A distributed safety mechanism for autonomous vehicle software using hypervisors

van der Perk, P.J.

Award date:
2019

Department of Electrical Engineering
Electronic Systems Research Group

A Distributed Safety Mechanism for Autonomous Vehicle Software Using Hypervisors

graduation project

Peter van der Perk

Supervisors:
prof.dr. Kees Goossens
dr. Andrei Terechko

version 1.0

Eindhoven, June 2019

Abstract

Autonomous vehicles rely on cyber-physical systems to provide comfort and safety to the passengers. The objective of safety designs is to avoid unacceptable risk of physical injury to people. Reaching this objective, however, is very challenging because of the growing complexity of both the Electronic Control Units (ECUs) and software architectures required for autonomous operation.

The thesis begins with an extensive survey of state-of-the-art safety concepts and software architectures for autonomous driving systems. Based on this survey we identified three system design challenges: fault handling in distributed processes, hazardous road situations in the absence of faults and freedom from interference in ECUs consolidating multiple functions.

Inspired by the E-Gas concept of distributed health monitoring, we split our safety mechanism into function, platform, and vehicle levels, which we run on different processors. For the software middleware to enable communication between distributed processors, we chose the Data Distribution Service (DDS) protocol, which is deployed in safety-critical applications across industries. At present safety cores are not capable to run the full DDS stack, however, there exists a DDS subset - DDS-XRCE specially designed for resource-constrained devices. We ported DDS-XRCE onto NXP S32R274 SoC, which is ready for the highest automotive safety integrity level, to enable this processor's participation in the safety mechanism. Remarkably, the DDS middleware exposes the application software state, which safety mechanisms can use to identify hazardous situations in the absence of faults.

Freedom from interference between multiple software stacks is supported by modern processors through hardware virtualization and hypervisors. In our study, we chose the Xen hypervisor, which can host and isolate multiple operating systems on top of it. Using Xen on NXP S32V234 ASIL B capable SoC (System-on-a-Chip) we instantiated two domains - one for vehicle control and one for health monitoring.

To validate the concepts of our safety mechanism, we built an experimental setup using the LG SVL simulator and Baidu Apollo software framework on the NXP BlueBox platform for autonomous driving. A take-over driving scenario was developed using the Python API of the LG SVL simulator to ensure reproducible and deterministic analysis of the safety mechanism. To validate our safety mechanism as part of this scenario, we programmatically injected a fault - a high CPU load in the software stack responsible for vehicle control in the target BlueBox system.

Based on our fault injection experiment, we concluded that safety mechanisms for distributed processing can rely on the DDS middleware. In our implementation, the middleware also enables resource-constrained safety cores to read application state, which in the future can be used to identify hazardous road situations in the absence of faults. Furthermore, we discovered that the hypervisors are not only useful in ensuring freedom from interference, but they can also implement a fail-silent behavior of faulty software stacks. In the final chapter, we listed promising future research directions, including the challenge-response mechanism in distributed health monitoring and a study of the non-atomic nature of the hypervisor-based software pause in a complex SoC.

Acknowledgements

This master project is conducted in the Systems and Applications group of NXP Semiconductors, Eindhoven. Throughout this research, I have received a great deal of support and assistance.

I would like to express the deepest appreciation to my supervisor Andrei Terechko, for giving me the opportunity to conduct my research and further my thesis at NXP Semiconductors. During my research, you have guided me through the wonderful world of functional safety, the automotive industry, and embedded systems. Your expert guidance and tremendous experience in these domains, has helped me to develop and improve my skills, both technical and non-technical. You have provided me invaluable encouragement and support in various ways. I am really indebted to you more than you know.

I would like to thank my university supervisor, Kees Goossens. During our meetings you have provided me with valuable feedback during the course my of research. Focusing on the academic aspect of my research and pinpointing valuable topics.

I would like to show my greatest appreciation to the colleagues of the Systems and Applications group at NXP Semiconductors in particular: Yuting Fu, thank you for your critical comments during our brainstorm sessions providing value feedback. Bart Vermeulen, thank your for joining some meetings and providing your expertise and suggestions.

I appreciate the feedback offered by Tjerk Bijlsma from TNO-ESI, Your expertise from the EcoTwin project and great understanding of functional safety, provided valuable input for my research.

Finally, I wish to express my thanks and gratitude to my parents, brothers, and grandparents for believing in me and being with moral support. In particular, I would like to thank my momther, Yke van der Perk, without her proper guidance and encouragement it would have been impossible for me to complete my Master's degree.

Peter van der Perk
Eindhoven, the Netherlands
June 2019

Glossary

ADAS Advanced Driver Assistance Systems. 3, 12

ASIL Automotive Safety Integrity Level. 11, 14, 15, 21

CAN Controller Area Network. 20

DDS Data Distribution Service. 11, 21, 22, 26, 30, 34–36

DDS-TSN DDS for Time-Sensitive Networking. 11

DDS-XRCE DDS for eXtremely Resource Constrained Environments. 11, 21, 26, 34–36

DSM Distributed Safety Mechanism. 22, 23, 30

ECU Electronic Control Unit. 1, 18, 22

fault tolerance property of a system to continue operating properly in the event of one or more faults [39]. 4

FCCU Fault Collection and Control Unit. 26

freedom from interference absence of cascading failures between two or more elements that could lead to the violation of a safety requirement [39]. 24, 36

GNSS Global Navigation Satellite System. 3, 4

HAD Highly Automated Driving. 2–5, 8, 16, 17, 20, 22, 24–28, 30, 31, 33, 34, 36

hypervisor A hypervisor or virtual machine monitor is computer software, firmware or hardware that creates and runs virtual machines [82]. 12–15

LiDAR Light Detection And Ranging. 4, 16

ROS Robot Operating System. 4, 5, 11, 20, 21, 30

RTOS Real-Time Operating System. 11, 12, 14, 15

SAE Society of Automotive Engineers. 3

SoC System-on-Chip. 16, 24–26, 30, 31, 33, 37

SOTIF ISO/PAS 21448, Road vehicles – Safety of the intended functionality. Standard draft. 7, 8, 18, 26, 35, 37

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research context

Modern automotive system development focuses on improving safety and comfort of the passengers. According to [66] the majority of car accidents are attributed to human errors. Therefore, autonomous vehicles or self-driving cars embark on automating the basic driving functionality and replacing the human driver with electronics and software. One of the key challenges, however, is to design an autonomous driving system that is to be considered safer than human drivers. Guaranteeing safety of an autonomous vehicle is challenging because of the growing complexity of both the Electronic Control Unit (ECU) and software architectures required for autonomous operation. The automotive industry has a long history of dealing with mechanical and hardware problems, but the rapid expansion of software in the car poses new challenges to the automotive safety community. According to [51] a modern vehicle runs on millions lines of code, even without achieving fully autonomous operation. This research analyzes software for future autonomous vehicles and empirically studies promising safety mechanisms.

1.2 Thesis outline

The thesis begins with an extensive state-of-the-art survey in Chapter 2. Following traditional safety standards practices, we begin this chapter with the description of the application functionality of the autonomous vehicles, followed by a survey of state-of-the-art safety concepts and standards. Chapter 2 ends with a presentation of a representative automotive hardware platform. Based on the state-of-the-art study, Chapter 3 formulates the tackled research problem in the context of autonomous driving, followed by Chapter 4 with a definition of the research methodology relying on empirical studies. Software design goals derived from the problem statement begin Chapter 5, which subsequently details promising elements of a distributed software architecture of the safety mechanism. Chapter 6 is dedicated to the experimental evaluation of the designed safety mechanism, which is validated using a fault injection in a simulated driving scenario. Based on the experimental results we conclude our study in Chapter 7 by summarizing how adequate the explored safety mechanisms appear to be for the formulated problem. Interesting future research directions wrap up the thesis in Chapter 8.

Chapter 2

State of the art overview

Full self-driving automation brings the need for new techniques in design, implementation, and safety of software [56]. Figure 2.1 shows the dependency graph of the elements presented in this chapter, which are required for the distributed safety mechanism. The components colored in gray are out of scope in this study. First, Section 2.1 introduces architectural concepts of Highly Automated Driving (HAD) function software frameworks. This chapter presents a state of the art overview of safety concepts in section 2.2. In section 2.3 we present the concept of software middleware that provides services and communication for distributed applications. In section 2.4 we present operating systems for running the applications. Finally, in section 2.5 we present automotive hardware platforms for running the whole software stack.

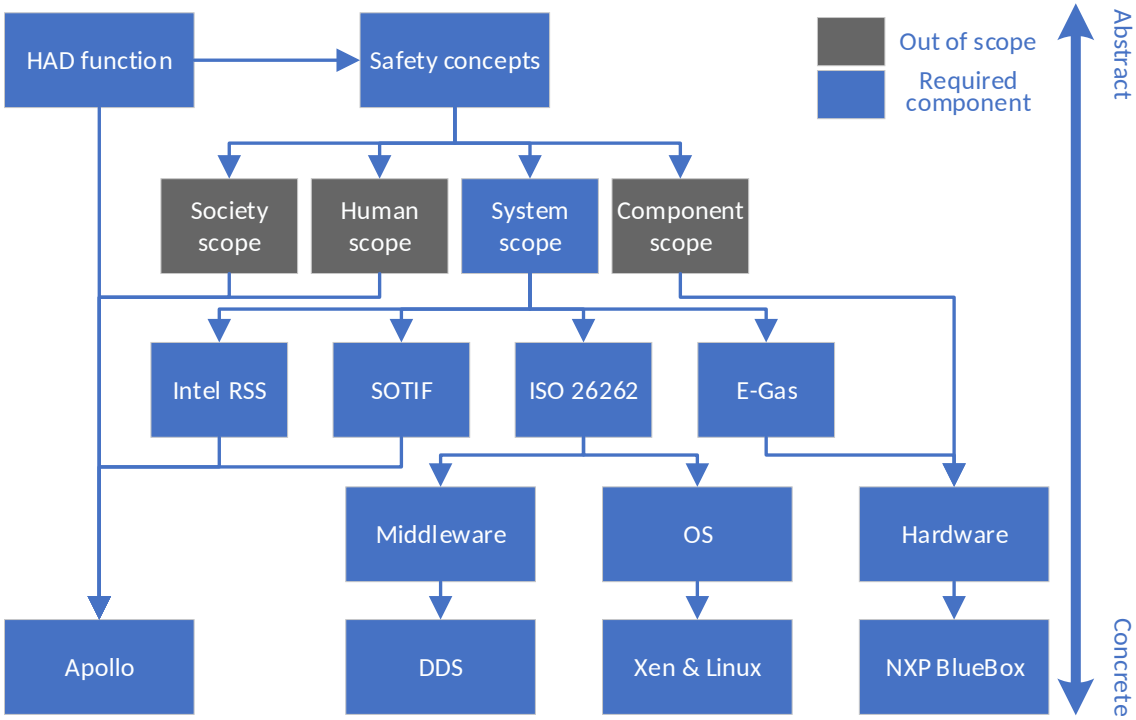

Figure 2.1: Dependency graph of state of the art elements

2.1 Highly Automated Driving (HAD) application

To categorize Highly Automated Driving (HAD) capabilities, consider the automation level classification proposed by Society of Automotive Engineers ranking the automation capabilities from level 0 through level 5 [69]. On level 1-2 automation systems are called Advanced Driver Assistance Systems which assists the drivers, but monitoring is still done by driver, whereas in level 3-5 the automated vehicle monitors the environment. The focus of this study is on level 3-5 automation.

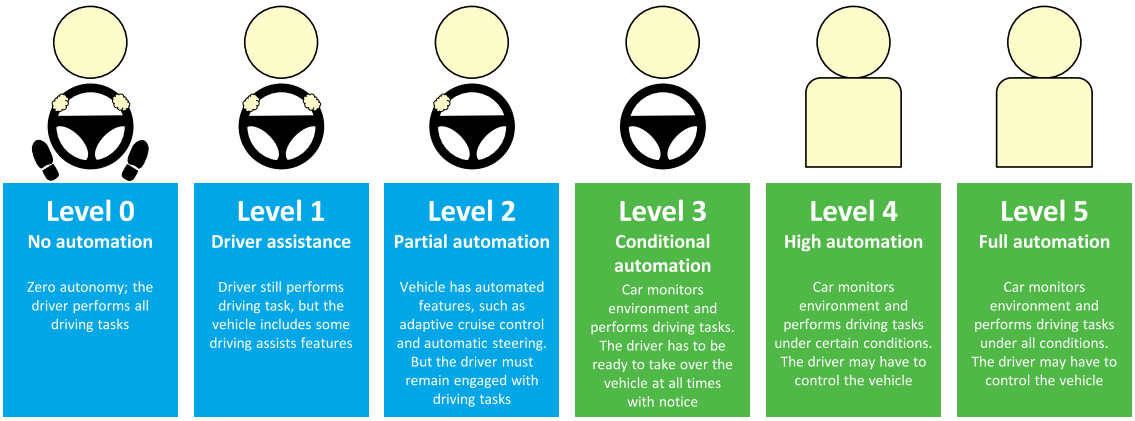

Figure 2.2: SAE automation levels [69]

2.1.1 HAD software frameworks

A HAD software framework consists of a set of self-driving modules composed of localization, perception, prediction, planning, and control capabilities. Figure 2.3 illustrates a simplified relation between different modules.

Localization

For car navigation, Global Navigation Satellite System (GNSS) is sufficient to localize the car and plan a route to drive. However, for self-driving cars GNSS accuracy of 5 meters and an update

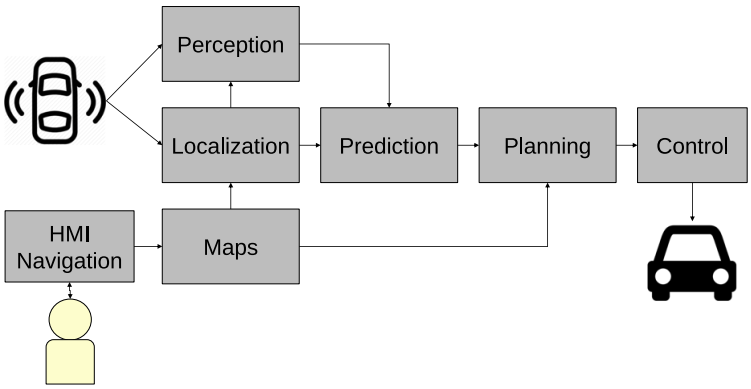

Figure 2.3: HAD framework functional architecture

rate of 1hz is not sufficient. Therefore, HAD frameworks use sensor fusion with other sensors such as LiDAR, Odometry, and Cameras to achieve accuracy of 1cm and higher update rates.

Perception

Is detection and classification of surrounding objects using sensor fusion of the following sensors; LiDAR, Radar, Camera and Ultrasonic. Objects will be used classified as cars, cyclists, pedestrians and other road users.

Prediction

The perception module receives the classified objects from the perceptions module and tries to predict the next location in time of these objects. Which is done through various algorithms such as: Kinematic filter, Bayes filter, Kalman filter, Neural networks.

Planning

Planning consists of 2 parts which are mission planning and motion planning. Mission planning is path planning that sets out exact waypoints on which lane to go using a High Definition Map (HD Map). Motion planning is path following, it uses the precise location from the localization module and the waypoint information from the mission planner and plans a trajectory for the car to follow.

Control

The control module is responsible for the steering, throttle, and brake and executes the planned trajectory from the motion planner. The trajectory information from the planning module goes through a proportional-integral-derivative controller or model predictive controller, which tries to make control actions as smooth as possible so that the passenger experiences a pleasant drive.

2.1.2 Feature analysis of HAD software frameworks

During the state of the art study we have found several frameworks that implement a (sub)set of the capabilities described in section 2.1.1. Table 2.1 presents an overview of these HAD frameworks. It can be observed that all these HAD frameworks are built on top high-level systems such as ROS, Linux, and Android. The Baidu Apollo [18], EcoTwin [22] and Elektrobit robinos [55] frameworks incorporate health monitor functionality.

Apollo version 3.0 uses a monitor and guardian module to improve safety: The monitor module monitors the states of all hardware such as camera's, LiDAR, radar, GNSS and the CAN bus and monitors all the Apollo ROS software modules. When a hardware fault e.g. LiDAR failure occurs or an Apollo ROS module is unresponsive. The monitor informs the guardian module to go into a safe mode. The guardian puts the car in safe mode by decoupling the control module from controlling the car. The car will be controlled by guardian during safe mode. The current implementation of the safe mode in the guardian is a basic safe stop that starts braking and puts the steering wheel in the center position.

2.1.3 Fault tolerance in HAD software frameworks

Fault tolerance is a property of a system to operate properly in the event of a fault in one of its components. The HAD frameworks shown in Table 2.1 have limited fault tolerance, some HAD frameworks implement a health monitor to inform the driver or a separate fall back system whether a fault has occurred therefore the car can be stopped safely. However, fundamentally most of these HAD frameworks lack fault tolerance since:

Table 2.1: HAD framework overview.

Name	Architecture	Health Monitor	Lidar Sensor	Radar Sensor	Camera Sensor	SAE Features
Apollo 3.0 [18]	ROS based	Yes	Yes	Yes	Yes	Level 3-4
Apollo 3.5 [18]	Cyber RT [29]	Yes	Yes	Yes	Yes	Level 3-4
Autoware.AI [41]	ROS based	No	Yes	Yes	Yes	Level 3-4
Autoware.Auto	ROS2 based	No	No	No	No	Level 3-4
EcoTwin [22]	ROS based	Yes	No	Yes	Yes	Level 2+ [22]
EB robinos [55]	Linux based	Yes	Yes	Yes	Yes	Level 3-4
NVDIA Drive [16]	Linux based	No	Yes	Yes	Yes	Level 2-4
Comma.ai [62]	Android based	No	No	No	Yes	Level 2 [35]

- Software is designed to run on a non-certified x86 [34] hardware platform
- Software is designed to run on Linux which in the current state is not a safe operating system [47]
- HAD components are based on ROS which is, as shown in Section 2.3.1, not a good foundation for safety critical tasks.
- HAD components are running on a central system where they can interfere each other.

2.2 Safety standards and concepts

Safety of autonomous vehicle relies on rigorous development processes and integrated safety mechanisms. Both the processes and mechanisms are captured in various safety standards, such as the Road Vehicles - Functional Safety standard ISO 26262. Besides standards, there exist design patterns or safety concepts, which constitute the state-of-the-art in this field, such as voting mechanisms [26]. This Section briefly describes fault models, relevant safety standards and concludes with a scoping diagram positioning various safety frameworks.

2.2.1 Fault models and monitoring techniques

A fault model [80] is an engineering model of something that could go wrong in the construction or operation of a piece of equipment. From the model, the designer or user can then predict the consequences of this particular fault.

At the system level, we can distinguish between hardware and software fault models. The electronics hardware fault models are well studied and documented, for example, stuck-at faults, open faults, and short faults. Software fault models, on the other hand, are less well studied [32], in our study, we have categorized them in Figure 2.4.

We divide software fault models in algorithm, computation, and communication models. Algorithm deficiencies require complex models and are algorithm specific therefore for our solution we assume that the author of the algorithm creates their own algorithm specific monitor. Computation faults originate from incorrect algorithm implementation in a particular programming language. For example, a poor memory management system can lead to a memory leak and subsequent system halt. An example of a computation control flow fault is a deadlock caused by locks with circular dependencies. Network packet corruption is a typical example of a communication fault model.

There exists many different monitoring techniques to detect faults. In Table 2.2 we classify monitoring techniques according to their nature (passive or active), performance impact, latency and interference effects, some classifications are unknown since it highly dependent on the implementation. When designing a safety monitor, it is important to select one with appropriate trade-offs among these classes. Active monitoring techniques rely on the system itself to report

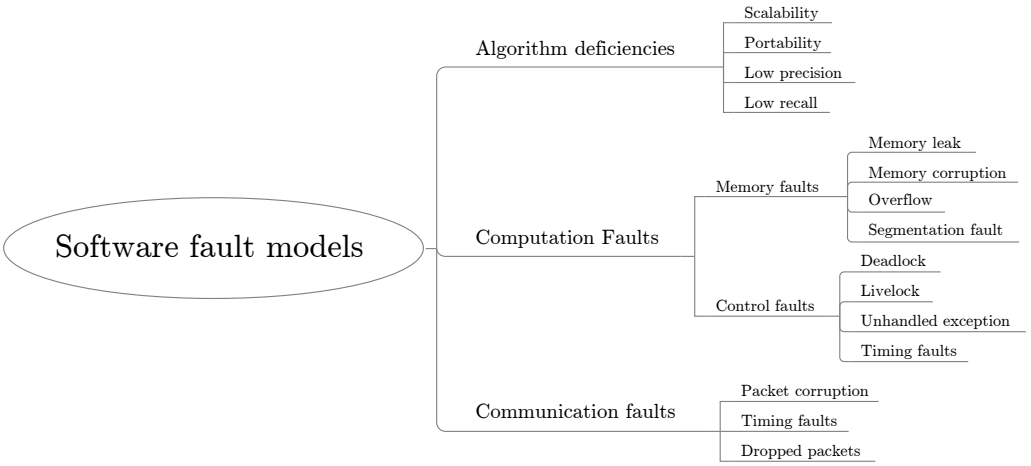

Figure 2.4: Software fault models

its health state, whereas passive ones observe the system outputs from outside. The performance impact column indicates how much execution time is dedicated for monitoring.

Below is a brief explanation of the techniques:

- Publish/subscribe. In an application with a publish-subscribe middleware, a health monitor can subscribe to topics of the monitored system and analyze the outgoing or incoming data for timing errors or data integrity issues.
- Memory polling. In a shared memory system, the monitor can read memory and analyze its content for data integrity issues.
- Power supply check. A power supply line on a printed circuit board or in an IC can be checked, for example, for undervoltage or overvoltage.
- Peripheral status bit check. A peripheral device on an IC can expose its status through a memory-mapped IO interface. The monitor can then check this status bit using a memory-mapped load operation.
- (Boundary) scan chain. A boundary scan chain, typically used for debugging or testing, can also be used to read the state of the integrated circuit or a printed circuit board.
- Network sniffing. Network interface drivers in OSes, such as Linux or BSD, allow for monitoring network packets by applications not involved directly in the network communication. This feature can be used by monitors to analyze network traffic of the system and identify faults.
- Port scan. In networking a port scan involves an external network node interrogating the port of a node, which is part of the system under monitoring. This enables checking if an active server application is running properly behind the port.
- Hardware heartbeat. An electronic circuit can generate a periodic pulse signal, indicating proper operation of the circuit itself. If the pulse is delayed or totally absent, the monitor can identify a system fault.
- Software heartbeat. Similarly to hardware, the software can also generate a periodic event, such as an interrupt or a memory cell change, to signal it's proper behavior. An external hardware or software monitor can then analyze the software heartbeat to identify faults.
- System load profiling. Processor load beyond a certain threshold can result in delaying outputs or even application crashes. Therefore, a monitor software, such as GNU/Linux top, can calculate the load of the system and compare it to the dangerous level.
- System status remote procedure call. An application software can also provide its health information through a dedicated API, such as XML-RPC, used for a remote procedure call.
- Challenge-response. Both hardware and software systems may have dedicated interfaces for

interrogations by monitors. Using this interface the monitor sends a challenge, for example, an input to a certain function, and the system needs to compute this function and return the response to the monitor.

Our work does not propose new fault models or monitoring techniques to detect existing fault models; we intend to reuse existing control flow, data integrity, and timing monitors. The focus of our work instead will be to construct a vehicle-level distributed platform for monitoring information exchanging and recovery mechanisms.

Table 2.2: Monitoring techniques

Technique	Active/Passive	Performance impact	Latency	Interference
Publish/Subscribe	Passive	Medium	Low	Yes
Memory polling	Passive	Low	Low	Yes
Power supply check	Passive	None	Low	No
Peripheral status bit check	Passive	Low	Low	Yes
(Boundary) scan chain	Passive	Low/High	High	Yes
Network sniffing	Active	Low	Low	Yes
Port scan	Active	Low	Low	Yes
Hardware heartbeat	Active	Low	?	?
Software heartbeat	Active	Medium	?	Yes
System load profiling	Active	Medium/High	High/Medium	Yes
System status RPC	Active	High	High	Yes
Challenge-response	Active	High	High	Yes

2.2.2 Device reliability

Reliability engineering is a well-established discipline in system engineering focusing on the dependability of devices. Reliability is defined as the probability of success or as a frequency of failures [87]. There exist multiple frameworks for electronic device reliability, such as AEC Q100 [77] and SN 29500 [2]. Device reliability is the foundation of functional safety standards, such as ISO 26262.

2.2.3 ISO 26262 Road Vehicles - Functional safety

ISO 26262 [39] is an automotive adaption of the IEC 61508 [38] standard which is a generic functional safety standard supporting the design, development, and operation of electrical/electronic/programmable safety-related systems. ISO 26262 defines functional safety for automotive equipment and addresses processes and practices of the product development phase, ranging from specification, design, development, verification, production, and support. ISO 26262 is a risk-based safety standard and specifies that during the concept phase of the product development an item definition, hazard analysis and risk assessment, and a functional safety concept must be made. Primarily ISO 26262 focuses on reducing or mitigation of faults and malfunction. Faults are categorized into systematic faults, including software bugs, and random hardware faults, occurring at run-time, for example, due to cosmic radiation.

2.2.4 ISO/PAS 21448 Road Vehicles - Safety of the intended functionality

ISO/PAS 21488 [40] is an emerging standard draft, which addresses the safety of the intended functionality (SOTIF) or safety in use. The SOTIF draft is complementary to ISO 26262, because the former does not concern with system faults and malfunction. Note, however, that to meet SOTIF standard, the system should first comply with ISO 26262. The goal of the safety of

the intended functionality draft is to reduce the risk of hazards due to performance limitations and foreseeable misuse. Both performance limitations and foreseeable misuse rely on situational awareness [63] based on perception sensor data to identify safety risks. While ISO 26262 primarily covered ADAS functions, the SOTIF concentrates on HAD.

2.2.5 Security and safety

It should be noted that security and safety are distinct items. Safety is the prevention of harm to humans caused by the system or environment, whereas security is the prevention of malicious human activities using the system. The automotive industry has been only considering security in a physical sense, such as protection against breaking into a car and car theft. However, in a fully autonomous car, new technologies such as Vehicle-to-everything (V2X) communication and smart sensors are introduced, which gives attackers new means of intentional malicious manipulations. Although our study does not focus on security, it should be considered during the safety-critical system design. In particular, a malicious attacker should be restricted from physically harming the passengers and road users. To compensate for the limited exposure of security topics in ISO 26262, the ISO standardization body is working on a draft for security in road vehicles ISO/SAE CD 21434 "Road vehicles cybersecurity engineering".

2.2.6 Safety mechanisms and processes

Besides safety-related standards, state-of-the-art contains many safety mechanisms to observe, analyze and response to faults and hazardous situations. For example, a built-in self-test (BIST) [78] is a popular hardware mechanism to detect faults in ICs. Other examples of safety mechanisms include watchdogs, error correction codes, redundancy with majority voting, etc. Note that besides mechanisms there exist many non-standardized processes to ensure quality and safety of a computer system, such as pair-programming, test-driven development, defensive programming, etc.

2.2.7 Scoping diagram of safety frameworks

Safety requires a holistic approach of different perspectives. To understand these perspectives we made Figure 2.5 with an overview of safety-related laws, standards and concepts, primarily inspired by the prior work from Carnegie Mellon University [44] and NXP safety community [46]. The color-coding distinguishes between laws or standards, drafts and concepts using blue, yellow and pink, respectively. Although the content of the diagram will never be complete and fully accurate, we attempt to categorize and position prominent safety frameworks relative to each other and identify the domain for our contribution.

In the diagram, we distinguish four major scopes:

1. Society
2. Human
3. System
4. Component

The society category identifies frameworks that operate in the society. Laws, standards and concepts, such as the California Car Accident law [12] and the Trolley problem [88] are popular in mass media but have a limited impact on the technology. Humans are heavily involved in the frameworks addressing operations, maintenance, and security. Furthermore, the component-level safety frameworks have been well-studied in literature and often do not involve the system as a whole. Societal, human, and component aspects of safety are not covered in this work, as we focus on the system-level design of an autonomous vehicle.

Interestingly, there are multiple approaches to system-level safety. For example, the Toyota Guardian [9] focuses on a seamless transition of vehicle control between the human and the autonomous vehicle itself. On the other hand, the safing gate from ANSYS SCADE [1] omits the human category and suggests using a rule-based safety net for Artificial Intelligence components. Furthermore, Intel/Mobileye's RSS (Responsibility-Sensitive Safety) [7] excludes both the human and rule-based mechanisms altogether and attempts to capture safety in terms of formal mathematics. The RSS authors identify the key challenge in automated driving to be the path planning and maintaining a safe distance to obstacles, while stating that perception can be properly tackled with heterogeneous sensor diversity and vehicle control is well covered with advanced control theory. The path planning is indeed challenging because it is essentially a multi-actor problem, where the ego-vehicle is not in control of other hard-to-predict road users. Overall, the system category exhibits highly diverse approaches to safety in autonomous driving and appears to be an active research field.

For fault monitoring in an engine controller, a consortium of car makers and Tier 1 companies, including BMW, Volkswagen, Porsche, Daimler and Audi, defined the E-Gas monitoring concept [89], which was later was extended for multicore processors in [33]. Essentially, E-Gas distinguishes three levels:

1. Level 1: functional level, which includes the function itself, such as engine control.
2. Level 2: function monitoring level, which focuses on detection of faults in level 1. In our work, we refer to it as a *function monitor*.
3. Level 3: controller monitoring level, which focuses on checking the controller and level 2. The controller level is split across the controller itself, as well as an external device (IC), such as a System-Basis Chip (SBC). The controller part of this level is referred to as a *platform monitor* in this work, while the external part of this level we call *vehicle-level safety mechanism*.

The E-Gas addresses several safety goals, among which the most stringent is prevention of unintended acceleration. According to the E-Gas concept, monitoring levels 2 and 3 contain Enable outputs, which can disable fuel injection into the motor. In case of a fault, the enable output is invalidated by any of the E-Gas monitors, resulting in absence of motor torque and, consequently, acceleration. In other words, the enable output implements the fail-silent behavior for the system, which satisfies the key safety goal.

Although this is proven in use industry standard primarily targeted engine controls, conceptually it is applicable to other automotive functions as well. In our work, we will derive a vehicle-level safety mechanism for autonomous driving from the E-Gas monitoring concept.

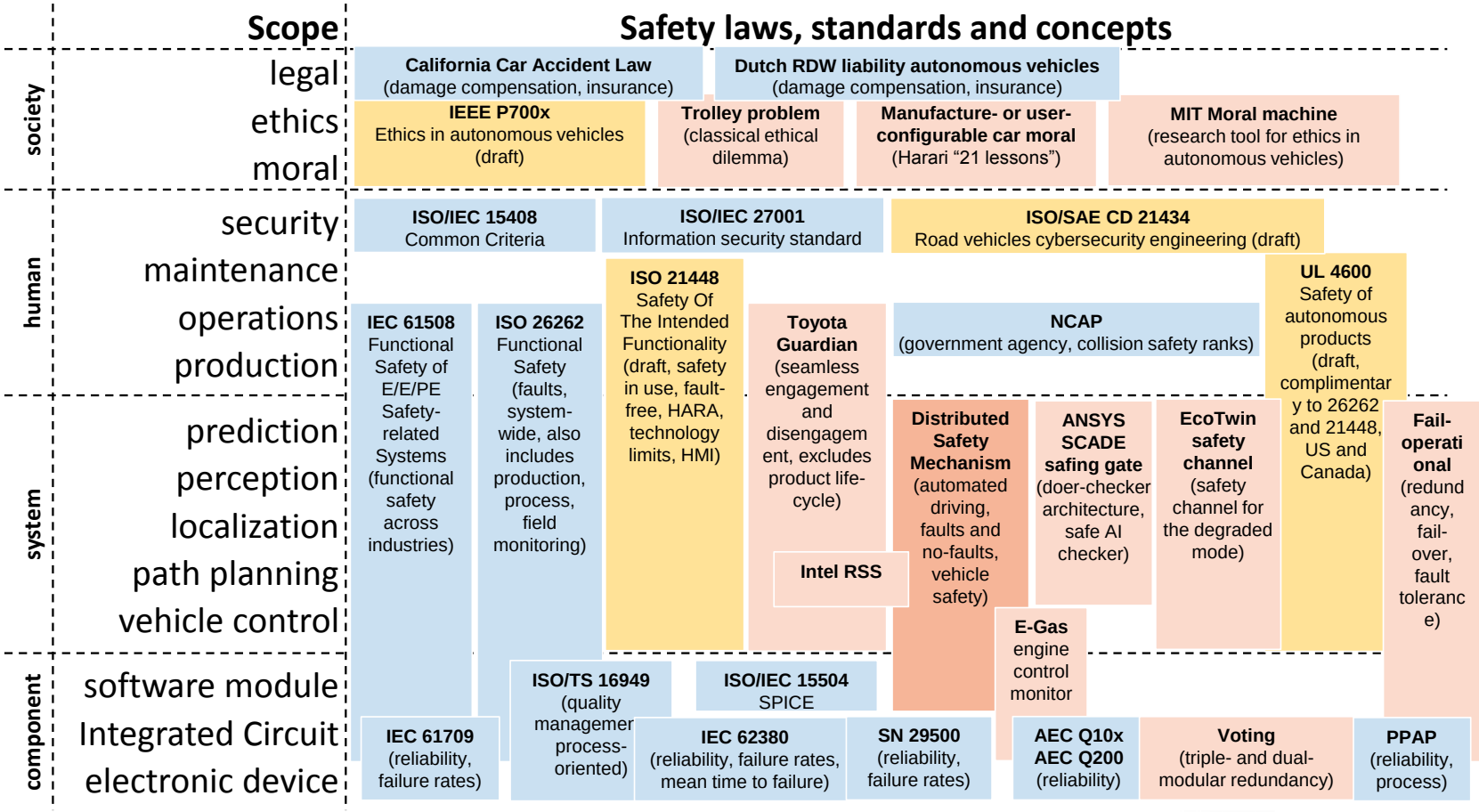

Figure 2.5: Safety laws, standards and concepts

California Car Accident Law [12], Dutch RDW liability autonomous vehicles [73], ISO 26262 [39], ISO 21488 [40], Trolley problem [88], User-configurable car moral [83], MIT moral machine [8], NCAP [85], Toyota Guardian [9], Intel/Mobileye RSS [7], ANSYS Scade safing gate [1], EcoTwin project [6], E-Gas [89], PPAP [86]

In general, this work will focus on the system-level safety, omitting the societal, human and component frameworks. In contrast to E-Gas and Intel/Mobileye RSS, we will attempt to address the full system scope. Having covered the automated driving application and basic safety concepts, in the next section we analyze the software middleware.

2.3 Software middleware

Middleware is a multipurpose software that lies between an operating system and applications running on it. A middleware provides services such as communication and data management for distributed software using a translation layer. Which eases the development process of a distributed application and provides robust communication.

2.3.1 Robot Operating System (ROS)

ROS [60] is an open-source software framework for robot software development, providing a middleware operating system running on top of a real operating system. ROS consists of a set of tools, libraries, and a communication model to simplify development. The communication model of ROS is a publish-subscribe pattern [23] where a ROS node can subscribe to messages which are routed through ROS topics to another ROS node. ROS also provides a request/reply interaction through services. A ROS service defines the set of request/reply messages where a ROS node can initiate a remote procedure call and awaits the reply. ROS is designed to run on modern operating systems such as Linux, Windows, and Mac, it is not possible to run ROS on an RTOS. Furthermore, ROS works with a centralized master, where communication with ROS nodes is done using the TCP/IP protocol. These design choices of ROS do not make it suitable for computing safety-critical tasks, therefore, the designers of ROS developed Robot Operating System 2 to overcome these shortcomings. ROS2 is designed with requirements such as; real-time computing, portability, functional safety, security, and, fully distributed computing in mind. With ROS2 it is possible to run a ROS node on a certified automotive grade safe RTOS.

2.3.2 Data Distribution Service (DDS)

The Data Distribution Service (DDS) [57] for real-time systems is an Object Management Group middleware standard that provides a decentralized distributed communication service using the publish-subscribe pattern with discovery functionality.

ROS2 uses DDS for its communication services which resolve most of the issues as discussed in section 2.3.1. Another goal of ROS2 is to make an abstraction layer between ROS2 and DDS which hides much of the complexity of DDS. Furthermore, it provides the possibility to use different DDS implementations when required for example using a DDS implementation certified for ISO 26262 ASIL-D can be used.

The Object Management Group also published some extensions to the DDS standard: DDS for eXtremely Resource Constrained Environments (DDS-XRCE) [54] which defines a protocol for resource constrained devices to allow them to participate in the DDS network. DDS Security an extension to the DDS standard focusing to mitigate the following threats: unauthorized subscription, unauthorized publication, tampering & relay and insider attacks. A new upcoming extension to the DDS standard is DDS for Time-Sensitive Networking (DDS-TSN) which incorporates the IEEE Time-Sensitive Networking set of standards [31] used in automotive Ethernet into DDS.

2.4 Operating Systems

An operating system is a computer software that manages the hardware components, memory, and provides an interface to schedule component software. Automotive vehicle software consists of various components that have to run on one more operating systems. There are many different operating systems providing different functionality and possesses different availability and reliability. This section gives an overview of the state of the art operating systems that can run automotive vehicle software but also studies various component isolation techniques used in operating systems.

2.4.1 Xenomai: Time-critical applications on a Linux-based platform

Xenomai is a Linux realtime extension adding an RTOS to the Linux kernel, this is done by modifying the interrupt handler to fully preempt the Linux kernel, during an interrupt the signals goes first through the Xenomai RTOS. Here decisions are made whether the interrupt is meant for Xenomai and passes it to the real-time application or it passes the interrupt through to the Linux kernel. Real-time Xenomai applications must implement both Linux glibc interface and the Xenomai libpthread_rt interface have real-time event support. Xenomai is suitable in mixed criticality ADAS systems and study [21] shows predictable real-time behavior.

2.4.2 Operating-system-level virtualization

Operating-system-level virtualization, also known as "containerization", is an operating system feature that allows the isolation of computer applications thereby reducing the number of required computing resources. Containing them in their own virtual environment where the application only sees their own file system and optional devices can be assigned to their environment. Containerization is generally faster than full virtualization because there is no operating system and hypervisor overhead. Docker [48] and Linux Containers (LXC) are popular implementations of containerization and are widely used in the industry. Since containerization is an operating system feature, applications are not fully isolated, therefore applications can still interfere with each other, for example a kernel call.

2.4.3 Hardware virtualization

Modern ARM CPUs with an ARMv8 instruction set include support for hardware virtualization [49]. On older ARM architectures virtualization had to be done through software virtualization which is expensive because it requires more computing resources caused by the software virtualization overhead [37], furthermore software virtualization does not provide good isolation since it cannot fully isolate the memory addressing between guest systems. The ARM virtualization extensions as stated in [49] provide multiple hardware improvements to reduce virtualization overhead [61] and enforces isolation between the guests address spaces.

CPU virtualization

Without the addition of the ARM virtualization extensions an ARM CPU had 2 exception levels: One for application mode (EL0) and one for kernel mode (EL1). The kernel mode exception level had more privileges than application mode so that a kernel can manage applications and the applications are isolated from each other. With the ARM virtualization extensions an extra CPU exception level has been added which has more privileges then kernel mode. This mode is called hypervisor mode (EL2), where it can arbitrate between different kernels. And these kernels are isolated from each other. Figure 2.6 shows the different exception levels of an ARMv8 CPU.

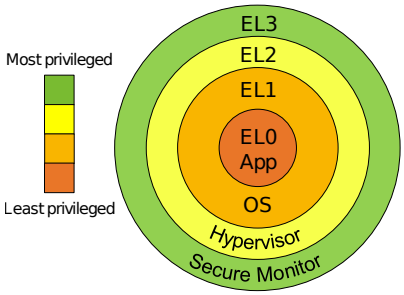

Figure 2.6: ARMv8 Exception Levels.

Memory virtualization Memory virtualization is already a common practice in modern operating systems. Applications that get launched in a modern operating system uses the virtual address space and the kernel uses physical address space. The virtual address to physical address conversion is accelerated by the memory management unit. For virtualization a kernel should not directly write to the physical memory otherwise it is possible to read and write memory from another kernel. The ARM virtualization extensions [49] add intermediate physical address support to the memory management unit. A virtualized kernel will use the intermediate physical address space and does not have a notion that is virtualized. A hypervisor will configure the mapping of intermediate physical address space to physical address space, Figure 2.7 indicates how the address space is partitioned.

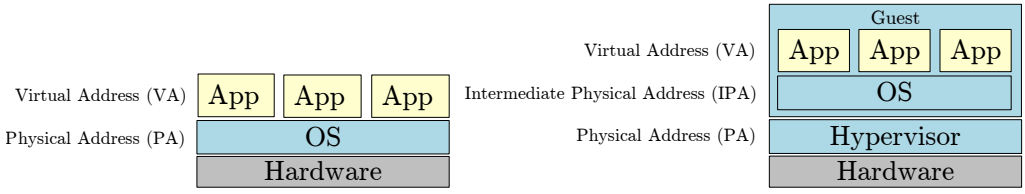

Figure 2.7: Memory virtualization

Interrupt virtualization

Interrupt virtualization is done by extending the interrupt controller with virtual interrupts allowing to trap the CPU into either directly to the kernel mode or hypervisor mode. The hypervisor kernel is responsible for configuring the ARM interrupt controller, furthermore the ARM virtualization extensions includes virtual timers for guest systems where hardware timers are translated into virtual timers which directly trap into guest system and therefore bypasses the hypervisor.

Hypervisors

A hypervisor is a kernel that allows virtualization. The hypervisor is responsible for managing and scheduling the guest systems. In this study we focus on hypervisor designed for the ARMv8 micro architecture, which means that a hypervisor for ARMv8 must implement the ARM virtualization extensions as listed above. A hypervisor can be classified into two types of hypervisors [59]. Type-1, a bare-metal hypervisor running directly on the hardware. Type-2, a hosted hypervisor that runs on top of an operating system. Based on studying the existing hypervisor architectures we observed that the two type classification of Popek & Goldberg (1974) [59] can be extended into the following subcategories:

- True bare-metal hypervisor that only implements a minimal subset of drivers for hardware virtualization.
- Microkernel based hypervisor where the device drivers are running on top the kernel as a service
- Monolithic hypervisor where all drivers are included inside the hypervisor.

See Figure 2.8 for a visualization of our extended hypervisor classification.

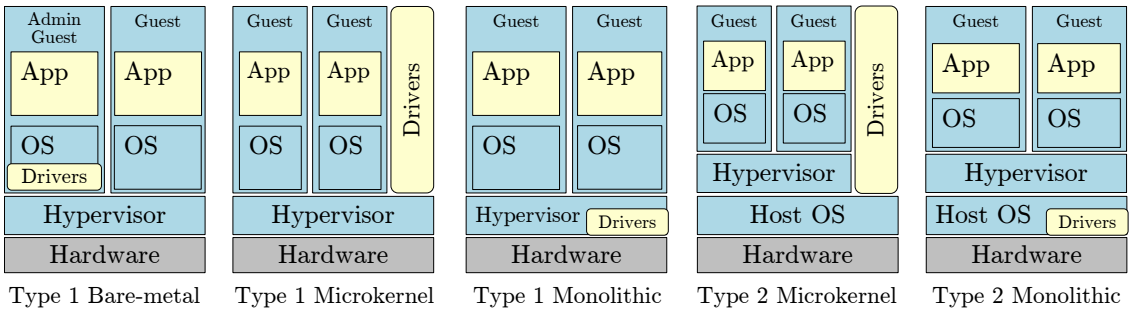

Figure 2.8: Extended hypervisor kernel classification

2.4.4 Siemens Jailhouse: Linux-based partitioning hypervisor

Jailhouse focuses on isolation and partitioning rather than virtualization [67]. Jailhouse is a type 2 hypervisor that runs on top of Linux. However, Linux is only used to initialize the hardware, guest systems on Jailhouse will get a 1:1 mapping to available hardware. This is done using CPU pinning where a dedicated CPU gets mapped to a guest, in contrast to other hypervisors in Jailhouse there is no CPU scheduling involved and therefore Jailhouse has low-overhead [71]. Figure 2.9 shows that a RTOS only runs on top of the partition layer and is not dependent on Linux.

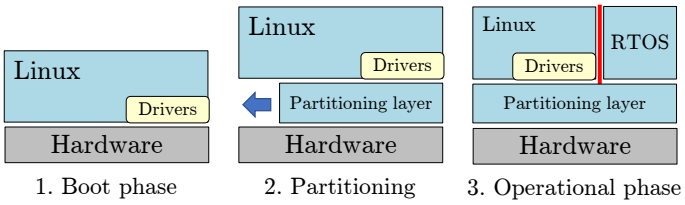

Figure 2.9: Siemens Jailhouse partitioning concept

2.4.5 Green Hills INTEGRITY RTOS and Multivisor hypervisor

Green Hills INTEGRITY is a commercial microkernel-based RTOS, specifically designed for safety-critical systems [42] and is ISO26262 [39] ASIL D certified. The microkernel architecture of

INTEGRITY isolates applications, device drivers, the file system, and networking capabilities. INTEGRITY provides a proprietary API and supports the POSIX API. INTEGRITY provides type 2 hypervisor functionality called Multivisor, which can virtualize Linux, Android, Windows, and QNX Neutrino operating systems.

2.4.6 QNX Neutrino RTOS and hypervisor

QNX Neutrino [45] is a commercial microkernel-based RTOS. Designed for automotive, medical, transportation, and industrial embedded systems and is ISO26262 [39] ASIL D certified. QNX Neutrino is POSIX-compliant and also implements its own API. Another product of QNX is the QNX hypervisor, which is a QNX Neutrino based type 1 hypervisor which can virtualize Linux, Android, and QNX Neutrino operating systems.

2.5 Automotive hardware

Modern vehicle hardware consists of complex mechanical, hydraulic and electronic systems. Figure 2.10 organizes the electronic systems functionality in three categories: sense, think and act. These functions are implemented in multiple electronic controllers, sensors and actuators, all of which are interconnected through in-vehicle networks, such as CAN, LIN, Ethernet, etc. It is important to note here, that vehicle electronics rely on distributed processing, where multiple processors in a SoC or on the in-vehicle network cooperate to perform a single function, such as engine control, electronic power steering or autonomous emergency braking. In this work we focus primarily on autonomy and powertrain subsystems.

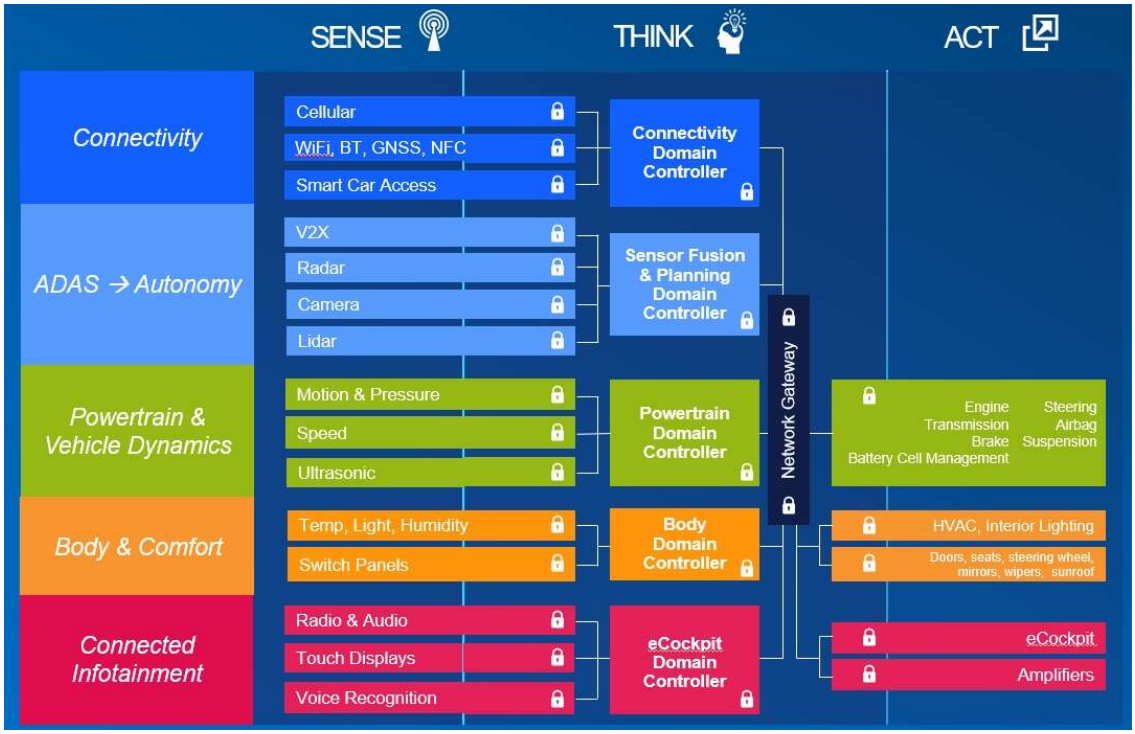

Figure 2.10: Sense, think and act categories of automotive electronics [91]

The powertrain subsystem compromises the main components that provide power to control the wheels. The powertrain subsystem is controlled by the powertrain domain controller which reads

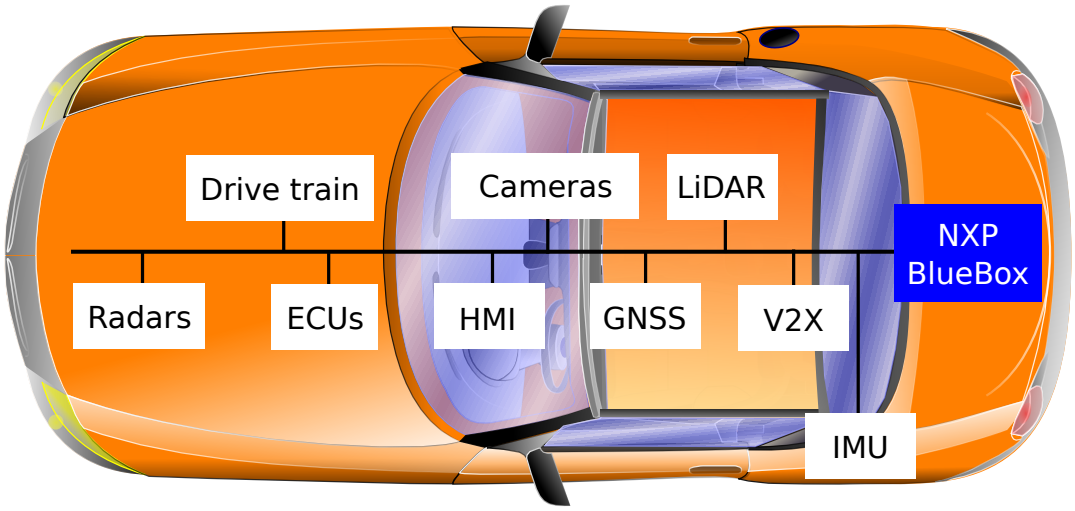

Figure 2.11: Hypothetical in-vehicle network for a autonomous car

the human control input. In a fully autonomous car the powertrain subsystem gets controlled by the autonomy subsystem, the planning domain controller calculates a path from sensor fusion data and sends an actuation signal to the powertrain. Sensor fusion is combining sensory from different sensors e.g. LiDAR, camera, radar such that resulting data yields higher accuracy than would be possible using the sensors individually. Figure 2.11 show a hypothetical in-vehicle network for a fully autonomous car. The existing in-vehicle network is extended with state of the art sensors required for autonomy which can be processed for example on the NXP BlueBox prototyping platform capable of the required performance, functional safety and automotive reliability for engineers to develop self-driving cars

Automotive research studies often utilize a hardware-in-the-loop prototyping platform, which is more convenient and safer than a complete vehicle. For our study, we selected the NXP BlueBox shown in Figure 2.12, which is a HAD prototyping platform developed by NXP semiconductors to build HAD systems, assess their performance requirements and experiment with functional safety mechanisms. Internally, the NXP BlueBox incorporates several networks and point-to-point interfaces, such as the Automotive Ethernet and PCIe, interconnecting three major SoCs with different safety integrity levels, as shown in Figure 2.13. These processors and networks allow for evaluation of safety mechanisms for reliable distributed processing, which was pointed out earlier in this section. Finally, the BlueBox includes automotive IOs to connect to the vehicle actuators and sensors, as well as an solid state drive to record data for post-processing.

Below is a brief explanation of major SoCs in the BlueBox:

1. S32R274: automotive-grade safety radar processor contains two Power [65] cores e200z7 and one Power e200z4 lock-step core, supporting ISO 26262 up to the highest ASIL D.
2. S32V234: automotive-grade vision processor contains four ARM Cortex-A53 CPU cores, a ARM Cortex-M4 core and an APEX machine learning accelerator. it supports ISO 26262 up to the high ASIL C.
3. LS2084A: powerful processor contains eight ARM Cortex-A72 CPU cores and according to ISO 26262 it is a quality managed device.
4. SJA1105: three automotive grade Ethernet switches with support for Audio-Video Bridging and Time-Sensitive Network protocols.

Figure 2.12: NXP BlueBox HAD prototyping platform

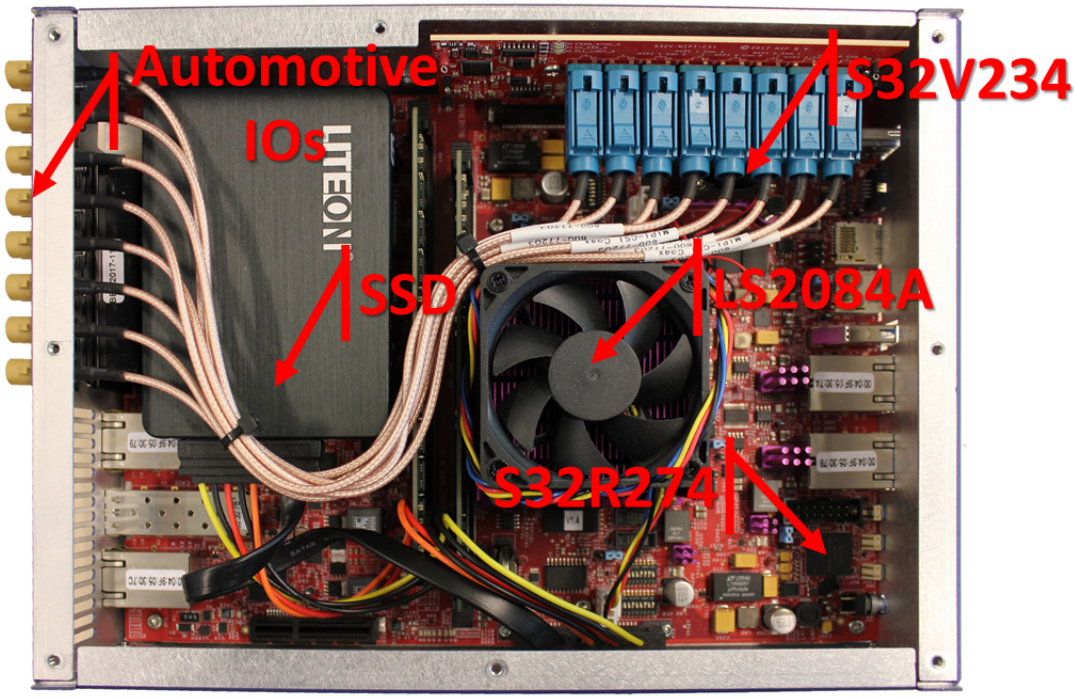

Figure 2.13: NXP BlueBox internals

Chapter 3

Problem statement

Based on the state of the art study we identified the following problems in autonomous vehicle software:

1. Automated driving relies on distributed processing with up to 80 ECUs [30]. In particular, application-specific hardware accelerators and multiprocessors are used to satisfy both high performance demands and efficiency of automotive applications. A central safety monitor is simple from the architecture perspective, however, for such a distributed system it has high reaction time and low observability. By observability here we mean the ability to detect faults and read hardware and software state of the distributed system components in sufficient detail.

2. Automated driving uses various sensors to analyze the surroundings of the car coupled with Artificial Intelligence algorithms for object detection and classification [56]. Both sensor and AI technologies have performance limitations, which may result in hazardous situations even when the system operates without any faults. Furthermore, as described in the SOTIF standard draft [40], foreseeable human misuse may jeopardize the passenger safety.

3. There is an ongoing trend in the automotive industry to consolidate several functions in a single device to reduce the vehicle cost [25]. Unfortunately, cascading failures between two or more functions in a consolidated automotive system can lead to a violation of safety requirements. Therefore, ISO 26262 [39] demands freedom from interference to avoid such violations.

In this project we aim at designing a new safety mechanism that improves safety of autonomous vehicle software in the three challenging areas listed above.

Chapter 4

Research methodology

We can distinguish between formal mathematical [81] and experimental methods for the investigation of solutions to the formulated problem statement. The experimental method [79] allows for more accurate and industry-relevant analysis on representative prototypes. In the experimental method one of the key safety goals is to ensure that the autonomous vehicle avoids colliding with obstacles. In our study we chose the NXP automotive prototyping platform, as described in Section 2.5.

Our research methodology included the following phases:

1. Survey state-of-the-art autonomous vehicle software to formulate design goals for the safety mechanism.
2. Design a prototype of the safety mechanism for autonomous vehicle software for the reality check of the solution.
3. Conduct a fault injection experiment to test operation of the safety mechanism.
4. Qualitatively analyze applicability of the solution to similar systems.

Chapter 5

Safety Mechanism Design

An autonomous vehicle has to reach the destination without creating hazards, this can be defined as a safety goal: the output commands do not command a path that collides with an obstacle. As identified in the problem statement in chapter 3 we propose to design a safety mechanism that helps in achieving this safety goal by increasing the safety of autonomous vehicle software. Section 5.1 lists the software design goals. Section 5.3 describes the architecture of the safety mechanism and discusses the design choices made.

5.1 Software design goals

Our target is to create a safety mechanism for a HAD software framework running on the NXP BlueBox prototyping platform. The problem statement in chapter 3 describes 3 challengers that our safety mechanism intends to address. For our safety mechanism design we introduce 3 corresponding goals that are detailed below:

G1 The safety mechanism should be able to detect relevant faults and react to them before occurrence of the hazard.

G2 Proper handling of fault-free hazardous scenarios requires situational awareness, as stated by the SOTIF standard. The HAD software state contains sufficient information to identify hazardous situations. Therefore, the safety mechanism should be able to read HAD software state internals.

G3 An automated driving system should isolate its individual functions to avoid cascading failures by utilizing inter-SoC or intra-SoC partitioning.

5.2 Communication interfaces

For goal **G2** a mechanism should be used to read the HAD software state internals. As identified in chapter 2 modern HAD software frameworks uses distributed high level protocols such as ROS which can be used on top of Automotive Ethernet. As described in Section 2.3.1 ROS uses a publish-subscribe model for communication which makes it very suitable for monitoring by subscribing to a topic of the autonomous vehicle software while having minimal impact on the behavior of the application. Other alternatives are using other communication interfaces such as Controller Area Network (CAN), Local Interconnect Network (LIN) or FlexRay to read state internals. However, all these communication interfaces lack the bandwidth required for autonomous vehicle software [20] and are mostly used for the low bandwidth safety critical data such as control output as described in 2.1.1.

However, using ROS for the safety mechanism would also not be feasible as stated in Section 2.3.1 ROS is designed for the Linux operating system, this problematic since it means it is

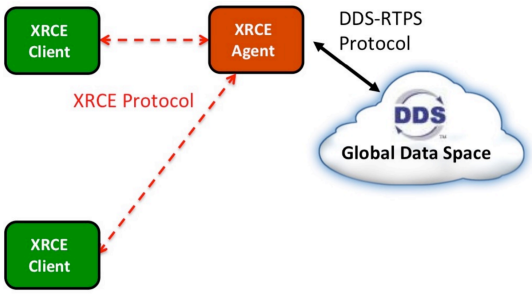

Figure 5.1: DDS-XRCE clients communicating through DDS-XRCE agent [54]

unable to run on an ISO 26262 ASIL-D certified hardware and is not suitable for computing a safety-critical tasks which are required for our safety mechanism. ROS2 overcomes some design problems in ROS, but at this moment it only runs on Linux, Windows, or Mac. However there is an approach called micro-ROS [52] which aims at running ROS2 on microcontrollers. Unfortunately, micro-ROS is still a work in progress and is not usable yet. Another alternative is to use the DDS middleware directly, as described in 2.3.2 ROS2 is built on top off DDS which makes it possible to communicate with ROS2 nodes through DDS. Unfortunately, DDS implementations tend to be larger than what typical microcontrollers memory can handle. For example, the eProsima Fast-RTPS DDS [14] implementation version 1.6.0 on a x86 [34] Ubuntu 18.04 machine has a size of 32MB, and typical microcontroller memory varies between 4KB and 4MB.

For our safety mechanism we propose to use DDS for eXtremely Resource Constrained Environments (DDS-XRCE) [54] which is a trimmed down version of DDS specifically designed for micro controllers with a low clock frequency and small memory. DDS-XRCE retains full DDS functionality through a DDS-XRCE agent as shown in Figure 5.1.

5.3 Distributed Safety Mechanism Architecture

The architecture of the distributed safety mechanism is based on the E-Gas [89] 3-level monitoring concept described in the state-of-the-art Section 2.2.7. Figure 5.2 shows the architecture of the E-Gas concept where a function (level 1) is monitored by a function monitor (level 2), the controller monitor (level 3) monitors the ECU internals, and an external monitor verifies this monitor using a challenge-response process.

The Distributed Safety Mechanism (DSM) concept extends the E-Gas concept by using the DDS middleware for distributed communication between the different levels. The DSM concept shown on Figure 5.3 consists of 2 data paths: a function path where, input and output data will be published and, a diagnostics path where all diagnostics data will be published. The data paths can be cost-effectively implemented on-chip as shown between L1 Function and L2 Function Monitor, as well as Ethernet network communication as shown between L1 Function and L3 Vehicle-level Safety Mechanism. On top of the fail-silent enable output of the E-Gas concept, our DSM in includes a safety mechanism, which can take over vehicle control with situational awareness using function data paths.

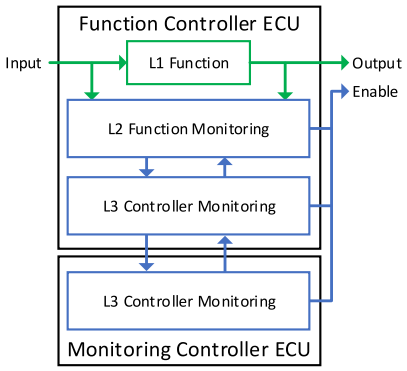

Figure 5.2: E-Gas concept [89]

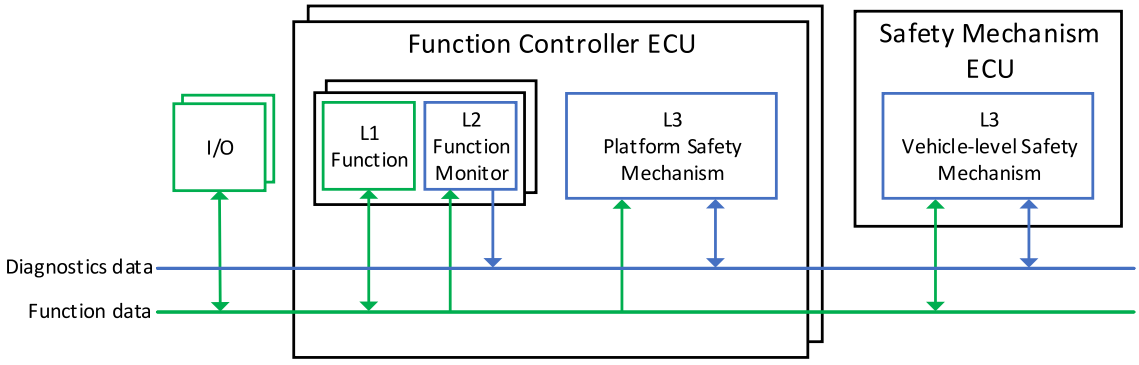

Figure 5.3: DSM concept

Figure 5.4 shows a system overview of our distributed safety mechanism on the NXP BlueBox prototyping platform. Table 5.1 shows the comparison between the traditional E-Gas components and the the distributed safety mechanism components used in our system. In Section 5.3.1 we will explain why we choose for the Apollo HAD framework. Section 5.3.2 and 5.3.3 reasons about the partitioning. Section 5.3.4 details the distributed health monitors to detect relevant faults. Finally, Section 5.3.5 will present the distributed safety mechanism.

Table 5.1: Comparision between E-Gas components and distributed safety mechanism components.

E-Gas level	**E-Gas component**	**DSM component**
Level 1	Function	Apollo Components
Level 2	Function monitor	Function Health monitor
Level 3	Controller monitor	OS Platform Monitor
Level 3	Controller monitor	Hypervisor Platform Mechanism
Level 3	Physically independent controller monitor	Vehicle-level Safety Mechanism

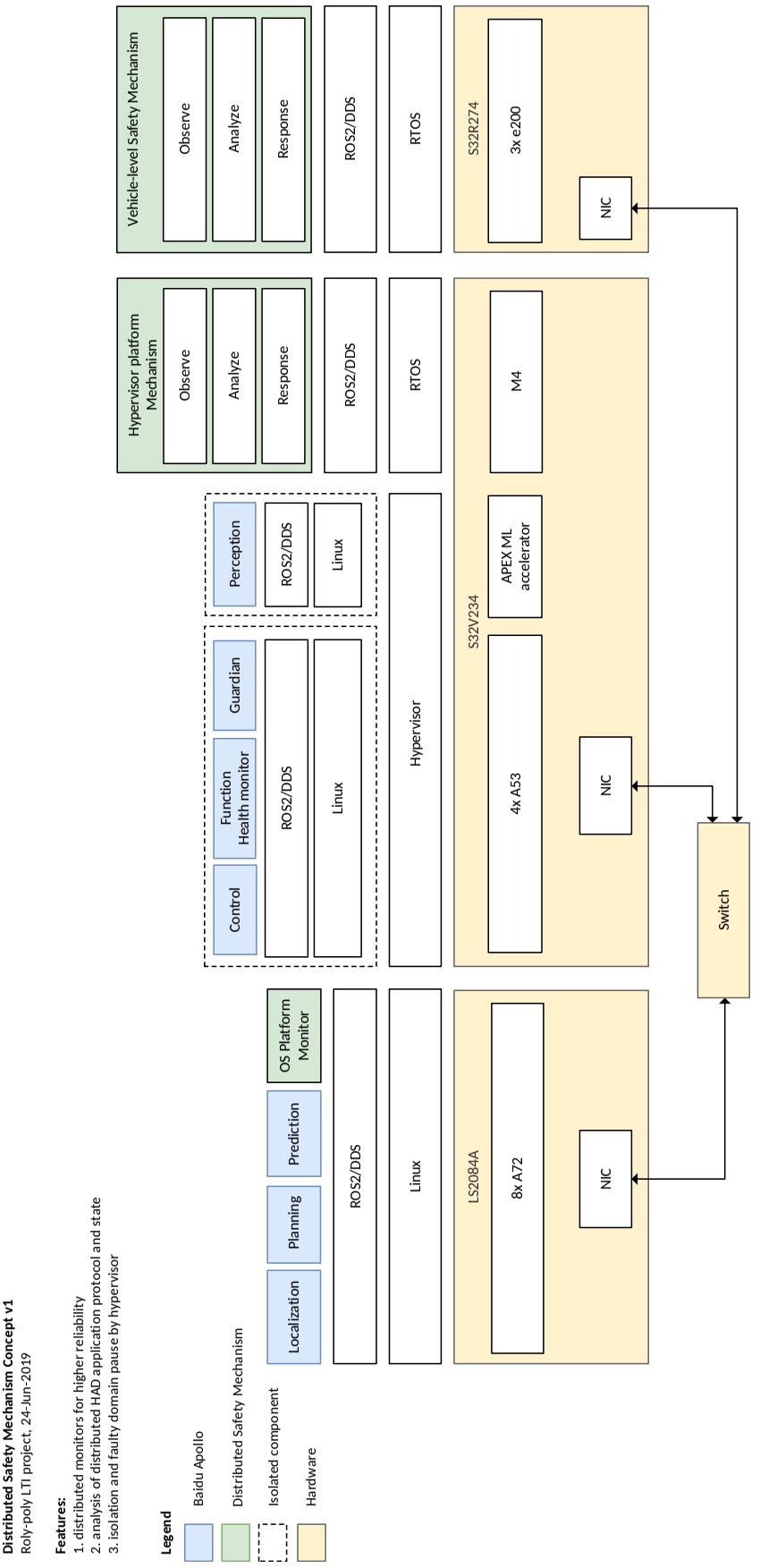

Figure 5.4: DSM concept on NXP BlueBox

5.3.1 Apollo HAD software framework

To satisfy goal **G1** and **G2** a functional HAD software framework is required. Table 2.1 a HAD software framework feature comparison has been made. From this comparison we chose for the Apollo framework because:

- Commercial support from the industry
- Vehicle-Tested on public roads
- Monitoring capabilities that could be extended
- ROS based tooling and communication middleware allows for rapid prototyping

5.3.2 Inter-SoC partitioning of the HAD framework

Ensuring freedom from interference is of great importance to goal **G3**. However, ensuring freedom from interference is hard since in ideal circumstances every software component should be fully isolated from each other. However, this is a trade-off between latency, throughput and costs. For our safety mechanism design, we decided to partition the Apollo HAD software framework on to the NXP BlueBox platform as shown in Section 2.5. A HAD software framework consists of a localization, perception, prediction, planning, and control module as discussed in Section 2.1.1. The localization, prediction, planning module are partitioned to the LS2084A SoC, the control and monitor module are partitioned to the S32V234 SoC. Finally our safety mechanism will run on the S32R27 SoC. Apollo also has its own safety mechanism called guardian, we have partitioned this module to the S32V234 SoC. Unfortunately, during testing of the guardian component, it seems that version 3.0 of Apollo only implements some basic checks and does not provide any active safety mechanism. The perception module cannot run on BlueBox platform because the S32V234 APEX machine learning accelerator [53] is incompatible with the Apollo machine learning algorithms. Since machine learning is not in the scope of this research, we chose for a workaround using a desktop PC with a machine learning accelerator that will execute the Apollo machine learning algorithms used in the perception module.

5.3.3 Intra-SoC partitioning using a hypervisor

As shown in Section 5.3.2 there are still some software modules running on the same SoC which will violate goal **G3**. To ensure freedom from interference on a single SoC we propose the use of a hypervisor.

A hypervisor had to be selected which can run on the S32V234 SoC in the BlueBox, which requires that the hypervisor implements ARMv8 hardware virtualization. Table 5.2 shows a list of feasible hypervisors that can run on the S32V234, classifying them into hypervisor type and porting effort. Type 2 hypervisors have low porting effort because the drivers are already ported to the Linux kernel and the hypervisor runs on top of it. A type 1 bare-metal requires medium porting effort where only CPU initialization and the serial communication device must be ported to the hypervisor. Type 1 Monolithic and Type 1 Microkernel hypervisors require considerable porting effort since all device drivers must be ported to the hypervisor. We chose for the Xen hypervisor [68] to implement on the S32V234 SoC because even though porting effort is harder than a type 2 a type 1 bare-metal hypervisor will yield better isolation because the common cause failure of a failing Linux operating system is removed as shown in Figure 2.8.

Hypervisors also provide domain pausing functionality, which can be used to implement fail-silent behavior [24]. This means when a fault occurs in a domain then the safety mechanism can pause this domain immediately ensuring no failure occurs because of faulty data. Furthermore, it ensures that the safety mechanism can take over the functionality of the domain without having collisions with the data output. However, it should be noted that the non-atomic effects of pausing a domain and the SoC peripheral interactions are unknown.

Table 5.2: Hypervisor comparison.

Name	Type	Porting effort
Xen hypervisor [68]	Type 1 Bare-Metal	Medium, only the bare metal hypervisor
L4Re Micro-Hypervisor [74]	Type 1 Microkernel	High all drivers must be ported
QNX Hypervisor [45]	Type 1 Microkernel	High all drivers must be ported
Xvisor [58]	Type 1 Monolithic	High all drivers must be ported
Linux KVM hypervisor [27]	Type 2 Hosted	Low, Linux base will be used
GHS INTEGRITY Multivisor [36]	Type 2 Hosted	High all drivers must be ported
Siemens Jailhouse hypervisor [67]	Partitioning (2.4.4)	Low, Linux base will be used

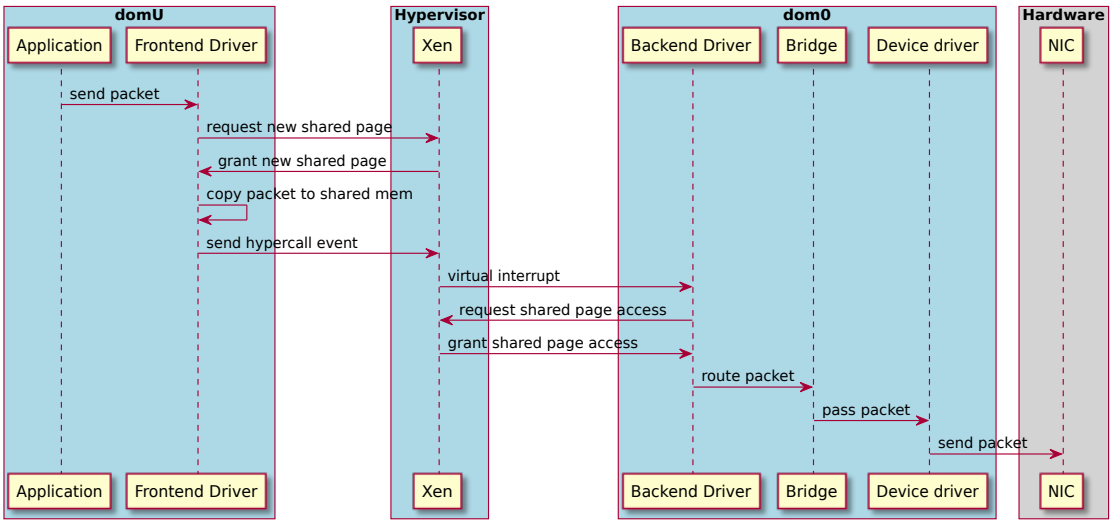

Figure 5.5: Xen network para-virtualization

For the network virtualization however, it is well defined, the Xen hypervisor uses a para-virtualized network device that uses a frontend driver in the guest domain (domU) and a backend driver in the host domain of the Network Interface Card (NIC). The hypervisor provides a shared memory page between the frontend and backend driver to send packets from the guest domain to the NIC, the full interaction is shown in Figure 5.5. These network drivers support pause functionality and do not suffer from deadlocks and starvation.

5.3.4 Distributed health monitors

To satisfy goal **G1** relevant faults have to be observed. A health monitor running on the same multicore or SoC can quickly observe details of the system components. Contrast with a remote health monitor: either observes component output only or requires modifications and correct operation of the component. The Apollo HAD framework provides a function health monitor that monitors the functions of the framework as described in 2.1.2. However, the Apollo monitor lacks some features: there is no support for monitoring components on a distributed system and therefore some monitor data is missing, it suffers from interference other Apollo components and can fail due to cascading failures which violates goal **G3**, SoC monitoring is limited to CAN bus status and disk space monitoring, no SoC hardware monitoring features such as CPU self-test and memory error rate are monitored.

Therefore, we introduce distributed health monitors that should run on each SoC on the NXP BlueBox Prototyping platform: The LS2084A runs the Linux operating system and a platform monitor should observe the Linux sysfs interface [50] to get information about device status and

the system bus and the Linux proc interface [11] should be observed to get CPU, memory, and process information. The NXP S32 family of SoCs have a Fault Collection and Control Unit (FCCU) which can be used for hardware monitoring. On the S32V234 the Xen Hypervisor should be monitored, this should be done using the application programming interface Xen provides. The hypervisor platform monitor should collect data about: CPU usage, memory usage for each isolated domain and the status of each domain e.g. paused, running, or destroyed. Ideally, the monitoring of the Xen hypervisor should be done on the M4 CPU in the S32V234 SoC. However, due to time constraints, we have decided to monitor the Xen hypervisor in a separate domain on the hypervisor. It should be noted that this workaround violates goal **G3**. The distributed monitors publishes the monitored status information using DDS middleware to the diagnostics path, so that all distributed monitors can monitor each other.

5.3.5 Distributed safety mechanism

The previous section described the ability to detect relevant faults for goal **G1** through monitoring. In this section, the distributed safety mechanism will be presented. The Vehicle-level Safety Mechanism will be a DDS participant which runs on the S32R274 safety microcontroller. By utilizing DDS all information from the Apollo function monitor, OS platform monitor, and hypervisor platform mechanism can be processed. Furthermore, DDS allows the Vehicle-level Safety Mechanism to monitor the communication between the Apollo HAD framework functions. The control module will be observed by the Vehicle-level Safety Mechanism to check for relevant faults as described in goal **G1**. Furthermore, according to SOTIF the mechanism will analyze the HAD state in the absence of system faults to detect potential hazards as needed for goal **G2**. An example of a hazardous situation without faults is when a pedestrian is approaching the driveway, which potentially can lead to a collision without system faults. A simple mitigation of the associated SOTIF risk is to slow down and warn the driver.

The Vehicle-level Safety Mechanism consists of an observer, analyzer and a response task: The observer task implements the DDS-XRCE [14] stack which processes the incoming DDS data and passes them to the analyzer task. The analyzer task summarizes the autonomous vehicle software to faulty and non-faulty state and should perform analysis of hazardous scenarios according to SOTIF. The response task also implements the DDS-XRCE stack to publish data to DDS topics. When a faulty state occurs in the autonomous vehicle software, the Vehicle-level Safety Mechanism response module will decouple the existing control module from vehicle control. The Vehicle-level Safety Mechanism will take over control and issues a safe stop.

For goal **G1** the reaction time is of great importance. To ensure a low reaction time for the distributed safety mechanism, communication paths must be as short as possible. This is done by distributed safety mechanisms, an example is the hypervisor safety mechanism. The hypervisor safety mechanism monitors the hypervisor domains. When a fault occurs the hypervisor safety mechanism signals the DDS domain but also immediately pauses the faulty domain to shorten reaction time. The Vehicle-level Safety Mechanism receives the fault signals and handles recovery further.

Chapter 6

Experimental evaluation of the safety mechanism

In Section 6.1, we explain the need of an automated driving simulator for experimental evaluation of the distributed safety mechanism and we discuss our choice of the automated driving simulator. In Section 6.2, we present our experimental setup. In section 6.4, we present the scenario that will be used for evaluation. Finally, in Section 6.5, we analyze the effect of, our safety mechanism in the case of a fault injection into the HAD framework.

6.1 Comparison of AD simulators for End-to-End verification

To evaluate our safety mechanism the HAD framework should be tested in a realistic environment. Using a real car in an urban area will expose the passengers of the car to unnecessary risk, the costs are high, and repeatability is very hard. Table 6.1 shows an overview of different evaluation methods to evaluate the safety mechanism. Using a real car compromises safety and MATLAB [84] simulation is not realistic enough for testing the safety mechanism, therefore, we choose to use end-to-end world simulation. A HAD framework as described in Section 2.1.1 requires sensory input such as cameras, lidars, and radar to produce an actuation output to control a car. To simulate the sensory input a virtual world, location, and an actuation signal is required.

Automated driving simulators implement Hardware-in-the-loop (HIL) simulation where the outputs of a HAD framework will control the car in a virtual world where tests such as collisions can be conducted without compromising safety. Furthermore, automated driving simulators can also provide scenario testing facilities to ease the process of testing reproducible scenarios. We have compiled a list of state of the art automated driving simulator in Table 6.2.

From this comparison, we choose to use the LG SVL Simulator [15]. Because it can simulate all the sensors required for the Apollo HAD framework and provides a scenario testing API. Furthermore, the LG SVL simulator provides full integration with the Apollo HAD which minimizes the effort of setting up the End-to-End verification.

Table 6.1: Evaluation methods for testing automotive software

Methodology	**Realism**	**Safety**	**Cost efficiency**	**Repeatability**
Driving a car in a urban area	+++	-	-	-
Driving a car on test track	++	+	-	+
MATLAB simulation	+	+++	+++	+++
End-to-End world simulation	++	+++	+++	+++

Table 6.2: AD Simulator feature comparison

Name	Lidar	Radar	Camera	Traffic	ROS interface	Scenario tester
Carla Simulator [28]	✓	✗	✓	✓	✓	✓
Gazebo [43]	✓	✓	✓	✓	✓	✓
LG SVL Automotive Simulator [15]	✓	✓	✓	✓	✓	✓
Microsoft AirSim [64]	✓	✗	✓	?	✓	✓
Udacity driving car sim [72]	✗	✗	✗	✗	✗	✗
TORCS simulator [90]	✗	✗	✗	✓	✓	✗
SUMO [19]	✗	✗	✗	✓	✗	✓
ANSYS [70]	✓	✓	✓	?	?	✓
TASS PreScan [17]	✓	✓	✓	✓	?	✓
dSpace [13]	✓	✓	✓	✓	?	✓
AVSimulation SCANeR [4]	✓	✓	✓	✓	?	?
Apollo Game Engine Based Simulation [3]	✗	✗	✓	✓	✓	?
NVIDIA DRIVE Constellation [10]	✓	✓	✓	✓	?	?

6.2 Experimental setup

The experimental setup consists of an NXP BlueBox and an Alienware x86 [76] workstation. Figure 6.1 shows the interaction of the experimental setup, as described in Section 6.1 we choose for the LG SVL simulator which runs on the Alienware and provides sensory input for the Apollo HAD framework running on the NXP BlueBox. As described in Section 5.3.2 the perception machine learning algorithms will run on the Alienware which has support for hardware acceleration. Figure 6.2 shows the modified distributed safety mechanism for use with our experimental setup.

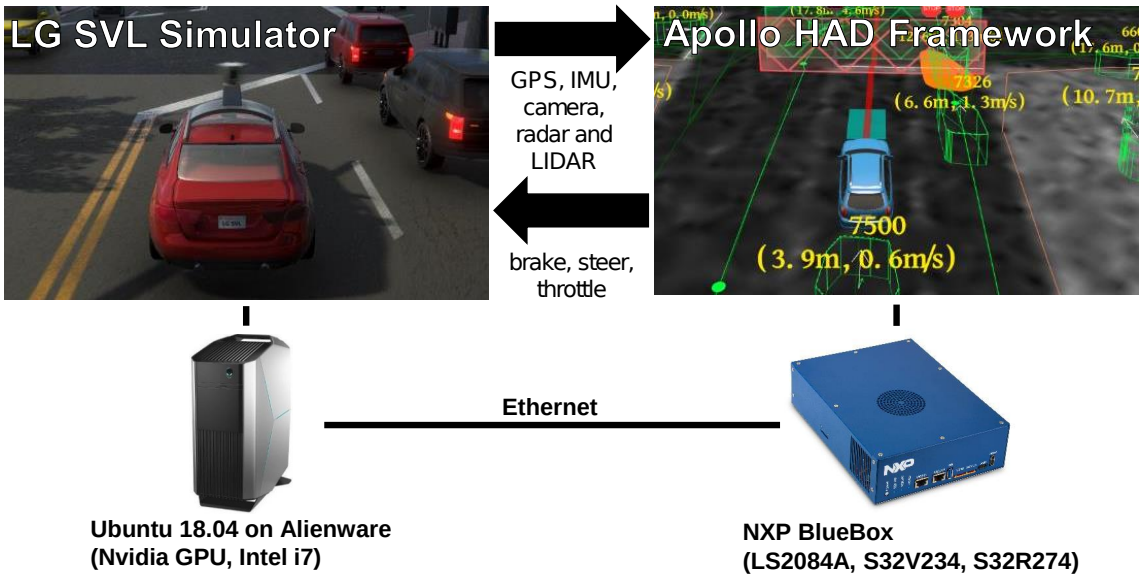

Figure 6.1: Experimental setup

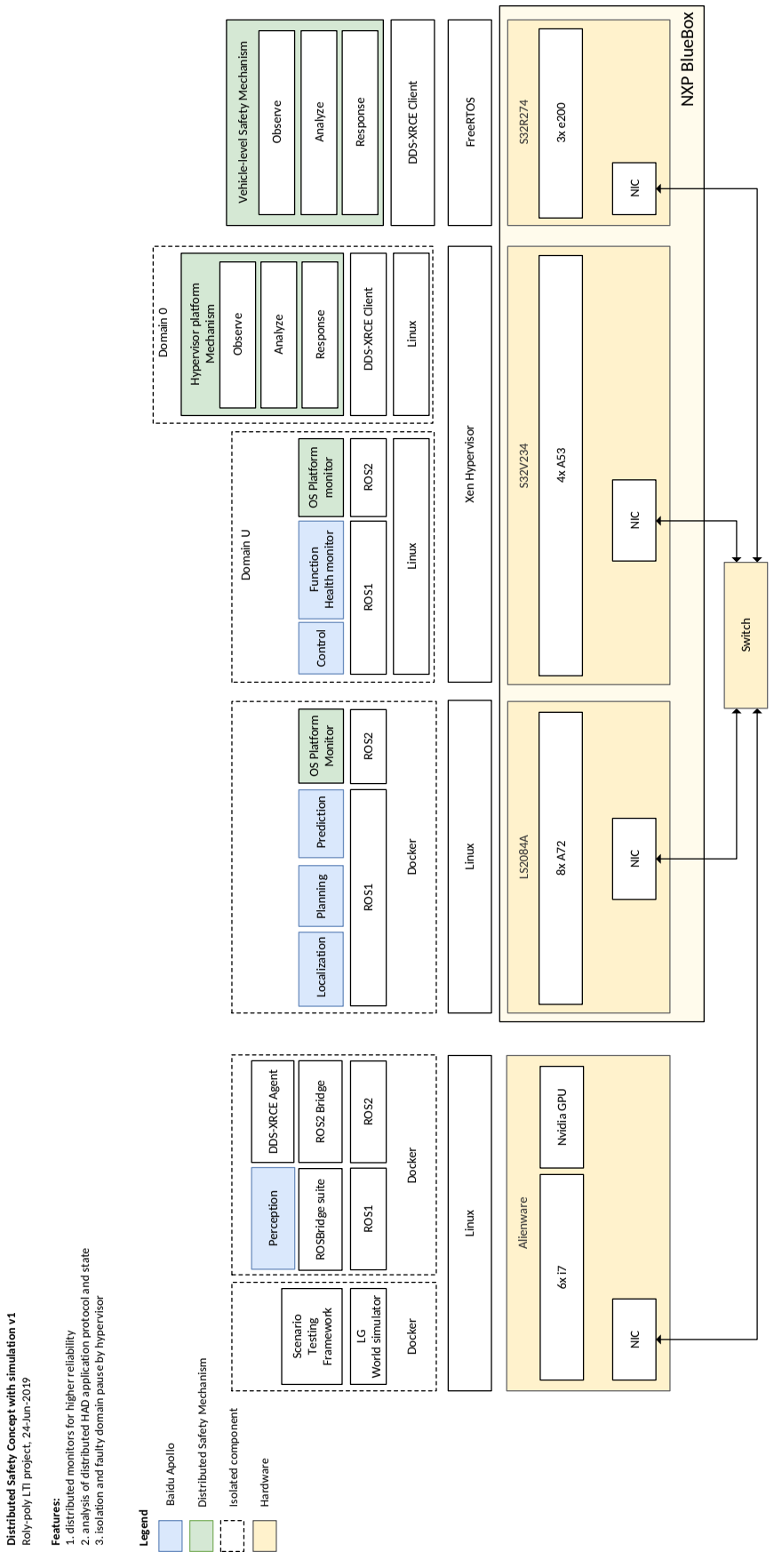

Figure 6.2: Distributed Safety Mechanism for Evaluation

In comparison to the original distributed safety mechanism as shown in Figure 5.4. The experimental setup uses both ROS1 and ROS2 this is because Apollo version 3.0 does support ROS2 nor DDS. Our solution for this was using the Rosbridge suite, which acts as a bridge between the ROS1 and ROS2 domain which allows seamless communication between ROS1 and ROS2 nodes. Apollo version 3.5 uses the CyberRT middleware which is compatible with DDS, unfortunately during this study, Apollo 3.5 did not support ARM-based platforms therefore, we choose to use Apollo 3.0.

Furthermore, the experimental setup is using a scenario testing framework that communicates with the LG SVL simulator API and the Apollo ROS nodes to create scenarios, execute them, and validate whether the vehicle caused any collisions and the destination waypoint of the scenario is reached on time.

6.3 Performance characteristics of experimental setup

With the experimental setup complete, performance characteristics have been gathered to give insight how state of the art HAD software frameworks and middleware performs on the NXP BlueBox in terms of bandwidth, latency, and message rates. This information can validate performance requirements of S32R274 to execute the vehicle-level safety mechanism. Our latency measurements on S32R expose the delays of 8.36ms, suggesting this SoC is capable of attaining the 100Hz message rate required for the vehicle-level safety mechanism.

6.3.1 Apollo HAD framework bandwidth

As stated in Section 5.3.1 we have chosen to use the Apollo HAD framework for our experimental setup. With the experimental setup in place we have profiled the Apollo ROS topics and visualized them using the same functional architecture introduced in Section 2.1.1. Figure 6.3 shows the functional components and their bandwidth usage and message production rate. The profiling gave interesting insight into the internal update rate of the Apollo HAD which is 10 Hz. Which seems to be low but do consider that version 3.0 of Apollo is designed for closed venue autonomous driving with a speed limit of 50 km/h. Therefore, it should be noted that more advanced functionality requires a higher message rate and bandwidth requirements. However the message and rate of 100Hz with a bandwidth of 50KB/s to control the car will probably will not change, and is within the specification the S32R274 performance to execute the vehicle-level safety platform.

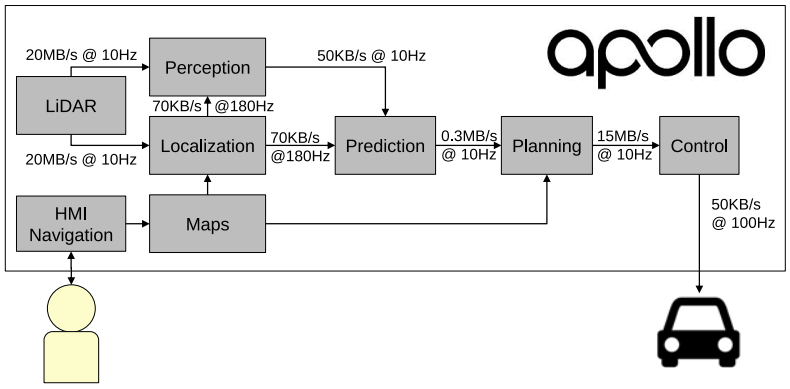

Figure 6.3: Apollo 3.0 component bandwidth and message rates

6.3.2 End-to-End latency between DSM components

As shown on Figure 6.2 communication from a component on the LS2084A SoC to the S34R27 SoC has to go through the NIC, the DDS-XRCE agent [14], and the Rosbridge suite this will

yield increase in latency when sending a message. Therefore, we have measured to end-to-end latency by sending a message from the LS2084A SoC to the S32R274 and back and measure the time it took. The measurement was done by sending a 16 byte message 10 times per second for 1 hour. Figure 6.4 shows the resulting end-to-end latency distribution. The max latency measured was 12.32ms, the 99th percentile latency (shown with green dashed line) was 8.36ms, the possible cause for the difference of approx. 4ms between the 99th percentile and max latency might be related that our experimental setup was not using Time-Sensitive Networking and the LS2084A is not a real-time SoC.

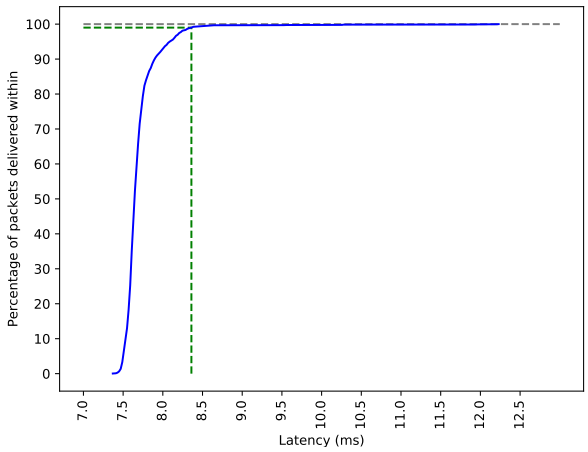

Figure 6.4: DDS & DDS-XRCE End-to-End latency

6.4 Safety scenarios

To evaluate the distributed safety mechanism we create a traffic scenario in the LG SVL simulator using our scenario testing framework. Traffic scenarios can differ from fairly complex to simplistic scenarios, we choose for a simple scenario because it suffices, to analyze the behavior of autonomous vehicle software in a realistic safety critical situation. The base scenario is shown in figure 6.5 it consists of a two-lane road. The ego vehicle (shown as an orange vehicle) is controlled by the Apollo HAD framework and has to overtake the stationary vehicle (shown as a blue vehicle) by lane changing to reach the destination. This scenario is defined in Listing 6.1 in the JavaScript Object Notation (JSON) which will be used by the Python-based scenario testing framework to construct the scenario. The scenario definition contains the waypoints that the ego vehicle must pass, the waypoints of non-ego vehicles, and the fault injection mechanism. Setting up the traffic scenario in Apollo yielded that Apollo stopped the car behind the stationary vehicle because routing algorithm disallowed lane changing. The solution for this was to setup a trajectory using 4 waypoints: the starting waypoint, go straight waypoint, a lane change waypoint, and a destination waypoint. The Apollo HAD framework behavior will be tested under 3 scenarios: normal circumstances, a fault injection, and a safe stop using the distributed safety mechanism.

Scenario 1: Normal scenario

The normal scenario consists of a two-lane road, the ego vehicle controlled by the Apollo HAD framework overtakes a stationary vehicle by lane changing.

The expected behavior is that the ego vehicle reaches the destination without hitting the stationary vehicle.

```
{
    "scene": "SanFrancisco",
    "sim_host": "localhost",
    "sim_port": 8181,
    "apollo_host": "localhost",
    "apollo_port": 9090,

    "routing_request": {
        "module_name": "dreamview"},
        "waypoint": [{"s": 18.664396579325256, "pose": {"y": 4182649.99, "x": 553025.0, "z
            ": 0.0}, "id": "lane_23"},
        {"s": 54.9611313526677, "pose": {"y": 4182612.34066193, "x": 552988.0363219585, "z
            ": 0.0}, "id": "lane_23"},
        {"s": 85.60216757113933, "pose": {"y": 4182592.3711510208, "x": 552973.5053166781,
             "z": 0.0}, "id": "lane_22"},
        {"s": 237.86915881972186, "pose": {"y": 4182482.4963298393, "x": 552859.2115879473
            , "z": 0.0}, "id": "lane_23"}],
    },

    "ego_state": {
        "transform": {"position": {"x":205.597808837891,
                                   "y":10.1244478225708,
                                   "z":7.94014930725098},
                      "rotation": {"x":5.73239667573944E-05,
                                   "y":271.46484375,
                                   "z":1.12486395664746E-05}},
        "velocity": {"x": 0.0, "y": 0.0, "z": 0.0},
        "vehicle_model": "XE_Rigged-apollo",
        "sensors": ["velodyne", "GPS", "Telephoto Camera", "Main Camera", "IMU"]
    },

    "npc_static": {
        "npc_model": "SUV",
        "transform": {"position": {"x": 143.416275024414,
                                   "y": 10.1233310699463,
                                   "z": 8.88082647323608},
                      "rotation": {"x": 359.973022460938,
                                   "y": 269.328979492188,
                                   "z": 0.0046530170366168}}
    },

    "fault_injection": [{
        "position": {"x": 552981.57, "y": 4182603.65},
        "target": "root@192.168.100.102:22",
        "command": ["nohup stress -c 48>nohup.out 2>error.out &", "iconv -c -f utf8 -t
            ascii <error.out >error.ascii", "cat error.ascii"],
        "message": "stress S32V cpu"
    }],

    "clear_fault":[{"target": "root@192.168.100.102:22",
                    "command": ["killall stress >/dev/null 2>error.out", "iconv -c -f utf8
                        -t ascii <error.out >error.ascii", "cat error.ascii"],
                    "message": "kill stress on target"}],
}
```

Listing 6.1: Safety traffic scenario definition

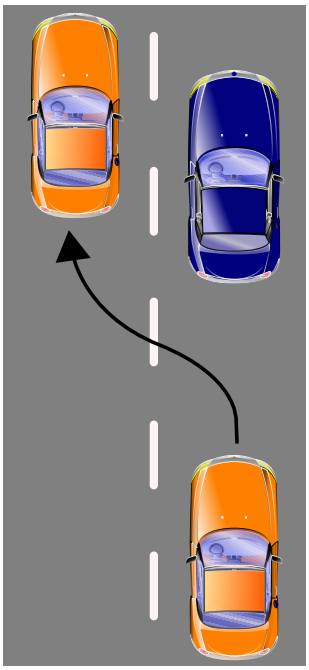

Figure 6.5: Safety traffic scenario

Scenario 2: Fault injection scenario

The fault injection scenario consists of a two-lane road, the ego vehicle controlled by the Apollo HAD framework overtakes a stationary vehicle by lane changing. During the scenario, a CPU stress fault is injected in the hypervisor guest domain on the S32V234 SoC.

The expected behavior is that the ego vehicle starts to wobble after the fault injection, because the control calculations can not keep up and therefore overshoots and eventually collides with the stationary vehicle or other objects.

Scenario 3: Safe stop scenario

The safe stop scenario is the same scenario as scenario 2, however, the distributed safety mechanism is now enabled.

The expected behavior is that the distributed safety mechanism will pause the faulty guest domain when the fault is injected in the hypervisor guest domain on the S32V234 SoC. Furthermore, the distributed safety mechanism will take over the control from the control task running on the S32V234. And issues safe-stop control commands to the ego vehicle to bring it to a safe stop without causing a collision.

6.5 Fault injection experiment

The experiment consisted of executing the 3 different scenarios as described in Section 6.4 using our scenario testing framework. The fault injection in scenarios 2 and 3 is done using the stress workload generator [75], ideally, we would have used a more sophisticated fault injection mechanism such as [32] to evaluate the fault detection coverage of our distributed safety mechanism. However, due to time constraints, we only used the stress CPU workload generator. The outcomes of 3 different scenarios where as follows:

Scenario 1 which acts as the baseline went as expected, the ego vehicle passes the stationary vehicle without colliding into the stationary vehicle or another object.

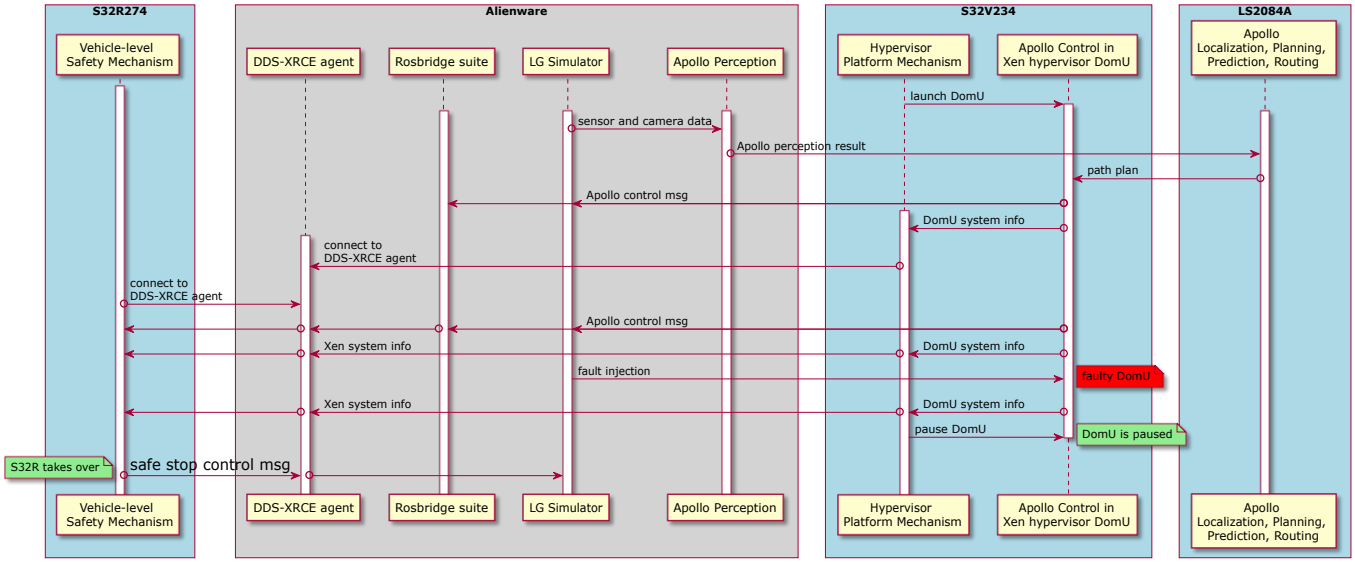

Figure 6.6: Safe-stop sequence diagram

Scenario 2 also went as expected, the ego vehicle starts wobbling when the fault is injected into the S32V234 guest domain and collides into an object. It should be noted during multiple executions the crash behavior is non-deterministic because it hits either the stationary vehicle or a static object inside LG SVL simulator virtual world.

In scenario 3 the distributed safety mechanism is enabled therefore we should expect different outcome then scenario 2. Figure 6.6 shows the sequence diagram of the expected behavior of the distributed safety mechanism. The distributed safety mechanism isolates the Apollo control module into an isolated domain called domU. The LG SVL simulator provides sensor and camera data to the Apollo perception module and the Apollo HAD framework calculates a path plan. The path plan goes to the Apollo control which calculates throttle, brake and steering wheel values, the control data will be published to LG simulator for car actuation but also to the vehicle-level safety mechanism through the Rosbridge suite and the DDS-XRCE agent for monitoring. Eventually, a fault will be injected into the S32V234 DomU domain which puts DomU in a faulty state. The hypervisor platform mechanism detects that DomU is faulty and pauses the domain. Immediately the hypervisor platform monitor publishes the DomU system info to the DDS world. So that the automotive-level vehicle monitor will be informed about the faulty domain, the automotive-level vehicle monitor takes over control and initiates a safe stop.

6.6 Experiment analysis

The impact of the distributed safety mechanism is reflected by the experimental setup executing a scenario where a fault gets injected. The observation of the experiment was that the ego vehicle will do a safe stop in the occurrence of a fault. The fault gets detected in the distributed system, by both the vehicle-level safety mechanism and the hypervisor platform mechanism. The reaction depends on which system detects the fault first. The hypervisor platform mechanism observes the CPU load of the guest domain and analyzes whether the CPU load gets too high. When the CPU load is too high the hypervisor platform mechanism pauses the faulty domain to ensure fail-silent behavior and informs the vehicle-level safety mechanism a fault has occurred. But the vehicle-level safety mechanism does not only rely on the hypervisor platform mechanism to detect the faults in our experiment. The vehicle-level safety mechanism also monitors the control topic inside the DDS domain, we observed that the fault injection causes the rate of messages from control decreases. Therefore, the vehicle-level safety mechanism also detects a fault in the autonomous

vehicle software in the experiment. The experimental evaluation showcased the working of the distributed safety mechanism. However, it should be noted that scenario 3 the safe stop scenario had some anomalies such as:

- Double stop. Sometimes during the safe stop we encountered that the car gets brought to a stop and then vehicle-level safety mechanism stops applying brake and then after a second it reapplies it.
- Control message rate. The LG SVL simulator requires that a control message is send every 10ms. If the vehicle-level safety mechanism could not keep up with this rate, the car control is disengaged.
- DDS-XRCE agent dependence. When the DDS-XRCE agent was not running the vehicle-level safety mechanism was unable to communicate to other DDS nodes, because it is highly dependent on the agent see figure 5.1. Therefore, the safe stop was not being executed correctly.

The presented results indicate that using a publish-subscribe middleware such as DDS and a hypervisor can provide a good basis for safety mechanisms with support for: distributed systems, strong isolation, fail-silent behavior, and analysis of hazardous scenarios from the SOTIF specification.

Chapter 7

Conclusion

One of the key safety goals of autonomous vehicles is to avoid colliding with obstacles. Based on our state of the art study, we identified three challenges that can compromise this safety goal: detectability of faults in a distributed system, hazardous situations in the absence of faults, and cascading failures. In this study, we set the goal to design a distributed safety mechanism addressing these three challenges and to identify available software tools to build this mechanism. The design methodology based on fault injection in a simulated environment was selected as the most promising approach for an industry-relevant result. As a criterion of success for our mechanism evaluation, we defined meeting the safety goal of avoiding collisions. To address the challenge of detecting faults in multiple distributed machines and analyzing hazardous situations in the absence of faults, our safety mechanism needs to reliably read the state of distributed systems. The DDS middleware addresses these requirements, while also meeting industry reliability and security standards. Interestingly, the DDS-XRCE subset allows running the safety mechanism functionality on safety cores with the highest integrity level. Furthermore, our safety mechanism incorporates hardware-supported hypervisors as a means of isolating faults and blocking propagation of cascading failures. During our experiments, we noticed that the hypervisor can also quickly pause execution of a software stack including an operating system. Therefore, hypervisors turned out to be not only useful in ensuring freedom from interference, but also in implementing a fail-silent behavior of faulty software stacks. The design of the distributed safety mechanism was evaluated using an end-to-end automotive simulator modeling a repeatable traffic scenario. During the scenario execution, a Python testing framework injected a fault of overloading a processor, leading the simulated vehicle to crash in the absence of safety mechanisms. When the scenario was repeated with our safety mechanism enabled, the fault was detected and the mechanism stopped the car before any collision occurred. Our evaluation of the distributed safety mechanism shows the potential of reliable middleware and hypervisor in handling faults. Furthermore, the DDS middleware enables the safety mechanism to read HAD software state for handling hazardous scenarios in the absence of faults.

Chapter 8

Future work

Our study covered only few safety challenges in autonomous vehicles. The future research directions and work listed below present opportunities for deeper and broader analysis:

1. Study applicability of well-known challenge-response mechanisms [5] for the distributed safety mechanisms. We presented a safety mechanism, where a probable failure in a lower safety component is addressed by a mechanism running on a higher safety component. To handle a less-probable fault model in the higher safety component, however, the distributed safety mechanism can continuously challenge the high-safety component. If no or incorrect response is returned on time, the lower safety mechanism can inform the driver or bring the system to the safe state.

2. Evaluate (hardware) acceleration or (software) optimization options for running a full DDS stack on a safety core. Modern safety cores, such as ARM Cortex-M7 or Cortex-R52, do not have processing power to run a full-stack DDS middleware protocol, which is the reason why we used DDS-XRCE. However, DDS-XRCE suffers from its dependency on the agent, which runs on a less safer machine.

3. New SOTIF-specific safety mechanisms to address fault-free hazardous scenarios in automated driving. For example, sensor data analysis can identify limitations of the sensor technology in certain situations, such as a dirt spot on a camera lens.

4. Analysis of non-atomic effects in hypervisor-based safety mechanisms. For instance, a hypervisor-driven shut-down of a domain in a complex SoC can suffer from ongoing on-chip operations outside of processor cores. Network-on-chip, caches, and peripheral devices may be busy with a long operation, when the hypervisor shuts down a domain. A separate study is required to understand if such operations can be properly finished without causing deadlocks or starvation.

5. Test the retargetability of the presented mechanism on other platforms, such as physical vehicles, and HAD frameworks, such as Autoware [41]. Furthermore, a different middleware software, such as Apollo Cyber RT [29], is a good candidate for evaluation of retargetability.

6. Study and design options for more advanced safe stop operations. For example, a safe stop within the lane would be more appropriate than the presented handbrake-like operation. However, such a safe stop would need to detect working sensors and actively steer the vehicle despite faults in other subsystems.

7. We intend to publish selected topics from this project in a scientific conference or workshop.

Bibliography

[1] ANSYS SCADE safing gate and the doer-checker architecture. https://www.ansys.com/-/media/ansys/corporate/resourcelibrary/article/drive-safely-aa-v12-i1.pdf. 9, 10

[2] Automotive Electronics Council. https://www.iso.org/standard/71691.html. 7

[3] Autonomous driving simulation with unity (presented by baidu usa). https://www.gdcvault.com/browse/gdc-19/play/1026207. 28

[4] Avsimulation - scaner studio. 28

[5] Challenge-response watchdog mechanism in dSpace MicroAutoBox II computers. https://www.dspace.com/en/inc/home/products/hw/micautob/microautobox2/mabx_watchdog.cfm. 37

[6] EcoTwin truck platooning project. https://www.youtube.com/watch?v=uf_1vbh75y0&feature=youtu.be. 10

[7] Intel/Mobileye Responsibility-Sensitive Safety. https://www.mobileye.com/responsibility-sensitive-safety/. 9, 10

[8] MIT Moral Machine. http://moralmachine.mit.edu/. 10

[9] Toyota Guardian. https://global.toyota/en/newsroom/corporate/26069806.html. 9, 10

[10] Virtual-based safety testing for self-driving cars from nvidia drive constellation. 28

[11] *proc(5) Linux User's Manual*, 4.16 edition, Apr 2018. Available at http://manpages.courier-mta.org/htmlman5/proc.5.html. 26

[12] California Car Accident Law. https://www.hg.org/car-accident-law-california.asp, 2019. [Online; accessed 31-May-2019]. 8, 10

[13] dSPACE Hardware-in-the-Loop Test Systems. https://www.dspace.com/shared/data/pdf/2019/dSPACE-Hardware-in-the-Loop-Systems_Business-field-brochure_01-2019_English.pdf, Jan 2019. 28

[14] eProsima FastRTPS and DDS-XRCE implementations. https://github.com/eProsima, May 2019. 21, 26, 30

[15] LG SVL simulator documentation. https://www.lgsvlsimulator.com/docs/getting-started/, May 2019. 27, 28

[16] NVIDIA DRIVE - Software. https://developer.nvidia.com/drive/drive-software, May 2019. 5

[17] Prescan. https://tass.plm.automation.siemens.com/prescan, May 2019. 28

[18] Baidu. Apollo an open autonomous driving platform. `http://apollo.auto/`, 2018. 4, 5

[19] Michael Behrisch, Laura Bieker, Jakob Erdmann, and Daniel Krajzewicz. SUMO–simulation of urban mobility: an overview. In *Proceedings of SIMUL 2011, The Third International Conference on Advances in System Simulation*. ThinkMind, 2011. 28

[20] Lucia Lo Bello. The case for ethernet in automotive communications. *ACM SIGBED Review*, 8(4):7–15, 2011. 20

[21] T. Bijlsma, M. Kwakkernaat, and M. Mnatsakanyan. A real-time multi-sensor fusion platform for automated driving application development. In *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, pages 1372–1377, July 2015. 12

[22] Tjerk Bijlsma and Teun Hendriks. A fail-operational truck platooning architecture. *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 1819–1826, 2017. 4, 5

[23] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, November 1987. 11

[24] F. V. Brasileiro, P. D. Ezhilchelvan, S. K. Shrivastava, N. A. Speirs, and S. Tao. Implementing fail-silent nodes for distributed systems. *IEEE Transactions on Computers*, 45(11):1226–1238, Nov 1996. 24

[25] Alan Burns and Robert Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, pages 1–69, 2013. 18

[26] Z. Chen, T. Ellis, and S. A. Velastin. Vehicle detection, tracking and classification in urban traffic. In *2012 15th International IEEE Conference on Intelligent Transportation Systems*, pages 951–956, Sep. 2012. 5

[27] Christoffer Dall and Jason Nieh. KVM/ARM: the design and implementation of the linux ARM hypervisor. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 333–348. ACM, 2014. 25

[28] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. *arXiv preprint arXiv:1711.03938*, 2017. 28

[29] Natasha Dsouza and Ning Qu. Apollo Cyber RT The Runtime Framework Youve Been Waiting For. `https://link.medium.com/fE7aFPAAdX`, Feb 2019. 5, 37

[30] C. Ebert and C. Jones. Embedded Software: Facts, Figures, and Future. *Computer*, 42(4):42–52, April 2009. 18

[31] J. Farkas, L. L. Bello, and C. Gunther. Time-Sensitive Networking Standards. *IEEE Communications Standards Magazine*, 2(2):20–21, JUNE 2018. 11

[32] Y. Fu, A. Terechko, T. Bijlsma, P. J. L. Cuijpers, J. Redegeld, and A. O. Ors. A Retargetable Fault Injection Framework for Safety Validation of Autonomous Vehicles. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 69–76, March 2019. 5, 33

[33] Manfred Grosmann, Mario Hirz, and Jurgen Fabian. Efficient application of multi-core processors as substitute of the E-Gas (Etc) monitoring concept. *2016 SAI Computing Conference (SAI)*, pages 913–918, 2016. 9

[34] Part Guide. Intel® 64 and ia-32 architectures software developers manual. *Volume 3B: System programming Guide, Part*, 2, 2011. 5, 21

[35] Emme Hall. Hands-on with Comma.ai's add-on Level 2 autonomous tech, Aug 2018. 5

[36] Green Hills. Integrity multivisor. https://www.ghs.com/products/rtos/integrity_virtualization.html, 2019. 25

[37] J. Hwang, S. Suh, S. Heo, C. Park, J. Ryu, S. Park, and C. Kim. Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones. In *2008 5th IEEE Consumer Communications and Networking Conference*, pages 257–261, Jan 2008. 13

[38] IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems. Standard, The International Electrotechnical Commission, 2008. 7

[39] ISO 26262: Road Vehicles : Functional Safety. Standard, International Organization for Standardization, Geneva, CH, September 2011. v, v, 7, 10, 14, 15, 18

[40] ISO/PAS 21448: Road vehicles – Safety of the intended functionality. Standard, International Organization for Standardization, Geneva, CH, January 2019. 7, 10, 18

[41] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '18, pages 287–296, Piscataway, NJ, USA, 2018. IEEE Press. 5, 37

[42] D. Kleidermacher and M. Wolf. MILS virtualization for Integrated Modular Avionics. In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, pages 1.C.3–1–1.C.3–8, Oct 2008. 14

[43] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE, 2004. 28

[44] Philip Koopman. The Big Picture for Self-Driving Car Safety. https://www.slideshare.net/PhilipKoopman1/the-big-picture-for-selfdriving-car-safety-standards, 2019. 8

[45] Rob Krten. *Getting started with QNX Neutrino 2: a guide for realtime programmers*. PARSE Software Devices, 1999. 15, 25

[46] D. Lopez and M. Clairet. Fail silent and robust power management architectures to enable autonomous driving embedded systems. In *2016 International Conference on Electrical Systems for Aircraft, Railway, Ship Propulsion and Road Vehicles International Transportation Electrification Conference (ESARS-ITEC)*, pages 1–6, Nov 2016. 8

[47] Nicholas Mc Guire. Linux for safety critical systems in IEC 61508 context. In *Proceedings of the Ninth Real-Time Linux Workshop in Linz*, 2007. 5

[48] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014. 12

[49] R. Mijat and A. Nightingale. Virtualization is coming to a platform near you. ARM white paper, 2011. 13

[50] Patrick Mochel. The sysfs filesystem. In *Linux Symposium*, page 313, 2005. 25

[51] Robert N Charette. This Car Runs on Code. *IEEE Spectrum*, 46, February 2009. 1

[52] Arne Nordmann and Ingo Ltkebohle. micro-ROS Overview. https://micro-ros.github.io/docs/home/, Apr 2019. 21

[53] NXP Semiconductors. *S32V234 Reference Manual*, Sep 2017. 24

[54] Object Management Group (OMG). DDS for eXtremely Resource Constrained Environments Specification, Version 1.0beta2. OMG Document Number ptc/19-03-27 (https://www.omg.org/spec/DDS-XRCE/1.0/Beta2/), 2019. viii, 11, 21

[55] Sebastian Ohl. Staying in lane on highways with EB robinos. May 2017. 4, 5

[56] U. Ozguner, C. Stiller, and K. Redmill. Systems for Safety and Autonomous Behavior in Cars: The DARPA Grand Challenge Experience. *Proceedings of the IEEE*, 95(2):397–412, Feb 2007. 2, 18

[57] Gerardo Pardo-Castellote. OMG Data-Distribution Service (DDS): Architectural Overview. Technical report, REAL-TIME INNOVATIONS INC SUNNYVALE CA, 2004. 11

[58] Anup Patel, Mai Daftedar, Mohamed Shalan, and M Watheq El-Kharashi. Embedded hypervisor xvisor: A comparative analysis. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 682–691. IEEE, 2015. 25

[59] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM*, 17(7):412–421, July 1974. 14

[60] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009. 11

[61] M. Raho, A. Spyridakis, M. Paolino, and D. Raho. KVM, Xen and Docker: A performance analysis for ARM based NFV and cloud computing. In *2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, pages 1–8, Nov 2015. 13

[62] Eder Santana and George Hotz. Learning a driving simulator. *arXiv preprint arXiv:1608.01230*, 2016. 5

[63] M. Satyanarayanan. Edge computing for situational awareness. In *2017 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–6, June 2017. 8

[64] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*, pages 621–635. Springer, 2018. 28

[65] Ed Sikha, Rick Simpson, C May, and H Warren. *The PowerPC Architecture: A specification for a new family of RISC processors*. Morgan Kaufmann Publishers, 1994. 16

[66] S Sing. Critical reasons for crashes investigated in the national motor vehicle crash causation survey. Feb 2015. 1

[67] Valentine Sinitsyn. Jailhouse. *Linux Journal*, 2015(252):2, 2015. 14, 25

[68] S. Stabellini. Xen ARM with Virtualization Extensions whitepaper. https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions_whitepaper, Apr 2014. 24, 25

[69] SAE Standard. J3016.. *Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems*, 4:593–598, 2014. viii, 3

[70] Tadeusz Stolarski, Yuji Nakasone, and Shigeka Yoshimoto. *Engineering analysis with ANSYS software*. Butterworth-Heinemann, 2018. 28

[71] Sebouh Toumassian, Rico Werner, and Axel Sikora. Performance measurements for hypervisors on embedded ARM processors. pages 851–855, 09 2016. 14

[72] Udacity. udacity/self-driving-car-sim. https://github.com/udacity/self-driving-car-sim, Jan 2019. 28

[73] Ministerie van Algemene Zaken. Wie is aansprakelijk bij een ongeluk met een zelfrijdende auto?, Oct 2018. 10

[74] Alexander Warg and Adam Lackorzynski. The Fiasco. OC Kernel and the L4 Runtime environment (L4Re). avail. 25

[75] Amos Waterland. stress is a deliberately simple workload generator for POSIX systems. https://people.seas.harvard.edu/~apw/stress/, Jul 2014. 33

[76] Wikipedia. Alienware. https://en.wikipedia.org/wiki/Alienware, Jun 2019. 28

[77] Wikipedia. Automotive Electronics Council. https://en.wikipedia.org/wiki/Automotive_Electronics_Council, 2019. 7

[78] Wikipedia. Built-In Self-Test (BIST). https://en.wikipedia.org/wiki/Built-in_self-test, 2019. 8

[79] Wikipedia. Experiment. http://en.wikipedia.org/w/index.php?title=Experiment&oldid=893555614, 2019. [Online; accessed 31-May-2019]. 19

[80] Wikipedia. Fault model. https://en.wikipedia.org/wiki/Fault_model, 2019. 5

[81] Wikipedia. Formal methods. http://en.wikipedia.org/w/index.php?title=Formal%20methods&oldid=896120926, 2019. [Online; accessed 31-May-2019]. 19

[82] Wikipedia. Hypervisor. https://en.wikipedia.org/wiki/Hypervisor, Jun 2019. v

[83] Wikipedia. Manufacture- or user-configurable moral in a self-driving car, Y.N. Harari 21 lessons for the 21 century. https://en.wikipedia.org/wiki/21_Lessons_for_the_21st_Century, 2019. 10

[84] Wikipedia. MATLAB. https://en.wikipedia.org/wiki/MATLAB, Jun 2019. 27

[85] Wikipedia. NCAP New Car Assessment Program, government agency, not a standard. https://en.wikipedia.org/wiki/New_Car_Assessment_Program, 2019. 10

[86] Wikipedia. PPAP - Production Part Approval Process. https://en.wikipedia.org/wiki/Production_part_approval_process, 2019. 10

[87] Wikipedia. Reliability Engineering. https://en.wikipedia.org/wiki/Reliability_engineering, 2019. 7

[88] Wikipedia. Trolley problem. https://en.wikipedia.org/wiki/Trolley_problem, 2019. 8, 10

[89] EGAS Workgroup. Standardized E-Gas Monitoring Concept for Gasoline and Diesel Engine Control Units. *Version*, 6:57, 2015. viii, 9, 10, 22

[90] Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. Torcs, the open racing car simulator. *Software available at http://torcs.sourceforge. net*, 4(6), 2000. 28

[91] Junko Yoshida. NXP's Reger Redefines CTO's Role, Jan 2019. viii, 15