POLITECNICO
DI TORINO

Safety Gaps filling of a Partial Networking Coordinator, developed in AUTOSAR Application Layer, as Safety Element out-of-Context

Giovanni Pinti
s261005@studenti.polito.it,

Politecnico di Torino
Computer Engineering - Embedded Systems
a.a. 2020-2021

Supervisor: Massimo Violante, DAUIN-Politecnico di Torino

ETAS
DRIVING EMBEDDED EXCELLENCE

I warmest thanks to:

My parents and my brother

My grandmothers

My grandfathers Vittorio and Giovanni, my uncle Mimmo and my aunt Silvia (in memoriam)

My uncles, aunts, my cousins and their partners and children

Mary

All my friends, colleagues, team-mates and home-mates

Professor Violante for the precious support

ETAS guys for the precious support

Thanks a lot to everyone

Contents

List of Tables

List of Figures

Abstract

Automotive Industry is a field where Embedded Systems are widely employed. Modern vehicles can contain up to 80 microprocessor-based systems, called Electronic Control Unit or ECU, they run millions of Lines of Code and they can represent the up to 80% of vehicle innovation. An important aspect, in terms of innovation, is the power saving. Some studies have proved that the power saving can have a positive impact on several parameters like the emissions: reducing the power consumption is possible to reduce the emissions by a Mathematical formula. One of solution to implement the power saving at vehicle context is to adopt the Partial Networking at communication level among ECUs.

The design and development of an Embedded System for a vehicle shall be done considering the safety-critical nature of Automotive products and so they shall meet several regulations and normative to ensure the State-of-Art. ETAS GmbH (Bosch Group) is an AUTOSAR premium partner it develops several application, tools and services for Automotive solutions. On of these is a Partial Networking Coordinator it manages the Basic Software Services as a central node in application layer, for each application software component that is mapped in Partial Networks, using a CAN bus system. The nature of the software component forces to identify it as a Safety Element out-of-Context (SEooC) and therefore its design and development, compliant to ISO 26262, requires a certain level of tailoring in safety activities. The following paper will analyse all aspect related to ISO 26262 for SEooC development and it will try to produce a solution in order to get compliance of ETAS Partial Networking Coordinator with ISO 26262 in all its parts that are considered in scope, using language subset as MISRA C to improve the safety too.

Chapter 1

Introduction

1.1 Computer Systems application in Automotive and consecutive problems

As early as 1983 a *GM* engineer wrote in an article about the "Transactions on Industrial Electronics" that software would represented the most important aspect, within new development engineering products [2]. Some years ago, an expert professor at Munich University has estimated that 80% of a car innovation is done by on-board computer and their development can arrive to be worth until the 50% of the total car cost for conventional vehicle, and up to 80% for the electric and hybrid ones (up now).
Nowadays, almost the totality of road vehicles are equipped over to 70 microprocessor-based Electronic Control Units (ECU) they run Embedded SW like RTOS (Real-Time Operating Systems). In 2009 and 2010 Several Researches ([2]) reported that "probably", the embedded software in a generic car has been developed over 100 millions of LoC (Line of Code), going near 200 ones in the next future, versus 5.7 millions LoC of a F-22 Joint Strike Fighter (scheduled for 2010). The size of the LoC in a vehicle probably does not always reach that value but, considering the large number of ECU within a vehicle context, it is a reasonable value; Over the years, even more systems in a vehicle will be under the Software Control. It is not so rash to assert that in a next future (or already now) there will be a Software routine for each vehicle component with a consistent increasing of LOC size and so on, software complexity.
For example the Brake Control Module (often identified by acronym *BCM*) is one of the most diffused Software controlled-system in car. This module can also embed more subsystems, like ABS (Anti-Blocking System) and ESC(Electronic Stability Control). Specifically, *ABS* is a system that prevents wheels blocking in the case of an immediate braking, activating an air-pumping mechanisms, to transform a continuous long braking into a series of short ones (impulsive). This solution avoids loss of vehicle controllability and it reduces the risk of crashes. From a functional point of view the embedded software shall harvest much information as: actual wheel speed, braking entity and grip status; all these allow to evaluate car situation and take a decision about anti-blocking activation. Only using logic reason, it is possible to imagine that software shall access to several speed sensors in a well-defined time; for sure, more than one measure shall be considered, maybe from more types of sensor. Once this has been checked, it is necessary to actuate the software decision by mechatronic systems as an air pump with valves manager. The entire process (from

data collection to physical actuation, passing for evaluation phase) shall be done in a short period (range of tenths milliseconds); to have a more concrete look to a real case: In some cases, the wheel blocking can happen just in front or rear wheels. This fact can involve itself in a fatal situation for motorcycles, due to the high probability of a fall down for the loss of controllability. To avoid similar situations it is possible to combine ABS with CBS system. This last is not an electronic system but it uses the braking force that is actuated on front wheel, to active an adjusting pressure pump, forwarding braking force on the rear wheel; this system combines an non-electronic solution with an electronic one, increasing the complexity of the control system that shall manage a large amount of parameters and algorithms to take the right decision.

The overall high complexity of the previous system, especially on software view, is rewarded in decreasing of car crashes, economic damages, injuries and fatalities. Since 2014-2015, several global researches have shown how the ABS, integrated in motorcycles (over 125cc) has reduced the injury crashes of 29% and the fatality ones of 34%, reducing the overall crashes numbers where motorcycles were involved of 22%(Data about Spain w.r.t. the previous years). In Italy data are very similar with a decrease of injury crashes of 24% with an overall reduction of 27% [3]. The importance of these data is also greater considering the global unit production of vehicle (110 millions of new units per year will be reached from 2025 [1]).

On the base of previous consideration, the real problem is **to ensure that an embedded systems implements the right safety mechanisms in the right ways**
Automotive software products are characterized by a high complexity, a big amount of LoC and a big number of *use-cases*. Therefore, ensuring that in millions LOC that describes routines there are not misbehaviours (or potential cases for), that might involve in Hardware faults and then in software failures, is far from to be simple: When a design is able to prove this, then the State-of-Art is reached. Such evidences can be provided only: **sticking very strictly to several rules and requirements** as indicated in **specific standards and regulations** that will be exposed later.

1.2 Road vehicles Functional Safety

1.2.1 Safety Critical Systems

Computer systems within a vehicle are considered Safety-critical ones i.e. very sensible to errors occurrences; each of them can potentially cause human casualties. Therefore software complexity becomes an essential features and it can be managed by requirements engineering models and model-based design. This does not mean that only software is affected by defects: ECU is a microprocessor-based system that is built by hardware elements and software components; so a defect might impact both the dominions with different effects and causes: In a Microcontroller Unit there will be hardware parts that are subjected to failure over the ageing while software does not; At the same time it is not possible to consider software behaviour as linear: if a functionality is asserted with an input pattern, then it will work forever with that but the same functionality might not work with all input patterns it should!
First common "defending line" can be the software testing, to discover errors. When a

software is developed, the possibility to test it shall be always feasible. However, its complexity might not allow an exhaustive testing phase i.e. applying the entire set of possible input combinations; an academic example is to consider for testing 2 inputs both on 32 bits: a complete test requires 2^{64} patterns (more than the cube of 10^6) that is not feasible. So the software test shall be done with a subsequence of patterns "enough exhaustive" to reach the right safety level.
Note that by a safety point of view, it does not matter very strictly if a functionality meets its requirement; that is reliability aspect. The real focus of safety is: to avoid the possibility that an unintended behaviour can increase the risk of failure. In other words a component can fail as long as its failure is handled and system is able to accordingly react, remaining in a safe state.

1.2.2 ISO 26262 - Road Vehicle Functional Safety

ISO 26262 is a standard normative for Automotive Industries. It drives the entire designing process for products that are built by hardware and software together. Process is divided in several (and mostly *mandatory*) activities they defines the safety measures. These will be evaluated and applied in order to obtain the functional safety. The last version has been released in 2018 with the extension to all road vehicles including the motorcycles.
The main scope of ISO 26262 is to standardize:

A) the design process for "fault prevention" and

B) the development activities and methods for fault detection and mitigation;

Its activities are described within 12 parts (or chapters) for second release, where anyone focuses itself on a specific part (planning, Requirements, development, etcetera...). Each chapters is divided in sections (clauses) that in turn contains ISO requirements. At the end of each clause there is a section that specifies which work products shall be produced to satisfy clauses' requirements and so the compliance to ISO 26262. The set of the whole work products that have been developed defines the **Safety Case**.

Safety can be defined as an intrinsic property of a safe product. An exhaustive example can be the following:

> Let's consider a crossroad: there is a reasonable risk of vehicles crash therefore the environment cannot be considered as Safe without a mechanism that is able to reduce as much as possible this risk: the safety level will be obtained with respect to functionality of this system (in the specific case it can be a traffic light); in summary: the safety will depend by the correct operating of the safety system to its input stimuli. When it will work correctly then the probability of risk is reduced until a *"reasonable level"*. [4]

For ISO 26262 "risk" is intended as probability of damage occurrence; greater this risk will be, better the design of safety mechanism will have to be. The source of damage is called *Hazard* and it is produced by malfunctions what a time are dependent from Failures(missing capacity of an element or more to fulfil its/their tasks). Failures can be

caused by Physical defects called faults. Measure of risk is done according to definition of *ASIL(Automotive Safety Integrity Level)* that is defined in the scope of the normative. ASIL measure is in a domain of four values: A (min), B, C, D (max); they are obtained by a matrix it defines them as the result of crossing between three others parameters:

- **Severity(*S*)**: entity of harm (injury on physical person);
  - *minimum S0:* No injuries;
  - *Maximum S3:* survival uncertain or impossible;
- **Controllability(*C*)**: possibility to avoid damage or harm by timing reaction of person;
  - *minimum C0*: In general Controllable
  - *Maximum C3:* Uncontrollable
- **Exposure(*E*)**: probability where a failure involves in an Hazard;
  - *minimum E0:* Incredible;
  - *Maximum E4:* High probability;

This measure will be directly assigned to the source of damage i.e. Hazard; According to it, the design will implement the safety functions to decrease it until the level of QM(reasonable risk).
A risk analysis to assign ASIL value can be formulated according with this mathematical model:

$$Risk(S, \mathbf{P}(C, E)) < Risk_{Th}$$

where S, E and C are described in the previous, while $\mathbf{P}$ is the probability function of risk occurrence and $Risk_{Th}$ is the threshold of tolerable risk (in the following Reasonable risk and tolerable risk are used as synonymous).
In the graph plot **Reasonable Risk Curve[1.1]**, the blue line represent the Tolerable risks. Let's consider Risk is exactly the Tolerable one as constant: this means that if $Risk(S, \mathbf{P}(C, E)) = const.$ Now let's consider again Risk function like the product of S and $\mathbf{P}(C, E)$ (inverse relationship between them). The target is to implement mechanism it can move the result of the relationship among S, C and E values on tolerable risk curve or in the below region (it is reasonable risk region); this means that if the S value is short, then a greater probability of Hazard can be accepted; by other hand, if the Severity is very high (Values from 7 to 10 can represents *S3*) the probability of hazard shall be very low (this means low loss of controllability and low probability of exposure). This consideration shall be done because sometimes, the determination of ASIL value has a very high impact on the production cost too.

By a very simple point of view, the ISO 26262 activities might be grouped into four macro phases:

- Planning and Concept phase;

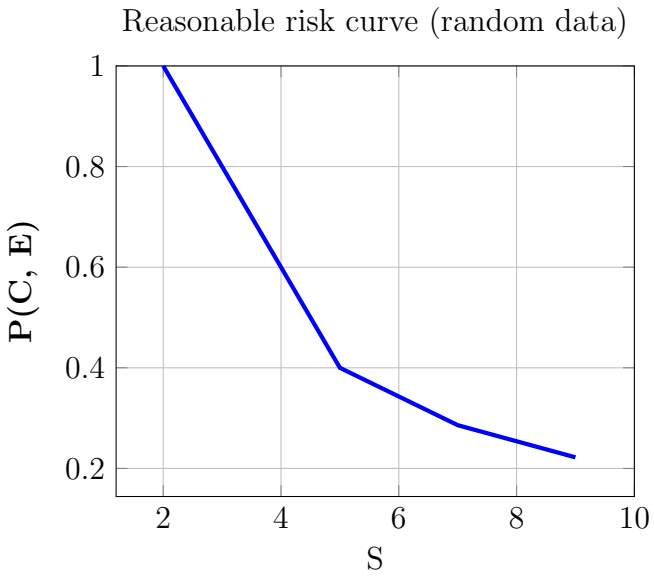


Figure 1.1: Reasonable risk curve

- Development at several product levels;
- Production, Service and decommissioning;
- guidelines, specific cases and supporting activities;

The overall design approach will start from a pure functional description at high-level, for going down to more detailed low-levels where activities will be strongly technical. Each software tool that will be used as supporter, it shall be qualified by ISO certification.
Planning and Concept phase can be considered as a Requirement Engineering one. The planning phase consists in the development of the safety plan, i.e. a document that will contain all aspects related to the team that is developing a product (roles, responsibilities, tasks) and planning of safety activities that will be adopted. Concept phase starts defining a top level entity that is called *item*; its formal definition can be: *"a combination of systems (Hardware and Software) they are designed to meet specific functionalities within a vehicle context"*. This definition shall be done by a purely functional description. Then these will be analysed in order to discover possible Hazards thanks to the *Hazard Analysis and Risk Assessment or HARA*. Its result aims to define the *Safety Goals* i.e. the target to achieve in order to reduce the risk that Hazards occurrence; at the same time Hazard occurrence containment shall be considered (ISO 26262 adopts prevention and impediment). Safety goals will have a double utility:

- Identify the ASIL value;
- Define the Functional Safety Requirements *(FSR)*;

These last describe what shall be done to achieve safety goal. Their description is still functional, in fact they do not consider any implementing technology. The next phase consists in the allocation of these in a "primordial" system architecture by their formulation in by technological point of view.

product development at system level starts to analyse technical concepts. It considers several aspects, as the separation between Hardware and Software tasks they will developed in different processes (in the most in parallel) and their meeting point (Hardware-Software Interfacing). The Functional Safety Requirements allocation defines the Technical Safety Requirements *(TSR)* and, it will complete the "Safety Concept". As mentioned, the product development at system level shall consider the separation between hardware and Software, so these TSR will be forwarded to the correct domain, for next development phase (in the case of Software they will take the name of *SSR - Software Safety Requirements*). The system architecture design shall include also its *verification phase*. the definition of TSR in System product and its Verification phase defines the upper vertex of the V-model that ISO 26262 defines for the development phase of lower level (Components and Units).
Development phase can be split in Hardware Development and Software one. This last case is directly considered in this paper. The goal is to implement what has been described in the concept phase in software solutions. ISO 26262 defines a specific chapter (6) for the development at software level. The chapter contains several methods for the implementation of a software product in hierarchical way, covering also some aspects about the correctness of modelling for model-based software design and the choice of programming language. Actually the most of Automotive software products are developed in C, C++ and their derivations. Chapter actually explains all nodes they build the V-model for development processes. The V can be divided into two symmetrical parts by a middle vertical line: the left side will represent the implementation of requirements derived by concept phase, while the second (right side) will be the verification by testing of the implementation. In the implementing process of V-model, a series of methods are explained for each clauses (V nodes): in general they cover implementation and then a verification. Methods are contained within tables with a specific level of recommendation for each ASIL that have been identified and they can be applied sequentially or alternatively, according to their identification (number addresses sequentially, different characters for the same number the alternative). If the verification of a process fails, then it requires to fix the actual work products or to trace back to higher hierarchical level and fix it. The process will explore the entire hierarchy of a software product until the software units, i.e. the elementary building of a components. The Unit design and implementation will represent the left base of the V-model. The respective activity on the right branch will be the unit testing. From this one, the tracing back on V starts. Each node will test the respective level on the left branch. If testing of one hierarchical level fails, the process will return to the respective level of the implementation branch; then it will be necessary to perform again all activities for each node of implementation, starting for reached one until the right vertex will not be reached. When the top right node of V-model is reached, the development of the software product can be considered as complete and so all work products can be included in safety case.
Production, Services and Decommissioning: The main scopes of these activities are to provide: a production process, including the maintainability of each safety-related elements or items and the necessary informations for users. The focus are the product parameters; they are divided into two categories: process ones and configuration. Briefly this phase explains how a good user manuals can be drafted on the base of development environment for Hardware, software and product development at system level. In some cases, the user manual shall be considered together to Safety Manual. This work product is well-explained in IEC 61508 and it has been adapted by ISO 26262 in order to meet special field requirements but maintaining the main features of IEC requirements. The

scope of the safety manual is to organize all identified safety requirements from concept phase or similar, to report all result about: safety analysis, safety mechanisms, and diagnostic aspects. In other words, it should specify the product features in terms of safety that have been developed in compliance with ISO specification. Besides it considers all aspect related to the production of a configurable product as calibration parameters and measures, in order to grant the right developed and functional safety assessment. One of these measures are related to the traceability that will be deeply analysed as active part of the work in the next chapters. Last goal of current phase is to evaluate the impact of changes on production flow in terms of technical aspects and employing time.

Supporting processes: The supporting processes contain useful information in terms of definition and management of ISO work products. One of these processes reports the mandatory informations that shall be contained in the documentation and how an exhaustive and complete one can be drafted. Another one explains how requirements should be defined and documented, including the mandatory attributes of them. At the scope of Safety Element out-of-Context it explains the change management in the case of missing qualification from integrators i.e. validation of assumptions fails and it is necessary to develop again, coming back to a concept phase. Verification reports and other information about tool qualification are also considered. The main ISO 26262 chapter that considered supporting activities is the number 8. Chapter 9 can be integrated in this macro aspect too: it contains all information about the analysis that can be done for obtaining evidence in favour of safety achievement or to identify possible hazards or weakness. At the end, a specific section is related to how it is possible to perform an ASIL decomposition within a item or an element.

Safety Element out-of-Context (SEooC) The definition of the SEooC is contained in ISO 26262 version 2018, chapter 10 clause 9. This one is entirely dedicated to the analysis of this element. The necessity to define a SEooC is strongly related to the heterogeneity of Automotive Industry: all the previous description in the current introduction of ISO 26262 considers the development of an item; the ISO activities usually are applied on an item i.e. a set of systems that works together at scope to solve a specific application well-defined within a vehicle context (e.g. ABS, Air-Bags, Parking Assistant, Light-Wiper automatic controller, etcetera). In the most of the cases the development of an item is strictly bond to a specific vehicles gamma. A product can be also developed to meet specific functionalities without a complete knowledge about its context in a vehicle, as for example others systems or components with which it will interact. This kind of element is developed once and potentially used in more vehicle contexts without a complete (or even without) redefinition for each one of them. At the same time, it shall meet safety aspects for road vehicle and so ISO 26262 requirements. In this case the product can be developed as a generic element (System, Component or Unit) called Safety Element out-of-Context. Due to its nature, it cannot be developed applying the whole safety life-cycle, how it has been defined by ISO 26262 for item development, and therefore it requires a dynamic level of tailoring in the safety activities. ISO 26262 explains which activities can be tailored for its development, it provides guidelines about how they can be tailored and it explains how the functional safety assessment can be done and especially *who shall do what*. The tailored activities and how they have been conduced for the Partial Networking Coordinator (element that has been developed at the scope of this paper) are well-detailed in the next chapters.

1.3 MISRA C

MISRA(Motor Industry Software Reliability Association) drives across the entire process of software production. It extends some standard requirements to ensure a high quality, in terms of safety, beyond the ordinary one. Software design can have large flexibility and capability to be changed and adapted without manufacturing variation. At the same time there is not possibility it involves in errors that are caused by ageing; in addition, Software products can manage very complex aspects, abstracting the physical components in lines of code (many and many).
The development of embedded software is always done for special purpose. This means, it shall meet specific requirements and quality aspects. In most of cases it shall reach the state-of-art. Product goal can be the optimization of computation, the power consumption or the information security. Embedded software that is developed for automotive industry has four main emphasis aspect:

- Data;
- Parameters;
- Adaptive controls;
- Diagnostic aspects;

Data are processed according to specific algorithms. In a safety-critical system, as vehicle one, it is important to ensure a continuous stream processing. Data are collected by sensors, computed by processor and then used to control actuators. This means that a wide amount of data of several nature shall be managed. In a software layered architecture as AUTOSAR, data types are defined according to the specific layers. Flexibility of software routine shall be ensured, to allow a big variety of data handling without specialized code for each of them.

Parameters : The high complexity of software is managed using model-based designing approaches. This means that systems are described using abstracted models they are able to capture system functionalities. This one can be described as the capability to produce certain outputs value at specific input value application. The inputs set is built by parameters and they shall be combined in order to obtain the maximization or minimization.

Adaptive Controls : the laws are subject to change with high frequency. Sometimes they can involve important upheavals in a design aspect. The normative on CO_2 emission is a very powerful example: the reduction of emission is a complex aspect that impacts on several components and processes, from mechatronic actuation until software controllers. At the same time during a single vehicle trip, the health status of each component can consider large scenarios as fuel level, battery state-of-charges and sensor ageing, until road surface status and weather conditions. Due to this is necessary to develop embedded software in order to get it adaptable to several dynamic condition.

Diagnostic the health status of vehicle system improves very much the safety conditions. Embedded software is forced to ensure the continuous system functionality also in presence of faults during a vehicle life-cycle; the reasonable risk level is not obtained only reducing fault occurrence but also preventing its possibility to involve into hazards. Due to this is necessary to design an on-boarding diagnostic system (hardware and/or software) to harvest data, ensure data integrity and operate in case of malfunction.

Software are developed by specific programming languages and each of them has its own features starting from type: Compiler one (C, C++, ...) or interpretive (Java, JS, Python, ...). A Standardized integrity level, called SIL (Software Integrity Level) is defined for the evaluation of the controllability that is measured on four levels: less controllability means less acceptable failure rate until SIL 0 for Quality Systems (ISO 9001) [8]. SIL level will filter the features of language and compiler to get compliance and the requiring testing level. MISRA drives the software processes according to specific language that is used for software development and in specific case for C with emphasis on language weaknesses:

- Language Syntax and type checking;
- runtime error handling;
- formatting constraints;

The focusing concept is: *"if something can happen then it will"*. Therefore MISRA C focuses itself to avoid the reliance to "it does not happen anyway", managing as much possible situation. At the same time, MISRA aims to meet developer needs for specific cases cataloguing its requirements as Required or Advisory.

1.4 AUTOSAR

1.4.1 General Overview

AUT*omotive* ***O****pen* ***S****ystem* ***AR****chitecture* or AUTOSAR was born as worldwide development partnership, among OEM, TIER and Service Providers, to standardize Electronic Control Unit Software architecture, at the beginning of 21^{TH} century. Electronic Control Unit or ECU are microprocessor-based systems they are able to run embedded software as Real-time Operating Systems (RTOS). The main goal of this standard is to improve **re-usability** and **ex-changeability** of software modules, avoiding troubles among product Stakeholders. Besides the adoption of a worldwide standards has reduced the Time-To-Market and the relative Software development costs. The re-usability problem is solved by increasing software abstraction from hardware platforms; this allows to build the AR structure as a software layered architecture similar to embedded operating system stacks. AR layers can be divided into 3 types: Application Layer (Highest), Run-Time Environment (Middle), Basic Software Layer (Lowest, above Hardware Platform). This last one can be divided in turn into three others sub-layers plus a global one: Basic Software Service layer (highest), ECU Abstraction layer (middle), Microcontroller Abstraction Layer (lowest); the global one takes the entire basic software layer without an additional sub-layering and

it is called Complex Drivers.
AR introduces an own workflow it provides specific work products as it is explained in

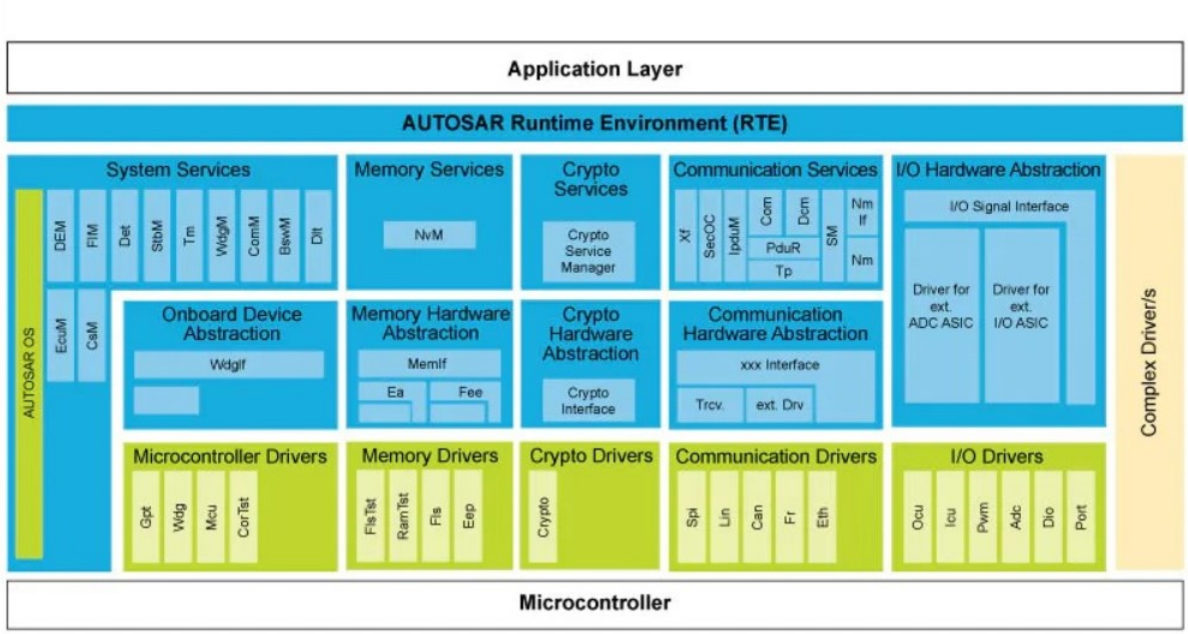


Figure 1.2: AR layered architecture by NXP

the following. AR workflow aims to operate on three milestones:

- ARCHITECTURE: it is the structural platform for hardware and software application;
- METHODOLOGY: it defines the data exchange formats and the templates for the configurations;
- APPLICATION AND INTERFACES: it defines the syntax and the semantics for ECU software and its dependencies.

The work products that are derived by this workflow, can be grouped in five topics:

1. Abstraction of System Design;
2. Virtual Functional Bus or VFB;
3. Software Component Description and system design;
4. Basic Software Modules Configuration and Integration;

When all work products are obtained, exactly in the previous order, it will be possible to obtain code (C or C++) that can be integrated within an ECU as runnable program.
Data exchange is a very important aspect: AR specification defines an own schema it allows to define and to describe (or configure): system constraints, Software Components

and ECU resources as Artefacts in *XML* files, with the special format of *ARXML*. An artefact is the tangible form in which work products are provided. The AR-schema is in relationship with the AR Meta-model that is the UML, by class-definition, using xml. The mandatory works products for AUTOSAR design are:

Abstraction design This work product defines a functions-based architecture, expressing the vehicle functionalities in an abstracting view. The mapping of ECU functionalities is not necessary in this phase. The key point is to define each application and to identify communication among us. The abstraction design is contained in Abstract System Description artefact;

Virtual Functional Bus and software component description When abstracting design is done, the application are mapped within atomic unit called Application Software Components (Application layer). Their description is documented by AR specification. Each software component is described in an artefact according to AR specification and they are built by:

- Ports: they implement interfaces. These last are described in arxml artefacts, according to AR specification and they shall be related to data types which interfaces can be fed. AR ports have:
  - Types: Sender-Receiver, Client-Server, Switch-Mode, ecc..;
  - Direction: P(out), R(in);
- Internal Behaviour: it defines the interaction between Software Component and RTE and it contains
  - runnable entities or C functions;
  - Events;
  - Data access definition;
  - Exclusive Areas

The RTE will use the software component descriptions to implement the virtual communication. This phase is important for the generation of software Component API in header (.h) files. The software component PI will contain function prototypes, constant and data structure will be used by software component source code

System Description Once the description of software component(s) is complete it is possible to design the system. A system is a set of more components they communicate among them in a Request-Provide way. To describe a system is then necessary to have:

A) A complete software components descriptions;

B) A complete network topology i.e.:

   b.1) topology between ECUs;

   b.2) software components mapping into the relative ECUs

C) The definition of a communication matrix in terms of active communication nodes, source and drain and typologies of CAN PDUs

This phase will produce a scheme called *SystemDescription.arxml*

Basic Software Component Description The basic software description consists in the configuration of basic software modules 1.2. Basic Software Modules are able to configure the Hardware resources for obtaining a dynamic code. The ECU abstraction layer can be considered as a "Hardware Interfaces" provider from Service layer while, Microcontroller Abstraction Layer has the scope to define the drivers of the hardware platform. This because AUTOSAR considers the ECU as figure 1.3.

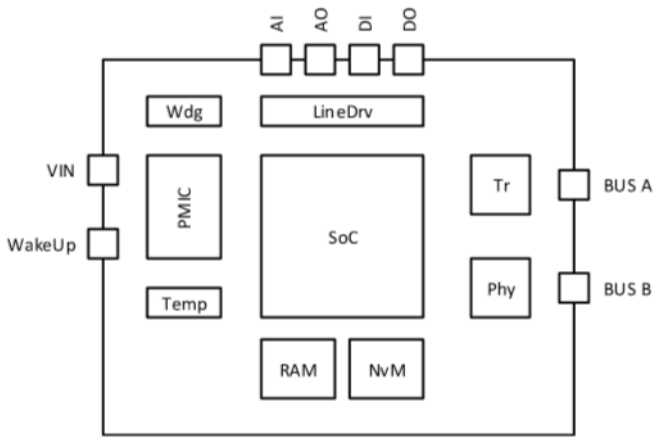


Figure 1.3: ECU autosar block diagram (logical architecture)

Instead Microcontroller is defined as a single device embeds several resources (look figure 1.4). AR allows the configurations of hardware platform, specifying APIs for each module.

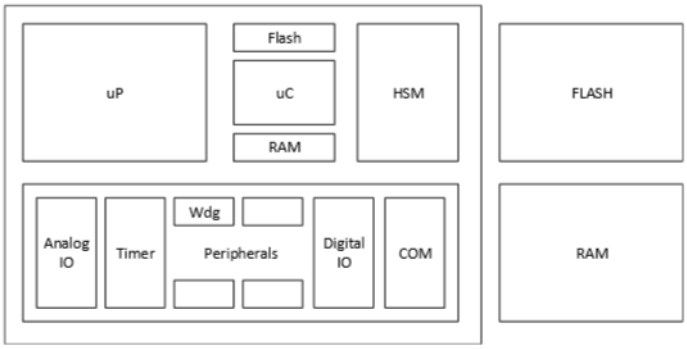


Figure 1.4: MCU autosar block diagram (logical architecture)

RTE code is finally generated by AUTOSAR Authoritative Tools as *ISOLAR* or *DaVinci*

When RTE code or dynamic code is generated, it is done for a single ECU it maps several application software Components. System description contains all software Components for all ECUs and so it is necessary to obtain another system description (actually a subsystem) it contains only Software Components that shall be mapped within a specific ECU or a group. Therefore, to generate a specific RTE code it is not necessary to provide the whole system description. A single ECU view is obtained by the ECU Extract description or configuration (the two terms are exchangeable) from overall System Configuration. It just consists in an arxml scheme that can be used by AAT. The ECU extract description has the property to be *flat* i.e. it does not contain nested software compositions. To sum-

marize the generation of dynamic code process for an ECU, the following list reports all steps that are necessary:

1. Generate overall System description/configuration;

   1.A If necessary, the extract system description is generated and provided (*Virtual View*);

2. Extract ECU description from overall system or an extract from virtual view, with SW-C instances, to obtain a *Flatted View*;

3. Generate RTE dynamic code for the specific ECU;

1.4.2 Functional safety related AUTOSAR parts

To complete the introduction to AUTOSAR it is necessary to specify it is not born to meet directly ISO 26262 requirements for Functional Safety; however AUTOSAR specification contains several chapters they deal the argument. In particular the Software architecture defined by AUTOSAR already meets some ISO 26262 requirements, especially for implementing Spatial, Temporal and Communication separation among Tasks and therefore avoiding the propagation of errors. This aspect is very important because in intrinsic manner it ensures the Freedom From Interference it is an aspect that is requested by ISO 26262 (chapters 6: *"Product Development at Software Level"*).
AUTOSAR documentation is divided into two main parts:

- Standard documentation and
- Auxiliary

The most of documents that are related to functional safety is labelled as auxiliary. Some of them provides several example about application of ISO 26262 for the development of a software component or more in general items using AUTOSAR as software architecture. Safety is an increasing aspect in AUTOSAR documentation and in the latest releases, especially in adaptive platform documentation is possible to find many chapters related to it.

1.5 Partial Networking in Automotive

How it has been reported in the introduction section, the number of ECUs in vehicle system is increasing across the years. Several studies have shown the relationship between the increasing of ECUs number and the power consumption. Another aspect is also the quantitative relationship between the power consumption and the fuel efficiency; It has been proved that a high power consumption has a very bad impact on fuel injection and then in CO_2 emission. In 2015 a scientific paper that was written by researchers of Sungkyunkwan University(South Korea) reported that in European Union, the target of CO_2 emissions was fixed to $130g/(Km * passenger)$ as average value [5]. Considering the increasing laws on environment protection, it is reasonable to imagine that target level of emissions will be reduced across the years; in fact, target CO_2 emissions in 2020, in European Union, it was fixed to $95g/(Km * passenger)$. The general optimization in power consumption

is not a new topic in academic and industrial fields. Especially in embedded systems, the studies about energy efficiency and reduction in power consumption are getting more and more popular. This because, embedded systems are delivered for long life time in a varied set of environments, with very short shutting-down periods. Sometimes, they shall harvest a continuous amount of data and information, using several protocols like in the case of IoT. The apparent simplicity that an Embedded System can have with respect to a general purpose computer, is involving in an higher complexity in order to meet the profile requirements for the delivering. Therefore, the power resources are increasing also for these systems and strategies to optimize their usage become a necessity. The methodologies range over from design aspects in order to implement ad-hoc solutions, to optimization in existent designs without drastic impacts on the original structure. Possible power optimization methods can be:

- **Processor power states as:**
  - Run;
  - Idles;
  - Sleep;
  - Stand-by;
  - ecc...
- **ISA optimization;**
- **transistor technologies:**
  - HVT cells;
  - LVT cells
  - ecc...
- **Communication protocol and bus encoding;**
- **RF and Network protocols;**
- **Memory addressing and partitioning;**
- **OS-based power saving mechanism:**
  - IEM (ARM);
  - ACPI;
  - Microsoft OnNow;
  - ecc...
- **Sensors and actuators algorithms**

Partial Networking is a communication protocol that has been introduced in automotive Industries, around 2010, as solution for the power saving. It involves into two topic of the previous list, i.e. the processor power state and the communication protocol on bus. In automotive industries, the bus system can be implemented by three main types:

- **CAN (Control Area Network):** most common, used in safety-critical communication among ECUs;
- **LIN (Local Interconnect Network):** is used as extension of CAN buses they can connect them, for sensors and actuator connections in a centralized control system
- **FlexRay:** it is faster and more reliable than CAN but more expensive;

To reduce the power consumption, for all of them, there is the possibility to implement the "Global Wake-up"; in the following only the CAN bus system is considered and analysed because it is in scope to the current paper.
CAN is a multicast, serial-data, communication protocol that has been introduced by Robert Bosch in 80s for Automotive applications. The big advantage using CAN bus, is in its capability to be applied at each environment condition, including ones they are affected by high electromagnetic disturbance. At the same time, it is possible to maintain a good transmission speed also on great distances (125Kb/s up to 500m). The tolerance to electromagnetic fields is obtained thanks to a transmission line with a voltage differential between terminals that can be used also in twisted pair to improve its immunity. CAN bus is standardized by ISO 11898 where it is described at *data-link* layer (with sublayers *LLC* and *MAC*) and at *physical* layer. The CAN communication packet is called *Frame*. Actually exists four type of frame but their differences are out of paper's scope and so they will not be analysed. The encoding for transmission is based on sequence of recessive and dominant bits (respectively 1 and 0). Each element that is connected to CAN bus, is called *CAN Node*. Single node shall be capable to send and receive messages or frames but not at the same time; these frames contain:

- Data Identifier which identifies the priority;
- One byte of Data;
- Overhead part it contains CRC, acknowledge slot and other information

A node shall integrate a **microcontroller**, it in turn integrates the **CAN controller** too and **CAN Transceiver** that represents the interface between microcontroller and bus line. CAN Node can interface with any other type of device of an embedded system by a simple digital logic, FPGA, PLD up to an Embedded computer it runs software. The general structures of Can Nodes connection to a bus line is shown in figure 1.5:

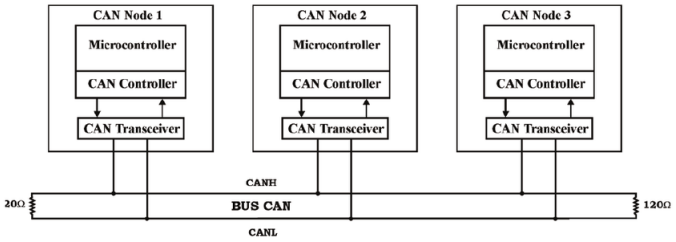


Figure 1.5: Can High-Speed bus node and physical layer (ISO 11898)

In the case of no dominant transmission, lines are put in recessive state and so $V_{CANH} \leq V_{CANL}$ with a nominal voltage difference of 0V. The *low speed* CAN can use a second

bus structure besides the linear one: star or multiple star buses. Also in this variant, the sub-structures shall be connected themselves by a linear bus with two resistors at each node terminals. These ones have a different value with respect to the first case (order of 100Ω). In this structure, the dominant bit is transmitted driving the V_{CANH}=V_{CC} (supply voltage) and V_{CANL}=0V (dominant state). Lines are put in recessive state when the previous values are inverted i.e. V_{CANH}=0V and V_{CANL}=V_{CC}.

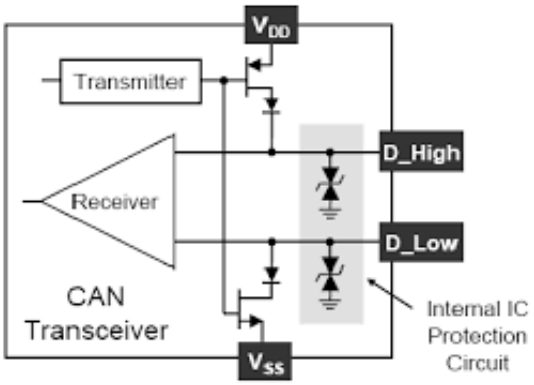


Figure 1.6: Can Transceiver(CanTx) for High Speed)

Basic version of CAN keeps each node always active, also in the case there are no message that are addressed to it. Each node shall check the frames data on CAN network at the same time. If more nodes are transmitting, they transmit until a bitwise collision (it happens when a node transmit a certain bit and it receives the opposite); at that point, the dominant bit transceiver wins the control on bus terminating its transmission, while the recessive will quit from bus controlling and it will attempt a new retransmission after a certain clock bits. In this way, basic CAN network is in a theoretical idle only when continuous recessive bits are transmitted and for all higher-priority message is ensured the transmissions without additional delays that are caused by collisions. This feature gets CAN protocol very suitable for real-time communication systems. At the same time is evident how basic CAN is not thought to be adapted for low power consumption system. In fact the communication between two end-point cannot be established at physical layer; each node will receive transmitted frame on network and if it is not addressed for it, there will be discharged by higher layers (that are activated anywhere).
Therefore, CAN bus has been modified over the years, in order to implement mechanisms they are able to wake-up nodes only when a transmission occurs. One of the first implementation of "selective" communication mechanism on CAN bus, it belongs to GM with GMLANS: it has implemented a system it was able to support the selective sleep on CAN bus and a global Wake-up. According to this mechanism, all receiving nodes were in a sleep state. When a frame was transmitted on network they were waked-up simultaneously. In this way the power consumptions were decreased but nodes they are not involved in communication were again waked up without reasons. The main features of CAN with the implementation of wake-up mechanisms are shown in the table 1.1.

The next idea has been to modify the CAN node at physical layer in order to introduce a filter it was able to understand if the sleeping node was involved in a communication or not; if it was, then node could be waked and it could receive the message, otherwise it could remain in a sleep state, decreasing the power consumption. In this configuration only

CAN Buses			
Type	Data Rate	Standard	Wake-up capabilities
High Speed	1Mbps	ISO 11898-1 ISO 11898-2/5	Active Bus (Global)
Low Speed	125Kbps	ISO 11898-1 ISO 11898-3	Active Bus (Global)
Signal Wire	83Kbps	GMLAN	Higher Supplier Voltage (Global)

Table 1.1: **CAN properties with wake-up implementation** [5]

active modules were: the bus master and the filers of each node they belong to Network. this method is called "Selective wake-up" and CAN networks they implement it are called "Partial Networks"; within them, network nodes can be simultaneously in a sleep state or active state, reducing the power consumption to the necessary. The main operating difference that are described, between a global wake-up bus system as the GMLAN and the Partial Networking can be seen in the figures 1.7 and 1.8

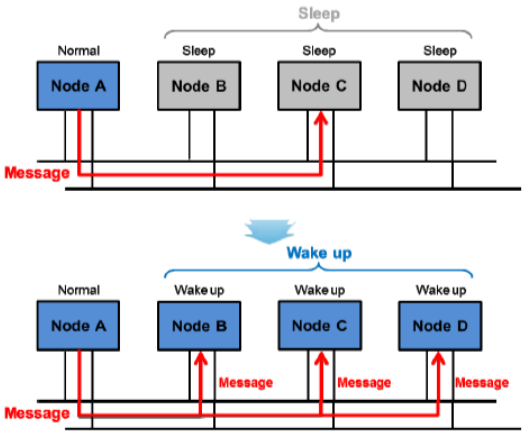


Figure 1.7: Global Wake-up CAN bus system from [5]

The key point is to understand how the physic modules have been modified in order to implement the scenario of figure 1.8: The basic idea has been to implement a logic mask it was able to understand at physical level if the node was involved in the actual communication. "Complex CAN transceiver" are nodes that are able to implement this filtering mechanism. The first new module shall be a configurable frame it addresses the wake-up sequence to receive. This wakeup frame can be configured using the SPI (Serial Protocol Interface). The network frame is received by the original CAN Receiver and it is decoded by a decoder that is timed by an own in-module oscillator. At this point, wakeup configured frame and received one can be matched by filtering algorithm. A simplified block diagram (based on TJA1145 by NXP [7]) for implementation of the previous description can be seen in figure 1.9. The wake-up frame will be configured in order to consider a fixed-length sequence of bits called *ID* and another for Data. In the first one, some bits are ranked as "Care" and the remaining as "Don't Care" and a value between 1 and 0 is assigned to each

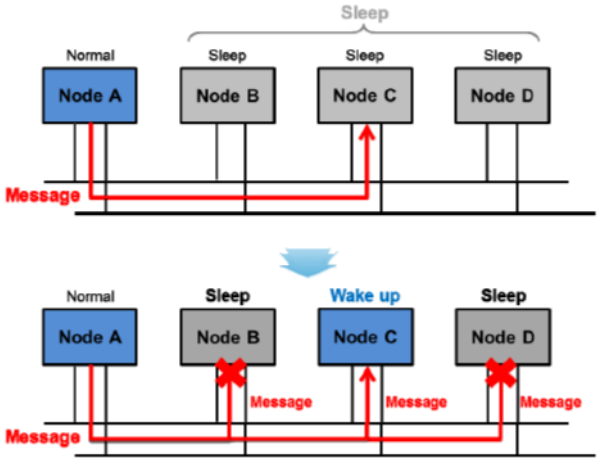


Figure 1.8: Partial Networking CAN bus system from from [5]

of them. At the same time, is configured a portion of data frame with a specific value. When a frame is received, filter will match the value in the ID configured frame and if the "Care" bits correspond to the respective the node is involved in the actual message. At the same time, filter checks the integrity part of data message: if at least one recessive bit in the configured frame will match in data field the message will be considered as integrated and node can be activated. This mechanism is related in special way to Complex CAN transceiver TJA1145 but the basic idea to match a configured identifier with frame values and the integrity of the data is common to each complex Can transceiver.

From a theoretical point of view, keeping nodes in a low power state for as much as

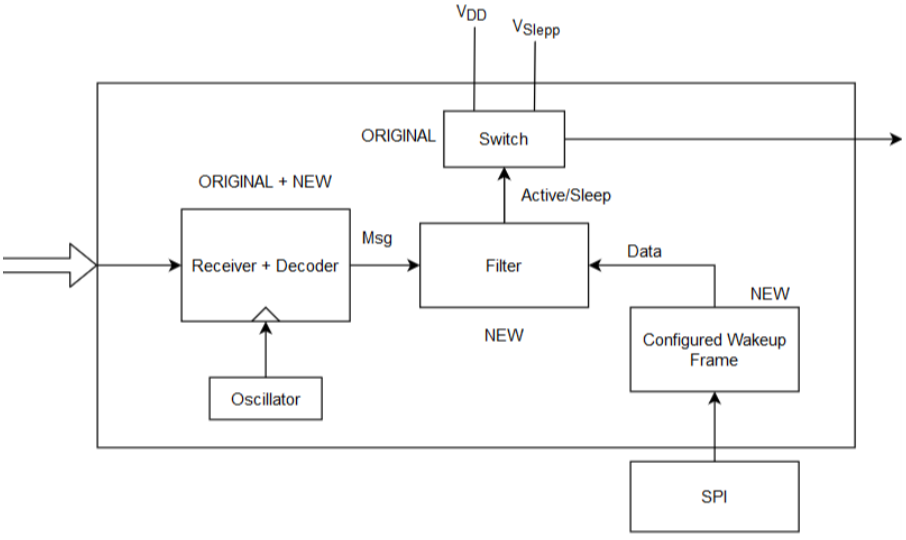


Figure 1.9: Complex CAN transceiver

possible should reduce the overall power consumption but it is necessary to understand how power can be saved increasing the complexity of the communication modules. Before looking some results about simulation of a target environment, it is necessary to specify that by software point of view, the introduction of Partial Networking introduces two troubles:

1. The transition between power states in a microcontroller-based system that runs software is not a atomic unit operation in terms of time and so it can affect the real-time operating system deterministic nature;

2. Considering topic *1)*, is strongly not recommended to map system they implement time-based safety functions in a Partial network;

This means that not the whole set of ECUs in a vehicle can be in sleep state for long time; some ECUs cannot be in a low power state. Therefore, the real power saving using the partial networking strongly depends from the mapping criteria used for identifying partial networks. Based on these criteria, in a Korean research [5], a vehicle context has been simulated using CAN bus with Partial Networking. Simulated target was built by 40 ECUs with the 30% of them could be mapped in PN. The network was built considering 12 ECUs that could be activated by activating ECUs; each of them could activate at maximum 3 ECUs in its own Partial network. At the end, simulation considered each ECU could consume at average value of 200mA if active and 0 if in sleep state. With these values, all Partial networks have been activated within the same simulation, using their operating sequence and showing a reduction of power consumption of 1.4A with respect to the basic CAN network, with a power saving of 17.88%. According to previous researches, the relationship between current consumption and CO_2 emission is of 0.35g/(Km*A). This means that according with the actual simulation, the adoption of Partial Networking has reduced the CO_2 emission of:

$$Reduction_{(CO_2)} = 0.35\frac{g}{Km * A} * 1.4A = 0.49g/Km$$

1.6 Real-Time Application - Partial Networking Coordinator development by ETAS

ETAS Company is an AUTOSAR Premium partner that joins in BOSCH Group. As Premium partner, it is a provider of "Tool and Services" and Applications for Embedded Systems that are deployed for Automotive Industry. One of the first product portfolio of ETAS has been tools for Model Based-software design. The reaching of the state-of-art for Company products in automotive ones is ensured, meeting the requirements of several standards and in particular ISO 26262. Actually, ETAS portfolio covers the following fields:

- Software Engineering;
- Testing and Validation;
- Measurements, Calibration and Diagnostic;
- Real Time application;
- Cyber-security protection;

Partial Networking in complex system, with a high number of ECUs ,requires the implementation of a coordinator. The development of the Partial Networking Coordinator, in ETAS, is linked to a series of developed products that are called SUM (Standardized Utility Module). ETAS has started to develop these products for General Motors for its own software platform. Since November of 2020, the SUMs have entered in ETAS products with name of RTA-SUM, where RTA prefix addresses all ETAS embedded products for Real-time application deployment. Considering the AUTOSAR Layered structure, SUMs are developed as a sort of lower sub-layers of Application Layer. Their task is to manage all services from Basic Software Layers for specific functionalities. Therefore the main task of the PNC (Partial Networking Coordinator) is to manage all services related to Partial networking in AUTOSAR application layer. The concept of "getting transparent" is strongly related to obtain a high level of flexibility and abstraction in application layer. The concept is to develop a product to be reused in each design type. ETAS has identified PNC as Safety Element Out-of Context as specified by ISO 26262:2018 chapter 10 clause 9. Company goal is to implement the Partial Networking Coordinator with the highest ASIL value (D) for covering all possible use case scenarios of own customers. According to this, the necessity is oriented to get compliance of the initial beta version of module, that has started as a functional implementation, without too much concerns about safety aspects, with ISO 26262.

Chapter 2

Reading of Standard, Initial analysis and Requirement Engineering

2.1 ISO 26262 parts in scope to SEooC development

The development of a Safety Element out-of-Context requires an ISO 26262 analysis, starting from its definition in chapter 10, clause 9 until the identification of mandatory requirements (that cannot be omitted) and others they can be tailored.
First of all, ISO specifies the difference between a *Safety Element out-of-Context* and a *Qualification of Software component* development and it requires the correct identification of the product within this classification. Their differences are not so immediate, especially in the case of Partial Networking Coordinator; in particular, the early released beta versions met in many topics the definition of qualification software, because PNC was not developed under ISO 26262 standard but potentially it could be used, as existing software, for a ISO 26262 item development. Besides, SUMs are developed under functional requirements specification that are provided by Customers (GM). The real reason, it forces the identification of this module like a SEooC, is the necessity to develop it as **reusable software component with a high level of flexibility**.
The development of a SEooC is based on assumptions. These ones will be validated during integration phase and in this specific case by Integrator engineers; the validation of assumption will represent the Qualification of the software Component. Identification of assumptions shall consider the following aspects:

- Starting Design Hierarchical Level;
- External Interfaces of software component according to its scope;

Design Hierarchical Level The hierarchical levels of a software element are described by the V-model.
To understand the right design level for partial networking coordinator, is necessary to interpret the meaning of element in ISO 26262 scope. In software engineering, The "element" definition can identify a generic software product it might belong to several architectural layers and for identifying something of similar to an item but that is not developed within a specific vehicle functionality (in Automotive context). The definition of element can be obtained looking the following diagram:

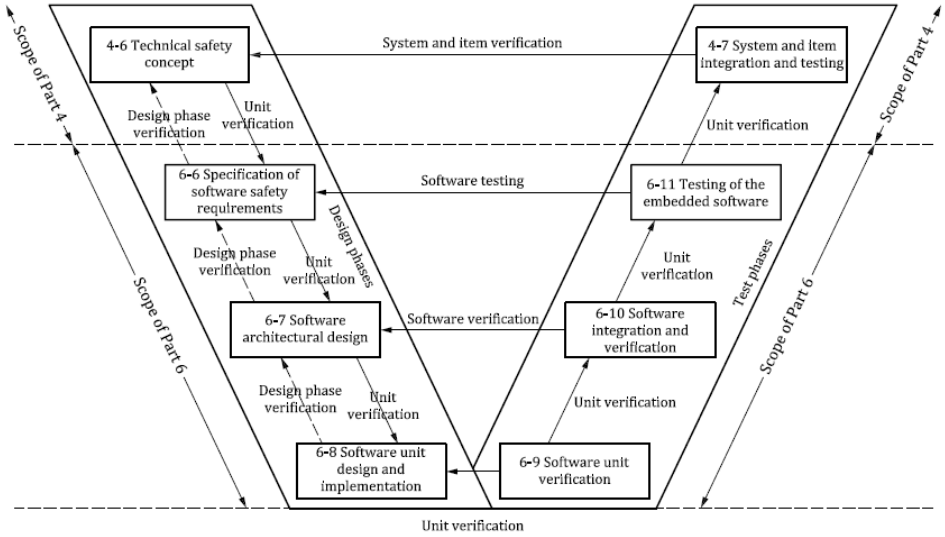


Figure 2.1: V-model process as suggested by ISO 26262

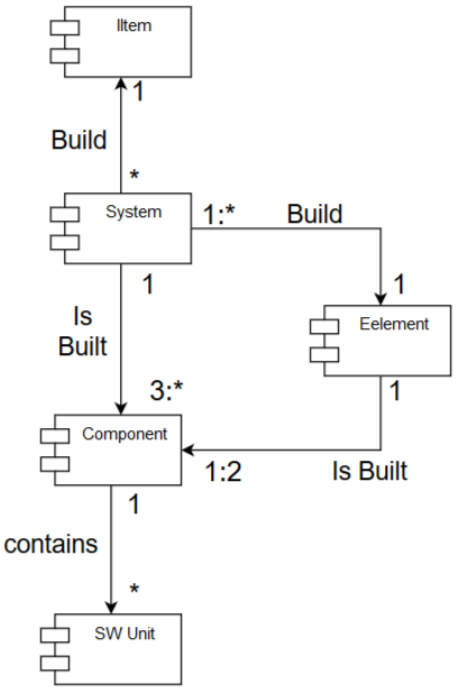


Figure 2.2: Class diagram to identify element definition

The Partial Networking Coordinator is actually a software module i.e. a **logical partitioning of functionalities that once it has been divided, its results in no more modules entities but they can maintain modular properties**. Instead a software component is built by Units. Components interaction defines a system with a certain architecture and **when a component is divided, the result will be always subcomponents**.
The goal of a component is to implement a group of functionalities. Therefore it is possible to say that *one or more components implement the logic functionalities that are partitioned into a module,by several software Unit.* Let's imagine the following example: an

ALU (*Arithmetic-Logic Unit*) groups all system functionalities related to arithmetic and logic operations or more in general the "computational" functionalities of a system. So an ALU can be considered like a **Module**. At this point let's imagine to split "computational" functionalities into two parts: one for arithmetic operations and another for logic ones. Now, there will be a sub-set of software units for implementation of arithmetic operations and another one for logic. The software Components will be source code files and their related headers, they contains these software units that are grouped according to the implementing functionality type.
For each module is possible to define specific ports they implement interfaces. By these last one is the possible to define communication channel that are based on specific protocols and they allow to define a **System**; ISO 26262 considers it as an interaction among at least three hardware components, with their related software definition where it is possible, they play roles of: sensor, processor and actuator. Each system implements more than one physical component (or device) but less than three can be considered as an element.
One that these terms and their definitions have been clarified, it is necessary to identify

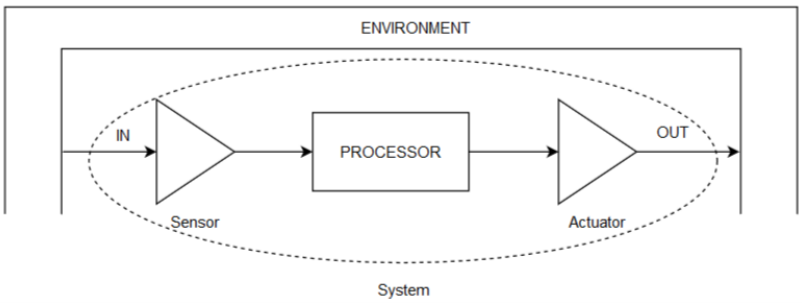


Figure 2.3: system as considered by ISO 26262

the PNC nature; From functional specification a partial networking coordinator shall evaluate criteria, in order to activate the Application software components that are involved in that specific partial network or it shall request the activation of a Partial Network by Basic Software Services. The inter communication requires a communication based on signals and frames (they can be seen as the "actuators" of PNC). If the Partial Networking Coordinator is considered with sensors or actuators embedded in the module (mixing hardware and software products), it may be developed as Safety Element out-of-Context at System Level. In the case where Coordinator implements only the control part, how in the current case, it will be implemented at component level and so all higher level in hierarchy will be just *assumed* in the definition of safety requirements.

External Interface The external design interfaces are related to module (and in turn its building software components) boundaries. These are the borderlines of component functionalities and relative implementation. In the development of a Safety Element out-of-Context is a very important aspect because it shall contain interfaces related to external modules they implement communication mechanism for in-vehicle functionalities. For example, several components run routines that are able to detect an error at unit or component level but they are not able to mitigate or managed it at system level to allow system keeps running. In this case, component shall implement an interface to another one it can manage error occurrence.

When the external interfaces have been also defined it is necessary to define the assumptions on safety requirements. These shall provide higher-level safety requirements with respect to component level, and in particular they are already oriented to Technical Safety ones. Requirements, once they will be allocated to the right product type, represents the starting point of the implementation.

Figure 2.4 summarizes the concepts that are expressed in the previous paragraphs. The

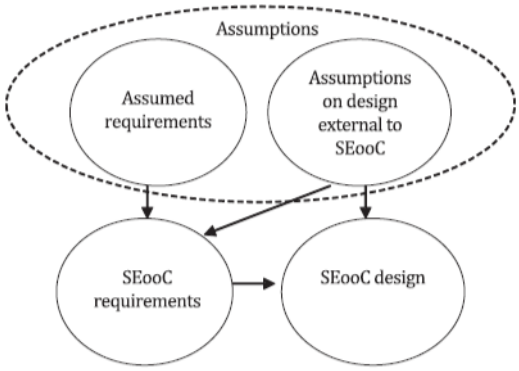


Figure 2.4: assumption as specified by ISO 26262

"SEooC Design" represents the development following V-model structure.
Although the specification of assumptions on safety requirements is central part of this initial phase, the guidelines on SEooC development specify all safety activities that must be considered too. The nature of design plays a fundamental role: clauses and requirements they are suggested in chapter 10, clause 9.2.4 can be tailored or at the same time not-applied. All supporting processes to main flow (described by chapters 7, 8 and 9) should be applied according to **safety plan**. Besides from this description it is possible to understand how much **Documentation** will matter: the compliance of a Software Component , developed as SEooC, with the standard can be evaluated like positive only if all assumptions and tailoring activities, including the omitted ones, will be documented with a good rationale about design choice; in summary is necessary to document that every activity in safety plan has been performed and it has been done in compliance with series of standard; everything it has not been applied, it could not be done due to project nature.

This last paragraph will be oriented to understand better the meaning of *Assumption*. They depends from the nature of element. In the case of a Software Component they can be split within two types:

Scope This includes a series of consideration. Within scope mean: software architecture, target environment, boundaries, functionalities and properties. All of these topics will be useful to derive the nature of software and at the same time its global features. From them it will be possible to draw a first view of the software architecture. Let's take the partial networking coordinator case: it is developed in AUTOSAR and this means that it will belong to a layered software architecture, in particular to the *Application* one. Its interfaces shall meet AUTOSAR model and behaviour specification. All aspect related to error detection will be compliant with AUTOSAR models or related to good-coding

practise, implementing specific units. Boundaries will be defined according to AUTOSAR specification too, in order to ensure the avoiding of fault propagation among components. The target environment can be considered like an ECU where it will be integrated. For sure the ECU will be used for a vehicle with an high number of them and that is able to implement a CAN complex Transceiver for Partial Network supporting. Really, the last part can be omitted in assumption scope because it will be considered in deep in the **Hardware-Software Interfacing (HSI)** clause of chapter 4. In the development of a SEooC the HSI will be identified in the safety manual document for integration guidelines.

Safety Requirement These assumptions aim to replace the classic concept phase that cannot be applied in the case of a SEooC. The assumed Higher-level Safety Requirements already consider allocation in the system architecture where element is placed, so they can be considered as **Technical Safety Requirements or TSR**. The main purpose of these assumptions is to derive from **TSR** the **Software Safety Requirements (SSR)** to implement the Safety Goal of Software Component. figure 2.5 is shown the SEooC process flow, how it is considered by AUTOSAR specification and in AUTOSAR document *Specification of Safety Extensions*, in figure 3.1, the representation of it with respect to the block view of a system

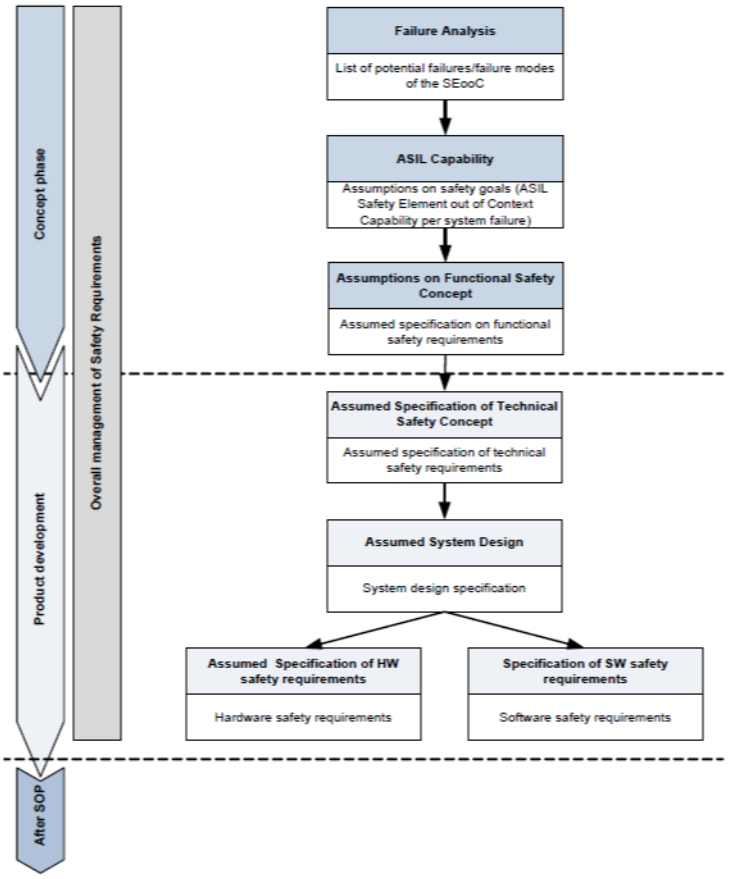


Figure 2.5: SEooC process flow (AR specification)

In this phase is strictly necessary to consider the ASIL value of the software component

because it has a high impact on the definition of SSR. The identification of SSR shall be done according to Higher-level Safety Requirements they involve element tasks, as the monitoring and the verification of PN status (and eventually the request to change it), communicating with the Basic Software Modules; In a similar case it is necessary to consider the possibility of errors: the coordinator operates in a system architecture, therefore it is able to communicate by interfaces with internal system component and external resources. In some cases, an error might arrive from outside, in others it might be caused by a component within the same system. So Safety requirements shall grant it will not propagate itself among other components with respect to which has generated it. This is valid for coordinator too: it shall be able to prevent the propagation of its errors. Coordinator shall not be able to manage all possible error they might occur inside the system but only ones are in its-scope.Let's imagine another component causes a malfunction in a basic software service that is used by the coordinator; this last shall be able only to handle an error, checking an integrity attribute on the data or the expecting result with the received one (just for example). Actually it might not able to know which component has generated that error and system reaction might be out-of-scope. Therefore the only thing it can do is to generate a return value to feed an interface for another component it can inform the system about the faulty status. Software Safety requirements might not be only derived by TSR: in the specific case of the Partial Networking Coordinator, the functional requirements are provided in a specific way by GM Company. The most of them are related to component functionalities therefore they are marked as "functional". In this case **SSR** can be derived performing a sort of validation on providing requirements. To be clearer, let's imagine the following simile example:

```
1. Coordinator shall activate PN
   "X" in a defined time;
2. If the activation will fail,
   it will attempt until state
   is successful;
3. The entire activation process
   (request, activation,
   acknowledge) shall be performed
   at max. in a certain time
   (Worst Case Execution time);
```

The previous requirements are affected by a safety Gap: their contents address that a Coordinator shall perform a set of operation they will move the PN state from an inactive one to active in a specified time, probably measured by a timer. If time expires and PN state remains the same, the operation shall be repeated. This case will take time and if the system is affected by a persistent error, the activation will fail again. The requirement specifies that process shall be repeated until its successful exiting, omitting the maximum number of attempts that can be done. The scenarios can revealed are the following:

- The coordinator tries to activate PN until it will do.
  - A persistent error affects the communication protocol at lower layer;
  - PN will never be activated **(Deadlock of Coordinator)**;

- The coordinator tries to activate PN for a certain numbers of attempts.
  - Each attempt will take time. PN is finally activated;
  - attempts "wasted" time will involve in missing deadline for ack reception or state transition;
  - PN Status is uncertain or PN is active but it is detected as inactive **(Failure, not deterministic behaviour, Performance Penalties)**;

In the development of SEooC these Safety Gaps can be filled by Assumptions on safety requirements. Looking the previous example, Safety requirement shall consider several aspects. First of all can be related to the number of attempts: considering single operations time with respect to the overall worst case execution time, it is possible to compute the maximum number of attempts as in the following:

$$T_{WCE} \geq (T_{Req} + T_{Activ} + T_{Ack})$$

Let's *Assume* that the sum of times is never equal to Worst Case execution time (T_{WCE}). This means that difference between The worst case and sum of them will be greater than 0. Therefore that time can be used to perform other request and activation routine attempts as the following formula specifies:

$$Attempt_{Num} = \frac{T_{WCE} - (T_{Req} + T_{Act} + T_{Ack})}{T_{Req} + T_{Act}}$$

This means that if the Worst Case Execution time for activation is $200ms$, and requesting, activation and acknowledge are respectively $10ms$, $30ms$ and $5ms$, the number of additional attempts will be equal to 3.875 i.e. 3.
If no attempt has been able to activate the PN they can set an error code for returning value at the expiration of time slot. This error code can be sent by interface to another component or for requesting to the Basic Software a diagnostic of the resource. After that assumption are formulated within a dedicated document, in general the **Safety Manual**, it is possible to proceed with the development according to ISO 26262:2018 chapter 6 clauses and requirements.

From ISO 26262 parts that is related to the definition and identification of Safety Element out-of-Context and ISO chapter in-scope to its development, it is necessary to identify the **work products** that Development Company shall provide and it shall plan their as activities. Again this list is not unique for the development of each Software Component it follows the SEooC design flow and each Development team can choice how to implement them in tailored manner. It is very important to produce work products that can be exhaustive in terms understandability, readability, traceability and maintainability characteristics and, at the same time, complete in terms of compliance with the series of standards. According to element nature, different processes might be applied but the real important aspect is to follow a common approach: let's look ISO 26262 required processes for a SEooC like grouped in the following macro step (excluding safety management activities):

1. Identification of Technical Safety Concept and Technical Safety Requirements;

2. Development according to SSR;
3. Functional Safety Assessment;
4. Qualification

This list already provides a better view on which information need to be documented to assess the compliance. In particular topic 4 cannot be executed totally by development team: the only possible qualification in development phase is related to the verification of what has been developed. In other terms Qualification can be considered as *Validation phase* and this can be done only by integrator with respect to supplier developer assumptions. According to previous description the following list, related to the identification of work products for the Partial Networking Coordinator developed by ETAS, shall be considered as exhaustive for the specific element development or in a certain sense as specific "safety case" even if it is not really correct to speak about safety case for the development of an out-of-context product. According to the previous analysis, the provided work product for Partial Networking Coordinator have identified in this list:

- Safety Plan (compliance with "Safety Management" part of ISO 26262);
- Software (Functional) Requirement Document (provided by GM);
- Software Design Document or SDD (compliance with "Product development at software level" part of ISO 26262);
- Software Verification Plan (compliance with a series of chapters of ISO 26262);
- Safety Manual to replace the Safety case in item development;
- Failure Mode and Effects Analysis or FMEA at System Level (series of chapter of ISO 26262);
- Functional Safety Assessment;

The next sections will analysed in a detailed manner each requiring work product, safety-related, that has been presented for the development of Partial Networking Coordinator, excluding the template creation for safety plan that is provided by Company.

2.2 Requirement Engineering: Assumptions on Safety Requirements

When the safety-life cycle is applied to the development of an item, the first step aims to define it considering:

- Scope of the item by a functional point of view;
- Its Boundaries;
- Its building Elements functionalities and architectures;

A good way to identify item functionalities is to define them as state operation of specific system modes or operating states. This enforces the determinism of item (or of a generic element too) behaviour. Operating states are defined by each *considerable* combination of elements parameters, they assume each value in its own dominion. "Considerable" means each value of a specific parameters it concurs to define the operating state that really affects the item functionalities. For example: an Item is developed for Parking Assistance in reverse (limited visibility), therefore it shall consider the actual gear of the vehicle; Gear values dominion can be $\{R, 1, 2, 3, 4, 5\}$ but only the case where Gear is R concurs for a real Operating States at Item scope. After that it is possible to identify the Hazard operations by HARA; at this scope some supporting processes can be conduced as for example a FMEA. A good methods might be to analysed each building element, going down as much as possible in its structure (also if it is a "virtual" one because described by a functional point of view) and to find each: parameter value, combination, event or element boundaries that might have a negative impact on Item functionalities. In the most of the case, they can identified as the remaining combination of elements parameters does not identify an Operating State (note that, for being an Operating State, item shall work fine). A cross matrix is able to help in the assignment of the right ASIL level at each Hazard Operation with respect to each Operating Sates (figure 2.6). Once it has been done it is a good practise to merge Hazards how it is possible, according to ASIL value too, in order to define the Safety Goals and their relative Functional Safety Requirements (FSR).

This procedure cannot be applied, exactly how it is described, to the "concept phase" of a SEooC. In particular, to be very strict, for a SEooC the concept phase does not exist or at least, it cannot have the same meaning. The scope of the current phase is to report how the SEooC equivalent Safety Concept has been defined for the Partial Networking Coordinator and which work products will contain it. Starting from this last one, in the development of a SEooC each assumption shall be reported inside the Safety Manual (look 5.2). ISO 26262 makes a precise separation among System, Hardware Element or Software Component that are developed as SEooC. When a Software module shall be developed as SEooC at component level, it is not possible to consider the safety aspects that are related to Software Component without consideration about the hierarchy where it will be integrated (this is mandatory). For this reason the starting point for the assumption on safety, about the Partial Networking Coordinator, shall be the system level. Fortunately, GM has defined The PNC like a member of a specific software Platform and so it is possible to derive important information about it and software module interaction. The identification of the interactions with others elements build the assumptions about the **External Interfaces**. They involve the safety aspects of the PNC because:

A) It shall evaluates the correctness of data that are requested on interfaces from other elements;

B) It shall ensure the correctness of data it provides to other elements by interfaces;

The correctness of data must be ensured and for this, several redundant mechanisms shall be specified in the safety Requirements. Having more knowledges about platform, the usage of the FMEA as supporting process for a "simile-HARA" is strongly recommended. In fact, safety analysis allows to identify a possible Safety mechanism for each software component, avoiding the propagation of a fault; once it has been identified and formalized as a Safety Requirement it will be implemented at Unit level. One big difference with

Operating States	Hazard	Risk Evaluation	ASIL	Notes
OS00	H1	**S**: **C**: **E**:	*A/B/C/D/QM*	
	H2	**S**: **C**: **E**:	*A/B/C/D/QM*	
	H3	**S**: **C**: **E**:	*A/B/C/D/QM*	
OS01	H1	**S**: **C**: **E**:	*A/B/C/D/QM*	
	H2	**S**: **C**: **E**:	*A/B/C/D/QM*	
	H3	**S**: **C**: **E**:	*A/B/C/D/QM*	
OS02	H1	**S**: **C**: **E**:	*A/B/C/D/QM*	
	H2	**S**: **C**: **E**:	*A/B/C/D/QM*	
	H3	**S**: **C**: **E**:	*A/B/C/D/QM*	

S: *Severity*;
C: *Controllability*
E: *Exposure*

Figure 2.6: Example of Cross-matrix for risk evaluation

the concept phase in Item development is that the SEooC is developed according to an *assumed* ASIL value too. This means when Hazard and Operating States are identified, it is not necessary to create the cross-reference matrix but at the same time, each hazards shall be "covered" by a Safety Requirement that defines a safety goal.
For PNC, assumption have been started from product development at System level. The identification of the TSR allows to define the **HSI** (Hardware Software Interfacing) and in turn the **SSR** for the PNC. The functional description of the System is useful to proceed to assumptions how in the case of the item development. All these information will be contained in the Safety Manual. The "concept phase" for the PNC has been solved as in the next paragraphs:

Approach with respect to Component scope and Target System : Usually the Partial Networking is a solution that is used for optimizing the power saving in a vehicle with a large number of ECUs. Some of these implements functionalities that are rarely requested during a single vehicle life-cycle. Besides, how it has been specified in Partial Networking, The partial networking performance might be strongly related to the start-up sequence duration and the duration of a Microcontroller wake-up. The combination of the

previous considerations should be enough to prove the impropriety of mapping a safety application on a Partial Network, because of not nondeterministic aspects. By the other side, several safety applications absolve their scope in rare cases and therefore they might be mapped in a Partial Network to save power; rationally these should not have an high ASIL value anyway, allowing the risk reduction with the caution of the driver. For this reason two possible conclusions have been evaluated as possible:

A - Safety Strongly Oriented: because of the possibility to map safety applications in Partial Network, the coordinator active the PN at the minimum perceived intention;

B - Performance Oriented: The module goal is dominant (saving power); the PN is activated only if the intention of activation is certain beyond each reasonable doubt;

This choice is very important considering the product nature. The final decision has been taken, performing a trade-off between the previous possibilities.
The first approach has been already adopted in the assignment of ASIL value: Partial Networking Coordinator meets ASIL D. The reason is related to ensure the **FFI** (Freedom From Interference) in the case of ASIL decomposition within system where PNC is employed. Basically this problem can occur in system where elements with different ASIL value coexist: potentially if an ASIL D element requires resource to an ASIL B one, lower level of safety of provider might cause problems to ASIL D element. Instead, if an element with ASIL B requires a resource from ASIL D, there are not safety problem because requester receives an "overheading" safety resource that does not impact on its operations. The PNC is considered an Utility Modules and for a generic Software Component in AUTOSAR application layer, it will be considered a service provider. At the same time, considering the previous considerations, the most important thing is that PNC meets its scope; if a strong approach about safety is adopted, it might request the PN activation or deactivation at each minimal variation in its sensors, decreasing the power saving. Therefore the safety requirements will aim to ensure *data integrity* and *consistency* on its interfaces, according to ASIL D methods. In this manner, PNC will be able to take an indisputable decision about the activation or deactivation of Partial Network.

Assumptions on Higher-Level Safety Requirement (equivalent TSR) : This phase is actually composed by two sub-phases. The first one makes assumptions about Target Environment i.e. at vehicle level, System nature and/or related software architecture; for example, several assumptions can be done about the availability of AUTOSAR services and modules on target environment. Then, considering elements scope, it is necessary to identify:

- Functionalities and Properties;
- Boundaries that are represented by external interfaces;

Let's note that some aspects are equal to previous description. To be clear, the previous discussed approaching phase is not effectively described as an integrating part of ISO 26262 but for the actual experience it has been found a very useful activities. The assumptions on functionalities and properties allow the identification of specific tasks that can be implemented by TSR. Properties can consider AUTOSAR safety aspects that are implicit to architecture but that requires attention: ISO 26262 specifies very well that a solution shall

be adopted with knowledge also if it belongs to product nature. From them it is possible to proceed to assumption on safety requirements at system level (they called by ISO 26262 **Higher-Level Safety Requirements** (version 2018, chapter 10, clause 9: "development of software component as SEooC"). How ISO explains, they can be derived in order to obtain a final technical formalization of safety-relevant aspects. In general, Higher level Safety Requirements can be considered as TSR and so they shall bear in mind the allocation to the right dominion (Software or Hardware). Each assumed Safety Requirements shall be covered at least by one assuming HSI topic and then allocated. This phase should allow a faster allocation of TSR within SSR than item relative case.
The identifying Safety Goals and SSR related to assumptions on PNC cover the following aspects:

- Ensure the error masking on input signals from sensors by *Forward Recovery*, in order to avoid unintentional activation or deactivation of a Partial Network;
- Ensure the error handling (detection and mitigation)on arithmetic operation for critical resource;
- Ensure the Control Flow Monitoring among Software Unit;
- Define a Safe State to avoid the occurrence of Hazards once an error has been detected in a software unit;
- Ensure FFI developing the Module with the highest ASIL and using AUTOSAR features and properties;

To improve the traceability and a verification on the quality of assumptions, in the safety manual there are several cross-matrix that are able to map the Assumed High-Level Safety Requirement with respect to identified Safety Goal at component level. This will help the integrator in the process of Qualification, according to ISO 26262, when assumptions will be effectively validated. The scope of ETAS is just to trace each assumption at several level, in order to verify the correct implementation of the requirements, starting from their source (more detail about traceability in Verification chapter).
Let's note that the Assumed Safety Goal at component level, aims to define which safety mechanisms and aspects can provide the component with respect to system; ISO 26262 explains a mapping should exists between TSR and SSR but to decrease the number of relationships multiplicity, The assumed TSR are assigned, considering the Safety Goals that in turn will contain Software Safety Requirements. In figure 2.7 is possible to see how the cross-matrix has been implemented within the safety manual and how the assumed SSR contains reference to the Safety Goal (table 2.1)

Id	Tag	Description	Realizes
Number of Requirement	Identifier used for traceability and identification	Requirement specification	Tag of realized safety goal

Table 2.1: Specification of SSR in safety manual

Table 6: Safety Goal and Higher-level Safety requirements allocation

A-HSR-PNC	SG_PNC00	SG_PNC02	SG_PNC03	SG_PNC04
00	X	X	X	X
01	X			
02		X		
03	X		X	
04	X			
05				X
06				X
07		X		
08	X	X	X	X
9				
10				
11				X
12				
13				

Figure 2.7: References between Safety Goals and HLSR

Chapter 3

Design and Implementation

3.1 Briefly overview on V-Model

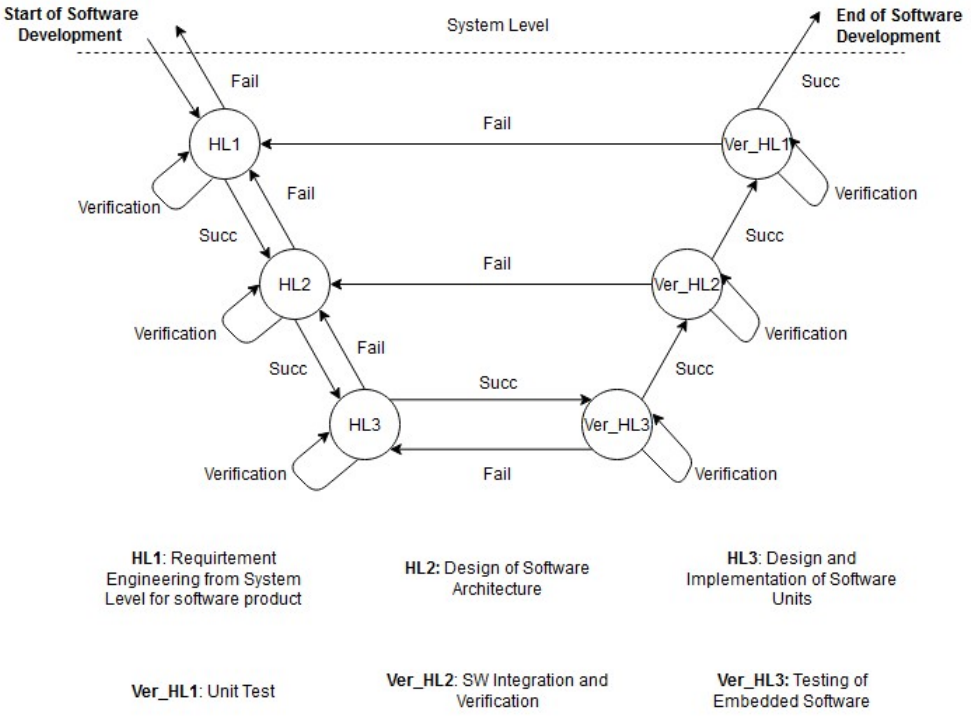


Figure 3.1: Other View of V-model

Figure 3.1 shows another view about activities they shall be executed for the development of a software product according to V-model. Each circle represents the development of a specific software hierarchical level, according to ISO 26262 requirements and methods. Activities for *HL1* have been already analysed in the last section of the previous chapter i.e. assumptions on safety requirements that shall be implemented. The identification of the transition between Hierarchical Levels (up or down) will be subjected to the result of verification methods, that are identified and discussed in Verification chapter and that are represented by loops (look figure). Verification in implementation branch (left one) should prevent as much as possible the case of failure in definition, designing and implementation, otherwise a failure in respective Testing phase (right branch), on the same Hierarchical

Level (*Ver_HLxx*), will force the process to come back again on implementation branch and repeat it.

3.2 Tricks on Modelling and Programming Language

This phase is not represented in figure 3.1 but it shall be considered totally in scope for SEooC development. It can be seen as a global aspect, to be considered at each hierarchical level for the software development. Clause considers particular aspects of a software development as for example the complexity by an algorithmic implementation, programming paradigms and concurrency. Besides, this clause makes mandatory MISRA C subset, especially for products with high ASIL value.
To evaluate the complexity of a software product, a good approach is to adopt the **Cyclomatic complexity**. This one uses the Control Flow Graph to measure the number of paths that are linearly independent. A good reason to adopt this measure is its applicability to classes, methods and functions in an independent way. The cyclomatic complexity for a program, a method or a function increases according to the number of constructs that are able to fork a path, i.e. those nodes they have more than one outputs (if, for, while, switch, etcetera...) called *decision blocks*. An easy way measure a function complexity consists in counting the number of the previous cited constructs and adding a 1 i.e.

$$CC = db + 1$$

Where CC is the Cyclomatic complexity of the software unit and db is the amount of decision blocks in software unit. This computation and this software measure have been defined by Thomas McCabe [14] and they are valid only in the case where the Control Flow Graph has not strongly connected component i.e. the ending node of CFG is not connected to the starting one. At the same time, a single entry point and a single return one shall be present according to MISRA C requirements (structured software). The lowest value of this measure is *1* i.e. an unit with a single path and so no *if* or iterations. This is very rare in an embedded software, especially considering, that safety measures often require redundancy and related matching among values. This consideration will be discussed in the following when the implementation of software Unit and an example regarding the development of PNC will be reported.
One of the most famous application of this software measure is to use it for assigning an upper bound value for software unit complexity equals to *10* (NIST suggested practise). When a software unit goes up the threshold a good practise might be to split it in more units; this shall be done considering that in embedded software might be not very correct, because each unit corresponds to a mode or a particular state; therefore another deriving idea might be to group specific statements, that are repeated in several units, in order to create a simpler one as a private function that can be accessed by component's units. The advantages are also related to the correlation that exists between number of defects and cyclomatic complexity: although studies have not been able to prove that a low value of McCabe's measures actually reduces the defects occurrence, it also has been proved that cyclomatic complexity might have a positive impact on program size (amount of LoC), especially considering the previous grouping method; By the way, a larger program in terms of LoC will have a major probability to contain errors.
The last remarkable feature is the possibility to use the cyclomatic complexity for evaluating the number of test cases that will be necessary to verify a software unit; in particular the

cyclomatic complexity can be used to obtain the lower-bound number of test cases that are necessary, for obtaining an appropriate coverage in *white-box* testing. Specifically, McCabe has provided a inequality where the cyclomatic complexity is the middle value between the number of test cases that are necessity to obtain the 100% of Branch Coverage and the ones for obtaining the 100% of Path Coverage.

$$TC_{BC} \leq CC \leq TC_{PC}$$

where CC is the cyclomatic complexity, TC_{BC} is the number of test cases to obtain the maximum of Branch coverage and TC_{PC} is the number of test cases to obtain the maximum of Path Coverage. At scope of ISO 26262, for code coverage, according to ASIL D, both Branch Coverage and MC/DC can be chosen. If the first one will be adopted, the identification of cyclomatic complexity for software unit can address the upper number of test cases they shall be written to obtain the 100% and at the same time, how NIST suggests, it is a good practise to write, for a software unit, at least a number of test cases equal to cyclomatic complexity. Instead, using the MC/DC like code coverage, this value might lose validity.

The second important aspect is to develop code according to MISRA C. This reduces language features to a simpler subset that can be asserted as safe. Mainly the following features shall be prevented:

* Dynamic Memory Allocation;
* Recursive programming;

Beyond of usage of language subset, it is important to consider the security aspects. Often the **defensive approaches** improve the security of the unit or component but they might have a negative impact on the safety aspect, reducing the capability for testing. Two examples can be:

- Abstract Data Type (ADT);
- Use of software safety mechanism to detect and prevent permanent faults;

First one aims to implement private struct. In this way is possible to make C very similar to concept of **private** attributes like C++ and Java. Therefore each component or module that includes the resource's header typically a test program with a **main** or a **client**) will be able to access only by pointer to the struct type without actually knowing its internal records; in this manner it cannot modify directly it but only calling the dedicating software unit. This aspect also enforces the encapsulation but for sure it makes more difficult to test. At the same time it might require dynamic memory allocation, calling *malloc* function and recursion algorithm to free memory (figure 3.2b, 3.3).
For explaining the second topic it is possible to consider a specific example: the detection of permanent hardware fault by a software safety mechanism; permanent hardware faults they affect a specific hardware module can involve in software failures, for this reason the achievement of functional safety requires , in addition to error prevention, error detection and mitigation. The problem is the implemented source code shall be tested on a general purpose computer by software unit testing.

(a) Header with pointer to Abstract Data Type

```
 8 #ifndef QUEUE_H_
 9 #define QUEUE_H_
10
11 #include "compAbs.h"
12
13 /*!
14  * Abstract variable
15  */
16 typedef struct QueueType Queue;
17
18 /*!
19  * Pointer to abstract type
20  */
21 typedef Queue *QueuePtr;
22
23 typedef unsigned short sint16;
24
25 QueuePtr InitQueue(sint16 value);
26 void AddNewRecord(sint16 value, QueuePtr *head);
27 void PrintQueue(QueuePtr head);
28 void FreeQueu(QueuePtr head);
29
30 #endif /* QUEUE_H_ */
```

(b) Implementation of software units for management of ADT Queue

```
10 #include "queue.h"
11
12
13 /*!
14  * @brief status definition
15  */
16 typedef enum {
17     Unavailable,  /*!< value cannot be used */
18     Available /*!< value can be used */
19 }StatusType;
20
21 /*!
22  * @brief struct contain a value and a status
23  */
24 struct QueueType {
25     sint16 value;           /*!< data value */
26     StatusType status;      /*!< data status */
27     struct QueueType *next; /*!< next element */
28 };
29
30 /*!
31  * @brief Initialize item
32  *
33  * @param [in] value to insert in queue
34  * @return pointer to queue heaed
35  *
36  * @note status is initialized automtically to 'available'
37  *
38  * @remark: cyclomatic complexity = 2
39  */
40 QueuePtr InitQueue(sint16 value) {
41     QueuePtr first = malloc(sizeof(Queue));
42
43     if(first == ((void*)0)) {
44         return ((void*)0);
45     }
46
47     first->value = value;
48     first->status = Available;
49     first->next = ((void *)0);
50
51     return first;
52 }
```

```
/*!
 * @brief Initialize item
 *
 * @param value [in]
 * @return pointer to queue heaed
 *
 * @note recursive
 *
 * @remark: cyclomatic complexity = 2
 */
void FreeQueu(QueuePtr head) {

    if(head->next == ((void *)0)) {
        head = ((void *)0);

        return;
    }

    FreeQueu(head->next);
}
```

Figure 3.3: Not-compliant C code: Dynamic Memory Allocation (3.2b), recursion for free pointers (actual)

For this reason might be not possible to test **exactly** the code that will be employed on a target system, because that hardware fault never might verify, reducing the obtained test coverage; To perform unit test, adopting a Fault injection method, might be necessary to modify the code, in order to allow the execution of a dummy wrong routine for software

```
FUNC(boolean, PNC_CODE)
PNC_safe_sum(P2VAR(uint8, AUTOMATIC, PNC_MEM) res, VAR(uint8, AUTOMATIC) a, VAR(uint8, AUTOMATIC) b) {
    uint8 Loc_res;
    uint16 Loc_res_dup;
    uint8 checker;
    boolean error;

    /* greater between two operands */
    checker = ((a > b) ? a : b);

    /* compute value */
    Loc_res = (a + b);

    /* duplicate operation i.e. 4a plus 4b */
    Loc_res_dup = ((a << 2u) + (b << 2u));

    /* by math: in a sum result is always greater than the major operand */
    /* second check detects permanent faults in sum module */
    if((Loc_res < checker) || ((Loc_res << 2u) != Loc_res_dup)) {
        /* if Loc_res is lower to max of two operands, then an overflow is occurred */
        *res = 0xffu;
        error = True;
    }
    else {
        /* overflow is not occurred, set result */
        *res = Loc_res;
        error = False;
    }

    return error;
}
```

Figure 3.4: Robust software unit for computation of a SUM to detect overflow and permanent faults

unit that simulates the behaviour of a permanent hardware fault in a certain resource An example is shown in figure 3.4: a dummy unit is obtained replacing consciously the operation with a wrong one in a mirroring unit.
Actually two others important methods can be the usage of particular representation for data organization and software architectural description in order to be uniquely understood: AUTOSAR scheme and UML sequential diagram, class diagram, context one are an example. second can be the definition of coding styles, guidelines and conventions on software data naming. for this last item an example can be a variable to contain a physical value: to avoid possibility of safety gap, the variable name can include the unit measure of data value (e.g. *actual_speed_Fan_radpsec* or *actual_motor_temp_celsius*); another example is related to AUTOSAR RTE code: when AAT generates RTE code, it is a good practise define the name of a port in order to understand if it is a Provider or a Receiver and a Sender-Receiver or a Client-Server or type of data point. for example:

```
Let's consider runnable that has
1) Receive Point (Read);
2) Port XXX (receiver);
3) Interface Data Prototype DDD;
```

The Rte functions to access to that variable might be generated as:

Rte_Read_R*XXX_DDD****(...);***

3.3 Software Architecture

> "The Software Architectural design aims to describe the architectural elements and their interaction in a hierarchical structure" (ISO 26262 chapter 6, clause 7.2)

This is the general definition provided by ISO about the software architecture. A generic architecture for Embedded Software is the stack one. In this manner, each layer is able to manage and implement specific features, properties and functionality, communicating, directly with the components on the same level, using lower layer components services and providing services at higher level. The highest one will be represented by Application layer i.e. implementation of functionalities or better applications with an high level of abstraction by hardware resource. The advantages of this kind of structure are various; first of all the separation among layer to avoid error propagation, second one the possibility to standardize each layer. AUTOSAR is exactly this: a standardized stack for software it runs on ECU.
Following the design flow, software architectural shall define the specific aspects (static and dynamic) that will implement the SSR and the HSI. This last will represent the joining point between software product and hardware one and in particular case, it will specify which services, interfaces and driver shall be configured in AUTOSAR basic software layer. The design of a software architecture shall be done very well in order to conduct the development of software unit as lowest level of implementation. The Software Architectural Design is in general specified in *Software Design Document* or *SDD*. The guidelines for the drafting of this document are provided by ISO 26262 and IEEE 1016 that structures it with an high level of categorization:

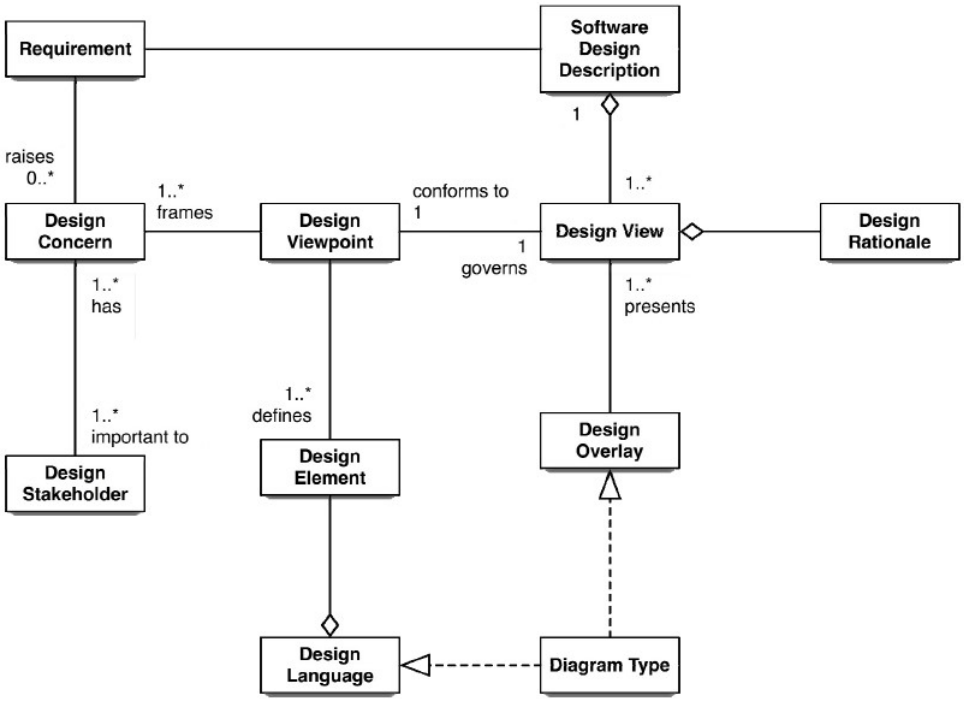


Figure 3.5: Conceptual model of SDD by IEEE 1016 v2009

Let's consider that software architecture still have not really implemented a part of functionality in scope to component but however it shall be designed in order to meet feasibility, maintainability and verifiability (by formal methods and testing). The following description addresses main design choices that have been done in order to meet ISO 26262 for PNC software architecture. Hierarchy design and complexity can be seen as the big two aspect of software architecture design.

Hierarchy : This concept can be bond to several feature. First of all the component shall be developed according to AUTOSAR specification. Therefore the component will be defined according to:

A- Ports for Provide/Receive data to/from other Application SW-C and/or BSW Modules and that implement interfaces with ad-hoc Application Data Types;

B- Internal Behaviour

- Runnable Entities that will become **public** functions;
- Access point for Data, Server, Parameters, modes and so on;
- Events (Time-based for OS scheduler, Operation Invoked and so on);

At Hierarchy scope, it is also necessary to specify the differences between static aspect and dynamic one. Static aspect can be seen as implementation choices. Actually the static code for an ECU software is considered the source code for a SW-C implementation. This can be obtained by hand (i.e. hand-written) or by Model-based approach with the usage of an Embedded Coder. The dynamic aspect for software architectural are related to the execution of software i.e. all aspects that cannot be planned at priori but depend from

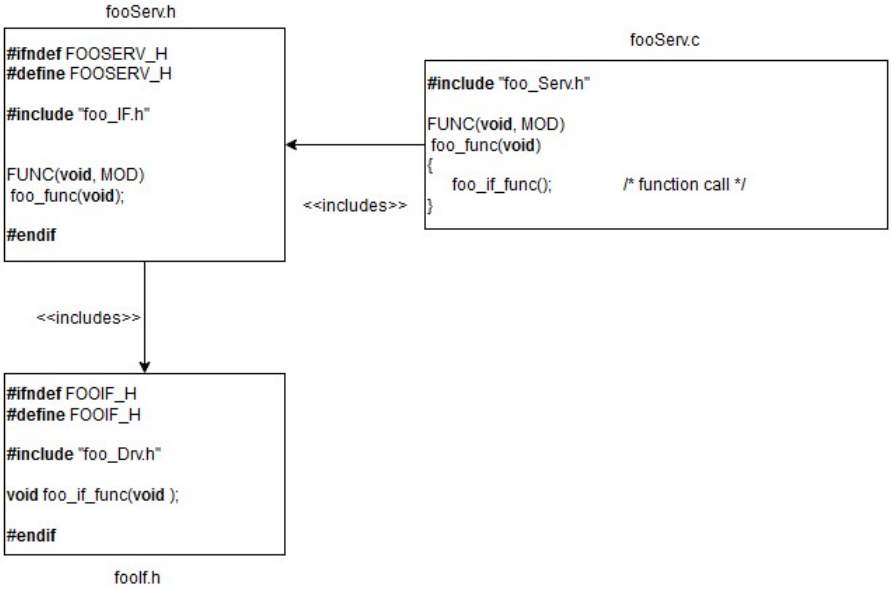
specific run-time case. To dynamic aspect it also related the RTE code because it strongly depends from the configuration of the AUTOSAR stack e.g communication, operating modes, accessing resource, etcetera and it is usually generated by Embedded coder of the AAT, starting from the AUTOSAR schemes (arxml). At the scope of the current paper, software design and implementation will be considered with an higher attention the static aspect of the PNC(manually developed). For this reason it will not be discussed the aspects related to safety of a configurable software that requires the application of Annex C of chapter 6 (ISO 26262 version 2018). Instead safety analysis will be described in detail in the dedicated section (chapter 5) as fundamental activity.
Hierarchy can impact also development aspects as library inclusion: when code is developed in C or C++, the modularity is obtained, defining source files and headers. Generally Headers contains structure, classes, public variable and functions declaration while source ones their definition. A module can be defined as the set of one or more source files. The inclusion of another component library in header or source files of an actual component, actually defines a **dependency**. MISRA C requires caution in header inclusion in order to respect the hierarchical constraints. This means that a component shall include only libraries they can provide services that are implemented at lower layer or at the same layer, if and only if:

- Lower Layer resources are requested by actual component;
- Transversal communication between same-layer components is allowed in specific case;

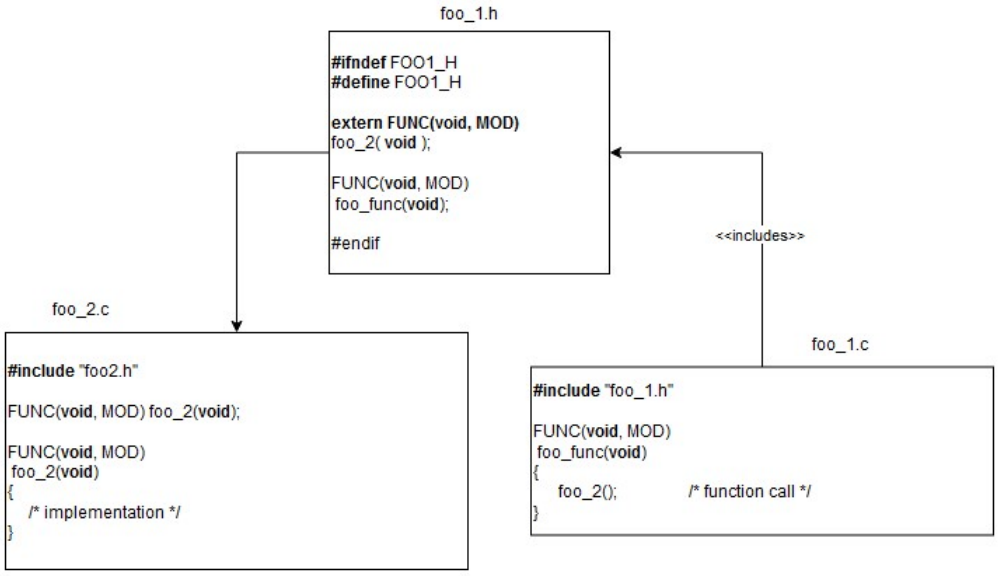
In this way the software architecture prevents the occurrence of Interference Failures among layers of the same one. The inclusion of libraries shall be done in order to avoid a multiple inclusion at least specific rules they allow it and the inclusion of software units, directly in source files bypassing header. The first case means that keyword **include** with the name of a specific header shall be present just once in a source file at least specific case as the Memory mapping for spatial isolation. The second one aspect is about the C keyword **extern**. It can be used to include a software unit in a component that does declare it. MISRA (version 2009) prevents the usage of extern outside a header file. The correct way to implement the previous clauses are described in figures 3.6a and 3.6b (next page)

Complexity and Sizing This paragraph aims to analyse the design of a software architecture that is able to achieve a fine level of complexity and sizing. How it has been specified in the previous section, a good complexity measure for embedded software is the cyclomatic one. For sure it will make easier the control flow analysis of a software unit (and so the complexity is reduced), and reducing in turn the complexity of the component. As collateral effect, it might increase the number of software units and therefore the code size. The number of interfaces might play another important role in the evaluation of a complexity. For example PNC shall manage all basic software services regarding the partial networks in application layer and then it shall notify the result of evaluation to other application software components. For this reason it is not so much possible to reduce the complexity, reducing the size of interfaces. Then the reduction of interfaces sizing cannot be adopted. The high number of calibration parameters increase the static complexity of the architectural design too. What ISO 26262 requires to software architectural design is:

- Capability to be tested, in order to obtain a specific measure of coverage;



(a) Include a unit function from a lower layer



(b) Include a unit function from same layer

- Implementation of safety aspects they shall handle errors;

A not required complexity, due to low modularity or encapsulation etcetera, might have a negative impact on it. Another way to reduce the complexity is to define **groups**. This one have a double value:

1. Prevent the possibility to allocate an asymmetric number of requirements among software units;
2. Enforce cohesion among software components;
3. Prevent the definition of unintended redundancy that is not define at scope of error detection and mitigation;

The first topic can be easily understood in the case of a FSM (as current case). A large number of embedded systems are modelled as FSM because they allow the management of complex scenarios with a certain determinism and reducing the complexity. Usually each state is defined in order to satisfy a requirements or more than one and at implementation level, each state is coded with a software unit that evaluates inputs to identify transitions and produces output, according to state operations. Another view might be deeper related to the development of robust code: let's imagine that more software units shall perform a sum between two variables; the result might be affected by two main errors: an overflow or a permanent faults in hardware module. For this reason every time that a sum is executed a duplicated variable and operation shall be checked with the original one (look figure 3.4). The occurrence of the same piece of code in more units will cause only a code explosion increasing cyclomatic complexity and test cases. Therefore all software unit shall do a safe sum can use another software unit that is able to do. This consideration can be applied at components level too. Strong cohesion between components and loose coupling are two other milestones. The good designing for one of these implies the achievement for the second too. The strong cohesion aims to obtain a strong dependency between data and functions that modify or use them (an example can be *getters and setter* methods for private attributes). Functions within the same component shall have a great level of dependencies in the computation of data, while they must minimize the access to their private data for other component instance and always by the usage of component functions (access point). Aggregation of code statements that implement the same routine in the many software units as a dedicated software unit is a measure of strong cohesion. Instead, loose coupling can be related to the definition of **encapsulation**. This means that a client or a generic software element cannot directly access to the internal record a defining data type as for example C-struct one; it can access by pointer to an hide (or better abstract data type) structure. Each interfacing is allowed only by pointer. This solution is very common for Objected-Oriented Programming as Java and C++ they implement constructor and destructor. For C it is a little different, however it is possible to implement , increasing complexity and pointers usage. Let's remember, this does not allow the usage of functions as *malloc* for pointer memory allocation. Several dummy examples are provided in-line with the architecture of the PNC.
In figure 3.7 *FSMType* and *FSM* are abstract data because actually, they are not defined in header file and so a client cannot know the internal definition of them. The two functions without keyword "extern", represent the public functions that can be called by other

```
 8 #ifndef PRIVATE_H_
 9 #define PRIVATE_H_
10
11 #include "compAbs.h"
12
13 typedef struct FSM_Type FSM;
14
15 FUNCP2VAR(FSM, FSM_CODE, FSM_MEM) InitFSM (VAR(sint16, AUTOMATIC) level, VAR(uint8, AUTOMATIC) start_time);
16 FUNC(void, FSM_CODE) FSM_MainFunc (P2VAR(FSM, AUTOMATIC, FSM_MEM) fsm_actual);
17
18 extern FUNC(sint16, FSM_CODE) getActualLevel (P2VAR(FSM, AUTOMATIC, FSM_MEM) fsm_actual);
19 extern FUNC(uint8, FSM_CODE) getActualTimeStamp (P2VAR(FSM, AUTOMATIC, FSM_MEM) fsm_actual);
20 extern FUNC(uint16, FSM_CODE) getActualStateCode (P2VAR(FSM, AUTOMATIC, FSM_MEM) fsm_actual);
21 extern FUNC(boolean, FSM_CODE) FSM_ErrorOcc (void);
22 extern FUNC(uint16, FSM_CODE) getPreviousState(P2VAR(FSM, AUTOMATIC, FSM_MEM) fsm_actual);
23
24 #endif /* PRIVATE_H_ */
25
```

Figure 3.7: Example of encapsulation of structure *FSM* in header (.h)

```
 8 #include "private.h"
 9
10 typedef enum {
11     S0 = 0x0au, S1, S2, S3, S_Err = 0xFF
12 } State;
13
14 struct FSM_Type {
15     sint16 level_val;
16     uint8 timestamp;
17     State current, previous;
18 };
19
20 static VAR(FSM, AUTOMATIC) fsm_value;
21 VAR(boolean, AUTOMATIC) error_occ;
22 CONST(sint16, AUTOMATIC) prescaler = (sint16)0x0a;
23
24 FUNC(void, FSM_CODE) S0_State_op (void);
25 FUNC(void, FSM_CODE) S1_State_op (void);
26 FUNC(void, FSM_CODE) S2_State_op (void);
27 FUNC(void, FSM_CODE) S3_State_op (void);
28
29 FUNC(sint16, FSM_CODE) getActualLevel (P2VAR(FSM, AUTOMATIC, FSM_MEM) fsm_actual);
30 FUNC(uint8, FSM_CODE) getActualTimeStamp (P2VAR(FSM, AUTOMATIC, FSM_MEM) fsm_actual);
31 FUNC(uint16, FSM_CODE) getActualStateCode (P2VAR(FSM, AUTOMATIC, FSM_MEM) fsm_actual);
32 FUNC(uint16, FSM_CODE) getPreviousState(P2VAR(FSM, AUTOMATIC, FSM_MEM) fsm_actual);
33 FUNC(boolean, FSM_CODE) FSM_ErrorOcc (void);
34
```

Figure 3.8: definition of structure *FSMType* in component source (.c) and private functions

elements, including the actual header. Let's note they work only on pointers of data type as returned value and parameters (Code contains AUTOSAR Macros).
Figure 3.8 shows the definition of the structure *FSMType* with internal fields. All the following functions declarations address private ones, i.e. they can be invoked by current module but cannot by a client (at least they are not declared in header with extern). States are encode with an *enum* type that is also private. No MISRA violation in this definition (PRQA tool analysis according to GM severity definition for MISRA advisories and directives).
Figure 3.9 shows the definition of the initializing functions. It simply updates a FSM variable (static) records and it is returned as a FSM pointer. No MISRA violation is present in this definition according to analysis with the same PRQA tool.
The figure 3.10 is a public function that might be periodically invoked as a task. It ba-

```
35 FUNCP2VAR(FSM, FSM_CODE, FSM_MEM) InitFSM (VAR(sint16, AUTOMATIC) level, VAR(uint8, AUTOMATIC) start_time) {
36     FSM *ret_fsm_ptr;
37
38     error_occ = False;
39
40     fsm_value.level_val = level;
41     fsm_value.timestamp = start_time;
42     fsm_value.current = S0;
43     fsm_value.previous = S0;
44
45     ret_fsm_ptr = &(fsm_value);
46
47     return ret_fsm_ptr;
48 }
49
```

Figure 3.9: Definition of public functions FSM_Init in component source (.c)

```
122 FUNC(void, FSM_CODE) FSM_MainFunc (P2VAR(FSM, AUTOMATIC, FSM_MEM) fsm_actual) {
123
124     switch(fsm_actual->current) {
125         case S0:
126         {
127             S0_State_op();
128             break;
129         }
130         case S1:
131         {
132             S1_State_op();
133             break;
134         }
135         case S2:
136         {
137             S2_State_op();
138             break;
139         }
140         case S3:
141         {
142             S3_State_op();
143             break;
144         }
145         default:
146         {
147             error_occ = True;
148             break;
149         }
150     }
151
152     fsm_actual = &(fsm_value);
153
154 }
```

Figure 3.10: Definition of public functions FSM_Init in component source (.c)

sically evaluate a field of structure and it calls state operations functions. This definition contains some MISRA warnings. However they are considered as warnings with a severity level equal to 2 in a scale from 1 up to 9(The function is not completed because it is not important at discussion scope). This code has been developed only for providing another PNC simile version for evaluation, according to MISRA C, that might improve the aspects of encapsulation, replacing the definition of the structure PNC in header file.
The software architecture shall also evaluate the necessity for **interrupts**. The extended usage of Interrupts has the advantages to save circuity in terms of stress, because only when an information is really present routine is triggered and at the same time it saves

the software to waste resources. However, interrupt usage might have a negative impact on dynamic aspects of the software architecture: first of all it increases the complexity of the architecture due to the interrupt handling (jump to routine, clear pending request, management of priority etcetera), and configuration. Second negative impact is related to scheduling: interrupt requires context switching and if the scheduler is not properly configured to manage a large set of cases it might occur in several missing of deadline. This forces the configuration of a Watchdog that might be not available for each customer target. For this reason PNC uses **polling**: the element shall only check criteria values from a Sensor but that is actually a software sensor. Besides for each time-based transition or evaluation is possible to implement software counters that decrease a constant value based on the periodicity of task execution from initial value which it is initialised, using dedicated software unit. The same consideration can be applied to shared resources: for safety reason it is necessary to implement spatial, temporal and communication separation by safety mechanisms (ad-hoc) and memory mapping. The separation to avoid interference is an intrinsic AUTOSAR features mapping runnable entities in tasks. These last are actually mapped within OS-Application (more tasks can belong to a single OS-Application) that is mapped with a multiplicity 1-to-1 in a partitions [15]. Let's note that how it is specified in AUTOSAR documentation, the partitioning by OS-Application ensures the Freedom From Interference only between partitions and not for tasks that are mapped within the same one. For sure the partitioning requires the assumption in HSI of a MMU (Memory Management Unit).
As last note, ISO 26262 suggests some methods for implementing the safety mechanism and redundancy as multiple storages is one of these. The entire process starts from definition until the implementation of the safety mechanisms by redundancy requires a good evaluation that is generally based on **trade-off**. In fact, they can be expensive methods sometimes both in terms of resources and spent time. A resource evaluation and a Worst Case Execution Time (WCET) identification might be very useful to understand how to define this trade-off, considering as main parameter the ASIL value.

3.4 Software Units Design and Implementation

The current section aims to report a safe implementation in C of software units, in order to satisfy the safety requirement that have been assumed. For the software unit design and implementation it is possible to assert that the entire ISO 26262 methods are totally based on MISRA contents. The real state-of-art of implementation has been obtained, analysing the implemented source code with a specific tool that is called *PRQA* for a quantitative identification of MISRA violations. In beta versions the static code software units have been developed in order to implement the functionality of a State-Flow. This one has been used by Customers (specification designers) to describe the functional behaviour of the PNC. Initially the PNC component contained almost 40 software units (in a set of private and public ones). The safety functions have been designed, in order to meet functional cohesion features and at the same time to respect the features of Basic Software Modules APIs (Application Program Interfaces), according to AUTOSAR specification. The implementation of the safety requirements have requested some changes in existing software units and *ex-novo* definition for others. In the following, only the safety functions implementation will be considered. The definition of safety requirements categories are in section 2.2.

Activation Sensor The activation sensor is a private functions it provides the criteria that shall be evaluated for the activation of a Partial Network. It shall be considered as a software component that provides some information via RTE. Using configuration parameters in AUTOSAR scheme it is possible to configure the number of "sensors" they provide the criteria for a single Partial Network activation. For this reason, according to ISO 26262 requirement 7.12 in chapters 6, it has been chosen to assign a minimal number of 3 "sensors" per PN (lower-bound). Provided criteria are **boolean** values: each sensor provides it value in range of *False* or *True*; the assigned value will be that one will reach the majority level i.e.

$$MajorityValue_{PN} = 0.5 * NumberOfSensor_{PN} + 1$$

This mechanism is based on **forwarding recovery** (figure 3.11) or passive redundancy (there are several papers and academic books they speak about it). The minimal number mitigates the case in which, sensors provide two opposite values for 50% of cases, preventing the possibility to mask the faulty sensors. To be clearer it is possible to see the next example (3.1):

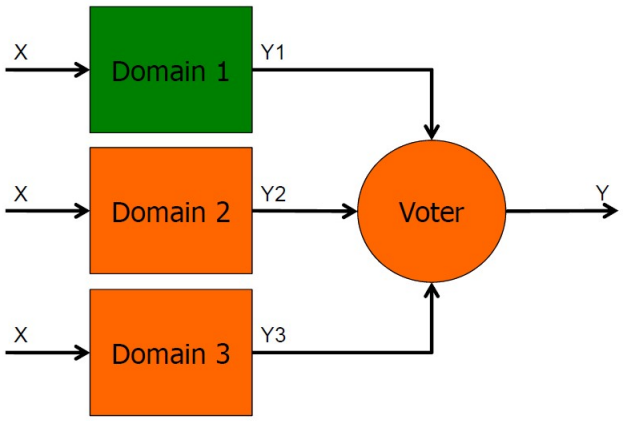


Figure 3.11: Forwarding recovery example (Tri-Module Redundancy in the specific case) [10]

Sensor 1	Sensor 2	Sensor 3	Majority Value (2)
False	False	False	False
False	False	True	False
False	True	False	False
False	True	True	True
True	False	False	False
True	False	True	True
True	True	False	True
True	True	True	True

Table 3.1: Example of error masking using Forwarding Recovery

This method can also work with a larger number of sensors and so the function shall be implemented in order to work with an arbitrary number of them. The majority value has

no problem in the identification of the majority value for the evaluation when the number of sensors is odd. Problems can occur when their number is even because potentially, no one might reaches the majority (example of 4 sensor that provides *True, True, False, False*). For this reason, it has been assumed that if the majority is not reached then value will be assigned to default *False*. This implementation is purely in software, so it is possible to assume that no error occurs in the voter it determines majority result. Let's consider there is no way to detect error, especially if the majority number of sensors are faulty. For sure if the number of sensors is equal to 0 is already possible to set criteria to False. Cyclomatic complexity is less than 10. Code is shown in figures 3.12, 3.13 and 3.14. The PRQA analysis has produced 7 warnings for MISRA rules but with a very low severity (2). Some of them are related to Stub definition for software unit testing.

```
35   /* majority value is the half plus 1u */
36   uint8 majority_value = (((SUM_PNC_Cfg[SUM_PNC_PNC_Number].Number_of_PNC_Activation_Sensors) >> 1u) + 1u);
37
```

Figure 3.12: Definition of Majority value in activation sensor

```
/* this replaced the rte calling */
Activation_Criteria_out = SUM_PNC_Cfg[SUM_PNC_PNC_Number].ptr_PNC_GaaSUM_Activation_sensor_Rte_Call_Interface[Loc_Index];

/*Aggregation of activation criteria for all sensors */
if(Activation_Criteria_out.ActCriteria_local == ((boolean)0x01))
{
   Activation_asserted_to_one ++;
}
else
{
  /* nothing to do */
}
```

Figure 3.13: Classification of sensor value: count the sensor value to TRUE

```
/* assign value to Activation Local Criteria according to majority voting */
  if(Activation_asserted_to_one >= majority_value)
  {
    SUM_PNC_Criteria_Sensor_Act_local[SUM_PNC_PNC_Number] = 1;
  }
  else
  {
    SUM_PNC_Criteria_Sensor_Act_local[SUM_PNC_PNC_Number] = 0;
  }
```

Figure 3.14: evaluation of value that has reach majority value (if does)

Permanent Faults or Computational Errors The computational errors are related to hardware and software aspects. Basically the most common form or computational errors are related to overflow aspects. An example can be the multiplication between two variable on 8 bits they produce a result greater than $0xFF_{Hex}$: let's imagine to have *0x80* times by *0x02* without sign: the result will be 0x100 that cannot be encoded on 8 bits. The specific

case for PNC regards a division. This is used to set the right value of a software counters by pre-scaling that produces an output when reaches the alarm value. For a division it is necessary to check (first of all) that the entity of division is not 0 (math error). If this operand is derived by a configuration parameter it is possible to set its minimum value to 1 in AUTOSAR scheme (figure 3.15). Second aspect is related to presence of a permanent faults in division module. For this reason it is not enough to detect the error but it is also necessary to mitigate it to reduce criticality. The software unit design aims to solve:

```
<ECUC-INTEGER-PARAM-DEF>
    <SHORT-NAME>Dividing_PNC_Timer</SHORT-NAME>
    <DESC>
        <L-2 L="EN"> Prescaler for timing division parameter </L-2>
    </DESC>
    <LOWER-MULTIPLICITY>1</LOWER-MULTIPLICITY>
    <UPPER-MULTIPLICITY>1</UPPER-MULTIPLICITY>
    <SCOPE>LOCAL</SCOPE>
    <IMPLEMENTATION-CONFIG-CLASS>
        <ECU-IMPLEMENTATION-CONFIGURATION-CLASS>
            <CONFIG-CLASS>PRE-COMPILE</CONFIG-CLASS>
            <CONFIG-VARIANT>VARIANT-POST-BUILD</CONFIG-VARIANT>
        </ECU-IMPLEMENTATION-CONFIGURATION-CLASS>
        <ECU-IMPLEMENTATION-CONFIGURATION-CLASS>
            <CONFIG-CLASS>PRE-COMPILE</CONFIG-CLASS>
            <CONFIG-VARIANT>VARIANT-PRE-COMPILE</CONFIG-VARIANT>
        </ECU-IMPLEMENTATION-CONFIGURATION-CLASS>
    </IMPLEMENTATION-CONFIG-CLASS>
    <ORIGIN>AUTOSAR_ECUC</ORIGIN>
    <SYMBOLIC-NAME-VALUE>false</SYMBOLIC-NAME-VALUE>
    <DEFAULT-VALUE>10</DEFAULT-VALUE>
    <MAX>100</MAX>
    <!-- It is used as denominator (safety measure) -->
    <MIN>1</MIN>
<ECU-INTEGER-PARAM-DEF>
```

Figure 3.15: safety mechanism in PNC configuration set by AR scheme

1. **Error Detection:** This is possible duplicating the result variables and checking if they have obtained the same result. The division software unit will return a boolean value to inform the caller if the result is valid or not. The final result will be stored in a variable that is passed by reference to the software unit with the operands. If an error occurred, then result value will be placed to full-scale one, according to type of variable (unsigned short). The redundant checking variable will perform the inverse operation with the local obtained value, in order to check if the original operand value is matched or simply it will perform the operation implementing a parallel algorithm;

2. **Error Mitigation:** This has been done implementing a **Recovery Block System** (figure 3.16), using an array that contains a constant number of function pointers (*&Names_of_Functions*). Each pointer will address an implementation of the same operation with different algorithm. The array allows the selection of a non-faulty software unit by an array identifier, starting from 0 until Number of implementing redundant software units minus one. A switch function will decide which software unit shall be selected, on the base of error status returned by that unit and it returns the result of the operation. If the software units are all faulty, then a global variable is set to True and it shall be evaluated in system architecture before calling again

switch function. Code is in figures 3.17 (declaration of software units and definition of software unit pointer used as array to switch between them by index) and 3.18 (switching mechanism)

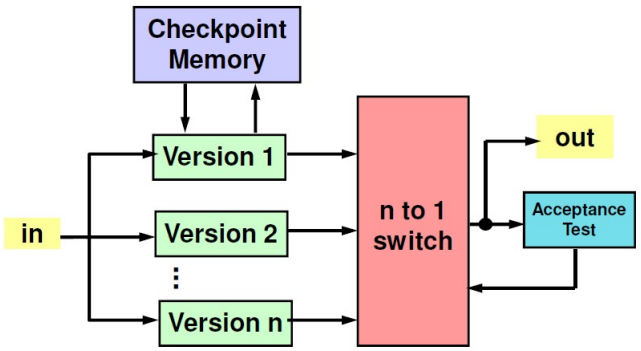


Figure 3.16: Recovery block representation [9]

```
23 typedef P2FUNC(boolean, AUTOMATIC, div_unit_block)
24 (P2VAR(uint16, AUTOMATIC, PNC_MEM) result, VAR(uint16, AUTOMATIC) divided, VAR(uint16, AUTOMATIC) dividing);
25
26 FUNC(boolean, PNC_CODE)
27 PNC_Safe_Div_Mul(P2VAR(uint16, AUTOMATIC, PNC_MEM) result, VAR(uint16, AUTOMATIC) divided, VAR(uint16, AUTOMATIC) dividing);
28
29 FUNC(boolean, PNC_CODE)
30 PNC_Safe_Div_Sub(P2VAR(uint16, AUTOMATIC, PNC_MEM) result, VAR(uint16, AUTOMATIC) divided, VAR(uint16, AUTOMATIC) dividing);
```

Figure 3.17: Division software units API

```
132 FUNC(boolean, PNC_CODE)
133 PNC_Div_BlockSelection (P2VAR(uint16, AUTOMATIC, PNC_MEM) result, VAR(uint16, AUTOMATIC) divided, VAR(uint16, AUTOMATIC) dividing) {
134     static uint8 recoveryBlock_id = 0x00;
135     VAR(boolean, AUTOMATIC) block_faulty;
136
137     block_faulty = PNC_div_units[recoveryBlock_id] (result, divided, dividing);
138
139     if((block_faulty == True) && ((recoveryBlock_id + ((uint8)0x01u)) == NUMBER_DIV_RB)) {
140         DIV_FAULTY |= ((uint8)0x01u);
141     }
142     else if(block_faulty == True) {
143         recoveryBlock_id ++;
144     }
145     else {
146         /* all is correct: nothing to do */
147     }
148
149
150     return block_faulty;
151 }
```

Figure 3.18: Safe Software Unit for division switching (recovery blocks mechanism)

The division operation has been obtained in two ways (let's consider the operation $C = A/B$)

1. The first method consist in the evaluation of the reminder between A and multiplication of B by an increasing value that starts from 0. When the reminder is less than

B, the quotient has been reached. The value will be checked performing the classic division operation, i.e. invoking /. The obtained results shall match themselves. The definition by a code point of view can be observed in figure 3.19

2. The second method considers the basic definition of the division: it is an iterative subtraction from A by B while the results is greater than B. The counting of the iteration will represent the quotient (hardware-based algorithm). The value will be checked considering the local quotient as *loc_q* and applying the following formula:

$$A == ((B * loc\_q) + Module(A, B)))$$

The definition by a code point of view can be observed in figure 3.20

The software units should be designed and implemented in order to have many similar features as for example WCET (at least by an initial analysis) and structure in terms of input variable and returned type (this does not mean that is a homogeneous redundancy). The features have been considered as similar maintaining the same complexity in terms of number of iterations and cyclomatic one. Besides to avoid the possibility of error in the checking operations, the validity of the result is done using two different methods. For both methods the cyclomatic complexity is *6*. Only 2 PRQA warnings have been produced with low severity (2).

Control Flow Actually this has been the last implemented safety mechanisms. It aims to detect run-time errors due to unexpected behaviours. This can be very useful when the software is implemented, to meet the definition of a state-flow model where at each input will respond transition to another state. The actual control flow mechanism has been done completely in software static code. AUTOSAR architecture allows the configuration of a watchdog to perform the Logic monitoring i.e. the control flow. ETAS develops the PNC from Application layer until Service Layer (Basic software) therefore potentially at least the Watchdog Manager could be configured. The reason why the control flow has been done in a static way is related to the very strict specifications of requirements that are provided by customer. They leave an high level of freedom for the implementation of static code but very low in terms of AUTOSAR Basic software configuration. This because the PNC shall run on not-ETAS Hardware platform (ECU Abstraction Layer and Micro-Controller Abstraction Layer). Therefore it is necessary to be aligned on the work products. However a test for configuring the watchdog manager, defining parameters and providing values for description (AUTOSAR schemes) has been done to evaluate the RTE code generation and the possibility to use the *11* Watchdog APIs but it is not reported at the scope of this paper.
For static implementation of control flow, a **ECCA** approach has been used. It consists in the definition of an unique signature for each Basic Block of Control Flow Graph (i.e. code blocks within a software unit that does not contains branches); this one is then asserted in order to verify that the actual control block has been reached by a legal one. At the moment, the control flow is implemented only at states level i.e. it controls just if the actual state has been reached by a legal transition with respect to the previous state. If not, nothing is executed, outputs are invalidated and the FSM is forced to reach as next state the safe one. The state signature is constant and unique for each state: it is a number that encodes each state that can reach the actual, value *1*, and a *0* otherwise. The signature will contain many bits as the number of states are (fixed). This control word

```
FUNC(boolean, PNC_CODE)
PNC_Safe_Div_Mul(P2VAR(uint16, AUTOMATIC, PNC_MEM) result, VAR(uint16, AUTOMATIC) divided, VAR(uint16, AUTOMATIC) dividing) {
    boolean error = False;
    boolean predicted;
    uint16 act_result;
    uint16 check_act_result;
    uint16 inc = 0x0;

    /* check if the division result can be predicted at priori */
    if(divided < dividing) {
        act_result = 0x0;
        predicted = True;
    }
    else if(dividing == ((uint16)0x0)){
        /* dividing is 0, set an error occurrence and full-scale as result */
        act_result = 0xffffu;
        error = True;
        predicted = True;
    }
    else {
        /* division result cannot be predicted at priori and requires computation */
        act_result = divided;
        predicted = False;
    }

    /* main loop: multiply dividing for increasing value until reach divided */
    while((predicted == False) && (act_result >= dividing)) {
        inc ++;
        act_result = (divided - (dividing * inc));
    }

    /* if result could not be predicted validate the computation */
    if(predicted == False) {
        /* redundant computation (heterogeneous) */
        check_act_result = (divided / dividing);

        if(check_act_result == inc) {
            /* computation has been validated */
            *result = inc;
        }
        else {
            /* an error is occurred during computation (possible permanent fault in HW) */
            *result = 0xffffu;
            error = True;
        }
    }
    else {
        /* result was predicted: assign to global result */
        *result = act_result;
    }

    return error;
}
```

Figure 3.19: Safety Div method 1

shall be defined within state software unit by developer and it requires to be redefined at each changes in the state-flow model. The encoding let's consider that PNC states are defined with an *enum* type starting from *0x00*. The State encoded with 0x00 (first one) will be the most significant bit in the signature: if state 0x00 can reach the actual state then the most significant bit will be 1, otherwise it will be 0. This will be done also for the current state itself according with the state-flow. At this point another software unit will generate a dynamic mask according to the value of the previous PNC state: it is done shifting on left a 1 for each position as much as the value of the previous state: for example if there are 4 states (*S0 = 0, S1 = 1, S2 = 2, S3 = 3*), and the previous state is *S0*, then the mask will be generated shifting a *1* by

$$NumberOfStates - (Previous + 1)$$

i.e. *3* and obtaining a control word on binary encoding 1000. Once that dynamic mask has been generated it shall be compared with the actual signature. Code for mask generation is in figure 3.23. Let's imagine that the state-flow is represented in figure 3.21 and that the actual state is *S1*
Then the actual signature will be the following

$$ControlFlowSignature_{S1}(S0, S1, S2, S3) = 1011_{bin} = 0x0B_{hex} \tag{3.1}$$

```
64⊕FUNC(boolean, PNC_CODE)
65 PNC_Safe_Div_Sub(P2VAR(uint16, AUTOMATIC, PNC_MEM) result, VAR(uint16, AUTOMATIC) divided, VAR(uint16, AUTOMATIC) dividing) {
66     boolean error = False;
67     boolean predicted;
68     uint16 act_result;
69     uint16 check_act_result;
70     uint16 dec = 0x0;
71
72     /* check if the division result can be predicted at priori */
73     if(divided < dividing) {
74         act_result = 0x0;
75         predicted = True;
76     }
77     else if(dividing == ((uint16)0x0)){
78         /* dividing is 0, set an error occurrence and full-scale as result */
79         act_result = 0xffffu;
80         error = True;
81         predicted = True;
82     }
83     else {
84         /* division result cannot be predicted at priori and requires computation */
85         act_result = divided;
86         predicted = False;
87     }
88
89     /* main loop: decrease divided by dividing until the first remains greater or equal than second */
90     while((predicted == False) && (act_result >= dividing)) {
91         act_result -= dividing;
92         dec ++;
93     }
94
95     /* if result could not be predicted validate the computation */
96     if(predicted == False) {
97         /* redundant computation (heterogeneous) */
98         check_act_result = (divided % dividing) + (dividing * dec);
99
100        if(check_act_result == divided) {
101            /* computation has been validated */
102            *result = dec;
103        }
104        else {
105            /* an error is occurred during computation (possible permanent fault in HW) */
106            *result = 0xffffu;
107            error = True;
108        }
109    }
110    else {
111        /* result was predicted: assign to global result */
112        *result = act_result;
113    }
114
115    return error;
116 }
```

Figure 3.20: Safety Div method 2

The actual checking will be performed by an **bitwise and** between the state constant signature and the dynamic mask. Let's imagine that mask is the same of the previous example and that control flow signature is the just computed one. The result will be:

$$ControlFlowSignature_{S1} \cdot mask_{S0} = 1011 \cdot 1000 = 1000$$

In the previous example is possible to see that if the state transition is permitted, then the checking result will produce exactly the mask value. In fact in figure 3.21, it is possible to see that transition $S0 -- > S1$ actually exists. In the case where transition is not allowed (for instance $S0 -- > S3$) the situation will be the following:

$$ControlFlowSignature_{S3} \cdot mask_{S0} = 0111 \cdot 1000 = 0000$$

This means that transition is illegal. The safe state in the PNC case, it is defined as a global restoring of PNC values as an initialization function. Safe State does not perform any kind of checking for the moment but a good idea for future might be to allow it to communicate with a diagnostic component by API, to understand why this violation has occurred. Safe State can be reached by any state less the initial one and it can reach only the initial state. No MISRA errors or warnings have been produced by generation mask software unit according to GM requirements. An example of employment of control flow is

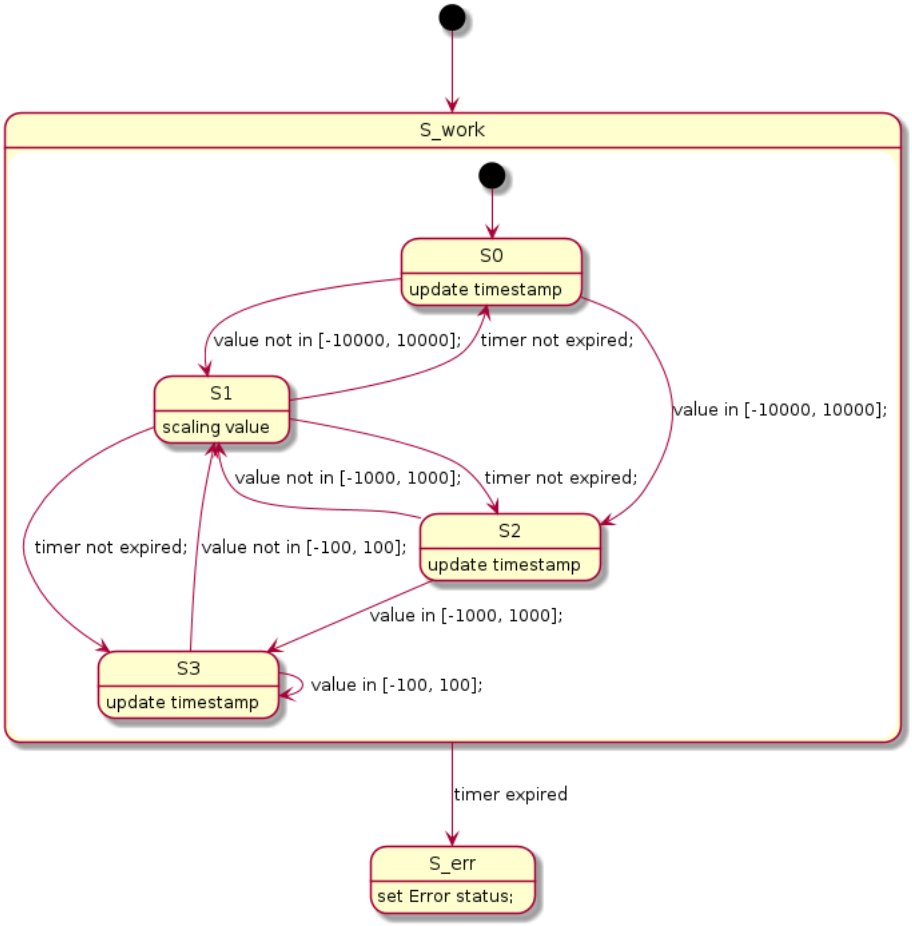


Figure 3.21: example of state-flow for control flow checking

done with respect to software architecture description about encapsulation. In particular in figure 3.24 is shown the definition of private function for *S1* state operation checking the previous transition. In figure it is assumed that error state has code 0xFF and it is reached in case where the transition is not legal, otherwise normal state operation setting output value is performed. This example has been used to evaluate the effect that is might be have on PNC code. All MISRA violation that have been obtained (warning with at maximum severity equals to 6) have been related to Stub nature of the component.

```
10 CONST(uint8, AUTOMATIC) Number_of_State = 0x04u;
11
12 FUNC(uint8, PNC_CODE) GenerateModeMask (VAR(StateType, AUTOMATIC) previousMode);
13 FUNC(boolean, PNC_CODE) ControlFlow_Checking (VAR(StateType, AUTOMATIC) previousMode);
14
```

Figure 3.22: Definition of private global variable number of state

Freedom From Interference has been actually met only implementing the PNC with the highest ASIL value and respecting the AUTOSAR specification for the software component implementation in application layer i.e. mapping the runnable entities in task in order to give them a specific partition in memory. Besides each critical resource as timer has been developed in order to be private in PNC source code component.

```
27⊖FUNC(uint8, PNC_CODE) GenerateModeMask (VAR(StateType, AUTOMATIC) previousMode) {
28      uint8 mask;
29      uint8 startSht;
30
31      if(((uint8)previousMode) <= Number_of_State) {
32          startSht = (Number_of_State - ((uint8)0x1u)) - ((uint8)previousMode);
33          mask = ((uint8)0x1u) << startSht;
34      }
35      else {
36          mask = (uint8)0x00;
37      }
38
39      return mask;
40  }
```

Figure 3.23: Generation of actual mask software unit (cyclomatic complexity is 2)

```
74⊖FUNC(void, FSM_CODE) S1_State_op (void) {
75      VAR(uint8, AUTOMATIC) ts_backup = fsm_value.timestamp;
76      CONST(uint8, AUTOMATIC) s0_cf_signature = 0x0b;
77      VAR(uint8, AUTOMATIC) cf_gen_mask;
78
79      cf_gen_mask = GenerateModeMask(fsm_value.previous);
80
81      if((s0_cf_signature & cf_gen_mask) == (uint8)0x00) {
82          fsm_value.current = S_Err;
83          fsm_value.timestamp += (uint8)0x0au;
84      }
85      else
86      {
87          fsm_value.level_val = (fsm_value.level_val / prescaler);
88          fsm_value.timestamp += (uint8)0x0au;
89
90          /* startSSR-17 */
91          if(fsm_value.timestamp <= ts_backup) {
92              fsm_value.current = S_Err;
93          } else {
94              fsm_value.current = fsm_value.previous;
95          }
96          /* endSSR-17 */
97      }
98
99      fsm_value.previous = S1;
100 }
```

Figure 3.24: Employment of control flow checking by dynamic mask and static signature in stub FSM

Chapter 4

Verification of implementation by analysis and formal methods

4.1 Verification Plan

Verification is a process aims to prove if **implementation is consistent with the specification**. Its can be done by formal methods and testing activities (look V-model). How it has been specified in the introduction, the testing phase is already a way to increase the safety of the product. The real problem is in its feasibility: for testing a product, there is the necessity to allocate resources,to spend time and to adopt strategies to make it feasible. In some cases testing phase for a product can be longer that the development one. The testing methodology shall be appropriated to the target system: testing hardware is different to test software; at the same time, test methodologies can be different also for testing same dominion, as in the case of model-based software design. In this case the verification plan might include methodologies to verify the model and generated code, running the development part in a target system. For it is possible to use *"in-the-loop"* methodologies (Model-in-the-loop, software-in-the-loop, hardware-in-the-loop and process-in-the-loop) applying testing method Back-to-Back. The verification methods are identified according to ASIL value too. for clauses 6-6 (*"Specification of Software Safety Requirements"*), 6-7 (*"Software Architectural Design"*) and 6-8 (*"Software Unit Design and Implementation"*), ISO lists several methodologies to verify the implementation and the specification; this phase should reduce the number of bugs that will be discovered by testing. In ETAS work products, Verification plan is contained inside the safety plan as an unique document. The main contents are the following:

A. mapping of ISO 26262 methodologies to apply for Software Component Verification;

- *a.1* - Requirements traceability;
- *a.3* - Testing methodology and Test case derivation one;

B. Time plan for verification processes;

At the moment ETAS is executing some tests also at system level, including software modules that are developed by Company itself with the employment of several own tools, to simulate a virtual ECU. Let's consider that Partial Networking Coordinator is AUTOSAR compliant. This means the software component will be developed according to a system

description and ECU description using AR-Schema. How it has been explained in software Unit Implementation and Design, the development of this component it is made by static and dynamic code. This last is generated using ETAS tool ISOLAR-A with a RTE code generator and it is strictly dependent from Basic Software configurations. To explain in a better way, the definition and the configuration of basic software
ISOLAR-A is certified by ISO 26262, therefore generated code can be considered as safe, but the artefact might be an error source because it has been written according to requirement specification. So Verification Plan shall consider also AUTOSAR verification requirements and a way to perform ISO 26262 verification methodologies on *.arxml* schemas.

4.2 Verification of requirements: Traceability

The traceability is a method aims to verify the development process by SSR implementation. Its scope is to create a connection or (better) a network of links (as a graph of links) "for measuring the relationship degree among predecessor and successor processes" [11] in hierarchical designs, considering their work products. Let's imagine the specification of SSR: they are derived by TSR once that design phase moves itself from product development at system level to product development at software level; if TSR have been not identified, it is not possible to define SSR. In this case the system level specification is the predecessor phase of Software level one and its work products, shall be used as input for the successor phase in turn. Traceability *shall prove consequentiality of these processes.* This mechanism can be implemented according to ISO 26262 methods of Table 2 in chapter 6 where there are two methods marked as mandatory(++) for ASIL D. The important thing, leaving aside which method will be adopted, is that from each node of this network is always possible to find a path until its *root* (in general the **Safety Goal**) and its *leaf* (in general the **software units and test scripts**). This case defines the *bidirectional traceability.* Let's consider that when a project is developed according to ISO 26262 series of standard, the classes of requirements and associating specification (considering only safety part of a design), that shall be traced, are not less than the following listed ones:

A - Functional Safety and Non-Functional Safety Requirements or Assumptions(in case of SEooC);

B - Technical Safety Requirements;

- **b.1 -** Hardware Safety Requirements;
- **b.2 -** Software Safety Requirements;
- **b.3 -** Hardware-Software Interfacing specification;

C - Verification Unit specification;

The traceability shall be able to trace at least each one of the previous topic and it shall be able to do that, among different types of work products: let's remember that work products include specification and implementation; in the case of software product, the specification can be considered as the documentation, while implementation consists in source code, libraries, scripts and AR-schemes. The different nature of these work products shall not be an obstacle by the adopted traceability method. This last can be implemented in form of

documentation or interactive tool as a program; indeed it also permits the automatic generation of tacking documentation using models, so long as the qualification of the adopted tool has been validated according to ISO 26262 requirements. Within AUTOSAR specification is possible to find a graphical way to implement the traceability of requirements. This one is directly obtained from ISO 26262 schemas but it can be useful only to track from Functional Safety Requirements until Software Safety Requirements(figure 4.1).

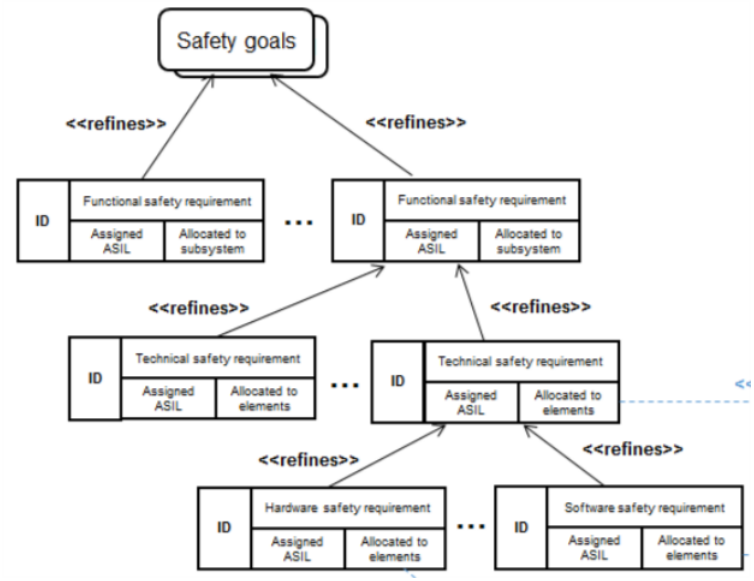


Figure 4.1: AUTOSAR scheme for requirements tracking (Specification of safety extension)

The big advantages of this kind of representation is the clarity, the understandability and the readability. The big issues can be due to the number of requirements to be traced: let's imagine that a safety analysis identifies 5 **Safety Goal**. Each one contains at least three **FSR**. Let's remember the multiplicity between **FSR** and **TSR** is not strictly 1-to-1. In the case, whether it will be applied to the development of a Safety Element out-of-Context, the tree might explode, considering all assumptions that have been made at system level to derive the **TSR** and all assumptions on software component, again considering its Safety Goal with respect to the system ones and its **SSR**. Let's consider that traceability method shall be chosen during the first phases of the system development and it will play a consistent role in the quality of design and for sure, it cannot be a bottleneck in terms of time. Graphical method for traceability might be too long for implementation and to be maintained. Therefore the graphical tree can be excluded. At the same time, also the traceability that is implemented by Data Base can be excluded because it might require too much time to be designed and no each stakeholders might have the right level of skills/knowledges for using a DBMS. For sure this solution is not always to discharged because it might require time for the first implementation but it allows a good maintainability and it allows to structure complex relationships.
ISO 26262 requires *bidirectional traceability* for high ASIL value. The bidirectional traceability is well-defined in "Bidirectional Requirements Traceability" paper by Linda Westfall [11], where it is defined the union of the following tracking approaches :

- Forward tracking;
- Backward tracking;

The first one is used to *track the health of the product evolution* or it is possible to say it *evaluates the rightness of a product development with respect to theory sources*. Forward tracking is very useful when it is necessary to develop a product it shall respond to specific theoretical aspects as for example mathematical models, electronic systems, physic applications and business model, following a **top-down** approach. Forward tracking can evaluate the impact of a source change on the whole design. For this reason, it is perfectly compliant with the development of a product according to ISO 26262 standard. The second one (Backward tracking) is useful to ensure the consistent implementation of requirements. In particular it can detect missing in implementation and prevention of the gold plating i.e. *implementation of something it does not belong to product specification*. Due to the nature of the SEooC, there might be a significant possibility to implement something that has not been considered as a tailored activity and consecutively to miss its documentation; the common effect of a missing specification can be the lack of test cases to ensure the rightness. If a failure is nested within untracked requirement in source code, it will be very difficult to be discovered. This will have a bad impact on safety achievement and in some scenario, missing implementation or wrong one, can be discovered only once it will cause a damage in a vehicle-context with the possibility of major injuries. Therefore the backward tracking will be very useful in many contexts, including the development of a SEooC based on assumption and tailored activities that *should* (actually *must*) be documented. The backward tracking uses a **bottom-up** approach.
A common method for traceability implementation is the **Traceability Matrix**; it identifies the main work products and their associated requirements type. For this is necessary to define an unique key that is able to identify a requirement type. In general they are:

- SRS : Software Requirements Specification;
- SDS : Software Design Specification;
- UTS : Unit Test Specification (which unit is tested);
- STS : Software Test Specification (Test Cases);
- Assumption : for Safety manual in case of SEooC;

In some cases it is possible to track Functional Requirements and Safety ones in different manners. This can be very useful, especially when functional specification is provided by a customer, while safety-related work products are required *in toto* to service and application provider. This separation can be very useful in the case where ISO safety-life cycle is not applied to the first product version but on beta ones. Besides the parallelism can improve the maintainability of the functional part on one side and the definition and implementation of the safety part on the other one. The result will be something of similar to figure 4.2
In the specific case, the functional block will aim to track GM requirements in the design specification and implementation, while Safety part will be tracked by ETAS after the assumptions on Safety. How it has been said in previous, the nature or format of work products shall not be an obstacle for traceability. The nature of ETAS software components will wander from documentation (LaTex, html, word, markdown, excel) until source codes (AR-scheme, .c, .h, .cpp, .xpt, .ext, ecc...). For sure, Static and Dynamic code, with relative models, represent the most interesting part for an ETAS's customer. Therefore an idea is to find a way to provide the most of the documentation already embedded within the code. The main advantages to reach this aspect are:

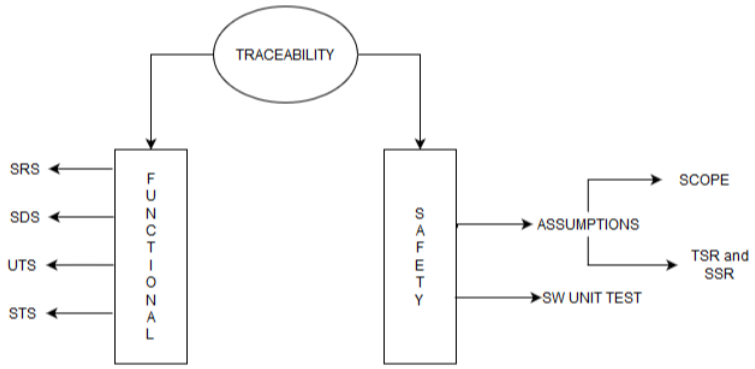


Figure 4.2: Parallel Traceability

A. Better portability of the work products;

B. Better understandability of code;

C. Good maintainability and identification of requirements implementation;

The main idea content wants to specify the requirements in code with particular syntax and semantic, that can be used to generate a portable documentation, with all the requiring information for a complete traceability; the generated documentation is linked to all supporting specification and implementation. In this way each time a requirement is changed within the specification and in turn its implementation requires different statement or a new software unit, it will be enough to change the identifier structure in code and to run again a generator for updating. The traceability Matrix is directly implemented in code documentation, avoiding the necessity to keep it in a separate document and solving its natural trouble about the maintainability. Another important one is the granularity: The matrix considers stakeholder and requirements, that are derived by theoretical aspects, as input keys and it tracks at work product level. To make an example, let's consider an ABS case (totally dummy by technical point of view):

```
Math Model
```

$$F(t) = K * m * \frac{dv(t)}{dt} * \mu(s) \tag{4.1}$$

$$s = 1 - R * \frac{\omega}{v} \tag{4.2}$$

$$M_{\omega} = T - RF \tag{4.3}$$

```
[SRS_00]:
The Braking Force to be
applied on wheel, shall be computed
on the base of:
1) actual speed;
2) Vehicle mass
3) Wheel Radius
```

It is possible to see how SRS is directly derived by Math source; for traceability matrix they will be an instance of identifier key (two attributes, one key). At this point, for these entries, matrix will consider the High-level system architectural point of view defining HSI, and the relative requirements (TSR or directly SSR as Low-Level design aspect):

```
High Level Design:
[4.1 Speed Meter]
[4.4 Pressure Pump]

Low Level Design:
[3.7 Constant Parameters]
[4.1.5 Read Speed API]
[4.2.2 Pressure Actuator API]
```

Until this point the traceability matrix is able to consider aspects in its completeness. After the identification of the API is necessary to track the software unit that is able to define the API declaration. This tracking will be strongly dependent from the implementation choices. Due to matrix structure, the tracking at unit level becomes difficult, especially if external facilities, as third-part libraries are used. Many times, a specific functionality is implemented by a specific software unit (that is a good approach, strongly recommended for embedded software) while software unit are being tracked without any reference about code lines or directly piece of code but just with component name that contains them. This can be a great trouble, especially speaking about Automotive code since it is able to reach ten thousands of LoC for single software component.
This aspect enforces "tracking-in-the-code" method. To implement it in a very useful manner, it is possible to adopt also a second method for traceability that is used in Backward tracking and that can be merged with Traceability Matrix, to enforce the portability, availability and readability: the **Tagging**.
This method is originally used for traceability in the documentation, referencing the requirements and assumptions specification by an own unique identifiers called **tag**. To make an example, remaining on the previous ABS case, the Tagging uses to implement tables within documentation with the following pattern:

Tag	Description	Source
SDD_05	The speed measurement is possible at software level calling *ReadActSpeed()* function by API	SRS_00
SDD_06	Vehicle constants as wheel radius are available as configuration parameter on 32-unsigned bits	SRS_00

Table 4.1: example of tagging traceability structure

In this way is easy to track back specification about SRS for evaluation of missing or gold plating. For the scope of embedded Traceability documentation is possible to merge Tagging with Traceability Matrix i.e. using a matrix to contain Tags of requirements. Once it has been done, when documentation is generated, it will contain a series of link to the requirement specification according to their own and unique tag; This link will help providing an interaction with the final user that can refer requirement specification directly

clicking on it, while it is reading code implementation. Note that this allows a certain level of dynamism in the sense that:

- **If the content of a requirement changes** then it is enough to keep the tag to maintain the trace in source code with the updating content;
- **if the implementation changes** is only necessary to move the tag trace to the new one respecting the semantic and syntax and generated again the documentation using the specific tool;

A similar solution has been adopted by Matlab&Simulink too: in Smulink it is possible to define a model, for example a state-flow diagram, and to generate its static code (i.e. that implements only the model in self) and dynamic, specifying a target system as a board or simply a processor type (ARM, Intel, ecc...). The embedded coder in Simulink, at the end of generation phase, will provide source code and its related documentation. This one will contain a part for requirement specification and another one for its traceability. In this case the documentation is provided in *html* format to allow hyperlinking among pages. an example of final traceability specification, obtained by Simulink embedded coder is in figure 4.3.

Object Name	Code Location
State 'initial_state' <S1>:4	Chart.c:82, 83, 86, 88, 91, 92, 93, 94, 95, 96, 97, 100, 107, 108, 111
State 'operating_mode' <S1>:6	Chart.c:93, 94, 95, 96, 97, 100, 104, 105, 106, 107, 108, 111, 113, 114 149, 150, 155, 156, 157, 158, 159, 161, 162, 163, 164
State 'Left' <S1>:10	Chart.c:94, 95, 106, 113, 114, 115, 116, 117, 119, 120, 121, 122, 127,
State 'Left_off' <S1>:7	Chart.c:94, 95, 114, 115, 116, 117, 119, 120, 121, 122, 129, 130, 131
State 'Left_on' <S1>:9	Chart.c:106, 120, 121, 122, 127, 128, 129, 130, 131, 133, 134, 135

Figure 4.3: Simulink traceability

How it can be seen, the objects are tracked within code in terms of lines. Objects represent each model construct as state, sub-state and transitions. The idea of "traceability-in-the-code" is something of similar to simulink results. This method can be implemented using an open source application called **Doxygen**. Basically this one allows the documentation generation, reading comments in source code with a specific format that can be customized thanks to the writing of configuration file with specific keyword. The usage of this tool is widely thanks to the supporting of the main programming languages as C, C++ with Qt extension, Java (Although for it is preferable Javadoc), Python, VHDL, C# and others. The produced documentation can embed html syntax or markdown. The end format can be chosen by several extensions, the main three are UNIX man, html and LaTex. At the same time, the documentation can also contain diagrams as UML class diagram, sequence diagrams and state-flow using Graphviz and plantUML syntax in source code comments and a good way to define hyperlinking through supporting documentation as the original idea foresaw (look figure 4.4).
ETAS code generator for RTE already comments the code with Doxygen comment styles, while for static code this has been done manually. Each software unit has been documented at least for general information like the data structures. The documentation has been generated in html and latex version considering the cross-reference to source codes. Each AR-scheme or script contains requirement implementation has been previously converted in a html page and then linked to Doxygen generated documentation. Let's consider that

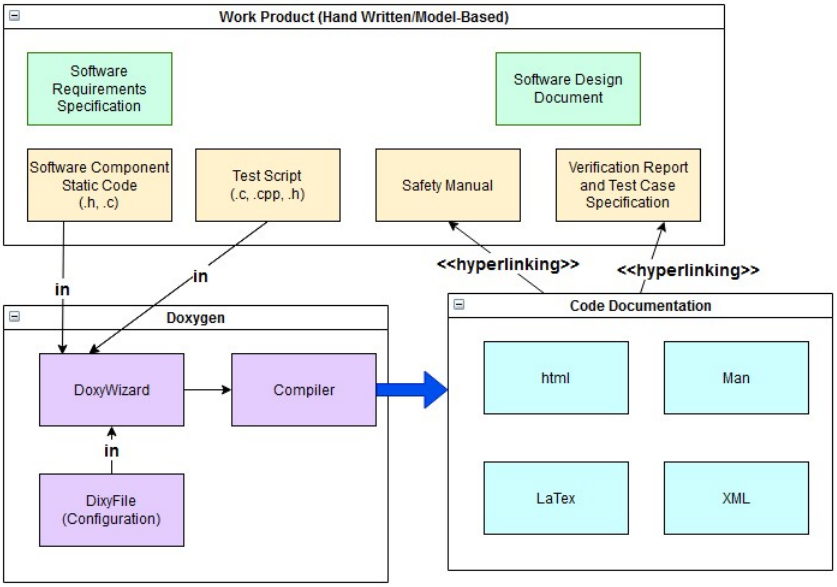


Figure 4.4: Doxygen workflow for current case

AR C code it is developed considering AR compiler abstraction *Specification of Compiler Abstraction* [12] and therefore it might be different by "common" C code that is written for general purpose application or for some embedded systems' firmwares. To make an example, it is possible to see figure 4.5 where an AR compliant code is shown. This is

```
1  #ifndef SPI_H
2  #define SPI_H
3
4  #include "Compiler_Cfg.h"
5
6  typedef enum {
7      SPI_CLK = 0, /* Serial Clock */
8      SPI_MOSI,     /* Master-Out, Slave-In */
9      SPI_MISO,     /* Master-In, Slave-Out*/
10     SPI_SS        /* Slave Select (Active Low) */
11 } Spi_ChannelType;
12
13 typedef uint8 Spi_DataType;          /* 0-255 */
14 typedef uint8 Spi_NumberOfDataType; /* 0-255 */
15
16 FUNC(Std_ReturnType, SPI_CODE) Spi_Buffer (
17     Spi_ChannelType Channel,
18     /* Type of SPI channel */
19     P2CONST(Spi_DataType, AUTOMATIC, SPI_APPL_DATA) *SrcDataBufferPtr,
20     /* Constant pointer to memory address of source buffer */
21     P2VAR( Spi_DataType, AUTOMATIC, SPI_APPL_DATA)  *DesDataBufferPtr,
22     /* Pointer to memory address of destination buffer  */
23     Spi_NumberOfDataType Length
24     /* Number of 8-bits element in buffer (originally SPI is 1) */
25 );
26
27 #endif
```

Figure 4.5: SPI API AR-compliant code

an important aspect to consider, especially when a Company develops memory mapping,

module source files and documents for a customer. The previous work products shall be considered as a single package that shall contain in addition the Compiler_Cfg.h API; it is a very important element for an integrator. The native C API code, with respect *Spi_Buffer* in code 4.5, will be something of similar to code in figure 4.6.

```
16 Std_ReturnType Spi_Buffer (
17     Spi_ChannelType Channel,
18     /* Type of SPI channel */
19     const Spi_DataType *SrcDataBufferPtr,
20     /* Constant pointer to memory address of source buffer */
21     Spi_DataType       *DesDataBufferPtr,
22     /* Pointer to memory address of destination buffer  */
23     Spi_NumberOfDataType Length
24     /* Number of 8-bits element in buffer (originally SPI is 1) */
25 );
```

Figure 4.6: SPI API AR-compliant code

Just to remark, it is possible to see in AR-compliant code the definition of Compiler_Cfg.h. Let's consider that this is only an example that has been in part extracted by another one AR specification and so it shall not be considered as exhaustive.
Let's consider that Doxygen is not able to recognize AR-compliant C code by default; due to this is necessary to operate on Doxygen configuration file, in order to document the code in clearer manner; in the specific case it has been chosen to document C code with the syntax that can be compiled by *GNU-GCC* compiler. In the following will be analysed a safety function, implemented in SUM PNC, with the necessary comments for the generation of the documentation.
The configuration file shall be written in order to consider software safety requirements that are globally implemented by a Software Unit and those are implemented by specific statements within the same Software Unit. Besides the generated documentation shall describe the main features of a single unit according to safety manual and software design document (SDD) for the functional part. To meet all requirements, the adopted solution has been the following:

A) The functional specification about software Units implementation are described within **SDD**

- The design and implementation of software units are always done according to ISO 26262 high recommended methods;
- The traceability of functional requirements is done by tracking tag insertion in each interested work product;
- A specific ETAS *"home-made"* tool is able to parse work-products searching tags and generating traceability documentation (bidirectional);

B) The Safety specifications are represented, from Concept phase work-products until SSR (chapters 3, 4, 6), by **Safety Manual**

- The design and implementation of software units shall be consistent with SDD and safety manual;
- the safe states and Safety mechanisms are specified within SSR and allocated to software Units;

- Each SSR refines at least one Component Safety Goal. These are in turn derived by higher level safety assumptions (equivalence to TSR);
- Each Assumption, goal and requirement is defined by a linkable unique identifier (tag) for internal document browsing and external hyperlinking;
- tags are inserted in source code comment, using markdown, to describe them in a matrix format. Each tag is defined with a customized Doxygen command that is configured in order to link itself to specification in safety manual.

To reduce the side effect of the overhead size for comments, due to the complex structure they shall represent in final documentation, it is possible to insert the functional description in Header file before unit declaration (i.e. comment is associated to API) and the safety traceability matrix before unit definition. This is a necessary operating mode due to Doxygen capacity up to 99999 lines of code that can be parsed for each component. An example of functional description as comment in API is shown in figure 4.7.

```
123 /**
124  * @brief Updates the Activation_criteria i.e True/False
125  *
126  * ## ETAS Confidential
127  *
128  * **FUNCTIONAL DES**: the actual function implements the reading from activation
    sensors to set
129  *                  local activation criteria
130  *
131  * @param[in]     None
132  * @param[out]    None
133  * @return        void
134  *
135  * #### Functional Requirements Tag
136  *  [$Satisfies $SDD_PNC 011]
137  */
138 FUNC(void, PNC_CODE) Activation_Sensor(void);
```

Figure 4.7: Doxygen comment style in API for functional description (**ETAS Confidential**)

How it has been specified in previous paragraphs the matrix shall implement a bidirectional traceability and therefore it is not enough to trace only the SSR in source code but also the assumed Safety Requirements at system level, the refined Safety Goal, the test case specification and the test cause implementation and result. Actually in code documentation there will be one matrix for safety requirements implementation and another one for verification. The contents will be respectively:

Requirement Implementation

C1- SSR Tag;

C2- Internal or Global Implementation i.e. it explains if the requirement is implemented by an internal specific statement, or with "global" if it is implemented by the whole software unit r a global variable;

C3- Safety Goal (SG) Tag that is refined;

C4- Assumed Higher-level safety requirement Tag;

Verification

C1- SSR Tag;

C2- Test Case specification Tag;

C3- test script name

C4- Id of the test unit;

Once code has been commented and the Doxygen configuration file has been refined, its compiler will be able to generate the documentation. For that of Partial Networking Coordinator, it has been decided, as previously mentioned, to document the code expanding AUTOSAR Macros. The comments contain customized commands in order to implement the hyperlinking to related specification as Software Design Document and Safety Manual, verification report and test case specification. In figure 4.8 it is shown the produced HTML documentation with respect to function *Activation_Sensor* (API in figure 4.7)

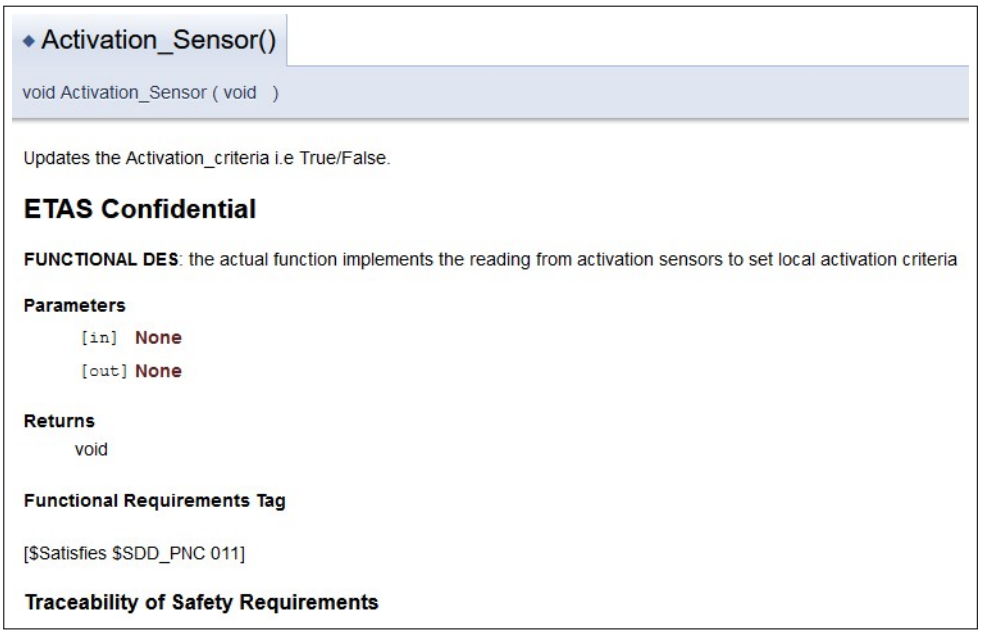


Figure 4.8: Produced Code Documentation (Implementation)

How it is possible to observe, the function name does not contain AUTOSAR macro *FUNC* but directly the function returned type. Each Software Unit is documented within a package graphical symbol. In documentation is also possible to see the tracking tag for functional specification that Actually is not done by Doxygen. The previous figure actually shows the documentation of comment block in .h file (4.7). The last row is "Traceability of Safety Requirements". From that point, the documentation is produced on the base of comment block in source file with the same Doxygen style. The scope is to produce the traceability matrix for implementation of safety requirements using the markdown syntax. The result is shown in figure 4.9. Let's consider each word that is written with blue or light blue font, in produced documentation, is a link. This means that clicking on it another document will be opened. The hyperlinking for implementation of requirements aims to show:

* Requirements specification;
* Safety Goal;
* Internal statement implementation (tracked);
* Specific software unit or global variables;

Traceability of Safety Requirements

Legend:

- *A-SSR*: Assumed Software Safety Requirements;
- *Location*: Global, Internal Statements;
- *TCS*: Test Case Specification;
- *TU*: Unit test (test script and unit name);
- *SG*: Safety Goal;
- *HLSR*: High Level Safety Requirements

Implementation

A-SSR	Location	SG	HLSR
7.1	Internal Statements	37	23, 24, 26, 27, 31
7.2	Internal Statement, Global		
7.3	Internal Statement		
7.4	Internal Statement		
7.5	Internal Statement		

Internal tracking: here

See also

SUM_PNC_Criteria_Sensor_Act_local

SUM_PNC_Criteria_Sensor_DeAct_local

Figure 4.9: Traceability matrix for implementation (generated)

Clicking on one link in traceability matrix, the browser will open the safety manual exactly to the paragraph where that requirement is explained. For instance, let's imagine to click on link *7.4* in SSR column. The result will be like in figure 4.10 (**CONFIDENTIAL**) The Internal tracking requires some additional explanation. Clicking on it, the view will be addressed to a specific page where it is possible to see the internal function statements. Some of them, directly implement the safety Requirements. To track The code statements, particular structures in comment block shall be defined; they are called **Trackers**. Actually

3	A-SSR4	It is necessary to check Majority value assessment for just one value. If it does not, the other is set	SG_PNC00
4	A-SSR5	If number of sensor Values to 1 does not reach Majority then 0 is selected	SG_PNC00

Figure 4.10: Opened section of safety manual using generated documentation link

they are able to track specific statements and to associate them to a SSR implementation but to do this, they shall be implemented by the definition of a keyword. It is necessary to specify that once a code statement or a piece are updated, it is not necessary to change the tracker, because it is able to find specific keywords inside the code. The only necessary update is for trackers: there must be one for each time it is invoked in code.
At this point there might be a problem: how it has been specified Trackers are just comment blocks in C code, so they have not an impact on the program functionalities but they have on the code size, increasing the comment overheading. In figure 4.7 is possible to see that for definition of a functional description, it is necessary to write tenths of LOC: they take up space reducing code capabilities for Doxygen compiler Boundaries and they increase a file size with not functionalities implementation; to generate The entire unit documentation (traceability Matrix and others information) might necessary more than 60 LOCs. This number can increases itself according to documentation structure complexity. A file with a big size it is difficult to maintain and this aspect is very important for ISO 26262 requirements, especially considering the development of a SEooC. For this reason the Trackers will be implemented in an additional C file. This will be provided as a member of work product by actually it shall not be compiled or tested. It just shall be parsed by Doxygen compiler to create internal tracking in dedicated documentation page. At the same time, the usage of dedicated file source will decrease the complexity in maintainability of that same Trackers in case of change, replacement and updating. Table 4.2 summarizes all positive and negative aspects of the dedicated source file for traceability versus the embedded ones in an unique source code. In figure 4.11 it is possible to observe the produced documentation for a tracker for requirements SSR 7.4

SSR-7.4

```
121       /* startSSR_7.4 startSSR_7.5 */
122       /* assign value to Activation Local Criteria according to majority voting */
123         if(Activation_asserted_to_one >= majority_value)
124         {
125             /* startSSR_7.2 */
126           SUM_PNC_Criteria_Sensor_Act_local[SUM_PNC_PNC_Number] = TRUE;
127
128         }   /* endSSR_7.2 */
129         else
130         {
131           SUM_PNC_Criteria_Sensor_Act_local[SUM_PNC_PNC_Number] = FALSE;
132         }
133        /* endSSR_7.4  endSSR_7.5 */
136         /* startSSR_7.4 startSSR_7.5 */
137         /* assign value to DeActivation Local Criteria according to majority voting */
138         if(Deactivation_asserted_to_one >= majority_value)
139         {
140           /* startSSR_7.2 */
141           SUM_PNC_Criteria_Sensor_DeAct_local[SUM_PNC_PNC_Number] = TRUE;
142           /* endSSR_7.2 */
143         }
144         else
145         {
146           SUM_PNC_Criteria_Sensor_DeAct_local[SUM_PNC_PNC_Number] = FALSE;
147         }
148         /* endSSR_7.4 endSSR_7.5 */
```

Figure 4.11: Internal Statement tracking for specific requirmenet (**ETAS Confidential**)

Last part of the traceability in C code is the tracking of the verification specification and the testing unit. At the scope of this paper, the verification plan aims to perform unit testing, i.e. the base of the V-model. Unit test has been performed using QA-Systems Cantata. Once test are launched, Cantata produces several work products. The most important are:

Tracker place	Compiling Code source	Increasing Work Product	Increasing code size	decrease Maintain-ability	decrease under-standabil-ity
Source code	Yes	No	Yes	Yes	Yes
Dedicated source	No	Yes	No	No	no

Table 4.2: Embedded Tracker vs Dedicated source

A) Verification Report it contains:

a.1) Test cases description;

a.2) Test case report according to rule coverages that have been defined;

B) Test Summary Report: Consider the overall testing result according to coverage rules;

Cantata reports are in html format. This allows to modify them, in order to create identifiers for allowing the hyperlinking of documentation. In this way a specific tag can be added in the traceability matrix that is related to verification part. At the same time it is possible to use Doxygen for documenting the test script unit functions. For the specific case, in the test unit documentation will be described:

A) The test case which test unit is referred;

B) Partial Network which is referred;

C) Interesting Input values

D) expected output

The traceability matrix shall be capable to create a link to each of the previous information. The produced tracking structure for the verification can be seen in figure 4.12.
Let's consider SSR 7.4: it is related to 4 test cases. Clicking on TC with tag 2, The result will be the same of figure 4.13.

Each Test case is inserted in traceability matrix by a mapping between a numerical identifier and test unit. Clicking on the name of test unit, under the bold text "See also", it is possible to consult its documentation that has been generated by Doxygen; otherwise clicking on the name of test script, it is possible to consult the documentation related to all test units that are defined within it. Just for instance, it is possible to see that Test Unit named *test_Activation_Sensor_2* tests the implementation of SSR 7.4 (figure 4.12); clicking on the name of the test unit the view will be moved to the documentation of it as reported in figure 4.14. Let's note the final line of the figure: it contains the cross-reference line to the source code. Clicking on word *sensor.c* it is possible to open the whole source code that has been written to test at unit level the sensor, while clicking on the number it is possible to open the source code to the specific line, where the implementation of sensor software unit starts.

Verification

A-SSR	TCS	testScript	Test Unit
7.1	0, 1, 2, 3, 4	test_sensor.c	0, 1, 2, 3, 4
7.2	Implicit	test_sensor.c	All test for Activation sensor
7.3	0, 2, 3, 4	test_sensor.c	0, 2, , 4
7.4	0, 2, 3, 4	test_sensor.c	0, 2, 3, 4
7.5	0, 2, 3, 4, 5	test_sensor.c	0, 2, 3, 4, 5

The number of Test Unit column are referred to the following units:

See also

test_Activation_Sensor [0]

test_Activation_Sensor_1 [1]

test_Activation_Sensor_2 [2]

test_Activation_Sensor_3 [3]

test_Activation_Sensor_4 [4]

test_Activation_Sensor_5 [5]

Definition at line 70 of file sensor.c.

Figure 4.12: Verification Matrix Traceability generated

AR allows to trace the safety requirements using its artefact too. The specification are available in *Specification of Safety Extension* in AR specification. This method can have different advantages: first of all it is standardized because it requires specific classes descriptors to be defined. Besides it allows the tracking at several product development level (System, Components or elements) and then they can be linked using AUTOSAR references classes. Its usage allows a faster requirement definition and a good maintainability especially in the case where safety mechanism, for error handling, are in scope of facilities as watchdog, E2E protection, Memory services and so on; then it is useful to allocate and then track the safety requirements for dynamic code. In this case, it is possible to link the safety mechanism in specific AR basic software module. The usage of Doxygen, for C code, has been preferred in this case because it requires a standardization in the structure of the documentation that can be decided by developers in according to specific cases.
just for comparing, of course Doxygen allows a good level of customization that can help in the verification of a good implementation a big amount of stakeholders. In other words, to read Doxygen documentation and then verify the correctness and the completeness of requirement implementation, with respect to other work products, it is not necessary to know AR schemes and to read a big mole of Specification (AR is not famous to have very short specifications). Besides the traceability of requirements using AR scheme it is allowed only starting from version 4.3.1. Nowadays there are Companies still develop products with version 4.2.2. Therefore Doxygen solves versioning problems because it requires only an enough updated browser as Chrome or Firefox or a generic html viewer it can support hyperlinking. At the same time, Doxygen can be used also to document dynamic code with the same comment blocks that are used to document traceability for static code (traceability matrix in code documentation has been based on AR specification

Test Case: test_Activation_Sensor_2 - Test Case verifies A-SSR01, A-SSR02, A-SSR03, A-SSR04, A-SSR05: PN has not activation sensors

Summary status	Passed
Checks passed	2
Checks failed	0
Checks warning	0
Script directives	2
Script errors	0
Script warnings	0
Call sequence failures	0
User comments	0

Script Directive: Start Call Sequence Validation

Source File: test_sensor/test_sensor.c
Line Number: 281

Description:

Check: SUM_PNC_Criteria_Sensor_Act_local[SUM_PNC_PNC_Number] = 0u Passed

Source File: test_sensor/test_sensor.c
Line Number: 288

Expected value: 0x00 <NUL>
Actual value: 0x00 <NUL>

Check: SUM_PNC_Criteria_Sensor_DeAct_local[SUM_PNC_PNC_Number] = 0u Passed

Source File: test_sensor/test_sensor.c
Line Number: 289

Expected value: 0x00 <NUL>
Actual value: 0x00 <NUL>

Call Sequence Validation: End Call Sequence Validation Passed

Source File: test_sensor/test_sensor.c
Line Number: 291

Description:

Outside Test Case

Script Warning

Source File: test_sensor/test_sensor.c
Line Number: 60

Name: Verify initialise_expected_global_data()

Figure 4.13: Cantata Test Report

test_Activation_Sensor_2()

void test_Activation_Sensor_2 (int dolt)

test unit for Test Case 2

The Test unit tests PN0 with the following conditions

- PN shall be processed
- Number of sensors is 3 (majority = 2)
- Activation sensor values = {F, T, F};
- Deactivation Sensor Value = {T, T, F};

Expected

PN processed	Local Activation Criteria	Local Deactivation Criteria
YES	FALSE	TRUE

Definition at line 298 of file test_sensor.c.

Figure 4.14: Documentation of a Test Unit using doxygen

for multi-level traceability).
In the following is specified how a TSR can be defined for a SEooC using AR classes:

```
<!-- Requirement class -->
<STRUCUTURE-REQ>
    <!-- Requirement Identification -->
    <SHORT-NAME>A-HLSR-00</SHORT-NAME>
    <LONG-NAME>
        <L-4 L="EN">Brief Description...</L-4>
    </LONG-NAME>
    <!-- Type of Safety Requirment -->
    <CATEGORY>SAFETY_TECHNICAL</CATEGORY>
    <ADMIN-DATA>
        <SDGS>
            <SDG GID="SAFEX">
                <!-- ASIL level fo requirement -->
                <SD GID="ASIL">D</SD>
                <!-- Status Attribute -->
                <SD GID="STATUS">PROPOSED</SD>
            </SDG>
        </SDGS>
    </ADMIN-DATA>
    <!-- Here should be placed allocation to higher level (FSR) (no for SEooC) -->
    <TYPE></TYPE>
    <DESCRIPTION>
        <P>
            <L-1 L="EN">
            Detailed Description...
            </L-1>
        </P>
    </DESCRIPTION>
    <RATIONALE />
    <DEPENDENCIES />
    <USE-CASE />
    <SUPPORTING-MATERIAL />
</STRUCUTURE-REQ>
```

Figure 4.15: Definition of a TSR in AR scheme

How it is possible to see, the Technical Safety Requirements can be defined with category Safety_Technical but they cannot be linked, or better Traced, to an upper link because it would be a FSR that does not exist in the development of SEooC. The actual TSR is specified with unique Identifier, using class SHORT-NAME and it can contain a brief description and a more detailed one within AR scheme. Status attribute and ASIL value are specified in order to meet all ISO 26262 requirements for the specification of a safety requirement (chapter 8). At this point TSR is defined but it shall be linked to a component to define all interaction. In the following case it is shown a software component in the application layer with a link to the TSR defined in figure 4.15. The name of Software component is **Component** and it belongs to AR-package **RTA**. The xml scheme for these is shown in figure 4.16

During the current section it has been said that a good traceability method shall be able to trace a requirement through different work products. This means that a single method shall be applied to a source code file and to an AUTOSAR scheme without major troubles. For the first application, it has been chosen to use Doxygen to generate documentation from comment in C code. For the second one is however possible to use Doxygen for tracking the requirements in .arxml files and for the generation of a readable documentation but this requires an additional operation because Doxygen does not support parsing of xml file for documentation but it supports markdown and html.
This means that files with these extensions can be included in the documentation that is produced by Doxygen. The original idea was to generate the traceability documentation

```
[...]
<AR-PACKAGE>
    <SHORT-NAME>RTA</SHORT-NAME>
    <ELEMENTS>
        <APPLICATION-SW-COMPONENT-TYPE>
            <SHORT-NAME>Component</SHORT-NAME>
                <ADMIN-DATA>
                    <SDGS>
                        <SDG GID="ASIL">
                            <SD>D</SD>
                        </SDG>
                        <SDG GID="REALIZES">
                            <SDX-REF DEST="STRUCTURED-REQ" BASE="SAFEX">>A-HLSR-00</SDX-REF>
                        </SDG>
                    </SDGS>
                </ADMIN-DATA>
        [...]
        </APPLICATION-SW-COMPONENT-TYPE>
    </ELEMENTS>
</AR-PACKAGE>
```

Figure 4.16: reference to a TSR from a SWC

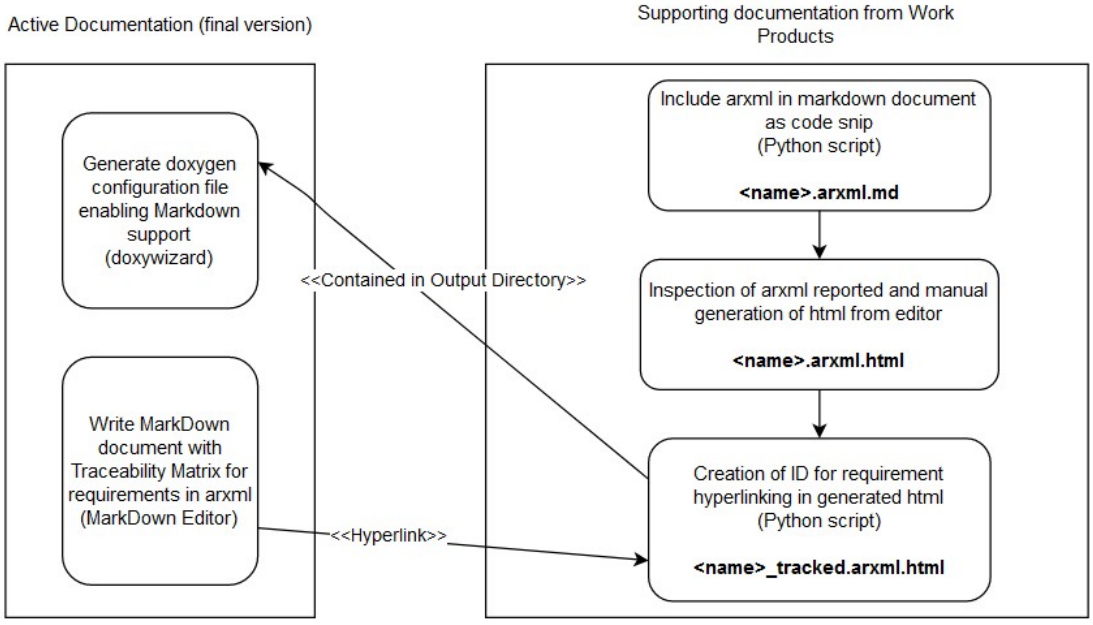


Figure 4.17: work-flow to implement dynamic traceability of an arxml file

for all work products that was not C files or Cpp ones in an automatic way. Actually this part has been partially covered because it is still required a manual operation that forces the exporting from MarkDown to html by a Markdown editor. This is necessary to obtain a final html page that implements some trackers, maintaining its original style.
A future implementation might be to define an own .css to allow to an html file the containment of a xml scheme as reporting code. Figures 4.17 reports the flow and the operation that are requested in order to include the traceability of the an arxml in Doxygen documentation. The final page organization, exlporing from index html is in figure 4.18 while the work product will be shown in figure 4.19 (figure shows a test result)

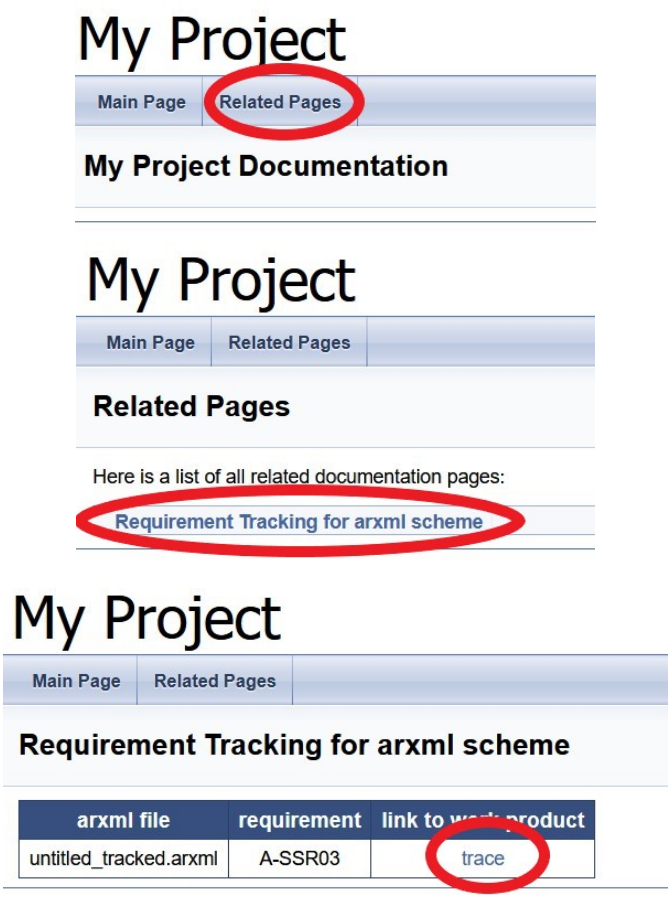


Figure 4.18: Traceability documentation for arxml file

4.3 Verification of Software Architecture

The software architecture will be built by static design aspect and dynamic ones as reported in section 3.3. How it has been explained, the software architecture shall be defined according to specific requirements and it is implemented according to features and properties of selecting programming language. Verification of Software architecture shall ensure the allocation of SSR in the designed one.
ISO 26262 defines a series of methods for that. How for the other methods, they shall be chosen according to the specific case, Company policies and mostly ASIL value (D). Each method can be applied to model or others products as documentation and code. In the first case the mostly activities can be done on model, reducing the number of products to be directly verified.
A good method for the verification of software architecture can be done by **Inspection**. This method allows a review of the whole work products, searching defects. Inspection has not the only effect to find defects in work products but also to improve the maintainability and to comply the work products to the Quality Assessment (QA). The review process that is executed on the Partial Networking Coordinator is a Change-based code review that it involves among all engineers of the software development team; generally, change-based review is done only of code samples that have been affected by changes according to the opening of a specific ticket, using FMEA as guideline to understand which units may be implicitly affected by an unit changes. In particular this kind of verification aims to

```
A-SSR03
 -->
                                    <AUTOSAR-ENGINEERING-OBJECT>
                                        <SHORT-LABEL>untitled_h</SHORT-LABEL>
                                        <CATEGORY>SWHDR</CATEGORY>
                                    </AUTOSAR-ENGINEERING-OBJECT>
                                    <AUTOSAR-ENGINEERING-OBJECT>
                                        <SHORT-LABEL>untitled_private_h</SHORT-LABEL>
                                        <CATEGORY>SWHDR</CATEGORY>
                                    </AUTOSAR-ENGINEERING-OBJECT>
                                    <AUTOSAR-ENGINEERING-OBJECT>
                                        <SHORT-LABEL>untitled_types_h</SHORT-LABEL>
                                        <CATEGORY>SWHDR</CATEGORY>
                                    </AUTOSAR-ENGINEERING-OBJECT>
                                </ARTIFACT-DESCRIPTORS>
                            </CODE>
                        </CODE-DESCRIPTORS>
                        <!-- ASSR15 -->
                        <PROGRAMMING-LANGUAGE>C</PROGRAMMING-LANGUAGE>
                        <RESOURCE-CONSUMPTION>
                            <SHORT-NAME>RsrcCons_untitled</SHORT-NAME>
                        </RESOURCE-CONSUMPTION>
                        <SW-VERSION>1.0.1</SW-VERSION>
                        <USED-CODE-GENERATOR>Embedded Coder 7.3 (R2019b) 18-Jul-2019</USED-CODE-
GENERATOR>
                        <VENDOR-ID>0</VENDOR-ID>
                        <BEHAVIOR-REF DEST="SWC-INTERNAL-BEHAVIOR">/Components/untitled
/untitled_IB</BEHAVIOR-REF>
                    </SWC-IMPLEMENTATION>
                </ELEMENTS>
            </AR-PACKAGE>
        </AR-PACKAGES>
```

Figure 4.19: work product with tracking id

discover three kind of defects:

- Missing of consistence between source code and documentation;
- Missing requirement implementation from Traceability;
- Bugs, safety gaps and vulnerabilities in source code implementation to be fixed or improved;

An important point is that, in general, the development and the inspection shall not be performed by the same person to be effective.
One common defect in automotive application codes can be related to the abstraction of the physical units in software data: let's consider that AUTOSAR implements three types of data:

A - Application data;

I - Implementation data;

B - Base type;

B type, is the basic data can be defined by C programming language. Basically they define boundary values for each kind of data according to the number of bits they can encode a real value. The safety definition and usage of a base type with C is defined in MISRA C and additional Company Coding style guidelines. For instance, if there is the necessity to define a variable for the actual vehicle speed (variable name is *actual_speed_value*) it can be defined from **unsigned char**. This defines a value on 8 bits without sign (so it is not considered in *2-C*). The actual speed value will be measure between 0 and 255. At Implementation level, it is possible to define a data type that will be able to contain a value encoding on a specific number of bits according to actual platform. In C this can be done in this way: **typedef unsigned char** *unsigInt8*. Therefore an I type named *unsigInt8* has been mapped with a base type (unsigned char). Let's note that by a Software point of view the measuring units have not sense because the data types define boundaries (in terms of encoding bits) and algebraic sets; by a physical point of view, a speed value needs measuring units to get sense ($120Km/h$ is different from $120Mph$). For this reason is necessary to define an Application data type, that can be used by application software component, to define data they can abstract the physical values; for example the application data type *Speed_Kmph* is defined to measure a speed value in Km/h. Now, let's consider that an OEM wants to limit the maximum speed to $230Km/h$. In this case the application data type upper boundary is such as to be represented by *unsigInt8* (Implementation type) therefore *Speed_Kmph* can be mapped in this implementation type with a specific Computational method that is able to convert one in another.
In Automotive software, considering the case of a long supply-chain, it is not so strange that *Speed_Kmph* data type is used for measuring an actual speed in Mph (Miles per Hours). This can cause several problems, from boundaries point of view and threshold actuation response too. A similar directive is also contained in MISRA C, where it suggests to define an unique variable, identified by name, for each virtual measuring unit i.e. each variable shall be declared for a specific scope. In the inspection of PNC code a bug has been discovered: in the activation sensor software unit, an index has been confused, allowing to measure both a flag for the processing of PN permission and actual sensors value: this might cause a boundary problem, because for the multiplicity of the design, a PN can have a different actual or maximum numbers of activation sensors and calibration parameter for PN processing. Another important aspect in terms of safety is the boundaries of the **for** indexes: the size in terms of encoding bits for a index shall be at least equal to the size of the actual counted value.

In addition to the Inspection others method have been applied. Let's note that in several case, embedded software for application component is modelled as a Finite State Machine or using state flow diagram. This because, generally, an embedded Systems shall produce an output according to specific input and this output is related to an operating mode of the system or the component. Therefore the modelling phase, that can be used to develop source code, using embedded coders or manually, shall define the states of a component, with the related operations they produce the output values and the inputs they will define the conditions for transition among them. In complex systems, the number of modes (and so the related combination of input conditions) can be very large and so it might be very difficult to manage all of them. For this reason the input conditions can be defined using *Don't Care conditions*. These last can cause a problem that is called **Overlapping**. An overlapping is a condition where a input combination can define a transition to more states, starting from another one. In this case the system deterministic behaviour is negative af-

fected, because the states are defined in order to be unique operating modes. To solve it, there is the possibility to use the **Control Flow Analysis**. It is a static code analysis aims to discover possible overlapping that might cause *losses of decidability* and **halting problem**. The Control flow Analysis is widely adopted in safety-critical system design due to particular advantages:

1. It can be performed only on safety related parts of the system or component;
2. It can be executed by Automated tools;

ETAS has an own tool, called SCODE, that is able to analyse the Control System completeness, determinism and predictability. At the same time the software tool is able to optimize the state definition and the transition using formal verification and it has a good level of integrability on common products as Eclipse and Matlab/Simulink. Tool is already qualified by ISO 26262 and so it does not request additional safety activities. In the following will be analysed an simile case of trouble that has been found, very similar, on the PNC. Let's imagine to have a Finite State Machine like figure 4.20

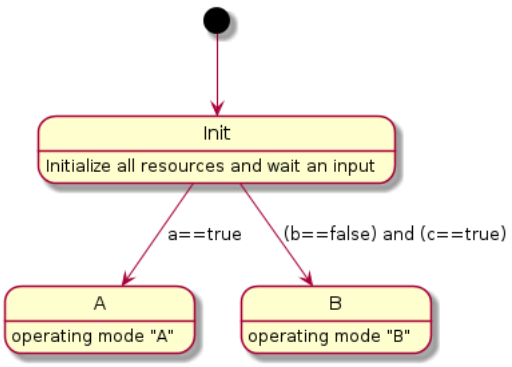


Figure 4.20: Erroneous Finite State Machine

Basically the Finite State Machine defines three states: The first (*Init*) is reached when control system is turn on or reset. Its scope is to prepare each resource shall be used by other state operation in order to produce the right output. Neglecting the produced output, the Control System will evaluate 3 inputs: a, b and c. This means that the input combinations will produce 8 possible cases to be evaluated. State A will be reached if the a input is true, considering the others apparently don't care; at the same time, B will be reached when b is false and c is true considering a, apparently don't care. In this case is very simple to note a problem about decidability: if the three input assume respectively values *true*, *false*, *true* both edges conditions are satisfied. In this case both transitions are possible but just only one should be executed. The Control Flow Analysis shall be able to consider the implementation of the following model as potentially vulnerable. A solution to solve this vulnerability is to specify better the condition, or assigning them a well-defined priority. Considering this last one and giving to a evaluation an higher priority the situation will be very similar to figure 4.21 while Table 4.3 showing the actual situation with respect to the actual consideration.

Now let's consider b and c as don't care for the transition Init-A because they are evaluated only once a is asserted as *false*. At the same time, in verification phase it is possible to

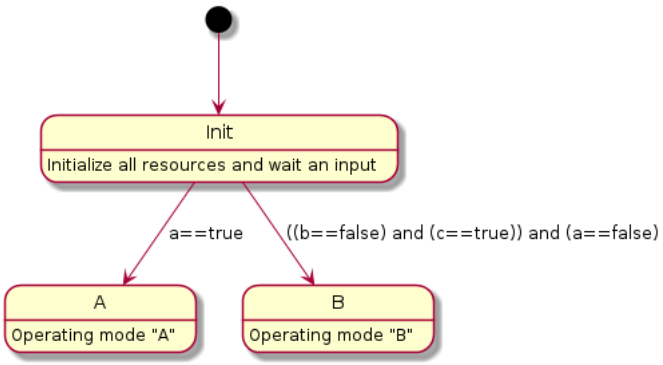


Figure 4.21: Correct Finite State Machine

a	b	c	Actual Sate	Next State
F	F	F	Init	Init
F	F	T	Init	B
F	T	F	Init	Init
F	T	T	Init	Init
T	F	F	Init	A
T	F	T	Init	A
T	T	F	Init	A
T	T	T	Init	A

Table 4.3: Truth Table for Correct FSM

assert if Init state will not execute any kind of transition simply evaluating conditions: $(a == false)$ *and* $(c == false)$ **or** $(a == false)$ *and* $(b == true)$. This will modify the truth table as following:

a	b	c	Actual Sate	Next State
F	-	F	Init	Init
F	F	T	Init	B
F	T	-	Init	Init
T	-	-	Init	A

Table 4.4: Truth Table for Correct FSM (optimized)

Now only 4 combinations are enough to evaluate the FSM behaviour in its completeness. Let's note that from this description is also easier to implement the respective C code and in this case it will be also robuster because it does not show vulnerability in terms of transition evaluation. In figure 4.22 it is possible to see an example for the implementation of considered FSM according to the last Truth Table.
How it has been described, the Control Flow analysis shall help the developer in the identification of the input condition in order to implement the previous code. Let's note that Software Unit does not contain any kind of control. In fact this part shall not be confused with the implementation of the software unit where Error Handling mechanism for controlling the transition, can be implemented.
The definition of priority is also used in model-based software design: for example Simulink allows the definition of a model by state-flow and it assigns a specific priority to each state

```
2  typedef unsigned char boolean;
3  typedef unsigned char uint8;
4
5  #define True  ((boolean)1)
6  #define False ((boolean)0)
7
8  typedef enum {
9      Init=0u,
10     A,
11     B
12 }State;
13
14 typedef struct Cond{
15     boolean a;
16     boolean b;
17     boolean c;
18 }Input;
19
20 typedef struct {
21     /* Don't Care */
22 }Output;
23
24 typedef struct FSM {
25     /* Coordinator values */
26     State state;
27 }PNC;
28
29 #define PNC_START_CODE
30 #include "PNCMemMap.h"
31
32 FUNC(void, PNC_SEC) Init_State(PNC *pnc, Input condition) {
33
34     if(condition.a == True) {
35         pnc->state = A;
36     } else if((condition.b == False) && (condition.c == True)) {
37         pnc->state = B;
38     } else {
39         pnc->state = Init;
40     }
41 }
42
43 #define PNC_STOP_CODE
44 #include "PNCMemMap.h"
```

Figure 4.22: Implementation of FSM

transition. These priorities will be used for code generation using embedded coder. This is very useful especially when **nested If** (or commonly **If-else-if**) statements are used: the transition cardinality can avoid the necessity to specify the values for each single condition that impacts on the decision. In this case the FSM will be something of similar to figure 4.23. The cardinality is expressed within square brackets therefore transition from *Init* to *A* state is considered first of any other.
This aspect can help in the expression of conditions, considering that how in the case of the real PNC, the number of conditions might be very large to specify and check in an if condition. The implementing C code of FSM in figure 4.23 will be the same of figure 4.22; in the case transition *Init-B* was ranked with an higher cardinality than *Init-A*, the implementing code would require just a change of evaluation i.e. the first if would evaluate *c* and *b* while the second *a*. To summarize, the usage of model-based software design can improve the design phase of a FSM, providing some facilities for the design phase but

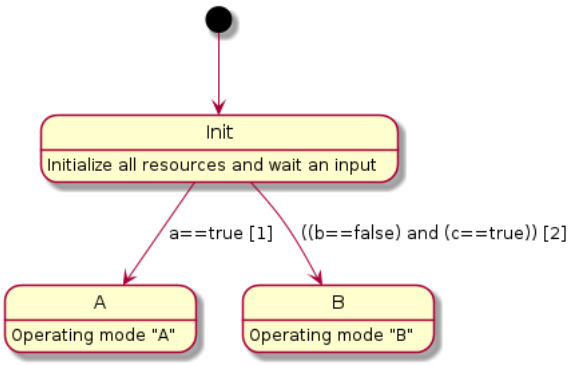


Figure 4.23: FSM with transition cardinality

however a Control Flow Analysis is necessary or it might be replaced by a **Simulation** of the model; By the way, the low software complexity is always suggested to make easier its verification too.
A final note shall be done for code **inspection** with respect to the previous presented code: MISRA C requires an *else* statement for each *if* one. for this reason the previous code can be implemented with an intuitive variant but that potentially might avoid the usage definition of a final else but actually it shall maintain as an empty node" (figure 4.24)

```
29 FUNC(void, PNC_SEC) Init_State(PNC *pnc, Input condition) {
30
31     /* default transition is remaining on actual state */
32     pnc->state = Init;
33
34     if(condition.a == True) {
35         pnc->state = A;
36     } else if((condition.b == False) && (condition.c == True)) {
37         pnc->state = B;
38     } else {
39         /* nothing must be done */
40     }
41 }
```

Figure 4.24: MISRAC compliant code

The presence of an *else* for each *if* can ensure the value assignment to a variable in any case. Let's remember that some safety problem can be present when a compiler has too much freedom in compiling the source code. Therefore the declaration of variables in **bss** or **data** segments shall not be done assuming how compiler might interpret it. The verification of software architecture by Control flow analysis shall also check these aspects. An example is provided in figure 4.25
In figure 4.25, the missing of mandatory *else* can cause a safety gap, because it leaves the assignment of default value to a variable to compiler (variable is *enable*). Figure 4.26 shows a safety implementation of a software unit with the declaration of an uninitialized variable. At the end Control flow analysis allows to discover an optimization too: observing the conditions, it is possible to note that the second if check an not useful condition: if first condition is false then return value will be false anywhere. for this reason the second if is not useful and so it can be removed. The advantages are: reduction of cyclomatic

```
48 FUNC(boolean, PNC_SEC) enableResource(Input condition) {
49     VAR(boolean, AUTOMATIC) enable;
50
51     if(condition.a == True) {
52         enable = True;
53     } else if(((condition.b == True) && (condition.c == False)) ||
54               ((condition.b == False) && (condition.c == True))) {
55         enable = False;
56     }
57
58     return enable;
59 }
```

Figure 4.25: Uninitialized variable

```
48 FUNC(boolean, PNC_SEC) enableResource(Input condition) {
49     VAR(boolean, AUTOMATIC) enable;
50
51     if(condition.a == True) {
52         enable = True;
53     } else if(((condition.b == True) && (condition.c == False)) ||
54               ((condition.b == False) && (condition.c == True))) {
55         enable = False;
56     } else {
57         enable = False;
58     }
59
60     return enable;
61 }
```

Figure 4.26: Mandatory else to assign variable value

complexity to 2 and reduction of test Case to obtain code coverage. The actual code will be shown in figure 4.27

```
48 FUNC(boolean, PNC_SEC) enableResource(Input condition) {
49     VAR(boolean, AUTOMATIC) enable;
50
51     if(condition.a == True) {
52         enable = True;
53     } else {
54         enable = False;
55     }
56
57     return enable;
58 }
```

Figure 4.27: Optimized cases

4.4 Verification of Software Unit

How the higher levels, software unit shall be verified by some methods before being tested. This section explores the main activities and methodologies that have been adopted to test the software units developed for the Partial Networking Coordinator. Again a good starting method can be the **Inspection** such as the verification of software architecture design.
for SEooC, in general, it is very difficult to execute an evaluation of **used resources** and at the same time it might be not so useful, therefore is not an useful verification method in this case. Even if ISO does not explain directly this, it is reasonable to adopt verification methods that is aligned with the implementation methods: for example the resource evaluation might be useful, to be consecutive to a scheduling analysis, in order to understand the resources that shall be adopted for a certain operation for meeting dynamic software execution. The adoption of the redundancy for Error detection and Handling is an example because, potentially for a certain operation dedicated resource shall be at least duplicated. Another example is related to the execution of parallel as in the concurrent units. However, in several cases, including the current, the resources evaluation can be replaced by other analysis.
The verification of the software unit has been done, dividing the activities into two logical sub-verification:

A) Methods for Verification by Analysis;

B) Methods for deriving test strategies;

Basically for topic *A)* the situation is very similar to the previous verification (software architecture): software units nature depends from many factors, first of all the programming language that is used. For example, considering the nature of the partial Networking Coordinator as unique AUTOSAR Component the Software Units are considered like C functions.
Data Flow Analysis can verify software unit design and implementation, considering the C functions as *white box*. Basically the verification by **Data Flow Analysis** might discover inconsistencies in data values, due to wrong scheduling of computation, or wrong computation on a static/global variable it survives for the entire program life-cycle. The base of Analysis is the definition of a *DFG* (Data Flow Graph) for each software unit at function level; graph nodes are actually *Basic Blocks* i.e. a series of code statements without branches (except first and last) that aims to identify which variables are modified and which variables are used for their computation within it. The analysis of data dependencies by Data flow analysis, are also done by **compilers** to optimize the source code of a program but in this case, even if they are the same things by concept, they shall not be confused: the data flow analysis at scope of verification is done to verify the correctness of data flow, for avoiding possible missing of consistence. This analysis is divided into two types: *forward* and *backward*. The nature and theory is the same for both of them but they focus themselves on different aspects:

- *Forward Analysis*: aims to identify *reaching definition* i.e. dependencies among variables in order to identify which one is used to modify the other values;

- *Backward Analysis*: aims to identify *live variable analysis* and *dead code elimination* i.e. respectively:
  - variable that maintains its value in current program execution point and therefore it might be used to compute an output variable or read before being written again;
  - variable it does not influence execution, because it is not accessed and therefore it can be eliminated

Code optimization is an active effect of this analysis. In fact the key point, to improve the safety at source code level is **to give as little as possible freedom to compilers**: in several case, safety problems on data are caused due to assumptions on compiler behaviour or leaving the a large optimization choice to it (this can be observed also analysing MISRA directives). Data Flow Analysis can be performed by tools as Static Code Analysis and this aspect has increased its popularity. However it requires several resources allocation: for example, the execution of a Data Flow Analyses in a software unit where there are cycles can be very expensive, using tools too. The reason why this analysis is described in this section is for focusing on three main aspects:

1. When ASIL value is high, mostly there is the necessity to apply more than one ISO 26262 methods even if they are alternative and not sequentially;
2. Data Flow and Control Flow analysis are complementary: a certain flow can have an impact on variable assignment as data have impact on the path that is executed;
3. Data and Control flow analysis can be executed by Automated tools as Static Code analysis in order to highlight bugs, warnings and issues and at the same time to optimize the program reducing compiles impact;

Topic *B)* The adopted methods provides an answer to question *"What shall be tested in a software unit?"*. The importance of testing activities and their related feasibility have been described in the introduction sections. At this point is necessary to define a feasible an exhaustive strategy to test the software units of the Partial Networking Coordinator. Again it is important to remember that ISO 26262 activities have been applied in the development of PNC starting from its beta versions; this aspect has forced to define the safety activities like parallel to the functional development and so the main task of the unit testing will be to provide evidence in favour of the correct implementation of software safety requirements. How it has been explained in 2.2 , the safety key point of the PNC is **to ensure data integrity and consistency and the right control flow**. The previous paragraph has specified, how it is possible to ensure, by software that an unit implements in a correct way the data-flow; but this is not enough because it is also necessary to ensure that a value of a variable is correctly obtained, operations produces the right result and the level of robustness of code is appropriate.
From specification is reported that PNC shall manage several information from Service Layer in Basic Software and it shall communicate with other software components in application layer and so ports will be defined and it will implement AUTOSAR interfaces.
Considering that PNC is a communication coordinator, it will have to use resources as timers or counters too; In particular they can be implemented using software solutions,

i.e. they will use the periodicity of tasks to compute deadline achievement or counting expiration. For this reason is necessary to ensure a certain level of robustness in the arithmetical/logic operations as safety mechanisms.
According to the previous description is possible to define methods for identifying software unit tests. Let's note that to identify what shall be tested does not mean to derive automatically test cases. This concept will be in scope to the next section. The software unit verification by testing aims to test:

- Requirement implementation;
- Interfaces values and
- Code robustness to presence of faults (permanent)by reaction to fault injection

In the case of a model-based approach (not in scope for PNC because static code is developed manually) might be a good method for high ASIL value, to adopt a Back-to-Back comparison between model and code.

4.5 Testing of Software Units

Once that Verification of the software unit design and implementation has completed and it does not show issues, it is possible to move on the right part of the V-model for rising on. The software unit test aims to prove the correct implementation of software units launching them in a controlled environment. What shall be tested in a software unit is identified in the verification of software units methods that has been named as **topic B)**. ISO 26262 chapter 6 clause 9 provides a series of methods that can be used, according to ASIL value of the element, for adopting strategies they can produce feasible test cases writing. Each software Unit is tested by at least one *test case* that is contained, generally, within a test script (a *.c* or *.cpp* file). The definition of test cases can be done following the next topics:

A) **Identification of requirements implementation**: the tests are based on requirements and their main scope is to prove the matching between requirements specification and their implementation. Therefore is necessary to identify, using traceability like a support, where and how requirements are implemented in software units;

B) **Identification and generation of equivalence classes**: this allows the reduction of the number of test cases, grouping the inputs values according to produces output;

C) **check of boundaries values**: this methodologies increase a lot the number of test cases. Although this aspect it is necessary to completely test a software unit. The probability of an error due to the occurrence of a boundary value is not an isolated condition;

To be clearer, the boundary values checking increases the number of test cases by a theoretical point of view because it requires to test the whole **neighbourhood** of the boundary values; this is necessary as for the neighbourhood of a defined range as the neighbourhood of the data type (e.g. 0-255 for **unsigned char**, 0-65535 for **unsigned short** and so on). However it is possible to reduce the number of test cases applying the equivalence class partitioning: it consists in the identification of boundaries values and in the generation

of partitions of equivalent values from a functional point of view. Therefore it is possible to define a single case for each equivalence class, selecting exactly the values at the boundaries. An example can be seen in the next figure:

Figure 4.28: Example of equivalence class partitioning and boundaries

In the figure it is possible to see three partitions (Yellow, Blue and Orange); each of them represents an equivalence class of values. Let's assume that the ranges are applied to an integer value, therefore two consecutive values depart ± 1. The Boundary analysis will be different in the case where the black points at partition borderlines are included in a partition or in another. To make an example, let's assume again that variable Y shall be in the blue range, with extremes that are considered as range members ($RLB \leq Y \leq RUB$); in this case each of the six points in the figure shall be used to analysed the boundaries values, building at least six different test cases. Instead, using equivalence classes, without boundary values analysis, the test cases would be three, i.e. one for each partition. This means boundary values should be checked only where it is really useful, for example, how it is suggested by ISO 26262, for evaluating interfaces values while for the tests that are related to Requirements Analysis, it is possible to adopt the equivalence classes.

The quality of testing is measured by **coverage measure**. For programs, this is called *Code Coverage* i.e. the measurement of quantity of code that is covered by execution of test cases. There is not a single way to obtain a coverage measure; each of them will define a specific "quantity" that is covered. Considering the highest ASIL value for the development of the PNC, it has been chosen to adopt the **MC/DC** as coverage measure. The reason is that, obtaining the 100% of code coverage with this measure, for sure the statement coverage and branch coverage will reach the 100% of coverage. Vice versa is not true. MC/DC is very useful especially adopting C as programming language because in a condition with logical operators, C stops to evaluate the condition once it has reach the first determinant value for taking a decision. Considering the logical operators in a conditions evaluation as only *AND*, *OR* the situation will be in table 4.5
The table exactly shows the flow that is used by C, to evaluate a logical condition: in fact it starts evaluating the first condition and if its result is already able to establish the decision it stops there. How it is possible to see, this strongly depends from type of logical operator: just one False is enough to decide the entire condition for an AND, just one true is enough to decide the entire condition true for OR. At the same time, tables already provide the test cases to write for obtaining the 100% of MC/DC coverage: its scope is to prove that condition is able to take a decision alone, actually done it and this for the whole combinational stimuli are able to do it. Let's derive the test cases for AND operator:

(a == true) AND (b == true)		
a	b	Decision
False	-	False
True	False	False
True	True	True

(a == true) OR (b == true)		
a	b	Decision
True	-	True
False	True	True
False	False	False

Table 4.5: Table for MC/DC evaluation

1. with a False, the decision will be false. If b is considered in turn as False, it is not possible to prove that decision is False due to only a value and so it shall be considered as True. Then First Test case is

$$\{a, b\} = \{False, True\}$$

2. with b False, the only way to prove that it is the only head of decision is taking a True, otherwise it will not be evaluated. The second Test case is

$$\{a, b\} = \{True, False\}$$

3. The last is when it is necessary to evaluate both conditions to take a decision and in this case it consists to force the evaluation of a (True) and then the evaluation of b. This is the only case where exchanging the order of conditions however they require to be both evaluated for the final decision (so b is True too). Last Test case is

$$\{a, b\} = \{True, True\}$$

With the application of these three test cases it is possible to obtain the 100% of MC/DC and at the same time also the complete code coverage branch coverage. considering PNC nature , it will contain several multiple conditions they establish the actual state or mode of the communications and Partial Networks status too. For this reason the most of test cases are provided with the intention of obtain the 100% of the MC/DC coverage on functional requirements and on Software Safety Requirements.
Let's note that identification of test cases for the MC/DC is strictly dependent from the number of decision that are involved in evaluation and the type of logic operator it merges them. For this reason the identification of MC/DC conditions might be not so easy in all cases like in the previous example. The **Cyclomatic Complexity** that is adopted in the development phase, can be used to have an idea about how much test cases should be produced to test each multiple conditions. This is another point in favour of developing of software with a not so large cyclomatic complexity it makes it hard to produce a reasonable amount of test cases.

It is important to spend some words about the tool that has been used for the definition and the execution of test cases: QA-Systems Cantata (knows also as Cantata++). This framework is used very commonly for Unit and Integration testing for safety-critical systems (certified by ISO 26262 [13]). It supports C and C++ as programming language for source code and tests, using Stubs, and it can be easily integrated within Eclipse environment. The test cases are considered as unit tests that are integrated within a test script that can be auto-generated by tool itself, using a Graphical Interface with semi-formal

notation. At the same time it is possible to execute automated testing and to obtain diagnostic results in several formats (in the specific case it has been chosen the html to execute customization operation as it is explained in Traceability section). The integration testing is also allowed for a very large target tests, configuring the right makefile and using GCC has default compiler. The real important things to report, about Cantata advantages, is the possibility to configure the coverage testing result (i.e. a successful coverage or a failing) according to the ISO 26262 ASIL value that is defined for the Component Under Test. In the PNC specific case, its value is D (how it is defined in ISO 26262 version 2011), therefore Cantata requires the following Coverages to produce a successful report, otherwise it will be created with a failing global status:

- Entry Point;
- Statement and Decision Coverage;
- Boolean Operand effectiveness coverage;

In other words, the previous points that Cantata analyses in order to produce the final report, aim to ensure that:

1. All Units in a module have been really tested;
2. Test cases have been able to produce the 100% of MC/DC coverage (let's remember it is the most powerful);

The checks that have been done in software unit test within test scripts aims to check:

- the expected returned value by a direct checking by cantata methods on
  - *a.* range;
  - *b.* type and
  - *c.* value;
- The function calling sequence by cantata methods;
- Memory checking for each structure variable or pointers in terms of value and sizing;
- Global Variable private and public value checking by Cantata methods;

The results of tests that have been executed, are contained in table 4.6. The legend is the following:

- **E**: Entry point Coverage;
- **S**: Statement Coverage;
- **D**: Decision Coverage (Branch Coverage);
- **MC/DC**: Multiple conditions/ Decision Coverage;
- **SC/DC**: Single Condition/ Decision Coverage;

SW Unit	Test Script	Coverage %					Test Cases
		E	S	D	MC/DC	SC/DC	
PNC_Safe_Div_Mul	test_div.c	100	90	90	100	100	3
PNC_Safe_Div_Sub	test_div.c	100	90	90	100	100	3
PNC_Div_BlockSel	test_div.c	100	100	100	100	100	1
GenerateModeMask	test_mask.c	100	100	100	100	100	6
Activation_Sensor	test_sensor.c	100	100	100	100	100	7

Table 4.6: Results of unit tests for introducing software unit implement safety mechanism

Let's note that division methods have obtained the same values. The reason why they have not reached the 100% of coverage is due to the branch that contains action to perform in case of a permanent fault occurrence that in unit testing does not appear. To test it, in the future, it is possible to define the same software unit with an intentional error in coding (e.g if operation executes a/b, permanent fault can be simulated with a/(b + 1)) and test if it works.

Chapter 5

Additional Work Products for safety and Conclusions

5.1 Failure Mode and Effects Analysis

The concept phase or the assumption phase (in the case of SEooC) can be done, using the safety analysis as supporting mechanisms. The main scope of this analysis is to map the possibility of failure in data flow at system level or unit one. Using a well-done safety analysis it is possible to map and to implement well-oriented safety mechanisms for error prevention and error handling. The main scope is to identify the presence of cascading failure to direct the Freedom From Interference safety mechanisms and Common Cause Failures. In ISO 26262 safety analyses have a dedicated clause in chapter 9. Safety Analysis can be executed more times during the product development; for example they can be done as supporting material for HARA to derive safety goal, then to identify the right safety mechanisms for implementing TSR and SSR and avoiding FFI and at the end to prove the functional safety assessment. For this reason they can be considered as supporting data for verification activities. Actually, when it is necessary to plan a safety analysis is necessary to identify it by two main characteristics:

- Quantitative or Qualitative;
- Inductive (bottom-up) or Deductive (top-down);

Failure Mode and Effects Analysis is an inductive and (generally) quantitative analysis. The main scope is to map a failure identified by a "Mode" to a specific effect. Therefore it starts knowing the causes they can involve in a failure and it identifies the effects. Generally, FMEA is applied to hardware elements where it is possible to provide and analyse quantitative data as for example the failure rates with tools support. When Diagnostic mechanisms are also known it is possible to execute another version that is called FMEDA. However, it is also possible to apply the FMEA at software level like in the case of PNC. In this case the analysis is mainly qualitative because it cannot provide at priori the failure rates values, instead it aims to map the failure effect, defining failure modes at system level by keyword, in different components they build the system itself.
The first task is to identify the fault models and then, for each of them, a series of deriving failure causes. Then it is possible to define the failure mode at system level that shall be based on HAZOP; for each HAZOP there will be an effect. ETAS idea is to consider each one of the previous step on a specific level:

1. **Failure causes** at software component level
    - They are actually caused by hardware faults used by software routines;
2. **Failure modes** at software component level (HAZOP);
3. **Failure effect** at integration or system level

The hierarchical level for analysis execution in the case of PNC is the Software Component Level until units. One of the more important point is to define key word for Potential Failure mode at module level. In fact HAZOP starts from the identification of association between a parameter (in software case *data*) and a keyword they represents a typical deviation it impacts on the same parameter. The HAZOP is done "section-by-section" on the system according to the reference diagram i.e. for software, the sequence Diagram it identifies the software units that are called. For each one the scope is to understand if a deviation can impact on the parameters that are considered by software unit. If it does, then it is necessary to analyse its cause. This cause will be the input of FMEA for analysing the single effect and in-chain ones. How it has been specified, FMEA can be used for several reason; to summarize the main scopes are:

1. To execute the HARA, in order to identify hazard operations and their causes;
2. To support definition of TSR and SSR and to identify safety mechanisms;
3. To provide evidence in favour of the functional safety assessment;

In the previous list, topic 1 and topic 3 can share the same data container that can be a Table. For topic 2, ETAS actually has a own template where data can be reported according to specific guidelines. The current paper will analyse an alternative structure to contain data, for FMEA on PNC. The structure of the analysis is defined by a Table that is shown in the next figure (5.1):

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
ETAS PNC FMEA															
SW Unit	Failure Mode	Effect	Cause	Initial parameters				Safety Mechanisms			Description	Final parameters			
				P	S (Worst Case)	(1-R)	tot	detection	mitigation	Reference		P	S (Worst Case)	(1-R)	tot
Activation_sensor	Wrong Sensor Reading	Unintended Behaviour	Failure in SWC "Sensor"	SeF	0,6	1	0,6*P	n/a	Redundancy	Safety Manual	At least 3 sensors for single reading. Majority value is taken	(0,5*N)*SeF	0,6	1	0,6*P
SetAlarm	Early/Late	1) Missing deadline; 2) Missing Synchronization;	Permanent Fault in computational resource	RPF	0,6	1	0,6*P	Duplicate variable, operation and then checking	n/a	Safety Manual	duplicate operation on different variable involving different resources or algorithms and then match	x	0,6	<1	0,6*P*(1-R)

Figure 5.1: Example of FMEA on PNC

This table can be considered with two separated sections: the first one, contains parameter evaluation for the beta version of the software, while the second half will contain parameter evaluation for the safe product. Before starting the discussion on the two sections, let's specify how pseudo-quantitative measures have been defined. The three evaluation measures are the following:

- **P**: Probability that potential failure mode happens;

- **S**: Severity evaluation of failure mode effect;
- **1-R**: Loss of Reliability in Detection of failures:
  * if *1* it means failure is never detected;
  * if *0* it means failure has been always detected;

These measures (they shall not be confused with ASIL determination!) are applied on software units because the analysis is conduced on component level; this means that failures modes are analysed on single software units while effects are considered at component level. This means that computational of a measure for a certain unit might be different by the same for another. The major effects for PNC have been simply considered as **unintended activation/unintended deactivation**. The causes are considered at software unit level and they are focused on failure modes that can be addressed to specific routines or hardware resources. These last one are of course only assumed considering there is not possibility to know nothing below System services layer. At the end, the final evaluation will be done on the base of the following formula that will provide the risk index of software unit according with the type of failure and its effect.

$$Risk = P * S * (1 - R)$$

The scope of the safety mechanisms will be reduce the Risk value. This can be done decreasing one or more measures in the computation. Safety requirements define how it is possible to reduce the risk by three main methodologies:

1. Prevention;
2. Detection;
3. Mitigation;

Discharging the prevention for this analysis, the detection allows to understand that a generic error is occurring. For this reason, implementing a detection mechanism it is possible to reduce the value of **(1-R)** (increasing detection, R value is increased and so (1-R) is decreased); at the same time, implementing a mitigation method, the effects of a failure is blocked or better **contained** and so **S** and **P** are reduced. Let's analyse how it has been possible to compute these values for the PNC.

A	B	C	D	E	F	G	H
							ETAS PNC FMEA
SW Unit	**Failure Mode**	**Effect**	**Cause**	**Initial parameters**			
				P	**S (Worst Case)**	**(1-R)**	**tot**
Activation_sensor	Wrong Sensor Reading	Unintended Behaviour	Failure in SWC "Sensor"	SeF	0,6	1	**0,6*P**
SetAlarm	Early/Late	1) Missing deadline; 2) Missing Synchronization;	Permanent Fault in computational resource	RPF	0,6	1	**0,6*P**

Figure 5.2: Risk evaluation on PNC beta version

In figure 5.2 are analysed two software unit. The first one is related to the evaluation sensor shall handle the criteria to evaluate an activation or a deactivation of a partial network. The main failure might be related to the provided measure by sensor. If a single signal provides this information as a *False/True* value, then the probability to failure occurrence might 0.5 (50%) or more: it depends from specific features of the Software Component and so it is considered as SeF; the severity will be computed considering State-flow transition: first consideration is about all coordinator modes they require the sensor measure to evaluate the next mode. From these it will be considered all transitions that happen due to sensor information (i.e. that are triggered by sensor measure). The **S** value is computed considering all those transitions caused by a wrong value of sensor measure with respect to use cases. In this case, sensor value can involve 5 transitions, where 3 could impact on the expected PN status in case of wrong value. Therefore value has been computed as 0.6. At the end, no detection mechanism was present and so the value **1-R** was 1 because probability of detection **R** was 0. The final value is $Risk = 0.6 * SeF$. Let's now consider the second section of the analysis:

Safety Mechanisms			Description	Final parameters			
detection	mitigation	Reference		P	S (Worst Case)	(1-R)	tot
n/a	Redundancy	Safety Manual	At least 3 sensors for single reading. Majority value is taken	(0,5*N)*SeF	0,6	1	0,6*P
Duplicate variable, operation and then checking	n/a	Safety Manual	duplicate operation on different variable involving different resources or algorithms and then match	x	0,6	<1	0,6*P*(1-R)

Figure 5.3: Risk evaluation on safety PNC

In figure 5.3 are considered the detection and the mitigation mechanism for Activation Sensor. The implemented method is the redundancy with majority voting. By definition this is a recovery mechanism that masks an error in a specific set of condition but it does not allow the detection. For this reason **(1-R)** remains the same value. What changes is the probability of occurrence. This because, the mitigation mechanism with redundancy with majority voting fails in the case, where majority of sensors are affected by error. This means if majority value is $0.5 * NumSensor + 1$, the failure is not mitigated when $0.5 * NumSensor + 1$ measure a wrong value. Therefore the probability of failure occurrence is SeF i.e. the probability of failure on a single sensor, times by itself until $0.5{*}NumSensor{+}1$ (the half number of sensors is approximate to lower integer value). In the case where sensors are 3 (minimum value to implement the redundancy, known also as **TMR**) and SeF is 0.5, the probability of failure occurrence is

$$\prod_{i=0}^{0.5*NumSensor+1} SeF_i = 0.25$$

and actually $0.25 < 0.50$. Therefore The risk index is reduced (note that this values are true if and only if the voter is not affected by errors). Let's note that this method works with an odd values of sensors. How it has been explained, this measures can contain mathematical traces about simile computation but they cannot be considered quantitative, especially at

software level. They main scope is to prove as much as possible an assessment on the risk reduction as in the previous case. In the case of set alarm unit, the operating flow is very similar: the severity value is computed at the same time of previous case i.e. considering the erroneous transitions that is caused by wrong data. This time the important parameter is the probability of detection **R**: let's note that the failure occurrence will depend from hardware parameters (permanent fault probability due to ageing or others) and it is not possible to operate on it. At the same time, the only possibility to mitigate it is to insert a configuration parameter that is able to choose among several "back-up safety values" if a computational module is faulty (actually it is not implemented). The basic operation that can be done is to duplicate variable and operation in order to match them and to verify if the module is working. This might require assumptions on which kind or hardware is used after compiling the source code and implement a code that might force to use, for the same operation, two different resource. Again it is not possible to provide a certain value for this case but writing a stub and performing unit testing is already reasonable to assume, for sure it will be less than 1.
To conclude, there was another possibility in consideration, for providing an evaluation parameter: one suggestion for unit testing is to test using fault-injection method i.e. to test a software unit with faulty input to check its response. For this reason might be considered a parameter for measuring the ratio between the test cases produced by faulty-injection that have failed with respect to the total test cases that have been produced, using faulty injection method. It is expected, implementing the safety mechanism this value will be lower (see the next formula)

$$FR = \frac{\#Failed\_TC\_FaultInj}{\#Total\_TC\_FaultInj}$$

5.2 Safety Manual

When ISO 26262 shall be applied for developing a SEooC, the Safety Manual is one of the most important work product. It is well-defined in IEC 61508 for programmable hardware and it is extended to ISO 26262. It is drafted during the entire design of an element and it can be seen as a global container for safety features and properties. For PNC the main scope have been identified:

- Contain identifier, class, status, title and author information;
- Contain a functional description of the element considering all hierarchical level where it is defined by Customer, tasks an operating modes;
- Contain a description of possible application of the element;
- Contain a report about the ISO 26262 chapters and clauses that are considered in scope.
- Replace the concept phase, reporting the assumptions on several levels and nature (target environment, properties, boundaries, etcetera...);
- Assume the Software Safety Requirements from higher-level (substitutes of TSR);
- List the assumption on Hardware-Software Interfacing;

- Assume safety-relevant software units that build the software component;
- Assume Safety Goal for software Component (if the element is a component);
- Assume SSR to satisfy Safety Goal;
- Identify Safety mechanisms for error detection and mitigation for safety-relevant units according to SSR;
- Report results of Safety Analysis or specify particular features on them;
- Implements Requirements Traceability;
- Contains Legal Information;

HSI builds an interesting role, especially in the development of a SEooC. This because for improving the safety part of a module development, external facilities might be necessary. A safety mechanism is developed considering the possibility that an Hardware Resource contains that module. This assumption shall be documented because such resource might not be available in the integrator's target and so, it must be removed after the validation of the assumptions (watchdog and MPU are examples).
The software unit plays a role for allocation of safety mechanisms and to control the possibility of failure propagation among them. Therefore safety analysis shall be carefully evaluated for this phase. The actual way that has been chosen for classifying the software units is shown in part in table 5.5

ID	Software Unit	Safety Relevant	Expected RTE Call	Called By [Id]	Description
1	SUM_PNC_Init	No	RTE for DTC Timer getter and DTC value setter	System	The initialization of the software component depends from system. It is assumed no error occurrence during initialization
2	Set_Alarm	Yes	N/A	6, 7, 8, 9	**(Private)** This functions set PNC private counter flags and it computes repeat message time counter initial value
3	Activation_Sensor	Yes	RTE to check Calibration parameter it determines if PN shall be processed or not and interface to read from Sensor	6, 7, 8, 10, 11	It evaluates activation criteria provided by a sensor software components as Local activation

Figure 5.4: Table for Software Unit evaluation in Safety Manual (ETAS Confidential)

How it is possible to observe, there are software unit that have been already discussed.
The last note aims to identify a possible way to implement internal and external traceability of the requirements. The fist one defined a section that contains internal document link for a faster relocation to the section where requirement is documented; the second one aims to define a link by identifier (unique) that can be used by external document to link to the specific section where a requirement is specified. In particular, this last one will be used by traceability that is implemented in source code as comments and it will be used for generation by Doxygen to link at specific work products. The situation will be very similar to the following figure:

Table 12: table for traceability for safety goal

Tag	Type	Current Document Reference	External Reference
A-SSR01	PNC Software Safety Requirements	A-SSR1	assr.7.1
A-SSR02	PNC Software Safety Requirements	A-SSR2	assr.7.2
A-SSR03	PNC Software Safety Requirements	A-SSR3	assr.7.3
A-SSR04	PNC Software Safety Requirements	A-SSR4	assr.7.4
A-SSR05	PNC Software Safety Requirements	A-SSR5	assr.7.5
A-SSR06	PNC Software Safety Requirements	A-SSR6	assr.7.6

Figure 5.5: Table for Software Unit evaluation in Safety Manual (ETAS Confidential)

5.3 Conclusion

The original scope of the current paper was to consider all aspects related to the safety design and implementation of a AUTOSAR Application Component, according to ISO 26262 ad MISRA C. The project specifications and the flexibility in safety life-cycle that is allowed by the element nature ahas required several analysis about how to identify the safety requirement in scope. The theoretical safety-relevant operations have required a trade-off that has forced to assume specific scenarios. These allowed the component to solve its important tasks (that will have an increasing impact in the next future, especially on electric motors) without excessive overloading or limitations due to safety mechanisms. For sure, the suggested solutions shall not be considered as totally exhaustive; actually they are in evaluation in ETAS as "starting point" for implementing a more robust code, strongly oriented to safety, for increasing the state-of-art of their product. To reach a very high safety level, for sure more solutions will have to be considered; several safety mechanisms will have to be analysed for dynamic aspects i.e. working on AUTOSAR basic software modules. Other approaches for static code development as model-based one might be considered too; this might increase the work, focusing developer attention on model design and definition. As last conclusion, the quality of the proposed safety mechanisms requires the complete verification phase of the V-model, performing integration testing and test of embedded software and adding the qualification of the assumptions by customer too.

Chapter 6

Additional

6.1 Acronyms and Abbreviations

- *2-C*: 2-Complement;
- *AAT*: AUTOSAR Authoring Tool: ETAS uses its own tool called ISOLAR;
- *AR*: AUTOSAR (AUTomotive Open-System ARchitecture)
- *A-SWC*: Application Software Component (AUTOSAR Application layer);
- *BSW*: Basic Software;
- *CFG*: Control Flow Graph;
- *DBMS*: Database Management Systems;
- *ECCA*: Enhanced Control-flow Checking by Assertion;
- *ECU*: Electronic Control Unit;
- *FSM*: Finite State Machine;
- *FSR*: Functional Safety Requirements;
- *LOC*: Lines of Code;
- *MISRA*: Motor Industry Software Reliability Association;
- *MPU*: Memory Protection Unit;
- *PN*: Partial Networking;
- *PNC*: Partial Networking Coordinator;
- *RE*: Requirement Engineering;
- *RTE*: Real Time Environment (AUTOSAR middle layer);
- *SEooC*: Safety Element out-of-Context;
- *SG*: Safety Goal;

- *SDD*: Software Design Document;
- *SSR*: Software Safety Requirements;
- *TSR*: Technical Safety Requirements;
- *WCET*: Worst Case Execution Time;

Bibliography

[1] Statistics about vehicle market for road vehicles: https://www.statista.com/topics/1487/automotive-industry/

[2] "This car runs code" by Robert N. Charette, 01/02/2009: https://spectrum.ieee.org/transportation/systems/this-car-runs-on-code

[3] "Effectiveness of Motorcycle Antilock Braking Systems (ABS) in Reducing Crashes, the First Cross-National Study" by Matteo Rizzi, Andres Kullgren, Claes Gustav Tingvall, June 2014

[4] "Model-Base Software Design: Introduction to Functional Safety and ISO 26262", Massimo Violante, Politecnico di Torino

[5] "Power Saving using Partial Networking in Automotive System" by Chae Hong Yi, Jae Wook Jeon, "International Conference on Information and Automation", August 2015

[6] CAN low-speed start connection image: https://www.researchgate.net/figure/Low-speed-or-fault-tolerant-CAN-10_fig2_322161084

[7] data sheet of complex CAN transceiver TJA1145: https://www.nxp.com/docs/en/data-sheet/TJA1145.pdf

[8] ISO 9001: Quality management system: https://www.iso.org/iso-9001-quality-management.html

[9] "Error Handling", Massimo Violante, Politecnico di Torino

[10] "Fault Tolerant Design", Matteo Sonza Reorda, Massimo Violante, Politecnico di Torino

[11] Bidirectional traceability definition and methods: https://web.archive.org/web/20160305213051/http://www.compaid.com/caiinternet/ezine/westfall-bidirectional.pdf

[12] AUTOSAR: Specification of Compiler Abstraction: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_SWS_CompilerAbstraction.pdf

[13] QA-Systems Cantata Official site: https://www.qa-systems.it/strumenti/cantata/

[14] "A Complexity Measure" by Thomas J. McCabe

[15] "Overview of functional safety Measure" (AUTOSAR specification)