the desired flight trajectory of the plane. This could be considered easier to implement compared with the autopilot control system for an autonomous ground vehicle that can take most of the driver responsibilities while driving, and this is due to the nature of the environment for both of the controllers.

Techniques for the autonomous system are still under development, and in many cases, they still need human supervision for adequate safety. Furthermore, autonomous systems are becoming more popular as they can sometimes be much cheaper to develop and deploy than human-operated systems [30]. Despite that the idea of giving a machine the freedom to think and make decisions might seem

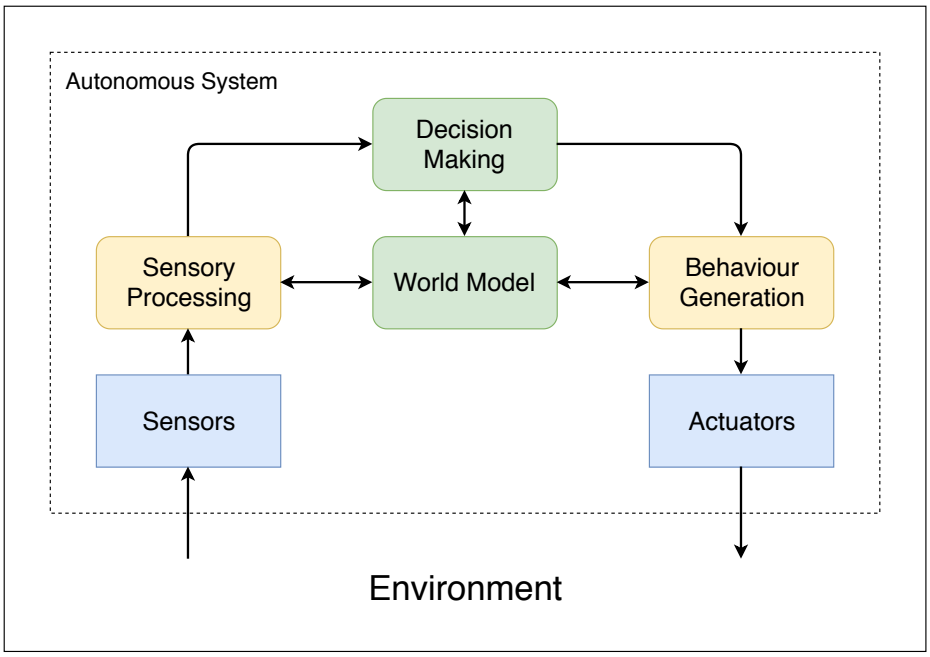


Figure 2.1: General architecture of an autonomous system [1].

dangerous. It is obvious, however, that we require such systems, especially [42]:

- when the robots work in unsafe or difficult environments for humans, such as contaminated areas or deep-sea explorations.
- when the system operates in extremely remote areas such as spaceships, where the signals need a very long time to travel in space, hence a fully autonomous system is needed.
- when the speed of the environment beyond human capabilities to deal with, such as high-speed manufacturing or minute and second trading on the stock exchange.

Another critical issue for the autonomous controller is reliability. The autonomous system should do well in situations that involve uncertainty in dynamics and the environment for a considerable amount of time, and they should be able to learn from system failure without external intervention. Behaviour of this kind is very desirable for developed systems.

The following methods of control system design could be adapted to acquire autonomy [44]:

1. Algorithmic-numerical methods, based on state-of-the-art conventional control, estimation, identification, and communication theory.
2. Symbolic methods of decision-making, for instance, those used in the field of Artificial Intelligence (AI).

A high degree of tolerance to failures should be part of the autonomous controller's characteristics along with the ability to supervise and tune the control algorithm where failures should be detected and isolated to ensure system reliability. The autonomous controller should be able to plan the sequence of procedures necessary to complete a complicated task while having the ability to learn to enhance the performance of the system. Several developed techniques should work together to acquire autonomy such as sensing, learning, and planning, along with conventional control systems [44].

In the next section, we have presented some background information about the autonomous vehicles, including a brief history and current progress, also their autonomy levels based on the society of automotive engineering (SAE).

2.2 Autonomous Vehicles (AVs)

The story of the driverless car is almost as old as the car itself. Probably the first presentation of the idea that captured the attention and opened the people vision to this futuristic concept came in 1939 as a part of the Futurama at the General Motors highways and horizons exhibition at the New York world's fair [45]. At that time, people thought that soon it would be possible to implement this concept on the ground - at least that was the vision.

This idea comes to light again after the vast development within a few decades in computer and communications systems which moved from big, expensive and

limited systems to those that were small, cheap and widely available. In 2004, the U.S. military DARPA department launched several competitions for driverless vehicles. While no vehicle managed to finish the 240 km path through the desert in the first run of the grand competition, in the second run which came the year after, five vehicles managed to reach the final destination, the fastest at an average speed of 30.7 km/h. In the third run, which happened two years later, and called the Urban challenge; four cars were able to finish a 96 km urban area road while following traffic rules. These successes not only attracted attention from different research groups from around the globe, e.g. [46, 47, 48], but also from the automotive industry, as can be seen in the huge investment from the car manufacturers and tech companies into the autonomous driving technologies.

Legally, self-driving cars are still in a Gray area. For example, at the time of writing this thesis, self-driving in highways and general driving scenarios are allowed as long as the driver agrees to supervise the vehicle at all times. Since a self-driving system tempts the driver to other tasks and since continuous observation over a prolonged time without involvement is physiologically very difficult to impossible, hence it still unclear if this shifting of responsibility by the car manufacturers will be successful.

In the past years, several proposals for the autonomy classification of vehicles were made, e.g. among the most notable ones: National Highway Traffic Safety Administration (NHTSA), 2013 [49]; Society of Automotive Engineering (SAE) International, 2014 [50]; SAE International, 2016 [39]; SAE International, 2018 [9]. These days, the latest categorisation by SAE International is usually the dominant standard. It differentiates six levels (0 to 5) of autonomy based on how much human intervention and monitoring is required and in what situations, see table 2.1 for further details. Following this categorisation, the above definition of AV is applied to any vehicle of SAE International (2018) level 4 for predefined areas and level 5 for all domains.

Level	Name	Definition	Execution of steering and acceleration/ deceleration	Monitoring of driving environment	Fallback performance of dynamic driving task	System capability (driving modes)
Human driver monitors the driving environment						
0	No Automation	The full-time performance by the human driver of all aspects of the dynamic driving task, even when enhanced by warning or intervention systems	Human driver	Human driver	Human driver	N/A
1	Driver Assistance	The driving mode-specific execution by a driver assistance system of either steering or acceleration/deceleration using information about the driving environment and with the expectation that the human driver perform all remaining aspects of the dynamic driving task	Human driver and system	Human driver	Human driver	Some driving modes
2	Partial Automation	The driving mode-specific execution by one or more driver assistance systems of both steering and acceleration/deceleration using information about the driving environment and with the expectation that the human driver perform all remaining aspects of the dynamic driving task	System	Human driver	Human driver	Some driving modes
Automated driving ("system") monitors the driving environment						
3	Conditional Automation	The driving mode-specific performance by an automated driving system of all aspects of the dynamic driving task with the expectation that the human driver will respond appropriately to a request to intervene	System	System	Human driver	Some driving modes
4	High Automation	The driving mode-specific performance by an automated driving system of all aspects of the dynamic driving task, even if a human driver does not respond appropriately to a request to intervene	System	System	System	Some driving modes
5	Full Automation	The full-time performance by an automated driving system of all aspects of the dynamic driving task under all roadway and environmental conditions that can be managed by a human driver	System	System	System	All driving modes

Table 2.1: Levels of driving automation definitions [9].

2.2.1 Levels of autonomy

A system that behaves by itself is known as: automatic or autonomous. An automatic system works by following an input given by an operator. The steam engine is an example of an automatic system; it can regulate the amount of torque using a mechanical speed sensor. An autonomous system goes beyond automatic, it can regulate its input without any external intervention, depending on the perception of the environment. In general, autonomous systems show different levels of autonomy. For instance, a complex air conditioning system can have many actuators and sensors and can make many decisions about its operation, but it is obviously "less" autonomous than an AV capable of driving autonomously. SAE standards describe six levels in total for ground vehicles, five with different levels of autonomy and one with none. A similar description can be found in [51]:

- (Level 0 - No Automation): At this level, the driver is in complete control of the vehicle driving operation with no warnings, assistance, or interventions

from any assistance system nether onboard nor remotely operated at all times.

- (Level 1 - Driver Assistance): The driving automation system will help the driver by performing part of the dynamic driving task such as stability control and pre-charged brakes while the primary driving task will be the responsibility of the driver who will exist behind the steering wheel at all times to drive the vehicle and maintain safe operation and also responsible for engage/disengage the driving automation system when needed.

- (Level 2 - Partial Driving Automation): The driving automation system/s will be capable of further assisting the driver using information about the driving environment in certain driving conditions such as some autopilot capabilities and the adaptive cruise control in certain locations and roadways. Again, the primary driving responsibility will be on the driver to maintain safe operation, engage/disengage the assistance systems, and take full responsibility when needed.

- (Level 3 - Conditional Driving Automation): The driving automation systems will further control the vehicle up to the limit of autopilot capabilities for driving the vehicle in certain conditions (for all dynamic aspects of the driving task) while there is no need for the driver to monitor the system, but must be available and conscious to respond immediately and appropriately to a request to intervene when needed. The assistance systems are capable of performing lane changing and holding in some cases; the system asks the driver to take over with sufficient warning when it detects some limits that the system cannot deal with.

- (Level 4 - High Driving Automation): At this level, the automated driving systems are capable of entirely driving the vehicle at all times in specific environments *or* for extended times for all environments up to the purpose of the vehicle, in specific driving applications there is no need for a driver. This means that all situations in the specific predefined driving application should be handled by the automated systems onboard. An example of this level is the

autonomous parking valet, and the geographically prescribed central business district where it is used to deliver supplies such as in the industrial or factory area.

- (Level 5 - Full Driving Automation): This is the upper limit of an autonomous driving system where it is possible to design the vehicle with no steering wheel. Also, there is no need to monitor or control the vehicle at any time. This system should be able to drive the vehicle at all times, in all driving environments and in all environmental conditions that can be managed by a human driver. At this level, there is no need for the driver (at any time) to take control of the vehicle. The vehicle is capable of monitoring roadway conditions for an entire trip while performing all safety-critical driving functions (for both unoccupied and occupied vehicles).

The above mentioned levels of autonomy are the main terminologies used to describe the vehicle. However it is also common in the automotive community to divide the levels of autonomy into 3 categories as below:

- Safe-driving: This level is also known as "partial autonomy" represented in SAE International (2018) as level 1 and 2. It represents vehicles which are still driven by a human driver in all situations, but which have several (combined) advanced assistance systems (e.g. adaptive cruise control, lane assistant). These systems provide an easier, more comfortable, and most of all, safer driving experience compared to non-assisted driving (SAE International (2018) level 0).

- Self-driving: In certain, predefined situations (e.g. highways or in congestion), the vehicle drives autonomously and handles all situations. In these situations, the vehicle can autonomously come to a safe emergency stop. A clear protocol is followed when control transitions from or to the human driver (e.g. before leaving the highway), allowing for enough transition time. This is an equivalent of the "high autonomy" category by SAE International (2018) levels 3 and 4.

- Driver-less vehicle: The vehicle can drive autonomously in all situations on every road (SAE International (2018) level 5) which is also called "full autonomy". This category represents the vision as defined at the beginning of this section.

From the definitions mentioned above, and up to the time of writing this thesis, the law forbids driverless vehicles on the public roads. Even those that are fully equipped with the necessary hardware and software are still considered within level 4 because of the need for a person who can take the driving responsibility in case of emergency.

Although autonomous driving has been an area of research interest for a long time, the DARPA Grand and Urban challenges inspired research community to develop a number of autonomous vehicle testbeds across the academia and the automotive industry. Stanford's Junior [52] provides a testbed with multiple sensors for recognition and planning. It is capable of dynamic object detection and tracking and precision localisation. Other few notable testbed vehicles born as a result of DARPA challenge are Talos from MIT [18], NavLab11 and Boss from CMU [17]. Costley et al. [53] discuss a testbed for automated vehicle research available at Utah State University. Researchers at the University of California have also developed a testbed named LISA-Q [54] (Different from LISA agent implementation presented in chapter 3 of this thesis) for autonomous and safe driving.

The main method used for the implementation of the different AVs during the DARPA Urban challenge is based on a three-layer planning system combines mission, behavioural and motion planning to drive in urban environments. The mission planning layer considers which street to take to achieve a mission goal. The behavioural layer determines when to change lanes, precedence at intersections and performs error recovery manoeuvres. The motion planning layer selects actions to avoid obstacles while making progress towards local goals. Mostly, those systems were developed from the ground up to address the requirements of the DARPA Urban challenge using a spiral system development process with a heavy emphasis on regular, regressive system testing.

An example of this approach is the DARPA winning vehicle (Boss) [17], where the motion planning subsystem consists of two planners, each capable of avoiding static and dynamic obstacles while achieving the desired goal. Two broad scenarios are considered: structured driving (road following) and unstructured driving (manoeuvrings in parking lots). For structured driving, a local planner generates trajectories to avoid obstacles while remaining in its lane. For unstructured driving, such as entering/exiting a parking lot, a planner with a four-dimensional search space (position, orientation, and the direction of travel) is used. Regardless of which planner is currently active, the result is a trajectory that, when executed by the vehicle controller, will safely drive toward a goal. The perception subsystem processes and fuses data from multiple sensors onboard the vehicle to provide a composite model of the world to the rest of the system. The model consists of three main parts: a static obstacle map, a list of the moving vehicles in the world, and the location of the vehicle relative to the road.

The mission planner computes the cost of all possible routes to the next mission checkpoint given knowledge of the road network. The mission planner reasons about the optimal path to a particular checkpoint, much like a human would plan a route from their current position to a destination. The mission planner compares routes based on knowledge of road blockages, the maximum legal speed limit, and the nominal time required to make one manoeuvre versus another.

The behavioural system formulates a problem definition for the motion planner to solve based on the strategic information provided by the mission planner. The behavioural subsystem makes tactical decisions to execute the mission plan and handles error recovery when there are problems. The behavioural system is roughly divided into three sub-components: Lane Driving, Intersection Handling, and Goal Selection. The roles of the first two sub-components are self-explanatory. Goal Selection is responsible for distributing execution tasks to the other behavioural components or the motion layer, and for selecting actions to handle error recovery.

The nearest AV design approach to the work presented in this thesis is the Cognitive and Autonomous Test vehicle (CAT) testbed (which is a model of a Ford

Escape vehicle). The simulation is based on ROS developed by a research group from the University of Arizona [55]. It provides packages for some sensor simulations (2D SICK laser scanner and a mono camera), and it also supports Hardware in Loop (HIL) to connect the system to a physical platform.

Availability of testbeds for vehicle research is not limited to ones mentioned above, but those (apart from the CAT vehicle) lack extensive support for HIL simulation. Also, those testbeds are mainly the result of a collaboration between the academia and the automotive industry providing their own closed source systems.

Most of the approaches presented above has been proposed and demonstrated to work within a particular environment. These approaches cover level 4 of autonomy, the same level that the work in this thesis presented. However, this thesis mostly focuses on the problem of safe decision-making, although it still presenting a new design of level 4 of autonomy based on ROS for an easy and low-cost development system.

It is still important to mention here that the new commercial approach to the design of fully autonomous vehicles (level 5 of autonomy according to SAE levels) is to adapt the deep learning approaches for an end-to-end vehicle controller, this topic is beyond the scope of this thesis and the interested reader could have a look at, for example [56, 57, 58].

In the next section, we have presented some background information about the software that we used to control the vehicle while driving, which is called a rational agent. We also mentioned its different implementations and related structures. Then we covered its formal definition with a simple example to clarify its different aspects. Then we went through the language that we used to define the reasoning cycle of our rational agent, which is called the Natural Language Programming (NLP).

2.3 Rational Agent (RA)

The representation of autonomy can be described by the decision-making ability that includes the generation of data abstractions for logic-based reasoning and symbolic processing along with cognitive modelling for the environment. Other features such as navigation, tracking, path planning, communications and control are considered to be necessary skills for Rational Agents (RAs) [1, 42]. Skills can be defined as sub-programs that are managed and controlled by the RA, these skills are usually connected to sensors as input devices or actuators as output devices to perform different tasks such as perception, planning or motion control of the AV.

An exhaustive common definition for an agent is given in [59], where the term is defined as follows:

> "An *agent* is a computer system that is *situated* in some environment, and capable of *autonomous* action in this environment to meet its design goals. The environment is the set of objects not part of the agent body, with which the agent interacts by sensing and acting".

In this work, we used the term 'rational agent' which could also be referred to as Cognitive or Autonomous agent. Rational means rule-based system that can do reasoning based on logic. The RA definition provided in many references including [1, 29, 42, 59, 60, 61], where the characteristics of this agent could be summarised as follows:

- Re-activeness: Agents are provided with sensory systems that allow them to perceive their environment and respond to expected or unexpected changes in order to meet their final goal.
- Pro-activeness: Agents are capable of carrying on goal-directed behaviour. This means that the agent does not need to wait until there is a change in the environment, but instead, it could take the initiative of action in order to meet their final objective.

- Autonomy: Agents are capable of running without external assistance or intervention. The level of autonomy depends on the type of objectives and the environment where the agent is supposed to work in. In general, we specify goals to the agent; therefore, the generation of plans and sub-goals is ultimately bounded by the goal that we specify.

- Social Ability: Agents can communicate with each other, or possibly with humans, and cooperate with them to increase the efficacy of their specific task.

The two main pillars for an agent-based system are the *agent program* and the *agent architecture* [62]. The agent program is the actual code that built to communicate with a set of skills in order to drive the vehicle. The skills are peace of software and hardware working together to provide the necessary data to the agent in order to make a decision or to receive a decision from the agent and control the vehicle based on it. For example, the perception subsystem is a skill that perceives the environment, analyse the information and produce useful data for the agent to make decisions. Another example is the planning subsystem that receives the information from both the perception subsystem and the agent then plans a route for the AV to move safely and meaningfully in the environment.

The agent architecture describes the outlines or structure of the agent program and the interface with the different levels of skills. In [63] the authors describe the agent architecture as "the backbone of robotic systems". Different applications have different needs, hence choosing the right architecture is vital to reflect the agent program in a correct way. A systematic overview of those architectures have been given in [42, 64] including: purely *logic-based* [65, 66, 67], *situated* [68] or *behaviour-based* [69, 70], *situation calculus* [71, 72, 73], *Belief-Desire-Intention (BDI)* [74, 75, 76]. These different architectures are not completely distinct. Usually, modern architectures are structured in a *layered* way, those layers represent different levels of abstraction, as explained in [77, 78], and in few implementations in [79, 80, 81].

Figure 2.2 represents the general architecture of an agent-based system. The

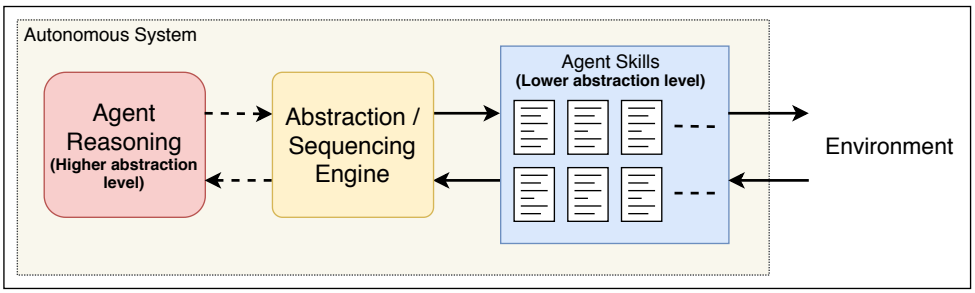


Figure 2.2: General structure for agent-based control system.

agent reasoning represents the top-level part of the overall system (with a higher level of abstraction), which is connected to the lower levels of the system (with a lower level of abstraction), and are referred to as *skills* as mentioned above. The agent controls the vehicle by making decisions represented by issuing action commands to the lower level skills, before that, the agent collects information from the environment using the sensors through the *perception* skill to update the necessary abstracted data called symbolic Boolean variables.

The formal definition of an agent system is described in the following subsection. The RA presented in chapter 3 is based on the BDI architecture, which is the most popular and trusted model [38] for agent reasoning. This agent implementation can be described by three symbolic sets of data: *Beliefs*, *Desires* and *Intentions*. The Beliefs set represents the knowledge derived from sensors to provide an observation of the current state of the environment (the information that the agent has about the world). The Desires set to correspond to the long-term goals that the agent wants to accomplish using predefined sets of actions that the agent *might* perform to accomplish the task. The Intentions set represents the short-term goals the agent is committed to working towards.

There are different agent implementations along with their agent programming languages, based on the BDI paradigm. The main implementations developed in a BDI approach for programming rational agents include: Logic-based Procedural Reasoning System (PRS), Jason [82, 83], 3APL [84, 85], Brahms [86], Jadex [87], Gwendolen [88], and GOAL [89, 90]. Rational agents belong to one of these implementations usually have a set of beliefs, a set of plans, and a set of goals. Plans

determine how an agent can act based on its current beliefs and pre-programmed goals. This method forms the basis of practical reasoning in such agents. Executing a plan will be reflected on the current beliefs, and the short term goals of an agent may change while the agent performs actions in its environment. In [91, 92], the authors have mentioned the role of plans in the reasoning process in more details.

A new agent implementation of the BDI architecture (which is a modified version of Jason) is called Limited Instruction Set Agent (LISA) [4], which has been used in this work. We have described LISA in details in chapter 3 of this thesis.

2.3.1 Formal definition of a rational agent

The Rational Agent (RA) definition of our autonomous vehicle follows [42, 59, 93] and is based on BDI architecture for robotic agents. The basic principles of this agent design are defined as:

Definition 2.1 (**Rational BDI agent**)**.** A rational agent in the BDI architecture can be fully defined and implemented by listing the following characteristics:

$$\mathcal{A} = \{B, B_0, L, \Pi, A, A_0\}$$

where:

- B is the atomic belief set, the set of all possible beliefs that the agent may encounter during operation. The current belief base at time t is defined as $B_t \subset B$. During operation, beliefs may be changed. This occurs through *events* so that at time t, beliefs may be added, deleted or modified. These *events* are represented in the set $E_t \subset B$, which is called the *Event set*. Events are divided into *internal* or *external* events. Internal events are described as "mental notes" if they are generated by internal actions. External inputs will appear through input from a sensor hence called "percepts" as they represent a measurement of the environment.

- B_0 is the *Initial Beliefs* set, the information about the world that is available to the agent at the first iteration. Initially, once the agent is initialised, it will have a set of beliefs about the environment.

- $L = R^P \cup R^B = \{l_1, l_2, \dots l_{n_l}\}$ is a set of *implication rules* and it consists of both the Physics based Rules R^P and the Belief Rules R^B. These are logic-based and represent a description of how the beliefs B can be linked together and interpreted. It describes *theoretical* reasoning about physics and behaviour rules to enable the agent to adjust its current knowledge about the world and influence its decision on actions to be taken.

- $\Pi = \{\pi_1, \pi_2, \dots, \pi_{n_\pi}\}$ is the set of *executable plans* which formulate the *plans library*. At any given time t, there will be a collection of plans π_t which could be activated. These are subsets of the complete plan library, $\Pi_t \subset \Pi$, which is commonly named the *Desire* set. A set $I \subset \Pi_t \subset \Pi$ of intentions is also defined. This set contains plans that the agent is committed to execute. Each plan is built up as a sequence $\pi_j(\lambda_j)$ of actions where π_j is a triggering condition for the plan, and λ_j provides a single or set of actions belong to that plan that will be carried out.

- $A = \{a_1, a_2, \dots, a_{n_a}\}$ is a set of all available *actions*. Actions may be either *internal*, when they either modify the knowledge base (current beliefs) or generate internal events, or *external*, when they are linked to external functions that operate in the environment.

- $A_0 \in A$ is the set of initial actions. The initial actions $A_0 \subset A$ are a set of actions that are executed when the agent is first to run. Typically these actions are general goals that activate specific initial plans set up by the programmer. □

Each triggering condition for the plans in Π is composed of two parts: a *triggering event* 'e' and a *context* 'c', and it is usually expressed in the form '$e : c$'. An *event* is a belief paired with either a '+' or '−' operator to indicate that the belief

is either added or removed. By defining the plan library, a set

$$E \subseteq B \times \{+, -\} \tag{2.1}$$

of events is implicitly defined by the set of all triggering events. The *context* is a logic condition that the agent verifies against the current Beliefs when a plan is triggered. The expression

$$B \models c \tag{2.2}$$

signifies that the Beliefs set B "satisfies" a logic expression 'c', or in other words when the conditions expressed by 'c' are true on B.

The *reasoning cycles* of a Belief-Desire-Intention (BDI) agent are usually operated on indefinite bases (continuously from the time the system start until its shutdown). The following definition introduces the operational sets of a rational BDI agent that are regularly updated throughout the agent operation.

Definition 2.2 (**Operational sets of a rational BDI agent**)**.** Given a rational BDI agent $\mathcal{A}$, if 'time' $t \in \mathbb{N}_{\geq 1}$ is the integer count of reasoning cycles:

- $B[t] \subset B$ is the *Current Beliefs* set, the set of beliefs available at time t. Beliefs in $B[t]$ can be negated (usually with a '\' symbol).
- $E[t] \subset E$ is the *Current Events* set, which contains events that are active at time t.
- $D[t] \subset \Pi$ is the *Applicable Plans* or *Desires*, which contains all plans π_j such that $B[t] \models \pi_j(0)$.
- $I[t] \subset \Pi$ is the *Intentions* set, which contains plans π_j that the agent is committed to execute. Any plan stays in the Intentions set until all the actions listed in it have been executed, unless a *plan withdrawal* action is issued to cancel the plan. □

For most BDI agent implementations the *reasoning cycle* is operated as follows: At the beginning of every cycle, $B[t]$ is updated by checking for external inputs

and *internal actions*; from the changes that happen at each *reasoning cycle* to the *current beliefs*, a set of *events* is generated and added to $E[t]$. The *plan library* is then search for plans that feature a *triggering condition* that satisfies the *current beliefs* set ($B[t] \models \pi(0)$). These plans are then copied to $D[t]$. A single plan from the *Desires* set is then selected for execution and added to $I[t]$. Then the agent takes applicable actions from the active plans in $I[t]$ and executes one (or more) of them. At this point the cycle is complete and $B[t+1]$ is generated.

To illustrate the above definitions, we have provided the example below for an AV in a parking lot scenario.

Example 2.1. An AV is left in a parking lot, so the *initial belief* is that the AV is in a parking lot and it should start looking for a free parking space. The AV starts moving, and in the meantime, a pedestrian is moving nearby, the agent will get this information from the perception system as an *external event* occurred at time t, and this will trigger the associated *belief*. Based on the *current beliefs* set and the set of *rules* (R^P and R^B) the agent will choose a *plan*, the *external event* also called the *trigger event* of a *plan* that is available to the agent in order to proceed to the target, a *plan* has an *action* or a set of *actions*, such as *move forward for* $2m$ *then turn right for* 90° and so on. The *beliefs* set will keep updating based on the information coming from the perception system as an *external events* or from an *internal events* such as the battery level of the AV. The agent will keep monitoring those *beliefs* and compare it with the AV *rules* then choose a *plan* from the *set of plans* available to the agent where each *plan* consist of a *single action* or a *sequence of actions* to execute and keep proceed until reaching the final destination.

2.3.2 Decision making in rational agents

The approach of agent-based decision-making, which is also called Agent-Oriented Programming (AOP), is originally evolved from Object-Oriented Programming (OOP). The first example of OOP, designed with decision-making capabilities, is probably given in [94]. A more recent example of this kind of decision-making can be found

in [95], where it presents a layered control architecture of OOP, consisting of deliberative, control execution and reactive layers. Usually, the OOP framework is linked to Hybrid Systems (HSs) modelling software. Some example applications of this kind can be found in [96, 97, 98].

A vital feature of an AV is the ability to process multiple objectives at the same time, such as planning, collision avoidance, and decision-making to choose the best action to execute. On the other hand, the AV should behave in ways that are robust to kinds of uncertainties, such as sensors affected by noise, unpredictable obstacle movement, wheel slip, etc. To process a specific task successfully, the AV must be able to make a plan, proceed through the steps of the plan, and decide when to reinitialise the plan. A critical plan in almost any AV is to find a collision-free path. During execution, other sub-goals are involved in processing unexpected changes in the environment. In general, rational agents have to deal robustly with uncertainties. To overcome uncertainty, several approaches have been presented. In [99], an example is presented using an extension of Markov Decision Processes - Partially Observable Markov Decision Processes (POMDP). A POMDP model the uncertainty in navigation, including actuators and sensor uncertainties and approximates the current configuration knowledge. Related work in this field [100, 101] is aimed to find the best sequence of actions to reduce localisation uncertainty.

Model Predictive Control (MPC) is considered to be an alternative approach to model uncertainty within a planning framework [102]. The representation of uncertainty is estimated in the frame of the Extended Kalman Filter (EKF) [103]. MPC also used for lane-keeping and obstacle avoidance [104], also as a steering wheel controller [105]. Note however that the MPC approach does not accommodate a reasoning system with memory which agents do. MPC also cannot handle advance conceptual perceptions, long term goals and plan execution. It also does not have a computational framework that combines both discrete and continuous states. Hence agents represent a higher-level framework that is suitable to apply for AVs reasoning system while the MPC approach could only be used to control individual subsystem as mentioned above.

2.3.3 Natural Language Programming (NLP)

Natural Language Programming (NLP) is an ontology-assisted way of programming in terms of natural language sentences, e.g. English. A structured document with content, sections and subsections for explanations of sentences forms a NLP document, which is actually a computer program. The smallest unit of a statement in NLP is a sentence. Each sentence is stated in terms of concepts from the underlying ontology [106].

An AV's decision-programming is complicated, time-consuming and error-prone, and requires expertise in both the AV platform and the intended tasks. Within the automotive industry, there are many vendor-specific tools and programming languages, which require specific proficiency.

In this work, we aim to provide an AV decision system that could be readable and understandable by any driver interested. We are also looking to extend the driver's understanding of how the decision-making is performed for vehicle choices it makes. This also enables to check the system by law enforcement authorities, insurance companies and lawyers (before a lawful use of the system and after an incident to investigate). We used the system-English (sEnglish) meaning-definition-system as a Natural Language Programming (NLP) [107], where English sentences are used in high level programming of agent decisions.

An ontology encompasses a representation, formal naming and definition of the categories, properties and relations between the concepts, data and entities that substantiate one, many or all domains of discourse. More simply, an ontology is a way of showing the properties of a subject area and how they are related, by defining a set of concepts and categories that represent the subject [108].

In sEnglish, an ontology is defined as a formal data framework that matches a set of concepts within a domain; in other words, a vocabulary that the agent uses for reasoning in a specific domain and can be understood by non-experts such as drivers of vehicles. An example of such kind of description is in [93, 107] for autonomy in space missions. sEnglish has been used to build both the agent code

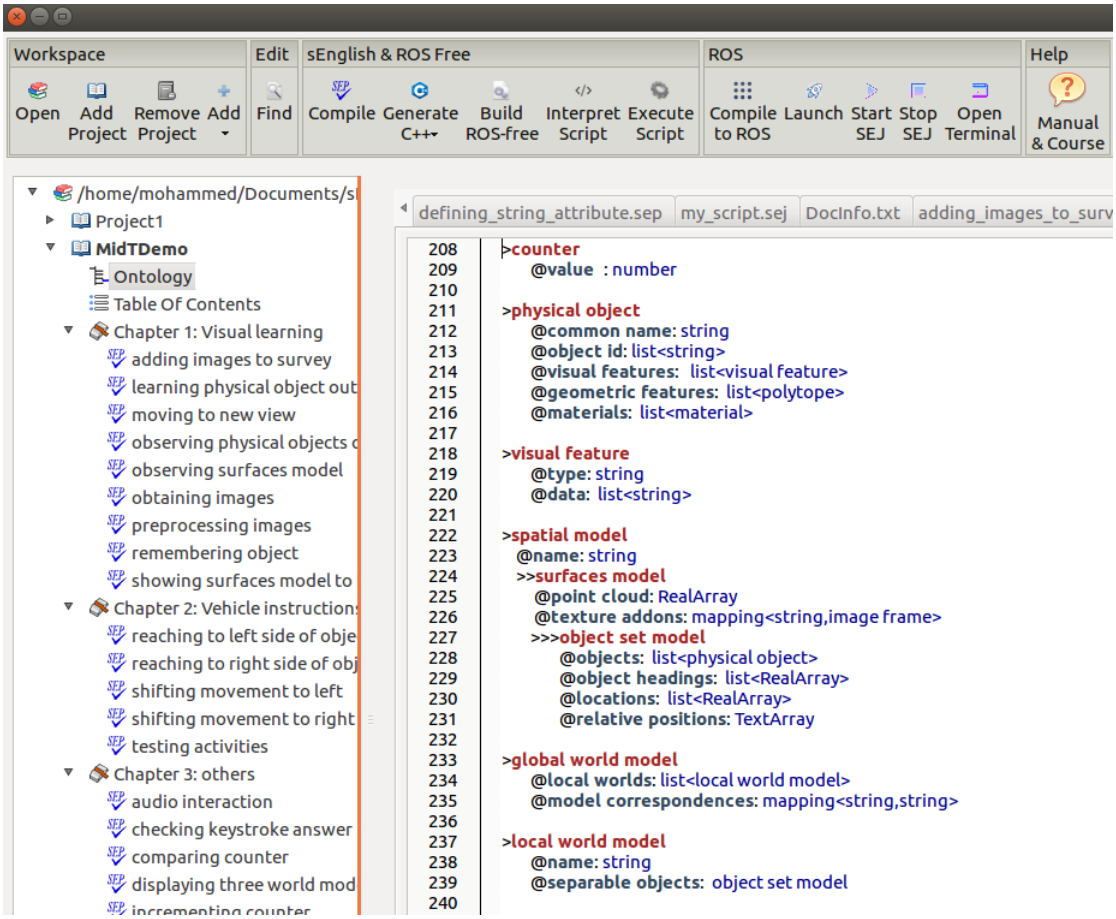


Figure 2.3: sENGLISH editor with an ontology segment in the Machine Ontology Language (MOL) expressing the concepts and related data structures used. *Classes* are indicated by a single '>' symbol, *subclasses* by multiple '>' symbols and attributes by the '@' symbol.

and external functions that could be used to deal with different skills in the system.

The sENGLISH agent description language has been developed to simplify the programming of agents and make their reasoning transparent to people who interact with robots. Figure 2.3 shows a code fragment of Machine Ontology Language (MOL) that has been used in sENGLISH documents. The sentences of the program divided into classes and subclasses, where the classes are indicated by single '>' character and subclasses indicated by two or more '>' characters, according to the programming hierarchy. Every class features attributes indicated by '@' character that is specified a colon ':' followed by a standard data type or another class type.

The programming environment for sENGLISH is called the sENGLISH Publisher.

A suitably written, structured script in an sEnglish document can contain the high-level code of the complete reasoning of a LISA agent in English sentences. The agent can then control either a simulated or real environment containing the vehicle and other traffic participants.

Two types of sEnglish statements can be used: sentences and mental notes. The first is defined within a square bracket, [...], an associated sEnglish document, and the latter is defined within square brackets preceded by a 'hat' operand ^[...]. BDI agent implementations, such as Jason and LISA, can be fully defined and implemented by specifying the Initial Beliefs and Goals, Initial Actions, Perception Process, Reasoning and Executable Plans, see [109].

One of the well described frameworks for AOP is the Cognitive Agent Toolbox (CAT) [110], which supports the development of agent reasoning with NLP in sEnglish and is also used to link multiple external software suites such as Matlab and Prism creating a unified agent framework that supports the verification process. An sEnglish document is represented by a *reasoning file* and multiple *action files*. An sEnglish reasoning file is structured in multiple sections as follows: `INITIAL BELIEFS AND GOALS`, `INITIAL ACTIONS`, and `PERCEPTION PROCESS`, these sections are used to describe and configure the model of the world, `REASONING` represents a list of the logic-based implication rules, while `EXECUTABLE PLANS` represents the sets of Plans or Plan Library.

The action files list and describes the available actions; those actions could be implemented in different programming languages as a sequence of sEnglish sentences associated with predicates that can be used by the reasoning file.

In the next section, we have defined and presented the three different aspects that we used to check our autonomous vehicle system and those are: validation, verification and testing. Then we went into the details of the formal verification method that is the topic of the next chapter. We mentioned how we verified the logic and decisions of our rational agent during both design-time and run-time operation and the two verification tools used in this work: MCMAS and Prism model checkers and their related algorithms.

2.4 Verification

2.4.1 Verification of dynamic systems

Safety and reliability of autonomous systems need thorough assessment, especially the safety-critical systems. System checking can be divided into three main categories:

1. Validation (usually through simulation);
2. Formal verification;
3. Testing (usually through hardware implementation).

Validation through simulation is a close imitation of the studied system or operation of its process; developing a model is the first act required for simulation. This model should be a well-defined description of the simulated subject and describe its key characteristics, such as its abstract, functions, behaviour or physical properties. Here the model represents the system itself, while the simulation represents its operation over time. The simulation can then be used for validation to check the correctness of the system and its individual components based on proposed scenarios [111, 112]. The simulation might point out to unsafe states during its iterations. In this case, one can show that a system is unsafe. However, one cannot prove that the system is safe if the simulation did not discover an unsafe state since there exist infinitely many possible trajectories due to uncertain initial states, inputs, and parameters. Thus, simulation is not sufficient alone since the trajectory that hits the unsafe set may have been missed.

Testing is usually a practical approach to check if a given system consisting of software and hardware matches an abstract specification of that system. Testing can only be applied to an existing prototype of the system [113, 114]. The same thing mentioned above on simulation is applied here on testing. This approach alone cannot prove that the system is safe. However, testing is necessary as a final

step to check other aspects that simulation and verification cannot go through, such as the behaviour of the system under real-life circumstances.

Formal verification, on the other hand, works on a model of the system (rather than a prototype) and is based on a mathematical proof of the correctness of the model.

In the field of testing, there is an increasing demand in the tools and algorithms development, starting from a formal system specification for the automatic selection and generation of tests. In the simulation approach, there are different simulators for different systems and purposes. Formal verification has been presented to be an essential support for the certification of safety-critical systems [115, 116].

We can divide approaches to the formal verification of systems into two broad classes [59, 60]:

1. Axiomatic (*theorem proving*);
2. Semantic (*model checking*).

Axiomatic verification means to derive a systematical logical theory from the system program that represents the behaviour of that system. This is usually referred to as Automated Theorem Proving (ATP) and uses tools called theorem provers, which deal with the development of computer programs capable of showing some statement (the conjecture) as a logical consequence of other statements (the axioms). ATP is used to establish a logical consequence of a program, using a computer. The ATP system inputs are the theorem statement and a set of axioms (including hypotheses), while the output is a proof that the conjecture is a logical consequence of the axioms. Computers cannot understand the meaning of a statement in any spoken language because the logical consequence is independent of the meaning. In the same way, a computer is just a tool for establishing logical consequence where an ATP system is not capable of 'proving' non-logical consequences [117].

It is possible to reduce axiomatic verification to a proof problem. The difficulty

of this proof problem restricts axiomatic approaches. Proofs are hard enough, even in classical logic; the addition of modal and temporal connectives to logic makes the problem considerably harder. Hence, more efficient verification approaches have been developed. Model checking [118] as the name suggests is based on the semantics of the specification language, while the axiomatic approaches generally rely on a syntactic proof.

The other difference is that the theorem proving method can only be used during design-time. In contrast, model checking can be used during both design-time and run-time verification (both of these approaches have been used in this thesis). Model checking is defined as [2]:

"*A formal verification technique which allows for desired behavioural properties of a given system to be verified based on a suitable model of the system through systematic inspection of all states of the model*".

The power of this approach comes from its capabilities to be performed automatically with the ability to provide counterexamples in case of failure (a model fails to satisfy a given property) which provides necessary debugging information. Besides, the model checking tools have proved to be mature enough to be used by a large number of successful industrial applications [2].

System verification is used to prove that the product or the designed model has specific properties. For instance, a system should never reach a state where it cannot go any further (a deadlock case); this can be obtained from the specification of the system. In case of no matching between the model of the system and one or more of its properties, then the model is considered to be "wrong". Otherwise, the model is "correct". The diagram of the model checking verification approach is shown in figure 2.4.

2.4.2 Verification of autonomous agent

Recently, autonomous transportation systems have entered the public domain (e.g. transportation drones, autonomous cars). The concern arises whether those systems

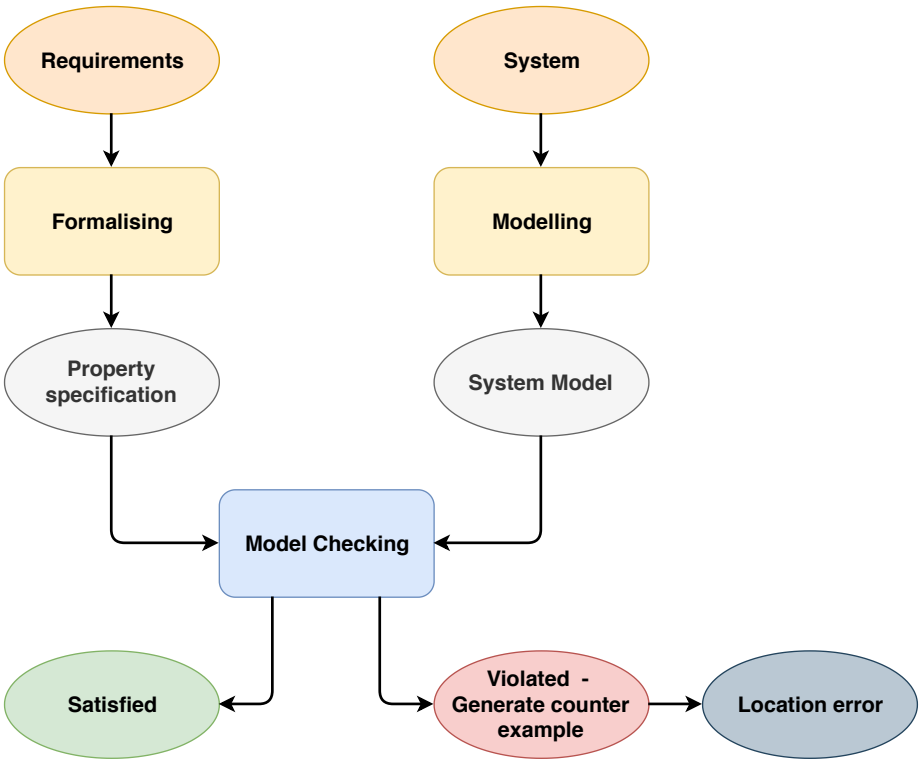


Figure 2.4: Schematic diagram of the model checking method [2]

are efficient, reliable, and most importantly whether they are safe or not. AVs systems can cause accidents with severe damage to property and human life, and such systems are considered as safety-critical. These systems must be certified according to applicable standards as adequately safe before they can be used [1].

While testing is still a necessary part of the verification process, validation through simulation and formal verification are considered vital tools in this domain, mainly at the early stages of design where experimental testing can be considered to be dangerous and infeasible. Simulation is similar to an implementation; simulation runs are usually incomplete, and it used to show the continuous dynamics and allows to check the behaviour of a system. On the other hand, verification through model checking allows us to formally verify the properties of a finite representation (model) of the system. Formal verification of system behaviour is a growing area due to the wide-spread of agents with a decision-making ability in the field of autonomous systems, and those agents have to be proven safe [31].

The precision of verification models mainly depends on the accuracy of the

abstraction operation. Probabilistic model checking can deal with the uncertainty of the state of the environment, and it needs to be taken into consideration as it can affect decisions made by the agent [31]. Formal verification tools use temporal logic statements to model transitions between discrete states of the environment that are triggered by internal states, actions, and logic-based reasoning of agents.

All three concepts of system checking have been implemented in this work for the ultimate goal of designing a safe and reliable self-driving vehicle. Formal verification is proposed and demonstrated in chapter 3, simulation which is also used to validate the system is developed and presented in chapter 4, testing is deployed through the hardware implementation of the AV system and clarified in chapter 5. A detailed case study is presented in chapter 6.

2.4.3 Verification through model checking

Verification by model checking is inseparable from temporal logic [119, 120]. The model checking problem relies upon the close relationship between models for temporal logic and finite-state machines. The first probabilistic model checkers were proposed in the 1980s and 90s [121, 122, 123]. However, the first industrial-strength algorithms were developed in the early 2000s [124, 125]. It can be represented simply as: given a formula φ of language L, and a model M for L, determine whether or not φ is valid in M, i.e., whether or not $M \models_L \varphi$.

Suppose that φ is the specification for some system, and π is a program that claims to implement φ. Then, to determine whether or not π truly implements φ, we proceed as follows [60]:

1. Take π, and from it generate a model M_π that corresponds to π, in the sense that M_π encodes all the possible computations of π;

2. Determine whether or not $M_\pi \models \varphi$, i.e., whether the specification formula φ is valid in M_π; the program π satisfies the specification φ just in case the answer is "yes."

Usually, the generated model is independent of the given specifications; this means the given model could be used with different specifications without the need to rebuild the model again. However, it is still possible in some applications, to tailor the generated model to given specifications to reduce the size of the model with the cost of generalisation. In case the model does not satisfy the given specification, then the verification software will generate a *counterexample* [126]. The counterexample shows the first state or the set of states and its transitions in the state space that does not satisfy the given specifications.

For the above two steps of verification, the difficulties come in the first step: it is not a trivial task to automate the process of generating a model for the system under investigation, assuming that the second step of verification could be easier to perform with the help of the model checker tool. We have discussed a possible method to alleviate this problem in chapter 3. In the next two sections, we covered the two model checkers used in this work. The first is the MCMAS model checker which is used during the design-time phase to check the beliefs, rules, and actions for stability and consistency. The second is the PRISM model checker which is used during the run-time operation to check the probability of success for the intended action before execution. The detailed process for both model checkers is given in chapter 3.

It is essential to mention here that our interest is to verify the logic-based rules and decisions of a rational agent. There is another approach in model checking that has been used by others to assess the safety of autonomous vehicle movements through geometric and mathematical models of both the AV and other static or dynamic objects around. This method called Reachability analysis [127] and it is used to determine the set of states that a system can reach, starting from a set of initial states under the influence of a set of input trajectories and parameter values. Reachability analysis depends on mathematical models of the trajectories for nearby objects, an example of this approach [128, 129, 130] is to calculate the reachable positions of other traffic participants and verify that it does not intersect with the trajectory of the AV for a short prediction horizon, then safety can be guar-

anteed for that time horizon. This method is similar to our run-time verification; however, in reachability analysis, there is no guarantee that the other objects will behave as predicted. Hence we supported our verification method with a design-time rule-based verification of the AV's actions that prevent collision with other objects around, adding another layer of protection.

2.4.4 MCMAS model checker

MCMAS is a symbolic model checker for multi-agent systems. It enables the automatic verification of specifications that use standard temporal modalities as well as the correctness, epistemic, and cooperation modalities. These additional modalities are used to capture the properties of various scenarios.

MCMAS [36, 131] is specifically developed for agent-based specifications and scenarios. It supports specifications based on Computation Tree Logic (CTL) among others as described in [132, 133, 134]. The model input language includes variables and basic types, and it implements the semantics of interpreted systems, thereby naturally supporting the modularity presented in agent-based systems. MCMAS implements Ordered Binary Decision Diagrams (OBDD) based algorithms optimised for interpreted systems and supports fairness, interactive execution (both in explicit and symbolic mode) and counterexample generation.

Agents can be described in MCMAS by the Interpreted Systems Programming Language (ISPL). The approach is symbolic and uses OBDDs, thereby extending standard techniques for temporal logic to other modalities distinctive of agents.

Interpreted systems [133] provide the formal semantics for MCMAS programs. In this formalism, each agent is characterised by a set of local states and local actions that are performed based on a local protocol and the situation of its environment. The system evolves based on an evolution function starting from the initial states and determines the changes of the local states of an agent as a result of its local actions and the actions of other agents around. The evolution of all the agent's local states can be expressed as a set of runs to a set of reachable states. These

can be used to interpret formulae involving epistemic and temporal operators for reasoning about the correct behaviour of the agents, and the CTL operators to expressing states of affairs that agents can enforce.

Computational Tree Logic (CTL)

Computation Tree Logic (CTL) and Linear-time Temporal Logic (LTL) [135] are popular logic for verification of transition systems. They are used to specify properties of a system under investigation. LTL deals with one possible future behaviour, while CTL accounts for all possibilities of future behaviours. In this project, we use CTL to ensure stability and consistency using efficient implementation techniques of CTL model checking.

Definition 2.3 (**Computational Tree Logic (CTL)**)**.** Given a countable set P of atomic formulae, the language $\mathcal{L}_{\mathbf{CTL}}$ of *Computational Tree Logic* **CTL** [136, 137], is given by the following grammar [138]:

$$\begin{aligned} \varphi ::= p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid EX\varphi \mid EG\varphi \mid E(\varphi_1 \, U\varphi_2) \mid \\ EF\varphi \mid AF\varphi \mid AX\varphi \mid AG\varphi \mid A(\varphi_1 \, U\varphi_2) \end{aligned} \tag{2.3}$$

In this definition, $p \in P$ is an atomic formula; $EX\varphi$ means "a path exists such that at the next state φ holds"; $EG\varphi$ means "a path exists such that φ holds globally along the path"; $E[\varphi_1 U \varphi_2]$ means "a path exists such that φ_1 holds until φ_2 holds". Notice that CTL operators are composed of a pair of symbols: the first symbol (A) is a quantifier over paths, while the second symbol (E) expresses some constraint over paths. Also, notice that EU is a binary operator, that could be written as $EU(\varphi_1, \varphi_2)$.

The second line of the above equation means, respectively: "a path exists such that φ holds at some future point"; "for all paths, φ holds at some point in the future"; "for all paths, in the next state φ holds"; "for all paths, φ holds globally"; "for all paths, φ_1 holds until φ_2 holds". These additional CTL operators could be

used to simplify the specification process of various requirements.

The semantics of CTL is given in terms of *transition systems*: a transition system $\mathcal{M} = \langle S, S_0, T, H \rangle$ is a tuple in which S is a set of states, S_0 is the initial state, $T \subseteq S \times S$ is a *transition relation*, and $H : S \to 2^P$ is an evaluation function. The transition relation T models *temporal transitions* between states: given two states s and s' of S, $s\ T\ s'$ means that s' is an immediate successor of s. It is usually assumed that every state has a successor, i.e., the transition relation T is serial. the satisfaction for a CTL formula φ at state s in $\mathcal{M}$, denoted by $s \models \varphi$, is recursively defined as follows [139]:

- $s \models p$ iff $p \in H(s)$;
- $s \models \neg\varphi$ iff it is not the case that $s \models \varphi$;
- $s \models \varphi_1 \wedge \varphi_2$ iff $s \models \varphi_1$ and $s \models \varphi_2$;
- $s \models EX\varphi$ iff there exists a path π starting at s such that $\pi(1) \models \varphi$.
- $s \models EG\varphi$ iff there exists a path π starting at s such that $\pi \models \varphi$ for all $i \geq 0$;
- $s \models EF\varphi$ iff there exists a path π starting at s such that for some $i \geq 0$, $\pi(i) \models \varphi$;
- $s \models E(\varphi_1 U \varphi_2)$ iff there exists a path π starting at s such that for some $i \geq 0$, $\pi(i) \models \varphi_2$ and $\pi(j) \models \varphi_1$ for all $0 \leq j < i$;
- $s \models AX\varphi$ iff for all paths π starting at s, we have $\pi(1) \models \varphi$.
- $s \models AG\varphi$ iff for all paths π starting at s, we have $\pi(i) \models \varphi$ for all $i \geq 0$;
- $s \models AF\varphi$ iff for all paths π starting at s, there exists $i \geq 0$ such that $\pi(i) \models \varphi$;
- $s \models A(\varphi_1 U \varphi_2)$ iff for all paths π starting at s, there exists $i \geq 0$ such that $\pi(i) \models \varphi_2$ and $\pi(j) \models \varphi_1$ for all $0 \leq j < i$;

□

2.4.5 PRISM model checker

Computerised systems entered many aspects of our life, including safety-critical systems such as avionics and automotive applications. Also taking into consideration the increasing complexity of such systems necessitates the development of rigorous techniques to verify their correctness. Moreover, this analysis should also consider the quantitative aspects of the systems for verification. This includes both probabilistic behaviour and real-time characteristics; for example, strict timing requirements [140].

Quantitative verification techniques have received a lot of attention and progress in recent years. An important modelling formalism for real-time systems is timed automata, where well-known verification tools such as UPPAAL [141] exist. For probabilistic systems, the most commonly used models are Markov Decision Processes (MDPs) [142, 143] and Markov chains [144]. Probabilistic model checking tools such as MRMC [145] and PRISM are widely used and have been successfully applied to the verification of different systems. A general weakness of these tools is that they require the user to develop a model of the system in that tool's custom modelling language. To solve this issue and to encourage the use of these tools, it is important to generate the model of the system for the quantitative verification techniques in the languages used by real system designers. The case for non-probabilistic verification has been progressed in this direction where the model of the system to be used by the model checking tools can now be developed directly from mainstream programming languages such as C and Java [140].

PRISM is a probabilistic model checker [37, 146], a verification tool for modelling and formal analysis of systems that present probabilistic behaviour. PRISM has been used to analyse different kind of systems from different domains, such as planning and synthesis, communication, game theory, performance and reliability, security protocols, etc. PRISM can build and analyse several probabilistic models as listed below, plus extensions of these models with costs and rewards [146]:

1. Markov Decision Processes (MDPs).

2. Continuous-Time Markov Chains (CTMCs).

3. Discrete-Time Markov Chains (DTMCs).

4. Probabilistic Automata (PA).

5. Probabilistic Timed Automata (PTA).

PRISM can be used to build a probabilistic model of a system. It also provides support for automated analysis for a wide range of quantitative properties of these models for querying mainly about probabilities and timing properties, such as, "what is the probability of an airbag failing to deploy on-demand" or "what is the probability that the system will turn off within 3 hours because of failure?", or "what is the worst-case expected time taken for the algorithm to terminate?". PRISM combines both state-of-the-art symbolic data algorithms and structures based on the Binary Decision Diagrams (BDDs). It also includes a simulation engine for discrete events, providing support for approximate/statistical model checking, and implementations of various analysis techniques [147, 148, 149].

In this thesis, we describe a framework for the quantitative verification of software that exhibits real-time, probabilistic and non-deterministic behaviour. In addition to the list of some application domains mentioned above that PRISM can deal with, there are other areas where PRISM could be used to play a critical role in developing a safe system such as [31] where the authors used PRISM to verify the behaviour of an autonomous agent for an unmanned aerial vehicle (UAV). PRISM is a probabilistic model checker that can return answers about probabilities and timing properties based on the described models of the real-time probabilistic system, and the set of queries asked. With this been said, we took a forward step of utilising PRISM for a safety check of decisions made by decision-making system onboard an AV as illustrated in chapter 3.

Markov decision processes (MDPs)

MDPs [148], can describe both *probabilistic* and *non-deterministic* behaviour. Non-determinism could be used as a valuable tool for modelling concurrency; for this reason, an MDP is used to describe the behaviour of several parallel probabilistic systems. Another advantage for non-determinism is when the exact probability of a transition is unknown, or when it is known but not considered relevant. A formal definition of an MDP is as below.

Definition 2.4 (**Markov Decision Processes (MDPs)**)**.** An MDP can be described as a tuple $(S, \overline{s}, Steps, L)$ where:

- S is the finite set of *states*
- $\overline{s} \in S$ is the *initial* state
- $L : S \to 2^{AP}$ is the *labelling function*
- $Steps : S \to 2^{Dist(S)}$ is the *transition function*

□

The transition function *Steps* for MDP is used to map each state $s \in S$ to a non-empty, finite subset of $Dist(S)$, the set of all probability distributions over S (i.e. the set of all functions of the form $\mu : S \to [0, 1]$ where $\sum_{s \in S} \mu(s) = 1$). For a given state $s \in S$, the elements of the transition function *Steps(s)* represent *non-deterministic choices* available in that state.

A *path* is a non-empty sequence in the MDP of the form $s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} s_2 \ldots$ where $s_i \in S$, $\mu_{i+1} \in Steps(s_i)$ and $\mu_{i+1}(s_{i+1}) > 0$ for all $i \geq 0$. $Path_s$ is the set of all infinite paths starting in state s. It is important to resolve both the non-deterministic and probabilistic choices to trace a path through an MDP. It is supposed that an *adversary* makes the non-deterministic choices (also known as a 'policy' or 'scheduler'), which selects a choice based on the history of choices made so far. An adversary A can be defined formally as a function mapping every finite

path ω_{fin} of the MDP onto a distribution $A(\omega_{fin}) \in Steps(last(\omega_{fin}))$. We denote by $Path_s^A$, the subset of $Path_s$ which corresponds to adversary A.

To make the most of MDP verification, it is not enough to only verify the probabilistic behaviour of a single adversary of MDP, but we must also verify meaningful properties by computing the *maximum* or *minimum* probability that some specified behaviour is observed *over all possible adversaries*.

Probabilistic Timed Programs (PTP)

PTPs can be considered as an extension of MDPs with real-valued clocks and state variables, or as an extension of PTAs [149, 150, 151] with state variables [140, 152]. For timed automata formalisms, discrete variables are typically considered to be a straightforward syntactic extension since their values can be encoded into locations.

Given a set $\mathcal{V}$ of variables, let $Asrt(\mathcal{V})$, $Val(\mathcal{V})$ and $Assn(\mathcal{V})$ be a set of *assertions*, *valuations* and *assignments* over $\mathcal{V}$ respectively. Given a set S, let $\mathcal{P}(S)$ be the set of subsets of S and $\mathcal{D}(S)$ the set of discrete probability distributions over S. A set $\mathcal{X}$ of *clock* variables represents the time elapsed since the occurrence of various events. The set of *clock valuations* is $\mathbb{R}_{\geq 0}^{\mathcal{X}} = \{t : \mathcal{X} \to \mathbb{R}_{\geq 0}\}$. For any clock valuation t and any $\delta \geq 0$, the *delayed* valuation $t + \delta$ is defined by $(t + \delta)(x) = t(x) + \delta$ for all $x \in \mathcal{X}$. For a subset $Y \subseteq \mathcal{X}$, the valuation $t[Y := 0]$ is obtained by setting all clocks in Y to 0: $t[Y := 0](x)$ is 0 if $x \in Y$ and $t(x)$ otherwise.

A (convex) *zone* is the set of clock valuations satisfying a number of clock difference constraints, i.e. a set of the form: $\rho = \{t \in \mathbb{R}_{\geq 0}^{\mathcal{X}_0} \mid t_i - t_j \lesssim b_{ij}\}$. The set of all zones is $Zones(\mathcal{X})$.

Definition 2.5 (**Probabilistic Timed Program (PTP)** [140])**.** A PTP is a tuple of the form: $P = (L, l_0, \mathcal{X}, \mathcal{V}, v_i, \mathcal{I}, \mathcal{T})$ where:

- L is a finite set of *locations*;
- $l_0 \in L$ is the *initial location*;
- $\mathcal{V}$ is a finite set of *state variables*;

- $v_0 \in \mathit{Val}(\mathcal{V})$ is the *initial valuation*;
- $\mathcal{X}$ is a finite set of *clocks*;
- $\mathcal{I} : (L, \mathcal{V}) \to \mathit{Zones}(\mathcal{X})$ is the *invariant condition*;
- $\mathcal{T} : (L, \mathcal{V}) \to \mathcal{P}(\mathit{Trans}(L, \mathcal{V}, \mathcal{X}))$ is the *probabilistic transition relation*, where:

$$\mathit{Trans}(L, \mathcal{V}, \mathcal{X}) = \mathit{Asrt}(\mathcal{V}) \times \mathit{Zones}(\mathcal{X}) \times \mathcal{D}(\mathit{Assn}(\mathcal{V}) \times \mathcal{P}(\mathcal{X}) \times L)$$

□

A state of a PTP contains the valuation of L, $\mathcal{V}$ and $\mathcal{X}$, and written as (l, v, t). A new state can be reached by either an elapse of some time $\delta \in \mathbb{R}_{\geq 0}$ or a *transition* $\tau = (\mathcal{G}, \mathcal{E}, \Delta) \in \mathcal{T}(l)$ where $\mathcal{G} \in \mathit{Asrt}(\mathcal{V})$ is the guard, $\mathcal{E} \in \mathit{Zones}(\mathcal{X})$ is the enabling condition, and $\Delta = \lambda_1(f_1, r_1, l_1) + \cdots + \lambda_k(f_k, r_k, l_k))$ is a probability distribution over an *update* $f_j \in \mathit{Assn}(\mathcal{V})$, clock *resets* $r_j \subseteq \mathcal{X}$ and a *target location* $l_j \in L$.

The delay δ must be chosen such that the invariant $\mathcal{I}(l)$ remains continuously satisfied; since $\mathcal{I}(l)$ is a (convex) zone, this is equivalent to requiring that both t and $t + \delta$ satisfy $\mathcal{I}(l)$. The chosen transition τ must be *enabled*, i.e., the guard $\mathcal{G}$ and the enabling condition $\mathcal{E}$ in τ must be satisfied by v and $t + \delta$, respectively. Once τ is chosen, an assignment, set of clocks to reset, and successor location are selected at random, according to the distribution Δ in τ.

When the agent starts a reasoning cycle, it will obtain a set of actions that can be safely applied. If the set contains more than one action, then we use PTP to find the most suitable action for the self-driving vehicle to take. PTP models the dynamic and uncertain physical environment containing the self-driving vehicle itself and other static or moving objects, such as pedestrians and other vehicles.

Probabilistic Computational Tree Logic (PCTL)

PCTL [153] is an extension of the temporal logic CTL with the addition of probability. It is the same as the pCTL logic of [154]. PCTL is used to write specifications

for MDPs. A formal definition of PCTL is as below.

Definition 2.6 (**Syntax of PCTL**).

$$\phi \ ::= \ true \mid a \mid \phi_1 \wedge \phi_2 \mid \neg \ \phi \mid \mathcal{P}_{\bowtie \ p} \ [\psi] \tag{2.4}$$

$$\psi ::= X\phi \mid \phi_1 \ U \ \phi_2 \mid \phi_1 \ U^{\leq k} \phi_2 \tag{2.5}$$

where a is an atomic proposition, $\bowtie \in \{\leq, <, \geq, >\}, p \in [0, 1]$ and $k \in \mathbb{N}$.

A property of a model will be expressed as a state formula while the Path formulas occur as the parameter of the *probabilistic path operator* $\mathcal{P}_{\bowtie \ p} \ [\psi]$. Intuitively, a state s satisfies $\mathcal{P}_{\bowtie \ p} \ [\psi]$ if the probability of taking a path from s satisfying ψ in the interval specified by $\bowtie p$.

Path formulas use the operators X (*next*), U (*until*) and $U^{\leq k}$ (*bounded until*) which are standard in temporal logic. Intuitively, $X\phi$ is true if ϕ is satisfied in the next state; $\phi_1 \ U \phi_2$ is true if ϕ_2 is satisfied at some point in the future and ϕ_1 is true up until then; and $\phi_1 \ U^{\leq k} \phi_2$ is true if ϕ_2 is satisfied within k time-steps and ϕ_1 is true up until that point. □

The use of MDP-based architecture and PCTL is reported in chapter 3 in more detail. We refer the reader to [155, 156, 157] for more details on MDPs and PCTL.

Performance queries

Given a PTP, we can use the following PCTL queries to check its properties:

- $\mathtt{P}_{\bowtie=?}[\mathtt{F} \ a]$,
- $\mathtt{P}_{\bowtie=?}[\mathtt{F}_{\leq T} \ a]$,

where $\bowtie \in \{max, \ min\}$, a is a Boolean expression that does not refer to any clocks and T is an integer expression. The first query asks what is the maximum/minimum probability that a is eventually satisfied, and the second one inquires the

probability that a can be satisfied within a time-bound T. Based on these queries, we can compute the maximum/minimum probability of all target states that satisfy a without time limit or within a bound T. For example, we can ask what is the minimum probability for the AV to move to a specific location within a certain time. A concrete example is presented in chapter 6.

Agent reasoning in a most BDI agent-based system is intrinsically probabilistic in its nature, though often not described as such. Perception beliefs occur with some probability and so do events at time intervals in a stochastic fashion. One can reliably say that the environment can be modelled as a stochastic process against which the success of the agent, with its own decision-making, can be verified using a probabilistic model checker.

A BDI agent of this type is completely defined, in definition 2.1, by listing all beliefs and actions, a set of rules and a set of plans that operate on these beliefs by sequentially executing actions. This is in principle a system with well-defined states and transitions, assuming probability distributions of random inputs are known.

Here the PTP models represent the next possible actions of the AV, (the next steps of the AV), it also generates probabilistic steps of the other agents in the same environment. The model checker will check and compare all the PTPs using PCTL properties to check whether the proposed path or action of the vehicle has any negative consequences on the other agents moving around or on the AV and its passengers.

In the next section, we covered in some details the middleware we used to design our autonomous vehicle system which is called the Robot Operating System (ROS) and the Gazebo simulator where those are fully covered in chapter 4.

2.5 Robot Operating System (ROS)

ROS [34] is a powerful and flexible framework for writing robot software. It is a collection of libraries and tools that have been designed to simplify the task of

developing a robust and complex robotic system behaviour for a wide variety of robotic platforms. A significant advantage of this framework is the fact that it provides an extensive set of implemented algorithms and drivers used in robotics.

The ROS framework is based on a set of nodes that use messages, topics and services to communicate. Figure 2.5 illustrates the connection between the different ROS nodes and the ROS master node (roscore), which is the interior design of each ROS-based system. The ROS core master node is the first node to run in order for other ROS nodes to communicate. When the ROS core node is active and running, other nodes can exchange messages by subscribing and publishing to specific topics or by directly invoking the services and actions of the other nodes as shown in figure 2.6. The ROS-based system structure consists of the following [158]:

- Master node (roscore): The ROS master node works as an intermediate node that supports connections between different ROS nodes. The master has all the information about all nodes running in the ROS platform. It will exchange information of a node with another to establish a connection between them. After exchanging the information between those nodes, the communication will start between the two ROS nodes directly.

- Nodes: A base system unit in ROS middleware is called (Node). Nodes are used for different tasks such as device handling, data processing, or algorithm execution, and they use topics or services to communicate between them. ROS software is distributed into multiple packages. A package can contain one or multiple nodes, and it is usually developed for performing one type of task. A robotic system may have many nodes to perform its different computations: for example, an AV may have nodes for hardware interfacing, processing data from cameras or laser scans, localisation and mapping and other objectives. ROS nodes could be created using ROS client libraries such as *roscpp* and *rospy*.

- Topics: A stream of data used to exchange information between different nodes. It could be used to send a single message or sequence of messages of

one type. These messages could be a system input such as sensor readings or system outputs such as commands for the motor to change the speed. Each topic has a specific message type and a unique name. A node cannot publish and subscribe at the same time with a specific topic; however, there are no restrictions on the number of different nodes publishing or subscribing.

- Services: In this mode, one node (the server) registers service in the system. Later, any other node in the same system can ask for that service and get a response in a way similar to the client-server model. Compared with topics, services allow for two-way communication, where the request can also contain some data.

The main intention behind the building of the ROS framework was to become a generic software framework for robots. Even though robotics research was happening before ROS, most of the software was exclusive to their own robots. Their software may be open source, but it is challenging to reuse.

Access to a first-generation ROS network is not secured, which is a significant security threat for autonomous cars when using ROS. The communication among ROS nodes is not secured; thus, the whole system is vulnerable. Someone who gains access to the car's ROS network can access and alter the car's behaviour. However, this drawback has been addressed in the newest version of ROS (ROS version 2). Even with the present drawback, it can be argued that ROS is the right solution for developing autonomous driving prototypes [159, 160].

ROS has been used in this work to design the AV model and its skills such as perception, planning, and control. It is also possible to connect the AV system designed in ROS to a testbed vehicle through Hardware-In-Loop (HIL) process, as shown in chapter 4 and 5.

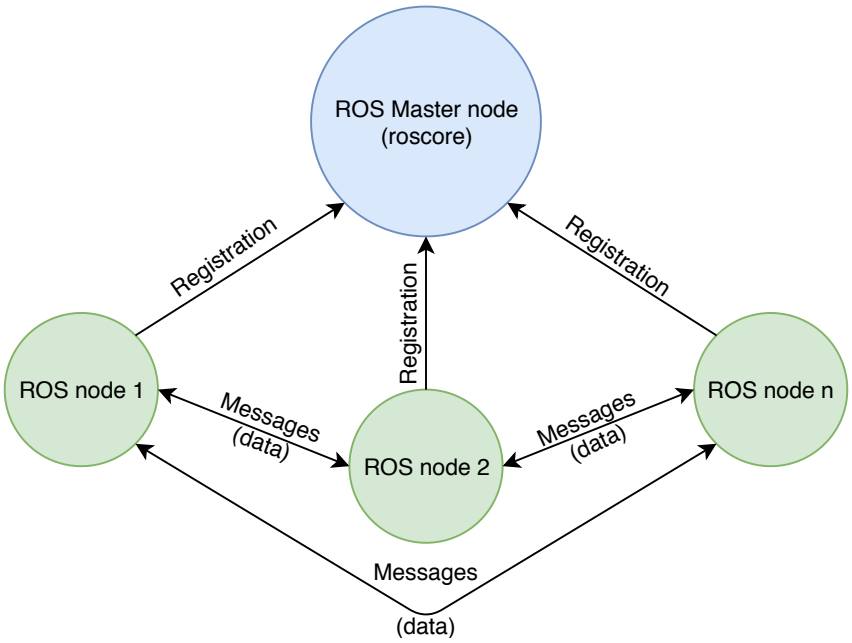


Figure 2.5: Illustration of ROS nodes and messages

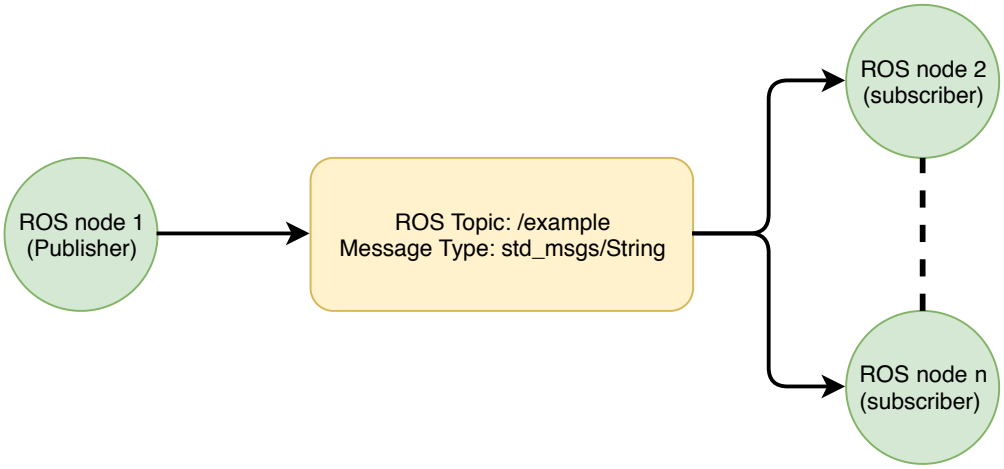


Figure 2.6: Visualisation of ROS concepts

2.5.1 Mathematical model of a ROS package

One way to describe a ROS based system is a tri-partite graph with vertices for nodes, topics and services. New topics and services can be easily introduced that can allow reconfiguration of the system to provide agents with the information they required, albeit sourced from different locations. All node communication must occur through topics or services [161]. A semi-formal definition of the ROS graph is:

Definition 2.7 (**ROS-graph**)**.** A ROS-graph is $G = (N, T, S, E, D, C, X, \lambda)$, where N is the set of vertices representing ROS nodes, T is the set of topics, and S is the set of services, C is a partially ordered set of object classes and X is a set of labels on vertices. $E \subset (N \times T) \cup (T \times N) \cup (N \times S) \cup (S \times N)$ is a set of directed edges to represent publishing of, and subscription to, topics and provision of, and subscription to, services, respectively. $D : E^- \to C^*,\ E^- = T \cup (N \times S) \cup (S \times N)$, is a data descriptor function where C^* is a notation for finite sequences of entries from the set of a data object classes C, which are used in services and topics to send information between nodes. Each of $N,\ T,\ S$ are labelled by a surjective labelling function $\lambda : N \cup T \cup S \to X$.

□

A ROS system enables the nodes to advertise or use services and to publish or subscribe to topics. G represents the maximum ability of the robot when the system has all nodes, topics and services nominally functioning. If some nodes are not available due to sensor, actuator or computational hardware breakdown, then G needs sufficient redundancy to enable continued functioning of the robot or at least some of its functionality. The ROS graph G defines all the possible data flows for sensor readings, signal processing and control activities in the environment.

2.5.2 Gazebo Simulator

Simulation of a robot is an essential tool for developing a robotics system. A well-designed simulator makes it possible to accurately test different algorithms, design robot parts, train an Artificial Intelligence (AI) system using realistic scenarios, and perform regression testing. Gazebo [35, 162] is a physics-based 3D simulator compatible with ROS which offers the ability to accurately and efficiently simulate complex robot systems along with their environments.

Gazebo provides capabilities to build three-dimensional worlds with robots, terrain, and other objects. All are powered by a physics engine for modelling different kinds of characteristics such as illumination, gravity, and other forces. Gazebo could be used to test and evaluate robots in different scenarios, usually quicker than using physical robots in the real world. Gazebo also makes it easier to test other aspects of the robot, such as battery life, error handling, navigation, and different machine learning algorithms [162, 163].

Gazebo has been used in this work to simulate the AV system and it's parking lot environment as shown in chapter 4.

2.6 Conclusion

In this chapter, we mainly presented the techniques, methods and algorithms that we used to design our self-driving vehicle. We mentioned how those could be used together to design the system, and we covered some of their different types. We also presented some other approaches and progress made by others for the design of a self-driving vehicle.

As mentioned in chapter 1, we covered and provided solutions for three important gaps in the field of research.

1. Design, a simple yet efficient software agent, to control and drive the vehicle in a restricted environment.

2. Develop new methods to check the safety and feasibility of the decision-making process of the self-driving vehicle.

3. Provide an open-source based reconfigurable autonomous vehicle system that supports hardware-in-loop by engaging a real vehicle platform, which could be used to validate the design and test different related algorithms.

Here we presented how different techniques are connected and fitted together as one platform simply and efficiently for the ultimate goal of presenting a fully working AV with a novel safety system. The schematic diagram of our AV system is clarified in figure 2.7 that cover the work done in chapter 3 and 4, and partially in chapter 5.

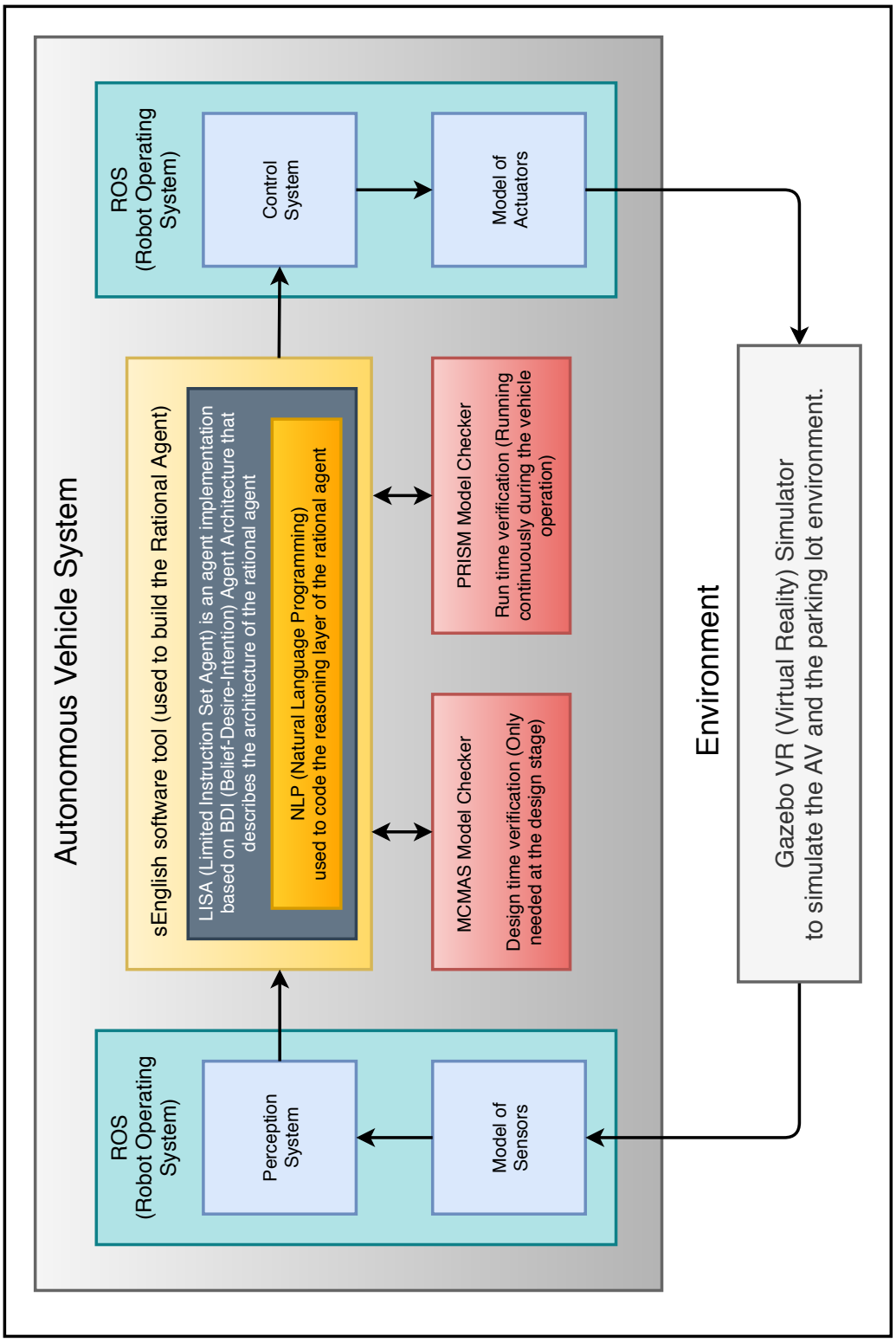


Figure 2.7: Overall system diagram.

Chapter 3

Rational Agent Design and Formal Verification

3.1 Introduction

An agent system can be described as a hybrid agent architecture [38]. Here the term 'hybrid' indicates systems with logical discrete decision-making separated from continuous control in a complete framework [164, 165]. This kind of system utilises logical decision processes to control physical system dynamics. This is usually the primary approach to the design of autonomous control systems where the decision-making process should make appropriate choices to control a system with well-defined functions; see figure 3.1. We can see some examples of such system in [96, 97, 98, 166, 167]. The general definition of a hybrid system is not specific to the logical decision-making; in its broader context it includes a variety of systems that combine both discrete and continuous subsystems working side by side; obviously, most robotic systems represent this kind of Hybrid System (HS) models. In this work, this discrete decision-making process represents a rational agent, able to make justifiable decisions, reason about them, and dynamically modify its strategy when needed [30, 168]. This architecture is operable for agents deployed in a variety of scenarios such as handling nuclear waste [169, 170] or coordinating driverless cars

on motorways [171]. This software agent is responsible for the behaviour of the system, receiving information and taking decisions, thus defining its response to changes in the environment.

The AV should make a 'correct' decisions, quickly and reliably. Consequently, we focused in this chapter on the high-level decision-making process, where an AV decision-maker is modelled as a rational agent. Using this agent-based approach we may write high-level plans for describing the AV decisions and actions and, since these plans are transparent and explicit, we can formally verify some properties related to this agent's behaviour [30], such as "it is always true the AV will stop in the event of an unexpected emergency" or " it is always true that the AV will not break the rules of driving".

The agent systems, in general, are structured in two parts: an agent reasoning and a set of skills. The agent reasoning responsible for applying decisions through actions, and the skills are responsible for gathering the necessary information from the environment and execute those actions on the ground. Verifying those actions before preceding with them represents a significant step to check the safety of the overall robotic system and other objects in the environment.

The Rational Agent (RA) uses cues from the environment in order to make a decision. These decisions are based around a set of *beliefs*, *desires* and *intentions* that define its behaviour [27, 61] as explained in section 2.3.1. Desires correspond to the long-term goals of the agent; for example, a desire for an AV might be to reach a free parking space. Beliefs represent the distillation of information derived from sensors to provide an observation on the current state of the environment, for example, if the sensors of an AV detect a person, the agent will hold the belief that a human is nearby. Intentions, contrasting with desires, represent short-term goals of the agent, for example, once a person is detected, the autonomous agent will have the intention to avoid that person while they are nearby. The agent can satisfy its intentions by having some knowledge and prediction of the state and intentions of the other nearby objects in the environment.

In the second half of the chapter, we will see how the agent decision can be

verified. The verification can be divided into two parts. The first of them is the design-time verification of the agent's predicates (beliefs, rules, and actions), this operation is needed once during the development stage of the agent using the MCMAS model checker. This process is to check the consistency and stability of the agent code and, if appropriate, generate a counterexample showing where the system model does not meet its specifications. The programmer could then correct this repetitively during the design-time stage. The second part of the verification process involves the run-time operation of the agent. It involves finding the most reliable action and checking the probability of its success using the PRISM model checker.

The first step in the run-time verification is to generate models that represent the agent behaviour and other objects' possible behaviour and can be understood by the model checker. The agent model represents the set of rule-based actions the agent is intended to perform, also a set of probabilities representing the probability distribution over the other agents' actions. For this work we have chosen the Probabilistic Timed Program (PTP) that has the same characteristics of Probabilistic Timed Automata (PTA) such as the ability to represent the continuous-time operation of the AV and the nondeterminism, with the addition of discrete-valued variables [140] to represent the path of the AV as a set of points. This is the most suitable model to use because the other models do not satisfy the system needs (probabilistic and non-deterministic). Once these models are generated, the model checker can offer many options to explore the agent possible and safe actions, then return the result to the agent for safe operation. A key point for run-time verification is the ability to introduce probabilistic information about the environment.

This chapter is dedicated to the first research question of 'How we can ensure the safety and feasibility of decisions made by the autonomous vehicle while driving by using the formal verification as the main approach?'.

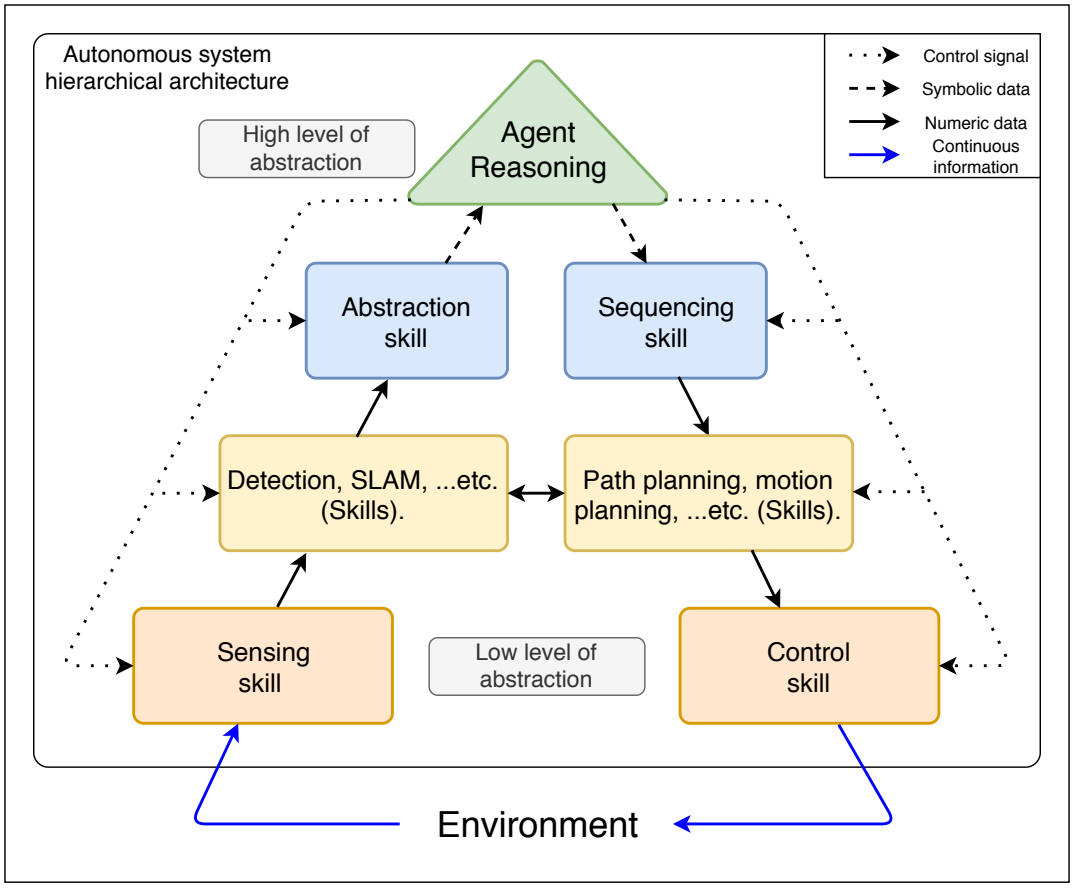


Figure 3.1: Hierarchical structure for LISA-based BDI agent architecture showing the different levels of skills which can interact with each other. The agent reasoning activates and controls each skill.

3.2 Rational agent design

Here we use the notion of a rational agent introduced by Bratman [172] and described in detail by Rao and Wooldridge [168]. Rational Agents (RA) are software entities that perceive their environment through sensing, generate beliefs about this environment, then use these beliefs in a reasoning process. Based on its own mental state, such as its intentions, a rational agent can take actions that may change the environment [168]. A rational agent can be implemented in several ways, but we choose to utilise the popular BDI (Belief, Desire and Intention) architecture [61].

We designed a unified framework which is necessary for developers and programmers to describe the agent reasoning and a model of its environment and to automate the process of generating a verifiable model describing both the agent reasoning and the surrounding environment.

The system presented in this work has a unified framework to model and verify the reasoning process of the LISA agent implementation [4, 173] (based on the BDI agent architecture) that has been designed to facilitate both the design-time and the run-time verification and to support automatic generation of verifiable models for the run-time verification process. It is essential to mention that LISA is not a contribution of this thesis. LISA has been used to facilitate the design of a RA that could be connected to a verification system.

Similar to other implementations of AgentSpeck based BDI architecture, the LISA system has been structured in a layered way with the reasoning layer in the top then subsystems represented by a set of skills in different abstraction layers. Those skills have been developed for the AV to perform specific tasks such as perception, localisation, mapping, and planning, and those tasks are controlled by the agent reasoning processes. Because the agent reasoning can only process symbolic data, we developed an abstraction/sequencing skills that situated between the agent reasoning and the other set of skills as shown in figure 3.1, which are used to convert the numeric data coming from the RA skills to symbolic data used by the RA

reasoning and vice versa.

Although Jason [83] is a suitable, popular and efficient agent implementation for Agent-Oriented Programming (AOP) to develop rational agents, there was a need for developing LISA agent implementation based on Jason. The reason is that Jason was developed with no automatic verification process in mind, means that adding a verification process to Jason implementation is difficult. An alternative and feasible option were to develop a new agent implementation based on Jason that would ease the verification operation. The result is called the Limited Instruction Set Agent (LISA), that we used in this thesis with some modifications (initially it is developed for an autonomous water surface vehicle) to make it suitable for our self-driving ground vehicle system. This agent implementation provides a reasoning operation that was developed to facilitate the modelling process by automatically generate probabilistic models represent the behaviour of the AV and the probabilistic behaviour expected from different objects in the environment that can be verified with a probabilistic model checker. The agent code is developed in a Natural Language Programming (NLP) based software called sEnglish [107] to provide an easy to read the document by both the developer and end-user without much advanced prior knowledge.

It is essential to mention here that the BDI architecture used in this work has both advantages and disadvantages. Starting with its advantages, it uniquely solves the problem of creating a computational framework where decisions about predicted continuous physical phenomena are combined with the application of traffic rules and social behaviours of the convention, which varies country by country. How to drive in a parking lot can be formulated in rules, and there is no need for a more complex paradigm. The verification challenge is to create decision rules which apply in most likely traffic situations. The essence of verification is that we need to account for all eventualities (which are physically much limited for vehicles) in a non-deterministic or probabilistic manner. Hence this paradigm making the verification process much easier compared with a more complex decision-making approach such as deep-learning.

The disadvantage of this architecture is that in its core, it is still a rule-based approach that provides a significant advantage of verification but with the price of generalisation of driving scenarios as it is limited by the number of rules that describe the intended driving scenario. This means poor scalability and iterative design where it needs the developers to start from the beginning if they want to provide an AV that can drive in another environment apart from the intended one. Another disadvantage is that agent-based systems in its current form do not accommodate a learning-based approach, means that the agent cannot learn from its previous mistakes compared with the Artificial Intelligent (AI) based systems that can have such important feature. However, we are not aiming in this work to provide an AV that can drive in general driving scenarios. We are also not providing a general verification framework that can be used with other approaches such as artificial intelligence. Our verification system could only be used with a rule-based system such as BDI architecture.

This AV system, in general, could be referred to as an Intelligent system because it looks forward in time and based on run-time calculation, it can make decisions. Those decisions are not pre-calculated; they actually calculated during run-time of the vehicle making the AV have the ability to predict and see the future.

In this thesis, we will go through the steps of how we designed a LISA based RA that is capable of guiding and driving our AV safely in a restricted environment represented by a parking lot.

3.2.1 Agent architecture

Our AV system is consists of two interacting elements: the rational agent logic (reasoning cycle) and the system skills that the agent needs to use to access the environment. Functionally, it should be possible to abstract properties from the dynamic system to provide the necessary abstractions for the agent logic to operate and reason over in a sense-reason-act loop.

This hybrid system design deals with two types of data. The higher-level part

of the system represented by the agent reasoning processes that can only deal with symbolic, discrete and highly abstracted data while the lower-level part represented by the skills of the agent can deal with numeric, continuous or discrete data at a lower form of abstraction. Middle layer skills do perform this conversion between the two forms of data types called the abstraction and the sequencer. The agent then will be able to understand the situation of the environment and compare it with the sets of beliefs, rules and objectives to decide what action to choose for the next step. After choosing a suitable action from the actions list, the agent then sends the commands to the sequencer skill to translate those commands to a sequence of data that could be understood by the planning and control subsystems of the AV to apply the selected actions on the ground.

The agent skills can operate in a single execution or a continuous execution mode. Single execution is when the program is only required to run once then to stop, send the generated result and wait for another request from the agent reasoning to run again. Examples include the startup check of the system components and the battery level check of the AV. The run repeated skills are more involved with the autonomous driving operation such as the perception, localisation, planning and control, where those need to keep monitor, process and send/receive data to/from the agent continuously. The agent program offers the possibility to define such sets of skills and actions that can be used for such purpose. Here we have examined the six categories of agent skills in more details:

1. **Sensing**: Sensing or state determination represents a fundamental skill which is the ability to determine the dynamic and kinematic state for the AV and other moving objects in the environment. The information on the state of the environment comes continuously from the physical sensors onboard the AV. A sensor is amenable to some noise; hence, this agent skill is also responsible for translating the noisy environmental data coming from sensors into a form of data that next skill (subsystem) can deal with.

   An example of sensing skill for the AV is the detection and recognition of different objects. Sensor fusion is also part of the sensing operation that produces

more accurate, more complete information based on the data received from the different sensors onboard the AV. The data generated from the sensing skill is continuously transferred to the other skills for further processing based on the autonomous driving operation requirements until reaching the final stage of conversion to a form of understandable data by the agent reasoning operation.

2. **Detection and Simultaneous Localisation And Mapping (SLAM)**: These primary skills, among others (Path Planning and Motion Planning) are the backbone of the autonomous driving operation. They work by processing the data coming from the sensing subsystem in order to establish a clear idea about the state of the AV in its environment and the states of other objects around. This operation includes the detection of different objects around the AV (this include classification and localisation of those objects), also the SLAM [174, 175, 176] to build a map for the environment and to find the position of the AV inside that environment at the same time. More detailed information is presented in chapter 4.

3. **Abstraction**: This intermediate subsystem situated between the previously mentioned subsystem and the agent reasoning. The abstraction skill converts streams of numerical input variables into a higher-level pre-defined set of Boolean variables that the agent reasoning can process. An example is when the perception system detects that the AV has reached the destination. It will send numeric data to the abstraction skill to translate it to symbolic representation such as 'I am near the destination' or 'I am at the destination'. It is also responsible for translating the external messages coming from the other agents or the system operator into belief commands in the current belief set.

4. **Sequencing**: This skill is doing the opposite work from the abstraction skill. Once the agent reasoning makes a deliberation and issues a command in symbolic form, the sequencer will translate this command into a numerical data-type for the control system to understand. For example, if the agent reasoning

of the AV issues a command such as '`Go to point X`', Sequencing skill would retrieve the numerical counterpart value of the variable '`X`', then send these data to the next planning algorithm to generate a sequence of safe waypoints, and finally pass this information on to navigation or control subsystems.

5. **Planning, and exploration**: The agent is responsible for high-level commands, such as: '`Explore the parking lot`' or '`Go to point Y`', but it does not know precisely how to perform the continuous sequence of steps in order to reach '`Y`'.

   Planning a route between two points represents the invocation and execution of the relative code as planning routines exist external to the feedback control mechanisms operating on hardware. Here is an example of a simple tasking process that may get triggered on a discrete instance to provide a feedback control with the required information. For planning a routine in a high-level agent implementation, a proper NLP sentence is '`Generate timed path TP0 from current state vector Xnow to desired state vector Xdes`'. In order to generate a sequence of safe waypoints, this sentence is then translated in the sequencing skill then passed to the Planning algorithm to retrieve an updated map from the SLAM skill, and so on.

6. **Control**: This subsystem used to control the AV actions in the environment. They start when the command issued in the agent reasoning is translated to some understandable numeric data form and then sent to the control skill for actions execution. The effect of this skill is monitored by the sensing operation and reported back to the agent for possible correction if necessary. The control skill represents the interface of the agent with physical actuators and hardware that influences the environment, for example, the vehicle motors. An example would be a 'waypoint following' that makes sure that the vehicle is heading towards the next waypoint defined by the path planning algorithm.

3.2.2 Agent reasoning

The agent reasoning block shown in figure 3.1 represents the core of the decision-making process for the agent-based system. It links the information available to the agent from different resources with the set of available actions in order to act and affect the surrounding environment to bring the agent to the desired state pre-programmed in the agent core knowledge [62]. The agent reasoning is operated in iterations called reasoning cycles. The agent code is developed with the NLP called system-English (sEnglish) [107].

The main reason behind the development of the LISA system was to provide automatic modelling, and probabilistic verification of agent actions by the inclusion of pre-programmed probabilistic distributions of the possible behaviour of other agents in the agent code based on prior observations or knowledge. The agent will then include this information within the generated probabilistic models to be used by the probabilistic model checker.

Figure 3.2 shows a schematic representation of the reasoning cycle of the agent, the reasoning cycle $\mathcal{R}$ can be summarised in the following 5 steps [3, 173]:

1. **Current Beliefs update**: The reasoning cycle starts by updating the current beliefs set from the recent data available. This step is done by the belief update function denoted f_{BU} in figure 3.2, as a result, the current beliefs set is updated from $B[t-1]$ to $B[t]$. The f_{BU} receives the updated set of beliefs then compares it with the pre-programmed instructions to check what it should do with each belief; i.e. either add them or delete them from the current beliefs set. The information comes from either the incoming messages, sensory perception, and action feedbacks of external actions, or from the internal actions generating mental notes.

2. **Current Beliefs review**: The generated events from the update of the current beliefs set will trigger plans for the agent to execute. These events are beliefs that are copied from the current beliefs set $B[t]$ to the current events

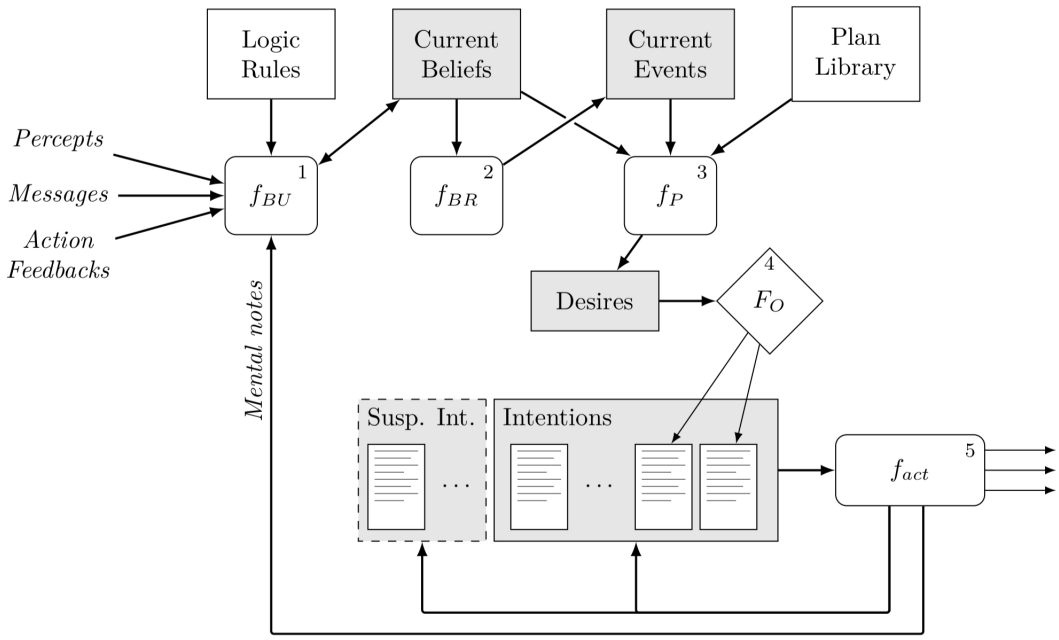


Figure 3.2: The Internal structure of the LISA reasoning cycle. Functions are represented by blocks with rounded corners, external functions represented as diamond shaped blocks, static sets with white square blocks, and dynamic sets with grey blocks. Numbers represent the order of executions for the functions in the reasoning cycle [3, 4].

set $E[t]$. This process is done by a function called the belief review function denoted f_{BR} in figure 3.2, which maps $B[t-1]$ and $B[t]$ to a new current events set $E[t]$.

3. **Retrieving applicable plans**: This is the stage where the agent starts to make decisions by choosing a plan for each event. The plan library Π contains a set of plans, each plan indicated by its contents $\pi_j(\lambda_j)$, where π is the plan, and λ is the contents of the plan, with $\lambda \in [0, n_{\lambda_j}]$, starting with the triggering conditions $\pi(0)$ and a context. The plan is triggered when there are two conditions satisfied: a match between its triggering condition and a current triggering event, and when the $B[t]$ satisfies the context ($B[t] \vDash c_j$). All the plans satisfy the two mentioned conditions are copied into a subset of the Desires set $D[t]$. This operation is performed on all events of $E[t]$, and then it resets to an empty set while the Desires set becomes:

$$D[t] = \{D_1[t], D_2[t], \ldots, D_{n_e}[t]\} \tag{3.1}$$

where each $D_j[t]$ is the set of plans triggered by an event $e_j \in E[t]$ and $n_e = |E[t]|$ is the number of events at time t. This process performed by the function f_P shown in figure 3.2.

4. **Plan selection**: If the agent finds that more than one plan is applicable for an intended event, then it should make a choice on which plan to choose for that particular event. This process is performed by a function called *Option Selection Function* denoted F_O.

$$F_O : \wp(\Pi) \rightarrow \pi \tag{3.2}$$

that maps a set of plans $\wp(\Pi)$ to a single plan π. Since the Desires set contain sets of plans relative to different events, the option selection function must be applied to each of them. The result of this selection process is a set of plans called *Intentions* that are copied into the Intentions Set $I[t]$.

5. **Actions execution**: Once a plan is part of the Intentions set, the agent is committed to executing it as a final step of the current reasoning cycle. At the end of the reasoning cycle, the agent takes the next available action from each plan, and it calls an external function if the action is external, or passes instructions to the f_{BU} to update (add or remove) mental notes from the current beliefs set in the next reasoning cycle. Otherwise once an action is issued, it will be removed from the associated plan. The function that performs these operations is indicated with f_{act} in figure 3.2.

   Because internal actions are executed within a single reasoning cycle, hence the agent only needs to be aware of the external actions, and this could be done through action feedback. If the last executed action from a plan has not yet returned action feedback, the plan is held within a particular subset of the Intentions set called suspended intentions. This operation is managed

automatically by the system, and no intervention from the developer is needed.

A simple example of AV parking scenario that clarify the steps above is presented in example 2.1.

3.2.3 Agent code

In this work, we used sEnglish software [93, 107] with its NLP to design a RA for a self-driving vehicle. For this application, we build semantically rich agent skills and world descriptions. Then we generate executable code for both virtual and physical AV system. This procedure has been designed with a focus on the applicability of these methods in an industrial setting with real-time constraints.

With sEnglish, the plans operate over a description of the world that is captured within the system ontology and maintained by data from sensors in the world model. The system ontology provides a translatable and straightforward description between concepts a programmer would readily understand, such as nouns, and those that an agent can use or manipulate, such as variables or pieces of data.

In sEnglish, an agent's plans are described using English sentences. The meaning of sentences is explained by other sentences until any further decomposition reaches the signal processing level, and no remaining concepts need to be defined.

The agent takes its decisions by relying on information from its environmental model or knowledge base, which is a database regularly updated via sensors and perception mechanisms, and potentially any learned inferences. This database is organised into a high-level ontology and provides information about the system, and in particular the current state of the environment.

The programmer declares plans, although this makes the agent less-creative at run-time, as the plan library is fixed and not dynamically generated by the agent. This has significant advantages in terms of fast execution and viable formal verification [177]. In many safety-critical systems, formal verification of the core agent is crucial. Hence, this kind of BDI agent combines the advantages of deliberative

agents with the advantages of reliability and clearness. The primary purpose here is to produce a unified world model understood by PRISM for the run-time verification process. The agent program consists of three main files:

1. Reasoning file (.sej): used to describe the agent logic.
2. Ontology file (.ont): used to define hierarchies of variable types that the agent needs to coordinate skills.
3. Action files (.sep): to describe the external actions and link them with their associated skills.

The syntax of sENGLISH can be summarised as:

- Natural language sentences are enclosed by Square brackets '`[...]`'. If these square brackets are preceded by a 'hat' symbol '`^[...]`' then this sentence is a *belief* (percept or mental note). A belief sentence can be negated with a tilde '$\sim$' symbol.
- *addition* or *deletion* of a belief is denoted by '+' and '−' symbols which precede the current beliefs set. The same syntax is used for expressing both *events* and internal actions.
- *external actions* can also be represented by sentences enclosed by square brackets '`[...]`' but should not be preceded by any symbol. Each external action should be associated with an action `.sep` file. External actions can only be listed within the list of initial actions or as a part of plans.
- Each action or sentence is terminated with a dot ' . ' except the triggering condition of plans.
- Common keywords are used for logic statements, such as `not`, `and`, `or`, `while` and others.

The structure of the sEnglish file is as follows [93]:

1. `INITIAL BELIEFS AND GOALS`.

   Contain a list of all the beliefs B and initial beliefs B_0, which are copied into $B[t]$ at the beginning of agent operation.

2. `INITIAL ACTIONS`.

   Contain a list of all the actions A and initial actions A_0 which are executed at the beginning of agent operation.

3. `PERCEPTION PROCESS`.

   Are used to configure objects for world modelling and Boolean symbolic sentences in order to represent perception inputs. This section lists all the percepts, except action feedbacks that can be listed in the dedicated action '`.sep`' files.

4. `REASONING`.

   All logic based implication rules L are listed in this section and are applied to $B[t]$ in all reasoning cycles. Implication rules can add or remove beliefs from the current beliefs set and are presented in the form:

   `If <condition> then <action>`

   where `<condition>` can be any logic based rule on beliefs from B and `<action>` is an internal action of addition or deletion of a mental note from $B[t]$.

5. `EXECUTABLE PLANS`.

   Each plan in the plan library Π is listed in the form:

   `If<triggering_event>while<context>then<action>`

   `<action>` could be internal, if it adds or deletes beliefs from the current beliefs set, or external if it calls external functions through their action files. A simple example of the above-listed points is given in figure 3.3 with a more detailed example given in figure 6.1 along with the explanation.

Action definition files `.sep` are used to describe the way the agent calls action executions, e.g. invokes skills. Each file includes:

```
1 //Plan 12
2 If ^[moving vehicle detected] while ^[distance more than 3m
      and less than 6m] and ^[object getting closer] then
3 [Activate slow mode.]
4 [Generate object avoidance waypoints WP.]
5 +^[object PTP generated]
6 [Update drive mode.].
```

Figure 3.3: Sample of a plan definition written in sEnglish. *Line 2* is the triggering condition, while the external and internal actions represented by *lines 3-6*.

- *Procedure name*, same as the file name.
- *sEnglish sentences*, that action is associated with.
- *sEnglish code*: the action can be defined as a sequence of subactions represented by other sEnglish sentences.
- *Input and output classes*: if the skill needs inputs or outputs, they must be defined here.
- *Process, repeat mode*: the subsystem in which the action is implemented and the type of action (`runOnce` or `runRepeated`).
- *Performance feedback*: the list of all possible action feedbacks for the executed actions.

Probabilistic modelling of the environment

The movement prediction of an object for autonomous driving applications could be acquired through two methods: The first is through short term prediction by tracking the target object and applying some path prediction algorithm to provide short term probability of the object's path [178]. This is mostly an accurate method, but it is difficult to merge with our RA system since the agent needs to provide the probabilities internally and include them with probabilistic models for each object for run-time verification. It also provides a short-time horizon that is valid for up to a few seconds.

The second method which we used in this work is a medium-term prediction method, which is a common pattern observation based on a large dataset of observation sequences in a simplified form of the work presented in [179]. This method works better for our system because it could be included in the reasoning cycle of the agent. This method is only valid for particular environments in which objects follow a predictable pattern, for example, pedestrians in a parking lot. It also requires the acquisition of those patterns through datasets or offline manual observations. After producing these patterns for a limited number of objects, they could be included in the agent code at the design stage to be used later during run-time operation. This method is less accurate than the first one because those objects may not always follow the pre-defined pattern. However, it is simpler and provides a better time horizon than target tracking, hence it is used in this work for our simple driving scenario to get the system works. It is also common to set a priori maximum time horizon after which the prediction is not valid.

For this application, the agent needs to have the ability to generate probabilistic models directly within the reasoning cycle; this means that the agent code needs to feature probability distributions that describe the probabilistic nature of the AV system and other objects in the environment.

The probability distribution and nature of any dynamic object mainly depends on the application and the environment. Probability distributions of dynamic environmental objects are obtained from large datasets collected using physical sensors or by simulation, while a manufacturer usually provides the rate of failure of sensors and actuators.

There are two sources of probabilistic behaviour in the LISA system; those are in the form of perception predicates: *sensory percepts* and *action feedbacks*.

Action feedbacks are percepts that are fed into the agent reasoning from action execution functions after executing an action. It is the programmer's responsibility to define all the possible action feedbacks and their probability distributions by including all the information needed to generate the probability distribution within *the action definition file* (`.sep`).

Sensory percepts present behaviour that is less predictable compared to action feedbacks as they are not guaranteed to occur within finite time intervals. Therefore they require probability distributions for modelling activation and deactivation. In order to describe their probabilistic nature over time with a finite number of numerical values, the approach taken here is to define a single probability distribution that is symmetric around an average Gaussian distribution, with a given variance, and evenly space copies of the same distribution over time by a given amount, for example the average value.

Another difference from action feedbacks is that a sensory percept does not necessarily have to be deactivated after a fixed amount number of reasoning cycles. This process must be accounted for with a second probability distribution for deactivation.

The reasoning file contains all the information needed to model the probabilistic nature of a sensory percept, and this information is listed under the '`PERCEPTION PROCESS`' section. This section should contain all the possible sensory percepts with their modelled probability values.

All beliefs in LISA are represented by predicates, which change over time in a probabilistic manner. This means that, with regards to a sensory percept, the time within which it changes is a random variable. For a given environmental model, the probability distribution to describe these random changes can be reasonably assumed to be known. Action feedbacks, which provide information on the success of agent actions can exhibit a different kind of random behaviour. To start with, they can only be activated upon invocation of the action. Also, irrespective of whether they fail or succeed, they do change to an affirmative completion value within some time limit from the first failure message generation. In the LISA framework, time is counted as an integer number of reasoning cycles.

sEnglish to ROS

The method we used to connect the agent designed in sEnglish to the rest of the ROS system is summarised here, while more information about the self-driving vehicle system designed in ROS, is given in chapter 4.

The sEnglish system is natively compatible with ROS, as shown in figure 2.3. The collection of sEnglish sentences that are setup by the programmer can comprise of more complex sentences until atomic actions are then reached. These atomic actions can either be represented as sentences linked to libraries or native C++ code. The programmer can directly interface this C++ code to existing ROS libraries; therefore, the agent can be directly linked into the distributed ROS system.

A recent example of this operation is shown in handling nuclear material for a robot arm [170] where an sEnglish agent is developed and linked to a ROS network, in one case controlling a KUKA IIWA manipulator. In another, the agent is plugged into a different, but compatible drive for a KUKA KR180 manipulator. The only difference is the underlying drivers, providing an identical interface is provided, typically through topics and services available in ROS. The programmer can rapidly configure an sEnglish agent to operate within a distributed network for different applications.

3.3 Formal verification

In our autonomous system, the decision-making subsystem has direct contact with the verification subsystem represented by the MCMAS and Prism model checkers in order to make safe and fast decisions. It is important to say that the verification tools do not interact directly with each other; their only interaction is with the RA. The second point to mention is that they do not operate at the same time, MCMAS is only used during the agent development stage, while Prism used when the vehicle is working. At design-time, MCMAS can check if the logical reasoning system of the agent is consistent and stable; this has another advantage of reducing

the number of iterations required in simulation to validate the system. The agent will depend on this logic rules and actions during run-time operation to choose a suitable action and check the probability of success using PRISM based on the current probability distribution of the moving object's actions in order to move safely in the environment.

3.3.1 Design time verification in MCMAS

As mentioned in the previous section, our autonomous system is a logic-based rational agent that has its own belief set to explain the reasoning behind its behaviour.

In order for an autonomous agent to keep permitted and safe behaviour, it uses logical inference represented by a set of rules. In our case, we investigated how an AV agent can use model checking to establish consistency between its perception-based beliefs, its rules, and its planned actions and their consequences. A set of rules can be modelled as a Labelled Transition System (LTS) using a Boolean Evolution System (BES). Consistency and stability properties can be formulated using Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) [139].

For a safety-critical system, it is vital to have the ability to make fast decisions based on logical consistency, e.g. [180, 181, 182, 183, 184, 185] and to be able to detect when an inconsistency occurs. A logic-based system, in general, has a belief set, which provides the basis of reasoning for a robot's behaviour [180]. An AV is prone to accidents if there is one or more inconsistent beliefs, e.g., the AV can hit an obstacle, instead of avoiding it, if it mistakenly believes that any route of avoidance could cause more damage, due to, for instance, misperception of the environment.

Consistency checking for a safety-critical system has implications for legal certification. Humans tend to formulate legal and social behaviour rules in terms of logical implications, and robots tend to do something similar. Future legal frameworks for certification of AVs will need to take into account a verifiable decision-making process.

MCMAS has been used to check the consistency of beliefs, rules and actions of

our AV in real-time against a set of formulae designed for this purpose. Inconsistent rules may lead to an unsafe action causing unacceptable behaviour or an accident when there is more than one possible action. Unstable rules will prevent the AV from selecting an action in a timely manner. The agent logic code could be long and complicated to be checked manually. An example is when there are multiple actions that could fit for a particular external event, these actions may contradict each other, causing wrong decision to be made while driving. This sort of scenario is so difficult to be checked manually and here where it comes the power of the verification tool.

In our approach to verification, we use CTL, a description of which can be found in section 2.4.4. Separate variables are used for each belief, and each pre-programmed plan is indexed by a variable. Using this method, one can define complex properties which formulate all kinds of aspects of the reasoning process. The MCMAS model checker can generate counterexamples. Counterexamples are traces in model execution that do not satisfy the CTL specification required for the system under consideration. Counterexamples can be used to correct an agent program in an iterative process. For instance, consider the situation where an agent is defined with two actions that are alternative to each other and that should never be co-executed, such as '`go left`' and '`go right`'. Although the intention of an agent programmer may be to avoid these two actions happening simultaneously, it is not guaranteed. A model checker, however, can reveal such a situation and point to corrections needed in the program.

Through the iterative process of design-time verification, a developer can improve the agent code by checking a model of the code against a set of properties in order to improve the decision-making capabilities of the agent reasoning.

Modelling agent perception and reasoning process

The Binary Decision Diagram (BDD) is very useful for real-time robotic agents with complex reasoning processes, to enhance solver efficiency to be capable of dealing

with large search space for both online and offline applications by compressing the search space and generating a succinct and unique representation of a Boolean formula.

BDDs have been widely adopted and applied successfully to the verification of large systems [119]. In this project, we used the BDDs based symbolic model checking approach [186] to describe the search space of the agent's predicate model and discover inconsistency in a set of logic rules and statements on relationships in a current model of the world, behaviour rules, planned actions, and past actions of a robotic agent. In case of discovering any inconsistency, the developer can correct with the guide of counterexample [119, 126] during design-time to improve the reasoning process.

Our knowledge representation of the AV is based on predicates that are abstracted and derived from the sensors in the simulated or real environments.

Definition 3.1 (Boolean Evolution System)**.** A $BES = \langle \mathcal{B}, \mathcal{R} \rangle$, where:

- $\mathcal{B} = \mathcal{B}^{known} \cup \mathcal{B}^{unknown}$ is a set of predicates (Boolean variables) $\mathcal{B} = \{b_1, \cdots, b_n\}$,
- $\mathcal{B}^{known} = \mathcal{B}^{true} \cup \mathcal{B}^{false}$ is a set of known predicates,
- $\mathcal{B}^{unknown}$ is a set of unknown predicates in its initial evaluation that could be determined later as $\mathcal{B}^{known}$ ($\mathcal{B}^{true} \vee \mathcal{B}^{false}$), or continue to be *Unknown*,
- $\mathcal{R}$ is a set of reasoning rules (evolution rules) of the form $X \rightarrow Y$, $\mathcal{R} = \{r_1, \cdots, r_m\}$ defined over $\mathcal{B}$.

□

$\mathcal{B}^{known}$ is a set of Boolean predicates, modelling certain predicates, and $\mathcal{B}^{unknown}$ a set of pseudo-Boolean predicates that model uncertain predicates. A pseudo-Boolean predicate has three values: *unknown*, *true* and *false*, and is initialised as *unknown*. X is a Boolean formula, called a *guard* or *enabling condition*, and defined over $\mathcal{B}$. Y is an assignment of the form $a := true$ (abbr. a) or $a := false$ (abbr. $\neg a$) with $a \in \mathcal{B}$.

At the beginning of a reasoning cycle, all predicates in $\mathcal{B}^{known}$ are initialised as either *true* or *false*, while all predicates in $\mathcal{B}^{unknown}$ initialised as *unknown*. The rules without any *unknown* predicates in their guard are evaluated and the assignments in all enabled rules, i.e., their guard becomes *true*, are executed simultaneously. This results from a new evaluation over $\mathcal{B}$ and all rules are re-evaluated over this new evaluation. The reasoning process continues until the evaluation is not changed anymore, or a timeout is triggered.

When a guard g of a rule is evaluated to *true* on a valuation $\overline{\mathcal{B}}$ of $\mathcal{B}$, we say that the rule is *enabled*. After applying all enabled evolution rules over $\overline{\mathcal{B}}$ simultaneously, we obtain a new valuation $\overline{\mathcal{B}}'$. If two enabled rules set a variable to different values in $\overline{\mathcal{B}}'$, then the reasoning system is *inconsistent*. Starting from valuation $\overline{\mathcal{B}}^0$, we can apply the evolution rules infinitely and obtain valuations $\overline{\mathcal{B}}^1, \ldots, \overline{\mathcal{B}}^i, \ldots$ if the reasoning system is consistent. However, the system is *unstable* if for any pair of adjacent valuations $\overline{\mathcal{B}}^i$ and $\overline{\mathcal{B}}^{i+1}$, we have $\overline{\mathcal{B}}^i \neq \overline{\mathcal{B}}^{i+1}$.

Stability and consistency check

Linear-time Temporal Logic (LTL) and Computation Tree Logic (CTL) [135] are popular logic for verification of transition systems. They are used to specify properties of a system under investigation. LTL deals with one possible future behaviour, while CTL accounts for all possibilities of future behaviours. In our work, we used CTL to study stability and consistency using efficient implementation techniques of CTL model checking.

The MCMAS modelling language which is called Interpreted System Programming Language (ISPL) used to model a Boolean Evolution System (BES) program [36]. Both consistency and stability can be captured by CTL formulae and the model can be checked efficiently by MCMAS. When the BES is not consistent or stable, a counterexample is generated by MCMAS to demonstrate the violation. This counterexample is very useful for the developers to correct the system. Due to its optimised implementation, MCMAS can handle large systems with hundreds

of predicates.

Definition 3.2 (Consistency)**.** Three problems might occur during evolution of a BES:

1. Two enabled rules try to update the same Boolean variable with opposite values at some time.

2. A variable in $\mathcal{B}^{known}$ is updated to the opposite value of its initial value at some time.

3. A variable in $\mathcal{B}^{unknown}$ is updated to the opposite value at some time after its value has been determined ($\mathcal{B}^{unknown}$ are initially set to *unknown*, which can be overwritten using the evolution rules).

If none of the above three points occur, then the system is said to be *consistent*. Otherwise, it is *inconsistent*. □

The first category of Inconsistency in the belief base can be expressed using the following CTL formula [32]:

$$AG(\neg(EX\mathtt{B}_1 \wedge EX\neg\mathtt{B}_1) \wedge \cdots \wedge \neg(EX\mathtt{B}_n \wedge EX\neg\mathtt{B}_n)). \tag{3.3}$$

The boolean evolution system is consistent in case the above formula evaluated to true.

The second category of Inconsistency can be expressed using the following CTL formula [32]:

$$\begin{aligned} AG(\neg(\mathtt{B}_{n_1+1} \wedge EX\neg\mathtt{B}_{n_1+1}) \wedge \neg(\neg\mathtt{B}_{n_1+1} \wedge EX\mathtt{B}_{n_1+1}) \wedge \\ \cdots \wedge \neg(\mathtt{B}_n \wedge EX\neg\mathtt{B}_n) \wedge \neg(\neg\mathtt{B}_n \wedge EX\mathtt{B}_n)). \end{aligned} \tag{3.4}$$

The boolean evolution system is consistent in case the above formula evaluated to true.

The third category of inconsistency can be checked in the same way mentioned above over the *unknown* variables.

Definition 3.3 (Stability)**.** A *BES* is *stable* if from any valuation and applying the rules recursively, it eventually reaches a valuation $\overline{\mathcal{B}}$ where no other valuation can be obtained, i.e., $\overline{\mathcal{B}}' = \overline{\mathcal{B}}$. We say that $\overline{\mathcal{B}}$ is a *stable* valuation, written as $\overline{\mathcal{B}}_s$. □

The instability problem can be checked by the following CTL formula [32]:

$$\begin{aligned} AF((&AG\ \mathtt{B}_1 \vee AG\neg\mathtt{B}_1 \vee AG\ \mathtt{K}_1) \wedge \cdots \wedge \\ &(AG\ \mathtt{B}_{n_1} \vee AG\neg\mathtt{B}_{n_1} \vee AG\ \mathtt{K}_{n_1}) \wedge \\ &(AG\ \mathtt{B}_{n_1+1} \vee AG\neg\mathtt{B}_{n_1+1}) \wedge \cdots \wedge \\ &(AG\ \mathtt{B}_n \vee AG\neg\mathtt{B}_n)). \end{aligned} \tag{3.5}$$

The boolean evolution system is stable in case the above formula evaluated to true.

Consistent rules cannot generate contradictory conditions throughout the whole reasoning process, which means that at no time can a predicate be assigned to *true* and *false* simultaneously. Stable rules make the reasoning process terminate in finite steps. In other words, a *stable* evaluation is reached eventually such that this stable evaluation is obtained by extending the reasoning process one step further. The detailed proofs of those lemmas are illustrated in our previous work [32].

Compilation from LISA to ISPL in MCMAS is reasonably straightforward if sensory events and all possible outcomes of continuous control actions are abstracted into a finite set of predicates. Sensing predicates define three values in ISPL, and operational predicates are represented as Boolean. The evolution system represents the logical inference in MCMAS. Time-step variables aid triggers, context and serial execution of actions in combination with activity and outcome predicates. The outcome predicates are defined to be able to take on all feasible combinations of their assignment in the evolution system representing executable plans in LISA. Three kinds of rules have been set to the agent to follow: physical rules, behaviour

rules, and consequence rules. To reduce the complexity of choosing an action, only physically feasible combinations of predicates of action-outcome are allowed. This means that the chosen action should not have any conflict with the physical set of rules. By using a model checker such as MCMAS, representation of reasoning can reveal logical inconsistencies in the agent program of the AV.

During the development time, the main objective of the developer is to generate a model that represents the agent reasoning operation. From there on, the verification process is straight forward using the MCMAS model checker to explore properties of the generated model in order to make sure that it meets the proposed specifications. If it is not, then the model checker will generate a counterexample identifying design flaws in the agent code that can then be corrected. A complete case study that clarifies the procedure outlined in this section can be found in chapter 6.

Implementation

In this section, we explain the steps involved in building the Boolean evolution system model in the ISPL program from which the corresponding transition system can be generated using MCMAS. We will go through the program generated for the AV in order to check for the stability and consistency of its predicates, as mentioned earlier in this section. This program is based on definition 3.1.

Programs written in ISPL contain definitions for a set of agents, initial states, and the specifications of the system as following:

```
Agent 1 ... end Agent
...
Agent n ... end Agent
InitStates ... end InitStates
Evaluation ... end Evaluation
Formulae ... end Formulae
```

The agents here represent not only the AV but also other agents (objects) defined within agent *environment*. Here we will explain our MCMAS program in more details.

1. The AV and the environment agents definitions are:

```
Agent AV
        Vars: ... end Vars
        Actions = {...};
        Evolution: ... end Evolution
        Protocol: ... end Protocol
end Agent

Agent Environment
    Vars:
        Pedestrian: boolean;
        Vehicle: boolean;
    end Vars
...
end Agent
```

In the definition of the RA, we can define all the predicates in section *Vars*. These include the *Beliefs*, *Uncertain beliefs*, and others. In the *Actions* section, we can define all the actions available to the agent at any time. The *Evolution function* specifies the transition relation between local states, while the *Protocol* can be used to define a set of enabled actions for each local state. The same thing goes for agent environment where it is possible to define similar sets for any given agent in the program. Each variable in *Vars* can be specified either as a *Boolean* variable if it is part of the $\mathcal{B}^{known} = (\mathcal{B}^{true} \vee \mathcal{B}^{false})$, or as an enumerated variable if it is part of the $\mathcal{B}^{unknown} = (\mathcal{B}^{true} \vee \mathcal{B}^{false} \vee Unknown)$.

The *Evolution* rule set of each agent is a list of variables from the *Vars* set translated into a form of "a `if` b", where a is a set of assignments and b is Boolean expression (*guard*) as shown below:

```
Evolution:
        A1=true if B1=true;
        A3=true if B1=false and B2=true and  B3=false;
        A1=true if B3=true;
        ...
end Evolution
```

2. *InitStates*: Having defined all the sets related to each agent, we come to the second step of defining the *InitStates* of the Program. We mentioned the initial state of each agent as shown below:

```
InitStates
       AV.P1=Unknown and AV.P2=Unknown and ...
       ...
end InitStates
```

Here AV refers to the agent name, while P1, P2, A1, A2 refer to the variables in *Vars* of that agent.

3. *Evaluation*: An atomic proposition of the form "x `if` y", where x is the name of the atomic proposition, and y is a Boolean expression that defines the set of states for which x holds. Part of the ISPL code is shown below:

```
Evaluation
        B1_true if AV.B1=true;
        B1_false if AV.B1=false;
        ...
        P6_false if AV.P6=False;
        P6_unknown if AV.P6=Unknown;
```

```
        ...
end Evaluation
```

4. *Formula*: In the last part of the ISPL code, we define the set of queries required to check for consistency and stability. Part of the code written for this purpose is shown below:

```
Formula
AG(!(EX B1_true and EX B1_false) and !(EX B2_true and EX B2_false) and ...
...
AF((AG B1_true or AG B1_false) and (AG B2_true or AG B2_false) and ...
...
AG !(A1_true and (A2_true or A3_true or A4_true or A5_true)) ;
AG !(A2_true and (A1_true or A3_true or A4_true or A5_true)) ;
...
end Formulae
```

The steps above are all that are required to construct the ISPL code to verify the agent code during design-time. It is important to mention again that this code is constructed by the programmer during the development time and need to be run once unless the code is modified. A full case study is explained in chapter 6.

3.3.2 Run time verification in PRISM

PRISM is a modern probabilistic model checker that allows within a very short time to verify relatively large models, providing a feasible technique to the run-time verification approach for improving the capabilities of decision-making of the agent [4].

In this section, we will go through the method used to verify AV actions during run-time operation. This process needs to discover through sensors the environment around for detecting possible threats or obstacles, and to look for free parking spaces in a parking lot. This information is then processed and abstracted before being sent to the agent to enrich the agent reasoning with data for its operational cycles.

The first step of the formal verification is to generate a model that represents the agent reasoning and fully describe its behaviour. In the case of an AV, it is also essential to include the interface with the environment by generating models that represent the behaviour of objects moving nearby. This interface consists of the symbolic data generated by the abstraction subsystem where this will be used to execute a sequence of various actions and mental notes.

A feature of the Limited Instruction Set Agent (LISA) system is that the user can include probability distributions of input variables within the agent code (see Section 3.2.3) during the design stage. The agent will later convert this to probabilistic Prism code that can be verified against a set of queries representing probabilistic specifications, for example, Probabilistic Computation Tree Logic (PCTL) specifications. This operation is described in this section.

When a complete probabilistic model of the system is available, it is possible to provide an estimation on the consequences of actions so that the agent can choose a suitable action based on this knowledge to bring the world to the desired state [3, 4].

We demonstrate in the next subsection that the RA system can be modelled as a PTP by translating the agent program automatically into a code for Prism. The model checking technique presented here is used to predict possible consequences of actions so that the agent can select the best strategy.

In this section, we assume that the response of the physical environment of the agent is also modelled as a PTP (E) in terms of the predicates feedback to the belief base of the agent under various environmental states. E is composed of environmental states, and transitions which under each state through the conditional probabilities of the environment corresponds to triggering of predicates through the sensor system of the robotic agent. Given that the agent has well-defined decision structures as described in this chapter, the environment-agent model will also be a PTP. This section describes how the combination of probability distributions, when combined with the environmental PTP and the logic-based decision-making of the agent, can be modelled in Prism.

First, we introduce the formal model of PTPs, the technique that will be used to describe the probabilistic models of different objects for PRISM model checker in this work. The PTPs combine:

- Stochastic behaviour, through discrete probabilistic choice;
- Non-determinism and concurrency, through parallel composition;
- Timed behaviour, through real-valued clocks in the style of timed automata;
- Control flow and discrete data variables to capture program behaviour.

PTP is a framework for quantitative verification of software that exhibits both probabilistic and real-time behaviour.

Quantitative verification is a formal method for the analysis of probabilistic and timed systems. It constructs a mathematical model that captures the system's behaviour, then the analysis of formally specified quantitative properties. These might include, for example, the probability of an AV failing to stop within 1.5 seconds based on the current AV speed, or the probability of a nearby pedestrian to continue walking towards the AV based on the current scenario. Here the probabilistic models are built based on pre-programmed probabilities collected from available datasets and observation of similar cases only as a proof of concept. Larger datasets and more accurate prediction methods need to be used for better and more accurate results in real-life scenarios. In this work, we obtain some common patterns for the behaviour of the pedestrian and drivers under some circumstances. We use PRISM to verify the success of the AV's actions within a highly probabilistic environment. Notice that this permits an analysis not just of a system's correctness, but also its performance and reliability.

Implementation

The PRISM language and verification algorithms provide a range of tools to verify our PCTL properties. In Section 3.2.2, we described the implementation of

the agent reasoning, including an approach for a feasible modelling solution for a probabilistic model of the agent environment. This has been implemented using sEnglish based on NLP to describe the probability distribution of the activation and deactivation of the action feedbacks and the agent percepts continuously.

The described model of the LISA system includes the behaviour of the agent reasoning along with its interface with the world; hence it is considered to be an internal model that can be used for probabilistic evaluation during the run-time verification process, for the outcome of possible plan choice.

We designed a translator that works as a part of the sEnglish system environment to translate the agent reasoning code to PTP models that can be verified by Prism; it is a direct text processing algorithm in C++ running as a ROS node that can run in few milliseconds (hence its time is neglected). The agent will also translate the properties of the models in PCTL and the query of questions the agent needs to ask. As soon as the equivalent PTP models are verified, then the agent will know about different properties expressed in PCTL. A Boolean variable for each belief is defined, and transition probabilities are taken from the probability distributions defined in the sEnglish code.

Here we have presented the method we used for the verification process to help in selecting the plans with the best possible outcome in terms of safety of the AV, passengers, and other agents in the environment.

Through the tests, we noticed that this time is still within the acceptable range to be included in the reasoning cycles of the agent without producing a significant delay or affecting on the overall agent performance, as we can see from the case study presented in chapter 6. However, it is important to mention that this is mainly because of the limited usage of Prism where this is reflected in the size of the PTP model generated and the number of PCTL queries.

However, despite the many capabilities that the model checker can provide once the model is generated, we did not utilise the model checker for in-depth investigation of multiple queries over the generated PTP models as it is not useful

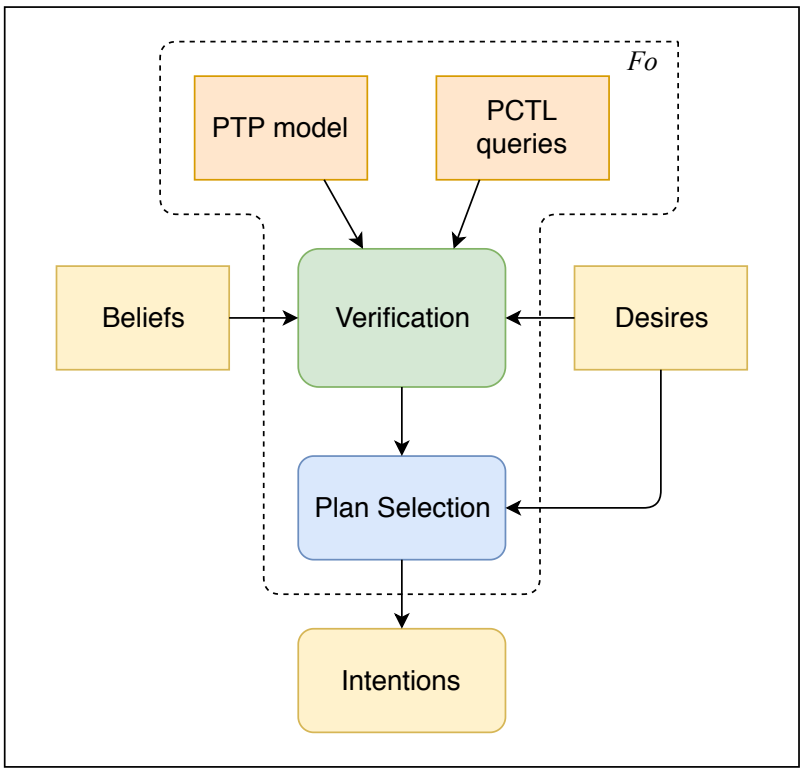


Figure 3.4: Plan selection function for the verification process.

during run-time operation because the time here is quite limited; also it is not necessary for our implementation and the targeted scenario. The reason is that the reasoning cycle for the rational agent takes 100 *ms* to complete, this time put restrictions on the size of the PTP models, and the size of the related queries. Hence we only targeted the collision probabilities between the AV and the other objects moving around within a specific distance.

For example, in case two objects are moving near the AV and detected by the perception system to be a moving pedestrian and a vehicle, then the total number of PTPs generated is three, one for the AV and one for each detected object. A further investigation and queries could take the time of verification exponentially and make the results generated from these verification queries not feasible for the reasoning cycle of the agent. We implemented a function called *plan selection function* that is part of the LISA agent architecture to utilise the PRISM model checking to get the probability of success for the available set of plans then to select the most suitable plan for execution. In [4], the authors used the same technique for a water surface

vehicle.

Figure 3.4 show the structure of the run-time verification process implemented through the Plan Selection function. The function is part of the reasoning cycle presented in figure 3.2, *the plan selection function (Fo)*. However, in the final operation of the agent, this function is still external to the agent program. Ideally, the run-time verification is fast enough to be executed at least once for each reasoning cycle. However, an even faster execution time might be needed when more than one event triggers multiple plans. A possible practical solution is that if the plan selection requires more time than what is provided by one reasoning cycle, the agent then suspends the decision-making process until the results of the run-time verification are available. A possible safety option for the AV during this time is to command a stop action until the verification operation is complete.

Here we will list PTPs models generated while the AV is moving in a parking lot in a simulated environment and there are a vehicle and a pedestrian nearby as shown in figure 6.5. The Agent generated three PTPs for this case to determine a safe plan selection and action execution:

```
module AV

    s1 : [0..6];
    x1 : [0..1] init 0;
    y1 : [0..12] init 0;
    c1 : clock;
```

In the above listed code, we can see the definition of the module name in the first line, then the rational agent define in the PTP code the number of steps for a specific predefined action, also a definition of the X and Y relative coordinates of the AV, and finally the clock that is reset at every step.

```
        [] s1=0 & c1 <= 2 & c1 >= 1 -> (s1'=1)&(c1'=0)&(y1'=3);
        [] s1=1 & c1 <= 2 & c1 >= 1 -> (s1'=2)&(c1'=0)&(y1'=6);
        [] s1=2 & c1 <= 2 & c1 >= 1 -> (s1'=3)&(c1'=0)&(x1'=1)&(y1'=9);
```

```
        [] s1=3 & c1 <= 2 & c1 >= 1 -> (s1'=4)&(c1'=0)&(x1'=1)&(y1'=12);
        [] s1=4 & c1 <= 2 & c1 >= 1 -> (s1'=5)&(c1'=0)&(x1'=1)&(y1'=15);
        [] s1=5 & c1 <= 2 & c1 >= 1 -> (s1'=6)&(c1'=0)&(x1'=0)&(y1'=18);
        [] s1=6 -> (s1'=6);
endmodule
```

After that the agent will define those steps (that form the action) seeking for its success probability as shown above. Each line contains the step number, clock, and the next coordinate that the AV is moving towards.

The same thing is applied to the other two objects moving nearby with one critical difference which is the inclusion of probabilistic distribution for those objects behaviour as shown below for the other vehicle moving towards the AV.

```
module vehicle
    s3 : [0..14] init 0;
    x3 : [-1..0] init -1;
    y3 : [4..16] init 16;
    c3 : clock;

    [] s3=0 & c3 <=2 & c3 >= 1 -> 0.6 : (s3'=1)&(y3'=14)&(c3'=0) + 0.1 :
    (s3'=2)&(y3'=16)&(c3'=0) + 0.3 : (s3'=3)&(y3'=12)&(c3'=0);
    [] s3=1 & c3 <=2 & c3 >= 1 -> 0.7 : (s3'=4)&(y3'=12)&(c3'=0) + 0.1 :
    (s3'=5)&(y3'=14)&(c3'=0) + 0.2 : (s3'=6)&(y3'=10)&(c3'=0);
    [] s3=2 & c3 <=2 & c3 >= 1 -> 0.6 : (s3'=7)&(y3'=10)&(c3'=0) + 0.1 :
    (s3'=10)&(y3'=12)&(c3'=0) + 0.3 : (s3'=8)&(y3'=8)&(c3'=0);
    [] s3=3 -> (s3'=3);
    [] s3=4 & c3 <=2 & c3 >= 1 -> 0.6 : (s3'=9)&(y3'=10)&(c3'=0) + 0.2 :
    (s3'=10)&(y3'=12)&(c3'=0) + 0.2 : (s3'=11)&(y3'=8)&(c3'=0);
    [] s3=5 -> (s3'=5);
    [] s3=6 & c3 <=2 & c3 >= 1 -> 0.6 : (s3'=11)&(y3'=8)&(c3'=0) + 0.3 :
    (s3'=9)&(y3'=10)&(c3'=0) + 0.1 : (s3'=12)&(y3'=6)&(c3'=0);
...
endmodule
```

Here we can see that every step is a probabilistic step means that the agent is

assuming that the object will behave in this way based on pre-programmed possible behaviour of those objects. The agent use the same method for generating the PTP model for the nearby pedestrian.

3.4 Conclusion

A BDI-based LISA agent implementation for a self-driving vehicle system is presented, with a new verification approach to allow for automatic modelling and verification of AV's decision-making using model checking tools. The reason that this design developed as part of an autonomous system for a self-driving vehicle is to provide an applicable system that can be analysed at both stages of developing and operation to make sure it is safe enough to be used for such safety-critical system.

The autonomous system consists of multiple layered architecture with the agent reasoning on top (represented by a high-level of abstraction), and lower-levels of skills (with a lower-level of abstraction). Agent reasoning is responsible for decision-making by generating run-time decisions to control the AV's system and guiding the AV until destination. The AV system contains different groups of skills that either receive information for the environment then process it and send the essential data to the agent, or to receive decisions from the agent, process these data and send commands to the actuators.

The agent program is based on the sEnglish language with a few improvements on the framework to allow the programmer to include probability distributions and to describe the outcome of perception beliefs and action feedback. Including all the necessary information enables the system to generate a complete and verifiable model of the agent reasoning.

In order to perform verification of the agent code, a model of the system is needed. For the design-time verification, we generated a CTL model that could be verified using MCMAS model checker. Once a model of the system is constructed from the agent code, then it is possible to run different queries to check the stability

and consistency of the agent logic. In case the model checker discovers something wrong, then it will generate a counterexample that could be analysed and used to correct the logic. This, in turn, allows improving the agent program by iteratively correcting design flaws.

A PTP model is chosen for the run-time verification. PTP models are essential to represent the information about the AV and its environment that is needed by the agent reasoning to decide in a probabilistic manner by using the model checker. During run-time verification, this process can be used to make probabilistic estimates of the future outcome of the AV actions. This has been established by connecting the Plan Selection function to a run-time verification process. PRISM initialises the probabilistic model according to the information included and generates probabilities for the plan success rates that are used later to select a plan that minimises failure rates and optimises the performance.

Chapter 4

Modelling and Simulation of the AV

4.1 Introduction

In this chapter, we have presented our Autonomous Vehicle (AV) and the parking lot environment in simulation. This chapter is divided into two main sections. First, we went through the basics of AV design for a parking lot scenario in a simple MATLAB/Simulink model that clarifies the background information needed such as the dynamics of the vehicle and the properties of the parking operation for the interested reader to understand. Then in section 4.3 we presented the main contribution represented by the AV system and the parking lot environment designed in ROS and Gazebo simulator which was also useful for system validation. The novelty of the work in this section represented by the fact that this is the first time such a comprehensive and complex AV system is designed in open-source software represented by ROS along with its proposed environment. We went through the stages of design and simulation represented by the perception system, the decision-making system (only mentioned briefly in this chapter because it has been thoroughly investigated and clarified in chapter 3), the planning system, and the control system. Despite that most of the techniques used in this work are off-the-shelf, however we

have presented an important contribution from both the academic and industrial points of view. There is a massive demand for such realistic AV system that could be easily connected to a physical platform, as discussed in chapter 5 for other research purposes. It is also important to mention that we have slightly modified the off-the-shelf algorithms used in some cases, in other cases it is heavily modified or new features added to make it compatible in a useful way for the AV system design. However, these modifications are mainly done on the code so it might not be clear for the reader. We will try to point out the main features when possible.

This chapter is dedicated to the second research question of 'How we can design a simple, feasible, realistic and reconfigurable autonomous vehicle system using the Robot Operating System?'.

4.2 Modelling and simulation in MATLAB/Simulink

The work presented in this section is only an introduction for the work presented in section 4.3. In this section, we only tried to explain the different aspects of designing an AV system. Section 4.2 is not a contribution, and it does not contain proper results. However, it has been presented in this chapter so that the interested reader can go through it to understand the basics of AV design for a parking lot scenario before going through the details mentioned in section 4.3. First of all, we have presented below some of the critical definitions that will help to understand the system design.

4.2.1 Coordinate systems in AV

In both the simulation and the real-world implementation, there are four types of coordinate systems [187]:

- World: A universal fixed coordinate system where all the objects in the environment, including the AV and its sensors, are placed. This coordinate system

is essential in path planning, motion planning, localisation and mapping, as shown in figure 4.1 below.

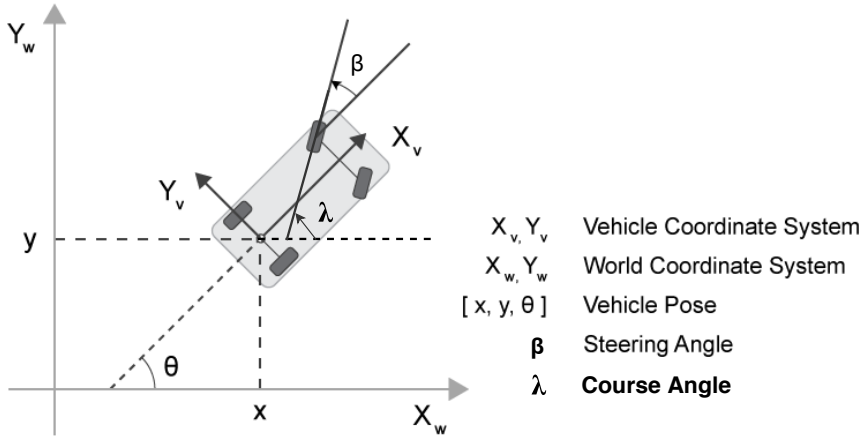


Figure 4.1: World coordinate system.

- Vehicle: This is the vehicle self coordinate system which is in the centre of the rotating axis (below the midpoint of the rear axle). The vehicle coordinate system is represented by (X_V, Y_V, Z_V) in figure 4.2.

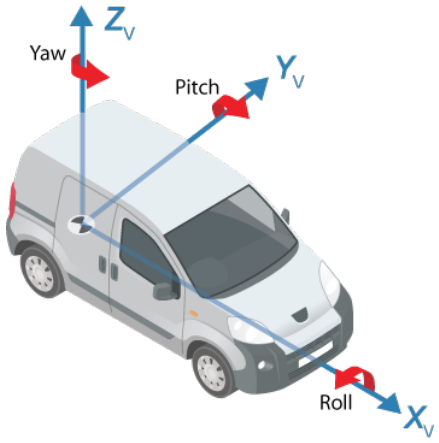


Figure 4.2: Vehicle coordinate system.

Values provided by any sensor are transformed into the AV coordinate system to merge it into a unified reference. For planning and Simultaneous Localisation And Mapping (SLAM), the state of the AV can be described using its position.

- Sensor: This is specific to every perception sensor of the AV. The location of the sensor contains the origin of its coordinates system. A camera is an example of a sensor that often used in an automated driving system. In a camera coordinate system, points are described with the origin located at the optical centre of the camera. A 3D body can be rotated about three orthogonal axes, as shown in figure 4.3. Borrowing aviation terminology, these rotations will be referred to as roll, pitch, and yaw [188]. A roll is a counterclockwise rotation about the x-axis. A pitch is a counterclockwise rotation about the y-axis. A yaw is a counterclockwise rotation about the z-axis.

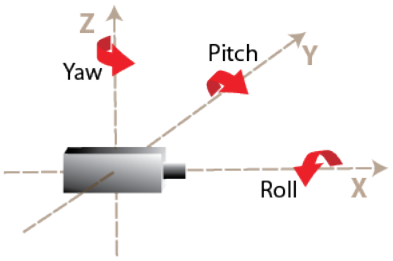


Figure 4.3: Camera coordinate system.

- Spatial: This coordinate system is specific to a camera sensor, particularly, to an image captured by a camera, as shown in figure 4.4. The locations on this system are represented by a 2D map of pixels of the image identified by an integer row and column pair. This is important to specify different objects locations in the image.

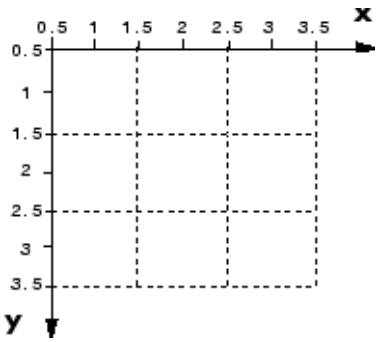


Figure 4.4: Spatial coordinate system.

4.2.2 Environment model

In our scenario, this is represented by the parking lot model, which includes the static (both free and occupied parking spaces), and dynamic objects represented by pedestrians and moving vehicles. The SLAM subsystem of the AV usually builds the occupancy map for the environment. In this example designed in Simulink, we just wanted to present the idea of parking operation; hence the occupancy grid has already been given to the vehicle, this could happen in a real scenario as well through for instance: a camera monitoring the entire parking space or using a vehicle-to-infrastructure (V2I) system. The example of a parking lot used here consists of three occupancy grid layers: Parked vehicles, Stationary obstacles, and Road markings.

Those three occupancy grid map layers have different kinds of obstacles that demonstrate different levels of risk for the AV navigating through it. With this representation of the structure, each occupancy grid can be processed and updated separately. Light cells represent free cells, and dark cells represent occupied cells. The summation of those three layers is shown in figure 4.6.

There is a value given to each cell in the grid, and those values are between 0 and 1, an occupancy threshold property would determine whether the cell is free or not. A cell is considered occupied if its cost is higher than the occupied threshold property and free otherwise.

We define the physical properties of the AV, such as its dimensions, and maximum steering angle, where those are needed while planning the trajectory and control of the AV. We also defined the starting point of the AV, which is shown in figure 4.6 from the left gate of the parking lot. The pose of the AV is specified in world coordinates as $[x, y, \theta]$. The centre of the AV's rear axle is represented by (x, y) in the world coordinate system while the orientation of the AV is represented by (θ) with respect to the world X-axis.

4.2.3 Simulation of the AV and its environment

Designing a control system that is capable of autonomously parking a vehicle is not an easy challenge. This system should control and steer the AV to explore the parking lot and guide the AV to an available parking space. Such a control system makes use of onboard sensors. For instance:

- LiDAR sensors for detecting obstacles, calculating accurate distance measurements and for localisation and mapping.
- Cameras used for detecting road signs, lane markings, pedestrians, other vehicles, and free parking spaces.
- A stereo camera used in addition to the above for distance measurements.

While the AV plans its path through the parking lot by perceiving the environment using its sensors, it should deal with dynamic changes in the environment, such as pedestrians passing nearby, and readjust its plan.

Our MATLAB/Simulink model implements a subset of features required for the AV; it mainly targets the planning and checking of the dynamic system of the vehicle. It also focuses on a feasible path planning. We excluded dynamic obstacle avoidance and map generation from this example. However, those essential parts of the AV system have been implemented with the ROS described later in this chapter.

Decision making layer

The system of the AV involves organising all relevant information into hierarchical layers. The higher layers are responsible for a more abstracted task. The decision-making layer (the rational agent described in chapter 3) is situated at the top of this stack. It is responsible for managing and activating/deactivating the different parts of the system, also for supplying sequence tasks of navigation. This layer collects information from all relevant subsystems, for the primary purpose of guiding the AV safely and efficiently.

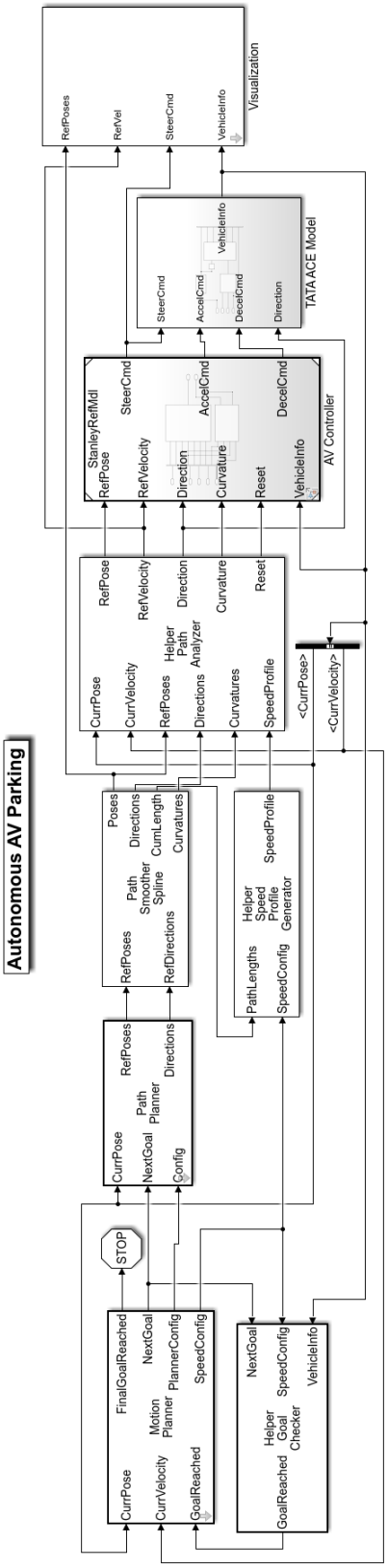


Figure 4.5: Autonomous vehicle system in MATLAB designed for parking lot scenario.

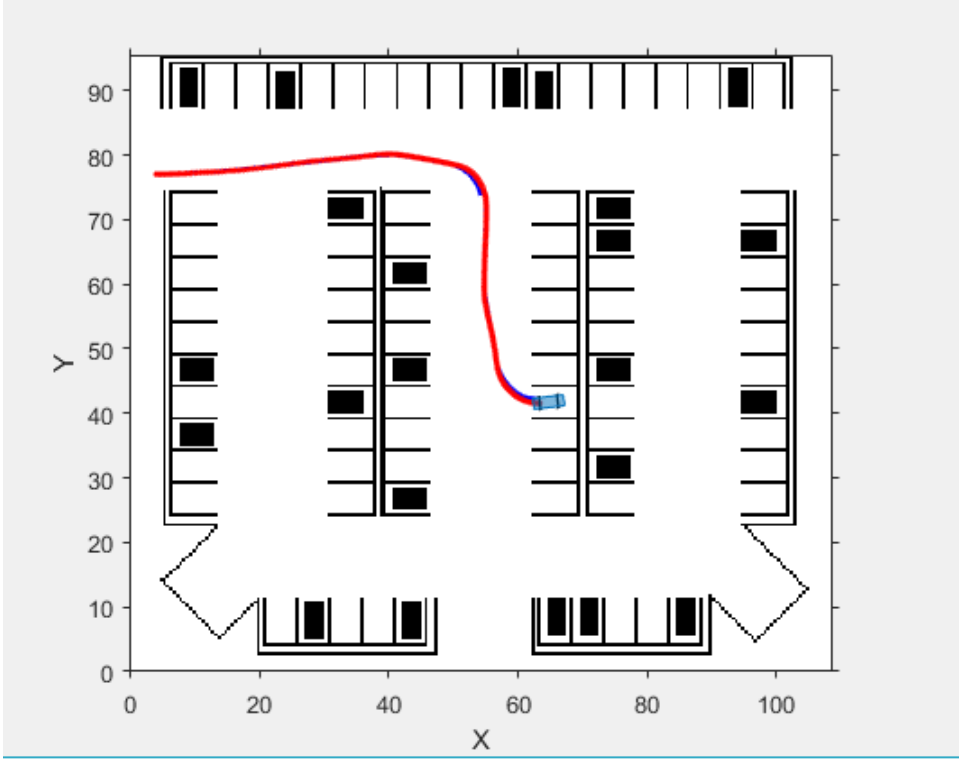


Figure 4.6: Simulation of a parking lot designed in MATLAB.

The AV should find its global path which could be obtained either from a vehicle-to-everything (V2X) (which includes both vehicle-to-vehicle communication (V2V) or vehicle-to-infrastructure communication (V2I)) or from a mapping service — dividing the global path into shorter parts allowing the dynamic trajectory for each point to be planned differently for that link. For example, the final parking manoeuvre needs a different speed profile compared with the previous path segments. In general, this becomes vital during the navigation in roads that involve different numbers of lanes, speed limits, and road signs. In this simple example, we represent the map as an occupancy grid, with locations of the available parking spaces and road links provided manually. Use a static global path plan which is stored as a table in MATLAB; this table determines the speed limit as well as the starting and ending positions. The global path plan can be described as a sequence of lane segments that leads to a parking space. In autonomous parking, the routing algorithm is provided either by a mapping service or the local parking infrastructure or obtained while exploring the parking lot.

Motion planning

After the AV has generated its global path (in this case the path is given manually as points saved in a table), the path must be divided into shorter trajectories or intermediate waypoints and keep doing this with every global path that the AV generates until reaching the final destination. Those shorter trajectories should be both dynamically feasible and collision-free. A feasible trajectory is one that can be followed by the vehicle based on its motion and dynamic constraints. A parking environment involves low accelerations and low velocities where this permit to safely neglect the dynamic constraints rising from inertial effects. We created a Rapidly-exploring Random Tree (RRT) object for our path planner [189]. This method can find a kinematically feasible trajectory by constructing a tree of collision-free and connected vehicle poses. The red line in figure 4.6 represents areas of the generated costmap where the centre of the vehicle (centre of the rear axle) should follow to avoid hitting any obstacles.

Vehicle control and Simulation

The smoothed path, along with the reference speed, produces a feasible trajectory that the AV can follow using a feedback controller.

A feedback controller is also used to correct errors in the trajectory that could come from, for example, inaccuracies in localisation or tire slippage. In particular, the controller consists of two components:

- Lateral control: Used to control the steering angle to direct the vehicle to follow the reference path.
- Longitudinal control: To maintain the reference speed of the reference path by controlling the brake and the throttle.

The feedback controller requires a simulator that can execute the desired controller commands using a suitable vehicle model. Here we simulate the AV using

the following kinematic bicycle model [190]:

$$\dot{x}_r = v_r \times \cos(\theta)$$

$$\dot{y}_r = v_r \times \sin(\theta)$$

$$\dot{\theta} = \frac{v_r}{l} \times \tan(\delta)$$

$$\dot{v}_r = a_r$$

Here (x_r, y_r, θ) represents the vehicle pose in world coordinates. v_r, a_r, l and δ represent the rear-wheel speed, rear-wheel acceleration, wheelbase, and steering angle, respectively. The position and speed of the front wheel can be obtained by [190]:

$$x_f = x_r + l\cos(\theta)$$

$$y_f = y_r + l\sin(\theta)$$

$$v_f = \frac{v_r}{\cos(\delta)}$$

In this Simulink model, we used Stanley block [191, 192] as showing figure 4.5 provided in Simulink as a vehicle controller subsystem to control the longitudinal and lateral values, means to regulate the pose and the velocity of the vehicle, respectively. The longitudinal controller Stanley block computes the acceleration and deceleration commands, in m/s, that controls the velocity of the vehicle. Specify the reference velocity, current velocity, and current driving direction. The controller computes these commands using the Stanley method [191], which the block implements as a discrete Proportional-Integral (PI) controller with integral anti-windup to control the throttle and the brake of the vehicle through the deceleration/acceleration values. The Stanley lateral controller uses a nonlinear control law to minimise the cross-track error and the heading angle of the front wheel relative to the reference path. The lateral controller Stanley block computes the steering angle command that adjusts a vehicle's current pose to match a reference pose. We set the Stanley block lateral controller to a dynamic bicycle model to provide realistic vehicle dynamics. To compute the steering command, we also set the current steering angle, the current yaw rate of the vehicle and the path curvature.

4.3 Modelling and simulation in ROS and Gazebo

A standard AV has a control architecture incorporating both low-level and high-level components. The low-level components include sensors and actuators for object recognition, navigation, environment modelling and mapping, route planning and other skills. High-level system components are responsible for decision-making based on data provided by low-level components.

The perception system provides a stream of images and 3D point cloud data obtained from sensors commonly used in AVs [193]. The AV consists of 8 mono-cameras (three on each side and two at the back), a stereo camera on the front and a LiDAR on top that can be shifted and tilted by the Rational Agent (RA) for better coverage. The stereo camera in front of the vehicle uses a deep-learning object detector that can detect different objects, including those that could exist in real life parking lot environment. The perception system can also locate free parking spaces depending on fiducial markers (landmarks or reference markers - details presented in section 5.3.1). These data are converted to abstracted sentences to be fed to the RA onboard the vehicle. In chapter 6, we carried out a case study for the parking lot scenario shown in figure 4.10 to demonstrate the method of verification and to show the feasibility of our approach.

Our novel AV system shown in figure 4.7 is based on a modular design that makes practical implementation relatively simple and allows for future updates. We used LISA agent implementation due to its capability to execute actions based on decision-making to pursue goals while also not being too complicated to enable verification. The decision process also uses rules and abstractions from predictions of the future (consequences of future events) and can re-plan the path of the AV when needed.

The RA is capable of sensing the environment and move the vehicle in a collision-free path without the need for human support. To achieve this, the perception system builds a model of the environment, locates surrounding objects and

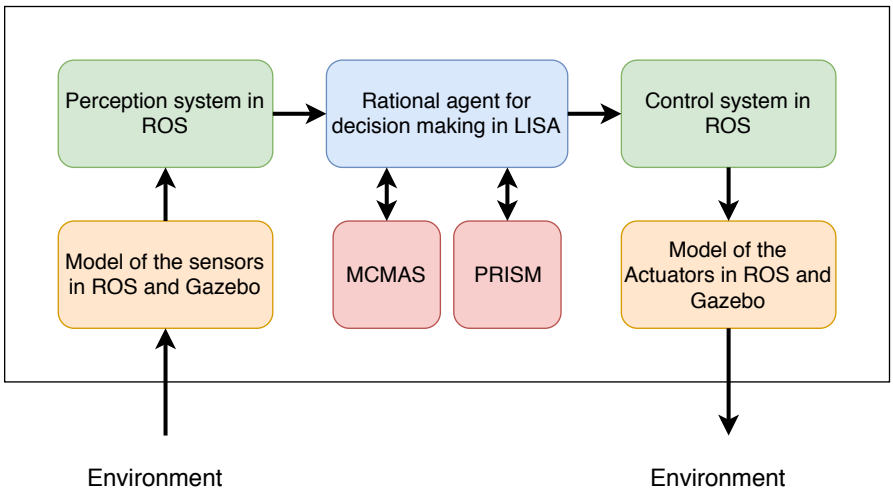


Figure 4.7: General autonomous vehicle system overview.

updates its model after each perception cycle. The software agent has rule-based reasoning, planning capability and some feedback control skills for steering and velocity regulation. Our RA has been implemented using Natural Language Programming (NLP) of agents and robots in sEnglish [93] that compiles into the LISA agent.

The vehicle in simulation supports a reconfigurable, scalable, and modular design to ease the implementation of different system parts and further development. The physics-engine based simulation consists of a model of our Tata ace electric vehicle, as shown in figure 4.8 with the same specifications and parameters as the real vehicle we were developing. All the sensors used in the simulation have the same properties as the real sensors used with the Tata electric vehicle shown in figure 5.1.

The AV is based on packages built in C++ and Python, and compatible with ROS. ROS provides libraries and tools for writing control and perception algorithms and other applications for AVs. With various levels of software and hardware abstraction, device drivers for a seamless interface of sensors, libraries for simulating sensors and visualisers for diagnostics purposes, ROS provides middleware and repository by which distributed simulation can take place, and the software installation is straightforward. Being a distributed computing environment, it implicitly handles all the communication protocols.

However, standard ROS packages lack domain-specific requirements for experimentation with a car-like robot. A typical setup of an AV consists of a vehicle controller and sensors strategically mounted on different points of the vehicle. In order to control motion seamlessly, we have created interfaces for control and consistent consumption of sensor data. We have tested the current state of the vehicle and issued control signals well before the real platform is engaged. Once all the algorithms have been validated in simulation, they were implemented on the real vehicle, and the physical platform has then replaced its equivalent simulated version.

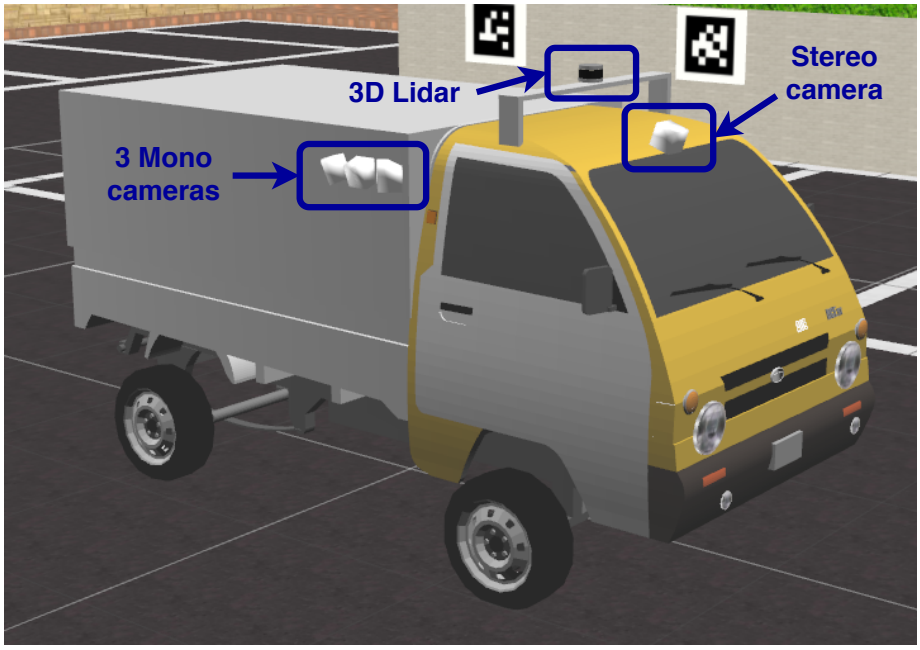


Figure 4.8: The test vehicle we designed in ROS and Gazebo showing sensor configuration.

In the following subsection, we will examine the AV system decomposed into four components, as shown in figure 4.9: First we have the perception system that used to get data about the surroundings and feed it to the second stage represented by the RA who decides on a local driving task the progress of the car towards the destination by rules of interaction and rules of the road. The next stages are the Global path planner and the Local path planner, which are responsible for generating the abstract path of the AV from the starting point to its destination, then select a continuous motion plan through the environment to achieve a local navigational task. The last component is the control system that executes the

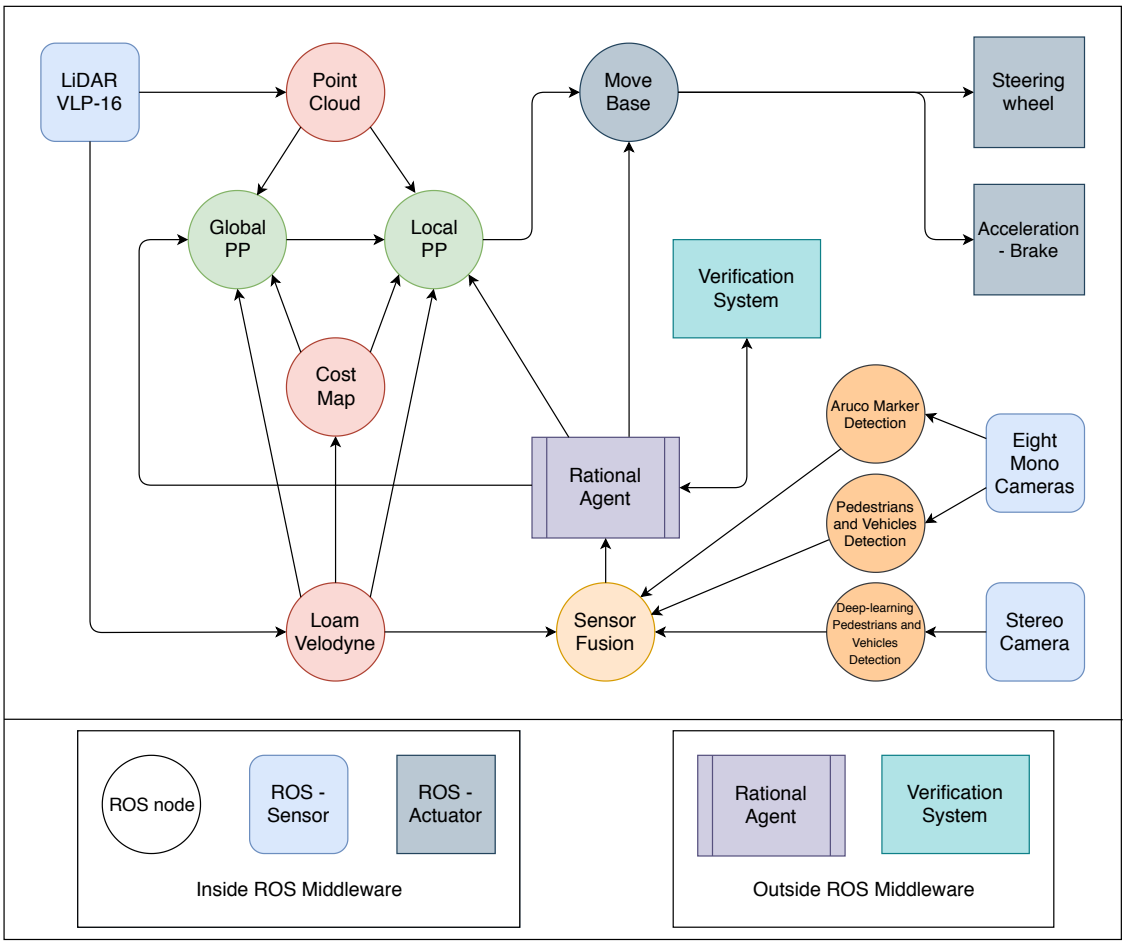


Figure 4.9: General AV system showing the main nodes designed in ROS (secondary and supporting nodes are not shown here), RA designed in sEnglish and verification system designed in MCMAS and PRISM verification tools.

motion using actuators and reactively corrects errors in the execution of the planned motion. In the remainder of the section, we discuss each of these components briefly.

4.3.1 Perception system

The perception system is responsible for providing a model of the world to the decision-making and planning subsystems. The model includes the moving objects (represented as a list of tracked objects), static obstacles (represented in a regular grid), and localising the vehicle relative to, and estimating the shape of, the roads it is driving on.

Nowadays, vision-based detection techniques work by extracting image features

to segment Regions Of Interest (ROI) then detect different objects within those regions. In particular, detection of people and vehicles has made significant progress in the autonomous and assisted driving areas [194], [195].

Radar is a good information source for perceptual tasks; however, the spatial resolution of radar is typically poor compared to camera and LiDAR. Thus, much recent perception research revolves around cameras and LiDARs. Detection methods based on mono cameras suffer in two ways: despite methods proposed for moving mono-cameras, fast and accurate range measurement remains an issue, which is vital for critical object detection in autonomous driving applications. Besides, optical sensors can suffer from a limited field of view and poor operation during low lighting conditions. On the other hand, LiDAR has usually been used with Advanced Driver-Assistance Systems (ADAS) applications. It has become part of the AV perception system because of the high precision range measurements and the wide fields of view that it provides. The main issue for the LiDAR-based system is that LiDAR scans can poorly distinguish between different objects, especially in a dense environment. Stereo cameras can provide more precise depth data and a wider angle compared with mono cameras; however, the detection angle still less than for 3D LiDAR and with less accuracy and speed of depth data, especially for long distances which are often vital for AVs. The integration of cameras and LiDAR sensors can therefore enhance the fast object detection and recognition performance [196]. This type of sensor fusion system is known as the classic LiDAR-camera fusion system.

In this simulation-based system, we used the same methods and techniques of our approach for the real AV system shown in chapter 5, where we used a Velodyne VLP-16 LiDAR, one ZED stereo camera, and eight raspberry pi mono cameras. The LiDAR is connected directly to ROS for point cloud data processing, the front-facing stereo camera is connected to a Jetson TX2 running YOLOv3 deep-learning based objects detection [197]. The mono cameras are using the processing power of their host raspberry pi system for Aggregated Channel Features (ACFs) object detector of pedestrians and vehicle [6]. Those mono cameras along with the stereo camera covering a 360° FOV. The camera system has also been equipped with a

Table 4.1: Properties of sensors for the AV in simulation.

Sensor type	**No. of sensors**	**Resolution**	**No. of frames /Speed of rotation**
LiDAR	1	3D 16-layer (up to 50m) 360°H/ 30°V 1864 PPS	300 RPM
Stereo camera	1	Colour 1344x376	10 FPS
Mono camera	8	Colour 640x480	6 FPS

method for fiducial follow that use aruco markers to detect location and orientation of free parking slots, as shown in figure 4.13 (right camera 1 and 2). along with the occupancy grid data generated by the LiDAR, the AV is capable of detecting free parking spaces simply and efficiently.

When a known object is detected by one of the cameras, the associated LiDAR measurements are processed for the distance calculation by matching the location of the detected object with the 3D point cloud data belonging to the same object. Based on the generated depth map, the positions of the objects are calculated from the ROI, those measurements from LiDAR are calculated according to the coordinates transformation.

We used LiDAR Odometry And Mapping (LOAM) [7] ROS package for Velodyne VLP-16 3D LiDAR. This package provides a real-time method for mapping and state estimation, where it contains two major threads running in parallel. An Odometry thread measures the motion of the LiDAR between two movements, at a higher frame rate. It also eliminates distortion in the point cloud caused by the motion of the LiDAR. Then there is a Mapping thread that takes the undistorted point cloud and incrementally builds a map, while simultaneously computes the pose of the LiDAR on the map at a lower frame rate. The LiDAR state estimation is a combination of the outputs from those threads.

Figure 4.11 shows the map built in RViz (ROS Visualisation tool) for the AV current path in the parking lot shown in figure 4.12, the sides of the objects that are facing the LiDAR are shown on the map with white lines. We added another layer of protection (inflation layer) using a costmap function which helps the AV

Figure 4.10: Parking lot designed in Gazebo simulator.

to keep an extra safe distance from any object within their inflation distance. The costmap is guiding the AV through the motion planning process by showing the cost of each cell that the AV could go through. The inflation layer is an optimisation that adds new values around obstacles (i.e. inflates the obstacles) in order to make the costmap represent the configuration space of the AV. Inflation is the process of propagating cost values out from occupied cells that decrease with distance. This could be set according to the environment type; it is represented on the map with blue lines surrounding the white lines. Cost means that a cell might be occupied and will cause a collision if the AV pass trough it while moving, hence the AV will try to move away from any object by the amount of the inflation layer distance. This distance could be dynamic, which means that the inflation layer for a dynamic object is different from static ones. Finally, the data for the detected objects and their locations are sent to the RA for further processing.

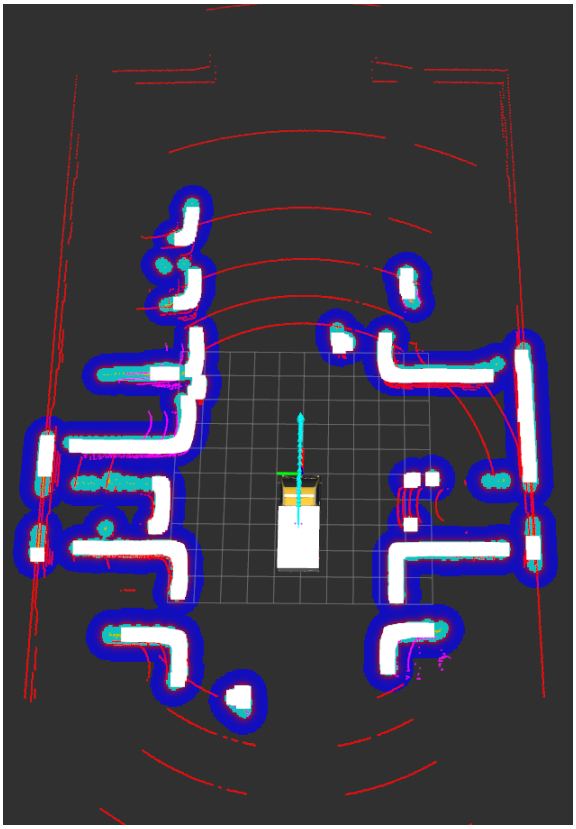


Figure 4.11: This map has been generated using the LiDAR 3D point cloud with the LiDAR odometry and mapping data for parking scenario, this is based on LOAM velodyne ROS package. The parked vehicles has been detected and the system add an inflation layer for protection.

Figure 4.12: Parking lot scenario developed in ROS and Gazebo simulator to check the proposed system.

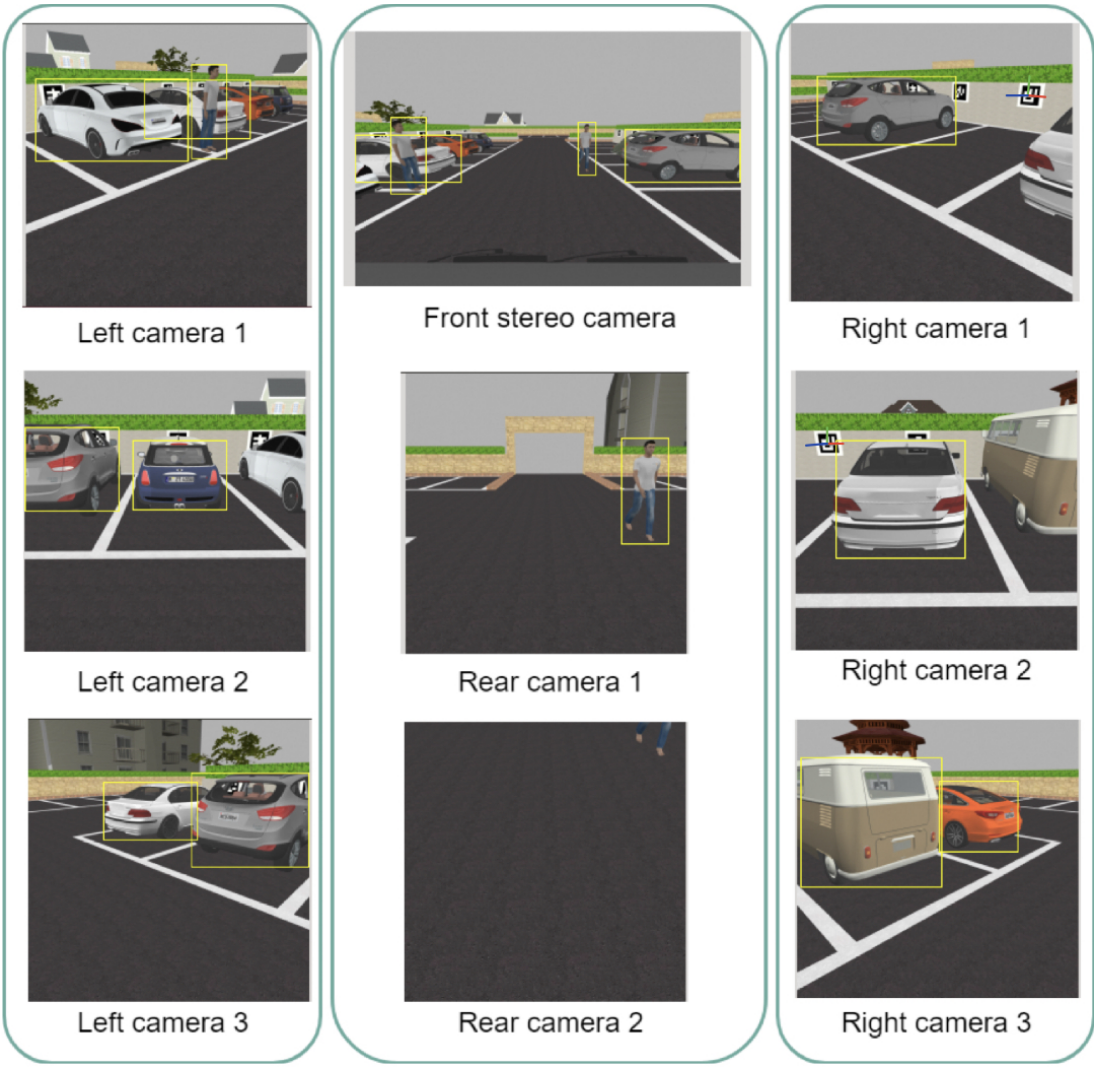


Figure 4.13: Pedestrians and cars detected by the AV using camera sensors. Right camera 1 and 2 show the aruco marker detection attached tot he parking spaces when this info combined with the occupancy grid generated by the LiDAR, it will be easy to detect the free parking spaces.

4.3.2 Autonomous behaviour (Decision making system)

Recent approaches for AVs have used prediction methods in order to avoid the collision by estimate the possible trajectory of the surrounding dynamic objects for the next few seconds.

However, to ensure safety in real-world driving scenarios, the AV needs to handle complex clutter and modelling interactions with other road users. Most solutions proposed for DARPA urban challenge were explicitly tailored to the competing demands and those approaches, e.g. [17] [198] use a rule-based finite state machine for AV to choose between predefined behaviours. This approach needs a safety assessment in order to deal with uncertainties. AV with human-like driving behaviour requires cooperation and interactive-based decision-making; hence, the intention of the other vehicles or pedestrians need to be modelled and integrated into a unified planning framework. While AVs need the ability to reason the intentions of other participants, those also need to infer the AV's intention reasonably [21].

By simulating the proposed traffic scenario, we can search for a possible best policy by scored against the AV's cost function and then the best policy is executed from the set of available policies for the AV. Possible trajectories can also be sampled, and the reaction to the environment can be determined according to the RA model. The AV must be able to interact with other traffic participants according to the driving rules and conventions of the road. Given a sequence of road segments specifying the route should be followed, the high-level behavioural layer (RA) is responsible for selecting an appropriate driving behaviour at any point of time based on the perceived behaviour of other traffic participants and road conditions. For example, when the AV is navigating in a parking lot looking for a free parking space, the RA will command the AV to observe the behaviour of other vehicles and pedestrians during its movement and let the AV proceed only when it is safe to go. Because the driving context and the available behaviours in each context can be modelled as finite sets, hence a natural way to automate the decision-making

process is to represent the behaviours as a Finite State Machine (FSM) with transitions controlled by the perception system (reflect the situation of the environment) and the model of the reasoning system. FSM, coupled with specific subsystems to considered driving scenarios has been adopted by most teams in DARPA urban challenge as a mechanism for behaviour control.

In the literature, similar approaches have been made by reducing decisions to a limited set of options and conducting evaluations with an individual set of policy assignments for each option. The probabilistic representation of the system models can be divided into four main types: Discrete-Time Markov Chain (DTMC), Continuous Time Markov Chain (CTMC), Markov Decision Processes (MDP), and Probabilistic Timed Automata (PTA). A typical implementation of a highway simulation showed the potential of the probabilistic approach represented by MDP to learning different driving styles [199]. In [200] authors showed an enhanced version of the algorithm and its performance by generating human-like trajectories in parking lots, with only a few demonstrations required during learning. A Partially Observable MDP (POMDP) which is an extension of MDP has been used in [201] to integrate the road context and the motion intention of another vehicle in an urban road scenario.

Our RA and its physical environment have been modelled as a PTP model in terms of the predicates fed back to the belief base of the agent under variant states of the environment while the PCTL is used for specification logic [152]. A PTP incorporate probability, dense real-time and date. Its semantics are defined as infinite-state MDPs consists of the states of the environment and the transition between those states, which, through the conditional probabilities of the environment, correspond to triggering of predicates through the sensor system of the AV. The rules are used to set the relationship between the perception predicates (beliefs) and the available actions, when this combination get verified by MCMAS then there will be no space for an unfeasible action in the agent's actions list, this will be reflected later on the run-time verification by reducing the state-space available to check by Prism.

4.3.3 Planning system

Planning modules are concerned with the vehicle's path, motion, and behaviour in the perceived environment. Typically they comprise collision-free trajectory generation, velocity and acceleration management and reactive control for collision risk mitigation. To provide the flexibility of updating and to handle large maps, environment maps are organised into sub-maps. These are integrated and corrected for changes by a graph optimisation approach with critical landmarks as nodes [202].

The Rational agent decides the path in the form of abstracted waypoints, those abstracted waypoints commands written in NLP is not possible to apply them directly to the control system, it first sent to the global path planning to generates proper route segments, in case of looking for a free parking space, then it will be multiple route segments to explore the parking space until reaching the destination. Route constitutes a sequence of road links and in path segments waypoints. Path planning includes functions for trajectory waypoint selection and trajectory evaluation on screened paths. Finally, trajectory control consists of longitudinal and lateral vehicle motion controllers for motion realisation.

In general, the Planning (Navigation) module consists of a global planner (Path planning) to find the optimal path with prior knowledge of the environment and local planner (Motion planning) to calculate the path that is dynamically feasible for the control system with collision avoidance [203]. The AV is navigation problem is reduced to a 2D space of a single plane (X, Y) where the (Z) dimension is neglected. The generated plan is a collection of waypoints for global, $P_i = (X_i, Y_i)^T$, and local $P_i = (x_i, y_i, \theta_i)^T$, $\in \mathcal{R}^2 \times S^1$, where S is the navigation environment of the AV.

In this work, we are concerned with path planning method based on Dijkstra method [204] and motion planning using Timed-Elastic Band (TEB) [205, 206] in both simulation and real experimental platform based on Ackermann model.

Path planning (Global planner)

This subsystem component computes the fastest route through the road network to reach the next checkpoint in the mission, based on knowledge of road blockages, speed limits, and the nominal time required to make special manoeuvres such as lane changes or U-turns.

The RA is setting the waypoints to move the AV in the environment. These high-level commands are then sent to the path planning ROS node to generate the route for the AV from the starting point to the desired destination. The entrance of the parking lot represents the starting point, and the destination is a free parking slot, which is an unknown place that needs to be discovered by the perception system while exploring the area. We used a ROS-based Dijkstra algorithm [207] for its simplicity and efficiency. This method represents the roads as a directed graph with weights represents the cost of passing a road segment. This process starts with a set of nodes (free space) that the AV can navigate and assigning a cost value to each one of them, this value is then increased with the next nodes, and the algorithm needs to find a path with minimum cost.

After finding the appropriate path, all the nodes in that path are translated into positions $P_i = (X_i, Y_i)^T$ in the reference axes. The outcome is not smooth, and some points are not compliant with the vehicle kinematics, and geometry, hence the second stage is necessary (motion planning).

Dijkstra algorithm

Dijkstra [208] is a Breadth-First-Search (BFS) algorithm for finding the shortest paths from a single source vertex to all other vertices. It processes vertices in increasing order of their distance from the source, which are also called root vertices. The shortest path between two vertices is a path with the shortest length (i.e. least number of edges), also called link distance.

Let $G = (U, V)$ be a weighted undirected graph, with weight function $w : E \rightarrow$

R mapping edges to real-valued weight. If $e(u, v)$, then we write $w(u, v)$ for $w(e)$.

The length of a path $p = \langle v_0, v_1, v_2, v_3 ... v_k \rangle$ would be the total of the weight of its constituent edges as in equation below:

$$length(p) = \sum_{1}^{k} w(v_{i-1}, v_i) \tag{4.1}$$

The distance from u to v, denoted by $\delta(u, v)$ is the length of the minimum path if there is a path from u to v; and ∞ is otherwise.

The general idea of Dijkstra's algorithm is to report vertices in increasing order of their distances from the source vertex while constructing the shortest path tree edge by edge; at each step adding one new edge, corresponding to the construction of the shortest path to the current new vertex. This is accomplished in the following steps:

1. Maintain an estimate $d[v]$ of the $\delta(s, v)$ of the shortest path for each vertex v.

2. Always $d[v] \geq \delta(s, v)$ and $d[v]$ equals the length of a known path ($d[v] = \infty$ if we have no path so far).

3. Initially, $d[s] = 0$ and all other $d[v]$ values are set ∞. The algorithm will then process the vertices one by one in some order. The processed vertex's estimate will be validated as being the real shortest distance; i.e. $d[v] = \delta(s, v)$.

The term "processing a vertex u" means finding new paths and updating $d[v]$ for all $v \in adj[u]$ if necessary. The process by which an estimate is updated is called relaxation. When all vertices have been processed, $d[v] = \delta(s, v)$ for all v.

Motion planning (Local planner)

The motion planning layer is responsible for executing the current motion goal issued from the behaviours layer. This goal may be a location within a road lane

when performing nominal on-road driving, a location within a zone when traversing through a zone.

In all cases, the motion planner creates a path towards the desired goal, then tracks this path by generating a set of candidate trajectories that follow the path to varying degrees and selecting from this set the best trajectory according to an evaluation function. This evaluation function differs depending on the context, but includes consideration of static and dynamic obstacles, curbs, speed, curvature, and deviation from the path. The selected trajectory can then be directly executed by the vehicle. For more details on all aspects of the motion planner, see [209].

In order to transform the global path into suitable waypoints, the TEB motion planner creates a shorter waypoints $P_i = (x_i, y_i, \theta_i)^T$ within the original path planner waypoints (as much as possible) taking into consideration the vehicle constraints and the dynamic obstacles. Hence the map is reduced to the area around the AV and continuously updating.

When the path planning node determines the path of driving to be performed in the current context, then the ROS-based TEB local planner algorithm we used [210] will translate this path into a shorter continuous trajectories that are feasible for the control system and actuators to track and follow, this trajectory should also avoid collision with obstacles, detected by the sensors on-board, also should be comfortable for the passengers. In case there is an object nearby then the agent will check the possibility of collision using Prism model checker and then modify the trajectory when needed.

Timed Elastic Band (TEB)

A Timed Elastic Band [211] is composed of a fixed number n of geometric waypoints or vehicle poses P_i. The following equations are based on the vehicle coordinate system shown in figure 4.2. The set of waypoints is described by:

$$Q = \{P_i\}_{i=1...n} \tag{4.2}$$

where each waypoint consists of the tuple:

$$P_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix} \tag{4.3}$$

Two consecutive waypoints are separated by a time interval ΔT_i. Our approach considers these time intervals as constant as in contrast to robot navigation there is no objective of a fastest trajectory. In the context of collision the timed elastic bands merely optimises the location of intermediate waypoints as there are no boundary conditions for the final vehicle state. The set of time intervals is given by:

$$\tau = \{\Delta T_i\}_{i=1\ldots n-1} \tag{4.4}$$

The vehicle velocity, turn rate and accelerations are obtained from the finite differences between a pair or triple of consecutive waypoints:

$$v_i = \frac{\sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}}{\Delta T_i} \tag{4.5}$$

The change of the velocity yields the longitudinal acceleration:

$$a_{x,i} = \frac{v_i - v_{i-1}}{\Delta T_i} \tag{4.6}$$

The course angle is defined by:

$$\lambda = \arctan\left(\frac{y_i - y_{i-1}}{x_i - x_{i-1}}\right) \tag{4.7}$$

The change of the course angle yields the course rate:

$$\dot{\lambda}_i = \frac{\lambda_i - \lambda_{i-1}}{\Delta T_i} \tag{4.8}$$

Based on the course rate and the velocity the lateral acceleration is given by:

$$a_{y,i} = v_i \dot{\lambda}_i \tag{4.9}$$

The total acceleration is described by:

$$a_{tot} = \sqrt{{a^2}_{x,i} + {a^2}_{y,i}} \tag{4.10}$$

The change of the acceleration, also called jerk, in both lateral and longitudinal direction is given by:

$$\dot{a}_{x,i} = \frac{a_{x,i} - a_{x,i-1}}{\Delta T_i} \tag{4.11}$$

$$\dot{a}_{y,i} = \frac{a_{y,i} - a_{y,i-1}}{\Delta T_i} \tag{4.12}$$

4.3.4 Control system

A control system is necessary to execute the proposed trajectory of the AV. The move_base [212] control node provides an implementation of actions by executing acceleration, brake, and steering messages. The move_base node provides a ROS interface for configuring, running, and interacting with the navigation stack. The move_base node receives a list of waypoints and target velocities generated by the planning subsystem and passes them to an algorithm that determines how much to steer, accelerate/brake, in order to follow the target trajectory.

The process of carrying out the planned motion and correct tracking errors is determined by the amount of input for each actuator, and a feedback controller controls this process. These tracking errors are due to the inaccuracies of the vehicle model; hence, this closed-loop system is vital to ensure the robustness and stability of the operation. Running the move_base node on the AV ROS system that is appropriately configured results in accurate trajectory following of the trajectory that

attempts to achieve a goal pose with its base to within a user-specified tolerance.

4.4 Conclusion

In the first part of this chapter, we went through the fundamental details related to the AV system in a MATLAB/Simulink model where it gives the reader the necessary introductory information for studying the different characteristics for the AV design and parking lot operation.

In the second part, we presented our method of the designed system that consists of the model of Tata Ace vehicle, the autonomous system for the vehicle that has been connected to the rational agent and verification system presented in chapter 3. It is important to mention again that the novelty of this chapter is represented by the fact that such a useful system is introduced for the first time. A system that is based on free and open-source platforms represented by the model of a full-size vehicle along with it real dynamic model, sensors and actuators, all controlled by a novel decision-making system that built from the ground to satisfy the need of safe driving in real-life driving scenarios. This decision-making system is connected to the vehicle platform from one side and to a novel verification system built for the ground to provide a safety layer over the decisions made by the rational agent.

We also want to mention here that the initial idea was to design the 3D model for the vehicle and its sensors, also the parking environment in ROS and Gazebo simulator, while to design the AV perception and control systems in MATLAB. After a few months of work, this method did not work properly, and there were many problems related to the synchronisation and slow processing of data in MATLAB also related to the connection of MATLAB to an external physical system. MATLAB is not capable of controlling a complex real-time physical system, at least at the meantime with the limited resources and capabilities of the related toolboxes, including the parallel computation and real-time toolboxes. Hence we then built the whole autonomous system in ROS.

Chapter 5

Implementation and Experimental Setup

5.1 Introduction

This chapter is a complement to the previous one. Here we presented our experimental AV, the Tata Ace electric prototype as shown in figure 5.1. The autonomous system we designed for this vehicle is described in chapter 4; hence we only mentioned in this chapter the implementation and the experimental setup that consists of: the sensors, main processing units, CAN bus system, and actuators including the steering wheel and rear wheels driving controllers. The methods take into account the distributed computing system we installed on the Ace-EV. Safety, low-cost, and ease of implementation are the central themes in this work. The reason for this engineering or technical contribution mentioned in this chapter is to support some requirements of the industry. The techniques and algorithms used in this chapter are off-the-shelf; however, those have not been designed to work seamlessly together in one system; therefore, we modified them based on the system needs so that they become compatible with each other and with the hardware tools used. These modifications have been mentioned in some details with the related sections in this chapter.

This chapter is dedicated to the third research question of 'How to use the ROS-based autonomous vehicle system to drive a real vehicle in real life driving scenario in a parking lot environment?'.

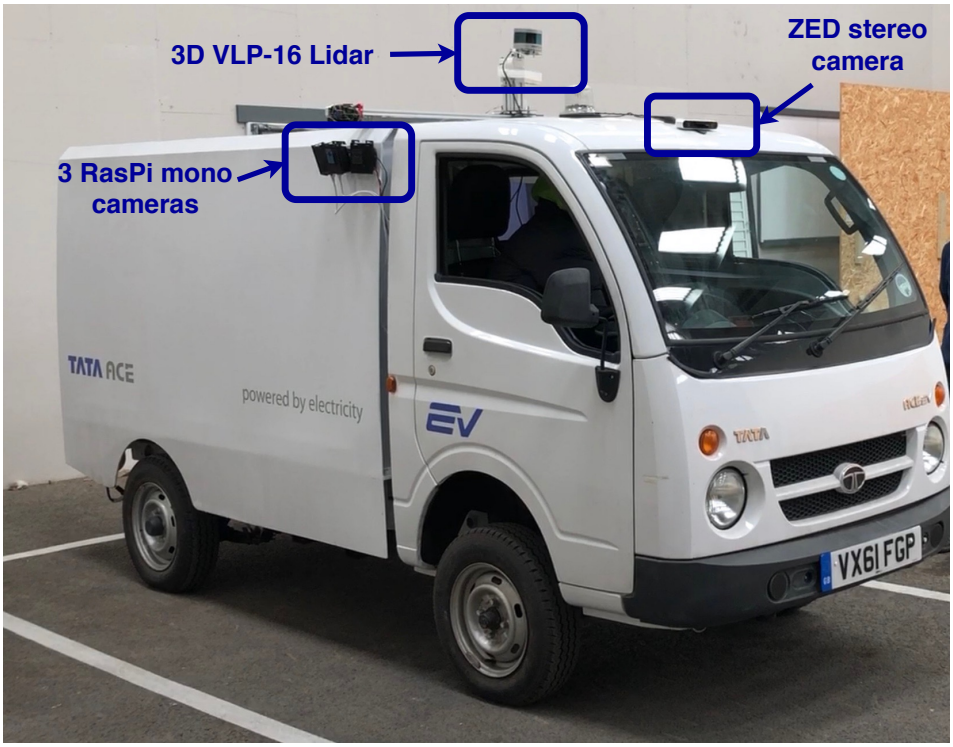


Figure 5.1: Our electric experimental vehicle showing the hardware and sensors attached to the body of the vehicle. The sensors configuration is the same as the AV in simulation in figure 4.8.

5.2 Tata Ace electric vehicle

Tata Ace electric vehicle was first launched into the UK market in 2012. The model that we got from Tata motors research centre is a custom-built electric vehicle modified for autonomous driving. Modifications were driven by the need to provide computer control and also to support safe and efficient testing of algorithms. Thus, modifications can be classified into two categories: those for automating the vehicle and those that made testing either safer or easier. A commercial off-the-shelf drive-by-wire system was installed and integrated into the Ace EV with electric motors to turn the steering column, depress the brake and acceleration pedal, because it

is an electric vehicle, there was no need to apply computer-controlled transmission shaft. The computer and communication systems were fitted in the cargo area.

Ace EV maintains normal human driving controls (steering wheel, brake and acceleration pedals) so that a driver can quickly and efficiently take control during testing. Ace EV has its original seats in addition to a custom centre console with power and network outlets which enable developers to power laptops and other accessories, supporting longer and more productive testing. The van has also been modified from a flatbed van to a panel van to provide a better place for a person to sit inside for extended periods indoors or outdoors, also to protect human occupants in the event of a collision or roll-over during testing. For autonomous operation, a safety button is fitted to disable autonomous driving and stop the vehicle.

Ace EV van has two independent power busses. The main power bus remains intact with its 80VDC/140Ah coming from 10 batteries (each 8V/140Ah) connected in series. This source used to provide power for the two DC motors connected to the rear wheels and to provide the power for the rest of the dashboard, steering wheel motor and the break/acceleration system. We also fitted an auxiliary 12VDC/230Ah power system connected to an inverter to provides power for the communication hardware along with three computers and eight mini-computers working together onboard the van.

For computation, Ace EV uses two laptops each core i7, with 16GB of RAM and SSD storage for faster handling of data. It also uses one Nvidia Jetson TX2 computer-on-board running on ARM (Cortex-A57 (quad-core) + Nvidia Denver2 (dual-core)) CPU, also 8GB of RAM and connected to external SSD for storage. The system also connected to 8 Raspberry Pis (model 3B) mini-computers onboard. All the computers and sensors are connected through a gigabit Ethernet network for reliable and fast communication.

Ace EV uses a combination of sensors to provide the redundancy and coverage necessary to navigate safely in a parking lot environment. The configuration of sensors on Ace EV is illustrated in table 5.1.

5.3 Perception system

Humans are continuously receiving information from the environment using our five primary senses, and we will not be able to decipher these data and know what is relevant without perception. An example for this is when we drive on a highway; there is a continuous stream of information around the driver such as the signs on the road, lane markings, nearby vehicles, pedestrians passing, and traffic jams. We need to be able to make instant decisions to maximise our profits while making sure all the agents around are safe; those decisions should also follow the rules of driving. This ability to extract and understand the relevant information from the surrounding is the central pillar for the safe and meaningful operation of any AV. Through perception, the AV can detect and track the other objects around from a distance using the cameras and other onboard sensors, also to identify any potential threats. The perception capability in the AV is available for 360° around the AV. Perception comes as a first stage in the computational pipeline in any AV system. It has got much attention that can be seen through the rapid development of the sensors and detection algorithms. Once the AV can extract relevant knowledge from its environment, it can plan its path and move safely, all without human intervention.

Our AV experimental perception system consists of one LiDAR sensor (Velodyne VLP-16) connected to the main computer running state-of-the-art LOAM Velodyne package for state estimation and mapping, one stereo camera (ZED stereo camera) connected to Nvidia Jetson TX2 running deep-learning state-of-the-art YOLOv3 object detector for detecting objects (pedestrians and vehicles) around the AV. Eight mono cameras connected to eight Raspberry Pi running OpenCV-based Aggregated Channel Features (ACF) object detector for detecting objects and aruco markers. The sensors configuration is shown in the schematic diagram in figure 5.2.

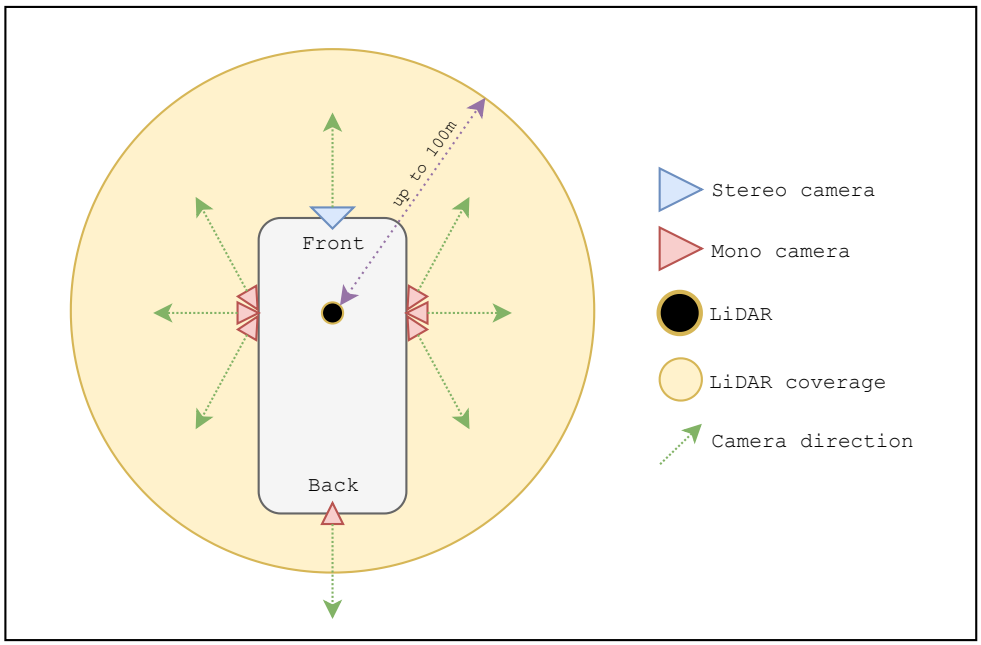


Figure 5.2: Schematic diagram of the perception system sensors configuration and their direction. Both the camera system and the LiDAR system provide 360° coverage each.

Table 5.1: Properties of sensors for the real testbed.

Sensor type	**No. of sensors**	**Resolution**	**No. of frames /Speed of rotation**
LiDAR	1	3D 16-layer (up to 100m) 360°H/ 30°V 300,000 PPS	600 RPM
Stereo camera	1	Colour 1344x376	10 FPS
Mono camera	8	Colour 640x480	6 FPS

5.3.1 Camera system

Our eyes can be considered as the main perception sense among our five primary senses for moving around. For the AV, the same thing could be said about the cameras as it is the most accurate way to create a visual representation of the environment. AVs relies on cameras attached to the body of the vehicle to create together a 360° field of view. Some have a wide field of views such as a stereo camera or fish-eye camera, and others have a narrower view like the conventional mono cameras.

Although they provide accurate visuals, cameras still have their limitations. It is difficult for them to detect objects in low visibility conditions, like rain, fog, or night time, it is also difficult for the mono cameras to provide accurate depth information of the detected objects.

Stereo camera

At the time of writing this thesis, ZED is still one of the best stereo cameras available in the market to sense space and motion accurately for outdoors usage. It is also fully compatible with ROS through ROS wrapper and also comes with an adequate price, for these reasons this camera has been used. As mentioned, we used one ZED stereo camera placed at the front of the AV to perceives the world in three dimensions, as shown in figure 5.1 and figure 5.3. Using binocular vision and high-resolution sensors, the camera can tell how far objects are around the AV from 0.3m to 25m at up to 100 FPS (depending on the set resolution and speed of the host machine). Full technical specifications of this sensor are listed in table 5.2. ZED stereo camera is connected to a host processing computer (Nvidia Jetson TX2).

Table 5.2: Technical specifications for the ZED stereo camera [10].

Video Output	
Output Resolution	Side by Side 2x (2208x1242) @15fps 2x (1920x1080) @30fps 2x (1280x720) @60fps 2x (672x376) @100fps
Output Format	YUV 4:2:2
Field of View	Max. 90° (H) x 60° (V) x 100° (D)
RGB Sensor Type	1/3" 4MP CMOS
Active Array Size	2688x1520 pixels per sensor (4MP)
Focal Length	2.8mm (0.11") - f/2.0
Shutter	Electronic synchronized rolling shutter
Interface	USB 3.0 - Integrated 1.5m cable
Depth Sensing	
Baseline	120 mm (4.7")
Depth Range	0.5 m to 25 m (1.6 to 82 ft)
Depth Map Resolution	Native video resolution (in Ultra mode)
Depth Accuracy	< 2% up to 3m < 4% up to 15m
Physical	
Dimensions	175 x 30 x 33 mm (6.89 x 1.18 x 1.3")
Weight	170g (0.37 lb)
Power	380mA / 5V USB Powered
Operating Temperature	0°C to +45°C (32°F to 113°F)

Figure 5.3: Front view to the AV showing the stereo camera, and the LiDAR which is attached to belt driven linear actuator.

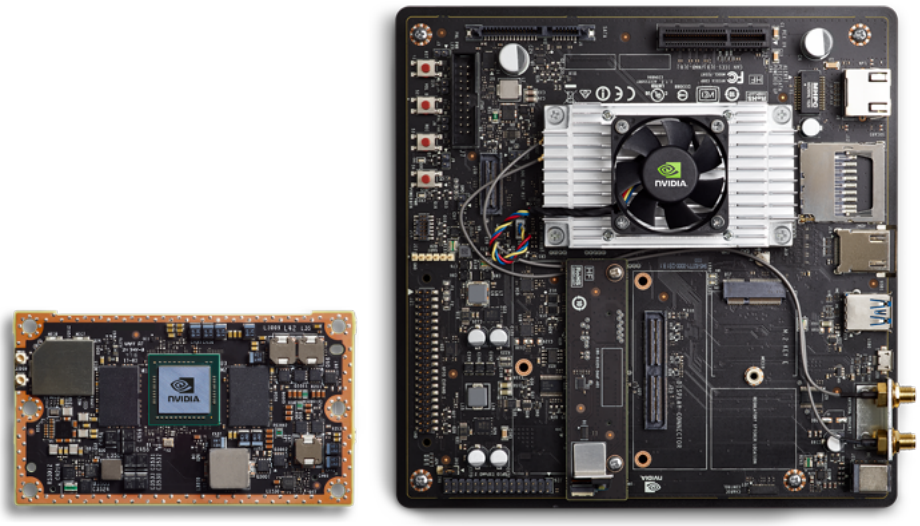


Figure 5.4: Jetson TX2 development kit

Nvidia Jetson TX2

Graphics processing units (GPUs) are currently playing an essential role in accelerating computations in the image processing area of autonomous driving. The massive parallelism provided by GPUs makes them suitable for accelerating computations and handling multiple input streams from sensors such as cameras, LiDARs, and radar. For this reason, companies including (Tesla, Audi, Volvo, etc.) have started using GPU-equipped computing platforms in realising autonomous perception features [213]. For this reason, we used the powerful commercial computer-on-board (Jetson TX2).

Nvidia Jetson TX2 shown in figure 5.4 is an embedded System-on-Module (SoM) with technical specifications mentioned in table 5.3 that is useful for deploying computer vision and deep learning applications [214], hence we used it with the deep-learning YOLOv3 objects detection to accelerate the process using such small size device. Linux operating system is installed on Jetson TX2 to run the YOLOv3 algorithm through CUDA toolkit [215] as a ROS node that is connected to the roscore in the main computer.

Table 5.3: Technical specifications for the Jetson TX2 module [11].

GPU	256-core Nvidia Pascal™ GPU architecture with 256 Nvidia CUDA cores
CPU	Dual-Core Nvidia Denver 2 64-Bit CPU Quad-Core ARM® Cortex-A57 MPCore
Memory	8GB 128-bit LPDDR4 Memory 1866 MHx - 59.7 GB/s
Storage	32GB eMMC 5.1
Power	7.5W / 15W

YOLOv3 object detector

Through just a glance, humans can know what kind of objects are in an image, where they are, and how they interact. The visual system for humans is accurate and fast, allowing us with little conscious thought to perform complex tasks like driving. Hence, we can say that fast and accurate algorithms for object detection would provide the necessary information about the environment to drive cars without specialised sensors.

Multiple fast and accurate computer vision algorithms such as: Faster Region-based Convolutional Neural Networks (Faster R-CNN) [216], Single Shot Detector (SSD) [217], You Only Look Once (YOLO) [197], can be considered as the state-of-the-art nowadays in this area. For our experimental tests and driving scenario we found that YOLO is outperforming the other methods in terms of providing accurate real-time detections. An in-depth comparison between those algorithms could be found in [218, 219].

In 2016, Redmon et al. proposed the end-to-end object detection method YOLO [5]. As shown in figure 5.5. YOLO divides the image into $S \times S$ grids and predicts B bounding box and C class probability for each grid cell. Each bounding box consists of five predictions: w, h, x, y, and object confidence. The values of w and h represent the width and height of the box relative to the whole image. The values of (x, y) represent the centre coordinates of the box relative to the bounds of the grid cell. The object confidence represents the reliability of the existing object in the box, which is defined as:

$$Confidence = Pr(object) \times IOU(truthPred.) \tag{5.1}$$

In the Equation above, Pr(object) represents the probability of the object falling into the current grid cell. IOU (truth pred.) represents the intersection over union (IOU) of the predicted bounding box and the real box. Then, most bounding boxes with low object confidence under the given threshold are removed. Finally, the Non-Maximum Suppression (NMS) method [220] is applied to eliminate redundant bounding.

To improve the YOLO prediction accuracy, Redmon et al. proposed a new version YOLOv2 in 2017 [221]. A new network structure Darknet-19 was designed by removing the full connection layers of the network, and batch normalisation [222] was applied to each layer. Referring to the anchor mechanism of Faster R-CNN, k-means clustering was used to obtain the anchor boxes. Besides, the predicted boxes were retrained with direct prediction. Compared with YOLO, YOLOv2 dramatically improves the accuracy and speed of object detection. However, as a general object detection model, YOLOv2 applies to cases where there are a variety of classes to be detected, and the differences among the classes are large, such as persons, cars, and bicycles. YOLOv3 [197] is not a big difference from its predecessor, the authors claimed small improvements to the algorithm with the training of a new classifier network that is better than the other ones.

You only look once (YOLOv3) is a state-of-the-art, real-time, single neural network object detection algorithm that depends on frame object detection as a regression problem to spatially separated the bounding boxes and associated class probabilities directly from full images in one evaluation. Since the whole detection pipeline is a single network, it can be optimised end-to-end directly on detection performance.

YOLO divides each image into a grid of $S \times S$, and each grid predicts N bounding boxes and confidence. The confidence reflects the accuracy of the bounding box and whether the bounding box actually contains an object(regardless of

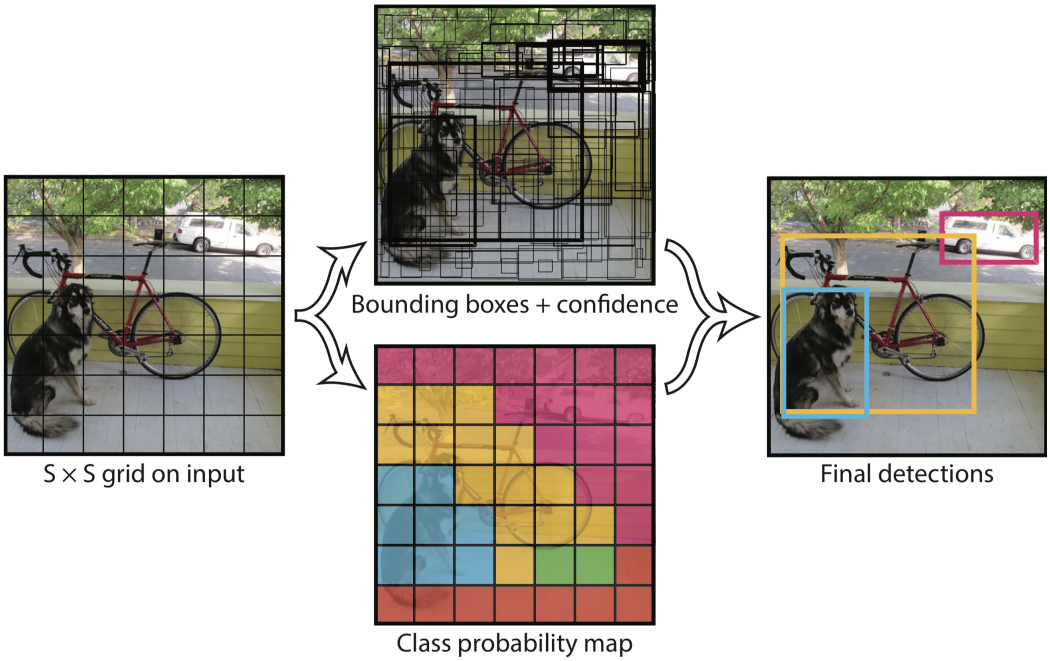


Figure 5.5: YOLO detects models as a regression problem. It divides the image into an $S \times S$ grid and for each grid cell predicts B bounding boxes, confidence for those boxes, and C class probabilities. These predictions are encoded as an $S \times S \times (B * 5 + C)$ tensor [5].

class). YOLO also predicts the classification score for each box for every class in training. It is possible to combine both the classes to calculate the probability of each class being present in a predicted box. However, one limitation for YOLO is that it only predicts 1 type of class in one grid; hence, it struggles with very small objects. So, total $S \times S \times N$ boxes are predicted. However, most of these boxes have low confidence scores, and it needs to set a threshold for confidence to remove most of them, as shown in figure 5.5.

In YOLOv3 as shown in figure 5.6, object detection is defined as a regression problem, and object bounding boxes and detection scores are directly estimated from image pixels. This approach eliminates the need for an object proposal step. First, the input image is resized to a resolution of 416×416 pixels. Next, the image is divided into 7×7 grid regions, and two centres of the bounding boxes are assumed in each grid cell. Therefore, each grid predicts two bounding boxes with their associated confidence scores (this means a prediction of at the most 98 bounding boxes per image). A single convolutional network runs once on the image to predict

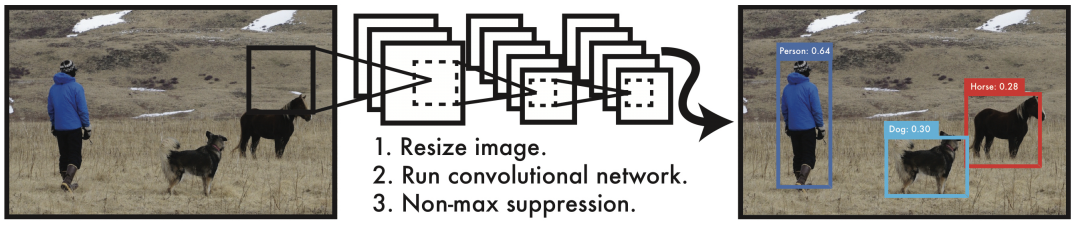


Figure 5.6: YOLO detection system. The image processing go through three stages: (1) resize the input image to 416×416 then (2) runs a single convolutional network on the image, then (3) thresholds the resulting detections by the model's confidence [5].

object bounding boxes. The network is composed of 24 convolutional layers followed by two fully connected layers which connect to a set of bounding box outputs. Finally, a non-maximum suppression is applied to suppress duplicated detections. YOLO looks at the whole image during training and test time; therefore, in addition to object appearances, its predictions are informed by contextual information in the image.

YOLOv3 have been designed to work within a PC environment using mono cameras. In this work, we used a ZED stereo camera connected to Nvidia Jetson TX2 working with Ubuntu OS 16.04. Hence we did some slight modifications to configure the algorithm for the different processing machine and the different type of camera, also to enable accurate range measurement using the stereo camera. The final result was very satisfying in terms of quality and speed as shown in table 5.1, also figure 5.15 and the video demonstration with the link provided at the end of this chapter.

Mono cameras

The mono camera system is consist of: The host computer, the camera module, and the objects detector algorithms for vehicles/pedestrians and the aruco markers.

Table 5.4: The Raspberry Pi model 3 B+ Specs [12].

SOC	Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC
CPU	1.4GHz 64-bit quad-core ARM Cortex-A53 CPU
RAM	1GB LPDDR2 SDRAM
WIFI	Dual-band 802.11ac wireless LAN (2.4GHz and 5GHz) and Bluetooth 4.2
Ethernet	Gigabit Ethernet over USB 2.0 (max 300 Mbps).
Thermal management	Yes
Video	Yes – VideoCore IV 3D. Full-size HDMI
Audio	Yes
USB 2.0	4 ports
GPIO	40-pin
Power	5V/2.5A DC power input
Operating system support	Linux and Unix

The host computer (Raspberry Pi 3 Model B+) and its camera module

As mentioned, we needed to cover 360° around the vehicle, and this has been done based on one stereo camera and eight mono cameras. The front-facing camera is the most important; hence we used a stereo camera with a deep-leaning object detector running on the Jetson TX2. However, it is not possible to adopt the same approach for the rest of eight cameras as this is very expensive and not necessary in our parking lot scenario; hence we used a much-less expensive method by using the budget Raspberry Pi system to run the object detectors with decent accuracy and speed as shown in table 5.1 also figure 5.11 and the video referred to in the end of this chapter.

The Raspberry Pi is a mini-computer, System-on-Module (SoM). It uses an ARM-compatible CPU and GPU units included in Broadcom system featured by Pi model. For the model used in this work (Pi 3 Model B+), the speed of the CPU varies from 700 MHz to 1.2 GHz and onboard memory of 1 GB RAM; detailed specs are shown in table 5.4. Each Pi is connected to a camera module. We used v2 Pi camera that has a Sony IMX219 8-megapixel sensor. The camera module can be used to take high-definition video, as well as stills photo [223]. The Raspberry Pi module, the camera module, and the mounting method is shown in figure 5.7. Raspberry Pi is running ACF object detector through a light version of Linux/Ubuntu called (Lubuntu 16.04), and ROS installed.

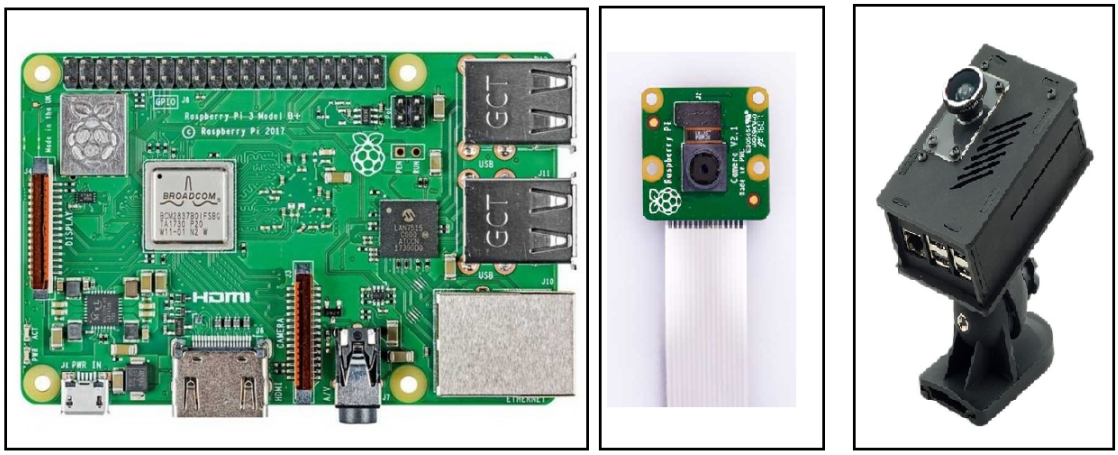


Figure 5.7: Raspberry Pi module, Raspberry Pi camera V2 module, and Raspberry Pi case for mounting on the vehicle.

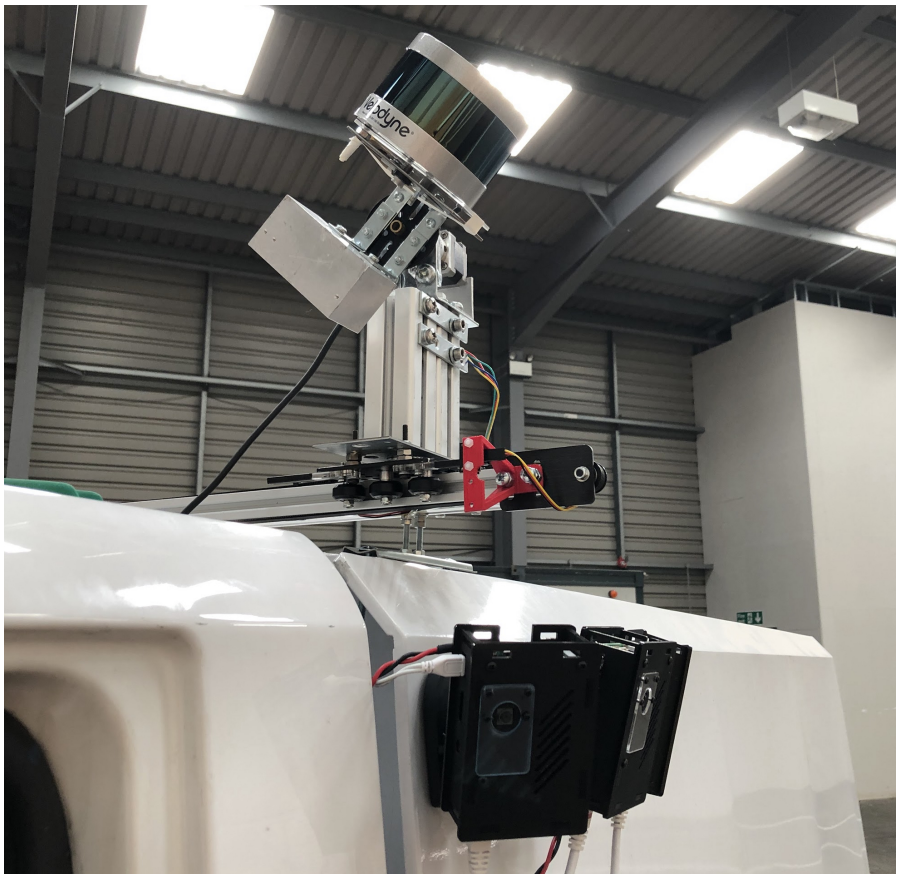


Figure 5.8: Side view to the AV showing three Raspberry Pi cameras attached and the LiDAR dragged to the left side of the AV then tilted for better view to the area near the left side of the AV.

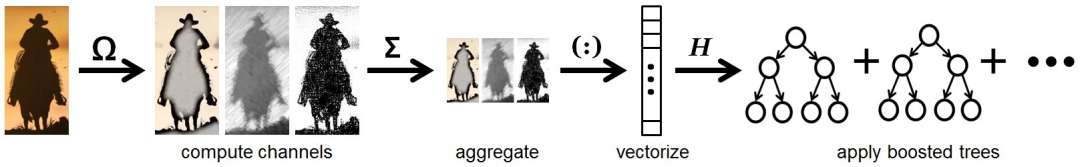


Figure 5.9: Overview of the ACF detector. Given an input image I, compute several channels $C = \Omega(I)$, sum every block of pixels in C, and smooth the resulting lower resolution channels. Features are single pixel lookups in the aggregated channels. Boosting is used to learn decision trees over these features (pixels) to distinguish object from background. With the appropriate choice of channels and careful attention to design, ACF achieves state-of-the-art performance in pedestrian detection [6].

ACF object detector for vehicles and pedestrians

Real-time image processing on a low-cost embedded system is still a challenging research area. For this embedded platform, there is a trade-off between accuracy and processing time. We used the Aggregated Channel Features (ACF) detector algorithm (to detect pedestrians and cars) that can perform in real-time on a Raspberry Pi embedded system while still keeping a decent accuracy.

ACF detector proposed in [6] used a combined feature which consists of three channels of LUV colour space (where L stands for luminance, whereas U and V represent chromaticity values of colour images), a normalised gradient channel and a six-channel Histogram of oriented Gradient (HoG) and then arranged in a boosted tree. The block diagram of the ACF detector method is presented in figure 5.9. ACF detector will extract the proposal region consisting of the positive area and negative region. Positive proposal regions are obtained from training data containing the bounding box of the ground truth area. Meanwhile, the negative proposal region is extracted automatically by the sliding window in all image area except the bounding box of ground truth area.

Let $I(x, y)$ be the RGB image of $m \times n$ size consisting of three channels. Firstly, the image is transformed into the LUV colour space, and then the gradient magnitude of the image I is calculated using the following formula.

$$M(i,j) = \sqrt{(\frac{\partial I(i,j)}{\partial x})^2 + (\frac{\partial I(i,j)}{\partial y})^2} \tag{5.2}$$

Also, the gradient orientation of image I is expressed by the following equation:

$$O(i,j) = \tan^{-1}(\frac{\frac{\partial I(i,j)}{\partial y}}{\frac{\partial I(i,j)}{\partial x}}) \tag{5.3}$$

where $\frac{\partial I(i,j)}{\partial x}$ derivative I at the coordinates (i,j) in the x-direction and $\frac{\partial I(i,j)}{\partial y}$ is the derivative I at the coordinates (i,j) in the y-direction. The gradient image is smoothed using a convolution operation between the gradient image and a triangular filter [1 2 1] / 4. After that, the smoothed image is normalised to get the details of the gradient scale by the following equation:

$$\widetilde{M}(i,j) = \frac{M(i,j)}{S(i,j) + c} \tag{5.4}$$

where $S(i,j)$ is the smoothed image and c is a small normalisation constant, e.g. $c = 0.005$.

The steps to compute the HoG with the bin number = 6 and the cell size = 4 are as follows. For each sub-window called a cell with 4×4 size, the normalised gradient, $\widetilde{M}(i,j)$, is quantised into six bin histograms, with the orientation range 0-180, based on the value of $O(i,j)$. The assignment of the gradient in the q-oriented bin uses the linear interpolation. The HoG feature size is $\frac{m}{cellsize} \times \frac{n}{cellsize} \times bin\_number$. The aggregate features obtained are arranged in a decision tree and trained using bootstrapping and AdaBoost classifier alternately and repeatedly as many N stages. At each step, the negative examples are extracted and accumulated with the previous ones [6, 224].

With the appropriate choice of channels and careful attention to design, ACF achieves satisfying performance in pedestrian detection [6, 225], also in cars detection [226, 227]. A sample of this method for pedestrians and vehicles detection in a parking lot is shown in figure 4.13. Each Raspberry Pi is also running Aruco

marker detection.

Aruco marker (Fiducial) detection

The standard method in parking scenarios to find a free parking space is to detect the white parking rectangle painted on the ground that point out to an unoccupied parking spaces, this sometimes could be difficult to detect if it is partially covered by the next parked vehicle or if there are vehicles parked on both sides of the free parking space where those will make the detection of the white rectangle from a distance a difficult problem. This method also does not provide the same level of information for the AV compared with the method used in this work.

We used a method called aruco marker detection (similar to QR codes) [228]. This method allows the robot to determine its position and orientation by looking at fiducial markers that are fixed in the environment of the robot [229, 230]. In our case, we flipped the concept, which means that the AV can determine the position and orientation of the aruco marker if the position and orientation of the AV are known in the environment. Initially, the position of a marker is specified or automatically determined. After that, a map (in the form of a file of 6DOF poses) is created by observing the fiducial marker and determining the translation and rotation between the camera and that marker with the condition that all the camera intrinsic parameters are known and defined, the aruco marker size should also be known and defined. An example from our parking lot is shown in figure 5.11.

Aruco markers are binary square fiducial markers that can be used for camera pose estimation. Their main benefit is that their detection is robust, fast and simple. The aruco module includes the detection of these types of markers and the tools to employ them for pose estimation and camera calibration [231]. An aruco marker is a square marker composed by a wide black border and an inner binary matrix which determines its identifier (ID). The black border facilitates its fast detection in the image, and the binary codification allows its identification and the application

Figure 5.10: Aruco markers patterns.

of error detection and correction techniques. The marker size determines the size of the internal matrix. For instance, a marker size of 4x4 is composed of 16 bits. Some examples of aruco markers are shown in figure 5.10 below.

The aruco_detect node detect the markers, this node is running inside the Raspberry Pi board (We installed a light version of Linux and ROS in the raspberry pi) and connected to the central ROS system running in the main computer through Ethernet switch (The Ethernet switch is connecting all the sensors and the eight Raspberry Pis with the two main computers and the Jetson TX2 computer board).

For every marker visible in the image, the algorithm generates a set of vertices in image coordinates, as shown in figure 5.11. Since the size of the fiducial and the intrinsic parameters of the camera are known, the pose of the fiducial relative to the camera can be estimated.

5.3.2 LiDAR system

Laser scanners are becoming more popular among self-driving cars. During the DARPA challenge competitions, all the top-ranked cars heavily depended on laser scanners in their perception systems [15, 16]. Google self-driving car project (Waymo) heavily relay on LiDAR as well, and many other car manufacturers [232, 233]. There are multiple reasons for this preference, as mentioned in the previous chapter. Below we have presented the LiDAR used in this project which is the Velodyne VLP-16.

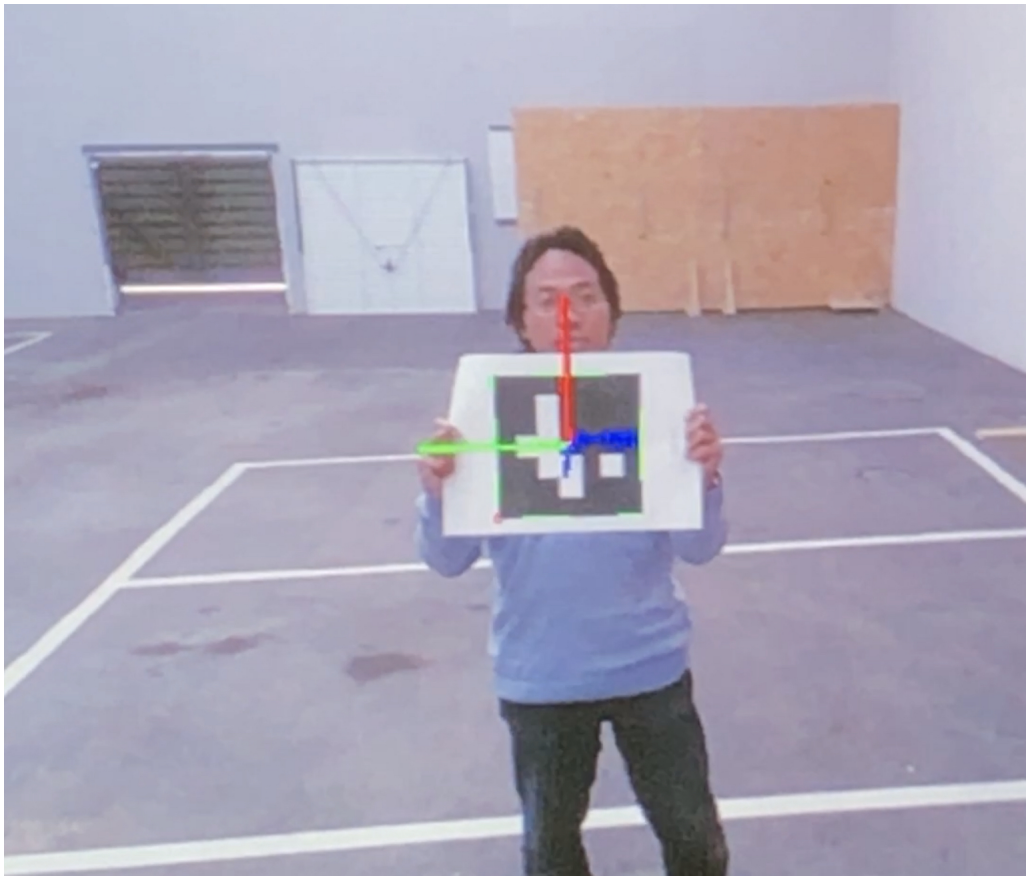
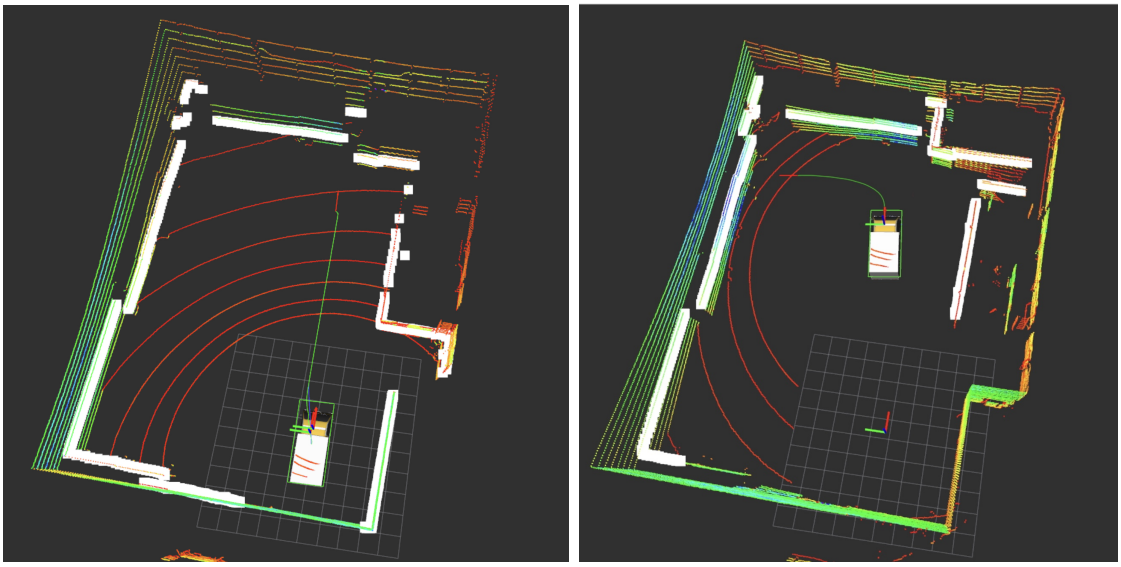


Figure 5.11: Aruco marker detection that has been used to determine a parking space position and orientation through a mono camera attached to the body of the AV.

Figure 5.12: Velodyne VLP-16 LiDAR



(a) AV start from the starting point looking for a free parking space, the first step is to move forward until the perception system detect the *Aruco* marker that indicates the existence of a free parking space.

(b) Here we can see that the AV detected the *Aruco* marker through the left first camera attached to the body of the AV, and start the process of parking based on the coordinated obtained from the aruco marker.

Figure 5.13: A 3D point cloud example generated by the LiDAR for the path planning of the AV, the coordinates are generated by the *Agent* then deliberated with the *Planning* system for guiding the AV in a safe and feasible path to reach parking space. For a detailed view of this operation, please have a look at the video refereed to in the abstract section of this thesis.

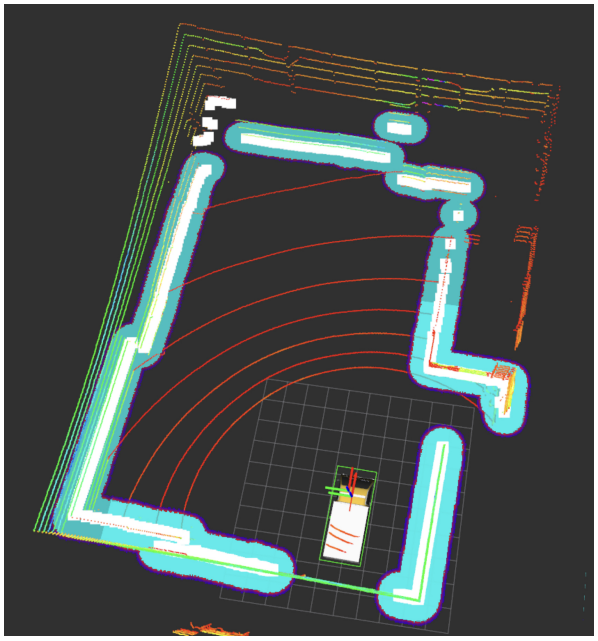


Figure 5.14: The inflation of the objects detected by the LiDAR where this adds a higher level of protection from a collision, the inflation layer is generated by ROS and controlled by the agent. The thickness of the inflation layer set by the developer and can vary from a few centimetres to a few meters.

Velodyne VLP-16 LiDAR

The Velodyne VLP-16 LiDAR (PUCK) shown in figure 5.12 is a multi-beam 3D LiDAR (Light Detection And Ranging) that scans the environment in 3D at very high speeds (up to 20Hz), generating around 300,000 points per second. It is manufactured by Velodyne Inc., a U.S. based company which evolved from the DARPA Grand Challenge in 2005. The company holds a great reputation for its class-leading LiDAR technologies, used in a variety of applications including autonomous vehicles, vehicle safety systems, 3D mobile mapping, and 3D aerial mapping [234]. The Velodyne VLP-16 LiDAR features are:

- Number of laser emitter/detector pairs: 16.
- Horizontal Field of View (FOV): 360°.
- Vertical Field of View FOV: 30°.
- Rotation rates: 5 Hz to 20 Hz.
- Maximum range of laser returns: 100 m.

- Low power consumption: 8 Watts.
- Ranging accuracy between 22 and 27 mm.

In figure 5.13 we can see using the RViz (ROS visualisation tool) the planning operation of the AV in the parking lot. In the beginning, the AV will not detect any aruco marker because it is still far from the parking spaces, hence the agent commands the vehicle to move forward as shown in part (A) of the figure. After few moments, the perception system detects an aruco marker using the left mono camera and the agent amend the trajectory through the planning operation as we can see in part (b) of the same figure, to guide the AV to the free parking space.

We have included another level of safety to the planning operation through the map inflation layer, as shown in figure 5.14. Here we can control how far we want the AV to be from objects around by adding an inflation layer with variable thickness, the value of thickness should be set by the developer. However, it could be activated and deactivated by the Agent through ROS.

Figure 5.15 shows a camera and 3D point cloud from the experimental autonomous system created by the stereo camera and VLP-16 LiDAR with rotation speed around 10Hz (600 RPM). The front camera detected a pedestrian moving in different positions. We can also see the pedestrian as a white point in front of the rendered vehicle in the point cloud, which could be used to determine the position, speed, and direction of the object. The LiDAR is placed on the top front side of the rendered vehicle, as shown in figure 5.3. The LiDAR can detect objects nearby, but with no recognition, the objects recognition in our system is only occurred in the cameras subsystem and then pointed out to the place of the object in the LiDAR point cloud.

LiDAR operating principles

LiDARs work on the principle of Time of Flight (ToF) measurement of laser beams. A laser emitter emits an infrared (typically 905nm wavelength) laser pulse and measures the time taken for the pulse to get reflected from an object. Then, the

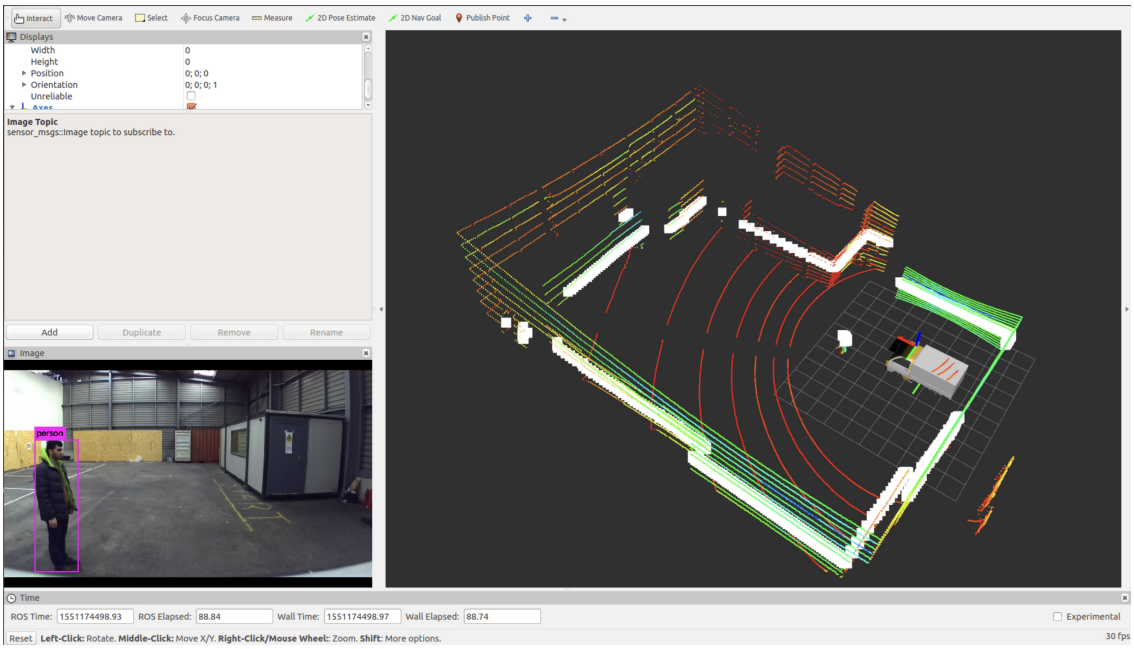
device calculates the distance of the object from the measured time. The Velodyne VLP-16 LiDAR consists of 16 laser emitter/detector pairs operating at the 905 nm wavelength. These 16 pairs fire about thousands of times per second in a sequential manner, thereby enabling users to create 3D point clouds in real-time. Figure 5.15 illustrates how the 16 laser emitter/detector pairs scan the environment.

The 16 laser emitter/detector pairs sweep horizontally gathering a series of ToF hits from the obstacles in front of them. The vertical separation between the two pairs is 2°, and this vertical angle increases exponentially with increasing distance due to beam divergence. In simple terms, the beams will appear far apart from each other as the distance increases. Apart from the ToF based distance, the VLP-16 LiDAR provides the azimuth angle, elevation angle, the reflectivity of an object with 256-bit resolution (independent of laser power), dual returns of the reflections from the transmitted laser beams (strongest and last), and timestamps based on either an attached GPS device or its internal clock. It furnishes all these data through an Ethernet port. The LiDAR also provides a neat web-server user interface that allows users to change device parameters like the port numbers, FOV, etc. [235].

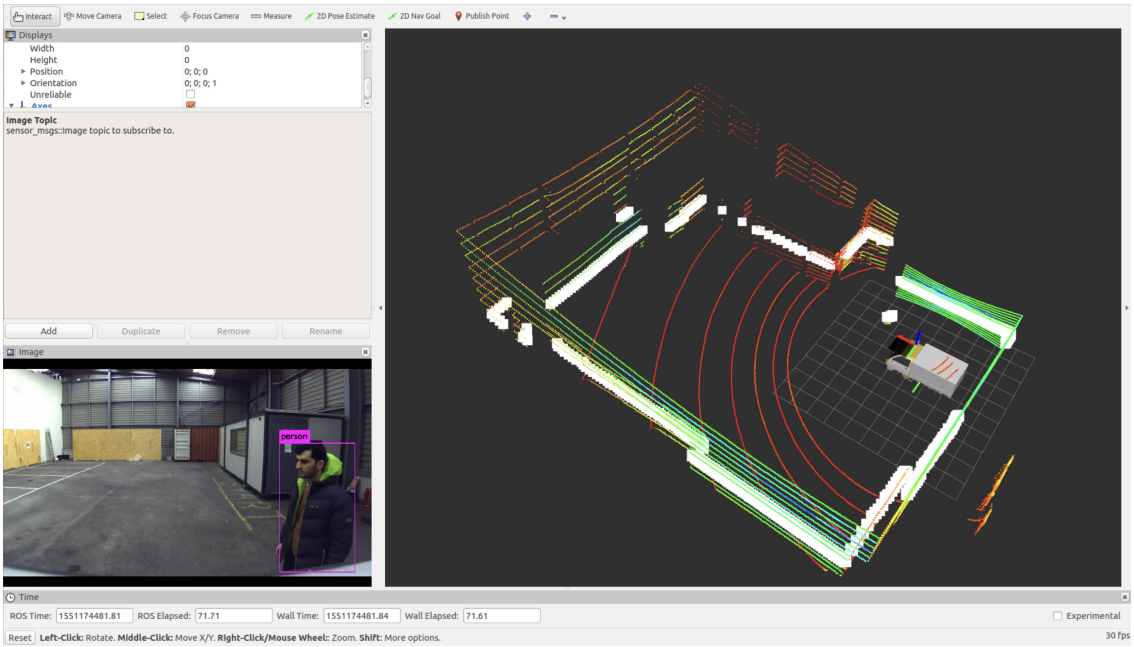
The LiDAR utilises a SLAM (Simultaneous Localisation And Mapping) package to localise the position of the AV in the environment and to build the map for that environment, this SLAM package called LOAM (Lidar Odometry And Mapping) Velodyne.

LOAM Velodyne

Mapping in 3D is a popular technology [236, 237]. Mapping with LiDARs is common as LiDARs can provide high-frequency range measurements where errors are relatively small constant irrespective of the distances measured. In the case that the only motion of the LiDAR is to rotate a laser beam, registration of the point cloud is simple. However, if the LiDAR itself is moving as in many applications of interest, accurate mapping requires knowledge of the LiDAR pose during continuous laser ranging. One common way to solve the problem is using independent position



(a) Person recognised to the left of the front side camera, we can see the same object detected by the LiDAR in RViz.



(b) Person in the right with the correspondence detection from the LiDAR frame in RViz.

Figure 5.15: A 3D point cloud example generated by a single rotation of the LiDAR sensor, the method used here is by detecting the object through the camera system, segment and classify them then project them back into the LiDAR frame for acquiring their data: distance, position, direction, and velocity. We can see in this figure a person detected by the front stereo camera and the correspondence LiDAR detection.

estimation (e.g. by a GPS/INS) to register the laser points into a fixed coordinate system. Another set of methods use odometry measurements such as from wheel encoders or visual odometry systems [238, 239] to register the laser points. Since odometry integrates small incremental motions over time, it is bound to drift, and much attention is devoted to the reduction of the drift (e.g. using loop closure).

The key idea to obtain a high-level of performance is to divide the typically complex problem of SLAM, which seeks to optimise a large number of variables simultaneously, by two algorithms. One algorithm performs odometry at a high frequency but low fidelity to estimate the velocity of the LiDAR. Another algorithm runs at a frequency of an order of magnitude lower for fine matching and registration of the point cloud [7].

As laser scanner scans while rotating, like the Velodyne VLP-16 LiDAR scans with a full-spherical rotation, and a full spherical rotation forms a sweep. LOAM extract edge point and surface point in each scan by evaluating the smoothness of each point of a sweep. The smoothness is calculated by:

$$c = \frac{1}{|S| \cdot \parallel \boldsymbol{X}^{L}_{(k,i)} \parallel} \parallel \sum_{j \in S, J \neq i} (\boldsymbol{X}^{L}_{(k,i)} - \boldsymbol{X}^{L}_{(k,j)}) \parallel \tag{5.5}$$

Where L denoting in the LiDAR frame, k denoting at the start time of the k^{th} sweep, i/j denoting the i^{th}/j^{th} point, S denoting the consecutive point set of point i, and X denoting the coordinates of points in the LiDAR frame at the start time of the k^{th} sweep. The points with the maximum c value are extracted as edge points and points with minimum c value as surface points.

Having all the features extracted, first LOAM transforms the feature points in current sweep and last sweep to the same coordinate, namely the end time of last sweep LiDAR frame. During the transformation, LOAM use linear interpolation of pose transforms (r, t) between two sweeps to correct the distortion as the scanner rotates when the platform is moving. Then it finds the correspondence of each feature point. The searching processing is occurred by minimising the distance of features to the corresponding lines or surfaces to get the optimal transform (r, t)

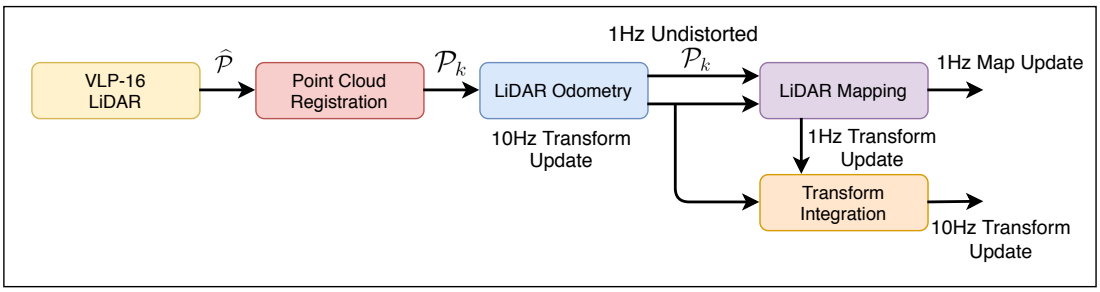


Figure 5.16: Block diagram of the LiDAR odometry and mapping software system [7].

between two sweeps. As each feature point to the corresponding line or surface provides a distance, LOAM stacking all the distances to a nonlinear function:

$$f(r,t) = d \tag{5.6}$$

Computing the Jacobian matrix J of f with respect to (r,t), then Eq. 5.6 can be solved by nonlinear iteration method, namely the Levenberg-Marquardt method:

$$T \leftarrow T - (J^T J + \lambda diag(J^T J))^{-1} J^T d \tag{5.7}$$

Where T denote the combination of r and t.

The mapping processing of LOAM is almost the same as odometry processing. The differences are that there is no need to correct distortion and the correspondence is found between current sweep and the point clouds in the map, which is accumulated during the mapping processing by adding feature points transformed from the LiDAR frame to the map frame. The map frame is fixed on the initial LiDAR pose. The last optimal object of mapping processing is the pose related to the map frame.

Figure 5.16 depicts the software system and the split between the mapping and transform computation.

The stages in the above block diagram are as follows: the point cloud registration refers to the points received by a laser scan, which in this step are saved into

a tensor. After this, the LiDAR odometry is computed assisting which both map generation and transform integration (which is finer odometry).

Starting with feature analysis, both sub-algorithms extract feature points located on edges and planar surfaces, and merge the edge points to an edge line, and planar surfaces to planar surface patches. The geometric distributions of local point clusters have correspondences to other clusters that can be found by studying the associated eigenvalues and eigenvectors.

Plane and edge detection are done using the Random Sample Consensus (RANSAC) plane fitting method. It can be interpreted as an outlier detection method. Thus, the inliers or data that can be explained by some model parameters can be separated from the outliers. In 3 dimensions, a plane of best fit can be found, and the outliers may mark other surfaces that RANSAC can be performed on again, iteratively sectioning out all the planes in each laser scan.

After a full scan is separated into all its planes (inliers) and edges, a scan can be separated into four sub-regions each with up to 2 edge points and four planar points. After the identification of planes and edges, the Iterative Closest Point (ICP) algorithm is used to find the transformation between frames. More specifically, Iterative closest point attempts to find the translation t and rotation R that minimises the sum of the squared error between two corresponding point sets.

Based on a number of point clouds, the transformation can be identified and then the clouds can be fused to produce a single cloud. As it can be imagined, the higher the number of points, the more time it takes to perform this numerical computation and find the transformation between two clouds. For this to be done in real-time, the point cloud must be downsampled, using the method of voxel-grid downsampling. This method will merge each point with several points near it to reduce significantly the number of points while preserving much of the spatial information.

The LiDAR mapping algorithm runs at a lower frequency than the odometry algorithm and is only called once per sweep. As the pose transformation between

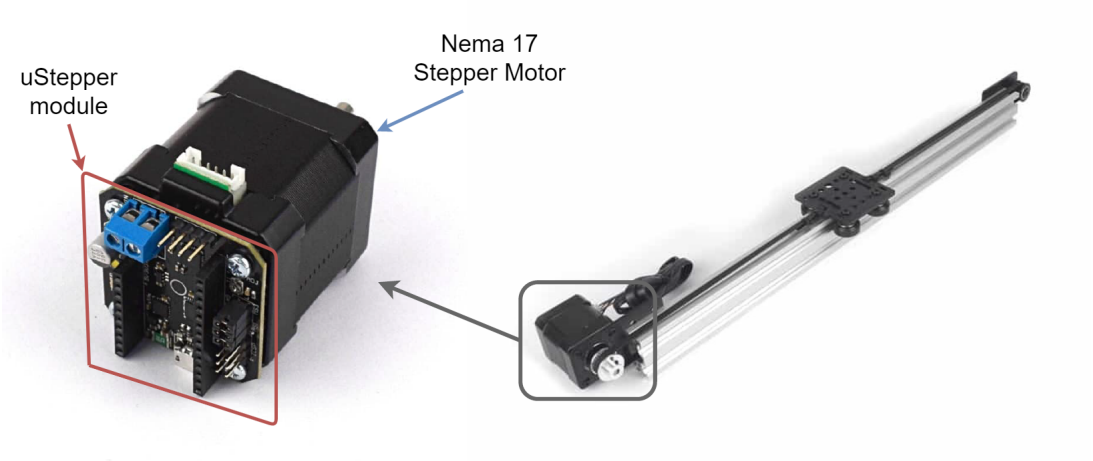


Figure 5.17: Belt driven linear actuator/Right - Stepper motor controlled by uStepper arduino board/Left.

two point clouds is known, the mapping algorithm matches and registers the point clouds in the world coordinates. As this will be done repetitively, the map will quickly grow to uphold a strong accuracy due to the LiDAR odometry.

The problem addressed in this work is to perform an ego-motion estimation with point cloud perceived by a 3D LiDAR, and build a map for the traversed environment to provide motion estimates for the guidance of an AV. The LiDAR is installed on a moving mechanism as shown in figure 5.17.

LiDAR moving mechanism

We used a belt-driven linear actuator shown in figure 5.17 placed on the top of the AV as a movement mechanism for the LiDAR, as shown in figure 5.8. The LiDAR is capable of moving between left and right sides of the AV for better visualisation of the surrounding when needed (e.g. when come to parking space and need to check distances from the vehicle aside. We build a tilting mechanism in the laboratory to enable the LiDAR not only to move freely between the two sides of the vehicle but also to tilt with a specific angle in order to have a clear vision of objects nearby. The LiDAR tilting mechanism is controlled through uStepper controller.

uStepper controller board

uStepper (Rev B) [240, 241] is a compact Arduino compatible board, with a built-in stepping motor driver and 12-bit rotational encoder, which allow the uStepper direct installation on the back surface of a NEMA 17 size stepper motor. This compact system allows developing of applications by using a stepping motor, with no need for messy wiring that we used to see in the separated Arduino-motor system. Additionally, tracking of the absolute position of the motor shaft can be done by a 12-bit rotary encoder, which allows the uStepper to identify the exact steps.

5.4 Control system

5.4.1 Controller Area Network (CAN bus)

The Controller Area Network (CAN) is an entrenched networking system, which developed by Robert Bosch during 1980s, had been explicitly designed in consideration of real-time requirements, low cost and ease handling resulted in expanding its utilisation by automotive and automation industries.

Initially, The CAN had evolved for control components interconnection in automotive vehicles. Because of complicated control functions which carried out by engine management systems, anti-lock brakes, traction controls and other systems, therefor, dedicated lines demanded to connect different control components. Nevertheless, the constant increase in complexity has resulted in a physical maximum in the number of wires demanded and the actual size of the connector. CAN facilitate a dramatic decrease in wiring complexity, besides it enables multiple devices to be connected using one pair of wires, allowing simultaneous data exchange between them. [242].

The basic features of CAN are:

- High-speed serial interface: CAN can be configured to work from a few kilobytes up to 1 Megabit transmission rates.

- Short data lengths: The short data lengths of CAN messages mean that data access time is very low compared to other systems.
- Low-cost material: can works on a simple twisted pair of wires, consequently, connecting a CAN network is less costly compared to a multi-core or Coaxial cables which often demanded by other bus systems.
- Multicast and peer-to-peer connection: With CAN, it is easy to broadcast information to all nodes or a subset of the bus and easily implement a peer-to-peer connection.
- Rapid reaction times: The ability to transmit information without the need for code or permission from the bus arbiter leads to quick reaction times.
- Troubleshooting: The high level of error detection and the number of error detection mechanisms provided by CAN devices mean that CAN is highly reliable as a network solution.

CAN operating principles

After the connection is established with the different nodes, data start to flow over a CAN network; no individual nodes are processed. Instead, an identifier message is set to act as a unique tag on its data content. The identifier determines not only the contents of the message but also its priority. When a node wants to transfer information, it just passes the data and the identifier to its CAN controller and sets the related send request. Next, it is up to the CAN controller to format the contents of the message and transfer the data in the form of a CAN frame. Once the node can reach the bus and transmit its message, all other nodes become receivers. After the message is received correctly, these nodes perform an acceptance test to find out whether the data is relevant to that specific device, based on the message ID. Therefore, the connection can be made not only on a peer-to-peer basis, where a single node accepts the message but also to implement simultaneous broadcasts and connections where multiple nodes can accept the same message using a single

transmission. Moreover, the ability to transmit data on an event basis means that Bus load usage can be kept to a minimum.

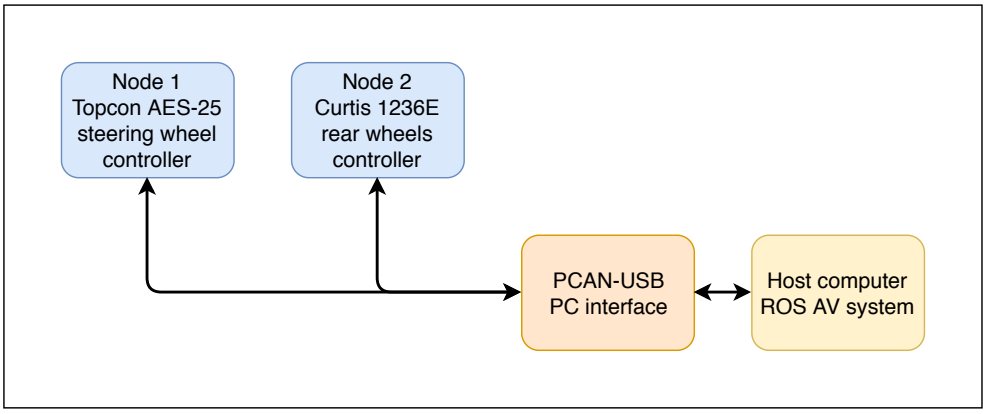


Figure 5.18: CAN bus system showing the two driving nodes for the AV.

Lateral control (Topcon AES-25 Steering wheel controller)

The AV is factory equipped with a lateral control (Topcon AES-25 Steering wheel controller). The motor controller generates the electrical signals to drive the motor to a prescribed angular position. The steering control system prescribes the position to which the motor is driven and communicated to the motor controller via a CAN bus link. The angular position of the motor is determined by an incremental optical encoder mounted inside the motor housing. The encoder provides the ability to resolve the 4096 steps per shaft revolution and indicate the direction of rotation. Four equally spaced index marks per revolution allow accurate commutation for the motor drive electronics [243].

Longitudinal control (Curtis 1236E rear wheels controller)

The AV is also factory equipped with a longitudinal control (Curtis 1236E) that provides advanced control of AC induction motors performing on-vehicle traction drive. This controller provide a highly cost-effective combination of power, performance and functionality to control the rear wheels powered by AC motor through CAN bus system [244]. The technical details about the vehicle control system are further explained in appendix B.

5.5 Conclusion

In this chapter, we have presented the hardware used to implement the autonomous system on the experimental Tata ace eclectic vehicle. The main parts of the autonomous system have been presented in the previous chapter; hence, we did not went through the same topics again but only presented the hardware parts of the system.

The perception system consists of one Velodyne LiDAR VLP-16 that is attached to a belt-driven linear-actuator on the top of the AV. The LiDAR can observe a 360° view around the vehicle, and it can also move left or right using a stepper motor when it reaches one side of the vehicle it can then tilt with a specific angle controlled by ROS through uStepper Arduino board as shown in figure 5.8. The perception system also has a high-resolution stereo camera that has been optimised to work with outdoors robots, and it can accurately estimate the distance of objects up to 20m from the vehicle, this is connected to Nvidia Jetson TX2 board for running the YOLOv3 object detector, this board running a Linux/Ubuntu 16.04 and ROS kinetic system that is also connected to the main roscore on the main computer through Ethernet connection. There are also eight mono cameras connected to Raspberry Pi 3 each. Those are running a light version of Ubuntu/Lubuntu 16.04 and ROS and connected to the main roscore through Ethernet connections as well.

Those sensors have been used for their low cost and adequate performance. The autonomous control system is connected to the physical vehicle actuators through the CAN-bus network, as shown in figure 5.18. This system proved to be capable of driving and guiding the vehicle efficiently in a parking lot, and this has been tested in a small parking lot designed for this purpose in the AMRC research centre in Sheffield / UK, as shown in the videos demonstration link attached below. The AV is capable of detecting and avoiding objects while driving, looking for a free parking space and commencing the parking operation. The prototype vehicle was designed with an emergency button that is easy to access when needed to stop the vehicle [1].

[1] `https://tiny.cc/mohammed-av-2019`

Chapter 6

Case Study

6.1 Introduction

This chapter presents a case study for both design-time and run-time verification of the agent logic and reasoning cycle. This agent is controlling the AV in simulation for a parking lot environment. The AV start the work autonomously from the entrance of the parking lot and continue until it finds a free parking space and executes the parking action successfully. Before going into the details of the verification system, we first presented an example of the agent code shown in figure 6.1 then we defined the meaning of its sentences (The agent reasoning code is attached in appendix A).

6.2 Agent design for a parking lot environment

An example of the agent code is shown in figure 6.1 that is cyclically repeated to guide the AV through the reasoning cycles and generate PTP models for the nearby traffic participants based on perception predicates.

In the example, the formation of plans is shown for an agent undertaking an autonomous parking manoeuvre. In this case, plans represent the agent's actions;

```
PERCEPTION PROCESS
Monitor the following Boolean:
Parking space located.{[],[0,5]}
Pedestrian detected.{[],[-2,4]}
Generate PTP for Pedestrian.
{[I am at global waypoint],[0,0]}

EXECUTABLE PLANS
//Plan 1
If ^~[Parking space located] then +^[Need to explore parking
   lot.].

//Plan 2
If ^[Need to explore parking lot] then [Generate exploration
   waypoints.]
[Update drive mode.].

//Plan 3
If ^[Free parking lot detected] then -^[Need to explore
   parking lot]
+^[Commencing parking operation].

//Plan 4
If ^[Commencing parking operation] then [Generate parking
   waypoints.]
[Update drive mode.].

//Plan 5
If ^[Pedestrian detected] while  ^[Distance more than 3m and
   less than 6m] and
^[Object getting closer] then [Activate slow mode.]
[Generate object avoidance waypoints.]
+^[Object PTP generated.]
[Update drive mode.].

//Plan 6
If ^[pedestrian detected] while  ^[distance less than 3m] then
[Activate stop mode.]
[Update drive mode.].

//Plan 7
If ^[moving vehicle detected] while  ^[distance more than 6m
   and less than 12m] and
^[object getting closer] then
[Generate object avoidance waypoints.]
+^[object PTP generated]
[Update drive mode.].
.
.
.
```

Figure 6.1: Example of the agent code used to control the AV, the full code has been listed in appendix A.

each is represented by a triggering condition. The perception process represents sensing data that is collected on every evaluation. The '^[...]' represents the evaluation of a belief condition that can be set by an internal event. In this case, both plans start by evaluating whether a specific belief is matched. Should this belief be matched, a series of actions is then planned, again any element headed '^[...]' shows then update of a belief. Elements shown within square brackets are executable sentences that contain code defined deeper within the structure which links to actuation.

Plan 1 can be read as follows: if I believe that no free parking space is detected, then I believe that I need to explore the parking lot. This is then extended by Plan 2, which can be read as if I believe that I need to explore the parking lot then a set of exploration waypoints should be generated, and these should be uploaded to activate the drive mode. Plan 3 is used to capture the condition when a parking space is detected and can be read as: if I believe that I have detected a free parking space, then I remove the belief that I need to explore the parking lot, and I believe I can commence parking operation.

Plan 4 contains the high-level code with trigger for planning this movement: if I believe that I can commence the parking operation, I should generate a set of waypoints for the parking space and update the drive mode to reflect this. Plans 5, 6, and 8 can be read as two pairs; each deals with the detection of an object, either a person or a moving vehicle. Generally, if the object is detected at a distance more than 12m then no action is required, if the distance is less than 12m and more than 6m, then do generate an object avoidance waypoints and update the drive mode. If it is detected at a distance less than 6m and higher than 3m, then the drive mode is switched to a slower mode, and a new set of waypoints is generated to avoid the object; otherwise, the vehicle is stopped.

6.3 Verification of decision making for a parking lot scenario

This section presents an example of a parking lot scenario, where the AV is searching for a free parking space. In order to showcase the approach, we will consider here a basic scenario in which the AV must decide what to do in a situation where it detects an obstacle and/or a complex environment. In the case of an obstacle, the AV may decide it can plot a safe path around it whereby it considers the obstacle to be avoidable, and so retains control. During this process, the RA will continuously monitor the road users in its environment and decides its actions and trajectory based on the data from the perception system. The RA then checks all the probability of success of the intended actions before any execution using PRISM model checker.

MCMAS is used to verify (during design-time) the rules (mainly represent the driving rules) and actions (the movements of the AV based on the environmental situation) that need to be considered within zone 1 and 2 of the AV, as shown in figure 6.3. We used a limited set of predicates (rules, plans and actions) for the parking lot scenario as a proof of concept; real-life driving scenarios will need more rules to determine the proper behaviour of the AV.

The AV needs to build a feasible trajectory and to maximise the distance from the objects around a suitable cost-map. The movements of the traffic participants usually amenable to a probabilistic model, as based on the type of environment. A trajectory for a pedestrian walking in a parking lot is estimated by a prediction method also accounting for previously collected datasets in similar scenarios, e.g. [245, 246]. A pedestrian may keep walking at the same speed if there is a car passing nearby or could reduce the speed, stop or change the path; the same idea can be implemented for car drivers, taking into consideration the driving rules and the car dynamics.

In this work, the agent generates probabilistic behaviour models for the non-

stationary objects based on the observed information and from previously recorded data for behaviour of pedestrians and drivers in similar real-life scenarios. The details of this operation have been clarified in chapter 3/section 3.2.3. The verification system will take into consideration probabilities for the moving objects, verifying the intended actions of the AV against them using the PRISM probabilistic model checker, to select the most likely-to-succeed trajectory/action for execution. The agent keeps updating the probabilistic models of the dynamic objects and sends it to an onboard PRISM in each reasoning cycle of the agent.

This operation is repeated as long as there are no objects within zone 1 of the AV shown in figure 6.3, as soon as one of the moving objects come across zone 1 then the AV will stop. If there is any moving object within zone 2, then the AV will halve the speed, based on pre-programmed rules and the PRISM feedback that verifying the predicted movement of the other objects and choosing the best action to execute.

6.3.1 Design time verification of agent logic in MCMAS

Here we first define then verify a set of predicates that will form the rules of basic driving, to be used later by the AV during its run-time operation as shown in table 6.1 and figure 6.2. The number of those rules could rapidly increase depending on the driving scenario and the environmental situation. While it is challenging for the designer to check that there is no conflict between them manually for this simple case study, it will be even harder when taking into account other general driving scenarios.

It is essential to mention that there is no direct connection between MCMAS and the agent reasoning cycle because this verification process is carried out at design-time. The programmer needs to design the sets of predicates for the agent, then use those in the verification software to check their stability and consistency, correct any errors that the model checker refer to through the counterexample then it is safe to use them with the agent code to take control of the vehicle during

run-time operation.

Predicates definition

Here we define three sets of predicates: *sensing abstractions*, *future events consequences*, and *actions*, as listed in table 6.1. The operational logic of the RA is restricted to the parking lot scenario. The RA will choose its decisions based on the sensory abstractions and a set of rules, as shown in figure 6.2, those rules determine the best action to be carried out by the AV based on the sensing abstractions and the possible future event consequences. MCMAS is used to compute with the resulting Boolean evolution system to verify the logical stability and consistency of those predicates.

```
56    Evolution:
57
58
59        AM1=true if FSNE1=true or FSNE2=true or FSNE3=true or SON1=true
60        or SON2=true or UON1=true or UON2=true or UON3=true;
61
62        AM1=false if FSNE1=false and FSNE2=false and FSNE3=false and
63        SON1=false and SON2=false and UON1=false and UON2=false
64        and UON3=false or AA1=true;
65
66        AM2=true if (SONO1=true or SONO2=true or SONO3=true or
67        FSFE1=true or FSFE2=true or FSFE3=true or SOF1=true or SOF2=true
68        or UOF1=true or UOF2=true or UOF3=true) and
69        (AM1=false and AA1=false and AA2=false);
70
71        AM2=false if (SONO1=false and SONO2=false and SONO3=false
72        and FSFE1=false and FSFE2=false and FSFE3=false and SOF1=false
73        and SOF2=false and UOF1=false and UOF2=false and UOF3=false) or
74        (AM1=true or AA1=true and AA2=true);
75
76    .
77    .
78    .
79    .
80    .
```

Figure 6.2: Fragment of the agent's evolution rules in MCMAS.

Table 6.1: List of sensory and action's abstractions of the agent.

1- The sensory abstractions of moving objects (Zone 1 / outside the trajectory of the AV) are:
   - SONO1: pedestrian detected.
   - SONO2: vehicle detected.
   - SONO3: object detected.

2- The sensory abstractions of moving objects (Zone 1 / outside the trajectory of the AV but predicted to come across) are:
   - FSNE1: pedestrian detected.
   - FSNE2: vehicle detected.
   - FSNE3: object detected.

3- The sensory abstractions of moving objects (Zone 1 / within the trajectory of the AV) are:
   - SON1: pedestrian detected.
   - SON2: vehicle detected.
   - SON3: object detected.

4- The sensory abstractions of moving objects (Zone 2 / outside the trajectory of the AV but predicted to come nearer) are:
   - FSFE1: pedestrian detected.
   - FSFE2: vehicle detected.
   - FSFE3: object detected.

5- The sensory abstractions of moving objects (Zone 2 / within the trajectory of the AV) are:
   - SOF1: pedestrian detected.
   - SOF2: vehicle detected.
   - SOF3: object detected.

6- The sensory abstractions of moving objects moving fast (Zone 2 / outside the trajectory of the AV but predicted to come nearer) are:
   - FASP1: pedestrian detected.
   - FASP2: vehicle detected.
   - FASP3: object detected.

7- The sensory abstractions of moving objects moving fast (Zone 2 / within the trajectory of the AV) are:
   - FISP1: pedestrian detected.
   - FISP2: vehicle detected.
   - FISP3: object detected.

8- The sensory abstractions for parking the AV:
   - PSA: parking space available.
   - PSNA: parking space not available.

9- The future events (consequences) for moving objects (Zone 1) are:
   - FCN1: pedestrian detected will hit.
   - FCN2: vehicle detected will hit.
   - FCN3: object detected will hit.

10- The future events (consequences) for moving objects (Zone 2) are:
   - FCF1: pedestrian detected may hit.
   - FCF2: vehicle detected may hit.
   - FCF3: object detected may hit.

11- The movement actions available to AV:
   - AM1: brake to stop.
   - AM2: proceed in reduced speed (2 mph).
   - AM3: proceed in normal speed (5 mph).

12- The parking actions available to AV:
   - AA1: generate new motion plan for parking.
   - AA2: return back to previous motion plan.

Worst case mathematical model

Each rule can be verified by computing the minimum space-time distance of the evolution of the progress of the oncoming car/pedestrian/object (denoted by E) and that of the AV (denoted by V):

$$E: \ E_c + [v_e t\cos(\alpha),\ v_e t\sin(\alpha), st],\ t > t_c \tag{6.1}$$

$$V: \ V_c + [v_a t\cos(\beta),\ v_a t\sin(\beta), st],\ t > t_c \tag{6.2}$$

which describe the future movements of the environmental object and the AV, respectively. t_c is the current time when sensing of E and AV decision have been completed, and E_c and V_c are the oncoming objects and the AV position at the time of the sensor measurement are abstracted, and the decision is made by the AV about what to do. We say that no collision occurs in the worst case if the geometric distance (in 2D) of these two lines is greater than $1m$ for any possible heading angle α and positions E_c as shown in figure 6.3. s is a time separation factor defined as $s = 1m/s$ to make the dimensions in space-time compatible and used as a scaling factor for time equivalence of space separation (the smaller s is chosen, the bigger will be the time difference requirement for two objects occurring in the same place). The validity of all rules in figure 6.2 have been checked using this type of simple worst-case analysis.

Table 6.2: Properties of the verified agent logic in MCMAS.

Execution time in (sec.)	**Number of reachable states**	**BDD memory in use (bytes)**	**Peak number of nodes**
0.038	1.05267e+06	6641468	16352

Verifying the predicates in MCMAS provided the data listed in table 6.2. From this table, we can see that the time needed for MCMAS to verify the agent predicates is (0.038 sec.), this time is not critical compared with the time needed for PRISM to verify the PTP model. The reason is that the MCMAS is used during the design-time stage, so even if it took a long time, it would not negatively affect

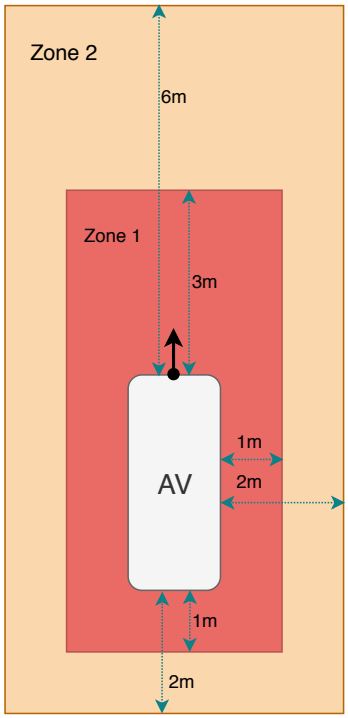


Figure 6.3: Schematic for the two zones defined in the agent rules in figure 6.2, the black arrow refers to the direction o driving, any moving object access to zone 2, the AV will slow down, while accessing zone 1, the AV will stop until it is clear then move again.

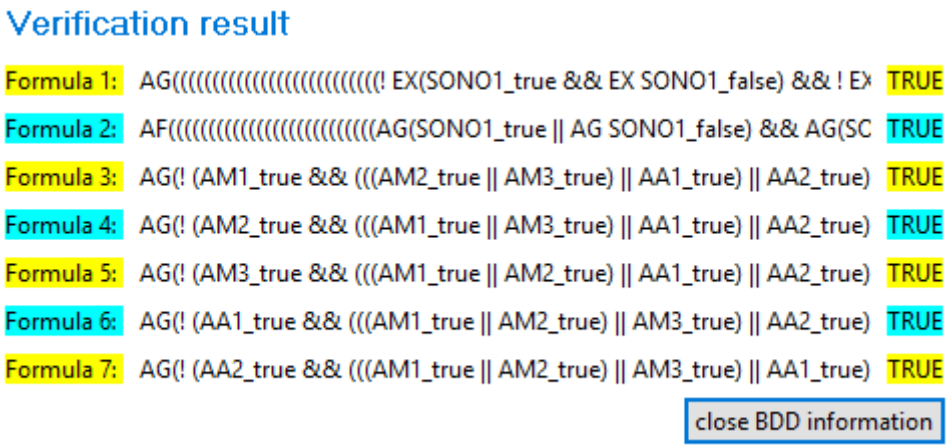


Figure 6.4: Part of the checking formulas in MCMAS with their verification result.

the decision-making system. The result of verifying the sets of queries explained in chapter 3/section 3.3.1 is shown in figure 6.4. Those are the queries that the developer ask the MCMAS model checker to check the stability and consistency of the agent logic. From figure 6.4, we can see that all the seven formulas ask to the model checker, their result are (true) means that the agent logic developed is stable and consistent. If any formula gave a (false) result, then the model checker will provide a counterexample to show which state or set of states does not satisfy the properties in queries. Then it will be easy for the developer to correct the logic based on the given counterexample.

6.3.2 Verification of agent decision making in PRISM

Probabilistic decision-making and threat-assessment methods assign probabilities to different events, e.g., how likely it is to collide with another object in the next few seconds given some assumptions on uncertainties. Figure 6.5 illustrates the proposed scenario for the AV in terms of trajectory generation based on the possible behaviour of other objects around where the AV is moving forward looking for a free parking space, at the same time, a pedestrian and a vehicle (P2, V1) is moving towards the AV, another pedestrian (P1) standing in position (x=3.5m, y=4m) from the AV in relative coordinates. The RA will generate PTP models for the two traffic participants (P2, V1) and also for the AV to find the best trajectory and speed under the current circumstances. The RA will keep updating the PTP models with every reasoning cycle (100 ms -this time is set by the developer) and verifying those PTPs using PRISM model checker.

Because the object (P1) is not moving and it is outside (zone 2), also not in the same path of the AV, hence the agent will ignore it, and the AV will continue moving in same speed (5mph). However, if the pedestrian (P1) entered (zone2), then the AV will reduce the speed according to sensory abstraction (SOF1) and action (AM2). For demonstration purposes, we discretised the trajectory by one meter apart. We also discretised the possible pedestrian's and vehicle's trajectories. While during the AV operation, the RA is getting these data continuously in real-

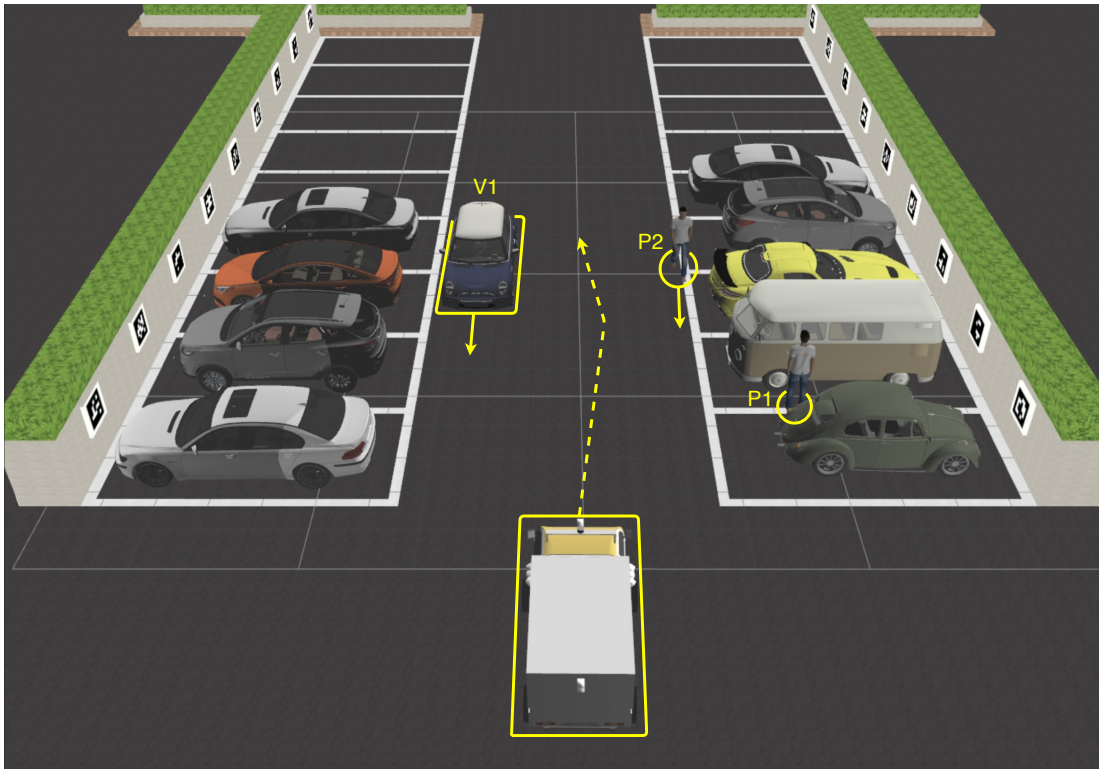


Figure 6.5: A driving scenario in simulation.

time from the perception system without the need for discretisation.

For the agent to build a meaningful PTP model while the AV is moving, it has been formerly equipped with a possible probabilistic behaviour for both pedestrians and drivers in such an environment. This possible probabilistic behaviour is designed and equipped to the agent mind during the design phase only as a proof of concept; hence it cannot be generalised to other driving scenarios, and it cannot be guaranteed an accurate prediction for real-life scenarios, it only presented here to show a simple yet precise and efficient example.

For instance, when a pedestrian notice that s/he is walking towards a moving vehicle, the pedestrian may slow down with a high probability. The pedestrian may also choose to stop at some point or even change the lane to a safer one, it is also common that the pedestrian may be distracted by something, e.g., using a mobile device, and hence, does not notice the AV. If this is the case, the pedestrian continues to walk at the average speed. The last case could be included in the generated model of the traffic participants using methods explained in [247]. While there are

some similar probabilities for the driver with limits to the dynamic movements of the vehicle, the driver may decide to continue driving same speed, reduce the speed or to stop in order to give a chance for the AV to pass easily.

From the above scenario in figure 6.5, we can see that the vehicle and the pedestrian are in the same horizontal line and this gives a small gap for the AV to pass through,

To simulate a realistic scenario, and to equip the RA with a possible behaviour of pedestrians and drivers, we recorded manually small dataset for such objects behaviour in a few parking lots, and we also used JAAD dataset [248] for pedestrians and drivers reactions to vehicles around them in different scenarios. A possible future work step mentioned in chapter 7 is to equip the AV system with a more accurate prediction method, e.g. [246, 249] beside the current one to precisely predict the behaviour of objects around the AV instead of the limited accuracy approach used in this work.

Regarding the AV exploration of a parking lot, it is essential to mention that efficient exploration of the environment is outside the scope of our work, and our system is not equipped with a proper space exploration method. It is only equipped with a general planning method that is enough for small and simple parking lots. In the case of large or complex parking lots, our AV planning system will struggle to guide the AV efficiently. The vehicle will start moving rapidly, and it may go through the same road a few times before finding a free parking space (in case there is a free parking space in the parking lot). We have pointed out this issue in chapter 7 as part of possible future work.

6.3.3 Example in detail

As mentioned before, all the possible states of the system can be explored during formal verification, including rare cases that may be difficult to discover during simulation and testing. A general parking scenario has been presented to illustrate the use of the RA predicates (sensory abstractions, future event consequences, and

actions) designed for this case study. We have defined two regions around the AV for safety purposes depending on the direction of movement, as shown in figure 6.3. Assuming the AV is moving forward, as soon as the AV detect an object within ($6m$) in front or ($2m$) any other side represented by (zone 2), the AV will slow down from average speed of ($5mph$) to ($2mph$), as soon as this object become within ($3m$) from the front of the AV or ($1m$) from any other side represented by (zone 1), the AV will stop. Here it is essential to mention that the experimental AV has been equipped with a mean for communication with other pedestrians and drivers using audio to prevent a deadlock state when the AV stop and wait for others to move and vice versa, the AV will play a recorded sound saying to others for example "you are free to move and the vehicle will be waiting for you". We have defined further details for the AV to deal with the traffic participants around by calculating the speed of those objects using the LiDAR sensor. Assuming there is an object moving fast towards the AV, as soon as this object enter (zone 2) the AV will stop instead of slowing down as shown in table 6.1, this will give more time for the other object (running pedestrian or fast-moving car) to reduce their speed, change direction or to stop, and this will reduce or eliminate any possible collision.

A simple proposed example of how the agent chooses its actions is as follows: based on the scenario in figure 6.5 there is a car (V1) coming on the opposite direction of the AV from a distance of ($8m$), and the driver starts to slow down when notice another vehicle coming, the AV is moving at its average speed and building its trajectory based on the map and the moving objects around. As soon as the other vehicle (V1) enter (zone 2) the sensing event (SOF2) from table 6.1 will be activated, and this will activate the future event (FCF2) then this will trigger action (AM2) which leads to slow down. In the meanwhile, the walking pedestrian (P2) reached within (zone 2), sensing event (FSFE1) and this may lead to collision according to a future event (FCF1), the AV here will not take any further action because it is already working in reduced speed. However, as soon as the car or the pedestrian or both entered (zone 1) (SON2 or FCNE1 or both), future event (FCN1 or FCN2 or both), this will trigger the action (AM1) to execute stop action. All the stationary parked cars in the parking lot will not be considered as a threat because

they are not in the proposed path of the AV, and they are not moving.

The regulation for the speed of vehicles in a parking lot is varying between different countries and it is usually between ($5mph$ and $10mph$), based on this and for safety reasons and prototype development we set the speed of our AV to be ($5mph$) in case of no moving objects within (zone 1 and 2).

As we mentioned, both the RA and the planning system will send control commands to the move_base control system to set the movements of the AV. However, actions like (AA1 and AA2) have a pre-programmed sequence for performing a parking manoeuvre as shown in the video link we referred to in the abstract.

In case there are two or more rules in conflict with each other, MCMAS will present this case by a counterexample showing how the inconsistency is reached. Also, it cannot be the case that two opposite actions are activated at the same time.

For the run-time verification, the initial PTP model generated by the RA for the AV's trajectory is shown in figure 6.6. We used a relative coordinate system considering that the LiDAR position on the top of the AV is the centre of the coordinates at any time, knowing that the RA is taking the dimensions of the AV into calculations while processing. In this example, we will refer to the coordinates of the participants according to a fixed moment at a particular time interval (x_1, y_1) to represent the coordination of the AV, (x_2, y_2) for the object (P2), the (x_3, y_3) for the object (V1). The (C) letter in the PTP models represents the *clock*, which will be counting and resetting with every transition.

Figure 6.7 shows the PTP model for the pedestrian's possible behaviour. For this example, we assume that the average speed of the pedestrian is near to the speed of the AV inside the parking lot. The pedestrian may prefer to stop after noticing the AV with a probability of (0.1) or to stop later when the distance became critical. We assume that the pedestrian will keep walking in the same lane with a probability of (0.6), s/he could also decide to change the lane and walk behind the moving car (V1) for more safety with a probability of (0.3). In both cases, the pedestrian may prefer to walk at the same speed or to reduce it with some

probabilities, as shown in figure 6.7.

Figure 6.8 shows the possible PTP model for (V1). We assumed that the driver might notice the AV and decide to stop with probability (0.1). With a probability of (0.6), the driver may decide to slow down, or may prefer to continue the same speed with a probability of (0.3). The RA will then modify the AV's PTP model according to the newly generated behaviour model of the other objects around.

We would like to clarify again and as mentioned before in section 6.3.2 that those probabilities representing the different movements of objects in a parking lot environment assigned to different events are pre-programmed to the agent mind during the design phase as a proof of concept, and it cannot be generalised to different driving scenarios. It is saved as a large table with direct access from the RA in run-time AV operation while building the PTP models.

Note that the parameters used to generate the PTP models, such as the speed and probability, may not reflect the exact behaviour of the AV, P2, or V1. The RA is building those PTPs based on the location, speed, and direction of the moving objects. In general, this framework will help to predict a possible behaviour for the different objects around, then to verify the current trajectory/action for the AV against the possible trajectory/action of the nearby objects, and this will help in reducing the possibility of collision. More accurate PTP models could be generated after collecting more behaving data through real driving tests.

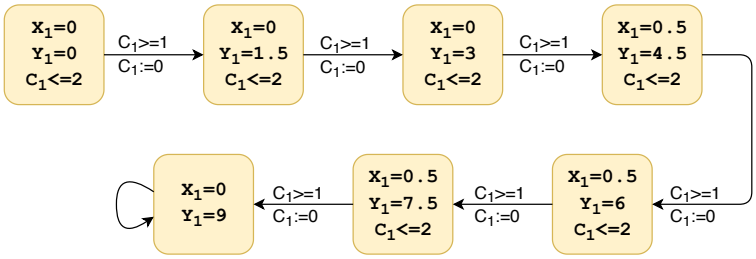


Figure 6.6: Initial PTP model generated for the AV's behaviour.

In case the probabilities assigned to different objects movement are not reflecting the real movements of those objects at any particular time and to avoid any possible collision, we require that the pedestrian and/or the vehicle is at least $(1m)$

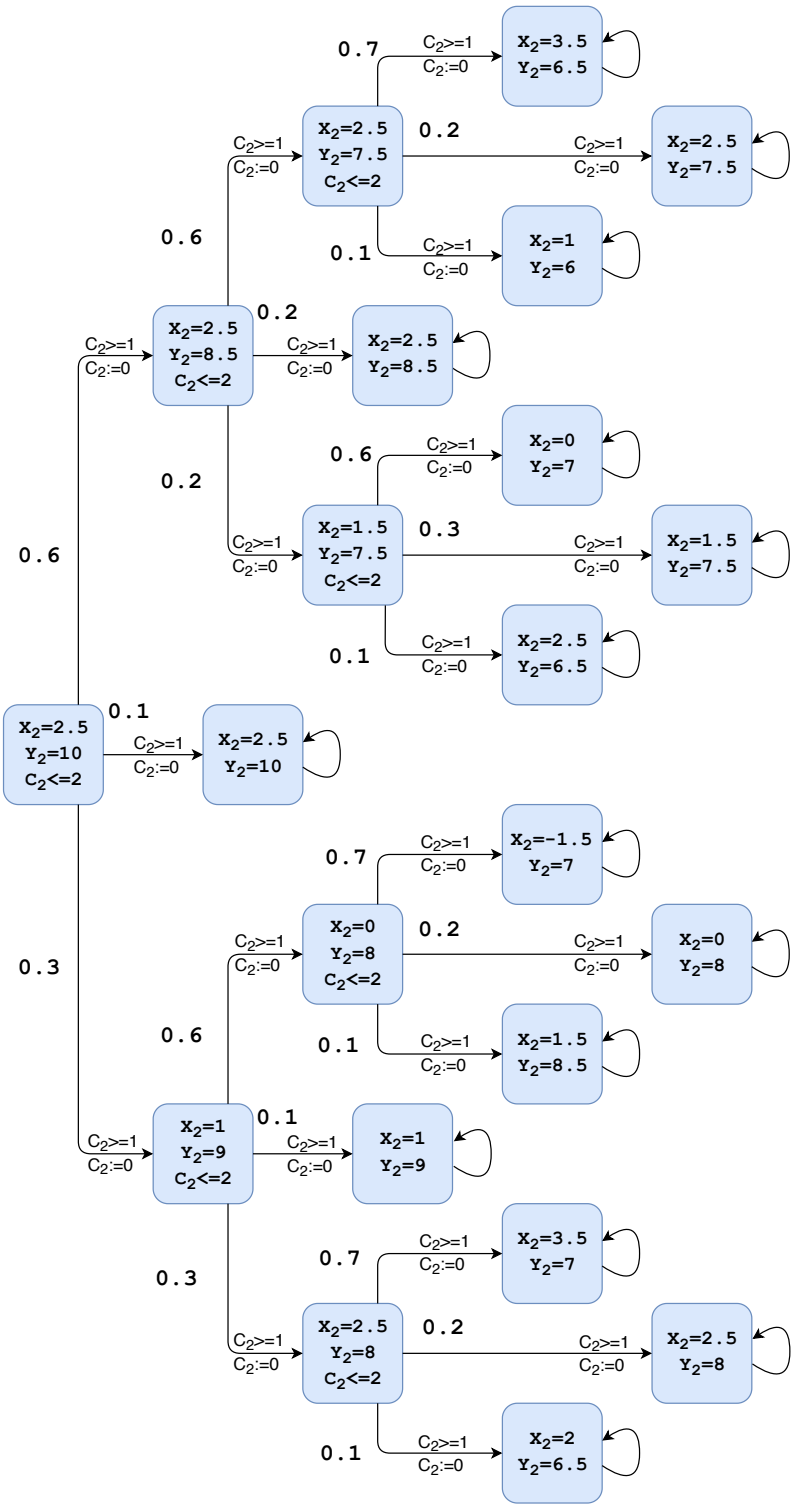


Figure 6.7: PTP model generated for the pedestrian's behaviour.

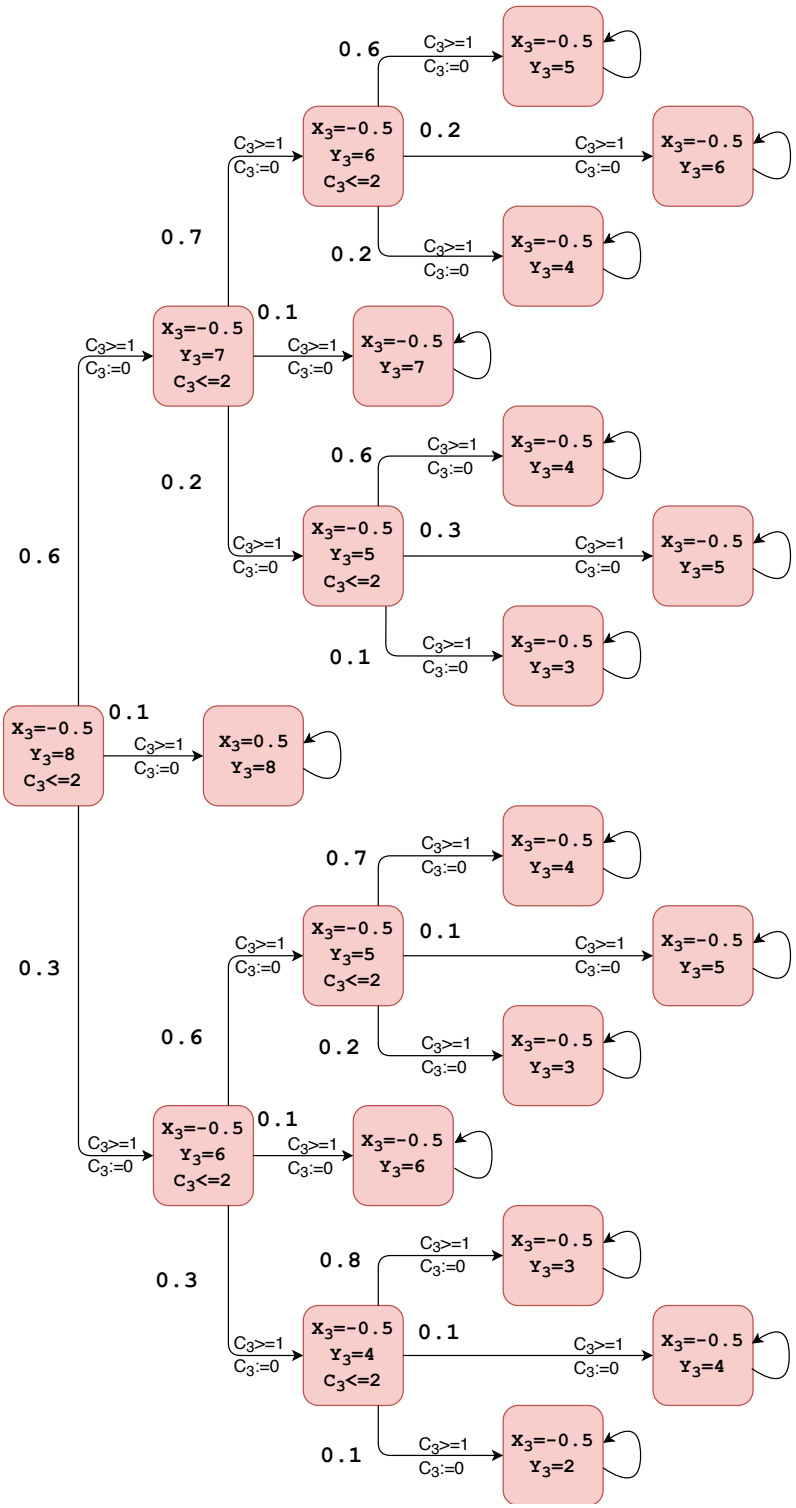


Figure 6.8: PTP model generated for the vehicle's behaviour.

away from the AV. This can be represented by the following expression:

$$\phi \equiv (x_1 - x_2)^2 + (y_1 - y_2)^2 >= 1. \tag{6.3}$$

$$\phi \equiv (x_1 - x_3)^2 + (y_1 - y_3)^2 >= 1. \tag{6.4}$$

Where (x_2, y_2), represent the coordinate of the pedestrian, (x_3, y_3) is the coordinate for the car. As PRISM cannot deal with real numbers, we multiple the distance by 2. We compute the maximum probability of the violation of Equation (6.3, 6.4), by the following PCTL property:

$$\mathtt{P}_{max=?}[\mathtt{F}\ \neg\phi]. \tag{6.5}$$

Due to the discretisation of the trajectory, the negation of Equation (6.3) is translated into the following expression:

$$\begin{aligned}(((x_2 > x_1 \wedge x_2 - x_1 \leq 1) \vee (x_1 > x_2 \wedge x_1 - x_2 \leq 1)) \wedge \\ ((y_2 > y_1 \wedge y_2 - y_1 \leq 1) \vee (y_1 > y_2 \wedge y_1 - y_2 \leq 1)))\end{aligned}$$

While the negation of Equation (6.4) is translated into:

$$\begin{aligned}(((x_3 > x_1 \wedge x_3 - x_1 \leq 1) \vee (x_1 > x_3 \wedge x_1 - x_3 \leq 1)) \wedge \\ ((y_3 > y1 \wedge y_3 - y_1 \leq 1) \vee (y_1 > y_3 \wedge y_1 - y_3 \leq 1)))\end{aligned}$$

The verification results for the proposed scenario are shown in table 6.3 returned from PRISM for formula (6.5), which indicates information about the model generated for both the pedestrian and the car and the probability of collision with every one of them under the current motion plan. Based on these data, the AV can choose whether to take a particular action or not. Basically, if the probability of the

Table 6.3: Verification results for the proposed scenario.

PTP model	States	Transitions	Choices	Ver. time	Maximum collision probability
Pedestrian	1238	3884	3624	0.036s	0.252
Car	659	2230	1960	0.019s	0.003

collision is high, then the RA should neglect the action and look for an alternative from the plan library. The last two columns are presenting the most important data regarding the time needed by PRISM to verify a particular PTP model. We designed the RA in a way that it can only generate light weight PTP models in order to get them verified within one reasoning cycle of the agent. Here the time is an important factor because all of the computations need to be done while the AV is in operation if more time is needed then PRISM will not be able to return the results within one reasoning cycle and this will result in RA hold its decision for taking a particular action and in the same time trying to slow down the speed of the AV if needed or even to stop. From all the tests we did, we can say that this case did not happen and the agent was always able to respond to the environmental situation in a timely manner. However, we have equipped the agent with such capability in case of future development, or more complex driving scenario needed that results in larger size and number of PTP models. The developer sets the maximum collision probability threshold, and in this case, we have set it to be (0.5) for demonstration purposes means that if the collision probability is higher than (0.5), then the RA will reject the proposed action and look for a safer alternative; otherwise, the RA will execute the proposed action. Even if the chosen action may lead to an accident, the rule-based system will try to prevent it by slowing or stopping the AV when necessary based on pre-defined driving rules. This, however, will produce less smooth driving experience.

All the computations in this case study carried out using two computers running on Ubuntu OS version 16.04, first is equipped with (Intel core i7 CPU, 16 GB of RAM and GTX 1070 GPU) for simulation, perception, planning and control systems

and the second with (Intel core i7, 16 GB of RAM and GTX 860 GPU) to run the agent code and the verification system. Both of the computers are using solid-state drives for faster handling of data with the main memory unit. With the hardware implementation of the vehicle platform, we also used the Nvidia Jetson TX2 and eight raspberry Pi devices in addition to the computers as mentioned above and as explained in chapter 5.

6.4 Conclusion

In this chapter, we have presented a detailed case study for the design-time and run-time verification of the rational agent. The RA is guiding the AV through a parking lot environment to look for a free parking space and park the AV. The MCMAS is only used during the development stage while the Prism is supporting the agent to choose suitable actions while the AV is moving. The AV and its environment have been designed in 3D virtual reality simulation with realistic physics-based dynamic models using the Gazebo simulator. The results of verification for both the MCMAS and the Prism model checkers are presented in this chapter.

Chapter 7

Conclusions and Future Research Directions

It is essential to make it clear to the reader that both the rational agent and the verification system are built from the ground to satisfy the requirement of our new AV system. We also wanted to make it clear that apart from the new and novel systems mentioned above, we also designed an AV system in simulation that could be easily connected to a real vehicle platform. This AV system is introduced based on the reliable open-source platform represented by the Robot Operating System (ROS). The developed platform will be available for other researchers in the future as open-source, which could be helpful to test different algorithms on a 3D physics-based simulated vehicle and environment rather than on a real platform which could be difficult for some to put hands on.

7.1 Conclusions

In this thesis, we present a novel method for the verification of decisions made by software agent onboard the AV as well as the design and implementation of the AV system (with Level-4 of *autonomy*) in simulation and hardware implementation. The AV is controlled by a RA through several skills necessary for autonomous

driving operation. We have focused on a design approach to verifiable decision-making process onboard the vehicle. The novel verification method comprises two stages of formal verification, which work separately for the ultimate goal of providing a safe driving experience. A simulation model of an AV in a parking lot was designed to validate the vehicle design and its sensors, also to check the quality and practical feasibility of our system. This was then followed by the implementation of the system on a real industrial vehicle (Tata Ace EV model).

In this chapter we present the summary notes and conclusions of each chapter presented in this thesis, followed by the future work section to give some possible ideas about how this system could be further developed or enhanced in the future.

In chapter 1

This chapter contains a quick overview on the history of autonomous systems and autonomous vehicles, including initiatives in self-driving vehicles during the DARPA challenges, followed by efforts from both academia and industry to take this further to a whole new level. A quick overview of our design of the vehicle system was presented with a focus on the safe operation of the system in its environment. Then the main challenges were presented on how to design such complicated control systems. The chapter concluded with a list of our contributions to the field that we hope will be useful to others with a list of publications during the four years period of this project.

In chapter 2

A background about most of the used methods and techniques are presented through this chapter. Different levels of autonomy have been defined in detail, including the differences between the primary three levels of technologies used in cars: safe driving, self-driving, and driverless. Then the methods and techniques for designing a rational agent, followed by the role of verification in control systems, then the two model checking techniques used in this work were presented. The chapter ends with

an overview of the internal structure of the ROS platform and Gazebo simulator and a semi-formal definition of the ROS graph. In the conclusion section, we presented an overall system diagram that shows how the different system components connect and interact together.

In chapter 3

This is the first contribution chapter. A new approach is presented for the verification of an agent-based decision-making system for a self-driving vehicle. The approach considers both design-time and run-time verification where this approach is introduced here for the first time for the verification of decisions made by a rational agent for a self-driving vehicle. The proposed system is also validated through simulation and tests to prevent the developer from accidentally equipped the vehicle with unsafe or unstable rules or actions, also to prevent the vehicle from choosing unsafe action while driving. A solution is introduced for safe agent design and safe navigation in a dynamic environment based on formal verification. The critical point addressed in this work is to guarantee the safety for both the AV and the other agents moving in the same environment, which are vehicles and pedestrians in our case study. While the AV is exploring the parking lot environment looking for a free parking space, it needs to move safely and purposefully. The problem of navigation in a dynamic environment is harder than the one in static or controlled environments, due to several problems that can be summarised as: (i) the need for the detection and tracking of moving objects, (ii) the prediction of their future state in the environment, (iii) and the run-time planning and navigation.

It has been clarified that AV decisions in its environment must take into consideration the limitation of the vehicle perception system, including the uncertain future trajectory and velocity of the moving objects. The run-time decision-making proposed is based on a probabilistic framework to represent the uncertainty in the environment.

The agent code has been written in sEnglish software using natural language

programming. The code structure allows to include probabilistic patterns of the other agents in the environment. This permits to automatically generate probabilistic models of those agents for formal verification. However, this still means that those probabilistic patterns that describe the environmental events should be defined by the developer to allow the run-time verification work appropriately.

A rational agent in a real traffic scenario usually faces a vast amount of situations and related behaviour rules. Many of these can be identified during the design stage. Remaining scenarios, with a possible probabilistic event in the environment, can then be handled by machine-learning-based evaluations.

Chapter 3 covers two important gaps, and those are:

1. Design a simple yet efficient software agent to control and drive the vehicle in a restricted environment.

2. Develop new methods to check the safety and feasibility of the decision-making process of the self-driving vehicle.

We would like to mention again that in this chapter we answered the first research question of 'How we can ensure the safety and feasibility of decisions made by the autonomous vehicle while driving by using the formal verification as the main approach?'.

In chapter 4

We started this chapter with a simple yet useful demonstration in Matlab/Simulink of an AV model to clarify some important aspects of the AV design that need to be known by the reader before going through the design in more details. We went through the implementation of this system with some necessary information that a development engineer would need to understand and be familiar with. Presented a simple self-parking operation to test the dynamics of the vehicle and to study the different aspects of the parking operation in a parking lot that we modelled through Simulink controls.

In the second part of this chapter, we have presented a method to design an AV using ROS, which is compatible with Virtual Reality (VR) Gazebo simulator. ROS is known for its high flexibility, efficiency, and capabilities for robotic systems design, which could be connected either to a simulator or to a physical platform.

There is a lack of such customisable and easy to use AV platform for researchers which could be used to test their perception, planning, and control algorithms. Hence this is a contribution in the form of a new open-source implementation code (of the Tata Ace vehicle) that will be publicly available for free for the interested researchers in the field to use and develop further. The current rare publicly available open-sources ROS-based vehicle platforms mostly belong to small size robots.

The car manufacturers are developing their own closed-source platforms as opposed to open-source platforms for designing self-driving cars means only that company has access to their software. In contrast, our implementation is open-source that will be available with easy access and well documented for all the aspects of design and implementation that could also be easily connected to a physical platform as discussed in chapter 5 for hardware implementation research purposes.

Chapter 4 cover an important gap in the research field which is to provide an open-source based reconfigurable autonomous vehicle system that supports hardware in the loop by engaging a real vehicle platform, which could be used to validate the design and test different related algorithms.

In this chapter, we answered the second research question of 'How we can design a simple, feasible, realistic and reconfigurable autonomous vehicle system using the robot operating system?'.

In chapter 5

The work demonstrated in this chapter is to fulfil our industrial partner needs to demonstrate that the system presented in chapter 4 is capable of handling the autonomous driving operation based on the set of sensors suggested by Tata motors. Hence this work could be considered as an engineering or technical contribution to

support some requirements of the industry.

The model of the AV developed in the previous chapter can be easily connected to a physical platform (Experimental vehicle) through the CAN bus system. This chapter demonstrates this connection with a testbed platform of a Tata Ace EV provided by the Tata Motors European Technical Centre (TMETC). All the hardware used has been described to implement the perception and control systems. Implementation of all the methods and algorithms are included in chapter 5 for the perception system with associated technical data of sensors.

It is essential to mention that ROS proved to be capable of handling such complicated robotic system with nearly real-time control of the physical platform. The ROS framework is easy to implement in any modern programming language, including C++ and Python. A ROS system is made up of a series of independent nodes which communicate with each other using a publish/subscribe messaging model. We can use different computers with different operating systems, even different architectures in our vehicle system. However, the first generation of ROS used in this projects still has its weak points such as it is not an actual real-time system, and it suffers from security-related issues that have been solved in the second generation of ROS.

In this chapter, we answered the third research question of 'How to use the ROS-based autonomous vehicle system to drive a real vehicle in a real-life driving scenario in a parking lot environment?'.

In chapter 6

Our verification system and AV simulation are presented through a case study in this chapter. The AV is moving among different objects in a simulation environment, in which a parking lot is being considered as a scenario where the software agent needs to guide the AV from the entrance to a free parking bay safely.

The power of the combination of the two verification tools helps the designer to eliminate any redundancy in the agent predicates. These tools also check the agent

rules for any possible instability or inconsistency in the system. A counterexample is generated when a faulty state is reached. In the run-time operation, verification deals with possible predicted movements of the traffic participants to determine the probability of success for the intended AV action. A limited set of beliefs, rules and actions have been presented as a proof of concept. The perception system will trigger different beliefs while the AV is moving, and the agent will try to satisfy its desires and intentions until reaching its goal.

7.2 Future work

The autonomous system presented in this thesis shows great potentials for the development of self-driving vehicles, with a focus on safety. In future, however, some points could improve reliability and productivity, as summarised in the following points:

1. For higher-levels of rationality, the agent could yet be equipped, during design-time, with a methodology for rules and other predicates generation. Such a system would be able to learn new driving scenarios for run-time verification by implementing a machine learning approach.

2. The real-time verification system presented in this thesis is only implemented on the vehicle in simulation; this could be implemented on the physical platform later with only minor changes.

3. Despite that the navigation system presented can work well with small size parking lots, it would struggle with bigger ones as we did not implement an exploration system. The system implemented has general planning (motion planning and path planning) system that deliberates information with the agent for path generation. For a more complex scenario, it would be necessary to implement an exploration algorithm in ROS that could be connected to the rational agent in a similar way.

4. The camera perception system implemented on the experimental vehicle consists of two parts: first, a single stereo camera which is connected to a deep-learning algorithm to recognise people and vehicles in the area in front of the AV, and the second part is using multiple Raspberry Pi cameras with an Aggregated Channel Features (ACFs) object detector. This latter has limited accuracy compared to the first method, but it is also less expensive and easier to implement. With considerable resources, we would propose to use the first method on all the sides of the AV for higher accuracy and to provide a redundant source of accurate depth information, which will support the depth data provided by the LiDAR.

5. In all the driving environments, the vehicle needs a method to be aware of the existence, position, and velocity, of the moving objects in order to predict their movements. The medium-term prediction method used in this project can give a decent prediction period. However, it only works in a particular environment and cannot be easily generalised to other ones. However, it is not always the case that the objects follow a pre-learned pattern during movement. Hence we would suggest a fusion of both short-term predictions represented by target tracking and medium-term prediction based on saved patterns. The new approach would need either the modification of the agent architecture to add a subsystem for running such prediction algorithms or to connect the agent with an external skill algorithm that can produce such knowledge for the agent within a short time.

6. The last point, we would suggest here for future work, could be implemented in smart parking lots through communication with an off-board perception system place into the parking lot. In case of such *smart* parking lot, there would be a sensor to check the occupancy of every parking space. This vital information could be delivered to oncoming vehicles entering, to inform them about the position of the free parking spaces and the best trajectory to take to reach them, also about the nearest exits, when needed. This would help to reduce the vehicle search time and overall congestion inside the parking lot.

Appendices

Appendix A

Agent code for the presented case study

```
INITIAL BELIEFS AND GOALS

System is operational.
~AV parked.
~Slow mode.
~Normal mode.
~Reverse mode.
Stopped mode.
~Mission in progress.
~Parking space located.
~Space search started.
~Started parking operation.
Parking requested.
~Commencing parking operation.
~AV in parking space.
~Pedestrian detected.
~Moving vehicle detected.
~Continue parking.
~Abort parking.
Need to explore parking lot.
~Parking space not available.
~Parking lot fully explored.
~Parking lot not fully explored.
~Collision occured.
~Send alert to owner.


INITIAL ACTIONS

Read connected sensors sensor_configuration from file 'config.
    knb'.
+!take_initial_actions <-
```

```
32 invoke(perception_subsystem,runRepeated,
       initial_parameter_for_perception,[],[""]);
33 invoke(control_subsystem,runOnce,initial_parameter_for_agent,[
       ],["Force"]);
34 invoke(planning_subsystem,runRepeated,plan_trajectory,["
       AV_position","AV_heading","Target_position","
       Finaltarget_direction","Option"],["PathPoints"]);
35 invoke(guidance_subsystem,runRepeated,apply_guidance_control,[
       "P","V","Option"],["Input"]);

37 PERCEPTION PROCESSES

39 Monitor the following Booleans :
40 +!configureSystem:true <- linkSystems(perception_subsystem,
       planning_subsystem,guidance_subsystem,triggering_subsystem)
       ;
41 !take_initial_actions.



45 UPDATING PERCEPTION

47 Monitor the following objects :
48 Update the environmental model Eb from sensors
       sensor_configuration.

50 +!linkSystems(complete):true <- checkPercepts.
51 !checkPercepts <-
52 updateSystems(perception_subsystem,control_subsystem,
       planning_subsystem,guidance_subsystem,triggering_subsystem)
       ;
53 !checkPercepts.


56 REASONING

58 If ^[Configure system] and ~^[Mission in progress] and ~^[
       Parking lot fully explored] then ^[Start mission]
59 If ~^[Pedestrian detected] or ~^[Moving vehicle detected] or ~
       ^[Collision occured] or ~^[Parking lot fully explored] then
        ^[Forward mode]
60 If ^[Collision occured] then ^[Send alert to owner]
61 If ^[Parking space not available] then ^[Abort parking]


64 EXECUTABLE PLANS
65 +configureSystem:true <- linkSystems(perception_subsystem,
       planning_subsystem,guidance_subsystem,triggering_subsystem)
       ;
66 invoke(perception_subsystem,runRepeated,
       initial_parameter_for_perception,[],[""]);
67 invoke(control_subsystem,runOnce,initial_parameter_for_agent,[
       ],["Force"]);
68 invoke(planning_subsystem,runRepeated,plan_trajectory,["
       AV_position","AV_heading","Target_position","
```

```
    Finaltarget_direction","Option"],["PathPoints"]);
invoke(guidance_subsystem,runRepeated,apply_guidance_control,[
    "P","V","Option"],["Input"]);


//Plan 1
If ^~[AV parked] and ^[Parking requested] then
+^[Need to explore parking lot.].

//Plan 2
If ^[Need to explore parking lot] and ^~[Space search started]
     then
[Generate exploration waypoints wp_exp.]
[Update drive mode with waypoints wp_exp.]
+^[Normal mode.]
+^[Space search started.].

//Plan 3
If ^[Parking space located at Position P_ in environmental
    model Eb] then
-^[Need to explore parking lot.]
+^[Commencing parking operation.].

//Plan 4
If ^[Commencing parking operation] and ^~[Started parking
    operation] then
[Generate parking waypoints wp_park for space at position P_
    in environmental model Eb.]
[Update drive mode md with waypoints wp_park.]
+^[Started parking operation.].

//Plan 5
If ^[pedestrian p_ detected in environmental model Eb at
    distance d_] and
^[pedestrian p_ at distance d_ more than 3m and less than 6m
    in environmental model Eb] and
^[pedestrian p_ getting closer] and ^[Parking requested] then
    [Activate slow mode.]
+^~[Normal mode.]
+^[Slow mode.]
[Generate object avoidance waypoints wp_avoid.]
[Update drive mode md with waypoints wp_avoid.].

//Plan 6
If ^[pedestrian p_ detected in environmental model Eb] and
^[pedestrian p_ at distance d_ less than 3m in environmental
    model Eb] and ^[Parking requested] then [Activate stop mode
    .]
+^~[Slow mode.]
+^[Stopped mode.]
[Update drive mode md.].

//Plan 7
If ^[moving vehicle mv detected in environmental model Eb] and
^[Parking requested] while
```

```
113 ^distance d_ to moving vehicle mv more than 3m and less than 6
       m in environmental model Eb] and
114 ^[moving vehicle mv getting closer] then
115 +^[Slow mode.]
116 +^~[Normal mode.]
117 [Activate slow mode.]
118 [Generate object avoidance waypoints wp_avoid.]
119 [Update drive mode md with waypoints wp_avoid.].

121 //Plan 8
122 If ^[moving vehicle mv detected in environmental model Eb] and
123 ^[Parking requested] while
124 ^[distance d_ to moving vehicle mv less than 3m] then [
       Activate stop mode.]
125 +^~[Slow mode.]
126 +~[Stopped mode.]
127 [Update drive mode md.].

129 //Plan 9
130 If ^[AV in parking space] and ^[Parking requested] then [
       Activate stop mode.]
131 +^~[Slow mode.]
132 +^~[Normal mode.]
133 +^[Stopped mode.]
134 [Update drive mode md.]
135 +[AV parked.]
136 +~[Parking requested.].
```

Appendix B

Tata Ace vehicle / operational modes

It is important to mention that this is not a standalone section, it is complemented by both the Topcon AES-25 user manual [8], and the Curtis 1236E user manual [250] and their related programming code.

Curtis 1236E AC induction motor controller provides unprecedented flexibility and power through the inclusion of a field-programmable logic controller embedded in a motor controller. The embedded logic controller runs a fully functional field-oriented AC motor control Operating System (OS) that can be user-tailored via parameter modification. The OS also contains logic to execute Original Equipment Manufacturer (OEM) developed software, called Vehicle Control Language (VCL), that can be used to enhance the controller capabilities beyond the basics. VCL is a software programming language developed by Curtis. Many electric vehicle functions are uniquely built into the VCL code, and additional functions can be OEM-controlled using VCL code. The CAN bus communications included in the 1236E, allow these AC induction motor controllers to be part of an efficient distributed system. Inputs and outputs can be optimally shared throughout the system, minimising the wiring and creating integrated functions that often reduce the cost of the system. For further information, please have a look at the Curtis

1236E user manual [250].

The testbed vehicle could be used in one of the following two modes:

1. **Manual**
    - For driving the vehicle manually.
    - 1236E controls drive in response to driver inputs.
    - Only the 1236E controller and manual controls are required.
2. **Autonomous**
    - For testing autonomous steering and/or driving functions under human driver supervision.
    - Vehicle may operate with both steering and drive under autonomous control or with only one of theses functions under autonomous control and the other under human driver control.
    - Auto Master Controller (AMC) controls steering and/or drive demand(s).
    - 1236E controls drive.
    - Topcon steering controller actuates steering (when required).

Safety "Interlocks" provided to:

- Prevent unintended motion.
- Ensure human supervisor is present (when driving autonomously).
- Disable autonomous controls upon human driver intervention.

Throttle Start Switch / Interlock: used as "dead-man's handle" to enable/disable drive (and steering actuation during autonomous driving). The human driver has to press the throttle pedal far enough for the switch to close, to enable the drive (and steering) – removing the driver foot from the pedal, in order to brake, will disable the drive (and steering) actuation.

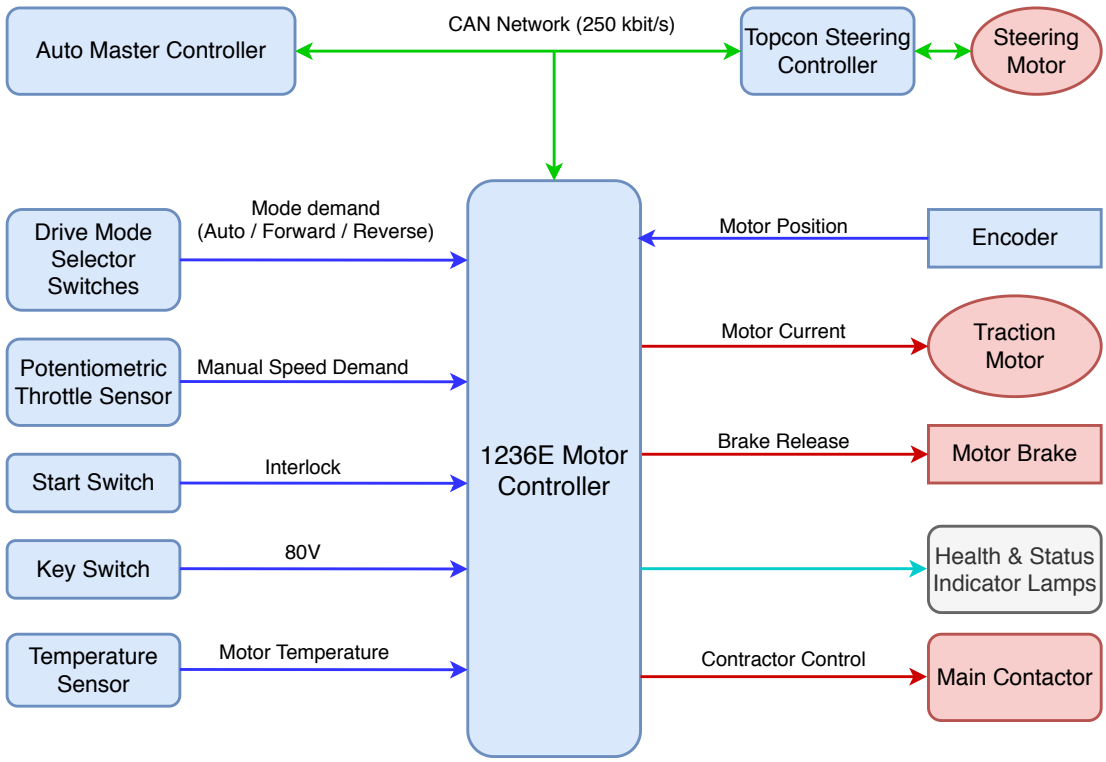


Figure B.1: Schematic of the 1236E related vehicle controls.

Emergency switch: the vehicle has been fitted with red, latching Emergency Stop button on dashboard used to disable traction drive and steering actuation from both inside and outside the vehicle. The schematic diagram for connecting the Curtis 1236E with the other system components is shown in Fig. B.1, while the Topcon system connection diagram is shown in Fig. B.2.

In Manual Mode

Drive mode: manual forward or manual reverse. Systems operation is as below:

- Direction control
  - Vehicle driving direction is set by the driver using the manual drive mode selector switch. Reverse (Sw_8) or Forward (Sw_7) may be selected.
- Speed control
  - Vehicle speed demand is controlled by the driver using a potentiometric

throttle input to the Curtis 1236E: VCL_Throttle = Throttle_Pot.

- The 1236E interprets the VCL_Throttle input as a speed demand.
- Max_Speed parameter is set according to drive direction: 4600 rpm forward and 1000 rpm reverse.

- Steering control
  - Steering Angle is manually controlled by the driver using the steering wheel.
  - Topcon Steering Actuator is disabled, but the index position acquisition is possible.

- EM brake control
  - EM brake is controlled by the 1236E in response to Drive Mode Selection, Start Switch (Interlock) and Speed: It will be released when a non-neutral state is selected, and the throttle pedal is pressed. It will be engaged when the throttle is released and speed falls below the threshold value or Neutral is selected. The EM brake will also be engaged when 1236E controller faults are detected.

In Autonomous Mode

Drive mode: with drive mode set to Autonomous, the Systems operation is as below with three sub-modes complete: both autonomous, only steering autonomous, and only vehicle speed autonomous. In our work, the AV was always working in complete Autonomous, means both steering and vehicle speed are master controlled. This mode applies the following:

- Direction control
  - Vehicle driving direction is set by the Auto Master Controller.

- Speed control

  - Motor Speed and Direction demand is set by the Master Controller and sent to the Curtis 1236E via CAN.
  - AMC enables/disables Speed control in response to 1236E Interlock (Throttle Start Switch input) signal received via CAN.
  - Max_Speed parameter is set according to drive direction: 3000 rpm forward and 500 rpm reverse.

- Steering control
  - Steering Actuation is supervised and enabled/disabled by AMC.
  - AMC enables/disables Steering Actuation in response to 1236E Interlock (Throttle Start Switch input) signal received via CAN.
  - Topcon AES Steering Actuator controls Steering Angle in response to AMC Steering Angle Demand.

- EM brake control
  - The EM brake demand is set by the Master Controller and sent to the 1236E via CAN.

When operational the 1236E transmits the following data (to the AMC):

- Motor Speed
- Battery Voltage
- Motor Torque
- Motor Current
- Contactor Status
- EM Brake Status
- Throttle Sensor Position
- Throttle Start Switch Status

- Drive Mode Status
- Emergency Rev Status
- 1236E Fault Status

The 1236E receives the following data (from the AMC):

- Motor Direction Demand (Forward/Reverse/Neutral)
- Auto Speed Enable Demand
- Motor Speed Demand
- EM Brake Demand

Manual Mode operation shall be possible when the 1236E is the only node connected to the CAN bus. The AMC should transmit the following data to the Topcon AES-25 controller:

- Max Steering Speed Limit
- Steering Angle Demand
- Steering Controller State Command (CAN Comms Enable, Steering Motor Enable, Precharge Enable, Steering Contactor Close, Alarm)
- Max Steering Torque Limit

The AMC should receive the following data from the AES-25 controller:

- Steering Motor Speed
- Steering Motor Torque
- Encoder Steering Angle
- Steering Controller Status

- Steering Controller Alarm
- Steering Motor Temperature
- Steering Inverter Temperature
- Steering Inverter Voltage
- Potentiometer Steering Angle

CAN Watchdog Behaviour

When Auto Speed Mode is selected:

- 1236E shall implement a CAN watchdog timer function.
- The function shall countdown the interval between correct reception of CAN messages from the AMC.
- In the event that the specified interval length is exceeded the 1236E shall indicate a fault, apply the EM brake and open the main contactor.
- Reset from this condition shall require a power off and back on of the 1236E.

When Auto Steer Mode is selected:

- Topcon AES controller shall implement a CAN watchdog timer function.
- The function shall countdown the interval between correct reception of CAN messages from the AMC.
- In the event that the specified interval length is exceeded the Topcon AES shall indicate a fault, disable steering torque and open the steering contactor.
- Reset from this condition shall require a power off and back on of the 1236E.

When Auto Steer Mode is selected:

- AMC shall implement a CAN watchdog timer function.
- The function shall countdown the interval between correct reception of CAN messages from the Topcon AES.
- In the event that the specified interval length is exceeded the AMC shall indicate a fault, demand zero speed and command EM brake application.
- Reset from this condition shall require user input.

When Auto Speed Mode is selected:

- AMC shall implement a CAN watchdog timer function.
- The function shall countdown the interval between correct reception of CAN messages from the 1236E.
- In the event that the specified interval length is exceeded the AMC shall indicate a fault, demand zero speed and command EM brake application.
- Reset from this condition shall require user input.

Figure B.2: Topcon AES-25 basic system connection diagram [8].

References

[1] S. M. Veres, N. K. Lincoln, and L. Molnar, "Control engineering of autonomous cognitive vehicles-a practical tutorial," *Cognitive Rational Agents*, pp. 1–31, 2011.

[2] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[3] P. Izzo, H. Qu, and S. M. Veres, "Reducing complexity of autonomous control agents for verifiability," *arXiv preprint arXiv:1603.01202*, 2016.

[4] P. Izzo, H. Qu, and S. M. Veres, "A stochastically verifiable autonomous control architecture with reasoning," in *55th IEEE Conference on Decision and Control, CDC'16*, pp. 4985–4991, IEEE, 2016.

[5] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

[6] P. Dollár, R. Appel, S. Belongie, and P. Perona, "Fast feature pyramids for object detection," *IEEE transactions on pattern analysis and machine intelligence*, vol. 36, no. 8, pp. 1532–1545, 2014.

[7] J. Zhang and S. Singh, "LOAM: Lidar Odometry and Mapping in Real-time," in *Robotics: Science and Systems*, vol. 2, p. 9, 2014.

[8] Topcon positioning, "Interface Design Guide/Specification AES-25 user manual." `https://www.topconpositioning.com/`, 2010. [Accessed:02/03/2020].

[9] SAE International, "Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles." `https://saemobilus.sae.org/content/J3016_201806/`, 2018. [Accessed: 15/05/2019].

[10] STEREOLABS, "ZED Camera and SDK Overview." `http://www.stereolabs.com/assets/datasheets/zed-camera-datasheet.pdf`, 2019. [Accessed: 03/02/2020].

[11] Nvidia, "Jetson TX2 Module." `https://developer.nvidia.com/embedded/jetson-tx2`, 2017. [Accessed: 13/05/2018].

[12] RaspberryPi, "Raspberry Pi 3 Model B+." `https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/`, 2016. [Accessed:20/04/2018].

[13] Association for safe international road travel, "Road safety facts." `https://www.asirt.org/safe-travel/road-safety-facts/`, 2019. [Accessed: 28/01/2020].

[14] S. Singh, "Critical reasons for crashes investigated in the national motor vehicle crash causation survey," tech. rep., National Highway Traffic Safety Administration, 2015.

[15] M. Buehler, K. Iagnemma, and S. Singh, *The 2005 DARPA grand challenge: the great robot race*, vol. 36. Springer, 2007.

[16] M. Buehler, K. Iagnemma, and S. Singh, *The DARPA urban challenge: au-*

tonomous vehicles in city traffic, vol. 56. springer, 2009.

[17] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, *et al.*, "Autonomous driving in urban environments: Boss and the urban challenge," *Journal of Field Robotics*, vol. 25, no. 8, pp. 425–466, 2008.

[18] J. Leonard, J. How, S. Teller, M. Berger, S. Campbell, G. Fiore, L. Fletcher, E. Frazzoli, A. Huang, S. Karaman, *et al.*, "A perception-driven autonomous urban vehicle," *Journal of Field Robotics*, vol. 25, no. 10, pp. 727–774, 2008.

[19] P. Furgale, U. Schwesinger, M. Rufli, W. Derendarz, H. Grimmett, P. Mühlfellner, S. Wonneberger, J. Timpner, S. Rottmann, B. Li, *et al.*, "Toward automated driving in cities using close-to-market sensors: An overview of the v-charge project," in *IEEE Intelligent Vehicles Symposium*, pp. 809–816, IEEE, 2013.

[20] C.-Y. Chan, "Advancements, prospects, and impacts of automated driving systems," *International journal of transportation science and technology*, vol. 6, no. 3, pp. 208–216, 2017.

[21] W. Schwarting, J. Alonso-Mora, and D. Rus, "Planning and decision-making for autonomous vehicles," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 1, pp. 187–210, 2018.

[22] Wikipedia, "List of self-driving car fatalities." `https://en.wikipedia.org/wiki/List_of_self-driving_car_fatalities`, 2019. [Accessed: 18/08/2019].

[23] Eric Vega, "Every single self-driving car accident and death." `https://www.ranker.com/list/self-driving-car-accidents/eric-vega`, 2018. [Accessed: 18/08/2019].

[24] Lulu Chang and Luke Dormehl, "6 self-driving car crashes that tapped the brakes on the autonomous revolution." `https://www.digitaltrends.com/cool-tech/most-significant-self-driving-car-crashes/`, 2018. [Accessed: 18/08/2019].

[25] A. J. Hawkins, "A day in the life of a Waymo self-driving taxi." `https://www.theverge.com/2018/8/21/17762326/waymo-self-driving-ride-hail-fleet-management`, 2018. [Accessed: 14/02/2020].

[26] A. Herrmann, W. Brenner, and R. Stadler, *Autonomous driving: how the driverless revolution will change the world*. Emerald Group Publishing, 2018.

[27] A. S. Rao and M. P. Georgeff, "Modeling rational agents within a BDI-architecture," *KR*, vol. 91, pp. 473–484, 1991.

[28] X. Huang and M. Z. Kwiatkowska, "Reasoning about cognitive trust in stochastic multiagent systems," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[29] M. Fisher, L. Dennis, and M. Webster, "Verifying autonomous systems,"

Communications of the ACM, vol. 56, no. 9, pp. 84–93, 2013.

[30] L. A. Dennis, M. Fisher, N. K. Lincoln, A. Lisitsa, and S. M. Veres, "Practical verification of decision-making in agent-based autonomous systems," *Automated Software Engineering*, vol. 23, no. 3, pp. 305–359, 2016.

[31] R. Hoffmann, M. Ireland, A. Miller, G. Norman, and S. Veres, "Autonomous agent behaviour modelled in prism–a case study," in *International Symposium on Model Checking Software*, pp. 104–110, Springer, 2016.

[32] M. Al-Nuaimi, H. Qu, and S. M. Veres, "Computational framework for verifiable decisions of self-driving vehicles," in *IEEE Conference on Control Technology and Applications*, pp. 638–645, IEEE, 2018.

[33] SAE Society of Automotive Engineering International, "SAE On-Road Automated Vehicle Standards Committee and others," *Taxonomy and Definitions for terms Related to On-Road Motor Vehicle Automated Driving Systems*, pp. 1–16, 2014.

[34] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, vol. 3, p. 5, 2009.

[35] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 3, pp. 2149–2154, IEEE, 2004.

[36] A. Lomuscio, H. Qu, and F. Raimondi, "MCMAS: A model checker for the verification of multi-agent systems," in *International Conference on Computer Aided Verification*, vol. 5643, pp. 682–688, Springer, 2009.

[37] M. Kwiatkowska, G. Norman, and D. Parker, "Prism 4.0: Verification of probabilistic real-time systems," in *Computer aided verification*, pp. 585–591, Springer, 2011.

[38] N. Lincoln, S. M. Veres, L. A. Dennis, M. Fisher, and A. Lisitsa, "An agent based framework for adaptive control and decision making of autonomous vehicles.," in *ALCOSP*, pp. 310–317, 2010.

[39] SAE International, "Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles." `https://saemobilus.sae.org/content/j3016_201609`, 2016. [Accessed: 15/05/2019].

[40] F. Zhang, D. Clarke, and A. Knoll, "Vehicle detection based on lidar and camera fusion," in *17th International IEEE Conference on Intelligent Transportation Systems*, pp. 1620–1625, IEEE, 2014.

[41] B. Okumura, M. R. James, Y. Kanzawa, M. Derry, K. Sakai, T. Nishi, and D. Prokhorov, "Challenges in perception and decision making for intelligent automotive vehicles: A case study," *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 1, pp. 20–32, 2016.

[42] S. M. Veres, L. Molnar, N. K. Lincoln, and C. P. Morice, "Autonomous vehicle

control systems—a review of decision making," *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, vol. 225, no. 2, pp. 155–195, 2011.

[43] P. J. Antsaklis, K. M. Passino, and S. Wang, "An introduction to autonomous control systems," *IEEE Control Systems Magazine*, vol. 11, no. 4, pp. 5–13, 1991.

[44] P. J. Antsaklis, K. M. Passino, and S. Wang, "Towards intelligent autonomous control systems: Architecture and fundamental issues," *Journal of Intelligent and Robotic Systems*, vol. 1, no. 4, pp. 315–342, 1989.

[45] N. B. Geddes, "Magic motorways." `https://archive.org/stream/magicmotorways00geddrich?ref=ol#mode/2up`, 1940. [Accessed: 01/08/2019].

[46] P. Lamon, S. Kolski, and R. Siegwart, "The SmartTer-a vehicle for fully autonomous navigation and mapping in outdoor environments," in *Proceedings of CLAWAR*, 2006.

[47] Z. Chong, B. Qin, T. Bandyopadhyay, T. Wongpiromsarn, E. Rankin, M. Ang, E. Frazzoli, D. Rus, D. Hsu, and K. Low, "Autonomous personal vehicle for the first-and last-mile transportation services," in *IEEE 5th International Conference on Cybernetics and Intelligent Systems*, pp. 253–260, IEEE, 2011.

[48] A. Broggi, M. Buzzoni, S. Debattisti, P. Grisleri, M. C. Laghi, P. Medici, and P. Versari, "Extensive tests of autonomous driving technologies," *IEEE Transactions on Intelligent Transportation Systems*, vol. 14, no. 3, pp. 1403–1415, 2013.

[49] U.S. Department of Transportation, "Preparing for the future of transportation: Automated vehicles 3.0." `https://www.nhtsa.gov/press-releases/us-department/transportation-releases-preparing-future-transportation-automated`, 2013. [Accessed: 03/07/2019].

[50] SAE International, "Taxonomy and definitions for terms related to on-road motor vehicle automated driving systems." `https://saemobilus.sae.org/content/j3016_201401`, 2014. [Accessed: 15/05/2019].

[51] R. E. Parasuraman and M. E. Mouloua, *Automation and human performance: Theory and applications.* Lawrence Erlbaum Associates, Inc, 1996.

[52] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt, *et al.*, "Towards fully autonomous driving: Systems and algorithms," in *IEEE Intelligent Vehicles Symposium*, pp. 163–168, IEEE, 2011.

[53] A. Costley, C. Kunz, R. Gerdes, and R. Sharma, "Low cost, open-source testbed to enable full-sized automated vehicle research," *arXiv preprint arXiv:1708.07771*, 2017.

[54] J. C. McCall, O. Achler, and M. M. Trivedi, "Design of an instrumented

vehicle test bed for developing a human centered driver support system," in *IEEE Intelligent Vehicles Symposium, 2004*, pp. 483–488, IEEE, 2004.

[55] R. K. Bhadani, J. Sprinkle, and M. Bunting, "The cat vehicle testbed: A simulator with hardware in the loop for autonomous vehicle applications," *Electronic Proceedings in Theoretical Computer Science, EPTCS*, vol. 269, pp. 32–47, 2018.

[56] C. Badue, R. Guidolini, R. V. Carneiro, P. Azevedo, V. B. Cardoso, A. Forechi, L. Jesus, R. Berriel, T. Paixão, F. Mutz, *et al.*, "Self-driving cars: A survey," *arXiv preprint arXiv:1901.04407*, 2019.

[57] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[58] M. Bojarski, P. Yeres, A. Choromanska, K. Choromanski, B. Firner, L. Jackel, and U. Muller, "Explaining how a deep neural network trained with end-to-end learning steers a car," *arXiv preprint arXiv:1704.07911*, 2017.

[59] M. Wooldridge, *An introduction to multiagent systems*. John Wiley & Sons, 2009.

[60] M. Wooldridge, *Reasoning about rational agents*. MIT press, 2003.

[61] A. S. Rao and M. P. Georgeff, "An abstract architecture for rational agents," in *Principles of Knowledge Representation and Reasoning*, pp. 439–449, 1992.

[62] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[63] E. Coste-Maniere and R. Simmons, "Architecture, the backbone of robotic systems," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings*, vol. 1, pp. 67–72, IEEE, 2000.

[64] S. M. Veres, "Principles, architectures and trends in autonomous control," *IEE Seminar on Autonomous Agents in Control*, 2005.

[65] P. E. Agre and D. Chapman, "Pengi: An implementation of a theory of activity," in *AAAI*, vol. 87, pp. 286–272, 1987.

[66] M. Fisher, "A survey of concurrent metatem—the language and its applications," in *International Conference on Temporal Logic*, pp. 480–505, Springer, 1994.

[67] Y. Lespérance, H. J. Levesque, F. Lin, D. Marcu, R. Reiter, and R. B. Scherl, "Foundations of a logical approach to agent programming," in *International Workshop on Agent Theories, Architectures, and Languages*, pp. 331–346, Springer, 1995.

[68] S. J. Rosenschein and L. P. Kaelbling, "A situated view of representation and control," *Artificial intelligence*, vol. 73, no. 1-2, pp. 149–173, 1995.

[69] R. C. Arkin, R. C. Arkin, *et al.*, *Behavior-based robotics*. MIT press, 1998.

[70] J. Rosenblatt, S. Williams, and H. Durrant-Whyte, "A behavior-based architecture for autonomous underwater exploration," *Information Sciences*, vol. 145, no. 1-2, pp. 69–87, 2002.

[71] S. K. Das and A. A. Reyes, "An approach to integrating HLA federations and genetic algorithms to support automatic design evaluation for multi-agent systems," *Simulation Practice and Theory*, vol. 9, no. 3-5, pp. 167–192, 2002.

[72] A. Ferrein and G. Lakemeyer, "Logic-based robot control in highly dynamic domains," *Robotics and Autonomous Systems*, vol. 56, no. 11, pp. 980–991, 2008.

[73] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl, "GOLOG: A logic programming language for dynamic domains," *The Journal of Logic Programming*, vol. 31, no. 1-3, pp. 59–83, 1997.

[74] M. Georgeff, B. Pell, M. Pollack, M. Tambe, and M. Wooldridge, "The belief-desire-intention model of agency," in *International workshop on agent theories, architectures, and languages*, pp. 1–10, Springer, 1998.

[75] A. S. Rao, M. P. Georgeff, *et al.*, "BDI agents: from theory to practice," in *ICMAS*, vol. 95, pp. 312–319, 1995.

[76] A. S. Rao, "Decision procedures for prepositional linear-time belief-desire-intention logics," in *International Workshop on Agent Theories, Architectures, and Languages*, pp. 33–48, Springer, 1995.

[77] E. Gat, R. P. Bonnasso, R. Murphy, *et al.*, "On three-layer architectures," *Artificial intelligence and mobile robots: Case Studies of Successful Robot Systems*, vol. 195, p. 210, 1998.

[78] J. P. Müller, M. Pischel, and M. Thiel, "Modeling reactive behaviour in vertically layered agent architectures," in *International Workshop on Agent Theories, Architectures, and Languages*, pp. 261–276, Springer, 1994.

[79] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, "An architecture for autonomy," *The International Journal of Robotics Research*, vol. 17, no. 4, pp. 315–337, 1998.

[80] R. Simmons and D. Apfelbaum, "A task description language for robot control," in *Proceedings. 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems. Innovations in Theory, Practice and Applications (Cat. No. 98CH36190)*, vol. 3, pp. 1931–1937, IEEE, 1998.

[81] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das, "The claraty architecture for robotic autonomy," in *2001 IEEE Aerospace Conference Proceedings (Cat. No. 01TH8542)*, vol. 1, pp. 1–121, IEEE, 2001.

[82] R. H. Bordini, J. F. Hübner, and R. Vieira, "Jason and the golden fleece of agent-oriented programming," in *Multi-agent programming*, pp. 3–37, Springer, 2005.

[83] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming multi-agent systems in AgentSpeak using Jason*, vol. 8. John Wiley & Sons, 2007.

[84] K. V. Hindriks, F. S. De Boer, W. Van der Hoek, and J.-J. C. Meyer, "Agent programming in 3apl," *Autonomous Agents and Multi-Agent Systems*, vol. 2, no. 4, pp. 357–401, 1999.

[85] M. Dastani, M. van Birna Riemsdijk, and J.-J. C. Meyer, "Programming multi-agent systems in 3APL," in *Multi-agent programming*, pp. 39–67, Springer, 2005.

[86] M. Sierhuis, *Brahms: A multi-agent modeling and simulation language for work system analysis and design*. PhD thesis, Doctoral, University of Amsterdam, 2001.

[87] A. Pokahr, L. Braubach, and W. Lamersdorf, "Jadex: A BDI reasoning engine," in *Multi-agent programming*, pp. 149–174, Springer, 2005.

[88] L. A. Dennis and B. Farwer, "Gwendolen: a BDI language for verifiable agents," in *Proceedings of the AISB 2008 Symposium on Logic and the Simulation of Interaction and Reasoning, Society for the Study of Artificial Intelligence and Simulation of Behaviour*, pp. 16–23, 2008.

[89] K. V. Hindriks, F. S. De Boer, W. Van Der Hoek, and J.-J. C. Meyer, "Agent programming with declarative goals," in *International Workshop on Agent Theories, Architectures, and Languages*, pp. 228–243, Springer, 2000.

[90] F. S. de Boer, K. V. Hindriks, W. van der Hoek, and J.-J. C. Meyer, "A verification framework for agent programming with declarative goals," *Journal of Applied Logic*, vol. 5, no. 2, pp. 277–302, 2007.

[91] M. E. Pollack, "Plans as complex mental attitudes," *Intentions in communication*, pp. 77–103, 1990.

[92] M. E. Pollack, "The uses of plans," *Artificial Intelligence*, vol. 57, no. 1, pp. 43–68, 1992.

[93] N. K. Lincoln and S. M. Veres, "Natural language programming of complex robotic BDI agents," *Journal of Intelligent & Robotic Systems*, vol. 71, no. 2, pp. 211–230, 2013.

[94] K.-W. Ng and C.-K. Luk, "I+: A multiparadigm language for object-oriented declarative programming," *Computer Languages*, vol. 21, no. 2, pp. 81–100, 1995.

[95] P. Ridao, J. Batlle, and M. Carreras, "O2ca2, a new object oriented control architecture for autonomy: the reactive layer," *Control Engineering Practice*, vol. 10, no. 8, pp. 857–873, 2002.

[96] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe, "Modeling and designing heterogeneous systems," in *Concurrency and Hardware Design*, pp. 228–273, Springer, 2002.

[97] A. Pinto, A. L. Sangiovanni-Vincentelli, L. P. Carloni, and R. Passerone, "Interchange formats for hybrid systems: Review and proposal," in *International Workshop on Hybrid Systems: Computation and Control*, pp. 526–541, Springer, 2005.

[98] B. I. Silva, O. Stursberg, B. H. Krogh, and S. Engell, "An assessment of the current status of algorithmic approaches to the verification of hybrid systems," in *Proceedings of the 40th IEEE Conference on Decision and Control*, vol. 3, pp. 2867–2874, IEEE, 2001.

[99] S. Thrun, "Probabilistic algorithms in robotics," *Ai Magazine*, vol. 21, no. 4, pp. 93–93, 2000.

[100] A. R. Cassandra, L. P. Kaelbling, and J. A. Kurien, "Acting under uncertainty: Discrete Bayesian models for mobile-robot navigation," in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 2, pp. 963–972, IEEE, 1996.

[101] N. Roy, W. Burgard, D. Fox, and S. Thrun, "Coastal navigation-mobile robot navigation with uncertainty in dynamic environments," in *Proceedings 1999 IEEE International Conference on Robotics and Automation*, vol. 1, pp. 35–40, IEEE, 1999.

[102] C. Leung, S. Huang, N. Kwok, and G. Dissanayake, "Planning under uncertainty using model predictive control for information gathering," *Robotics and Autonomous Systems*, vol. 54, no. 11, pp. 898–910, 2006.

[103] A. Widyotriatmo and K.-S. Hong, "Decision making framework for autonomous vehicle navigation," in *2008 SICE Annual Conference*, pp. 1002–1007, IEEE, 2008.

[104] Y. Gao, *Model predictive control for autonomous and semiautonomous vehicles*. PhD thesis, UC Berkeley, 2014.

[105] C. K. Law, D. Dalal, and S. Shearrow, "Robust model predictive control for autonomous vehicles/self driving cars," *arXiv preprint arXiv:1805.08551*, 2018.

[106] L. A. Miller, "Natural language programming: Styles, strategies, and contrasts," *IBM Systems Journal*, vol. 20, no. 2, pp. 184–215, 1981.

[107] S. Veres, "Natural language programming of agents and robotic devices," *SysBrain Ltd*, vol. 184, 2008.

[108] G. Budin, "Ontology-driven translation management," *Knowledge systems and translation*, 2005.

[109] SysBrain, "User's Manual, sEnglish Publisher." `http://sysbrain.com/cognitive_agent_toolbox/sEP_User_Manual.pdf`, 2011. [Accessed: 22/07/2018].

[110] S. M. Veres, L. Molnar, and N. K. Lincoln, "The cognitive agents toolbox (CAT)-programming autonomous vehicles," 2009.

[111] J. Banks, I. Carson, B. L. Nelson, D. M. Nicol, *et al.*, *Discrete-event system simulation*. Pearson, 2005.

[112] H. Klee and R. Allen, *Simulation of dynamic systems with MATLAB and Simulink*. Crc Press, 2016.

[113] P. Koopman and M. Wagner, "Challenges in autonomous vehicle testing and validation," *SAE International Journal of Transportation Safety*, vol. 4, no. 1, pp. 15–24, 2016.

[114] J. Zander, I. Schieferdecker, and P. J. Mosterman, *Model-based testing for embedded systems*. CRC press, 2011.

[115] J.-P. Katoen, *Concepts, algorithms, and tools for model checking*. IMMD Erlangen, 1999.

[116] R. Lipka, M. Paška, and T. Potužák, "Simulation testing and model checking: A case study comparing these approaches," in *International Workshop on Software Engineering for Resilient Systems*, pp. 116–130, Springer, 2014.

[117] G. Sutcliffe, "Automated theorem proving." `http://www.cs.miami.edu/home/geoff/Courses/CSC648-12S/Content/WhatIsATP.shtml`, 2014. [Accessed: 01/09/2019].

[118] R. H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge, "Model checking agentspeak," in *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pp. 409–416, ACM, 2003.

[119] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*. MIT press, 2018.

[120] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press Cambridge, 2008.

[121] M. Y. Vardi, "Automatic verification of probabilistic concurrent finite state programs," in *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pp. 327–338, IEEE, 1985.

[122] C. Courcoubetis and M. Yannakakis, "Verifying temporal properties of finite-state probabilistic programs," in *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*, pp. 338–345, IEEE, 1988.

[123] C. Courcoubetis and M. Yannakakis, "The complexity of probabilistic verification," *Journal of the ACM (JACM)*, vol. 42, no. 4, pp. 857–907, 1995.

[124] L. De Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala, "Symbolic model checking of probabilistic processes using mtbdds and the kronecker representation," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 395–410, Springer, 2000.

[125] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle, "A markov chain model checker," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 347–362, Springer, 2000.

[126] T. Han, J.-P. Katoen, and D. Berteun, "Counterexample generation in probabilistic model checking," *IEEE transactions on software engineering*, vol. 35, no. 2, pp. 241–257, 2009.

[127] A. Bouajjani, J. Esparza, and O. Maler, "Reachability analysis of pushdown automata: Application to model-checking," in *International Conference on Concurrency Theory*, pp. 135–150, Springer, 1997.

[128] M. Althoff, *Reachability analysis and its application to the safety assessment of autonomous cars*. PhD thesis, Technische Universität München, 2010.

[129] M. Althoff, O. Stursberg, and M. Buss, "Model-based probabilistic collision detection in autonomous driving," *IEEE Transactions on Intelligent Transportation Systems*, vol. 10, no. 2, pp. 299–310, 2009.

[130] M. Althoff and J. M. Dolan, "Online verification of automated road vehicles using reachability analysis," *IEEE Transactions on Robotics*, vol. 30, no. 4, pp. 903–918, 2014.

[131] A. Lomuscio and F. Raimondi, "Mcmas: A model checker for multi-agent systems," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 450–454, Springer, 2006.

[132] R. Alur, T. A. Henzinger, and O. Kupferman, "Alternating-time temporal logic," *Journal of the ACM (JACM)*, vol. 49, no. 5, pp. 672–713, 2002.

[133] R. Fagin, J. Y. Halpern, Y. Moses, and M. Vardi, *Reasoning about knowledge*. MIT press, 2004.

[134] A. Lomuscio and M. Sergot, "Deontic interpreted systems," *Studia Logica*, vol. 75, no. 1, pp. 63–92, 2003.

[135] A. Pnueli, "The temporal logic of programs," in *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pp. 46–57, IEEE, 1977.

[136] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on Logic of Programs*, pp. 52–71, Springer, 1981.

[137] E. A. Emerson, "Temporal and modal logic," in *Formal Models and Semantics*, pp. 995–1072, Elsevier, 1990.

[138] F. Raimondi, *Model checking multi-agent systems*. PhD thesis, Doctoral, University of London, 2006.

[139] H. Qu and S. M. Veres, "Verification of logical consistency in robotic reasoning," *Robotics and Autonomous Systems*, vol. 83, pp. 44–56, 2016.

[140] M. Kwiatkowska, G. Norman, and D. Parker, "A framework for verification of software with time and probabilities," in *International Conference on Formal Modeling and Analysis of Timed Systems*, pp. 25–45, Springer, 2010.

[141] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1,

pp. 134–152, 1997.

[142] R. Bellman, "A markovian decision process," *Journal of mathematics and mechanics*, pp. 679–684, 1957.

[143] E. A. Feinberg and A. Shwartz, *Handbook of Markov decision processes: methods and applications*, vol. 40. Springer Science & Business Media, 2012.

[144] J. R. Norris and J. R. Norris, *Markov chains*, vol. 2. Cambridge university press, 1998.

[145] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen, "The ins and outs of the probabilistic model checker mrmc," *Performance evaluation*, vol. 68, no. 2, pp. 90–104, 2011.

[146] "PRISM - Probabilistic Symbolic Model Checker." `http://www.prismmodelchecker.org/`, 2016. [Accessed: 20/06/2017].

[147] M. Kwiatkowska, G. Norman, and D. Parker, "Probabilistic symbolic model checking with prism: A hybrid approach," *International Journal on Software Tools for Technology Transfer*, vol. 6, no. 2, pp. 128–142, 2004.

[148] D. A. Parker, *Implementation of symbolic model checking for probabilistic systems*. PhD thesis, Doctoral, University of Birmingham, 2003.

[149] H. Jensen, "Model checking probabilistic real time systems," in *Proc. 7th Nordic Workshop on Programming Theory*, pp. 247–261, 1996.

[150] D. Beauquier, "On probabilistic timed automata," *Theoretical Computer Science*, vol. 292, no. 1, pp. 65–84, 2003.

[151] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston, "Automatic verification of real-time systems with discrete probability distributions," *Theoretical Computer Science*, vol. 282, no. 1, pp. 101–150, 2002.

[152] K. Dräger, M. Kwiatkowska, D. Parker, and H. Qu, "Local abstraction refinement for probabilistic timed programs," *Theoretical Computer Science*, vol. 538, pp. 37–53, 2014.

[153] H. Hasson and B. Jonsson, "A logic for reasoning about time and probability," *Formal Aspects of Computing*, vol. 6, pp. 512–535, 1994.

[154] A. Aziz, V. Singhal, F. Balarin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "It usually works: The temporal logic of stochastic systems," in *International Conference on Computer Aided Verification*, pp. 155–165, Springer, 1995.

[155] V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker, "Automated verification techniques for probabilistic systems," in *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pp. 53–113, Springer, 2011.

[156] M. Kwiatkowska, G. Norman, and D. Parker, "Stochastic model checking," in *International School on Formal Methods for the Design of Computer, Com-*

munication and Software Systems, pp. 220–270, Springer, 2007.

[157] M. Kwiatkowska and D. Parker, "Advances in probabilistic model checking," *Software Safety and Security: Tools for Analysis and Verification*, vol. 33, no. 126, pp. 126–151, 2012.

[158] L. Joseph, *ROS Robotics Projects*. Packt Publishing Ltd, 2017.

[159] H. robotics, "Ros introduction." `https://husarion.com/tutorials/ros-tutorials/1-ros-introduction/`, 2013. [Accessed: 02/07/2018].

[160] R. Tellez, "Ros tutorials." `https://www.theconstructsim.com/`, 2017. [Accessed: 03/05/2018].

[161] M. Al-Nuaimi, H. Qu, and S. M. Veres, "Testing, verification and improvements of timeliness in ros processes," in *Annual Conference Towards Autonomous Robotic Systems*, pp. 146–157, Springer, 2016.

[162] Open Source Robotics Foundation, "gazebo simulator." `http://gazebosim.org/`, 2014. [Accessed: 13/06/2018].

[163] Amazon Web Services, "AWS RoboMaker." `https://docs.aws.amazon.com/robomaker/latest/dg/simulation-tools-gazebo.html`, 2019. [Accessed: 13/06/2018].

[164] P. J. Antsaklis, J. A. Stiver, and M. Lemmon, "Hybrid system modeling and autonomous control systems," in *Hybrid systems*, pp. 366–392, Springer, 1992.

[165] A. J. Van Der Schaft and J. M. Schumacher, *An introduction to hybrid dynamical systems*, vol. 251. Springer London, 2000.

[166] L. Carloni, M. D. Di Benedetto, A. Pinto, and A. Sangiovanni-Vincentelli, "Modeling techniques, programming languages, design toolsets and interchange formats for hybrid systems," tech. rep., The Columbus Project, Tech. Rep. IST-2001-38314 WPHS, 2004.

[167] A. Pinto, L. P. Carloni, R. Passerone, and A. Sangiovanni-Vincentelli, "Interchange semantics for hybrid system models," *Proc. of 5th MATHMOD*, 2006.

[168] A. S. Rao and M. Wooldridge, "Foundations of rational agency," in *Foundations of rational agency*, pp. 1–10, Springer, 1999.

[169] L. A. Dennis, M. Fisher, J. M. Aitken, S. M. Veres, Y. Gao, A. Shaukat, and G. Burroughes, "Reconfigurable autonomy," *KI-Künstliche Intelligenz*, vol. 28, no. 3, pp. 199–207, 2014.

[170] J. M. Aitken, S. M. Veres, A. Shaukat, Y. Gao, E. Cucco, L. A. Dennis, M. Fisher, J. A. Kuo, T. Robinson, and P. E. Mort, "Autonomous nuclear waste management," *IEEE Intelligent Systems*, vol. 33, no. 6, pp. 47–55, 2018.

[171] M. Kamali, L. A. Dennis, O. McAree, M. Fisher, and S. M. Veres, "Formal verification of autonomous vehicle platooning," *Science of computer programming*, vol. 148, pp. 88–106, 2017.

[172] M. Bratman *et al.*, *Intention, plans, and practical reason*, vol. 10. Harvard University Press Cambridge, MA, 1987.

[173] P. Izzo, *Verification-driven design and programming of autonomous robots.* PhD thesis, Doctoral, University of Sheffield, 2016.

[174] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: part i," *IEEE robotics & automation magazine*, vol. 13, no. 2, pp. 99–110, 2006.

[175] T. Bailey and H. Durrant-Whyte, "Simultaneous localisation and mapping (slam) part 2: State of the art," *Robotics and Automation Magazine*, 2006.

[176] G. Bresson, Z. Alsayed, L. Yu, and S. Glaser, "Simultaneous localization and mapping: A survey of current trends in autonomous driving," *IEEE Transactions on Intelligent Vehicles*, vol. 2, no. 3, pp. 194–220, 2017.

[177] M. Fisher, L. Dennis, and M. Webster, "Verifying autonomous systems," *Communications of the ACM*, vol. 56, no. 9, pp. 84–93, 2013.

[178] A. Bera, S. Kim, T. Randhavane, S. Pratapa, and D. Manocha, "Glmp-realtime pedestrian path prediction using global and local movement patterns," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5528–5535, IEEE, 2016.

[179] J. F. P. Kooij, N. Schneider, F. Flohr, and D. M. Gavrila, "Context-based pedestrian path prediction," in *European Conference on Computer Vision*, pp. 618–633, Springer, 2014.

[180] S. Maghsoudi and I. Watson, "Epistemic logic and planning," in *Proceedings of Knowledge-Based Intelligent Information and Engineering Systems (KES'04)*, vol. 3214 of *Lecture Notes in Computer Science*, pp. 36–45, Springer, 2004.

[181] M. Shanahan and M. Witkowski, "High-Level Robot Control Through Logic," in *Proceedings of Intelligent Agents VII Agent Theories Architectures and Languages*, vol. 1986 of *Lecture Notes in Computer Science*, pp. 104–121, Springer, 2001.

[182] M. Singh, D.R.Parhi, S.Bhowmik, and S.K.Kashyap, "Intelligent controller for mobile robot: Fuzzy logic approach," in *Proceedings of International Association for Computer Methods and Advances in Geomechanics (IACMAG'08)*, pp. 1755–1762, 2008.

[183] C. R. Torres, J. M. Abe, G. Lambert-Torres, J. I. D. S. Filho, and H. G. Martins, "Autonomous mobile robot emmy iii," in *Proceedings of New Advances in Intelligent Decision Technologies*, vol. 199 of *Studies in Computational Intelligence*, pp. 317–327, Springer, 2009.

[184] F. W. Trevizan, L. N. De Barros, and F. S. C. Da Silva, "Designing logic-based robots," *Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial*, vol. 10, no. 31, pp. 11–22, 2006.

[185] P. B. Vranas, "New foundations for imperative logic i: Logical connectives,

consistency, and quantifiers," *Noûs*, vol. 42, no. 4, pp. 529–572, 2008.

[186] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and computation*, vol. 98, no. 2, pp. 142–170, 1992.

[187] Mathworks, "Coordinate systems in automated driving toolbox." `https://uk.mathworks.com/help/driving/ug/coordinate-systems.html`, 2018. [Accessed: 14/10/2018].

[188] S. M. LaValle, "Yaw, pitch, and roll rotations." `http://planning.cs.uiuc.edu/node102.html`, 2012. [Accessed: 28/02/2020].

[189] S. M. LaValle and J. J. Kuffner Jr, "Randomized kinodynamic planning," *The international journal of robotics research*, vol. 20, no. 5, pp. 378–400, 2001.

[190] R. Rajamani, *Vehicle dynamics and control*. Springer Science & Business Media, 2011.

[191] G. M. Hoffmann, C. J. Tomlin, M. Montemerlo, and S. Thrun, "Autonomous automobile trajectory tracking for off-road driving: Controller design, experimental validation and racing," in *American Control Conference*, pp. 2296–2301, IEEE, 2007.

[192] MathWorks, "Automated Parking Valet in Simulink." `https://uk.mathworks.com/help/driving/examples/automated-parking-valet-in-simulink.html`, 2017. [Accessed: 17/05/2018].

[193] S. Gu, T. Lu, Y. Zhang, J. M. Alvarez, J. Yang, and H. Kong, "3-D LiDAR + Monocular Camera: An Inverse-Depth-Induced Fusion Framework for Urban Road Detection," *IEEE Transactions on Intelligent Vehicles*, vol. 3, no. 3, pp. 351–360, 2018.

[194] N. Seenouvong, U. Watchareeruetai, C. Nuthong, K. Khongsomboon, and N. Ohnishi, "A computer vision based vehicle detection and counting system," in *8th International Conference on Knowledge and Smart Technology*, pp. 224–227, IEEE, 2016.

[195] T. Kato, C. Guo, K. Kidono, Y. Kojima, and T. Naito, "SpaFIND: An effective and low-cost feature descriptor for pedestrian protection systems in economy cars," *IEEE Transactions on Intelligent Vehicles*, vol. 2, no. 2, pp. 123–132, 2017.

[196] R. O. Chavez-Garcia and O. Aycard, "Multiple sensor fusion and classification for moving object detection and tracking," *IEEE Transactions on Intelligent Transportation Systems*, vol. 17, no. 2, pp. 525–534, 2015.

[197] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv*, 2018.

[198] J. Leonard, J. How, S. Teller, M. Berger, S. Campbell, G. Fiore, L. Fletcher, E. Frazzoli, A. Huang, S. Karaman, *et al.*, "A perception-driven autonomous urban vehicle," in *The DARPA urban challenge*, pp. 163–230, Springer, 2009.

[199] P. Abbeel and A. Y. Ng, "Apprenticeship learning via inverse reinforcement learning," in *Proceedings of the twenty-first international conference on Machine learning*, p. 1, ACM, 2004.

[200] P. Abbeel, D. Dolgov, A. Y. Ng, and S. Thrun, "Apprenticeship learning for motion planning with application to parking lot navigation," in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1083–1090, IEEE, 2008.

[201] C. Hubmann, J. Schulz, M. Becker, D. Althoff, and C. Stiller, "Automated driving in uncertain environments: Planning with interaction and uncertain maneuver prediction," *IEEE Transactions on Intelligent Vehicles*, vol. 3, no. 1, pp. 5–17, 2018.

[202] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles," *IEEE Transactions on intelligent vehicles*, vol. 1, no. 1, pp. 33–55, 2016.

[203] P. Marin-Plaza, A. Hussein, D. Martin, and A. d. l. Escalera, "Global and local path planning study in a ros-based research platform for autonomous vehicles," *Journal of Advanced Transportation*, vol. 2018, 2018.

[204] S. Skiena, "Dijkstra's algorithm," in *Implementing discrete mathematics: combinatorics and graph theory with mathematica*, pp. 225–227, Addison-Wesley Reading, MA, 1990.

[205] C. Rösmann, W. Feiten, T. Wösch, F. Hoffmann, and T. Bertram, "Trajectory modification considering dynamic constraints of autonomous robots," in *ROBOTIK; 7th German Conference on Robotics*, pp. 1–6, VDE, 2012.

[206] C. Rösmann, W. Feiten, T. Wösch, F. Hoffmann, and T. Bertram, "Efficient trajectory optimization using a sparse model," in *European Conference on Mobile Robots*, pp. 138–143, IEEE, 2013.

[207] Jihoonl, "Dijkstra algorithm-ROS package." `http://wiki.ros.org/asr_navfn`, 2014. [Accessed: 13/02/2018].

[208] S. A. Fadzli, S. I. Abdulkadir, M. Makhtar, and A. A. Jamal, "Robotic indoor path planning using dijkstra's algorithm with multi-layer dictionaries," in *2nd International Conference on Information Science and Security*, pp. 1–4, IEEE, 2015.

[209] D. Ferguson, T. M. Howard, and M. Likhachev, "Motion planning in urban environments," *Journal of Field Robotics*, vol. 25, no. 11-12, pp. 939–960, 2008.

[210] C. Roesmann, "teb_local_planner ROS Package." `http://wiki.ros.org/teb_local_planner`, 2015. [Accessed: 15/02/2018].

[211] M. Keller, F. Hoffmann, C. Hass, T. Bertram, and A. Seewald, "Planning of optimal collision avoidance trajectories with timed elastic bands," *IFAC Proceedings*, vol. 47, no. 3, pp. 9822–9827, 2014.

[212] N. Varas, "move_base." http://wiki.ros.org/move_base, 2017. [Accessed: 02/05/2018].

[213] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU scheduling on the NVIDIA TX2: Hidden details revealed," in *IEEE Real-Time Systems Symposium*, pp. 104–115, IEEE, 2017.

[214] Nvidia, "Jetson TX2." https://www.nvidia.com/en-gb/autonomous-machines/embedded-systems/jetson-tx2/, 2018. [Accessed: 02/09/2018].

[215] Nvidia, "Computer unified device architecture." https://developer.nvidia.com/cuda-zone. [Accessed: 23/01/2018].

[216] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, pp. 91–99, 2015.

[217] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *European conference on computer vision*, pp. 21–37, Springer, 2016.

[218] A. Sachan, "Zero to Hero: Guide to Object Detection using Deep Learning: Faster R-CNN,YOLO,SSD." https://cv-tricks.com/object-detection/faster-r-cnn-yolo-ssd/, 2018. [Accessed: 03/02/2020].

[219] N. Yadav and U. Binay, "Comparative study of object detection algorithms," *International Research Journal of Engineering and Technology (IRJET)*, vol. 4, no. 11, pp. 586–591, 2017.

[220] A. Neubeck and L. Van Gool, "Efficient non-maximum suppression," in *18th International Conference on Pattern Recognition (ICPR'06)*, vol. 3, pp. 850–855, IEEE, 2006.

[221] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7263–7271, 2017.

[222] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[223] R. P. T. Ltd, "Raspberry Pi Compute Module 3+, Raspberry Pi Compute Module 3+ Lite." https://www.raspberrypi.org/documentation/hardware/computemodule/datasheets/rpi_DATA_CM3plus_1p0.pdf, 2019. [Accessed: 04/02/2020].

[224] F. A. Hermawati, H. Tjandrasa, and N. Suciati, "Combination of Aggregated Channel Features (ACF) Detector and Faster R-CNN to Improve Object Detection Performance in Fetal Ultrasound Images," *International Journal of Intelligent Engineering and Systems*, vol. 11, no. 6, 2018.

[225] MathWorks, "People Detector ACF." https://uk.mathworks.com/help/vision/ref/peopledetectoracf.html?searchHighlight=aggregate%

20channel%20features%20(ACF)&s_tid=doc_srchtitle, 2017. [Accessed: 13/02/2018].

[226] J. Kim, J. Baek, Y. Park, and E. Kim, "New vehicle detection method with aspect ratio estimation for hypothesized windows," *Sensors*, vol. 15, no. 12, pp. 30927–30941, 2015.

[227] MathWorks, "Vehicle Detector-ACF." https://uk.mathworks.com/help/driving/ref/vehicledetectoracf.html?searchHighlight=aggregate%20channel%20features%20(ACF)&s_tid=doc_srchtitle, 2017. [Accessed: 13/02/2018].

[228] S. Garrido-Jurado, R. Muñoz-Salinas, F. J. Madrid-Cuevas, and M. J. Marín-Jiménez, "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recognition*, vol. 47, no. 6, pp. 2280–2292, 2014.

[229] JimVaughan, "fiducials." http://wiki.ros.org/fiducials, 2018. [Accessed: 10/01/2019].

[230] Ubiquity Robotics, "Simultaneous Localization and Mapping Using Fiducial Markers." https://github.com/UbiquityRobotics/fiducials, 2018. [Accessed: 11/01/2019].

[231] S. Garrido, "ArUco marker detection (aruco module)." https://docs.opencv.org/master/d9/d6d/tutorial_table_of_content_aruco.html, 2015. [Accessed: 06/02/2020].

[232] Waymo, "Lidar - Laser Bear Honeycomb – Waymo." https://waymo.com/lidar/, 2019. [Accessed: 30/09/2019].

[233] B. K. Paul Lienert, "A chaotic market for one sensor stalls self-driving cars." https://www.reuters.com/article/us-autos-autonomous-lidar-focus/a-chaotic-market-for-one-sensor-stalls-self-driving-cars-idUSKCN1QN0HW, 2019. [Accessed: 30/09/2019].

[234] Velodyne lidar, "Velodyne lidar VLP-16." https://velodynelidar.com/vlp-16.html, 2016. [Accessed: 22/02/2018].

[235] Velodyne lidar, "Velodyne lidar VLP-16 Manual." https://velodynelidar.com/downloads.html, 2016. [Accessed: 22/02/2018].

[236] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*, vol. 1. MIT press Cambridge, 2000.

[237] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. J. Leonard, and F. Dellaert, "iSAM2: Incremental smoothing and mapping using the Bayes tree," *The International Journal of Robotics Research*, vol. 31, no. 2, pp. 216–235, 2012.

[238] K. Konolige, M. Agrawal, and J. Sola, "Large-scale visual odometry for rough terrain," in *Robotics research*, pp. 201–212, Springer, 2010.

[239] D. Nistér, O. Naroditsky, and J. Bergen, "Visual odometry for ground vehicle applications," *Journal of Field Robotics*, vol. 23, no. 1, pp. 3–20, 2006.

[240] uStepper, "uStepper (Rev. B) product description." http://ustepper.com/product_sheet_revB.pdf, 2017. [Accessed: 10/06/2018].

[241] uStepper, "uStepper." https://github.com/uStepper/uStepper, 2016. [Accessed: 12/06/2018].

[242] M. Farsi, K. Ratcliff, and M. Barbosa, "An overview of controller area network," *Computing & Control Engineering Journal*, vol. 10, no. 3, pp. 113–120, 1999.

[243] T. systems, "TOPCON AES-25 user manual." https://www.topconpositioning.com/agricultural-machine-control, 2013. [Accessed: 21/05/2018].

[244] C. Instruments, "Curtis 1234/36/38 Manual." https://www.curtisinstruments.com/products/motor-controllers/, 2011. [Accessed: 13/05/2018].

[245] A. Rasouli, I. Kotseruba, and J. K. Tsotsos, "Understanding pedestrian behavior in complex traffic scenes," *IEEE Transactions on Intelligent Vehicles*, vol. 3, no. 1, pp. 61–70, 2017.

[246] K. Saleh, M. Hossny, and S. Nahavandi, "Intent prediction of pedestrians via motion trajectories using stacked recurrent neural networks," *IEEE Transactions on Intelligent Vehicles*, vol. 3, no. 4, pp. 414–424, 2018.

[247] A. Rangesh and M. M. Trivedi, "When vehicles see pedestrians with phones: A multicue framework for recognizing phone-based activities of pedestrians," *IEEE Transactions on Intelligent Vehicles*, vol. 3, no. 2, pp. 218–227, 2018.

[248] A. Rasouli, I. Kotseruba, and J. K. Tsotsos, "Agreeing to cross: How drivers and pedestrians communicate," in *IEEE Intelligent Vehicles Symposium*, pp. 264–269, IEEE, 2017.

[249] J. Wiest, M. Höffken, U. Kreßel, and K. Dietmayer, "Probabilistic trajectory prediction with gaussian mixture models," in *IEEE Intelligent Vehicles Symposium*, pp. 141–146, IEEE, 2012.

[250] Cutis Instruments Inc., "1234/36/38 Manual." http://www.revagwizclub.co.uk/PDFs/CURTIS%201234_36_38%20USER%20MANUAL.pdf, 2009. [Accessed: 02/03/2020].