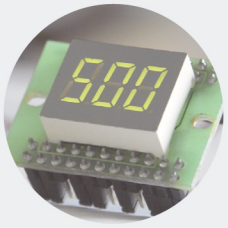
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES






Testes Automaticos utilizando o Vector CANoe num Sensor Inercial

FILIPA RAQUEL SILVA PEREIRA
novembro de 2020

POLITÉCNICO
DO PORTO

isep Instituto Superior de Engenharia do Porto

Automated Testing using Vector CANoe on an Inertial Measurement Unit

Verification and Validation Project

Mestrado em Engenharia Eletrotécnica e de Computadores

Filipa Raquel Silva Pereira

Supervisor ISEP:
Prof. André Fidalgo
Coordinator Bosch Car Multimedia Portugal, S.A - Braga:
Engº. Pedro Neves

Ano Letivo: 2019-2020

Instituto Superior de Engenharia do Porto
Departamento de Engenharia Eletrotécnica
Rua Dr. António Bernardino de Almeida, 431, 4200-072 Porto

Abstract

The purpose of this dissertation is to develop and enhance an Automation Process of Software Tests performed on the Vector CANoe tool. The SW Tests were performed on an Inertial Measurement Unit's verification and validation project.

This thesis has as a base of discussion the state of the Automotive Industry and the development of Software Tests in that sector, with focus on Safety norms and regulations, the general tools and protocols used on SW development and SW testing and, more than that, what Continuous Integration is and what it is used for. The proposed objectives focus on the full automation of the sensor's Software Qualification Tests and their integration on a Jenkins Continuous Integration Server, being the documentation of the performed work an important topic.

The problems regarding the manual execution of the Software tests on Inertial Measurement Units are given and discussed. Each step of the automation process is described in detail, with specific examples. The performed steps for setting up a Jenkins Server and test bench are also presented and discussed.

In general, the work performed was satisfactory and the end results greatly improved the team's work efficiency and quality.

Resumo

O objetivo desta dissertação é a melhoria do processo de automatização de testes de software realizados com a ferramenta de Vector CANoe. Os testes de software foram realizados para um projeto de verificação e validação de software para Sensores Inerciais.

Esta tese tem como base de discussão o estado da Indústria Automóvel e o desenvolvimento de testes de software nessa área, com foco em normas e regulamentos de segurança, ferramentas e protocolos utilizados para o desenvolvimento de software e de testes e, para além disso, o que é a Integração Contínua e para que é utilizada. Os objetivos propostos concentram-se na automatização completa dos testes de qualificação de software do sensor e sua integração num servidor de integração contínua Jenkins, sendo a documentação do trabalho realizado também um tópico importante.

Os problemas relacionados com a execução manual de testes de software em Sensores Inerciais são apresentados e discutidos. Cada etapa do processo de automação é descrita em detalhe, com exemplos específicos. Os passos efectuados para a configuração de um servidor Jenkins e de uma bancada de testes são apresentados e discutidos.

No geral, o trabalho realizado foi satisfatório e os resultados finais melhoraram imenso a eficiência e a qualidade do trabalho da equipa.

Contents

List of Figures

List of Tables

Acronyms

Abbreviation	Description
3D	*3 Dimensions* or Three Dimensional
ABS	*Anti-Blocking Brakes or Antilock Braking System*
ACC	*Adaptive Cruise Control*
ADAS	*Advanced Driver Assistance Systems*
AFS	*Active Front Steering*
ALM	*Application Lifecycle Management*
AHRS	*Altitude Heading Reference Systems*
APP	*Application Layer*
AR	*Augmented Reality*
ASAM	*Association for Standardization for Automation and Measuring Systems*
ASIL	*Automotive Safety Integrity Level(s)*
ASPICE	*Automotive Software Performance Improvement and Capability Determination*
ASTQB	*American Software Testing Qualification Board*
BCS	*British Computer Society*
BSW	*Basic Software*
CAN	*Controller Area Network*
CANH	*CAN High Line*
CANL	*CAN Low Line*
CANoe	*CAN Open Environment*
C-ITS	*Cooperative Intelligent Transport Systems*
CCP	*CAN Calibration Protocol*
CD	*Continuous Deployment*
CDel	*Continuous Delivery*
CI	*Continuous Integration*
CRC	*Cyclic Redundancy Checks*
CTO	*Command Transfer Object*
DAQ	*Data Acquisition*
DEM	*Diagnostics Event Manager*
DFM	*Dämpfersteurung or Damper Control*
DoF	*Degrees of Freedom*
DLC	*Data Lenght Code*
DTO	*Data Transfer Object*
ECC	*Error Correction Code(s) or Error Checking and Correction*
ECU	*Electronic Control Unit*
E/E	*Electronic and Electrical*

Abbreviation	Description
ESP	*Electronic Stability Program*
GPS	*Global Positioning System*
GPSD	*General Product Safety Directive*
HARA	*Hazard Analysis and Risk Assessment*
HIL	*Hardware-in-the-loop*
HHC	*Hill Hold Control*
HW	*Hardware*
IMU	*Inertial Measurement Unit*
IoT	*Internet of Things*
ISO	*International Organization for Standardization*
ISEB	*British Information Systems Examination Board*
ISQI	*International Software Quality Institute*
ISTQB	*International Software Testing Qualification Board*
LiDAR	*Light Detection and Ranging*
LIN	*Local Interconnect Network*
MEMS	*Micro-electromechanical System(s)*
MOST	*Media Oriented Systems Transport*
MTA	*Memory Transfer Address*
NAVI	*Navigation System*
OEM	*Original Equipment Manufacturer*
OSI	*Open Systems Interconnection*
ODT	*Object Descriptor Tables*
PII	*Personal Identifiable Information*
PLS	*Physical Signaling*
QA	*Quality Assurance*
QC	*Quality Control*
QM	*Quality Management*
RBT	*Requirements Based Tests*
RF	*Radio Frequency*
RFP	*Renesas Flash Programmer*
RTE	*Runtime Environment*
RoSe	*Rollover Sensing*
RSC	*Roll Stability Control*
SPI	*Serial Pheripheral Interface*
SDLC	*Software Development Lifecycle*
SW	*Software*
uC	*MicroController*
UDS	*Unified Diagnostic Service*
VDC	*Vehicle Dynamics Control*
VMO	*Vehicle Motion Observer*
VR	*Virtual Reality*
XCP	*Universal Measurement and Calibration Protocol*

Acknowledgments

This work wouldn't be over without thanking everyone that, in one way or another, contributed to its conclusion.

To Engº. Pedro Neves for being a great coordinator and mentor and for giving me the idea of creating this dissertation based on the work we were already doing in the project. Thank you also for always being available to answer my questions and doubts.

To Engº. André Fidalgo for giving me this last chance at completing my Master's Degree, for helping me understand what was needed to be done and also for being understanding of my work schedule. Without his support this dissertation wouldn't have been written like this.

To my colleagues from the IMU team at Bosch Car Multimedia Portugal, S.A - Braga, for all the support and moral boosts they have given me during all this time, with a special mention to José Sampaio, Beatriz Gonçalves and Ana Alves.

My final acknowledgments go to my family, especially to my Boyfriend and my cute little dogs for always being there for me and helping me through the hard times.

Chapter 1

Introduction

1.1 Context

The idea for the creation of this dissertation came up in a curious way. During a group discussion regarding work methods, and the need to have everything documented.

The author having recently joined Bosch Car Multimedia - Development and Innovation, in Braga (henceforth called the Company), to work in a team project related with Software Verification and Validation of an IMU sensor, was the perfect fit to document this process, with the help of a group of SW Test Engineers responsible for the testing of the newest Inertial Measurement Unit, capable of providing accelerations and angular rates to a multitude of electronic control units present in the car's bus.

In the technological information area, the users are constantly bombarded with keywords that define the current state of the art of technology.

The Verification and Validation area is no different. Agile methodologies require more agile work processes and also the reduction of repetitive work. This in turn gives rise to the automation of tasks in an alarming pace.

With this in mind, the IMU Verification and Validation team (henceforth called the Team) needed to adapt a more agile way of work. In order for this to be performed, the already existing tests needed to be updated and automated. When completed, this would reduce the amount of repetitive work that the Team needed to perform for each new SW update or iteration.

Another thing need to optimize the Team's work would be an automatic way of running each new Test Campaign. This is where a Jenkins Automation Server comes in handy.

With all these information in mind, the idea came that the author of this dissertation should be responsible of gathering all the information needed to start the automation process of the tests, of recording the various methods implemented and also start the work related with having a Jenkins Automation Server working, and also create the respective documentation and tutorials. This work would greatly help the Team and give it better work quality, speed and credibility.

1.2 Objectives

The work developed in this dissertation is of high importance, due to the rapid development of solutions involving the **Inertial Measurement Unit [IMU]** and its various uses as a Passive Safety Car Sensor, capable of working with various others car sensors, like **Anti-Lock Braking System [ABS]** and **Electronic Stability Program[ESP]**. With these topics in mind, this work contributes to increase the *know-how* of the IMU sensor, and also contributes to a better testing of the sensors Quality and Functionality Assurance.

The objectives of this dissertation are to better understand the Testing Process, fully automate the testing of the IMU sensor, integrate a working Jenkins server and document every step of these various new processes and transmit the information to the Team in order to better help on their daily work.
In order to accomplish these objectives the following tasks are necessary:

- Study the Inertial Measurement Unit, Automotive **Software [SW]** Testing, the **International Software Testing Qualification Board [ISTQB]** testing standards and the **International Organization for Standardization [ISO]** standards;
- Study and learn how to use the Vector CANoe Tool;
- Study the testing process for the IMU;
- Study the various communication protocols;
- Study and learn how to program using CAPL language;
- Study the automation process of Tests;
- Study the use and application of Jenkins Servers;
- Apply knowledge to optimize already existing manual tests;
- Apply knowledge to automate the existing manual tests;
- Apply knowledge and assemble an Test Bench capable of connecting to a Jenkins Server;
- Apply knowledge to get an Automated Test Campaign running on an Jenkins Server;
- Apply knowledge and support the rest of the team in learning new/existing concepts;
- Apply knowledge and support the rest of the team in implementing the Test Campaign on a Jenkins Server.

1.3 Structure

This dissertation details the work performed in order to achieve the previously presented objectives.

A brief explanation of this document organization is shown in this section.

In Chapter 1 the theme of this Dissertation is presented with the respective context, a list of objectives, the planned timeline and the structure of the document.

In Chapter 2 the State of the Art is presented. This consists of the research performed during the study phase of the Objectives, regarding the topics of the Automotive Industry - more specifically about the what is Automotive Engineering, what is part of the Automotive domain and how **Electronic Control Units [ECU]** work in Automotive. The second part of Chapter 2 is regarding the Inertial Measurement Unit and it's use cases and way of employment. The Automotive SW Testing, where SW Testing is explained, a small overview of the evolution of the Automotive SW Testing is given and also the current challenges in Automotive Testing. What is different with SW testing for Safety-Critical Sytems, what are the most important ISO Standards, what is AUTOSAR, **Automotive Safety Integrity Level(s) [ASIL]** Levels and ASPICE and why they are important. What is ISTQB and the corresponding Testing Methods.

In Chapter 3 the Definition of the Problem is discussed. This Chapter starts with an overview of the Testing Tools and Knowledge of the Inertial Measurement Unit's Verification and Validation Team - where more information is presented about the **Application Lifecycle Management [ALM]**, the Vector CANoe tool, the various Protocols used, the SPI Simulyzer, Renesas Flash Programmer and Jenkins. The Testing Problems are also described in this chapter, since the Testing Process, CANoe testing, CAPL Testing and Manual Building and Flashing of the Sensor.

In Chapter 4 the Resolution of the Problems is shown. The Chapter starts with a description of the Automated Tests - From Manual to Automatic Tests process. The chapter then continues with the explanation of the Automated Tests with use of the Jenkins server, with all it's Hardware/Software/Test related dependencies and solutions and the End Results with the Jenkins Automation Completion, followed by the discussion of the results.

In Chapter 5 the Conclusion is presented. In this section a detailed conclusion of the dissertation is given, with emphasis on the highlights regarding the final status of Objectives, and all the positive results this has brought to the Team. An overview of all the remaining aspects of the dissertation is also given in this chapter, regarding all the difficulties and also the extra work performed.

1.4 Plan

The thesis planned in accordance with an work schedule already underway for the Team.

With this in mind, Figure 1.1 presents the plan for the creation of this Dissertation, based on work developed in parallel in the Company.

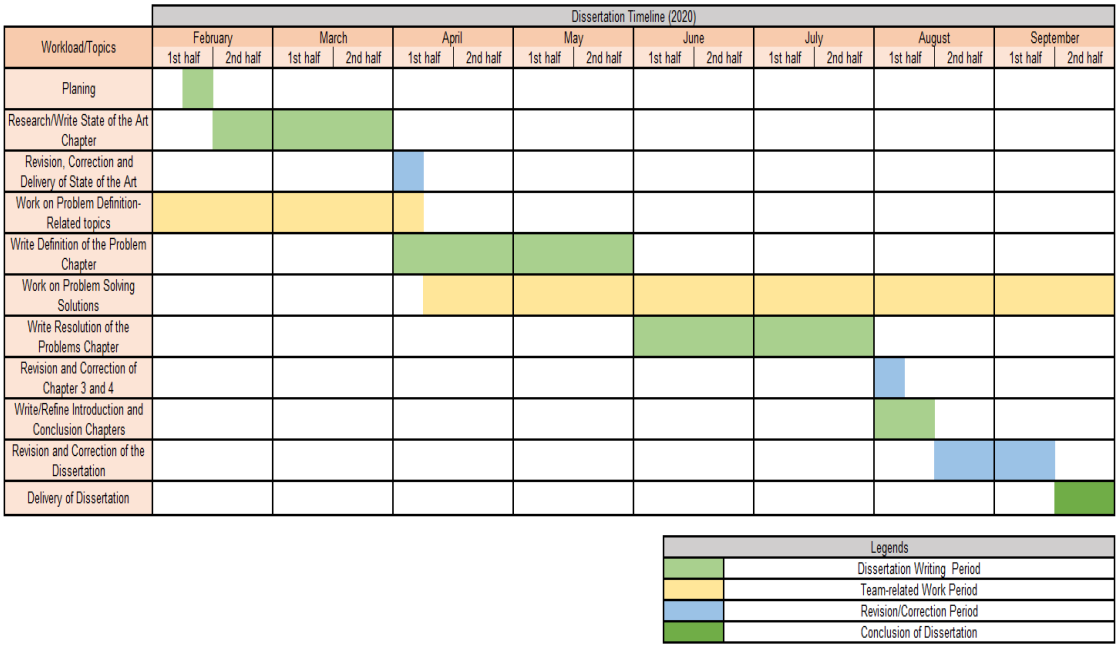


Figure 1.1: Dissertation Timeline

Chapter 2

State of the Art

In this chapter the current state of the art is presented. The state of the art is a way of understanding how the current technologies are being development and can help better contextualize all the important topics in this dissertation.

2.1 Automotive

2.1.1 Automotive Engineering

Automotive engineering, as well as aerospace engineering and naval architecture, is a branch of vehicle engineering, which incorporates elements from various fields, like mechanical, electrical, software and safety engineering. This branch applies to the design, manufacture and operation of motorcycles, automobiles, trucks and their respective engineering subsystems.
Electronics control more and more functionalities in a modern vehicle. The electrical components are assembled in modules with specific requirements for electrical, mechanical and chemical environments which are later mounted in the car. These modules are known as **Electronic Control Units** [**ECU**]. From the 1970s, when electronics started penetrating the automobile industry, the amount of functionality taken over by electronics increased permanently.
This massive penetration of electronics in the automotive industry has mainly the following access points:

- Replacement of an existing mechanical systems;
- Merging of mechanical functions with electronic ones in mechatronic systems (e.g. Anti-blocking brakes [**ABS**]);
- New functions entirely possible only by using electronics (e.g. Navigation systems).

The continuous growth of electronics into the vehicle functions has left its mark in history [1]. Here is a list of some examples:

- The electronic engine control in the late 1970s;

- The ABS in the 1980s;
- The airbags in the 1990s;
- Infotainment and GPS in the 2000s.

It is important to note that automotive electronics are different from consumer electronics due mainly to a result of specific constraints such as [1]:

- Functioning under harsh environmental conditions like temperature range, mechanical stress, humidity, etc;
- Strict requirements for electromagnetic compatibility;
- Stringent availability and reliability requirements which are key factors for customer perception;
- Stringent safety requirements enforced by national and international regulations;
- Specific life cycles, as in Figure 2.1.

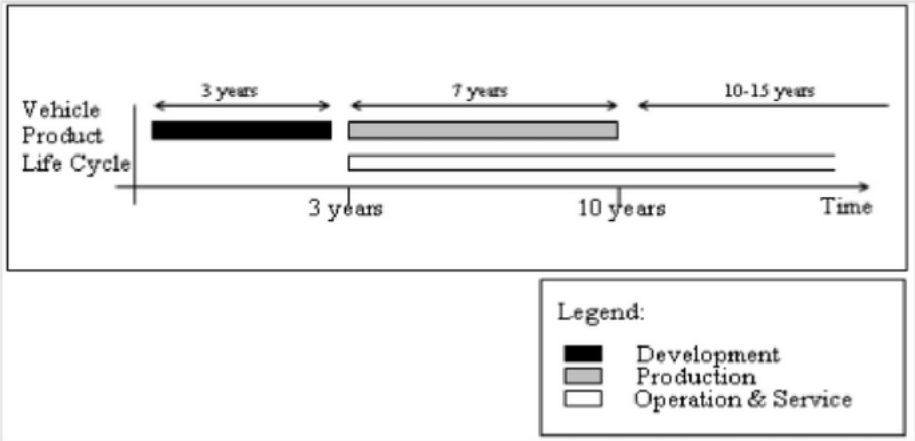


Figure 2.1: Product Life cycle of a Vehicle [2]

2.1.2 Automotive Domain

Recent studies show that the strongest impact of embedded systems on the market has to be expected in the automotive industry. Prominent examples of such electronic systems are safety facilities, **advanced driver assistance systems** [**ADAS**], or **adaptive cruise control** [**ACC**]. These functionalities are employed by software within ECUs. A modern car has up to 80 ECUs. Furthermore, the complexity of car software dramatically increases as it implements formerly mechanically or electronically integrated functions. Yet, the functions are distributed over several ECUs interacting with each other. At the same time, there is a demand to shorten time-to-market for a car by making its software components reliable and safe.

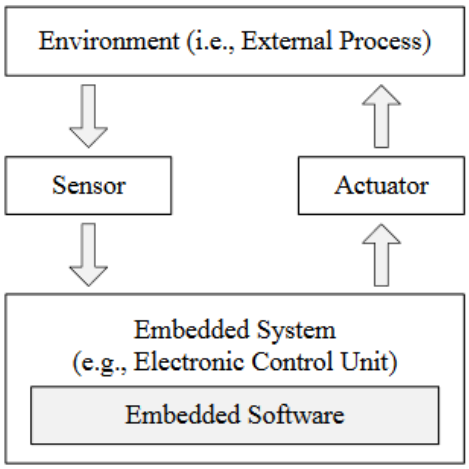


Figure 2.2: Interactions of the ECU with the Environment [3]

An embedded system is any computer system or computing device that performs a dedicated function or is designed for use with a specific software application. It is frequently connected to a physical environment through sensors and actuators, as shown in Figure 2.2. It is considered an real-time system when initiation and termination of activities must meet specific timing constraints. It has to obey hard, soft and statistical real-time properties. **Hard real-time properties** are timing constraints which have to be fulfilled in any case. **Soft real-time properties** are time constraints which need to be satisfied only in the average case, to a certain percentage, or just fast enough. In **statistical real time**, deadlines may be missed, as long as they are compensated by faster performance elsewhere to ensure that the average performance meets a hard real-time constraint.

In the automotive domain, one of the most important aspects is security [4]. Due to the rapid technology development and new business opportunities, automotive systems are undergoing a tremendous transformation into "computers on wheels". The inherent problems of information security and privacy become real in the automotive domain. The main factors for security and privacy concerns come from recent innovations in autonomous drive and cooperative **Intelligent Transport Systems** [**C-ITS**] [5]. Autonomous Driving takes partial or full control of a car from a human driver. At the same time, the computer-based automotive systems consume and generate a large amount of data. C-ITS enables data exchange among the cars and between the vehicles on the road and any host in the transportation systems or any device connected to the Internet. The implication of the transformation from isolated mechanical systems to "**Internet of Things**" [**IoT**] is enormous on the technical front as well as on the social aspect. Automotive systems collect driving behavior data and locations and communicate with central systems in order to optimize the overall traffic flow. While it would be beneficial for privacy to sanitize the data of all **Personally Identifiable Information** [**PII**] there is a need to retain a minimal set of information about the sender in order to ensure authenticity and trustworthiness of the data. It is a challenge to identify and analyze various facts and prognoses in a dynamic setting.

2.1.3 ECUs in Automotive

ECUs are embedded systems operating inside the car. It controls one or more of the electrical subsystems in a vehicle. In a car, ECUs are connected via bus systems such as **CAN**, **LIN**, **MOST**, **FlexRay** among others.

Based on Figure 2.2,it is understandable that sensors provide information about the current state of the external process by means of so-called monitoring events. They are transferred to the controller as input events. The controller must react to each received input event, and depending on the event corresponding states of the external process are determined and actuators receive the results given by the controller that then are transferred to the external process.

2.2 Inertial Measurement Unit

In the Automotive area if an ECU is considered the brain of the car, then an IMU is one of its various limbs with a multitude of technical appliances. With this in mind, this dissertation is focused only on the IMU, due to the fact of being part of the Team responsible for developing the SW tests for this sensor. The following sections of this chapter will help further understand what an IMU is, and how it is fundamental for the car's safety and control.

2.2.1 What is an IMU?

Before focusing on what a **Inertial Measurement Unit** [**IMU**] [6][7] is in particular, like the one in Figure 2.3, let's start more broadly. A sensor is a device, module, machine or subsystem whose purpose is to detect events or changes in its environment and send the information to other electronics. It is always used with other electronics.

Figure 2.3: An example of a Bosch's IMU [8]

2.2.1.1 What can a Sensor Measure?

An IMU is a specific type of sensor that measures angular rate, acceleration and sometimes magnetic field. IMUs are typically composed of a 3-axis accelerometer and a 3-axis gyroscope, which would be considered a 6-axis IMU.

They can also include an additional 3-axis magnetometer, which would be considered a 9-axis IMU. Technically, the term "IMU" refers to just the sensor itself, but IMUs are often paired with sensor fusion software which combines data from multiple sensors to provide measure of orientation and heading.

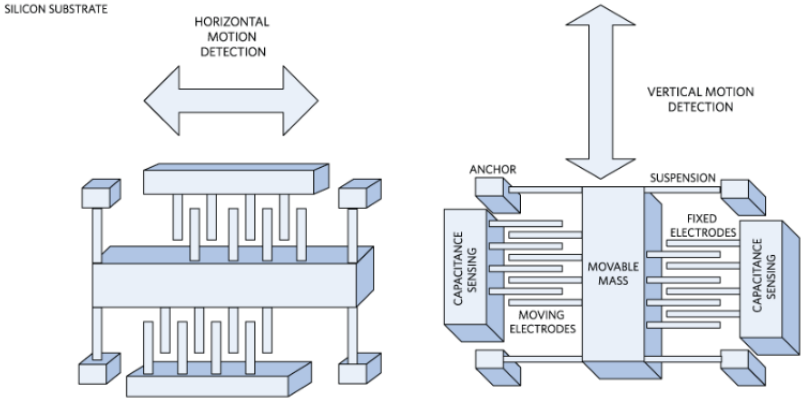


Figure 2.4: Accelerometer MEMS Working Principle [12]

In common usage, the term "IMU" may be used to refer to the combination of the sensor and sensor fusion software; this combination is also referred to as an **Attitude Heading Reference System** [**AHRS**].

2.2.2 History of IMUs

Inertial Measurement devices have a very rich history [9]. The field of study began with motion-stabilization of gun sights for war ships and later developed to be used by guidance systems of aircrafts and missiles.

Recent advances in **micro-electromechanical systems** [**MEMS**] and other micro fabrication techniques have led to low-cost, more compact devices, while having the processing power of personal computers increase exponentially, which means that it is now possible for inertial systems to reach end-users, due to the lower cost of production and less processing power needed to operate.

Recent uses of inertial sensors in major products have tended toward the automotive area. The first major applications of MEMS accelerometers, Figure 2.4 , was as a cheap, reliable trigger mechanism for airbag deployment [10]. Gyroscopes are most often used to provide turn rate information to four wheel steering systems to help the front and rear tires match speed [11]. They have been used to provide heading information for in-vehicle tracking systems, which obtain position from the speedometer or a Global Positioning System Unit.

2.2.3 How does the sensor work?

An IMU provides 2 to 6 **Degrees of Freedom** [**DoF**], which refers to the number of different ways that an object is able to move throughout 3D space. The maximum possible is 6 DoF [13], which would include 3 degrees of translation (flat) movement across a straight plane/along each axis and 3 degrees of rotational movement across the X, Y and Z axes, like shown on Figure 2.5.

Figure 2.6 shows an example of how the 6 DoF are perceived on a 2-wheel vehicle.

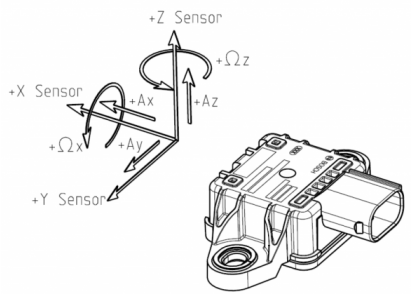


Figure 2.5: 6 DoF of a IMU from Bosch

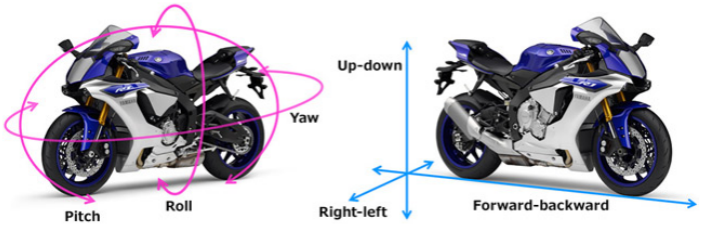


Figure 2.6: Example of the 6 DoF of a Motorcycle [14]

IMUs can measure a variety of physical quantities, including acceleration, angular rates and magnetic field flow.

2.2.4 IMU Applications

Common applications for IMUs include determining direction in a GPS system, tracking motion in consumer electronics such as cell phones and video game remotes, or following a user's head movements in **Augmented reality** [**AR**] and **Virtual Reality** [**VR**] systems. This motion and orientation information also apply to maintaining a drone's balance, improving the heading of a robot vacuum cleaner, and other IoTs and connected home devices. In industrial use, you may use IMUs to align and measure positioning of equipment like antennas. IMUs are also used to help maneuver aircraft, with or without a manned pilot. In the consumer airspace, some in-flight entertainment systems use IMUs in their remotes to add accessibility in addition to touch. A similar approach can be seen with LG Smart TV Remotes which allow users to control the TV's user interface with an intuitive point and click approach instead of directional buttons.

In the automotive sector, IMUs sensor data can be processed by actuators like **Vehicle Dynamics Control** [**VDC**] and **Electronic Stability Program** [**ESP**]. The IMU has direct influence on lateral dynamics, e.g., it conceives the actual driving situation (actual state), conceives the drivers desire (target state), generation of corrective Yaw-moment by braking and de-braking of single wheels.

On Figure 2.7, it can be seen the normal operation of ESP, using the IMU sensor data.

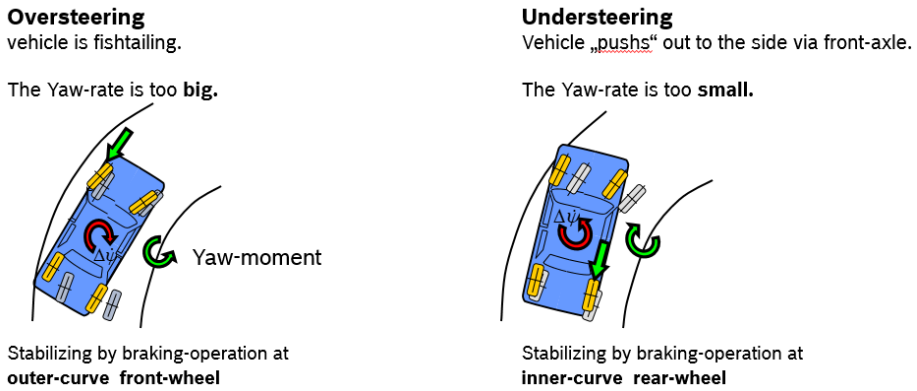


Figure 2.7: The normal operation of ESP using IMU data

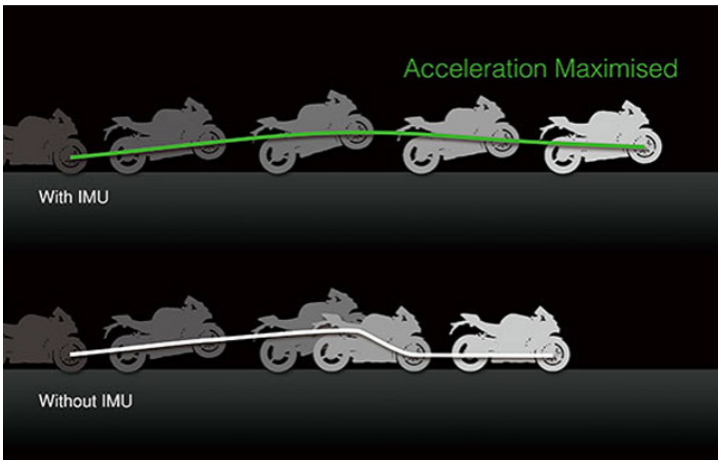


Figure 2.8: Acceleration Graph of Motorcycle with and without an IMU sensor [14]

Figure 2.8 shows a working example of a motorcycle acceleration graph. An IMU alone doesn't optimize a vehicle, but when used with other actuators make it possible to be optimized.

There are many other applications of the IMU in the Automotive area, and the ones shown in Figure 2.9 are only a few examples:

- Electronic Stability Control [**ESP**]
- Active Front Steering [**AFS**]
- Roll Stability Control [**RSC**]
- Rollover Sensing [**RoSe**]
- Front & Side Crash Plausibility
- Hill Hold Control [**HHC**]
- Built In Car Navigation [**NAVI**]

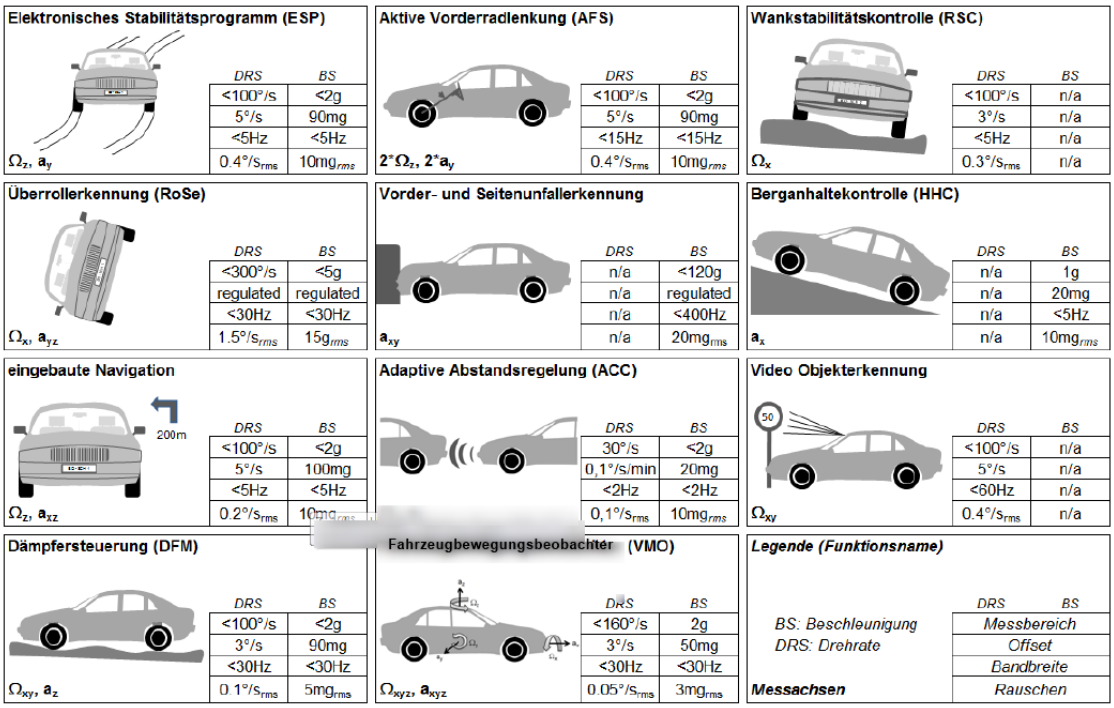


Figure 2.9: Vehicle Functions based on Inertial Signals

- Adaptive Cruise Control [**ACC**]
- Video Object Detection
- Damper Control [**DFM**]
- Vehicle Motion Observer [**VMO**]

Future applications will likely see tighter integration and fusion of the IMU with technologies like **Global Positioning System** [**GPS**], **Radio Frequency** [**RF**] and **Light Detection and Ranging** [**LiDAR**], which will enable accurate localization of people, vehicles and equipment (including autonomous ones), both indoors and outdoors.

2.3 AUTOSAR

AUTomotive Open System ARchitecture [**AUTOSAR**] [15] is a worldwide development partnership of vehicle manufacturers, suppliers, service providers and companies from the automotive electronics, semiconductor and software industry.

2.3.0.1 AUTOSAR Vision

AUTOSAR aims to improve complexity management of integrated E/E architectures through increased reuse and exchangeability of SW modules between OEMs and Suppliers [16], like seen on the Figure 2.10.

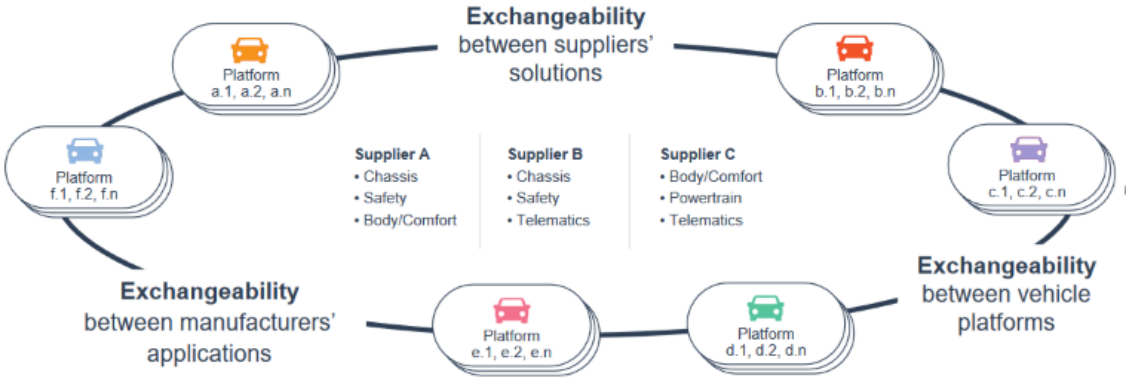


Figure 2.10: AUTOSAR Exchangeability between Suppliers and Manufacturers [16]

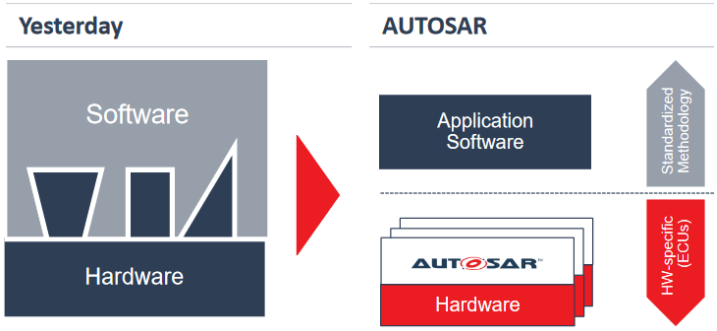


Figure 2.11: How AUTOSAR simplifies the connection between HW and SW [16]

AUTOSAR aims to standardize the software architecture of ECUs. AUTOSAR paves the way for innovative electronic systems that further improve performance, safety and environmental friendliness, as seen in Figure 2.11.

2.3.0.2 AUTOSAR Architecture Layer

AUTOSAR subdivides Embedded-Software into five layers [17]:

- Basic Software [**BSW**]
  - Micro controller Abstraction Layer
  - ECU Abstraction layer
  - Services Layer
- Runtime Environment [**RTE**]
- Application Layer [**APP**]

2.3.0.3 Application Scope of AUTOSAR

AUTOSAR is dedicated for Automotive ECUs. Such ECUs have the following properties:

- Strong interaction with HW (sensors and actuators)
- Connection to vehicle networks like CAN, LIN, FlexRay or Ethernet
- Micro-controllers (typically 16 or 32 bit) with limited resources of computing power and memory (compared with enterprise solutions)
- Real Time System
- Program execution from internal or external flash memory

In the AUTOSAR sense an ECU means one micro-controller plus peripherals and the according SW/configuration.

2.3.0.4 Architecture - Overview of SW Layers (Top View)

The AUTOSAR Architecture distinguishes on the highest abstraction level between three SW layers: **Application**, **Runtime Environment** and **Basic Software** which runs on a **microcontroller [uC]**, as you can see on Figure 2.12.

The AUTOSAR BSW is further divided in its layers: *Services*, *ECU Abstraction*, *Microcontroller Abstraction* and *Complex Drivers*.

The **BSW Layers** are even further divided into functional groups. We can take the Service layer as an example, which can be divided into *System*, *Memory* and *Communication*.

The **Micro Controller Abstraction Layer** is the lowest SW layer of the BSW. It contains the internal drivers, which are SW modules with direct access to the uC and internal peripherals. This layer task is to make the higher SW layers independent of the uC.

The **ECU Abstraction Layer** interfaces the drivers of the Micro Controller Abstraction Layer. It offers an API for access to peripherals and devices regardless of their location (uC internal/external) and their connection to the uC (port pins, type of interface). This layer's task is to make the higher SW layers independent of ECU Hw layout.

The **Complex Drivers Layer** spans from the hardware to the RTE. This layer's task is to provide the possibility to integrate special purpose functionality, e.g. drivers for devices.

The **Services Layer** is the highest layer of the BSW which also applies for its relevance for the Application SW. While access to I/O signals is covered by the ECU Abstraction Layer, the Services Layers offers:

- OS functionality
- Vehicle Network Communication and Management Services
- Memory Services (NVRAM management)
- Diagnostic Services (including UDS communication, error memory and fault treatment)
- ECU state management, mode management

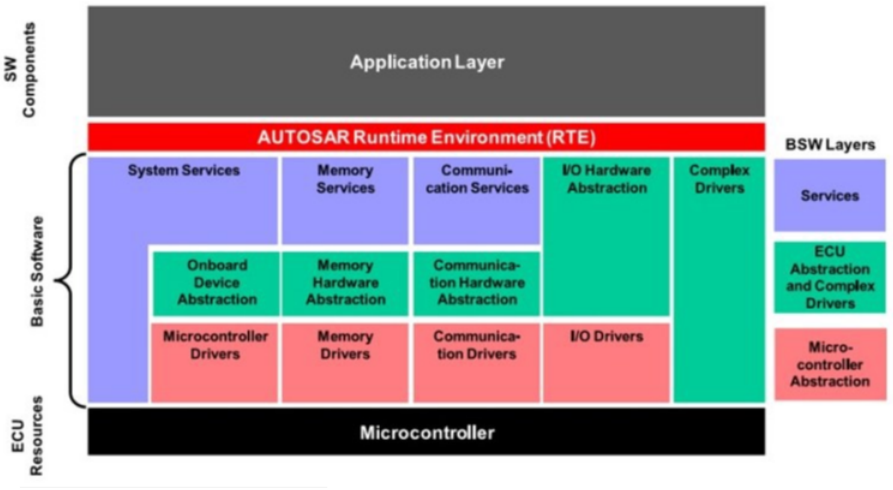


Figure 2.12: AUTOSAR SW Architecture Full Overview [17]

- Logical and temporal program flow monitoring

This layer's task provides basic services for Applications, RTE and BSW modules.

The **RTE** is a layer providing communication services to the Application SW. Above the RTE, the SW Architecture style changes from "layered" to "component style". The AUTOSAR SW Components communicate with other components (inter and/or intra ECU) and/or services via the RTE. This layer's task is to make the AUTOSAR SW Components independent from the mapping to a specific ECU.

The following Figure 2.12 represents the AUTOSAR SW Architecture complete overview.

The Sensors/Actuator AUTOSAR SW Component is a Specific type of AUTOSAR SW Component for sensor evaluation and actuator control. Though not belonging to the AUTOSAR BSW, it is described here due to its strong relationship to local signals. This layer's task is to provide an abstraction from specific physical properties of Hw Sensors and Actuators, which are connected to an ECU, as seen in Figure 2.13.

AUTOSAR also has Libraries. These Libraries are a collection of functions for related purposes.

Libraries:

- Can be called by the BSW modules, SW-Cs, libraries or integration code
- Run in the context of the caller in the same protection environment
- Can only call libraries
- Are re-entrant
- Doesn't require any initialization
- Are synchronous, i.e. they do not have wait points.

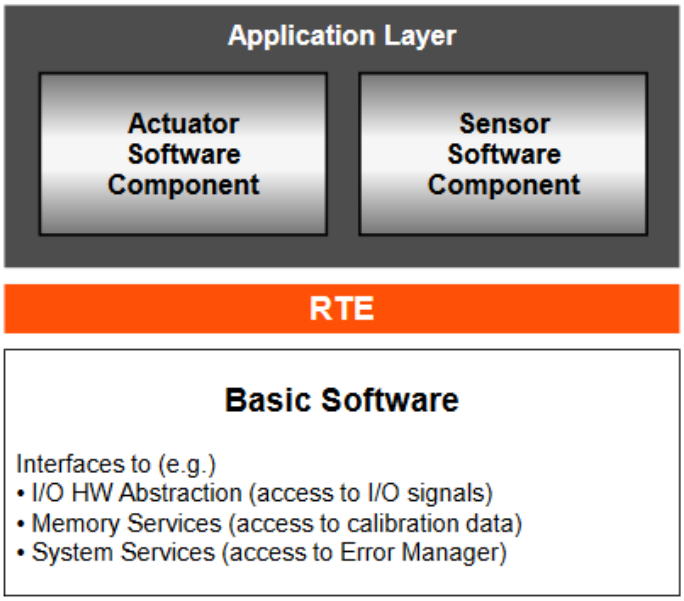


Figure 2.13: Application Layer Interaction with RTE and BSw [17]

AUTOSAR is a major part of the Automotive industry and is also a major requirement for every single SW developed. The SW tested by the Team is mainly located in the APP SW layer of AUTOSAR. But it isn't the only layer being tested. Various other layers of AUTOSAR are tested, like for example the BSW and the Communication layer.

2.4 ISO Standards and ASIL Levels

All SW created has to follow a certain standard to ensure Quality. This is where the ISO Standards and ASIL Levels come in. They are the norms and standards that should be followed to ensure the highest Quality SW and the highest Safety parameters. In order for a ISO standard to be applied, a corresponding ASIL level needs to be upholded.

In this section, the ISO standards and the ASIL levels will be explained, and also why they are interconnected.

2.4.0.1 ISO 26262

ISO 26262, Road Vehicles - Functional Safety [18], is a risk-based safety standard that defines functional safety for all automotive electronic and Electrical [E/E] safety-related systems. This standard is an adaptation of the Functional Safety Standard, IEC 61508, and is applicable throughout the life-cycle of all safety-related systems that include electronic and/or electrical systems.

ISO 26262 specifies four **Automotive Safety Integrity Levels** (ASIL A to D) with ASIL D as the highest safety level. This enables hazards to be classified based on a combination of the likelihood of the event occurring and the probable severity of the event should it occur.

2.4.0.2 ISO 25010

ISO 25010:2011 - Systems and Software Engineering - Systems and SW Quality Requirements and Evaluation - System and SW quality models [19].

ISO/IEC 25010:2011 defines:

- A quality in use model composed of five characteristics that relate to the outcome of interaction when a product is used in a particular context of use. This system model is applicable to the complete human-computer system, including both computer systems in use and SW products in use.
- A product quality model composed of eight characteristics that relate to static properties of SW and dynamic properties of the computer systems. This model is applicable to both computer systems and SW products.

The characteristics defined by both models are relevant to all SW products and computer systems. These characteristics provide consistent terminology for specifying, measuring and evaluating system and SW product quality. The scope of application of the quality models includes supporting specification and evaluation of SW and SW-intensive computer systems from different perspectives by those associated with their acquisition, requirements, development, use, evaluation, support, maintenance, quality assurance and control, and also audit.

Hence safety becomes the fundamental requirement of an automotive application development. For an automotive vehicle, in specific, the functional safety is a very crucial paradigm at every stage of production and decommission.

From this, we take that in the Automotive SW area, the ASIL levels are of critical importance.

2.4.0.3 Definition of ASIL

ASIL refers to Automotive Safety Integrity Level. It is a risk classification system defined by the ISO 26262 standard for the functional safety road vehicles [20].

The standard defines functional safety as "the absence of unreasonable risk due to hazards caused by malfunctioning behavior of electrical or electronic systems". ASILs establish safety requirements based on the probability and acceptability of harm - for automotive components to be compliant with ISO 26262.

The journey of safety life-cycle, of any automotive component, begins with the definition of the system and its safety-criticality at the vehicular level. This is achieved by conducting Hazard Analysis and Risk Assessment (HARA) for the corresponding automotive component, the occurrence of which can be critical for vehicle safety. HARA is followed by identifying the safety goals for each component, which are then classified according to either the QM or ASIL levels, under the ISO 26262 standard.

Systems like airbags, anti-lock brakes, and power steering require an ASIL-D grade - the highest rigor applied to safety assurance - because the risks associated with their failure are the highest. On the other end of the safety spectrum, components like rear lights require only an ASIL-A grade. Head lights and brake lights generally would be ASIL-B while cruise control would generally be ASIL-C, like on the Figure 2.14.

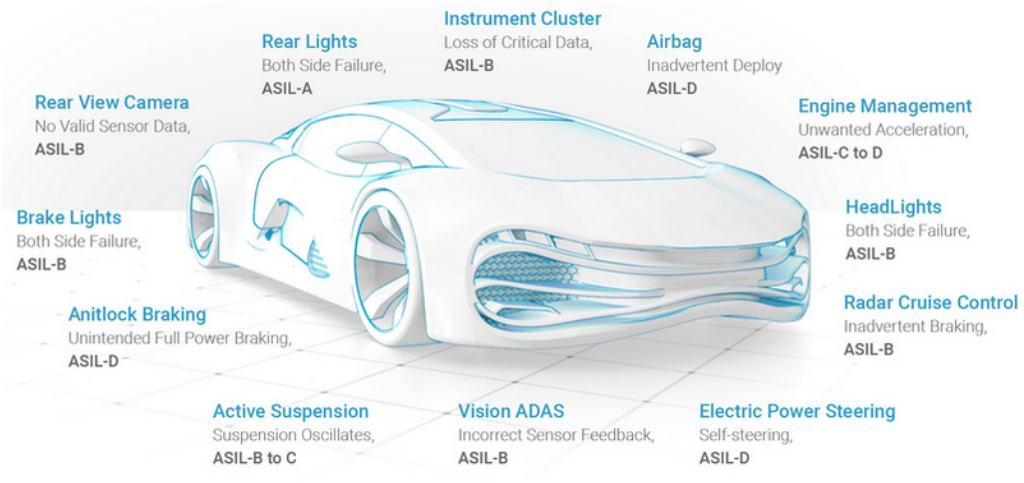


Figure 2.14: Typical ASIL Automotive Classifications [20]

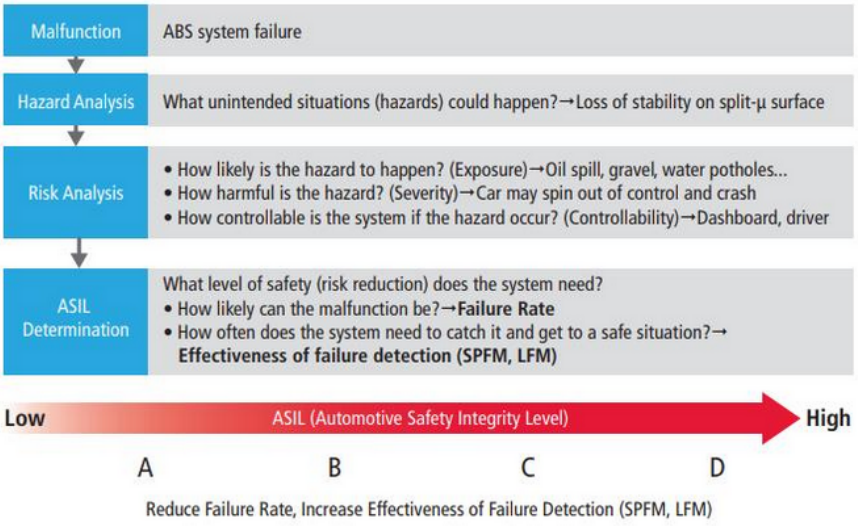


Figure 2.15: ABS System ASIL level [21]

2.4.0.4 Working with ASIL

ASILs are established by performing hazard analysis and risk assessment. For each electronic component in a vehicle, engineers measure three specific variables known as **Severity**, **Exposure** and **Controllability**.

Figure 2.15 demonstrates the steps involved in the determination of ASIL for an ABS, as an example.

For any particular failure of a defined function at the vehicle level, a HARA helps to identify the intensity of risk of harm to people and property.

Once this classification is completed, it helps in identifying the processes and the level of risk reduction needed to achieve a tolerable risk. Safety goal definition as per

ASIL is performed for both Hw and Sw processes within automotive design to ensure the highest levels of functional safety.

These safety levels are determined based on three important parameters:

Exposure (E) This is the measure of the possibilities of the vehicle being in a hazardous or risky situation that can cause harm to people and property. Various levels of exposure such as E1 (very low probability), E2 (low probability), E3(medium probability), E4 (high probability) are assigned to the automotive components being evaluated.

Controllability (C) Determines the extent to which the driver of the vehicle can control the vehicle if a safety goal is breached due to failure or malfunctioning of any automotive component being evaluated. The order of controllability is defined as: C1<C2<C3 (C1 for easy to control while C3 for difficult to control)

Severity (S) Defines the seriousness or intensity of the damage or consequences to the life of people (passengers and road users) and property due to safety goal infringement. The order of severity is: S1 for light and moderate injuries, S2 for severe and life-threatening injuries, and S3 for life-threatening incidences.

2.4.0.5 Benefits of ASILs

ISO 26262 is a goal-based standard that's all about "preventing harm". Despite their challenges, ASIL classifications are intended to "prevent harm" and help us achieve the highest safety rating possible for myriad automotive components across a long and often disjointed supply chain.

Some of the best benefits include establishing safety requirements to mitigate risks, managing and tracking safety requirements and ensuring that standardized safety procedures have been followed in the final product.

Due to the complexity of the SW present on the current ECUs, more and more clients require ASIL D in all things related with Automotive area. The IMU under presentation in this thesis is fully tested and developed following the ASIL D standards.

2.5 Engineering Modules and Processes

Nowadays more and more clients want SW that follows certain standards, and the development process of the IMU is no different. In this section an overview of ASPICE is given, and explain why it is a reference in terms of traceability and evidence of production of SW Development with high grade of quality.

2.5.0.1 ASPICE

Automotive Software Performance Improvement and Capability Determination [ASPICE] is a standard made by German car makers [22]. It provides rough guidelines to improve your software development process and to assess suppliers. This means that practically the whole European automotive industry must follow ASPICE.

AutomotiveSPICE is derived from the generic SPICE (ISO/IEC 15504) standard. While there are other instances, ASPICE seems to be the only one which got any traction.

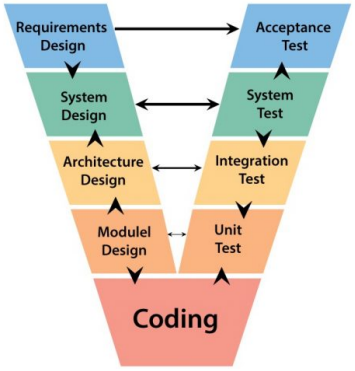


Figure 2.16: Example of a V-Model [23]

It builds on the V-Model (Figure 2.16), explained in the following subsections, which means for every Sw development level since the requirements to the source code there is a corresponding test level. The general idea is this sequence of processes:

1. Eliciting requirements from the customer.
2. A Requirements Engineers maps these into system requirements. This step is necessary to restructure customer requirements into a structure you can work with. It also provides a place to include requirements from other stakeholders.
3. A system architect breaks down the requirements into logical services and system components. This includes design decisions like what to do in hardware, in software, where to run what, and how to communicate.
4. For each software component, you derive software requirements from the system requirements. You can usually distinguish system and software requirements by their wording. Something concerning the "car" is a system requirement, while software requirements are often about input and output signals of services and processing logic.
5. A software architect breaks down the software requirements of a service into units.
6. The developer designs and implements each software unit. This can happen in code or with more abstract models (usually of state machines) which are transformed into code.

Each work item created during the SW development phase have a corresponding work item during the SW Testing phase:

1. Unit Tests for the design. Does the code match the design? Are non-functional requirements (like not crashing, for example) fulfilled?
2. Integration tests for the software architecture. Does the composition of units into a service still work?

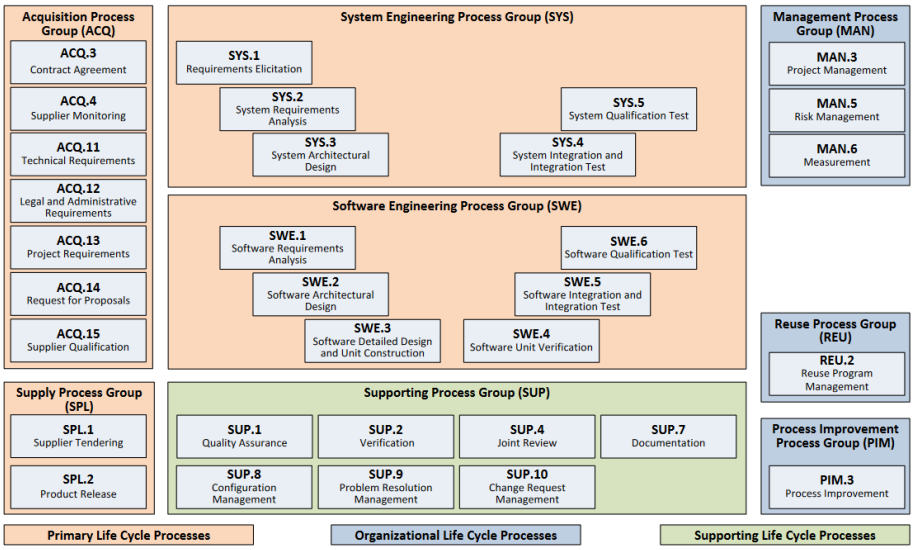


Figure 2.17: Automotive SPICE Process Reference Model - Overview [22]

3. Software qualification tests for the software requirements. Does a service match its requirements? So far, there was no need to use the actual hardware platform since we only test the software.

4. System integration tests for the system architecture. Composing all the services into the full system, does it work?

5. System qualification tests for the system requirements. Does the whole system/car match the requirements?

6. Acceptance test done by the customer.

In addition to these V-Model processes there are also supporting and management processes.

ASPICE itself is generic and does not specify concrete tools or methodologies. An agile approach can be used with ASPICE, where you would start with very few requirements but would still do the whole V-model.

An ASPICE assessment results in ratings of multiple processes from level 0 to 5, classified from "not achieved"(N) to "fully achieved" (F).

ASPICE defines a set of base practices, as seen in Figure 2.17.

Traceability is a big topic which occurs across all processes in the V-model as it connects the work products. It becomes important when the customer claims that you do not fulfill one of their requirements. It helps you to either fix the problem or trace the code back to another customer requirement to reveal an inconsistency. Also, traceability enables you measure the project progress by keeping track on which requirements are done.

A weakness of ASPICE is that an assessment only rates a single snapshot in time. If you assess a supplier that is good enough, to actually reap the benefits of doing it right, you must follow ASPICE from the beginning.

2.6 Automotive SW Testing

2.6.1 Software Testing

Testing, also known as an analytic means for assessing the quality of software, is one of the most important phases during the SW development process in regard to quality assurance. It *"can never show the absence of failure"* [24], but it aims at increasing the confidence that a system meets it's specified behavior. Testing is an activity performed for improving the product quality by identifying defects and problems. It cannot be undertaken in isolation. Instead, in order to be in any way successful and efficient, it must be embedded in an adequate software development process and have interfaces to the respective sub-processes.

2.6.2 Evolution of Automotive SW Testing

When General Motors introduced the first software based engine controller unit [ECU [1]] in 1980, nobody could foresee how critical software would become for the automotive industry in terms of safety, security and cost [25]. At the time, complexity was not high and software testing was done primarily in the car. Nowadays the big bulk of testing has already happened using other methods that were introduced for the sake of efficiency, quality and obviously cost savings. Automotive companies followed a divide and conquer approach where the different subsystems (PowerTrain, Chassis, Body, etc) are tested separately. **Hardware in-the-loop [HIL]** was then born. Fueled by the adoption of model-based design during the design of new control functions, physical models of the plant started to be reused for functional software testing on the ECU before a prototype of the physical plant was available. This has been key to increasing the quality and reducing the cost of developing cars with more and more software running on them. Clearly the trend in testings evolution is the move from full car, to subsystem test-beds, to HIL systems, all with the goal of enabling teams to test software earlier in the design cycle. Completely removing the dependency on physical hardware to develop, integrate and test software can gain automotive companies valuable months, that will result in higher quality products. Testing methods are continuously evolving and virtual prototypes of the digital ECU hardware are the next natural step on this path.

2.6.3 Challenges in Automotive Testing

"Even the Highest quality organizations have trade offs when it comes to their testing coverage." [26]

In Japan, Europe and the United States automotive manufacturers are aiming to enhance automotive functions by using software. In the automotive industry, many companies are **original equipment manufacturers [OEM]**, meaning they make most

[1] Another way of using the same Acronym, its use depends on the situation its used on

of their automotive parts in-house. While, on other hand, Sony, Panasonic, and other electronic manufacturing companies are researching ways to increase their sales and profit.

2.6.3.1 Trade-offs of Coverage and Risk

Even the highest quality organizations have trade-offs. As old as SW Testing, Automotive SW Testing has the same issues. Companies have to balance coverage and risk with cost and time. High risk needs more testing, more people, more time and more coverage. Less risk often means less testing, less time and less coverage. No one wants to make these trade-offs - specially in a safety-critical product. In a regulated industry, these trade-offs are often standardized levels demanded for regulatory compliance.

2.6.4 SW testing for Safety-Critical Systems

The industry faces a significant challenge: Thriving to reach the goal of releasing cars with zero software defects. Recalls and redesigns due to SW problems often show the magnitude of this challenge and have grown exponentially over the last decade. There are major economic implications for manufacturers since the cost of recalling a vehicle can be huge. Manufacturers already spend about 75% of the cost of SW development on SW testing, and is expected to grow in accordance with the amount of tests needed to run by the manufacturers.

But simply increasing the number of tests is not always the best way to reduce defects, Improving tests also improves quality, so that every corner case can be triggered by normal operation.

Standards like **ISO 26262** [18] address the planning and development of safety-critical systems and place further demands on SW Testing. ISO 26262 provides an automotive-specific, risk-based approach based on ASIL. It specifies the requirements and recommended guidelines for validation of the safety levels including fault-injection testing.

2.6.4.1 Fault-Injection Testing

Fault Injection helps to determine whether the response of a system matches its specification in the presence of faults. It helps system engineers understand the effects of faults on the target system and what would be its reaction. Fault injection can also improve test coverage of safety mechanisms by covering corner cases that are difficult to trigger during normal operation.

2.6.4.2 Fault Categories

We can categorize faults as either SW or **Hardware [HW]** faults. Within the hardware category, faults are either:

Permanent Triggered by component damage;

Transient Triggered by environmental conditions, also known as soft erros;

Intermittent Caused by unstable HW.

2.6.5 ISTQB

2.6.5.1 History and Importance

ISTQB is the International Software Testing Qualifications Board [27]. It is composed of representatives from each existing national board, such as the **ASTQB**, the **American Software Testing Qualifications Board**. The ISTQB decides on the standards for certification and accreditation as an ISTQB training provider. Working parties within ISTQB are responsible for developing and maintaining the various software testing syllabi and exams.

At both the Foundation Level and the Advanced Level, the programs supporting the ISTQB Certified Tester bridge the gap between research, standard training and industry practice. ISTQB is utilized to certify SW Test Engineers, with information regarding various methods and standards and where good practices are given in order to produce quality SW.

In Europe, industry demands pushed the development of this certification program. The success of the program can certainly be attributed to its transparent structure, volunteer and non-profit character, and the representation of national interests through national boards.

2.6.6 Testing Method

Most of the work done regarding testing is based on the ISTQB standards, e.g., the 2018 Syllabus which are based on SW Quality standards.

From this study guide, we can take out that for any given project, the objective served by testing may include:

- To evaluate the requirements, user stories, design, and code;
- To verify that all specific requirements have been fulfilled;
- To validate that the test object is complete and works as the users and stakeholders expect;
- To build confidence about the level of quality of the test object;
- To prevent defects;
- To find defects;
- To provide sufficient information to stakeholders to allow them to make informed decisions, especially regarding the level of quality of the test object;
- To reduce the level of risk to software quality;
- To comply with contractual, legal, or regulatory requirements or standards.

From this guide we can also know that Debugging and Testing are completely different points, like seen in Figure 2.18. Debugging is the development activity that finds, analyses and fixes the defects. Subsequent conformation testing checks whether the

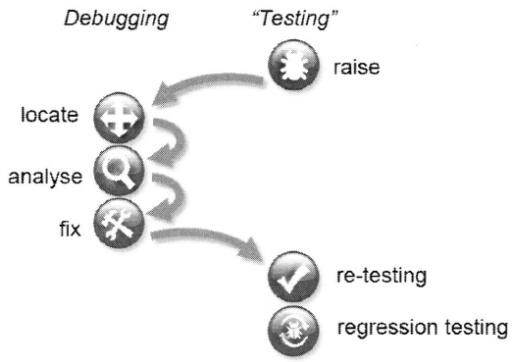


Figure 2.18: Debugging and Testing Order [28]

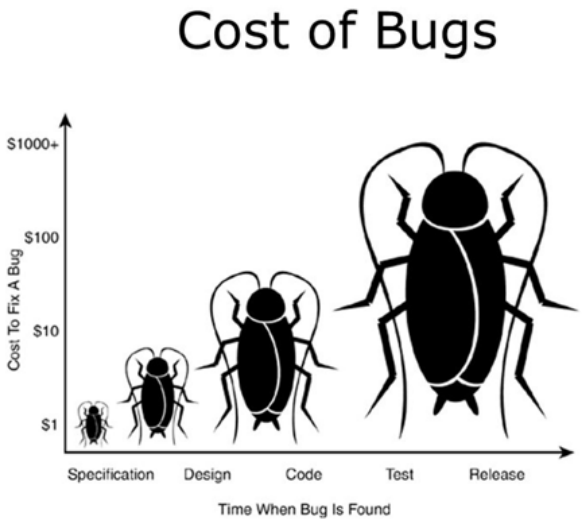


Figure 2.19: Cost of Code Bug[28]

fixes resolved the failures. In some cases, testers are responsible for the initial test and the final confirmation test, while developers do the debugging and associated component testing.

Rigorous testing of software and systems, and their associated documentation, can help reduce the risk of failures occurring during operation. When defects are detected, and subsequently fixed, this contributes to the quality of the software or system. Figure 2.19 shows us that the sooner a bug, defect and/or failure is detected, the lower the cost of it to the whole project.

The fundamentals of testing are based on **Errors, Defects and Failures**.

For example: A person can make an **error (mistake)**, which can lead to the introduction of a defect (fault or bug) in the software code or in some other related work product. An error that leads to the introduction of a **defect** in one work product can trigger an error that leads to the introduction of a defect in a related work product. If a defect in the code is executed, this may cause a **failure**, but not necessarily in all

circumstances.

From this, we can establish that **defects, root causes and effects** are also important. The root causes of defects are the earliest actions or conditions that contributed to creating the defects. Defects can be analyzed to identify their root causes, so as to reduce the occurrence of similar defects in the future.

2.6.6.1 Testing Principles

A number of testing principles have been suggested over the past 50 years and offer general guidelines common for all testing. ISTQB defines the following ones as key:

1. **Testing showns the presence of defects, not their absence** Testing cannot prove that there are no defects. But if no defects are found, testing is not proof of correctness.
2. **Exhaustive testing is impossible**
3. **Early Testing saves time and money**
4. **Defect Clustering** A small number of modules usually contains most of the defects discovered during pre-release testing, or is responsible for most of the operational failures.
5. **Pesticide Paradox** If the same tests are repeated over and over again, eventually the same set of tests cases will no longer find any new defects. If there is a need to detect new defects, existing tests and test data may need changes, and new tests may need to be written.
6. **Testing is Context Dependent** For example, safety-critical software is tested differently from an e-commerce site mobile app.
7. **Absence-of-Errors Fallacy** It is a mistaken belief to expect that just finding and fixing a large number of defects will ensure the success of a system.

2.6.6.2 Testing within a life cycle model

A SW development life-cycle model, like on Figure 2.20, describes the types of activity performed at each stage in a Sw development project.

In any Sw Development life-cycle model, there are several characteristics of good testing:

- For every development activity there is a corresponding testing activity.
- Each test level has test objectives specific to that level.
- The analysis and design of tests for a given test level should begin during the corresponding development life cycle.
- Testers should be involved in reviewing documents as soon as drafts are available in the development life cycle.
- Test levels can be combined or reorganized depending on the nature of the project or the system architecture.

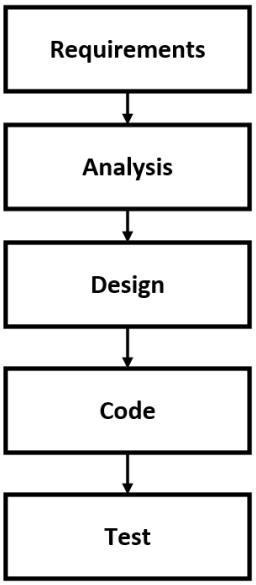


Figure 2.20: Software Development Life-Cycle Model [28]

2.6.6.3 Test Levels

A group of testing activities that are managed and organized together. Each test level has a specific objective.

Component Testing Units, modules, programs, objects, classes. Searches for defects and verifies the functioning of software bits that are separately testable. May include testing of functionality, specific non-functional characteristics (memory leaks), as well as structural testing. Usually done by developers - often the author of the code. Defects fixed when found, not usually recorded.

Integration Testing Can be done at a component level (after component testing) or system level (after system testing). Tests interfaces between components, interactions to different parts of a system or interfaces between systems. Concentrates on the interaction between items, rather than functionality. Can involve both functional and structural approaches.

System Testing Concerned with the behavior of the whole product as defined by the scope of a development project. The test environment should correspond to the final target or production environment as much as possible. Should cover both functional and non-functional requirements. Often carried out by an independent test team.

Acceptance Testing Often the responsibility of the customers or users of a system. Goal is to establish confidence in the system, finding defects is not the main focus. Can involve testing both functional and non-functional aspects of a system.

Figure 2.21: Type of Box Testing Techniques[28]

2.6.6.4 Types of Tests

Focus on a particular objective, verify based on a reason or target for testing.

Functional Testing Testing of "what" the system does. Testing of functions that a system, subsystem or component is to perform. The functions are "what" the system does. May be performed at all test levels considers the external behavior of the software (black-box testing) includes security testing and interoperability testing.

Non-Functional Testing Testing of non-functional software characteristics. It is the testing of "how" the system behaves. May be performed at all test levels.

Structural Testing Also known as white-box testing. Testing of software structure.

Change based Testing Testing related to changes. Tests must be repeatable. For example, **Confirmation testing**: re-testing to confirm that a defect has been fixed. **Regression testing**: re-testing of unchanged areas.

Figure 2.21, exemplifies the amount of knowledge of the system needed for the corresponding box technique.

2.6.6.5 Sequential Development Models

A Sequential Development Model describes the software development process as a linear, sequential flow of activities. This means that any phase in the development process should begin when the previous phase is complete, like shown on Figure 2.20.

The Waterfall model In the Waterfall model, the development activities are completed one after another - Figure 2.20 - In this model, testing is viewed as a separate phase which takes place after development has been completed.

The V-model The V-model integrates the testing process into the development process, implementing the principle of early testing - Figure 2.16 - the V-model includes test levels associated with each corresponding development phase.

Following the norms and processes referred in this section will ensure that Quality SW is delivered on time. And using the ISTQB testing methods is a perfected example of how shortening the testing times and lessening the testing topics contributes to better Quality assurance and timeline.

2.7 Tools and Protocols

In this section, an overview is given about the tools and protocols utilized daily by the Team, and how this tools and protocols affect the SW being developed.

2.7.1 Application Lifecycle Management

Bosch, like most companies, deploy software updates daily, sometimes with a higher frequency. In order to reach such levels of productivity, businesses need a plan for managing their software from beginning to end. That is where an **Application Lifecycle Management [ALM]** tool comes in hand [29]. ALM helps businesses make smart decisions about their software and manage it efficiently over time.

ALM is an integrated system of people, processes and tools that manage the life of an application from concept to retirement. ALM is similar to **Software Development Lifecycle (SDLC)**, but more comprehensive in scope.

Depending on the software development methodology, ALM might be split into distinct phases or fully integrated into a continuous delivery process, as seen in Figure 2.22.

Figure 2.22: Application Lifecycle Management [30]

There are several key features to look for when utilizing the ALM tools:

1. Requirements Management
2. Estimation and Planning
3. Source Code Management
4. Testing and Quality Assurance
5. Deployment
6. Maintenance and Support
7. Version Control
8. Application Portfolio Management
9. Real-time Planning and Team Communication

In the IMU project, the ALM tool used is based on Rational Jazz Team Server's IBM Rational Software, known as Rational Solution for Collaborative Lifecycle Management.

From this ALM server, the team has access to various tools needed to maintain the artifacts and its relationships in order to achieve the aimed project's ASPICE level.

2.7.2 Protocols

In this section the various Protocols used during testing will be discuss.

2.7.2.1 CAN Protocol

Controller Area Network [CAN] bus is a serial data communication protocol which was developed by Bosch in the 1980s to solve data exchange among numerous control and test equipment in modern vehicles. CAN bus has been widely used in electric vehicle control systems, due to its advantages such as good real-time feature, high reliability, quick communications rate, simple structure, good interoperability, perfect error handling mechanism of bus protocol, high flexibility, low price and so on.

The usage of the CAN protocol with the help of the CANoe tool, it is possible to realize the total digital simulation of bus application system based on total virtual nodes. CANoe supports the whole process of the bus development - from the initial design, simulation and final test analysis, and test system for vehicle body CAN network.

According to the requirement analysis, the design of the simulation topology and test system for vehicle body CAN network can be exemplified as shown in Figure 2.23.

But CAN wasn't always this simple to use. Before one way to exchange data between ECUs were point-to-point connections where each signal uses a dedicated wire, like in Figure 2.24. This strategy was first used in the field of electronics in motor vehicles. Which lead to the "Old Paradigm" - "For every signal a dedicated wire". But with growing numbers of ECUs, sensors and actuators this approach became too complex and error-prone, not to mention the cable-harnesses becoming bigger and heavier [32].

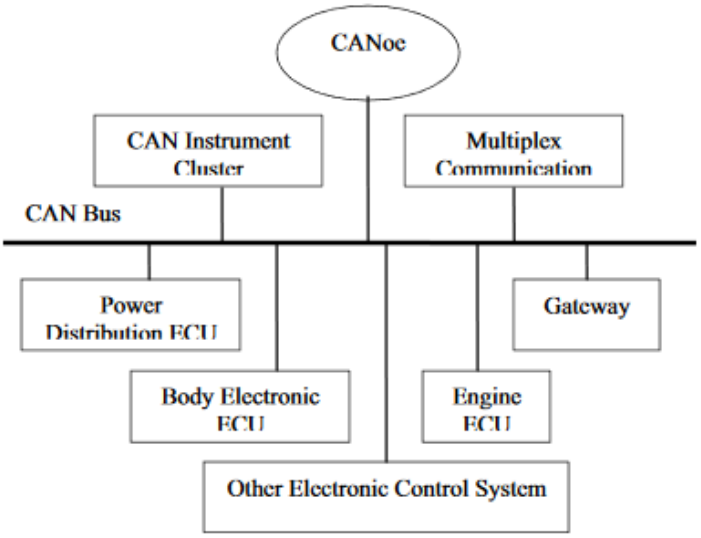


Figure 2.23: CAN Network Topology [31]

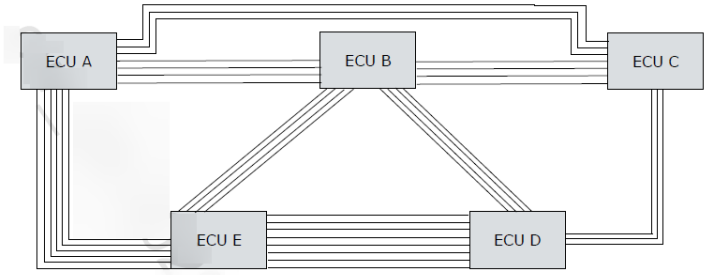


Figure 2.24: Point-To-Point Networking in Motor Vehicles [31]

Due to these consequences, a completely new approach was sought after. The Bus communication was introduced as a solution, where all the ECUs time-share a common communication medium. The idea is that there is one physical communication medium for all ECUs, like in Figure 2.25. The advantages were that the wire harnesses remained manageable and light, errors could be diagnosed and extensions to the network were easy to establish.

CAN Bus Networking involves several services and requirements to proper function, shown in Figure 2.26. From this we take it that CAN specific requirements are on the **Open Systems Interconnection [OSI]** Layers 1 and 2.

CAN technology comprises the CAN and CAN FD protocols and two different CAN Physical Layers. The CAN protocol covers the entire second layer and the **Physical Signaling [PLS]** of the first layer. The sub-layers, LLC, MAX of the Data Link Layer together with sub-layer PLS from the Physical Layer are specified in ISO 11898-1.

CAN High-Speed transceivers are generally used in the Powertrain and chassis area of the vehicle, while CAN Low-Speed transceivers are used in the convenience area.

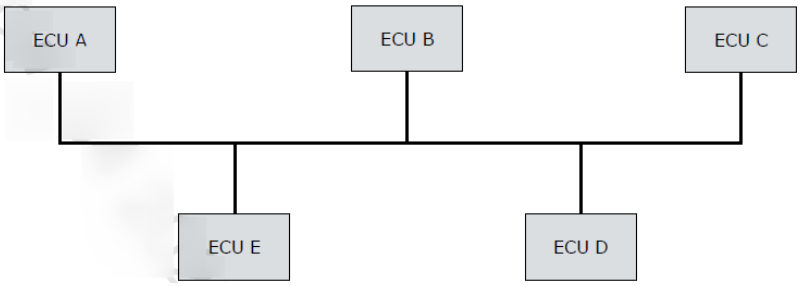


Figure 2.25: Bus Networking in Motor Vehicles [31]

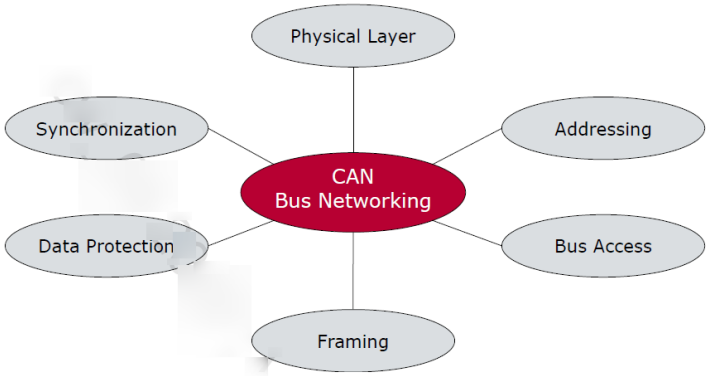


Figure 2.26: Services Provided by Bus Networking [31]

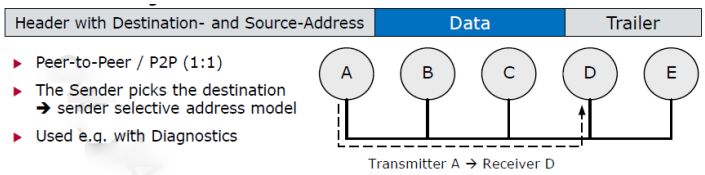


Figure 2.27: Node Addressing [31]

There are two address models for bus communication:

- **Node Addressing** is used to transmit information between two fixed nodes, Figure 2.27. This type of addressing always requires a Destination Address of the receiver node to assure exclusive delivery of the frame to this node only.
- **Broadcast Addressing** does not denote source or destination of data but in contrast identifies its contents, Figure 2.28. Information is not transmitted between two predefined nodes. Instead, all nodes may receive the information transmitted. This enables to send data to multiple receivers by one frame only.

CAN uses the broadcast addressing method, seen in Figure 2.29. A receiving node does not necessarily process all messages transmitted on the bus. There is a means to filter incoming messages. The so-called Acceptance Filter can be configured to allow

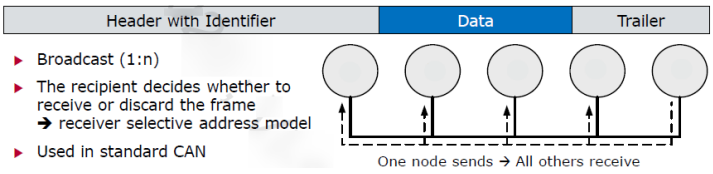


Figure 2.28: Broadcast Addressing [31]

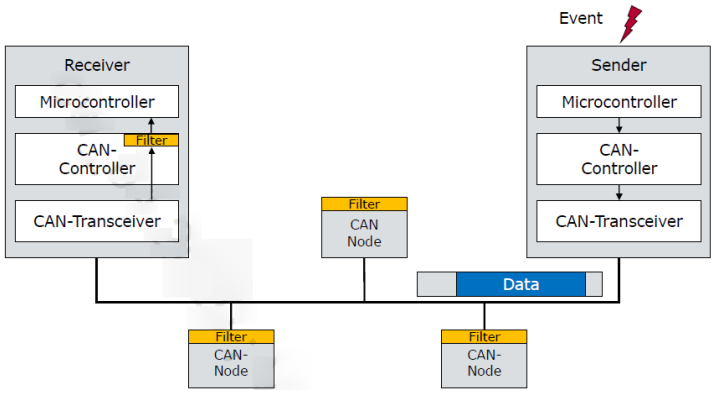


Figure 2.29: Broadcast Addressing and Acceptance Filter [31]

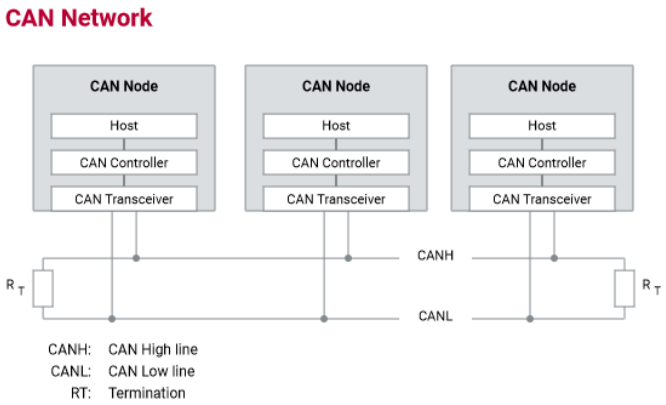


Figure 2.30: CAN Nodes Network [31]

only a subset of the messages to pass. All other messages are discard and not forward to the application running on the microcontroller.

In a CAN network (Figure 2.30), the CAN nodes differ in the number of CAN messages, each at a cycle of ten milliseconds, while another CAN node just needs to receive on CAN message every 100 milliseconds. These obvious differences have resulted in two fundamental CAN controller architectures: CAN controllers with and without object storage.

Physical signal transmission in a CAN network is based on transmission of differential voltages (differential signal transmission). This effectively eliminates the negative effects of interference voltages induced by motors, ignition systems and switch con-

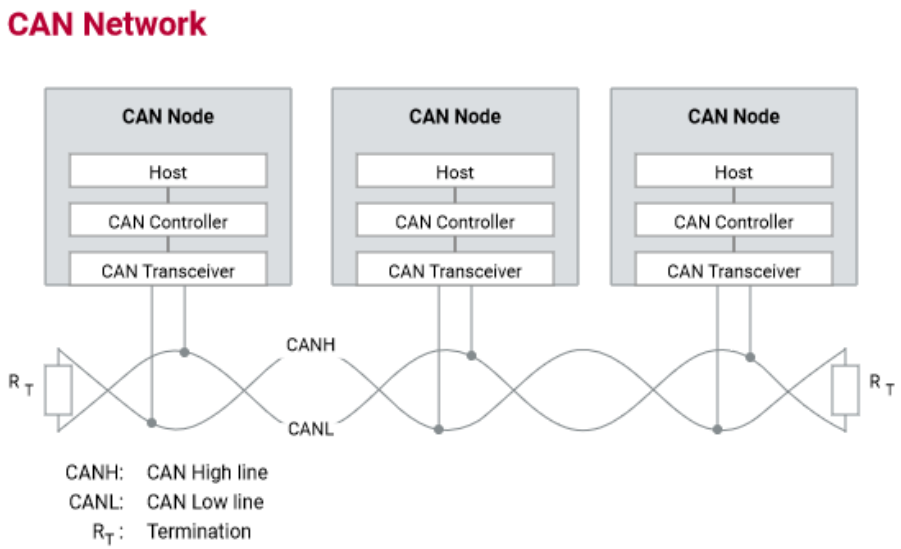


Figure 2.31: CAN Network relation with CANH and CANL [31]

tracts. Consequently, the transmission medium (CAN bus) consists of two lines: CAN High Line (CANH) and CAN Low Line (CANL) 2.31.

Physical signal transmission in a CAN network is based on differential signal transmission. The specific differential voltages depend on the bus interface that is used. A distinction is made here between the high-speed CAN bus interface and the low-speed bus interface.

Safety-critical applications, such as those in the powertrain area, place severe demands on a communication system's availability. So it would be disadvantageous to assign responsibility for bus distribution to just a single bus node. Failure of this vulnerable bus node would cause all communication to fail. A much more elegant solution is to decentralize bus access, so that each bus node has the right to access the bus. That is why a CAN Network is based on a combination of multi-master architecture and line topology: essentially each CAN node is authorized to place CAN messages on the bus in a CAN Network. The transmission of CAN messages does not follow any predetermined time sequence, rather it is event-driven, like on Figure 2.32.

The communication channel is only busy if new information actually needs to be transmitted, and this allows for very quick bus accesses. In principle, real-time data transmissions on the magnitude of milliseconds are no problem in a CAN network, because of the ability to quickly react to asynchronous events and very high data rates up to 1 MBit/s.

For transmitting user data, ISO 11898-1 prescribes the so-called data frame. A data frame can transport a maximum payload of eight bytes. For that there is the so-called data field, which is framed by many other fields that are required to execute the CAN communication protocol. They include the message address (identifier or ID), data length code [DLC], checksum (cyclic redundancy checks sequence - CRC sequence) and RX acknowledgment located in the acknowledgment field, like in Figure 2.33.

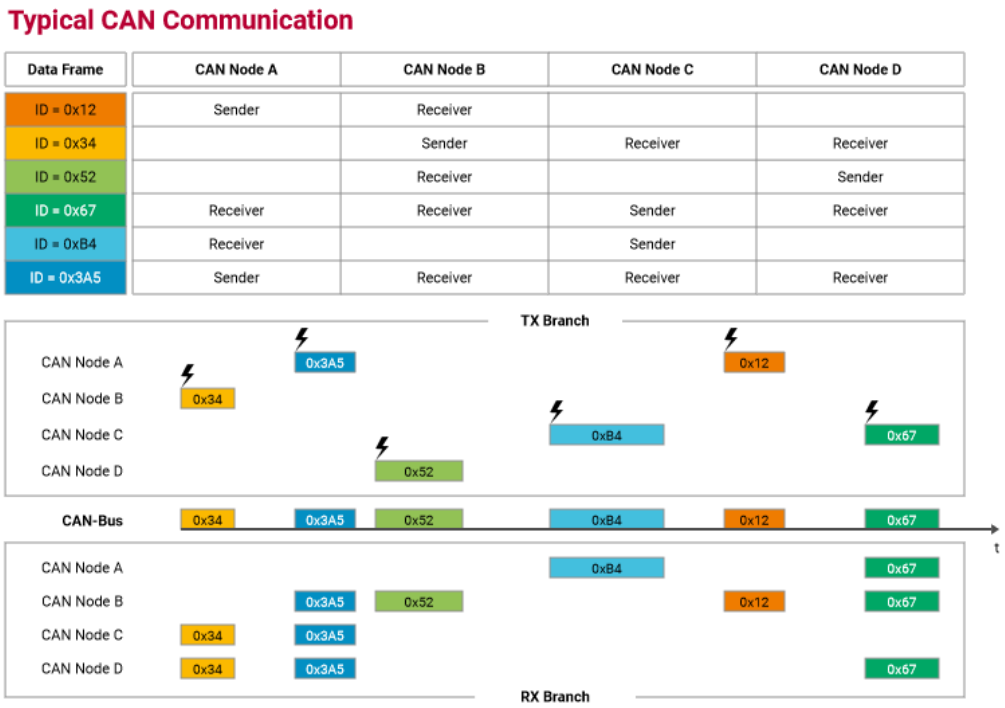


Figure 2.32: Typical CAN Communication [32]

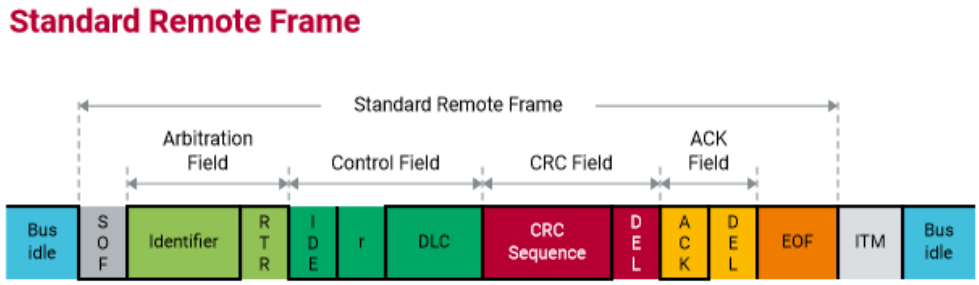


Figure 2.33: Standard Remote Frame [32]

The CAN communication protocol is extremely important for the project being developed. CAN is used as a means of communication with the sensor, as if it was simulation a real-time vehicle testing process. More on this topic will be discussed in **Chapter 3 - CAPL Testing**.

2.7.2.2 UDS Protocol

UDS stands for **Unified Diagnostic Services** and it is a diagnostic protocol to identify any kind of error or fault belonging in vehicles. Its implementation specified into ISO-14229 which offers some unified (Uniform for all ECU suppliers) services through which a UDS enabled diagnostic tool can perform maintenance (when the vehicle is in

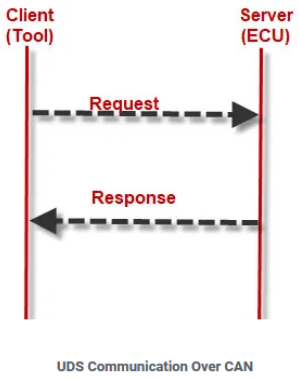


Figure 2.34: UDS Communication between Client and Server [33]

the garage) diagnosis for faulty ECU.

ISO offers some services under UDS with fixed functionality. As per requirement UDS enabled external diagnostic tool sends requests to the UDS enabled server ECU and the server ECU answers back to the client. Figure 2.34 can be seen as an example of this type of communication.

The Complete UDS Protocol development was documented in three segments by ISO's technical team:

1. Diagnostic Requirement and Specification
2. Session Layer Services
3. Diagnostic Implementation

In order to get rid of compatibility issues OEMs and Suppliers agreed to rely on a standard protocol, that is known as UDS.

UDS specification works on the AUTOSAR Application Layer (14229-1) and Session Layer (14229-2). But ultimately Server (ECU) and Diagnostic Tool (Client) are connected over an CAN Physical Medium so UDS request and response should follow the Bosch CAN specification to send data over CAN Bus. Because of this many more standards at different OSI layers come in to picture. Figure 2.35 shows the applicable standard over each OSI Layer. In terms of the work being developed, the SW will focus mainly in the application layer. This means that the test will give more focus to this specific layer.

There are basically four types of frames in a UDS Protocol

- Request Frame with Sub-Function ID
- Request Frame without Sub-Function ID
- Positive Response Frame
- Negative Response Frame

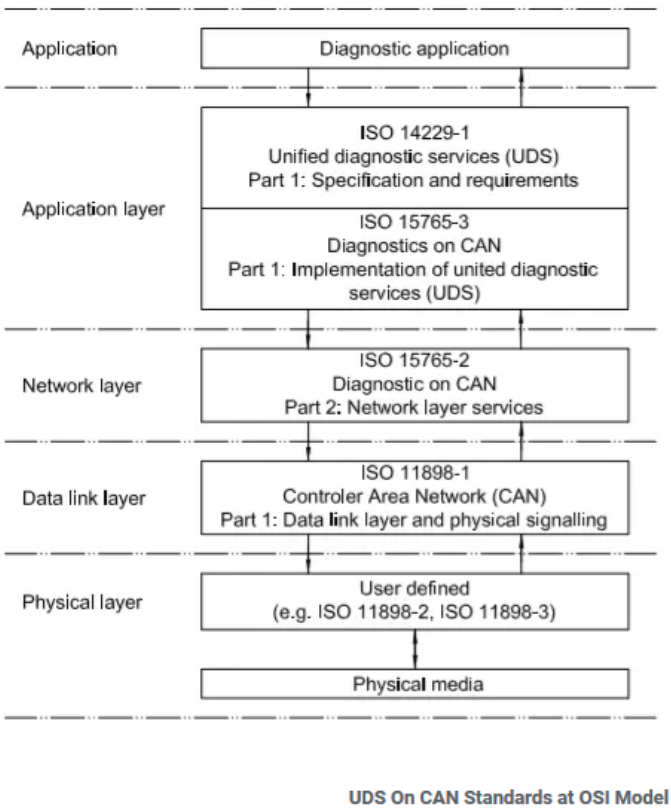


Figure 2.35: The Various Applicable Standards over the OSI Layers [33]

The UDS Protocol Application layer implementation are categorized in two segments by the ISO norms:

- UDS Specification and Requirements (ISO 14229-1) - Listed UDS Services
- UDS Implementation Over CAN (ISO 14229-3) - Implementation of UDS by 14229-1 on CAN

All UDS are defined in 14229-1 ISO Specification and this standard is independent of data link while the ISO 14229-3 explains the data link requirement through which the diagnostic tool and ECU communicate over CAN.

There are many services and each service may have many sub-functions to perform various kinds of functions in the ECUs. So the Application Layer of the Diagnostic Services are categorized into five functional units on the basis of functions they perform:

Diagnostic and Communication Management Its a function group of the UDS services that contains the most basic services needed (From Diagnostic Session Control to ECU Reset)

Data Transmission This Functional Unit has services (Table 2.1) used for the data transmission between the server and the client.

Table 2.1: Data Transmission

Read Data by Identifier	0x22
Read Memory by Address	0x23
Read Scaling Data by Identifier	0x24
Read Data by Periodic Identifier	0x2A
Dynamically Defined Data Identifier	0x2C
Write Data by Identifier	0x2E
Write Data by Memory Address	0x3D

Table 2.2: Stored Data Transmission

Service	Request Service ID	Response Service ID	SubFunction
Clear DTC Service	0x14	0x54	No
Read DTC Service	0x19	0x59	Yes

Stored Data Transmission Its a Functional Unit that contains the services use to read or clear data, as seen on Table 2.2

Input/Output Control This unit has only one service ID, and is used by the testers to control the Input signals/internal functionalities or Output of UDS enabled Server ECU

Remote Activation of Routine This function unit only has one service ID (0x31 "Remote Activation of Routine") and is used by the client to perform various project specific functions through OEM-Specific routine identifier.

Upload/Download Functional Unit This unit has the services through which a client can write or read new firmware from the server's memory (Table 2.3)

Table 2.3: Upload/Download Functional Unit

Service	Request Service ID	Response Service ID	SubFunction
Request Download	0x34	0x74	No
Request Upload	0x35	0x75	No
Transfer Data	0x36	0x76	No
Transfer Exit	0x37	0x77	No

This protocol is always used during SW testing performed by the Team. Since the most basic act of turning on the sensor, to a more complex one like corrupting the data memory of the sensor. More on this topic will be discussed in **Chapter 3 - CAPL Testing**.

2.7.2.3 XCP Protocol

The XCP is an evolution of the CCP (CAN Calibration Protocol) [34]. The difference between the CCP and XCP is that the CCP protocol only supports the CAN Protocol (CCP over CAN) for the calibration and measurement whereas the XCP protocol is available for the CAN, FlexRay and Ethernet protocol.

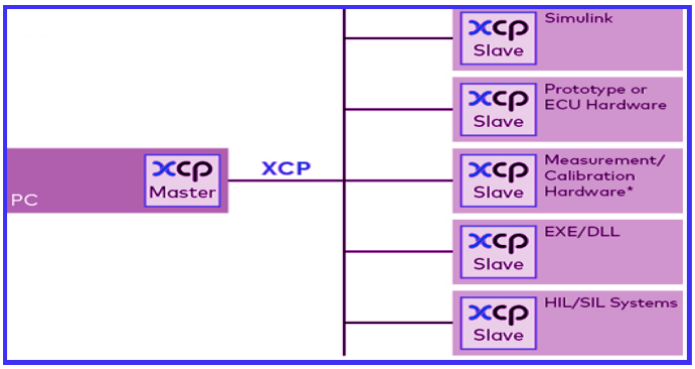


Figure 2.36: XCP Master-Slave Communication [34]

The XCP (Extended CAN Calibration Protocol) is a universal measurement and calibration based networking protocol originated from the ASAM (Association for Standardization for Automation and Measuring Systems) for connecting the calibration systems to the ECU.

It is commonly used in the automotive industry to measure and calibrate the ECUs. The ECUs can correctly detect and operate the vehicle on the road so that no accidents occur, improving human safety.

Mostly, the engineers use it to read the measurement data and write the parameters to the ECU during the development, testing, and in-vehicle calibration. The main objective of the XCP protocol is to enable the read and write access to the variables and memory contents of the microcontroller system runtime. The ASAM standard states that the primary purpose of the XCP protocol is to adjust the internal parameters by writing commands and acquire the current values of the internal variables by reading commands of an ECU runtime. In addition, the XCP protocol supports both the synchronous and asynchronous serial interfaces.

The XCP Single-Master is a measurement and calibration system, often a PC, and the ECUs operate as Slaves, like shown on Figure 2.36. The Master establishes a communication channel, a point-to-point connection, with the slave. Multiple masters are not allowed but a master can have multiple communications channels active in the same network and at the same time. Every slave in the system has an ECU description file, A2L-file, which specifies which checksum method is being used, the physical meaning of the data, the link between variable names and their address range, and so on. The XCP master can access these description files and read out all the necessary information that might be needed.

The communication between master and slave is done through the XCP driver that is integrated into the master, like in Figure 2.37. In XCP Standard Communication mode, each request packet sent from the master, the slave will respond with a response packet or an error packet. The Master will not send a new packet until it receives a response or error packet from the slave regarding a previously sent request.

To speed up the process of uploading and downloading to the memory and programming the flash, **Block Transfer Communication Mode** can be applied. This communication mode is similar to the one that is used in UDS. In this mode, multiple re-

quests are sent at the same time as a block of data and the slave responds by sending a block of responses back to the master.

Another way to enhance transfer speed is to use the **Interleaved Communication Mode** where the master doesn't wait for a response packet until it sends a new request. The number of requests that can be sent interleaved by the master depends on the queue size of the slave. Block Transfer Communication Mode and Interleaved Communication Mode cannot be used in the same system.

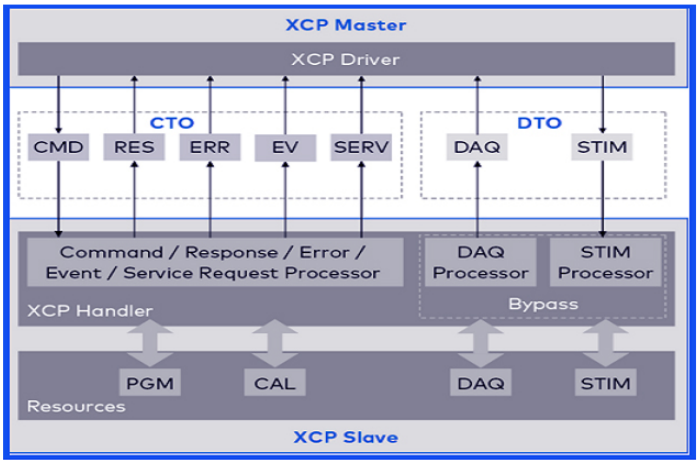


Figure 2.37: XCP Data Communication - Drivers, Handlers, Bypass and Resources [34]

Application of the XCP Protocol:

- It's used in ECU Development.
- It's used in Systems for function and environment tests of ECUs.
- It's used in a test bench for combustion engines, gearbox, climate control, Instrument Cluster Calibration, or wheel alignment, etc.
- It's used for measurements and calibration in pre-production vehicles.
- It's also used in general CAN applications outside the vehicle industry.

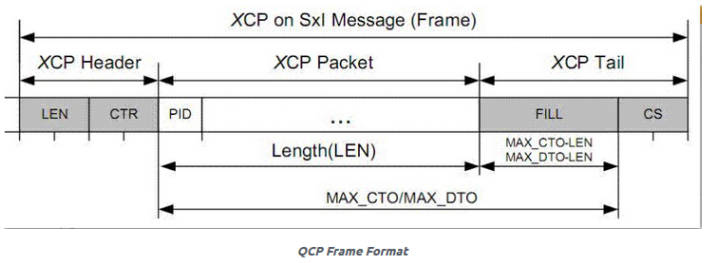


Figure 2.38: XCP Protocol Frame Format [34]

An XCP message consists of three parts, Header, Packet and Tail. The Header and the Tail are part of the transport layer and consist of a Control Field which content depends on what bus is used. The Packet part of an XCP message consists of an Identification Field, Timestamp field, and a Data Field and its considered a generic part of the protocol layer. Figure 2.38 shows us an example of how a XCP message is usually presented.

Two different type of packets are used in the communication via XCP, **Command Transfer Object (CTO)**, and **Data Transfer Object (DTO)**. The CTO has five types of objects:

- CMD (Command)
- RES (Response)
- ERR (Error)
- EV (Event)
- SERV (Service Request Processor)

The DTO has two types of objects that are used for event-driven reading of variables from, or writing values to, also known as the slaves' memory.

- DAQ (Data Acquisition)
- STIM (Stimulation)

Data Acquisition (DAQ) A core feature of the XCP protocol. XCP offers the ability to configure lists that take care of the transmitting requested data at a given interval. Each DAQ list (Figure 2.39) has a number of **Object Descriptor Tables(ODTs)** that in turn contain ODT Entries as described in Figure 2.40. Each of the ODT Entry has an address and a length, these make out the description of the parameter that it represents. When the DAQ list has processed the contents of the list are copied to the corresponding address of each entry in each ODT. The slave doesn't receive an acknowledgment that the master has received the data correctly.

For each DAQ-list configuration, a number of ODT's are defined, each having a unique identifying PID.

Each ODT entry in a DAQ list points to a memory element with specified address and length.

DAQ-list Configuration The XCP has two different ways of configuring the DAQ-lists, static and dynamic. While static configuration is mandatory according to the specification, dynamic configuration seems to be the preferred way. Which configuration method that is to be used is decided exclusively by the slave, there is no way for the master to request one or the other, nor can both be used in parallel. In addition to the configurable DAQ-list (static or dynamic), the slave can also have a number of predefined address address and size. The only thing the master is allowed to do is configure their direction, Prescaler, Priority, and which event channel it should be connected to.

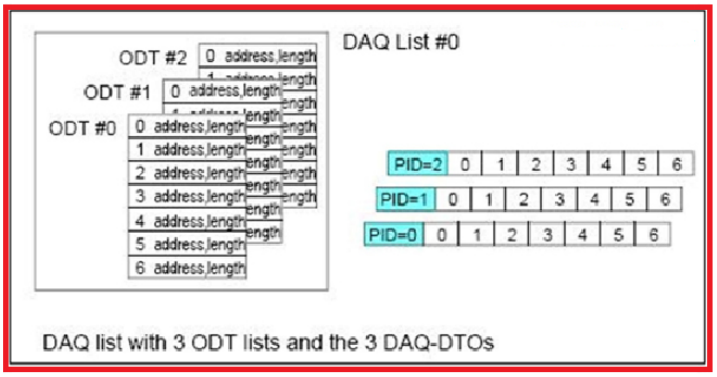


Figure 2.39: DAQ-List Configurations [34]

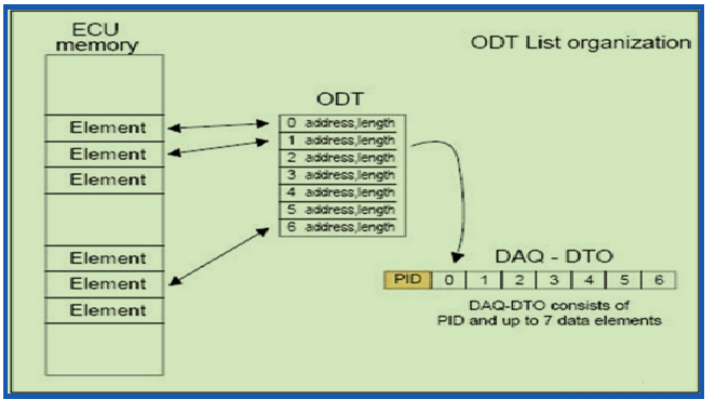


Figure 2.40: ODT Entry List [34]

In the **static configuration**, the slave already has a structure of DAQ-lists with ODTs, that then have entries. This configuration cannot be edited. If there is only one DAQ-list, that has three ODTs and the ODTs have 5 entries each, then this is all the master has at its disposal. The entries can be edited. A lot of the DAQ-list properties can also be changed just as in the case with predefined list.

The **dynamic configuration** is, as the name suggests, less restricted. The master can request allocation of any number of the DAQ-lists, each DAQ-list can have any number of ODTs and the ODTs can have any number of entries. There are restrictions to the command sequence of the allocation, which can be seen in Figure 2.41 and the following list:

- The allocation must always start with sending a command to clear the previous allocations; this is done with the FREE_DAQ. This will reduce the number of DAQ-lists to the predefined lists if any exist on the device.
- The next step is to allocate the DAQ-lists; this can either be done one at a time or by allocating all the lists at once. If a FREE_DAQ has not been executed

before this step the device will return an error message.

- After allocating the DAQ-lists the master can start allocating ODTs to the different lists. All the ODTs in all the DAQ-lists need to be allocated before the first entry is allocated.
- Finally, the entries are allocated as in Figure 2.41.

The parameter MAX:ODT is defined in the AUTOSAR specification as a maximum number of ODTs available on the slave, its range, however, suggests that it is instead the maximum number of ODTs available in the DAQ-list.

after	FREE_DAQ	ALLOC_DAQ	ALLOC_ODT	ALLOC_ODT_ENTRY
FREE_DAQ	✓	✓	ERR	ERR
ALLOC_DAQ	✓	✓	✓	ERR
ALLOC_ODT	✓	ERR	✓	✓
ALLOC_ODT_ENTRY	✓	ERR	ERR	✓

XCP Dynamic DAQ Diagram

Figure 2.41: XCP Dynamic DAQ Diagram [34]

STIM Lists - Data stimulation lists The opposite of DAQ-lists are STIM-lists. They provide a means for the master to write (stimulate) to the slave in a controlled manner. When the master writes to a STIM list, the data is buffered in the Slave until the STIM list is executed at which point the simulation data is copied to specified memory addresses of the ECU. It allows the ECU to apply new parameters at controlled points in time. The STIM-lists are built up in the same fashion as DAQ-lists. They consist of ODTs and ODTs entries. Each ODT is transmitted in a single STIM packet from the master to the slave and consists of multiple ODT elements.

Processing Event Channel Each DAQ-List is connected to an event channel that dictates how often the DAQ-list should be executed.

XCP Signal Bypassing Bypassing is a feature of the XCP that allows replacing some part of an ECUs control logic with code that resides on the XCP master.

XCP - Security Access:

To ensure that access to the ECU is restricted to authorized persons only a security mechanism called **Seed & Key** is a part of the XCP Specification. A slave unique key is needed to grant access to a slave and the master must ask the slave for a seed to be able to compute the key and gain access. The algorithm to calculate the key is unknown to the master to ensure confidentiality. The key algorithm is encapsulated in a file called **SeedNKey.DLL** which the master uses to calculate the key with the help of the seed provided by the slave.

XCP-ECU Flash Programming:

In the XCP object called **SECTORS** describes the physical layout of the ECU memory. Start Addresses and sizes of the SECTOR are of high importance when reprogramming the ECU.

Programming of flash memory in a slave device can be divided into three parts:

Administration before programming: To check what version of the software is currently loaded in the memory, for example.

The flash process: This is when the programming of the memory takes place.

Administration after programming: To do the checksum control to see if the flashing process was successful, for example.

The XCP Standard does not support special commands for version control because the administration steps are very project-specific and depends on the ECU, but the ECU functional description can verify which standard XCP commands can be used for the administration actions like version control.

XCP - Flash Memory Access:

There are two different processes to the flash access methods supported by the XCP protocol to be used in the flash process. **Absolute Access Mode** and **Functional Access Mode** which both uses to the same commands but with some different parameters.

- Absolute Access Memory
- Access by Address

This is the default mode and is used when the physical layout of the flash memory is well known to the programming tool. The flash content to be programmed must be available and the address information of the data must be known. The physical layout information can be read out of the ECU or in the description file depending on how it is done in the project. The block of data is contained in the CTO will be programmed into the flash memory starting at the Memory Transfer Address (MTA) which will be incremented by the number of bytes of data that are being programmed into memory, like shown in Figure 2.42.

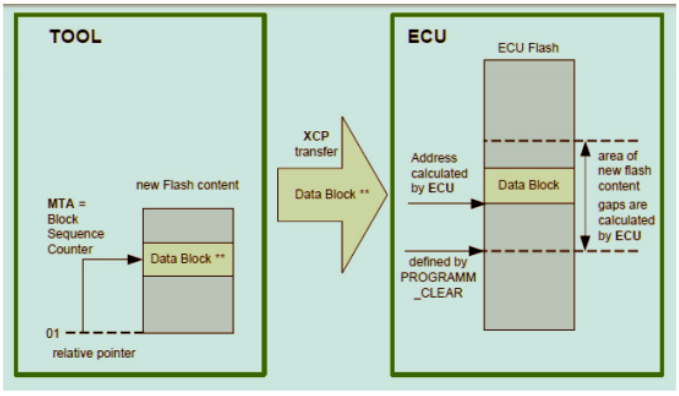


Figure 2.42: XCP - Functional Data Access Communication [34]

The XCP communication protocol is also one of the most important functions used for testing this project's sensor. Without the use of this protocol, most of the tests performed would not have much real-time information from the sensor. More on this topic will be discussed in **Chapter 3 - CAPL Testing**.

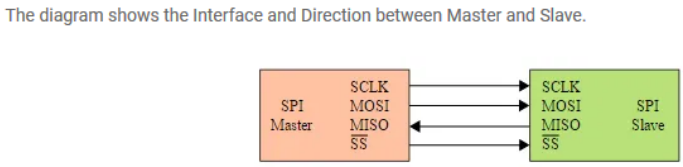


Figure 2.43: Interface and Direction Between Master and Slave [35]

2.7.2.4 SPI Communication

SPI stands for **Serial Peripheral Interface** and works on a Master-Slave Concept [35]. It is synchronous serial communication used for short distance communication.

It is a full-duplex communication protocol as it has two separate lines for transmission (**MOSI**) and reception (**MISO**). The SPI communication is possible with a single master and a single slave as well as a single master and multiple slaves.

It is also known as a 4 wire protocol due to its 4 wire interface , like shown in Figure 2.43.

SPI Communication follows the following steps:

- Master selects Slave device through Slave Select Line - SS.
- Master's device also configures the SPI peripheral Clock supported by the Slave - SCLK.
- Data availability on Data Out (MOSI) and Data In (MISO) line is controlled by the master generated Clock Pulse (SCLK).
- Master generated **Clock Pulse Polarity (CPOL) and Phase (CPHA)** decides the data drive on the data line.

  CPOL: Decides the polarity of the Clock

  CPHA: Determines the timing of the data bits relative to the clock pulse.

On Figure 2.44 there is an example of the SPI Communication concept.

This protocol is also used during SW testing. More specific information about how it is used and for what purposes can be found in **Chapter 3 - CAPL Testing**.

2.7.3 Vector CANoe

CANoe, short for **CAN Open Environment**, is a development and testing SW tool from Vector Informatik GmbH [36]. The SW is primarily used by automotive manufacturers and ECU suppliers for development, analysis, simulation, testing, diagnostics and start-IP of ECU networks and individual ECUs [37]. Its widespread use and large number of supported vehicle bus systems makes it especially well suited for ECU development in conventional vehicles, as well as hybrid vehicles and electric vehicles.

CANoe supports **CAN** [38], **LIN** [39], **FlexRay** [40], **Ethernet** [41] and **MOST** [42] bus systems as well as CAN-based protocols such as **J1939**, **CANopen**, **ARINC 825**, **ISOBUS** and many more.

CPOL	CPHA	SCLK initial State	Data Change On Data Line	Data stable on Data Line At
0	0	Idle state is '0'	SCLK Falling edge	SCLK Leading Edge
0	1	Idle state is '1'	SCLK Leading edge	SCLK Falling Edge
1	0	Idle state is '0'	SCLK Falling edge	SCLK Leading Edge
1	1	Idle state is '1'	SCLK Leading edge	SCLK Falling Edge

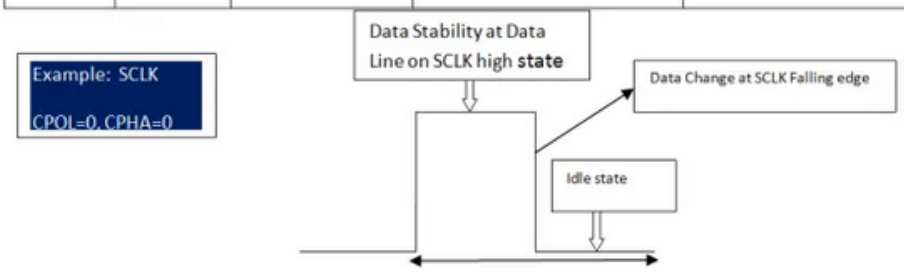


Figure 2.44: SPI Communication Concept [35]

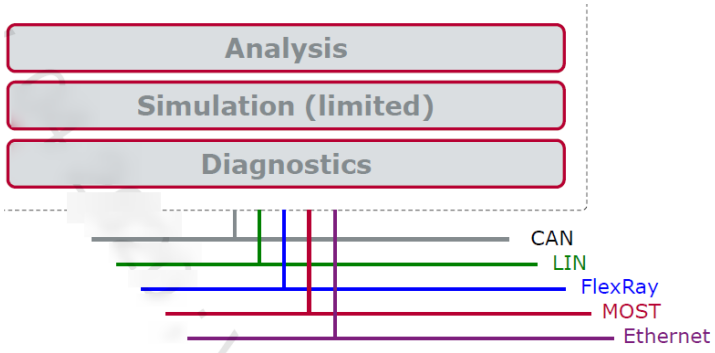


Figure 2.45: The CANalyzer Bus Connections [31]

CANoe Data is displayed and evaluated in either raw or symbolic format. Back in 1992, Vector developed the DBC data format, which has become a de facto standard for exchanging CAN descriptions in the automotive field.

While CANoe can simulate the whole communication in a vehicle, it also includes a Test Feature Set, for creating fully controlled (automated) test sequences. These automated test sequences can be fully controlled by the usual **Continuous Integration** Tools **[CI]**, such as Jenkins. The Test Feature Set included in CANoe has a long history and is therefore available in variants. Test Cases can be created in CAPL, - a C-like programming language - in XML or in C#. The tests can either be manually programmed or generated automatically by different generators.

2.7.3.1 CANoe/CANalyzer Fundamentals

CANalyzer is a powerful tool to analyze networked systems, shown in Figure 2.45.

Besides the named bus systems CAN, LIN, FlexRay, etc, CANalyzer also supports the following protocols:

- CANopen (Automation)
- J1939 (Utility Vehicles)
- J1587
- ISO 11783 (Agricultural Utility Vehicles)
- NMEA 2000 (Nautical Vessesls)
- IP

CANoe, is a powerful tool, which supports the entire development process of networked systems from planning to putting into service. Like CANalyzer, CANoe also has a similar bus connection, like seen in Figure 2.46.

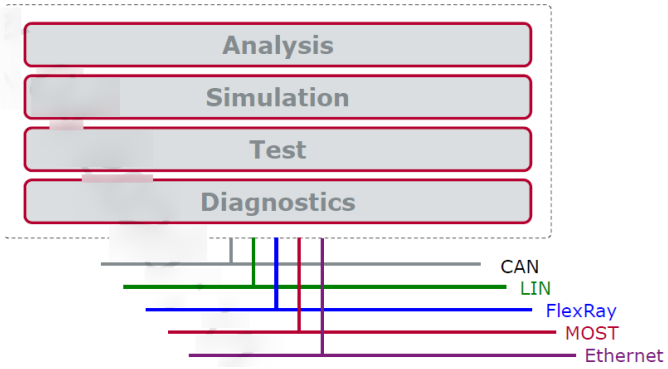


Figure 2.46: The CANoe Bus Connections [31]

Besides the named bus systems CAN, LIN, FlexRay, etc, CANoe also supports the same protocols as CANalyzer with only small difference regarding.

CANoe can be used as a Simulation Tool, and its capable of simulating individual nodes or complete networks (Figure 2.47). Simulations can be implemented in different bus systems simultaneously. Various options are available for creating such models:

- "Manual" Programming of Simulation Nodes
- Automatic generation of the simulation nodes
- Integration of an Interaction Layer

When using CANoe as a Test Tool (Figure 2.48), the test specification can be programmed and described in a CAPL file. An XML file is also used in this structure to help access individual test functions and also show the test's structure on the CANoe interface. A test report is created at the end of the test, which in turn can be further processed or converted.

If the ECU's HW Input and Outputs are to be tested in the procedure, and additional HW Interface is also necessary.

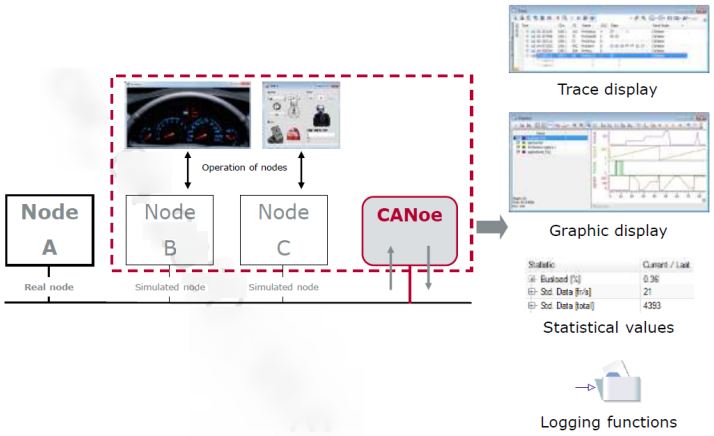


Figure 2.47: CANoe as a Simulation Tool [31]

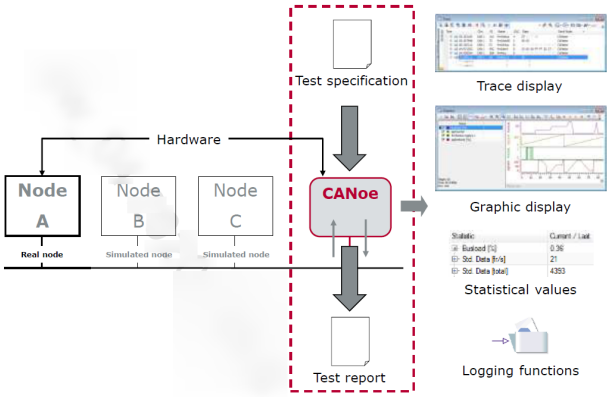


Figure 2.48: CANoe as a Test Tool [31]

The CANoe System uses USB Devices like CANcaseXL or VN8900, shown in Figure 2.49. They work with so called CANpiggies as transceivers and are built-in and some interfaces provide an exchange mechanism. Beneath CAN interfaces Vector also provides interfaces for other bus systems like LIN, FlexRay or Ethernet. For the acquisition of digital and analog data CANoe supports I/O HW. This can be from Vector or from other manufacturers like e.g. National Instruments or Keithley.

Vector CANoe is a tool that has been certified and included in the SW Testing Process tools list, by and for the Company. For more information regarding how it is utilized and how the tests are perform refer to **Chapter 3 - CANoe Testing** and **CAPL Testing**.

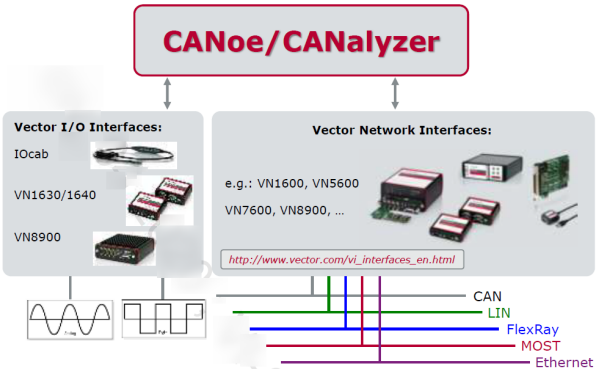


Figure 2.49: CANoe/CANalyzer various interfaces [31]

2.7.4 SPI Simulyzer

The **SPI-Simulyzer** [43] is an interface box with which data can be read out quickly and easily, and simulation data can be used to actively intervene in the testing process.

Sensor modules are part of the sensors, like the one being worked on in this project, that communicate with the microcontroller via SPI communication.

It is a mechanism that simulates SPI signals in a way that emulates a sensor module. It also verifies that the signals are being processed by the SW in the microcontroller.

The SPI Simulyzer is important for the Team since it enables part of the SW tests to be performed and simulate the actual behavior needed by the SMUs. More information about how this is used can be found on **Chapter 3 - Testing Problems**, in the section related with **Fault Injection related Test**. The SPI-Simulyzer is connected via USB cable with a Windows-based software. There are two types of Simulyzer-Versions:

Single ended Has two Interface connections and a 25-pol. SUB-D female connector;

LVDS version Differential signal transmission adaption.

There also different working modes (Figure 2.50):

ECU mode The SPI Simulyzer supplies the sensor with voltage and generates defined master data. Parallel the data which are sent will be recorded. The data communication between the simulated ECU (SPI master) and the up to 4 sensors (SPI Slaves) are displayed and recorded.

Sensor mode The SPI Simulyzer simulates up to 4 sensors by generating according data. The data communication between the ECU (SPI master) and simulated sensors (SPI slaves) are displayed and recorded.

Passive mode The data communication between the sensors and the control unit (ECU) will be recorded and visualized.

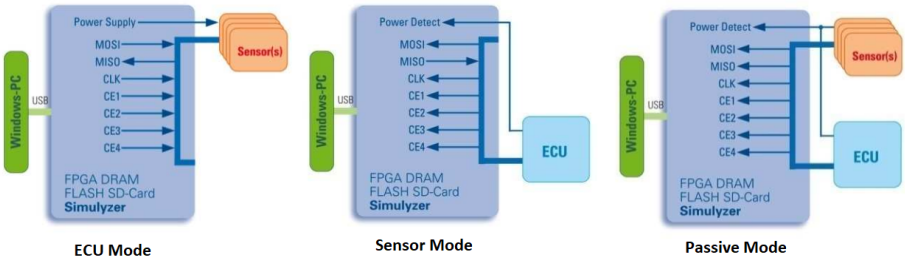


Figure 2.50: SPI Simulyzer Working Modes [43]

2.7.5 Renesas Flash Programmer

The **Renesas Flash Programmer [RFP]** [44] is a software that uses an E1 emulator, E20 emulator, E2 emulator or E2 emulator lite via a serial or USB interface to erase, write and verify programs on a target system on which a Renesas Electronics MCU with on-chip flash memory is mounted. The RFP is great for small production batches, for lab use and for testing. This has the advantage of using the same development tools while keeping the source code under wraps.

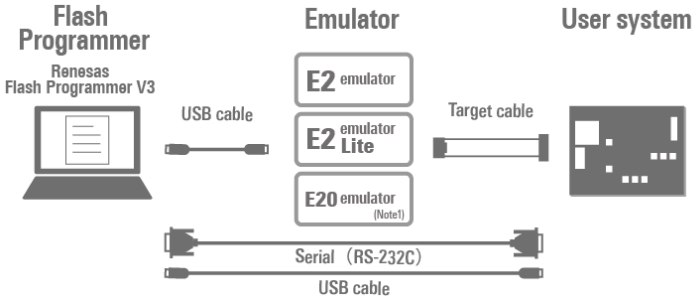


Figure 2.51: Renesas Flash Programer System [44]

In the Figure 2.51, its possible to see how the RFP connects to the computer and with the sensor to flash it, with each type of connection possible.

Due to the simpleness of this programmer, it is one of the few programs approved and included in the SW Testing Process tools list, and in turn used by the Team.

It is mainly used to flash the sensor, only needing a application SW file. For more information related with the Renesas Flash Programmer can be found in **Chapter 3 - Manual Building and Flashing**.

2.8 Continuous Integration

2.8.1 What is Continuous Integration?

Continuous Integration [**CI**] is a development practice where developers integrate code into a shared repository frequently. Each integration can then be verified by an automate build and automated tests. The key benefits of integrating regularly is that

you can detect errors quickly and locate them more easily. As each change introduced is typically small, pinpointing the specific change that introduced a defect can be done quickly [45].

In recent years CI has become a best practice for SW development and is guided by a set of key principles. Among them are revision control, build automation and automated testing [46].

One can't talk about CI without also discussing **Continous Deployment** [**CD**] and **Continuous Delivery** [**CDel**]. While CI is the practice of integrating changes from different developers in the team into a mainline as early as possible. This makes sure the code individual developers work on doesn't divert too much. When you combine the process with automated testing, continuous integration can enable your code to be dependable. On the other hand, CD is closely related to CI. It refers to keeping your application deployable at any point or even automatically releasing to a test or production environment if the latest version passes all automated tests. CDel is the practice of keeping your code base deployable at any point. Beyond making sure your application passes automated tests it has to have all the configuration necessary to push it into production.

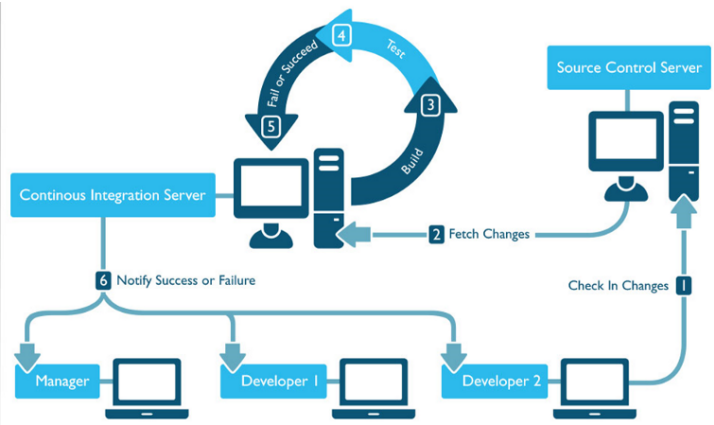


Figure 2.52: Continuous Integration Cycle [47]

CI has many benefits, like already referred. A good CI setup can speed up the workflow of the Team and encourage them to push every change without being afraid of breaking anything, like in Figure 2.52. It helps reduce the risk level of the project, as it is possible to detect bugs and code defects earlier. This makes the errors easier to fix and the sooner they are fixed, the cheaper it is. When you have a CI process working, it is easier to share the code regularly. This code sharing helps to achieve more visibility and collaboration between team members, and in time increases communication speed and efficiency within your organization as everybody is on the same page. With the CI process, not only does it benefit the SW developers and testers but also their managers. Both parties can gather valuable feedback and gain insights much faster. More data is available which can be analyzed to check if the product is heading into the right direction. It can also help reflect on the progress of the project more frequently which enables faster technological and business decisions.

This is a very important topic for the work related with this dissertation as it would greatly value the work done by the Team and demonstrate the workflow in the meantime. More information about this topic, and how it is employed can be found on **Chapter 4 - Automated Tests with Jenkins**.

2.8.2 Jenkins

Jenkins [48] is known as the leading open source automation server, and provides hundreds of plugins to support building, deploying and automating any project. It is a self-contained, open source automation server which can be used to automate all sorts of tasks related to building, testing and delivering or deploying software, and can be installed through native system packages, or even run as standalone by any machine with a Java Runtime Environment installed.

Jenkins offers a simple way to set up a CI or CD environment [49] for almost any combination of languages and source code repositories using pipelines, as well as automating other routine development tasks. While Jenkins doesn't eliminate the need to create scripts for individual steps, it does give a faster and more robust way to integrate your entire chain of build, test, and deployment tools that you can easily build yourself, like shown in Figure 2.53.

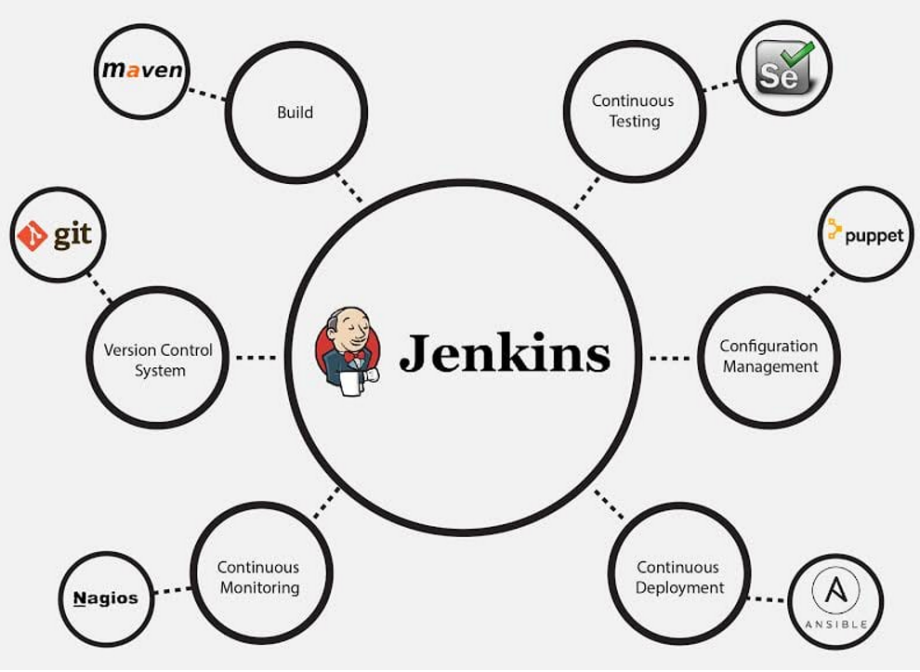


Figure 2.53: Jenkins CI and CD [48]

"Don't break the nightly build!" is a cardinal rule in software development teams that post a freshly daily product version every morning for their testers. Before Jenkins, the best a developer could do to avoid breaking the nightly build was to build and test carefully and successfully on a local machine before committing the code. But that

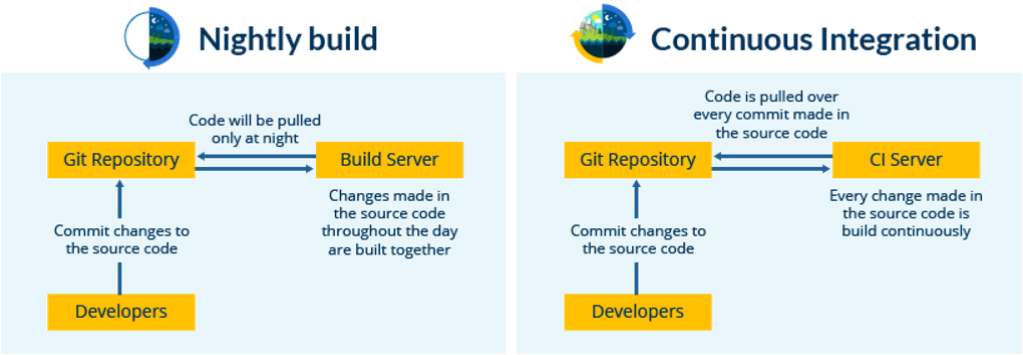


Figure 2.54: Jenkins - Nightly Build and CI Differences [48]

meant testing one's changes in isolation, without everyone else's daily commits. There was no firm guarantee that the nightly build would survive one's commit.

CI is a development practice in which the developers are required to commit changes to the source code in a shared repository several times a day or more frequently. Every commit made in the repository is then built. This allows the teams to detect problems early. Apart from this, depending on the CI tool, there are several other functions like deploying the build application on the test server, providing the concerned teams with the build and test results (Figure 2.54).

Like referred before on the CI section, more information regarding Jenkins and how it was executed, can be found in **Chapter 4 - Automated Tests with Jenkins**.

Chapter 3

Definition of the Testing Problems

In this Chapter, an overview of the Project's Testing problems is detailed.

In order to maintain the Quality of the SW and of the tests, the Company's Process and Methods has to be followed as closely as possible. Maintaining the process and methods is a challenge in its own way, from how tests are made to the problems presented to a daily routine. For the purpose of this work, only the Qualification Tests level of the V-Model will be discussed. CANoe testing is explained here, from the common knowledge of what a Requirement Based Test is, to how the tests are performed. The Vector CANoe tool is also explained and shown in the testing method used by the Team. Right after, CAPL Testing is explained and the current testing problems are viewed in detail and how they the Team. In the end, a overview of what problems exist with Manual Building and Flashing of the SW is seen, and how it can be resolved.

3.1 Testing Process

In a project of this size all processes need to be followed to maintain coherence and traceability.

The SW engineering process for the IMU project is defined in an Internal Process Library. Figure 3.1 was taken from the referred process libary. From it, it is possible to understand the workflow in the testing process. This process starts from Developing a SW Test Strategy, and then create and refine the Sw Test Schedule. After this is done by the SW Test Coordinator, the SW Test Engineers can start the development of the SW test case specification and scripting. Only after the proper Review process has been completed, can the Test be executed and the Results saved and processed to be used on the Overall Test Report for the SW Release Notes. This process also applies to all test levels, including the one being discussed here.

3.1.1 CANoe Testing

CANoe is a powerful tool used not only for testing but also for debugging or even simulating the ECU's behavior, as mentioned before. The Team uses CANoe for the purpose of doing the Qualification Tests, also known as **Requirements Based Tests [RBTs]**.

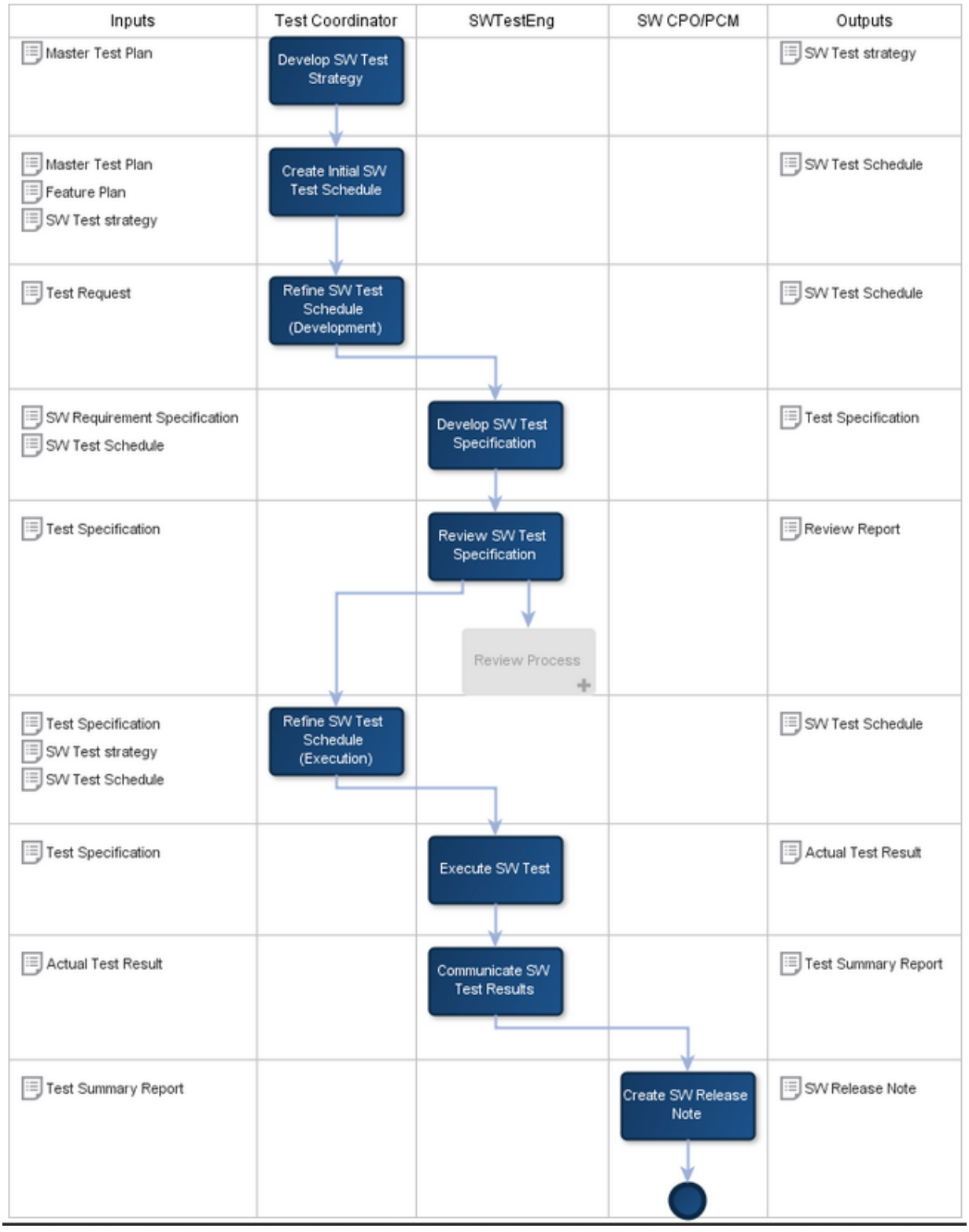


Figure 3.1: IMU's VerVal Team Process Chart

Tests are performed as **Hardware-on-the-loop [HIL]**: An ECU is flashed with the SW under test and the ECU is connected to a host PC or test bench. The Communication Interface with the sensor can only be accessible to testers or developer and not the final customer. For communication purposes one has access to the UDS protocol, with access privileges, and also has to XCP communication via CAN, mainly used for debugging the code.

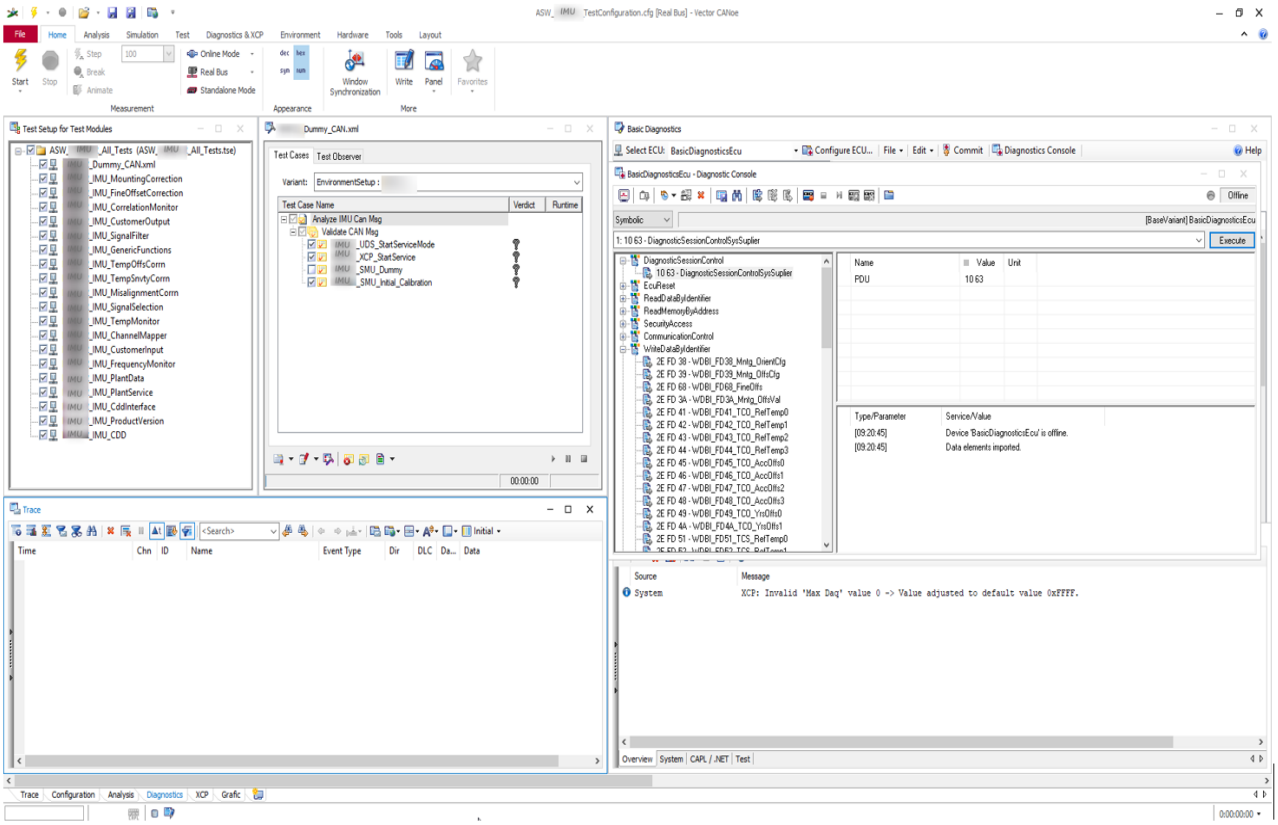


Figure 3.2: CANoe interface and configuration file

To start using CANoe, the corresponding Variant Configuration file for testing should be available. A Variant is the name given to a SW specific to one or more clients, and so a different configuration file is needed for each type of variant under test. This enables different settings to be given on each configuration file, enabling different tests to be performed. For the purpose of this dissertation, the Variant used will be called **Variant A**.

Figure 3.2 shows how the tool looks like for Variant A. On the Test Setup panel, the Components to be tested can be found on. The middle panel has the test scripts of a specific component. From here the tester executes and waits for the test results. The results will show green or red, depending if the test passed or failed, respectively. A test report is also generated as a **.html** report file.

The CANoe tools has an interesting feature where it is possible to store and save various different parameters related to the tests, i.e., configuration of the CAN commu-

nication (CAN transceiver port), UDS configuration (diagnostic messages), CAN XCP configuration, and others. This last case can be seen as a configurable desktop, where the testers can associate a Trace Window, a Graphic Window, and CAN Communication Bus Load Window. The UDS configuration and the CAN XCP configuration can be used by the tester as a support tool while running tests.

3.2 CAPL Testing

CANoe has its own testing language, known as CAPL, like described on **Chapter 2** regarding Vector CANoe. CAPL is similar to C and C++, but has an feature test set that permits the generation of a report in accordance with the verification that the tester wishes to perform. Normally a CAPL test should have an *include section*, where the needed libraries are called, and a *variables section*, where global variables and/or structures are declared. What comes next depends on the kind of test being done. The tester normally creates tests based on the **Test Case Specification** that is included on the Component's **Test Suite** details. These specifications can be found in the ALM tool and are based on the Requirements found on **DOORs Next Generation [DNG]**.

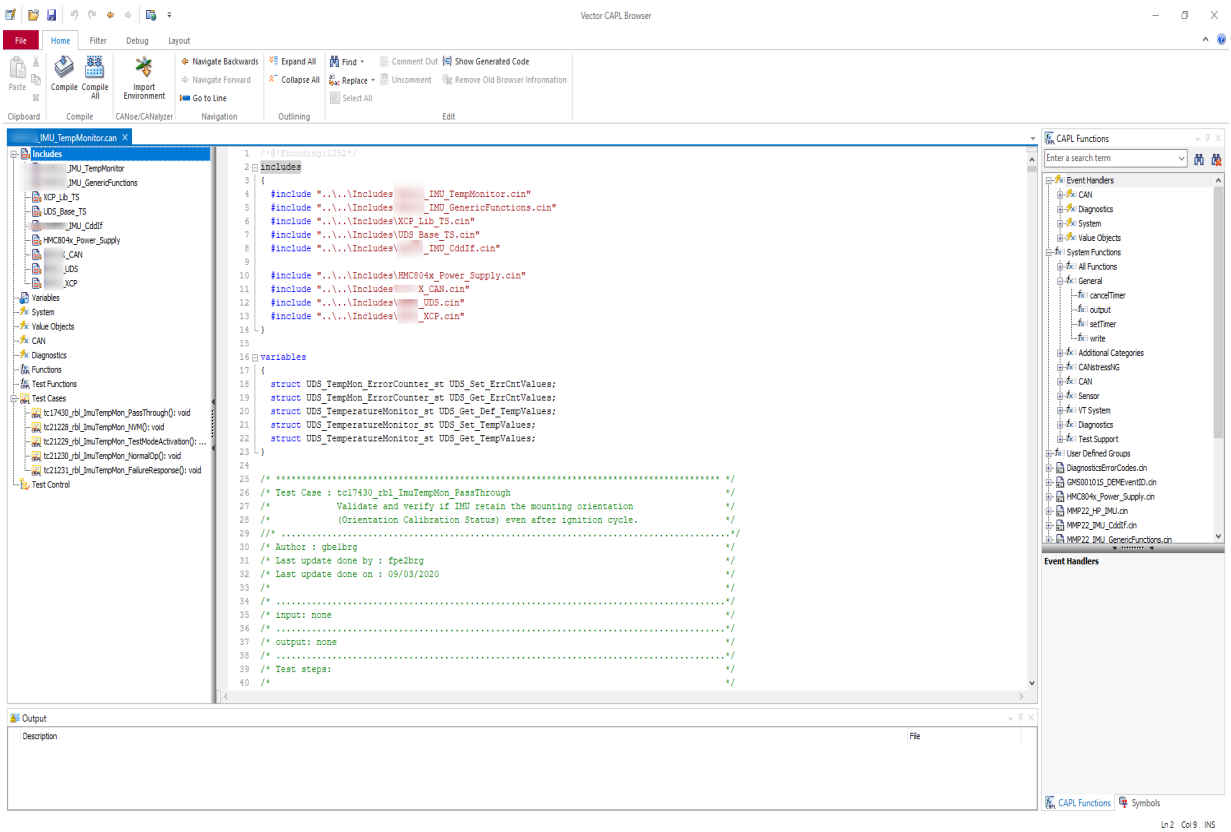


Figure 3.3: CAPL Test Example

Figure 3.3, is an example of how a CAPL test usually looks like. On the right of the browser is an index tree that helps the user to easily move around the test. On the left

there's a library panel where all CAPL available functions can be accessed. In the center there is a CAPL Editor to edit/write the test scripts.

In CAPL testing, the test files will always have the extension *".can"*, while library files will have the extension *".cin"*. Most of the libraries used by the IMUs VerVal team were self made, in accordance with the team's needs for testing, based on CANoe's basic CAPL Functions Library.

Considering a Component's Test Suite, the CAPL test code will be divided into various functions called *testcase TC_ID_Function_Name ()*. Each *testcase* function has its own header comment where you can find the Test Case ID and function name, followed by a small description of what the function is used for. It then has the ID of the original author and also when the last update was done and by whom.

Depending on the kind of testing needed for each test case specification, there is a set of functions created for the purpose of simplifying each task. From the Library files, named after the component under test or specific group of functions, the tests has access to a vast kit of functions that uses either the UDS or the XCP protocols to **Set** the default values to the sensor's memory or to **Get** (Read) the values in the sensor's memory. Both the usage of this UDS and XCP functions read/write to the sensor's RAM via CAN with real-time refresh rates.

Which kind of protocol to use depends on the kind of test being done. Most tests are divided in to the following category list:

- Test Switch related Tests
- Fault Injection related Tests
  - NVM Memory Corruption Tests
  - SPI Simulizer related Tests
- Normal Operation related Tests

Before starting with each type of test that needs to be implemented, firstly there is a need to understand how the tests are performed depending on the protocol used.

3.2.1 UDS-based Testing

In **Chapter 2 - Tools and Protocols**, the UDS protocol is presented as the industry uses it. In terms of testing with this protocol, it is presented in a different way. The UDS protocol plays a major role in the testing process of the IMU sensor, as it allows to read and write SW configuration parameters directly from the NVM memory. And in turn provides the possibility of validating different types of behaviors from the component. This protocol is present in practically all the tests performed.

Each component test module has the UDS library added to its include files. In this library are located the functions used for testing. In this example, it will be discussed how the Temperature Monitor component is tested.

In Figure 3.4, functions that use the UDS Protocol to **Set** the parameters of the variables and then to **Get** the variables parameters for verification purposes are shown. *UDS_Set_TempMonActivation* is a function from this component's library that is used to

```
testcase tc37978_rbl_ImuTempMon_NVM(void)
{

  testReportAddExternalRef("url"," Test Case 37978","
  TestCaseTitle("Test Case 37978 ", " Validate NVM ");

  /*************************** Step 1 ************************

  /* 1.1- Set component to passive: Main switch = 0 */
  UDS_Set_TempMonActivation("1.1.1", 0);
  /* Reboot sensor */
  if( RebootPowerSupply(1)== 1)
    testStepPass("1.1.1","Successful Ignition Cycle");
  else
    testStepFail("1.1.1", "Unsuccessful Ignition Cycle");

  /*wait some time to make sure the default configuration of
  testWaitForTimeout(500);

  /* Get component status: Main switch = 0 */
  UDS_Get_TempMonActivation("1.1.2", 0, 0);

  /* 1.2 - Set warning and limit error counter values */
  UDS_Set_ErrCntValues.Warning_Increment_u16 = 5;
  UDS_Set_ErrCntValues.Warning_Decrement_u16 = 5;
  UDS_Set_ErrCntValues.Warning_Limit_u16 = 3000;
  UDS_Set_ErrCntValues.Warning_Hold_u16 = 60000;
  UDS_Set_ErrCntValues.Error_Increment_u16 = 2;
  UDS_Set_ErrCntValues.Error_Decrement_u16 = 2;
  UDS_Set_ErrCntValues.Error_Limit_u16 = 4000;
  UDS_Set_ErrCntValues.Error_Hold_u16 = 25000;

  UDS_Set_TempMonErrCntCfg("1.2.1", UDS_Set_ErrCntValues);
  /* Reboot sensor */
  if( RebootPowerSupply(1)== 1)
    testStepPass("1.2.1","Successful Ignition Cycle");
  else
    testStepFail("1.2.1", "Unsuccessful Ignition Cycle");

```

Figure 3.4: UDS Set and Get Examples

set the Activation status of the Temperature Monitor Main Switch, where "0" signifies a passive state and "1" is active state.

What this function does is shown in the code presented on Figure 3.5. The test runs a **Diagnostic Request** for the *Write Data By Identifier* associated with the requested variable, in this case *TempMon_Activation*. The UDS service is started and proceeds with loading the data to be written and sends the write request. It then waits for a bit and starts checking if the variable data was written correctly before going back to the test.

UDS_Get_TempMonActivation is a function mostly used right after a *UDS_Set_VariableNAme* function with the purpose of verifying if the variable was correctly saved and in the correct memory space.

In the code example seen in Figure 3.6, the Get function is similar to the Set function. It starts with a DiagRequest for the *Read Data By Identifier* of the same variable, compares the value read and validates if its actually the wanted value or not before continuing the

```
394     /* Diagnostic service to use */
395     diagRequest BasicDiagnosticsEcu.WDBI_FD2A_TempMon_Activation WriteData;
396     long returnValue;
397
398     /* If the Diagnostic Session No Enable, Start Service */
399     UDS_StartServiceMode_IfServiceNotEnable();
400
401     /* Load data to write */
402     returnValue = diagSetParameter(WriteData, "TempMon_Active", MainSwichStatus);
403     DiagErrCode(TestCaseID, returnValue);
404
405     /* Send Request */
406     returnValue = diagSendRequest(WriteData);
407     DiagErrCode(TestCaseID, returnValue);
408
409     /* Waits until the previously sent request has been sent to the ECU.  */
410     if(TestWaitForDiagRequestSent(WriteData, 100) == 1){
411       testStepPass(TestCaseID, "Request sent successfully!");
412     }
413     else
414       TestStepFail(TestCaseID, "Request could not be sent!");
415
416
417     /* Test wait for response */
418     returnValue = testWaitForDiagResponse(WriteData,100);
419     if(returnValue == 1)
420     testStepPass(TestCaseID, "Response received.");
421     else if(returnValue == 0)
422     testStepFail(TestCaseID, "The timeout was reached.");
423     else
424     testStepFail(TestCaseID, "Internal error cocurred.");
425
426
427     /* Check the response */
428     returnValue = diagGetLastResponseCode(WriteData);
429     if(returnValue ==-1)
430     {
431         testStepPass(TestCaseID, " Temperature Monitor status configuration parameter was written with Success over UDS.");
```

Figure 3.5: UDS Set Function Code

```
741 byte UDS_Get_TempMonActivation(char Test_ID[], int MainSwichStatus, byte verifyNRC){
742
743   #if 0 /*--------------------------- using ReadDataByIdentifier command ---------------------------------*/
744
745     /* Diagnostic service to use */
746     diagRequest BasicDiagnosticsEcu.RDBI_FD2A_TempMon_Activation ReadData;
747     long returnValue;
748     long Read_status;
749
750     /* If the Diagnostic Session No Enable, Start Service */
751     UDS_StartServiceMode_IfServiceNotEnable();
752
753     diagSendRequest(ReadData);
754
755     testWaitForDiagResponse(ReadData,100);
756
757     if(diagGetLastResponseCode(ReadData)==-1 && !verifyNRC)
758     {
759       Read_status = diagGetRespParameter(ReadData, "TempMon_Active");
760       testReportWriteDiagResponse(ReadData);
761       if (MainSwichStatus == Read_status)
762         testStepPass(Test_ID, "Temperature Monitor status configuration parameter is set correctly!");
763       else
764         testStepFail(Test_ID, "Temperature Monitor status configuration parameter was not set correctly!");
765       return 1;
766     }
767
768     else{
769         if((diagGetLastResponseCode(ReadData)>0) && verifyNRC)
770           testStepPass(Test_ID, " Temperature Monitor configuration parameter return a NRC as EXPECTED");
771         else
772           testStepFail(Test_ID, " Temperature Monitor status configuration parameter read by UDS without success!");
773
774         return 0xFF;
775     }
```

Figure 3.6: UDS Get Function Code

```
/* Load data to write */
diagSetParameter(WriteData, 1, "Warn_Limit_high",  UDS_TempMon.Warn_Limit_high_s16);
diagSetParameter(WriteData, 1, "Warn_Limit_low",   UDS_TempMon.Warn_Limit_low_s16);
diagSetParameter(WriteData, 1, "Error_Limit_high", UDS_TempMon.Error_Limit_high_s16);
diagSetParameter(WriteData, 1, "Error_Limit_low",  UDS_TempMon.Error_Limit_low_s16);
```

Figure 3.7: UDS Set Code for Struct Type Variable

test.

Also in Figure 3.4, there is a structure called *UDS_Set_ErrCntValues.XXXX*, used for configuring the Temperature Monitor's Error Counter values, that are then used on *UDS_Set_TempMonErrCntgCfg* function.

In this case, the *UDS_Set_* function is similar to the *UDS_Set_TempMon_Activation*, but instead was reworked to set multiple variables in accordance with the struct used in the test (shown in Figure 3.7).

Like already seen in the last few examples, all the base UDS functions can be adapted and altered accordingly with what component is under test. It is usually always the same type of functions used.

With this small overview of how the UDS-based testing is performed, the rest of the testing code examples shown in this dissertation will be more understandable.

3.2.2 XCP-based Testing

Like the UDS protocol, the XCP protocol is also presented in a specific way for industrial use. But this protocol also plays an important role in the IMU sensor testing process. XCP allows the possibility to access in real time the RAM memory. This a very useful tool for testing due to the fact that it enables the read and write of global variables while the SW is working in normal mode, without needed of debugging it.

Like the UDS-based tests, XCP-based tests also have an XCP library file with a set of functions that can be used during testing.

Taking the Temperature Monitor as an example, in a normal test, the tester usually starts with the *XCP_StartService* function that is then followed by the read function called *Get_XCP_VariableInt*.

```
XCP_StartService();

SMI0_Temp = Get_XCP_VariableInt("1.5.1", "rbl_ImuTempMon_InputSmxChannelSet_st::Temp_ast_0_::Value_s16");
SMI0_Temp = (SMI0_Temp*5/1000)+50;
SMI1_Temp = Get_XCP_VariableInt("1.5.2", "rbl_ImuTempMon_InputSmxChannelSet_st::Temp_ast_1_::Value_s16");
SMI1_Temp = (SMI1_Temp*5/1000)+50;
SMI3_Temp = Get_XCP_VariableInt("1.5.3", "rbl_ImuTempMon_InputSmxChannelSet_st::Temp_ast_3_::Value_s16");
SMI3_Temp = (SMI3_Temp*5/1000)+50;
testStep("1.5.4", "SMU1 Temperature Value is %f | SMU3 Temperature Value is %f | SMU0 Temperature Value is %f", SMI1_Temp, SMI3_Temp, SMI0_Temp);
```

Figure 3.8: XCP Get Function

A code example from the Test Case related to Temperature Monitor can be seen in Figure 3.8.

In this case, the test starts the XCP service and then calls the *Get_XCP_VariableInt* to get each SMU current temperature, and then validates if the temperature values are inside the reference values.

```
CheckValidSignals("2.6", IMU_Variant_ID, 0);
CheckExpected_XCP_VariableInt("2.7.1","rbl_ImuTempMon_OutputSmxChannelSet_st::Temp_ast_1_::Status_u8",0x0);
CheckExpected_XCP_VariableInt("2.7.2","rbl_ImuTempMon_OutputSmxChannelSet_st::Temp_ast_3_::Status_u8",0x0);
CheckErrorCounters("2.8", IMU_Variant_ID, 0, 0);
```

Figure 3.9: XCP Signal Checking Functions Examples

Another type of functions is seen in Figure 3.9. *CheckExpected_XCP_VariableInt* is a function from the XCP Library file used to validate if the variable read via XCP is as expected. This type of variable read is performed in real time, and compares the read value with the expected parameter, in this case it should be equal to **"0x0"**.

CheckValidSignals is a function adapted to this specific Test Component Module used to check if the various signal status are valid ("0x00") or not ("0x83").

CheckErrorCounters is also a function adapted and is used to check the Error Counter values for the Temperature Monitor. It first checks the status of the expected variable signals and then validates if the Error Counter variables are being counted as expected.

These last two functions are specific module functions adapted from the XCP library file. This means that they were adapted and used for this specific case scenario. This is a common practice for testing purposes since most components present similar variables to be tested.

3.2.3 Test Mode Switch related Tests

In a real life scenario, when the sensor detects something wrong with the values being read, it would send an error or warning message to the ECU to let it know of a potential failure. That is where tests that revolve around switching modes in the sensor and analyzing if the correct execution after this switch is performed. To activate this mode switch during tests the XCP protocol is used to switch between state of operation. It then waits until the RAM memory has the expected input value. Usually when writing to the local memory of the sensor, it will be put in a invalid mode and stop working like expected until the failure is corrected.

An example of this kind of testing can be seen in the Figure 3.10. In this Figure you can see three different components that are interconnected between them. Component A gets the data from the SMUs and sends the output values of "*X,Y,Z*" to Component B. This component gets the values from A and converts them into output values of "*A,B,C*, and sends them to Component C. Component C then just sends the values via CAN Output. In case the Test Mode Switch has been activated, Component C won't get the real values expected from Component B, instead it will get the Buffer's values of *1,2,3*. Since this mode was activated, the sensor would detect the test mode switch variable activation and place invalid outputs to the bus by setting the signal statuses as invalid.

These tests are simple to reproduce and don't usually cause any problems to the testers when being performed. One of the only possible problems with this kind of tests is usually due to CANoe not being able to interpret the data it gets from the XCP

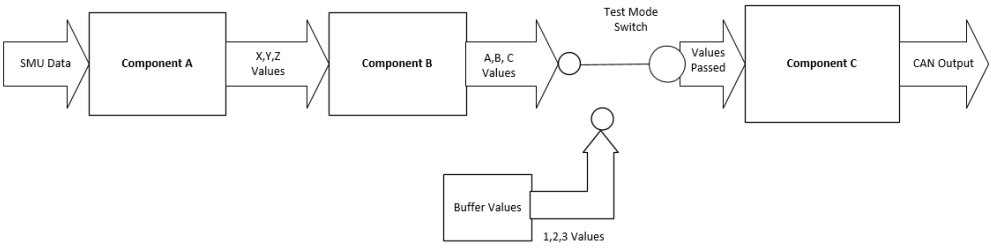


Figure 3.10: Test Mode Switch Example

messages, causing some of the tests to fail due to the value being read isn't the expected one.

This problem can't be corrected, but can be mitigated. The less contact the tester has with the tool during the tests, the better it can communicate with the sensor, and possibly get the expected results more constantly. For this to happen the tests should be automated and sent to run automatically on a Jenkins server, without the need of tester configuring and starting the tests each time something goes wrong. This is the proposed solution discussed and resolved in this dissertation.

3.2.4 Fault Injection related Tests

In this kind of tests, the tester injects errors to check if the sensor has the expected behavior according to the specified requirements. These tests can be divided between NVM Memory Corruption Tests and SPI Simulyzer Tests.

3.2.4.1 NVM Memory Corruption Tests:

The NVM Memory is where the SW configuration parameters are stored, and also where the diagnostic errors can be found and seen. If, during runtime, an error occurs the sensor should have a specific reaction to it. In these tests, the tester starts by getting the data from the sensor, depending on the Component Under-Test. The next step would be to use a Bosch Proprietary Application capable of accessing the RAM memory (it's also able to read and write in the memory directly) and corrupt the corresponding variable. This would make the sensor have a behavior flag turn up, corresponding to the invalidity status of the signals, being sent over CAN, happening after the memory corruption.

On Figure 3.11 it can be seen the first manual step needed to corrupt the NVM Memory on the right side and on the left side what is the current state of RAM Memory.

Right after rebooting the sensor, the tester needs to go back to CANoe and continue running the test. This way it can detect that the error is shown both on the RAM Memory list and on the Trace Window of CANoe, via the "*_Inv*" flags. Figure 3.12 shows how this is seen.

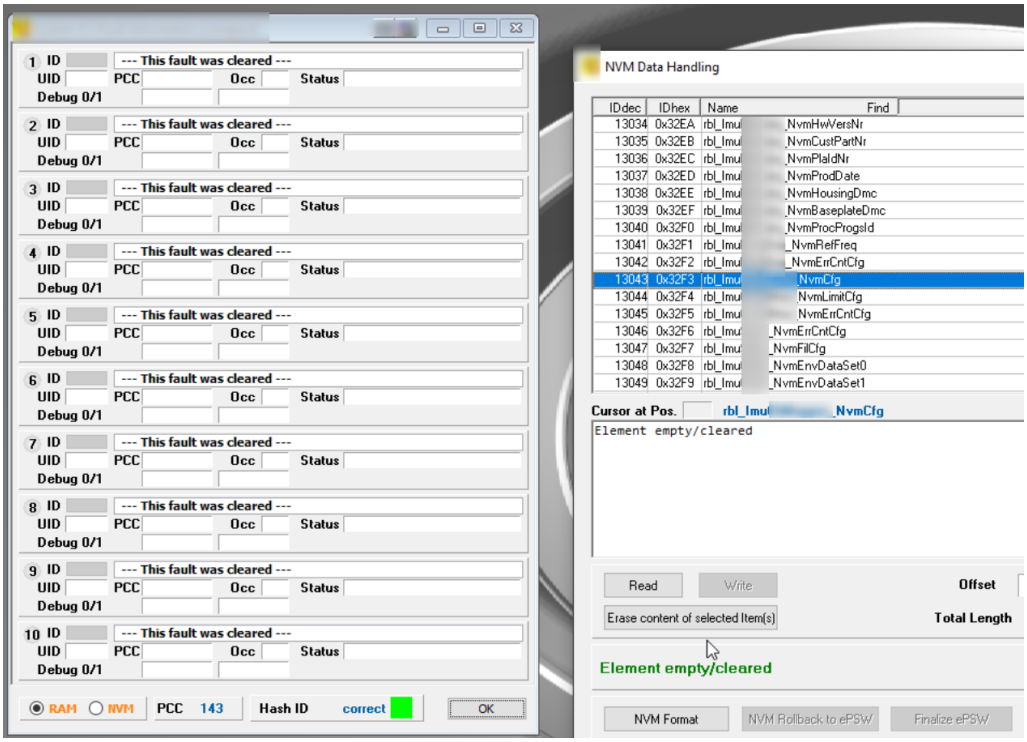


Figure 3.11: NVM Memory Corruption Step 1

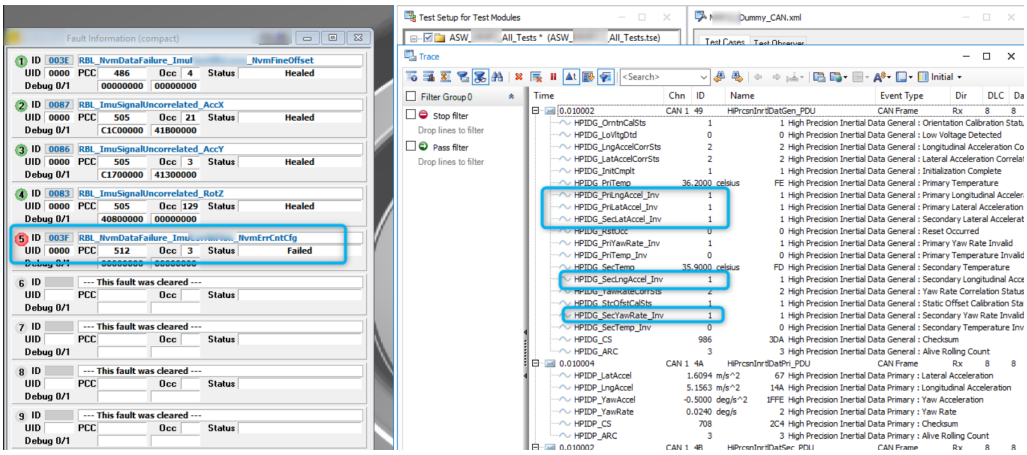


Figure 3.12: NVM Memory Corruption Step 2

With a test being performed, this kind of analysis is time consuming and prone to "*human error*", due to the fact that the tester starts running the tests on CANoe, and then during some set intervals the tester needs to manually interrupt the test and use the Proprietary Application to corrupt the specific variable from the memory, reboot the sensor to validate these changes and continue running the CANoe test. Like it was said before, these steps have to be correctly followed or it would lead to having the test redone multiple times due to the sensitive nature of manipulating the RAM memory. A solution to this problem would be automating these kind of tests, and then having them run in a Jenkins server, without the tester needing to supervise it.

3.2.4.2 SPI Simulizer Tests:

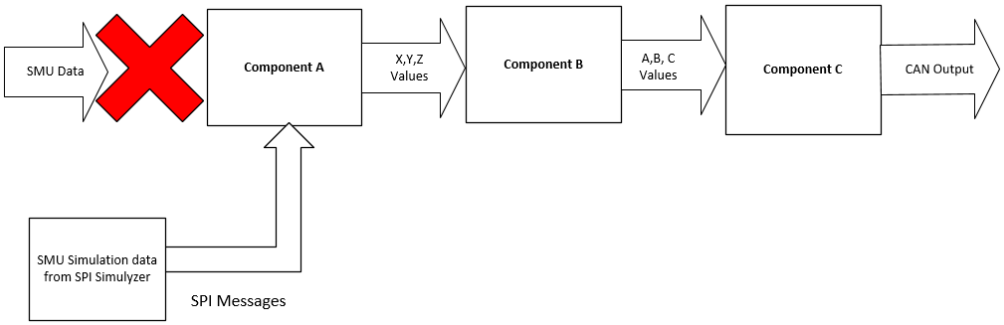


Figure 3.13: SPI Message Example

SPI Simulyzer tests allow fault injection into the SPI communication between the SMU and the microcontroller. These tests are performed using the SPI Simulyzer tool, used to simulate error scenarios, for example, sending data frames with fail messages or sending the temperature values out of the expected range.

The tests are performed on CANoe where the sensor is running its start up steps. The execution is then stopped after a specific timeout and the tester needs to continue with the SPI Simulyzer to keep performing the tests. An example is shown in Figure 3.13, where the expected behavior of this test can be understood.

This tool works in a very specific way with the help of a MathLab generated script. With Mathlab a script is created to generate the SPI frames in a specific time frame and with the expected fault. The SPI Simulyzer then reproduces these frames in the bus communication, exactly as how an SMU would do. These messages sent over the SPI allow the tester to verify if the IMU would have the expected behavior in case it was a real SMU generating those SPI frames.

An example of how the SPI Simulyzer normally runs during an test can be seen on Figure 3.14. The Sensor Data window (lower left side) is where the data being transmitted through the SPI Communication shows up. The Logic Analyzer window (top right side) gives the tester a graphical interpretation of the Sensor Data.

Figure 3.15 is an example of how the CANoe usually looks during these tests. The important information can be seen on the "Trace" window, where the tester sees which SMU is being simulated. On the XCP window of CANoe, the tester can select what

are the signals being tested and that should be verified on the "Data" window. In this example, the tester verifies that the Sensor Data Signal regarding the Acceleration 0 ("ACC_0") Status has an Error status in this time of the test.

The SPI Simulyzer tests are also another type of tests prone to "*human errors*" which would cause time to be wasted in case of a failed run. Also, these tests are hard to perform due to the fact that two different IMU sensors are needed to correctly test this behavior: a normal IMU connected directly to the Tester's Test bench with a normal number of SMUs, and another IMU with no SMU's on it. The SPI Simulizer cables are then connected to this second IMU in order to simulate how a SMU would respond during these errors.

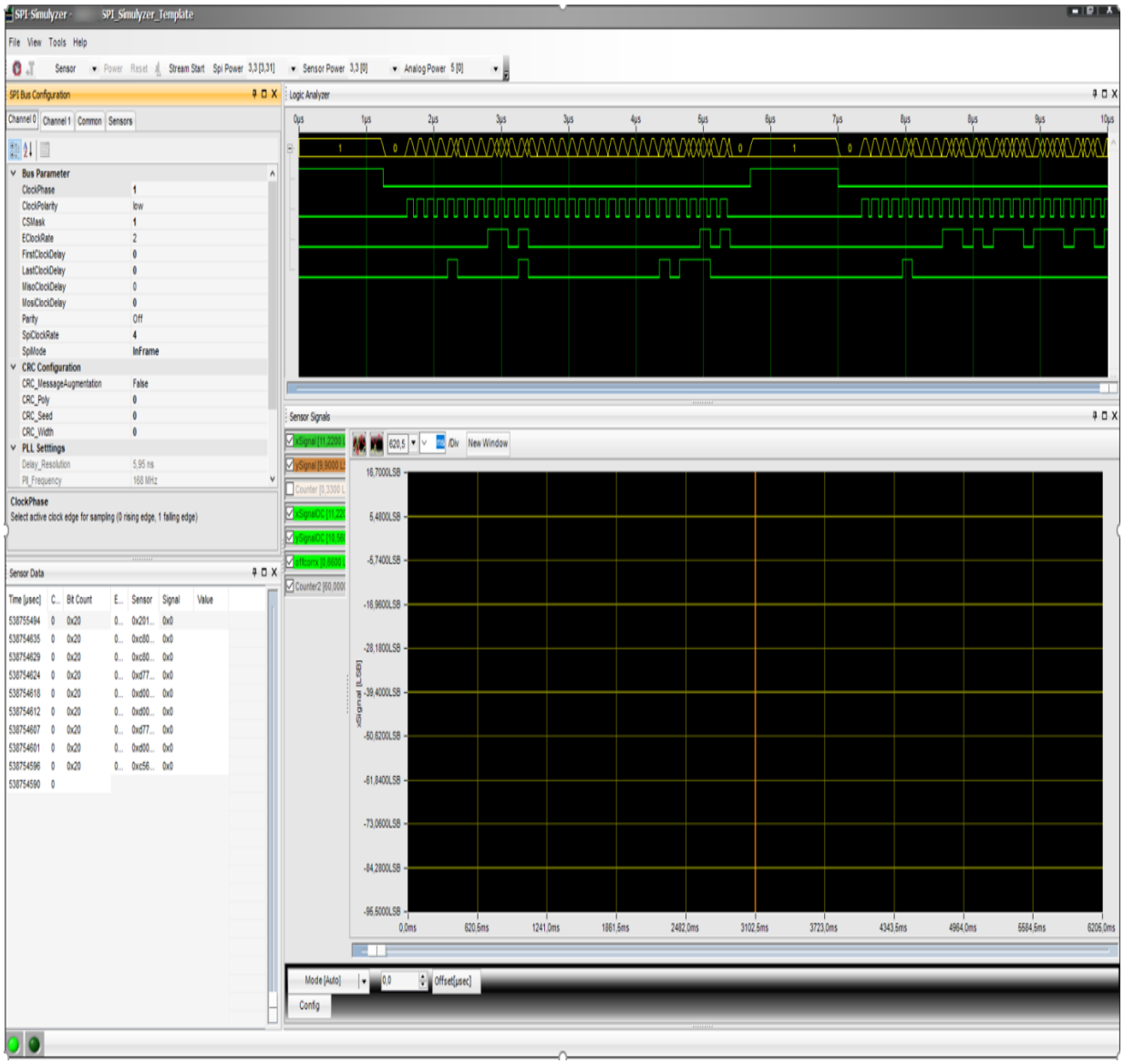


Figure 3.14: SPI generated Data Frame Example

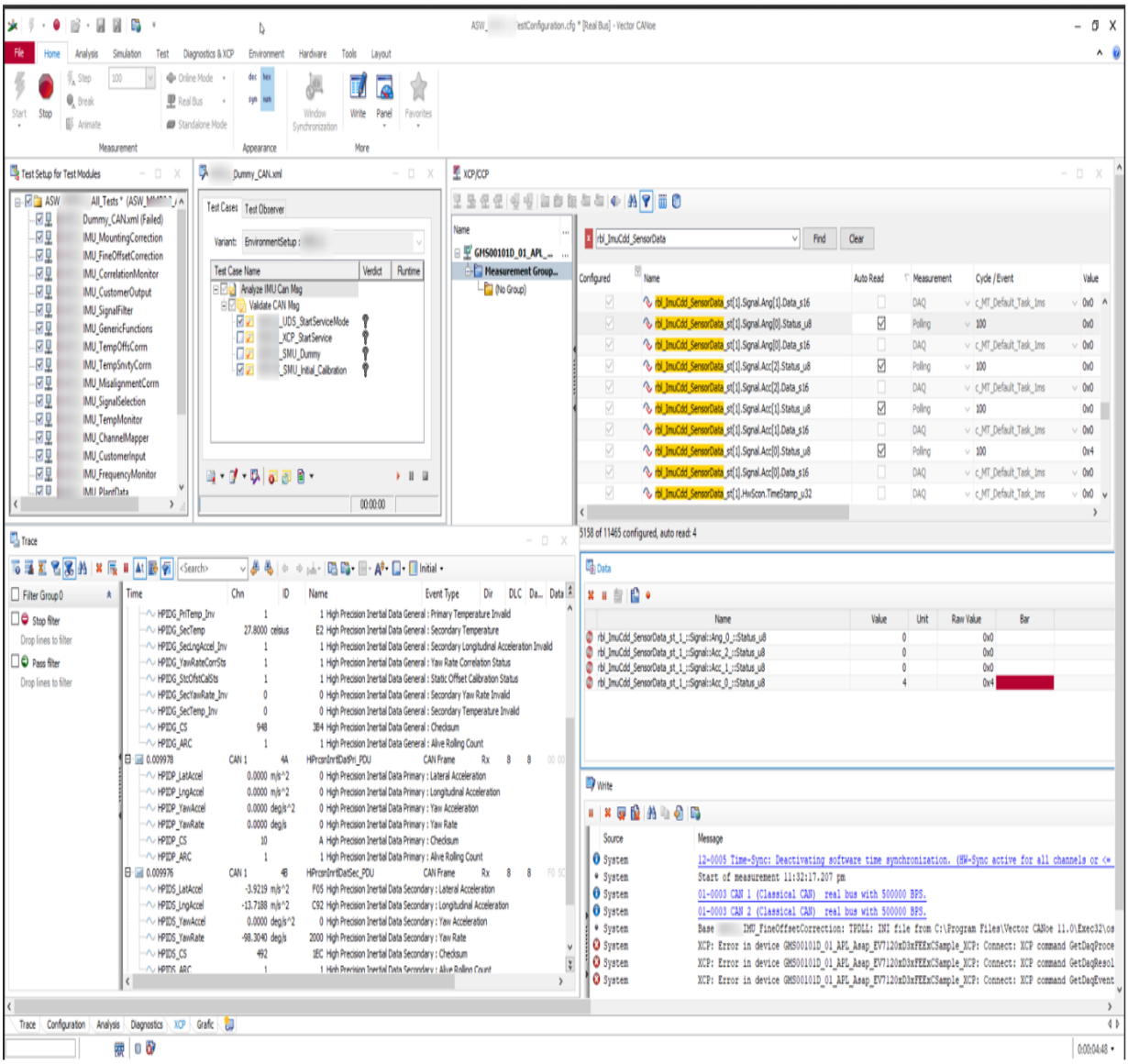


Figure 3.15: SPI generated Data Frame Example - CANoe test side

3.2.5 Normal Operation related Tests

Tests that consist on verifying if the sensor can operate normally, following a set of Requirements are know as Normal Operation tests. They can either use the UDS or the XCP protocol throughout the test, depending on the Component Under-Test, and they can be simple tests like verifying if the sensor is reading the correct temperature values or something more complex like switching the signal values between the SMU's communication channels.

In Figure 3.16 there is a example of how the Normal Operation usually occurs. The Temperature Check Component gets the Temperature values from the IMU sensor and sends them over to the Scramble Component. In this component the signals are scram-

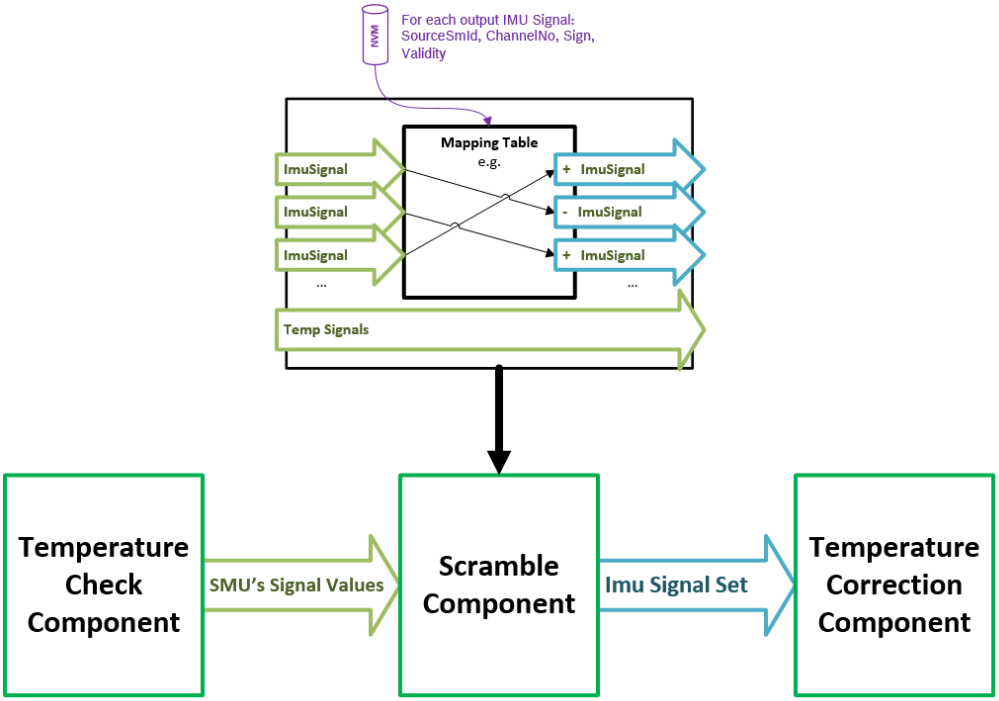


Figure 3.16: Normal Operation Example

bled having the NVM data as a base for the mapping table. The final IMU Signals are data that the micro controller can understand and proceed with the corrections done in the next component.

Due to the unstable nature of running XCP communication tests on CANoe, more times than expected the tests end up failing due to minimal variations of the read message. In order to mitigate this the automation of the Normal Operation tests is a solution, with the possibility of having them run on a Jenkins Server, where human iteration is unnecessary.

3.3 Manual Building and Flashing

Depending on the Test Campaign being run during large periods of time, the IMU's sensor SW needs to be updated. This requires that the SW Project is build successfully by the Developers and then made available to the Testers that would then Flash the SW into the sensor. Flashing the HW with the corresponding SW can be quite simple, and done in two different ways:

Flashing via Renesas Flash Program The sensor is connected to a needle bed and the flashing is executed with a JTAG connection to the debugging pin. Only the Sw's built hex file is needed for this operation and normally takes under one minute to finish.

Flashing via CAN A more complicated, and avoidable, process of flashing when no needle bed is available. A different hex file is used for this process together with a Bosch Proprietary Application for flashing. If not done correctly it could damage the sensor.

Normally, during the IMU's development life cycle, these SW updates occur quite often, due to bug fixing and new features being added to the SW. Due to SW updates being constantly created by developers when fixes are needed, not always do the testers have the most recent SW to test and verify. The sharing of the newest SW between teams usually takes a bit of time and could lead to the Team getting the update too late, leading to big delays on the delivery.

A possible solution to this would be with the implementation of a Jenkins server, where both the SW Project source code could be compiled and built. Giving the testers a place to check the existence of a new SW update whenever there was an update of source code.

3.4 Proposed Solutions

After understanding what type of tests are performed, and also all the other steps needed to perform in between work routines, only one solution seems viable, and has been described on all the last sections. The automation of SW test performed and the creation of a Jenkins server specific for testing.

This solution came to mind due to the fact that most of the problems seen while testing were due to either human errors or human interaction with the tool. As already explained, with the automation process of theses tests, time would be saved during test campaigns and also effort needed to re-run a failed test attempt. With the addition of a test-specific Jenkins server for the IMU sensor, almost all of the process can be automatically run online, whenever a SW update shows up or whenever a major change to the SW variables were made.

Chapter 4

Resolution of the Problem

This chapter contains the discussion on how to solve and correct the issues presented on Chapter 3 of this dissertation.

4.1 Automated Tests

As discussed on **Chapter 3**, most of the problems present while testing manually were due to Human Errors and their side-effects. This means that a possible way to address the problem was to automate how the tests are made and executed.

In order to understand what steps were needed to automate the shown testing process, firstly an evaluate of the whole process is needed, as previously regarded on the sub-section **"Testing Process"**.

It is important to understand that the tests performed by the Team are know as SW Test on the System and for this to be done the SW under test needs to be flashed on the sensor.

Testing only starts after the **Application Container** (i.e. the built project), is made available to the Team. This is due to the fact that the first test process step, show in Figure 4.1, needs such application container.

The Application Container is a collection of files generated during the project's build, that are then used to flash the sensor with the specific SW version present on the given Application Container. From this container only a few files are actually important for the testing, the ".a2l" file which is necessary to use the XCP's DAQ mode, the "DiagnosticDataList.csv" file which has all the UDS NVM memory ID addresses, the "EEPROM" container that has all the UDS NVM memory variables and the ".hex" file used to flash the sensor.

Usually these files need to be patched before usage in order to be possible to use them for testing. For example, both the ".a2l" file and the "EEPROM" container have their name changed to a more generic name, always used in the CANoe's configuration file (.cfg). In addition, the ".a2l" file also needed to be adapted in order to run XCP's DAQ Mode in a very specific reading rate. After the patching is performed, these files

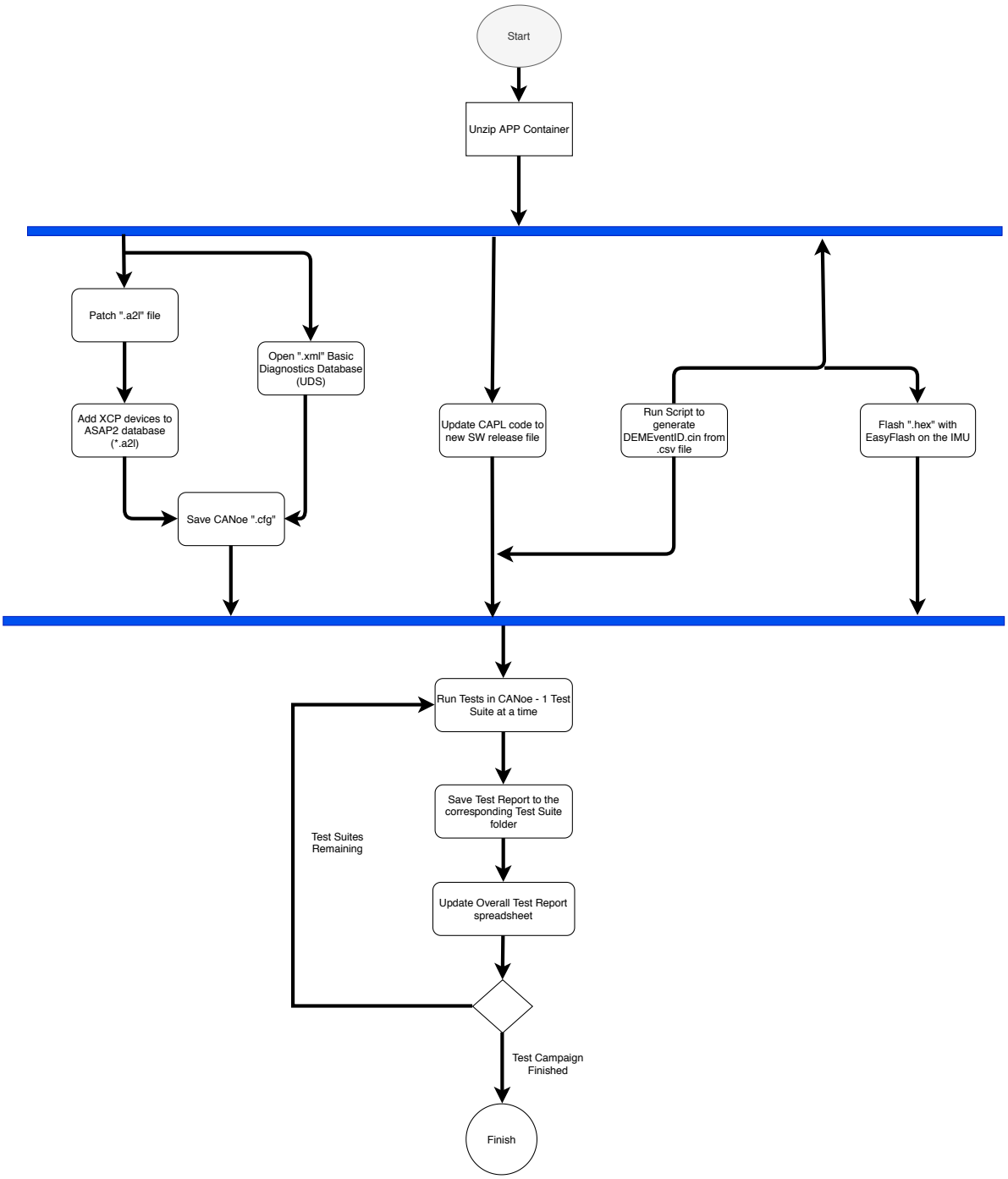


Figure 4.1: Testing Process Activity Diagram

are then added to the CANoe's variant-relevant configuration options. These steps are very important in order to have the testing working properly.

After all these steps are completed, the Qualification Test Campaign can be executed using CANoe.

The developed Automation Process of the CANoe testing focuses on the steps shown on the previously described Figure 4.1. During the test activities performed for the SW release, according to what is described on sub-section **"CAPL Testing"**, Fault Injection tests were deemed the most critical point. This was due to the fact that they needed to be reworked in order to function independently of the SW Version and of the SW Tester. The Normal Operation related tests also were in need of few alterations in order to function independently of the SW Version, in this automated environment.

4.1.1 From Manual to Automatic Tests

Based on the previously shown Figure 4.1, the automation process can be simplified after the creation of a script capable of performing changes to the **Diagnostics Event Manager [DEM]** IDs stored in the test files.

Each new Application Container has a series of different files needed to run the newest SW. From this selection of files, in regards to the DEM IDs, the focus is on the **"Diagnosis Data List Project" csv** file. The structure of this csv file is always the same, although the IDs values may change occasionally.

A script was created to simplify the process of manually checking and changing these values in each Component's test. The script reads the "Diagnosis Data List Project" csv file and translates the columns related to the DEM variables and their corresponding IDs and writes the output into a new CAPL include file (the .cin files).

The script runs from the Windows Command line, with the needed inputs (usually the location of the .csv file and the location where the ".cin" file should be stored).

These .cin files are given a name based on the Variant being worked on and the SW Version derived from. The code inside will be divided into the variable initialization, Figure 4.2, and the the function used to call the respective values for the Tests, Figure 4.3.

With this out of the way, the automation process became easier due to simplification of loading NVM values for each test.

```
S_DEMEventID.cin
/*@!Encoding:1252*/
includes
{}
variables
{

 /* Voltage monitor of microcontroller has detected a failure of the microcontroller supply voltage. */
 byte RBuCSupplyError[2];

 /* HWBIST detected a error. */
 byte RBHWBISTError[2];

 /* RAM Failure */
 byte RBRAMFailure[2];

 /* Flash failure detected (incorrect checksum&#44 */
 byte RBFlashFailure[2];

 /* Microcontroller safety logic has detected a failure. */
 byte RBuCSafetyFault[2];

 /* Startup safety test detected a failure of uC safety logic. */
 byte RBuCSafetyLogicFault[2];

 /* CPU exception occurrd. */
 byte RBCPUException[2];

 /* Stack over- or underflow */
 byte RBSYS_StackOverUnderFlow[2];

 /* Internal interrupt based system error */
 byte RBSYS_SYSErrorHook[2];

 /* Operating System error (e.g. task overrun&#44 */
 byte RBSYS_OSErrorHook[2];

 /* Runtime overload */
 byte RBSYS_TaskOverRun[2];
```

Figure 4.2: DEM Event ID Generated Library - Variable Initialization Example

4.1.1.1 NVM Memory Corruption Tests Automation

The NVM Memory Corruption Tests are heavily influenced by the DEM IDs changes. With this part already automated, it was possible to adapt the tests and have them working automatically, with no need to pause the it mid-way and corrupt the memory.

To start off, the tester had to look at how the tests were written before. The NVM Memory Corruption Tests were divided into three different sub-tests that had to be run sequentially with the help of an Bosch Proprietary Application. This application was used to corrupt the memory, and then the Tester had to "confirm" if the manual steps had been performed, before and after running each sub-test. On Figure 4.4 its possible to see an example of how the sub-tests were written before. The first sub-test had a *testwaitforTesterConfirmation* function, which was used to get the result of the test from the tester over an input that was then passed to the next sub-test, and depending on how the Tester was working, the test could easily fail and need to be restarted from the beginning.

This whole process had a lot of complications, like explained before on **Chapter 3. - CAPL Testing**.

After studying how the CANoe tool works and the various CAPL functions available to use, these manual steps were replaced by automatic steps on CANoe.

In Figure 4.5 there is a function used to check if the Fault Memory List is free of errors and a function used to delete the NVM memory associated with the variable in question (in this example it was the *Frequency Monitor Error Counter*).

```
void loadPlantDiagID()
{
  RBuCSupplyError[0]  = 0x00;
  RBuCSupplyError[1]  = 0x01;

  RBHWBISTError[0]  = 0x00;
  RBHWBISTError[1]  = 0x02;

  RBRAMFailure[0]   = 0x00;
  RBRAMFailure[1]   = 0x03;

  RBFlashFailure[0]   = 0x00;
  RBFlashFailure[1]   = 0x04;

  RBuCSafetyFault[0]  = 0x00;
  RBuCSafetyFault[1]  = 0x05;

  RBuCSafetyLogicFault[0]   = 0x00;
  RBuCSafetyLogicFault[1]   = 0x06;

  RBCPUException[0]   = 0x00;
  RBCPUException[1]   = 0x07;

  RBSYS_StackOverUnderFlow[0]   = 0x00;
  RBSYS_StackOverUnderFlow[1]   = 0x08;

  RBSYS_SYSErrorHook[0]   = 0x00;
  RBSYS_SYSErrorHook[1]   = 0x09;

  RBSYS_OSErrorHook[0]  = 0x00;
  RBSYS_OSErrorHook[1]  = 0x0A;
```

Figure 4.3: DEM Event ID Generated Library - Function Example

```
    TestWaitForTesterConfirmation("Read the DEM with NVMAPP. The DEM shall have no errors. Proceed by clearing the error counter parameters in NVM", 60000);
}

testcase  tc17328_ImuFreqMon_OpFail_ErrorCounter_b()
{
  long resultTesterConfirmation;

  resultTesterConfirmation = TestWaitForTesterConfirmation("Is the DEM empty? Were the error counter parameters cleared in NVM?", 60000);

  if(resultTesterConfirmation!=1) testStepFail("6","The DEM was not read with NVMAPP or there were errors in DEM.");
```

Figure 4.4: Manual Step Confirmation - Example Code for NVM Memory Corruption Tests

Since it was no longer needed to manually corrupt the NVM memory, variables had to be set, so that the tester knew what was being corrupted and what was being checked. With the DEM IDs .cin file, the tester just had to call the function to load the variables IDs needed and use them directly on the test.

Figure 4.6 show the variable definition method and how it is used to verify the status of the DEM Fault Errors list. It also shows the method used to verify if that variable is present on the list or not.

For this example, the DEM ID being checked is regarding the *Frequency Monitor Error Counter* (ID obtained directly from the DEM ID .cin file), and the expected status would be of "1" (Failed). The status that can be verified using this method are **Failed ("1")** (Failed status only occurs after memory corruption has been made), **Healed ("2")** (Healed status can only occur after the correction of a Failed status) and **Nothing ("0")** (only occurs when nothing has altered the check variable).

```
/* Check DEM record.  */
UDS_Check_RbFaultMemEntryNoX_Empty("5");

/* Corrupt / Delete the NVM entry where the frequency monitor error counter limits are saved */
UDS_EraseDataMemory("6", rbl_ImuFreqMon_NvmErrCntCfg);
```

Figure 4.5: NVM Corruption Example - Code for checking the Fault Memory list and for Corrupting the Memory

```
/* Verify DEM new entry */
/*  "NVM Data rbl_ImuFreqMon_NvmErrCntCfg missing or corrupt" EventID shall be present with status "Failed" */
Expected_UDS_RbFaultMemEntryIndex_st.DemEventID[0] = RBL_NvmDataFailure_ImuFreqMon_NvmErrCntCfg[0];
Expected_UDS_RbFaultMemEntryIndex_st.DemEventID[1] = RBL_NvmDataFailure_ImuFreqMon_NvmErrCntCfg[1];
/*  Failed -> EventStatus = 1  */
Expected_UDS_RbFaultMemEntryIndex_st.EventStatus = 1;

UDS_Find_RbFaultMemEntryNoX("10.1", Expected_UDS_RbFaultMemEntryIndex_st, 1);

/*  "NVM Data rbl_ImuFreqMon_NvmRefFreq missing or corrupt" EventID shall not be present in DEM */
Expected_UDS_RbFaultMemEntryIndex_st.DemEventID[0] = RBL_NvmDataFailure_ImuFreqMon_NvmRefFreq[0];
Expected_UDS_RbFaultMemEntryIndex_st.DemEventID[1] = RBL_NvmDataFailure_ImuFreqMon_NvmRefFreq[1];

UDS_Find_RbFaultMemEntryNoX("10.2", Expected_UDS_RbFaultMemEntryIndex_st, 0);
```

Figure 4.6: NVM Corruption Example - Code for verifying Fault Errors list

With the automation of these kind of tests, the tester stopped needing to change the DEM IDs values manually each time a new SW was available. This in turn minimized the risks of having failing tests due to DEM IDs values not being the correct one to that specific test case. It also save time during the execution of tests, due to the fact that now the specific memory was being corrupted automatically.

4.1.1.2 SPI Simulyzer Tests Automation

Getting back to the SPI Simulyzer Tests, they proved to be quite difficult to automate. Like mentioned before on **Chapter 3 - CAPL Testing**, SPI Simulyzer tests are prone to human errors due to the number of manual steps needed for each test run.

In order to help minimize the problems regarding these tests, the automation process was divided into three different steps:

- Generate a new MathLab script with new SPI frames in a specific time frame;
- Create a script to run the SPI Simulyzer application on the background;
- Adapt the CANoe Tests to run the SPI Simulyzer scripts.

Using the code example of a test case about " CN bit failure", the tester adapts the code to run the SPI Simulyzer program in the background and in silent mode, making it possible to have the test running automatically and with no need for manual outputs.

The Invalid_CN.xml file is the one specified on **Chapter 3 - Fault Injection related Tests**. This .xml file is generated using the MathLab script, which has been altered to run on specific time frames for the developed automated process. It contains a values list of time frames that the SPI Simulyzer reads, adapts and then runs as a simulation of a SMU behavior.

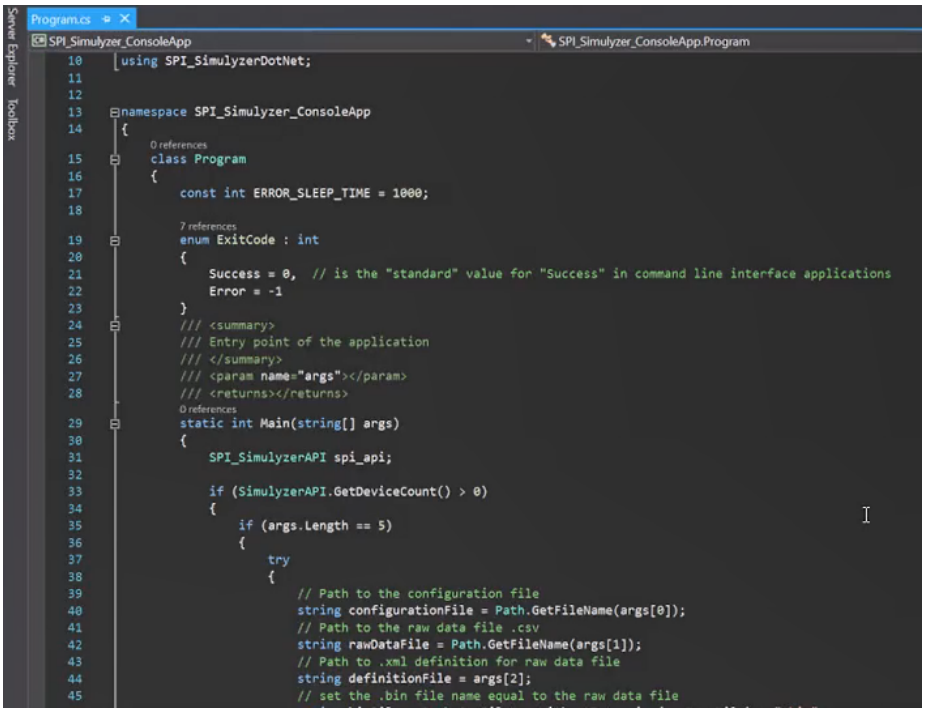


Figure 4.7: Script for Running SPI Simulyzer

A script (Figure 4.7) was created in order to call and run the SPI Simulyzer program in the background, silently, with only the test case inputs. The inputs given are regarding the location of the .xml file read by the SPI Simulyzer program, the path of the executable test script and the timeout given before the test can keep running. Figure 4.9 exemplifies how the test script processes this information.

This script then communicates with the program using the installed SPI Simulyzer DLL's files, in a way that is meant to work how the tester wants and needs it to.

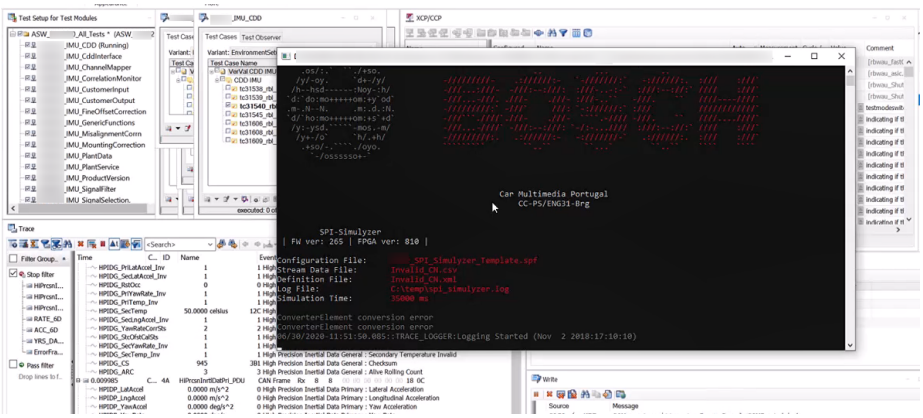


Figure 4.8: SPI Pop-Up test window

Figure 4.8 shows how the script window pops-up when performing the specific tests

that used SPI Simulyzer.

In Figure 4.9 it's possible to see that this code turns the power supply off at the beginning of the test, sets the absolute paths to the script that will run the SPI Simulyzer program, and then, using a *sysExec* function, executes the script, with the IMU SPI Simulyzer Template, the " Invalid_CN.xml " file and the path to store the log file.

After this step has finished, the test waits until the end of the designed timeout. This ensures that the SPI Simulyzer program has finished its simulation and the test can continue.

```
testcase tc31540_rbl_ImuCdd_CNbitFailure()
{
  struct rbl_ImuStdTypes_ImuSignalSet_tst Expected_CAN_Output_ImuSignalSet_st;
  char absFilePath[2048];
  char absPath[2048];


  testReportAddExternalRef("url"," Test Case 31540","https://rb-alm-06-p.de.bosch.com/qm/web/console/Complex%20Sensors%20-%20Quality
  testCaseDescription("Verify CDD handling CN bit failure.");


  /*  1 Set Power Supply OFF  */
  SetPowerOFFChannel(2);
  testStepPass("1", "Set power supply power OFF.");


  /*  2 Run SPI-Simulyzer simulation  */
  getAbsFilePath("SPI-Simulyzer\\SPI_Simulyzer_ConsoleApp\\SPI_Simulyzer_ConsoleApp.exe", absFilePath, elcount(absFilePath));
  getAbsFilePath("SPI-Simulyzer\\SPI_Simulyzer_ConsoleApp", absPath, elcount(absPath));
  //Invalid_CN
  sysExec(absFilePath, "     _SPI_Simulyzer_Template.spf Invalid_CN.csv Invalid_CN.xml C:\\temp\\spi_simulyzer.log 35000", absPath);

  testWaitForTimeout(10000);


  /*  3 Set Power Supply ON */
  SetPowerONChannel(2);
  testStepPass("3", "Set power supply power ON.");
```

Figure 4.9: SPI Simulyzer Code Calling Example

This automation process took longer than expected to conclude due to the fact that working with SPI Protocol messages communication is extremely time sensitive and these timings had to be properly accounted in order to have the expected message flags.

4.2 Automated Tests with Jenkins

An CI server is actually a collection of test benches interconnected between them as Nodes. In order to setup a Jenkins Server specific for the Team's tests, a new test bench is needed to connect to the master CI Server. This test bench function will be of exclusive use to Jenkins Testing, as it will be running the Jenkins client interface. Having a designated test bench for Jenkins Testing will allow the Team to automatically run a full Test Campaign whenever it is needed, without the tester's interference. Running a full Test Campaign is basically a full run of all the existing Test Suites and saving the Test Results. It is important to always run a full Test Campaign whenever there is a big alteration to the SW.

Now that the Test Suites are fully automated, the Team is one step closer to getting Jenkins up and running as planned and described.

But in order to start this task, a few things needed to be implemented first. Namely, the Jenkins Setup requirements to connect to a CI Server.

The Jenkins Setup requirements were as follows:

- Hardware related:

  IMU Client specific sample board;

Vector Box with CANoe and XCP;

Desktop PC;

Power Supply;

Debugger;

- Software related:

  The SW needed to run a CANoe test;

  Creation of a profile, were all the specific SW and Tools needed are stored, in case everything has to be installed in a new Test Bench;

- Tests related:

  Batch Script to start the Test Execution;

  Configure the Relation between the Jenkins Server and the Test Bench;

  Configure the Interface Shown on the Jenkins Server;

  Nightly and manual build execution (test case selection), calibration, reprogramming.

Only after all these requirements are solved could the team's Jenkins-specific Test Bench be connected to the CI server.

4.2.1 HW related Requirements and solutions

The HW related dependencies were due to the fact that a new desktop would have to be prepared as a "Jenkins-Only" machine, and so it needs all the HW components usually needed for a normal Tester's Test Bench. After everything was collected and assembled, the system architecture would look like the one shown in Figure 4.10.

With this completed, the test bench would only need the required SW to be able to connect to a Jenkins CI server and function as a node of the system.

4.2.2 SW related Requirements and solutions

Regarding the SW related requirements, everything was already ready to be installed in the new machine.

As a way to automatically distribute the SW Tools to the Test Bench, a profile account was created. This profile account would be known as the Test User account. The account would then be used to obtain and register the SW and tools necessary to execute the its test bench functions. With everything needed to operate registered to this account, that would mean that whenever it was used to login in a new test bench, it would automatically download and install the required SW and tools registered to it.

This step was quite simple to solve as everything was ready to be installed and configured.

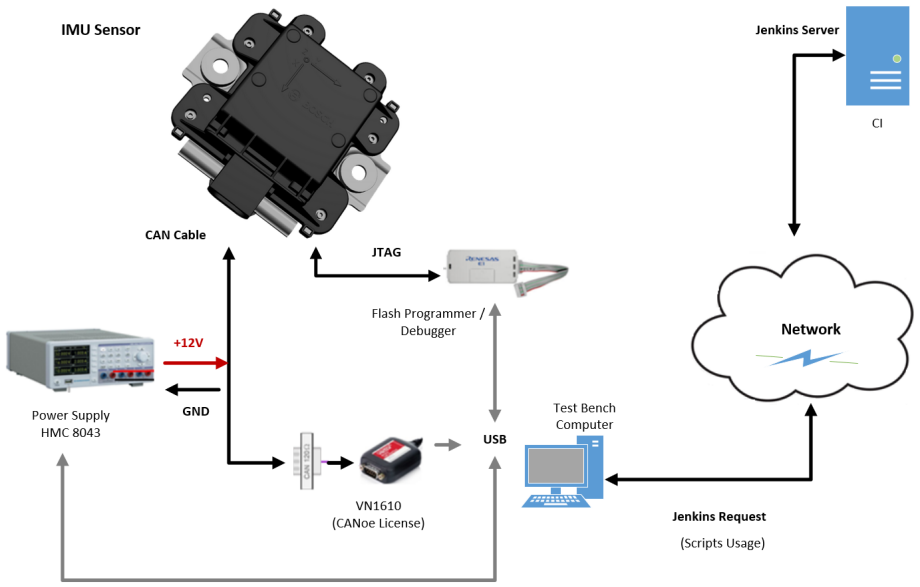


Figure 4.10: Jenkins Test Bench Setup Architecture

4.2.3 Test related Requirements and solutions

Regarding the Test related requirements, this is where the most work had to be done. Only after the SW and HW dependencies had been solved, could the connection between the Test Bench and the CI server be possible. With the connection established, a script was needed to execute the exact steps of a Automated Test run. At the same time configurations between the Jenkins Server and the Jenkins Test Bench were needed to ensure the correct functionality between them on each Build executed. A plan also needed to be established to ensure that there existed a nighly and a manual build. A Nightly build would only contain "Released Tests" and would only run once each night to guarantee that the development stream didn't contain SW that would break the tests. "Released Tests" meant tests that were considered ready and with a small margin of failing. Their purpose would be to test the sensor and be executed on each nightly build. A Manual build would contain all the tests available. Even the ones not yet ready and passable. The purpose of the build would be to check if the tests were ready to be run on a Nightly build bases or if they still needed to be corrected.

4.2.3.1 Test Execution Script creation

The Test Execution Script had to fulfill different requirements in order to be executed on the Jenkins CI Server:

- A way to get the Application Container;
- A way to select the Variant Under Test;
- A way to flash the sensor with the SW from the Application Container;
- A way to select the Tests to be executed;

- A way to run the tests and save the test reports.

All of these steps are part of a set of PowerShell scripts created by the CI Team for another project. These scripts were then greatly altered and upgraded to run specifically for the IMU sensor. The adaptations ranged from changing the type of flashing used for this specific sensor, introducing skip flashing functions, picking certain files from the Application Container and changing their names and contents and then storing them in a specific folder, and many more.

These script would then interacts with all the others and is called *CAPLTest.ps1*. In order to run this script two different base script libraries are needed, one for the power supply named *HMC804X.ps1* and one for the flashing of the sensor, *RFPE1RH850.ps1*. The Application Container is obtained directly from another Jenkins pipeline, that specifically performs the build of the system whenever a change happens to the SW's stream, where the project code is stored. Another file needed to execute the *CAPLTest.ps1* script is a CAPL Test JSON file, where the variants would be organized with the structure shown on Figure 4.11.

```
- Structure:
        |-- Variants:
                |-- <Variant>:
                        |-- TestList:
                                |-- ReleasedTest: [
                                                |-- archive : [String]
                                                |-- expression : String
                                                |-- file : String (Relative Path from the $CAPLTestsRootPath)
                                                |-- timeout : int
                                        ]
                                |-- TestUnderDevelopment: [
                                                |-- archive : [String]
                                                |-- expression : String
                                                |-- file : String (Relative Path from the $CAPLTestsRootPath)
                                                |-- timeout : int
                                        ]
                |-- (...)
```

Figure 4.11: CAPL Tests JSON file example structure

The *CAPLTest.ps1* script can then be run locally to see if everything is running like expected, or in the Jenkins pipeline specific to the Team's test execution.

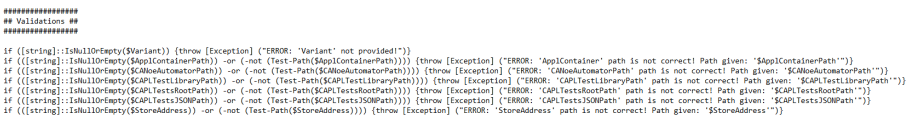
To run the script locally, the following Command line commands are used:

```
powershell.exe C:\path\to\script\CAPLTest.ps1
-Variant "VariantA"
-ApplContainerPath "C:\path\to\zip\file\ApplContainer_BB99999_VariantA.appl.zip"
-CAPLTestsRootPath "C:\path\to\folder\with\testautomation"
-StoreAddress "C:\path\to\folder\where\to\store\Results"
-Cleaning
```

The command(s) commands execute the script *CALPTest.ps1*, provides the correct variant to be considered, the path on which the Application Container can be found, the path where the CANoe tests are located and a path to store the final test results.

The script first validate if all the inputs given are correct or not (Figure 4.12).

The script then sets some constant variables and imports the needed libraries (Figure 4.13).



```
###############
## Validations ##
###############

if ([string]::IsNullOrEmpty($Variant)) {throw [Exception] ("ERROR: 'Variant' not provided!")}
if (([string]::IsNullOrEmpty($ApplContainerPath)) -or (-not (Test-Path($ApplContainerPath)))) {throw [Exception] ("ERROR: 'ApplContainer' path is not correct! Path given: '$ApplContainerPath'")}
if (([string]::IsNullOrEmpty($CANoeAutomatorPath)) -or (-not (Test-Path($CANoeAutomatorPath)))) {throw [Exception] ("ERROR: 'CANoeAutomatorPath' path is not correct! Path given: '$CANoeAutomatorPath'")}
if (([string]::IsNullOrEmpty($CAPLTestLibraryPath)) -or (-not (Test-Path($CAPLTestLibraryPath)))) {throw [Exception] ("ERROR: 'CAPLTestLibraryPath' path is not correct! Path given: '$CAPLTestLibraryPath'")}
if (([string]::IsNullOrEmpty($CAPLTestsRootPath)) -or (-not (Test-Path($CAPLTestsRootPath)))) {throw [Exception] ("ERROR: 'CAPLTestsRootPath' path is not correct! Path given: '$CAPLTestsRootPath'")}
if (([string]::IsNullOrEmpty($CAPLTestsJSONPath)) -or (-not (Test-Path($CAPLTestsJSONPath)))) {throw [Exception] ("ERROR: 'CAPLTestsJSONPath' path is not correct! Path given: '$CAPLTestsJSONPath'")}
if (([string]::IsNullOrEmpty($StoreAddress)) -or (-not (Test-Path($StoreAddress)))) {throw [Exception] ("ERROR: 'StoreAddress' path is not correct! Path given: '$StoreAddress'")}
```

Figure 4.12: CAPL Script - Validation function

```
########################
## Constant Variables ##
########################

$CurrentDir = (Get-Location).Path

## Variable to determine if the status of the execution
$global:CAPLTestsNumberFailures

[String] $HexFilePath = "$CurrentDir\ApplContainer"
[String] $HexFilePattern = "PRJ_Hexfile_*_$Variant.hex"


#############
## Imports ##
#############

## Import PowerShell CAPL Test Library
Import-Module "$CAPLTestLibraryPath"

if (-not $SkipFlash.IsPresent) {
        ## Import Flash Scripts
        Import-Module "$FlashScriptsRootPath\lib\Commons.ps1"
        Import-Module "$FlashScriptsRootPath\lib\HMC804X.ps1"
        Import-Module "$FlashScriptsRootPath\lib\RFPE1RH850.ps1"
        Import-Module "$FlashScriptsRootPath\app.ps1"
}
```

Figure 4.13: CAPL Script - Constant Variables and Imports

The following steps on the script are meant to read the CAPL Tests JSON file and evaluates which tests shall be executed. It then proceeds to download and extract the Application Container, get the hex's file Filename and flash the board with it (Figure 4.14).

After the flash process is finished, the script calls the CANoeAutomater application (Bosch Proprietary Application code), that opens the CANoe test configuration file and runs the test modules selected for this test run. The tests are executed and the generated tests reports are stored in a newly created zip file (Figure 4.15).

If the script is executed successfully the local build and test run were successful and the respective files can then be sent to the stream. Once the files are on the development stream they can be executed on the Jenkins CI Server.

```
#################################
## Process CAPL Tests JSON File ##
#################################

[Hashtable] $CAPLTestsJSONObject = ParseJson -File "$CAPLTestsJSONPath"

## If there are not CAPL Test configured to this Variant, skip the CAPL Test execution
[System.Collections.ArrayList] $CAPLTestsList = $CAPLTestsJSONObject.variants.$Variant.testList.$CAPLTestList

if ($CAPLTestsList.Count -le 0) {
        Write-Warning "There are no '$CAPLTestList' CAPL Tests configured for the Variant: '$Variant'!"
        Write-Warning "CAPL Testing Skipped!"
        return
}

#######################################
## Download and Extract ApplConatiner ##
#######################################

Write-Host "|-- Start Download ApplConatiner..."

[String] $ApplContainerFilename = Split-Path $ApplContainerPath -Leaf

CopyFile -Filepath "$ApplContainerPath" -Destination "$CurrentDir"

ExtractCompressedFile -ZipFilepath "$CurrentDir\$ApplContainerFilename" -Destination "$HexFilePath" #-RemoveZip

Write-Host "    |-- Download ApplConatiner succeed!"

########################
## Get '.hex' Filename ##
########################

$HexFilename = GetHexFilename -SourceFilePath "$HexFilePath" -Variant $Variant -HexPattern $HexFilePattern

        |
## Flash using EasyFlash
FlashEasyFlash -FlasherToolPath "$FlasherToolPath" -HexFilePath "$HexFilePath\$HexFilename"
```

Figure 4.14: CAPL Script - Process JSON File, ApplContainer and Flashing

```
#######################################
## Perform CAPL Tests with CANoeAutomator ##
#######################################

PerformCAPLTests -CANoeAutomatorPath "$CANoeAutomatorPath" -CAPLTestsList $CAPLTestsList -Variant $Variant -CAPLTestingResultStorePath "$StoreAddress\$Variant" -CAPLTestsRootPath $CAPLTestsRootPath -Cleaning:$Cleaning

#############
## Cleaning ##
#############
if ($Cleaning.IsPresent) {DeleteArtifact -ArtifactPath "$HexFilePath"}

##################
## Exit Function ##
##################
$host.SetShouldExit($global:CAPLTestsNumberFailures)
```

Figure 4.15: CAPL Script - Run Tests and Store Reports

4.2.3.2 Setting up Jenkins Connection

This was without a doubt the hardest part of getting the Jenkins connection started, due to the fact that the IMU Team depended on the CI Team's support. This team is located in a different region, the same location on which the Jenkins server is physically located. The support was needed due to the fact the CI Team had the knowledge and capacity to create and maintain a Jenkins CI Server. Knowledge-sharing is also an important topic between different projects.

A precondition to get this started was to have the base HW and SW ready on the Test Bench to be used, plus having a script able to flash the sensor, run the tests and generate and store the test reports. All of this has been accomplished, like mentioned before this subsection.

After sending the scripts to the CI team in Germany, and waiting a few days, a pipeline was established to the IMU's Jenkins Test Bench that was waiting to be connected.

With the creation of the Testing pipeline, configuring the Jenkins Testing server was

the next step.

Figure 4.16 shows us how the Jenkins Server Main page looks like. From here the testers can either select to go directly to the Project specific-pipeline and start and/or analyze the result of the tests. Or they can have access to a collection of different tools used to fix problems regarding the use of the server, e.g., requesting to disconnect the Test Bench from the Server connection in order to perform a local correction, or maintenance.

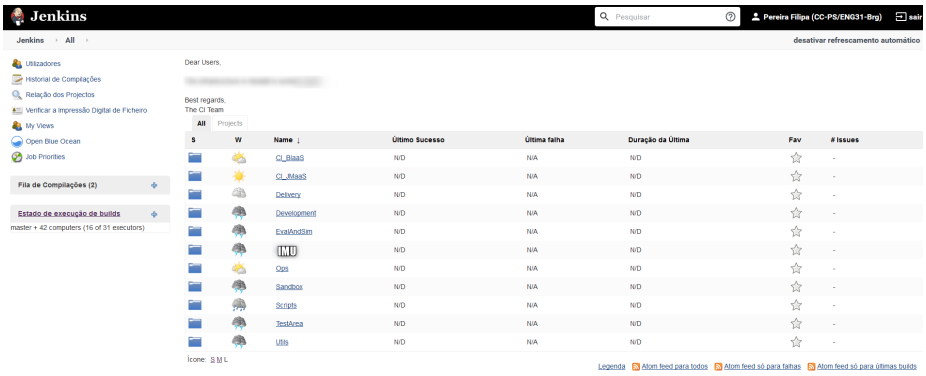


Figure 4.16: Jenkins Main Page for IMU Project

Selecting the IMU option, seen in Figure 4.16, takes the tester to a page where all the different pipelines can be accessed. These pipelines are needed in order to run the tests on the Jenkins Server.

The main branch pipeline is where it is possible to get the most recently generated Application Container. This Application Container is automatically generated each time something new is added to the project's stream, for each variant directly affected by the addition. Figure 4.17 shows how this page usually looks like.

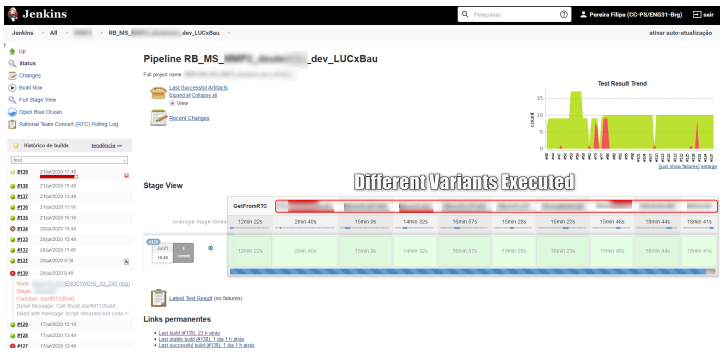


Figure 4.17: Jenkins Server Main Pipeline

Regarding Automated Test Execution, a new pipeline was created for the IMU Team to use for running tests automatically (Figure 4.18). The refereed pipeline is directly connected to the Jenkins Test Bench that was prepared beforehand.

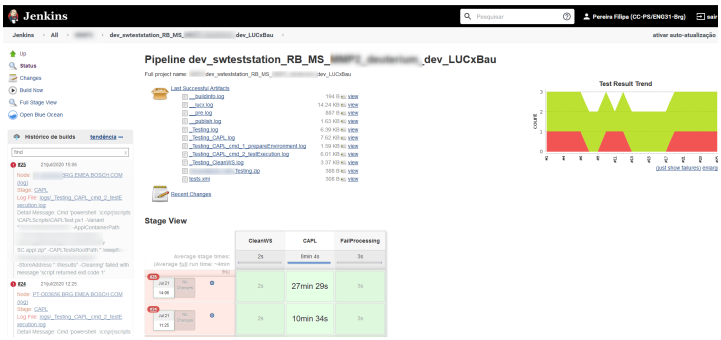


Figure 4.18: Jenkins Server Test Pipeline

Another automated script had to be prepared in order to solve most issues regarding getting the Jenkins build started. It is called *dev_swtestation.yaml* and its main purpose is to tell the Jenkins CI Server what to execute, and in what order. It is also responsible for how the "Stage View" (Figure 4.18) is represented in the Jenkins Testing Pipeline page.

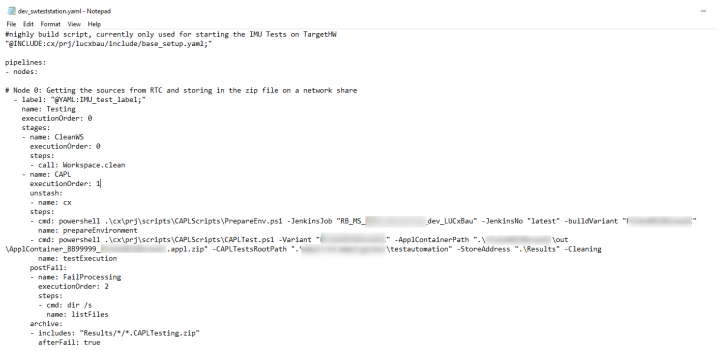


Figure 4.19: Jenkins Automated Script

The script seen in Figure 4.19 shows us how the process is done in the background and in which order. The most recent build of the Main pipeline is downloaded and used in accordance with the script for flashing the board, running the tests and storing the test reports.

4.2.4 The End Results - Jenkins Automation Completion

With all these topics solved, the creation of the new Jenkins Testing Pipeline was created and was ready to be used. It was now the Team's job to adapt and fix each and every Test Suite and have them running automatically on the Jenkins CI Server.

Since all the Components' Tests have already been automated locally and tested before being sent to the Project's stream, it is finally possible explain how this process is done.

The folder structure seen in Figure 4.20 is how the scripts are structured now, for easy access in each step of the way of the process.

From these folders only three files need special attention:

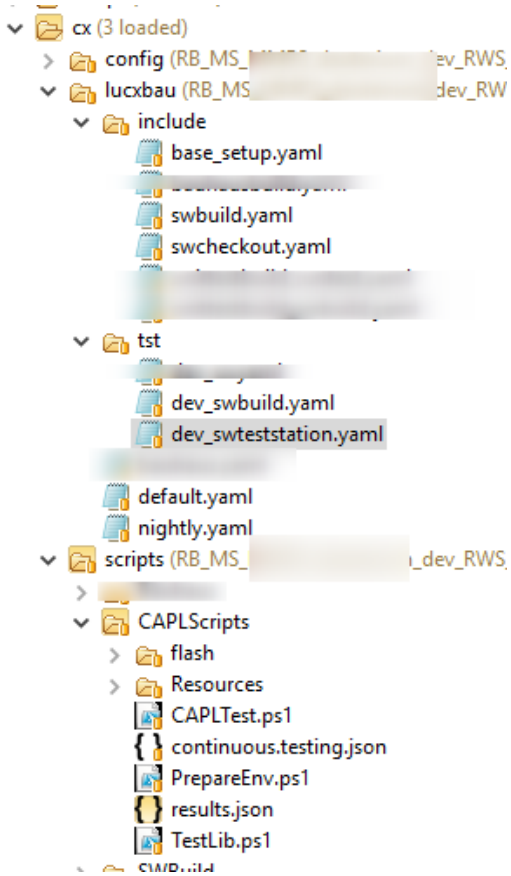


Figure 4.20: Jenkins Scripts Folder Setup

CAPLTest.ps1 The script created to flash the sensor, run the tests and store the reports;

continuous.testing.json The JSON file where the tests that shall run in the server are selected. It is also here that the path to the specific CANoe configuration file is given;

dev_swteststation.yaml The Jenkins script that runs everything in the background of a Jenkins Build. Here the Server gets the information of what Variant to use from the most recent build.

CAPLTest.ps1 and the *dev_swteststation.yaml* were already explained.

The continuous.testing.json file (Figure 4.21) is a compilation of all available variants. Which tests can be executed and from which test configuration they can be found. In this example, there is a VariantA and the component modules to test are the Temperature Monitor and the Frequency Monitor. Both these tests modules are present in the CANoe's IMU_Jenkins_TestConfiguration.cfg file.

For the purpose of this thesis, the Jenkins Test Pipeline was set to only execute Test Runs when manually requested by the Tester. This could be performed by pressing **Build Now** in the Jenkins Server Pipeline page (see Figure 4.18 for an example of the

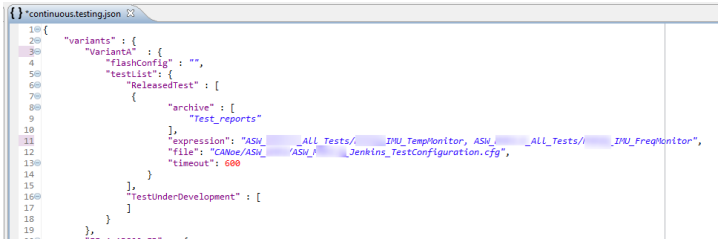


Figure 4.21: JSON file used to Run Tests on Jenkins

page). When a build finishes its execution, it generates various outputs that can be seen in the Build's results page (Figure 4.22). From these output files what the tester should always check is the **testExecution.log** and the **Variant_CAPLTesting.zip**.

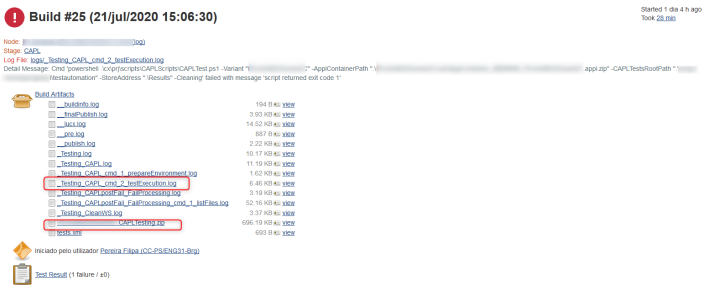


Figure 4.22: Jenkins Test Build Example

The testExecution.log gives us detailed information regarding the build. The log contains information regarding the flashing process, to information about the Test Report storing location, and also what was the Test Modules results. On Figure 4.23 there is an example of how this log file usually looks.

The Variant_CAPLTesting.zip is where the CANoe test reports generated during the build are stored (Figure 4.24). This file shall also be checked to understand if there are any problems with a Components' Test Module or not. These test reports are then used for consulting purposes, e.g., figuring out why a test has failed.

The work regarding Jenkins Automation is currently a work in progress, with much better performance and a improved way to run the tests.

Right now, the only tests not being performed using the Jenkins CI Automation are the ones regarding SPI communication testing due to the shortage of SPI Simulyzer HW available on site at the moment.

At the start of this dissertation, it was mentioned that work performed regarding Test Automation had to be documented and shared with the rest of the team. This has also been accomplished. A series of video tutorials (Figure 4.25) were made by the author during presentation of the tools and their functions. These video tutorials were then shared to the Team by e-mail and by storing the videos on one of the Teams shared folders.

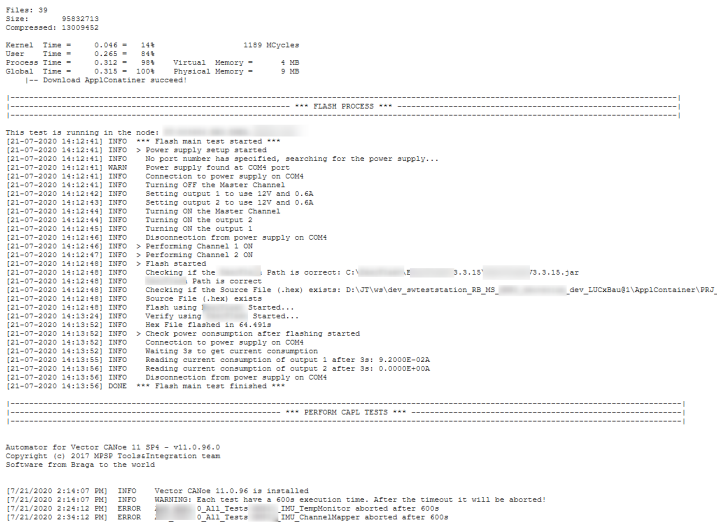


Figure 4.23: Jenkins Build Test Execution Log

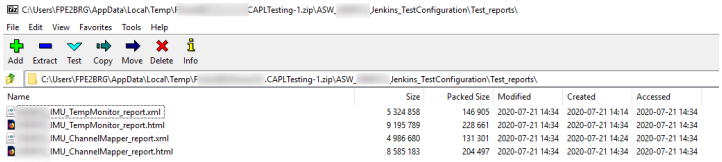


Figure 4.24: Jenkins Build Test Reports

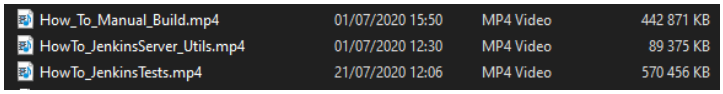


Figure 4.25: Jenkins Video Tutorials

PowerPoint presentations were also created, in order to exemplify and point out specific points on certain steps of the process. Figure 4.26 shows the step-by-step guide that was created and presented to the Team.

Also, in order to better help the team and the project's management team understand the state of the Test Automation, an overview was created and presented to all. Figure 4.27 shows the presentation given at that time.

In a way to help team members to better understand how the tests were made, and to help them out, a set of Docupedia pages were created. In these pages, a set of guidelines and coding rules are given following the Team Testing Process Guidelines, as seen in Figure 4.28

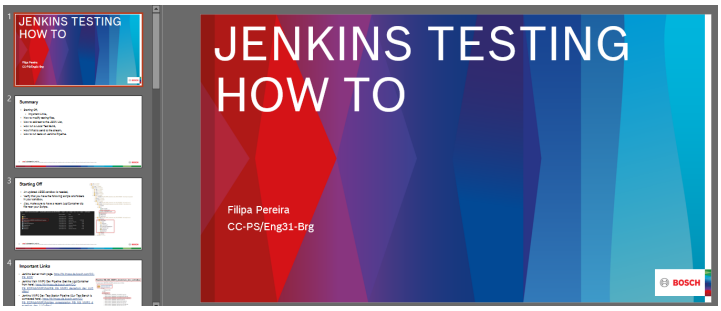


Figure 4.26: Jenkins Testing Step-by-Step Guide

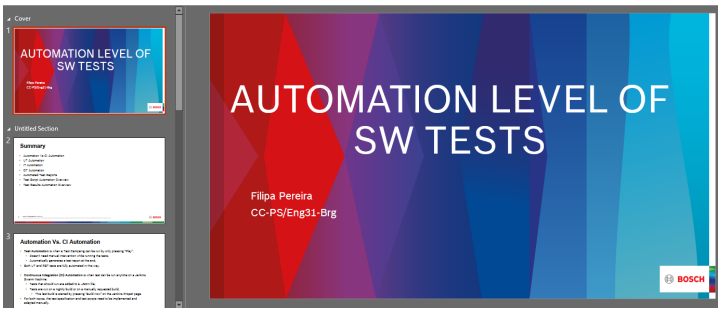


Figure 4.27: Automation Level of SW Tests

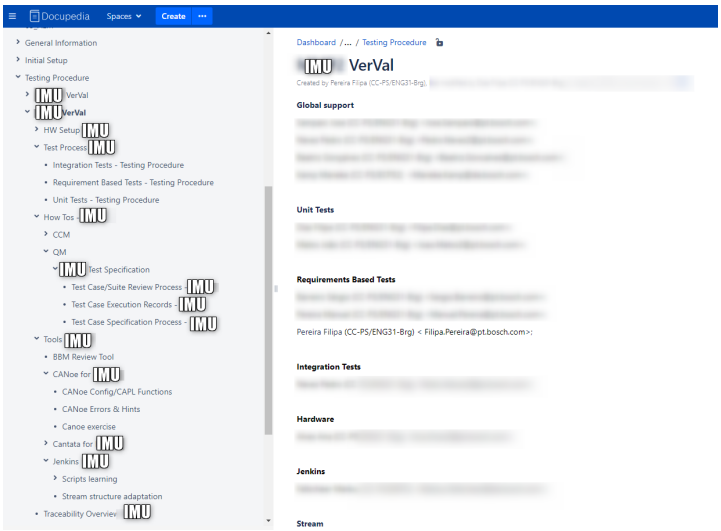


Figure 4.28: Docupedia Documentation About the IMU

4.3 Results Discussion

In general, the Automation Process of the SW Tests was well performed. The tests now run faster and are more accurate with their test reports and this has given the Team a morale boost as well as an increase in workflow and work quality. The SW tests are now also easier to read, which is a great help when someone has to adapt a test suite that was created by another tester.

In regards to the Jenkins Test Server, the plan to have it setup and start working with it was successfully performed. The Team can now select what tests are ready to be executed by the server, in each manual build request. There are still some issues regarding the usage of the Jenkins Test Server, but it is a work in progress that will be solved in due time.

The documentation created throughout this dissertation managed to greatly help the Team understand the new mechanics of SW Testing and support them in creating their how automated tests and have them run on the Jenkins Test Server.

Chapter 5

Conclusion

The goal of this Master's Thesis was to support the development and documentation of the IMU's VerVal team in regards to the Automation Process of the Qualification Tests performed on the Vector CANoe tool.

The Automation Process was divided into two steps: Automation of the existing Manual Tests and Jenkins Server Test Automation. The Automation of the Tests had per base the purpose of gaining knowledge regarding how the CANoe tool works and how the CAPL tests could be adapted to better perform the required tests with minimal user interference in order not to compromise the test results.

Fully automated tests were the most important step needed in order to get a Jenkins Server for testing the sensor. With this task completed, getting the Jenkins Test Server running was the next major step. The help given by the CI team from Bosch Germany, and also from the colleagues from Braga, was a major boost to get the server up and running with all the requirements that the VerVal team needed.

Initially, a small research was performed to better understand the current state of the Automotive Industry, and also to better understand what is the current state of knowledge regarding the Inertial Measurement Unit's sensor, its application uses on par with the car's Passive Safety sensor's group and how the Automotive Software Tests are viewed and performed. After that, a small presentation of the Testing Tools and Knowledge of the IMU's Verval team is given. In this section, the basic tools used by the Testing team like Vector CANoe and SPI Simulyzer are presented and explained, and also in conjunction with a overview of the various protocols implemented in the sensor and used during testing and development.

We then have the Team's Testing problems that range from the normal Testing Process, to how CANoe is used for testing and also the tool's testing language CAPL. CAPL plays a big role on how the tests are created and managed thorough all the team's test life cycle. The problems regarding the Manual Building and Flashing of the project and the sensor were also explained in detail.

With the problems already explained, the next chapter presents to the reader details about how everything was managed and corrected. The process of converting Manual Test into Automated Tests is explained in detail, with a big emphases on the NVM Mem-

ory Corruption Tests and the SPI Simulyzer Tests. The Automation Process wouldn't be finished without explaining how the implementation of Automated Tests on the Jenkins Server was solved. A list of Hardware, Software and Test related dependencies and solutions is described in this chapter. It then concludes with the End Results of a full Jenkins Automation process.

Most of the work related to this project is not present in this dissertation due to the fact that this Thesis is based on a Industry Working Environment Project, that involve a big group of people, and customers, knowledge and information that can't and shouldn't be shared before the actual sensor is able to reach the market.

Another type of work that couldn't be written in these dissertation was about the long hours passed by the team in training and seminars that would prepare and train each and every member to be the best SW tester they could be to better promote and refine the Quality and Excellency that is Bosch's stamp of guarantee to it's clients. The tester's know-how is not learned on an University environment but is gained while working and learning with others in the same position.

Tools like Vector CANoe and its inherited language CAPL are tools that are only utilized in specific industries and work environments, and knowledge regarding them can only be transmitted directly from Vector's workshops and trainings or from work colleagues that have already been to these workshops and have had time to apply and master the knowledge gained from them. Many other tools also utilized have this type of learning conditions, specific for each work situation, and couldn't be completely explained in the writing process of this dissertation.

Another important topic to explain here is the fact that the work being done is never finished, and this dissertation represents only a small part of all the knowledge gained since joining the IMU testing team.

5.1 Future Improvements and Comments

Improvements to the work mentioned here are already being made. From updating and better adapting the test scripts used, to getting better results from the Jenkins Server by means of new scripts. Due to the fact that new client's acquisition is a constant occurrence, there is a need to keep updating and perfecting the test scripts for each new client.

Another major improvement that would greatly benefit the Team would be the linking of ALM's Test Suites with the Jenkins Server. Each time a new release is implemented, the Jenkins Test Campaign results could be stored and linked to each Test Suite's test execution records for a better tracking and Quality of the SW. A next step for the Jenkins Test server would be the automatic generation of an overall test report, that would be kept updated for each test campaign that Jenkins executes successfully and that would show each component's test results. An accurate test result trend chart in a given build number timeline would also be a useful thing to have. This would provide the testing team a better overview of how the test results have been in a given set of time, with each build's information present. Differentiation between a Nightly build and a test run build is also included into the next steps list. With the Nightly build, a specific group of tests would be executed each night and in the morning the testing team

would know if something capable of breaking the tests was implemented or not. A test run build would be quite similar to what is currently happening in the Jenkins Server, where the under-development tests are executed whenever a tester requests a build to be run. Also to help out on the Nightly build and test run build implementation, a test status list should be created. In this list the maturity level of a given test would be differentiated between a "finished test" and a "under-development test". With the completion of this list, the test modules to be executed on a Nightly build could easily be assigned.

Another thing that was left for future improvements was the use of Robot Framework to better enable the automation process of tests. This was initially part of the planned work for this thesis topics, but had to be left out due to the fact that the tool wasn't yet approved internally within Bosch, and there was no way of getting support for it in case something went wrong. From what is already known, the tool is already in a validation process to be certified and verified for use by Bosch's developers.

References

[1] R. Zalman, "Rugged autonomous vehicles." [Quoted on p. 5, 6]

[2] A. Mjeda, "Standard-compliant testing for safety-related automotive software." [Quoted on p. 6]

[3] J. Zander-Nowicka, "Model-based testing of real-time embedded systems in the automotive domain." [Quoted on p. 7]

[4] F. Lonetti and E. Marchetti, *Advances in Computers*. [Quoted on p. 7]

[5] Z. Ma, W. Seböck, B. Pospisil, C. Schmittner, and T. Gruber, "Security and privacy in the automotive domain: A technical and social analysis." [Quoted on p. 7]

[6] What is an IMU sensor? [Online]. Available: https://www.ceva-dsp.com/ourblog/what-is-an-imu-sensor/ [Quoted on p. 8]

[7] What is IMU? inertial measurement unit working and applications. [Online]. Available: https://www.arrow.com/en/research-and-events/articles/imu-principles-and-applications [Quoted on p. 8]

[8] Sensors ibusiness. [Online]. Available: https://www.bosch-ibusiness.com/media/images/products/sensors/xx_pdfs_1/sensors_i-business.pdf [Quoted on p. 8]

[9] D. Mackenzie, "Inventing accuracy." [Quoted on p. 9]

[10] R. Bicking, *Automotive Accelerometers for vehicle ride and comfort*. [Quoted on p. 9]

[11] S. R. Fiorentin, *Sensors in Automobile applications*. [Quoted on p. 9]

[12] Accelerometer and gyroscopes sensors: Operation, sensing and applicacitons. [Online]. Available: https://pdfserv.maximintegrated.com/en/an/AN5830.pdf [Quoted on p. 9]

[13] A. A. Santiago, "Extended kalman filtering applied to a full accelerometer strapdown inertial measurement unit." [Quoted on p. 9]

[14] Motorcycle inertial measurement unit explained. [Online]. Available: https://www.bikesmedia.in/reviews/motorcycle-inertial-measurement-unit-imu-explained.html [Quoted on p. 10, 11]

[15] A. Org. AUTOSAR - the standardized software framework for intelligent mobility. [Online]. Available: https://www.autosar.org/ [Quoted on p. 12]

[16] ——. AUTOSAR - introduction. [Online]. Available: https://www.autosar.org/fileadmin/ABOUT/AUTOSAR_EXP_Introduction.pdf [Quoted on p. 12, 13]

[17] Autonom. AUTOSAR fundamentals. [Online]. Available: https://www.autonomtech.com.tr/en/autosar-fundamentals/ [Quoted on p. 13, 15, 16]

[18] ISO 26262. [Online]. Available: https://www.qa-systems.com/solutions/iso-26262/ [Quoted on p. 16, 23]

[19] ISO 25010. [Online]. Available: https://www.iso.org/standard/35733.html [Quoted on p. 17]

[20] I. Synopsys. What is ASIL? [Online]. Available: https://www.synopsys.com/automotive/what-is-asil.html [Quoted on p. 17, 18]

[21] How to determine the asil value for an automotive application, as per iso 26262 standard. [Online]. Available: https://www.embitel.com/blog/embedded-blog/understanding-how-iso-26262-asil-is-determined-for-automotive-applications [Quoted on p. 18]

[22] What is ASPICE? [Online]. Available: https://beza1e1.tuxen.de/aspice.html [Quoted on p. 19, 21]

[23] V-model and w-model software testing. [Online]. Available: https://www.testbytes.net/blog/v-model-and-w-model-software-testing/ [Quoted on p. 20]

[24] Dijkstra, *E. W.: Notes on Structured Programming*. [Quoted on p. 22]

[25] Evolution of automotive software testing. [Online]. Available: https://www.embedded.com/evolution-of-automotive-software-testing/ [Quoted on p. 22]

[26] J. Takahasi. Challenges in automotive testing. [Online]. Available: https://www.logigear.com/magazine/test-automation/challenges-in-automotive-testing/ [Quoted on p. 22]

[27] ISTQB. [Online]. Available: https://www.istqb.org/ [Quoted on p. 24]

[28] *Certified Tester - Foundation Level Syllabus*. [Quoted on p. 25, 27, 28]

[29] What is application lifecycle management. [Online]. Available: https://stackify.com/application-lifecycle-management/ [Quoted on p. 29]

[30] Cloud application manager. [Online]. Available: https://www.ctl.io/cloud-application-manager/application-lifecycle-management [Quoted on p. 29]

[31] *Vector Trainings - Confidential Trainings for Companies*. [Quoted on p. 31, 32, 33, 34, 46, 47, 48, 49]

[32] CAN fundamentals - vector e-learning. [Online]. Available: https://elearning.vector.com/mod/page/view.php?id=335 [Quoted on p. 30, 35]

[33] UDS protocol. [Online]. Available: https://embedclogic.com/uds-protocol/ [Quoted on p. 36, 37]

[34] XCP protocol. [Online]. Available: https://piembsystech.com/xcp-protocol/ [Quoted on p. 38, 39, 40, 42, 43, 44]

[35] SPI communication. [Online]. Available: https://embedclogic.com/serial-peripheral-interface/ [Quoted on p. 45, 46]

[36] Vector. [Online]. Available: https://en.wikipedia.org/wiki/Vector_Informatik [Quoted on p. 45]

[37] U. Kumar and K. K, "CANoe tool for ECU automated communication testing." [Quoted on p. 45]

[38] CAN bus. [Online]. Available: https://en.wikipedia.org/wiki/Controller_Area_Network [Quoted on p. 45]

[39] Local interconnect network. [Online]. Available: https://en.wikipedia.org/wiki/Local_Interconnect_Network [Quoted on p. 45]

[40] Flexray. [Online]. Available: https://en.wikipedia.org/wiki/FlexRay [Quoted on p. 45]

[41] Ethernet. [Online]. Available: https://en.wikipedia.org/wiki/Ethernet [Quoted on p. 45]

[42] MOST bus. [Online]. Available: https://en.wikipedia.org/wiki/MOST_Bus [Quoted on p. 45]

[43] SPI Simulyzer. [Online]. Available: https://www.seskion.de/en/products/seskion-spi-simulyzer/ [Quoted on p. 49, 50]

[44] Renesas Flash Programmer. [Online]. Available: https://www.renesas.com/tw/en/products/software-tools/tools/programmer/renesas-flash-programmer-programming-gui.html [Quoted on p. 50]

[45] Continuous integration essentials. [Online]. Available: https://codeship.com/continuous-integration-essentials [Quoted on p. 51]

[46] Continuous integration. [Online]. Available: https://www.thoughtworks.com/continuous-integration [Quoted on p. 51]

[47] How to devops? part 3: Continuous integration. [Online]. Available: https://softwarethatmattersdoneright.com/wp/how-to-devops-part-3-continuous-integration-ci/ [Quoted on p. 51]

[48] Jenkins. [Online]. Available: https://jenkins.io/ [Quoted on p. 52, 53]

[49] What is Jenkins? The CI server Explained. [Online]. Available: https://www.infoworld.com/article/3239666/what-is-jenkins-the-ci-server-explained.html [Quoted on p. 52]